The Journal of Systems and Software 222 (2025) 112307

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# COMET: Generating commit messages using delta graph context representation[☆,☆☆]

Abhinav Reddy Mandli[1], Saurabhsingh Rajput[*,1], Tushar Sharma

Dalhousie University, Canada

## ARTICLE INFO

Dataset link: Zenodo

*Keywords:*
Commit message generation
Large language models
Code property graphs
Graph neural networks
Convolutional neural networks
Code change representation

## ABSTRACT

Commit messages explain code changes in a commit and facilitate collaboration among developers. Several commit message generation approaches have been proposed; however, they exhibit limited success in capturing the context of code changes. We propose **Comet (Context-Aware Commit Message Generation)**, a novel approach that captures context of code changes using a graph-based representation and leverages a transformer-based model to generate high-quality commit messages. Our proposed method utilizes *delta graph* that we developed to effectively represent code differences. We also introduce a customizable quality assurance module to identify optimal messages, mitigating subjectivity in commit messages. Experiments show that Comet outperforms state-of-the-art techniques in terms of BLEU-norm and METEOR metrics while being comparable in terms of ROUGE-L. Additionally, we compare the proposed approach with the popular `gpt-3.5-turbo` model, along with `gpt-4`—the most capable GPT model, over zero-shot, one-shot, and multi-shot settings. We found Comet outperforming the GPT models, on five and four metrics respectively and provide competitive results with the two other metrics. The study has implications for researchers, tool developers, and software developers. Software developers may utilize Comet to generate context-aware commit messages. Researchers and tool developers can apply the proposed *delta graph* technique in similar contexts, like code review summarization.

## 1. Introduction

Commit messages are essential in documenting changes carried out in each commit to maintain and understand code evolution (Hindle et al., 2009; Barnett et al., 2016; Rebai et al., 2020). Writing good commit messages promotes effective communication and seamless collaboration within a software development team (Tao et al., 2012; Huang et al., 2014). Meaningful commit messages ease the burden of many tasks, including debugging and software lifecycle management (Yan et al., 2016; Hassan, 2008). A good quality commit message is succinct yet clearly articulated and informative to convey the nature and purpose of the change (Agrawal et al., 2015; Cortés-Coy et al., 2014), *i.e.,* the *why* aspect, as well as the summary of the changes made to the code *i.e.,* covering the *what* aspect (Tian et al., 2022), along with the context of the code change.

*Context* plays an important role in commit message generation because the effectiveness of the generated message depends on the surrounding context in addition to code changes (Sillito et al., 2008). The context refers to the surrounding code in which the change occurs; it provides vital information to understand the rationale behind a change. As shown in Listing 1, looking at the diff of changed lines suggests migrating HTTP to HTTPS connection. However, the surrounding unchanged code reveals the URL is used to retrieve user data on profile pages. This context helps generate an informative message that the purpose is to *retrieve user data securely*.

However, developers often do not provide relevant, good-quality commit messages that align with the commit context. According to a study on five popular open-source projects on GitHub (Tian et al., 2022), approximately 44% of commit messages lack both *what* and *why* aspects. Automated commit message generation approaches attempt to reduce the burden on developers by generating effective commit messages taking into account the performed changes within a context.

☆ Editor: Gabriele Bavota.

☆☆ **Funding**: The authors are partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through grant NSERC Discovery RGPIN/04903.

* Corresponding author.
*E-mail address:* saurabh@dal.ca (S. Rajput).

1 These two authors contributed equally to this work.

https://doi.org/10.1016/j.jss.2024.112307
Received 25 January 2024; Received in revised form 12 October 2024; Accepted 2 December 2024
Available online 16 December 2024

```
// User profile page
public void viewProfile(User user) {

// Retrieve user data

- String url = "http://example.com/data";
- HttpURLConnection connection = (HttpURLConnection) url.openConnection();

+ SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
+ sslContext.init(null, null, null);
+ HttpsURLConnection connection = (HttpsURLConnection) url.openConnection();
+ connection.setSSLSocketFactory(sslContext.getSocketFactory());

// Render profile view
renderPage(user);
}

// Commit message without context: Migrate HTTP connection to HTTPS
// Commit message with context: Use HTTPS to securely retrieve user data on profile pages
```

**Listing 1** Commit message generated with and without code context.

Over the years, several approaches have been proposed to generate commit messages automatically. The initial approaches, such as *ChangeScribe* (Linares-Vásquez et al., 2015) used a code summarization approach to generate commit messages. However, such rule-based approaches (Buse and Weimer, 2010; Moreno et al., 2017) are limited in their ability to capture the context and subjectivity of code changes. Subsequently, researchers proposed a Neural Machine Translation (nmt)-based method to generate commit messages from *code diffs* (Jiang et al., 2017). However, these approaches do not fully leverage the syntactic and structural properties of code, and fail to capture the complete context of source code changes. Similarly, techniques based on information retrieval (Liu et al., 2018; Huang et al., 2020; Wang et al., 2021b) rely on a reference dataset during inference and are prone to the limitations of information retrieval models, such as the out-of-vocabulary problem (Xu et al.; Dong et al., 2022; Liu et al., 2020).

Recently, transformer-based large language models (LLMs) (Feng et al., 2020; Wang et al., 2021a) have demonstrated impressive capabilities in various code-related tasks, such as generating natural language text from code, including code comments and commit messages (Jung, 2021; Shi et al., 2022; Sharma et al., 2024). These models have shown to be effective at capturing complex relationships and patterns in source code by utilizing attention mechanism. However, the majority of the past approaches that utilize these models do not consider the structural and semantic information of code changes, as they treat code as a linear sequence of tokens. In addition, their capability to capture the code context in which the changes have been made is yet to be proven.

In a ML-based method, the dataset quality significantly influences the training process and the generated outcome. Previous studies have used various filtering techniques for their datasets based on factors such as message length, code diff length, verb direct object dependency, and number of classes (Liu et al., 2020; Jiang et al., 2017). However, the *quality of the commit message* itself has not been considered as a metric for filtering in most of the past approaches. The poor quality of commit messages that are used in these studies as training samples can significantly impact the performance of these models, leading to substandard results. This can be attributed to the "Garbage in, garbage out" principle (Sanders and Saxe, 2017), where subpar input data leads to inferior output. Therefore, it is imperative to use high-quality commit messages in the training phase to improve the performance of the used machine learning (ML) models.

Previous approaches for commit message generation have utilized various data structures to represent code changes. Early techniques, such as NMT-based approaches (Jiang et al., 2017; Loyola et al., 2017; van Hal et al., 2019), treat code as a flat sequence of tokens, ignoring the syntactic and semantic information of the code changes. Some approaches, such as *CoDiSum* (Xu et al.), attempt to capture code structure and semantics but represent code as a linear sequence of tokens.

More recently, researchers have proposed using graph-based representations to capture the structural and semantic information of code changes for commit-related tasks. For example, FIRA (Dong et al., 2022) uses fine-grained graphs to represent code changes, while ATOM (Liu et al., 2020) incorporates abstract syntax trees (ASTS) to represent code changes. Additionally, RACE (Shi et al., 2022) employs a hybrid generation and retrieval module to encode commits, while NNGen (Liu et al., 2018) utilizes an information-retrieval-based model. When it comes to commit message classification, there have been studies such as Zhang et al. (2023) that use system dependency graphs and slicing techniques to represent code diffs for commit message classification, and Meng et al. (2021) that construct change dependency graphs (CDGs) based on ASTS. However, the approaches utilized for commit message generation have limitations in capturing the complete context surrounding the code changes, which is crucial for generating informative commit messages.

The format of commit messages may vary depending on the context, purpose, and organizational needs. In practice, varying conventions and templates (The Conventional Commits Specification, 2023; qoomon, 2023) exist to guide developers in structuring their commit messages, such as specifying change types including "fix", "feat", or "chore", at the beginning of a commit message. These conventions aid developers in ensuring a standard is followed across their organization. It implies that automated commit message generation approaches need to be flexible to accommodate organization-specific message types while still being generalizable enough to cover base scenarios. The current approaches do not provide this flexibility.

We propose **COMET** (Context-aware Commit Message Generation) a commit message generation technique that aims to overcome the above-mentioned limitations of past research. Specifically, to improve the quality of the training dataset, we filter out substandard and poor-quality messages. We propose a new mechanism *delta graph*, a graph-based representation of code changes, to capture the context of the source code effectively by generating a union of code, before and after the code changes in each commit. Our commit message generation module leverages state-of-the-art Encoder-Decoder models fine-tuned on *delta graph* results to generate high-quality commit messages. Finally, we introduce a *quality assurance* module based on Graph Neural Network (GNN) to ensure that our approach selects the best possible message among the generated options based on the format required by the organization. We evaluate our approach using a variety of performance metrics used by similar studies in the field, including BLEU, ROUGE, and METEOR. Specifically, we employ BLEU-norm (Loyola et al., 2018), a variant of BLEU due to its strong correlation with human judgment (Tao et al., 2021). The results of our evaluation indicate that our approach significantly outperforms the state-of-the-art technique in terms of these metrics. Specifically, we achieved a BLEU score of 7.38, a ROUGE score of 13.45, and a METEOR score of 12.26, which are substantially higher than the state-of-the-art approach. This represents an increase of 17.7% for BLEU, and 3% for METEOR compared to the state-of-the-art results, while it remains comparable in terms of ROUGE-L. We also evaluated our approach against GPT models (`gpt-3.5-turbo` and `gpt-4`). We found our approach outperformed these models on five and four metrics, respectively, and provided competitive results with the rest of the metrics. These results demonstrate the effectiveness of our representation of code changes in generating high-quality commit messages and show promise in applying *delta graph* to similar software engineering problems.

Our paper makes the following contributions to the field.

- The study proposes ***delta graph***—a novel graph-based representation of code changes that incorporates the context along with code modifications. Unlike previous approaches that relied on raw text diffs or abstract syntax trees to represent code changes, *delta graph* is the first representation specifically designed to capture both the changed code and the surrounding context for the task of commit message generation. Researchers in the field may use and extend *delta graph* for other software engineering problems as well where changes in the code with context are required.
- The study provides a **filtered dataset** of high-quality commit messages along with their corresponding *delta graph* and code property graph representations. This dataset is curated from an existing dataset by filtering out low-quality, noisy, and irrelevant commit messages, ensuring that such a dataset contains only informative and meaningful samples. The inclusion of *delta graph* and code property graph representations of the curated dataset enables researchers to leverage the rich structural and contextual information of code changes for commit message generation and other related tasks (COMET, 2023).
- The proposed approach **outperforms the state-of-the-art techniques**. Our experimental results demonstrate that our method achieves significant improvements over the state of the art, with a 17.7% increase in BLEU score and a 3% increase in METEOR score while maintaining comparable performance in terms of ROUGE-L. In a qualitative assessment, COMET found to generate more accurate, contextually relevant, and informative commit messages compared to the state-of-the-art methods.
- The study offers a **customizable quality assurance module** that enables us to choose the best message among the available options, where *best* is defined by the employed customizable strategy. This module provides flexibility to accommodate organization-specific commit message conventions and templates, a feature lacking in current commit message generation techniques.

**Replication package:** We have made our tool and scripts publicly available (Smart lab Dalhousie university, 2023) to facilitate other researchers to replicate, reproduce, and extend our study. We have also provided the filtered dataset of high-quality commit messages online (COMET, 2023).

## 2. Background

In this section, we briefly discuss relevant concepts and techniques used in the presented work.

### 2.1. Version control systems and commit messages

A Version Control System (VCS) (Spinellis, 2005) is responsible for recording and managing the changes that occur in digital artifacts such as computer programs and documents. Prominent examples of VCSs are Git, TFS, and SVN. Typically, a VCS creates a *repository* that represents a database to store metadata of the changes in each version. A developer creates a *commit* to record the changes in the repository; a *commit message* is the description and rationale of the recorded changes in a commit in natural language text.

### 2.2. Deep learning models

Deep Learning (DL) models are a type of machine learning algorithms that use artificial neural networks with multiple layers to learn hierarchical representations of data for variety of tasks including classification, summarize, or synthesize. In this work, we focus on transformer-based models, which have shown remarkable performance in various natural language processing tasks.

#### 2.2.1. Transformer-based models

Transformers (Vaswani et al., 2017) are a type of neural network architecture that has become increasingly popular due to their ability to process input sequences in parallel and handle long-range dependencies more effectively than traditional recurrent neural networks. The transformer architecture comprises an encoder and a decoder, each consisting of multiple layers of self-attention and feedforward neural networks. The attention mechanism allows the model to selectively focus on parts of the input sequence that are most relevant to the current task, making transformers effective in processing both sequential and non-sequential data, such as graphs.

#### 2.2.2. Graph neural networks

Graph Neural Networks (GNNS) (Scarselli et al., 2009) operate on graph-structured data and capture complex relationships between entities effectively. GNNS learn node and edge representations by aggregating information from their local neighborhoods, enabling them to model complex relationships among nodes in the graph. GNNS have demonstrated promising results in various graph-related tasks, including node classification (Bhagat et al., 2011), link prediction (Liben-Nowell and Kleinberg, 2003), and graph classification (Errica et al., 2019).

### 2.3. Program analysis representations

In this section, we provide an overview of various program analysis representations referred in our work.

**Abstract Syntax Tree (AST)**: An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code. Each node in the AST represents a construct occurring in the source code, such as a variable declaration, a function call, or a control flow statement.

**Control Flow Graph (CFG)**: A Control Flow Graph (CFG) is a directed graph that represents the control flow of a program. Each node in the CFG represents a basic block, which is a sequence of instructions with a single entry point and a single exit point. Edges in the CFG represent the flow of control between basic blocks.

**Program Dependence Graph (PDG)**: A Program Dependence Graph (PDG) is a directed graph that represents both control and data dependencies in a program. Nodes in the PDG represent statements or expressions, while edges represent control or data dependencies between nodes.

**Code Property Graph (CPG)**: A Code Property Graph (CPG) (Yamaguchi et al., 2014) merges classic program analysis concepts ASTs, CFGs, and PDGs into a single, graph-based representation of code along with its context. CPGs capture essential code properties such as syntax, semantics, data flow, and control flow, making them an effective tool for analyzing and understanding code.

### 2.4. Evaluation metrics

We use several evaluation metrics to assess the quality of the generated commit messages:

**BLEU**: BLEU (Bilingual Evaluation Understudy) (Papineni et al., 2002) is a metric for evaluating the quality of machine-generated text by comparing it with human-generated reference text. BLEU calculates the n-gram precision between the generated and reference texts and applies a brevity penalty to penalize short-generated texts.

**ROUGE**: ROUGE (Recall-Oriented Understudy for Gisting Evaluation) (Lin, 2004) is a set of metrics for evaluating the quality of summarization systems. ROUGE measures the overlap of n-grams between the generated summary and a set of reference summaries, focusing on recall rather than precision.

**METEOR**: METEOR (Metric for Evaluation of Translation with Explicit ORdering) (Banerjee and Lavie, 2005) is a metric for evaluating machine translation quality that considers both precision and recall. METEOR aligns the generated and reference texts based on exact, stem, synonym, and paraphrase matches and then computes a weighted harmonic mean of precision and recall.

## 3. Methods

This section elaborates on the methods and mechanisms to implement our proposed approach.

### 3.1. Research questions

The *goal* of the research is to implement a commit message generation approach considering the context of code changes in a commit to produce a high-quality commit message. We derive the following research questions based on the goal of the presented study.

**RQ1. Does capturing the code context improve the accuracy of the commit message generation model?**

A code change is not an isolated transformation and therefore, changes must be understood considering the surrounding context. The proposed research question aims to determine whether capturing the code context improves the accuracy of the commit message generation model. Answering this research question will provide insights into the role of the context while generating a commit message.

**RQ2. How does the proposed approach perform compared to existing state-of-the-art automated commit message generation approaches?**

Through this research question, we compare our approach with existing commit message generation methods. Furthermore, we qualitatively evaluate our approach, comparing it against the best performing baseline. Exploring this question will show whether and to what extent the proposed technique improve the state of the art.

**RQ3. How does the proposed approach compare against the latest GPT models?**

`GPT-3.5` and `GPT-4` models, given an appropriate prompt, generate natural language text or code. The `gpt-3.5-turbo` is the most popular model, while `gpt-4-1106-preview` is the most capable model from OpenAI at the time of writing this text. They are optimized for chat applications; the models currently power ChatGPT (Huggingface, 2022), a popular chatbot used in various abstractive and extractive tasks belonging to diverse domains (Dwivedi et al., 2023). The research question aims to evaluate the performance of the proposed approach against these models to provide insights into the comparison between learning from a large amount of generic data and learning from a small amount of relevant data.

Fig. 1 presents an overview of the proposed approach. The approach uses a pre-processing module that applies a set of filters, to filter out low-quality commit messages from the training dataset. Also, we convert code from the filtered dataset into *delta graph* representation using our *delta graph* module. This representation is then passed through a transformer-based DL module that learns to generate commit messages given code modifications and its context in the *delta graph* format. The generator module generates multiple commit messages. We employ a *Quality Assurance* (QA) module that ranks messages based on user-defined criteria to ensure emitting a high-quality message. Such a mechanism results in a versatile approach that prioritizes relevant and informative messages as per users' needs. We elaborate on the individual component of our approach in the subsequent section.

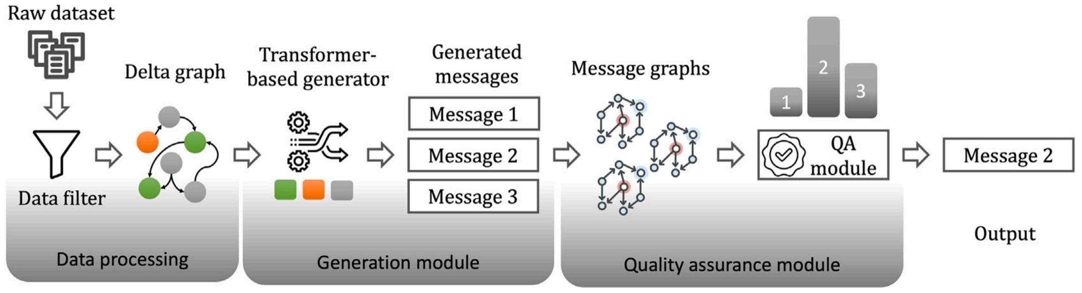

**Fig. 1.** Overview of the approach.

## 3.2. Data pre-processing

We train and evaluate COMET on the MCMD dataset proposed by Tao et al. (2021). We choose this dataset because first, it has been used in other similar studies (Shi et al., 2022; Dey et al., 2022). Also, the dataset provides corresponding complete code in addition to code diff that is crucial for generating high-quality commit messages. Also, the dataset offers the ability to trace back each code diff to the original repository, enhancing control over the dataset. The dataset originates from the top 100 starred projects on GitHub, representing a diverse range of code repositories in six different programming languages.

We limit the scope of our study to the Java programming language. Though we train and evaluate our model on a Java dataset, the only code-dependent element in our approach involves parsing the code into CPG. Consequently, our approach can adapt to any programming language, provided a language-dependent tool or library to transform source code into CPG.

After obtaining the dataset consisting of 450,000 commits, we filter out files with extensions other than `.java`.

Similar to previous approaches in this domain Jiang et al. (2017), we truncate the commit messages to the first line as it is considered the summary of the whole message. Researchers have found that the initial sentence in commit messages (Jiang et al., 2017) and API comments (Gu et al., 2016; Anon, 2019) usually encapsulates the key information and context.

However, data quality plays a significant role in training deep learning models. Tian et al. (2022) highlight the critical role of commit messages as an "audit trail" in collaborative software development. Their study reveals that approximately 44% of commit messages could be improved, indicating a significant proportion of sub-optimal messages in the raw, uncurated datasets. They also investigate the possibility of automatically segregating "good" commit messages by identifying "why" and "what" aspects of the commit messages that we use in our study.

They used Bi-LSTM layers trained on a labeled dataset of relevant and irrelevant commits. During the filtering process, the trained model takes the commit message and corresponding code changes as input and outputs a relevance score. Commits with scores below a certain threshold are considered irrelevant and filtered out.

For example, the filter implemented by Tian et al. (2022) filters out commits corresponding to the following commit messages.

1. *Commit message:* "refactoring"
   Code changes include renaming variables, extracting methods, and moving code between files across the entire codebase.
2. *Commit message:* "fix bug"
   Code changes include multiple edits to several files to handle a complex edge case.
3. *Commit message:* "update dependencies"
   Changes include bumping the version numbers of various third-party libraries in the project's configuration files.

Examples of commits that are retained by the filter are presented below.

1. *Commit message:* "Add support for custom user avatars"
   Code changes include implementation of a new feature allowing users to upload and display custom avatar images, including both backend and frontend changes.
2. *Commit message:* "Fix issue with login form not handling special characters"
   Changes include modifying the login form validation logic to properly handle usernames and passwords containing special characters.
3. *Commit message:* "Refactor database connection module for improved performance"
   Changes include restructuring the database connection code to use a connection pool, adding retry logic, and optimizing frequently used queries.

The retained commit messages provide a good level of details about the changed source code elements and corresponding rationale, making them well-suited for our model's training and evaluation. Finally, we obtain 18,214 commit messages that meet the filter criteria for model training. We obtain code property graph (CPG) of each sample in the filtered dataset using `Joern` (Joern Documentation Blog RSS, 2023). We provide the full list of filtered and retained commit messages in our replication package (COMET, 2023) to support reproducibility. It is important to note that all baseline techniques used in the study were trained on filtered data, ensuring fairness for all considered techniques.

To mitigate the potential threat of relying solely on automatic filtering, we conduct a manual validation of the filtered samples. We randomly sample a subset of 377 filtered commit messages, which provides a confidence level of 95% with a margin of error of 5% for our population of 18,214 filtered commit messages. We manually reviewed each sampled commit message to assess the accuracy of the automatic filtering process. One of the authors conducted the review. Commit messages were classified as either correctly filtered (relevant) or incorrectly filtered (irrelevant). The manual review revealed that 86.5% of the sampled commit messages were correctly filtered as relevant, while only 13.5% were irrelevant commit messages that were not filtered out. To establish the statistical significance of these results, we conducted a one-sample proportion test with a null hypothesis that the proportion of correctly filtered commit messages is equal to or less than 95%. The test resulted in a $p$-value of $1.6e-10$,

**Table 1**
Project types and domains of the filtered samples.

| Domain | Count | Examples |
|---|---|---|
| Web Frameworks and Libraries | 12 | Spring Framework, Vert.x, Jenkins |
| Networking and Communication | 8 | Netty, OkHttp, Retrofit |
| Database and Data Management | 7 | Elasticsearch, MyBatis, Druid |
| Android Development | 10 | ExoPlayer, Glide, Lottie |
| Distributed Systems | 6 | Apache Kafka, Apache Flink, Dubbo |
| Build Tools and CI/CD | 2 | Bazel, Jenkins |
| Utility Libraries | 8 | Guava, Gson, RxJava |
| Performance Monitoring | 4 | Skywalking, Zipkin, Cat |
| Security and Cryptography | 2 | Signal-Android, Sentinel |
| Game Development | 1 | libGDX |
| Code Analysis and Reverse Engineering | 3 | Jadx, Ghidra, Bytecode Viewer |
| Machine Learning | 1 | Deeplearning4j |
| Others (UI, Testing, etc.) | 12 | Material Components, Selenium, NewPipe |

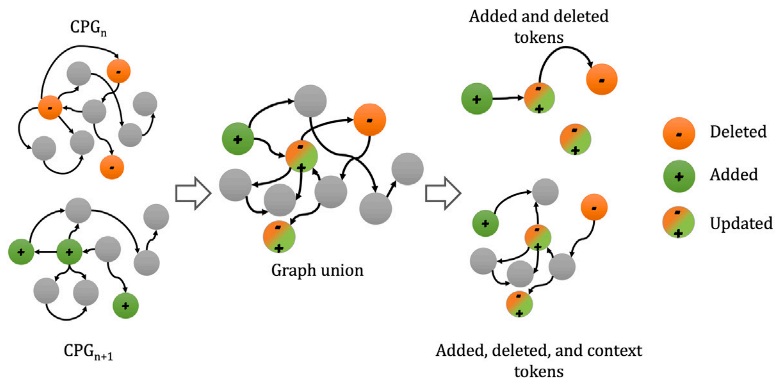

**Fig. 2.** Overview of *delta graph* creation.

allowing us to reject the null hypothesis and conclude with high confidence that the actual proportion of correctly filtered commit messages is significantly greater than 95%. These manual validation results indicate that the filtering process is reasonably reliable.

The 18,214 commits are not evenly distributed across the 100 Java projects. The number of commits per project ranges from 1 to 1530, with an average of 169 commits per project. Filtering the original MCMD dataset for projects containing Java code resulted in 76 remaining projects, forming our final dataset. These projects span a wide range of domains within the Java ecosystem, as detailed in Table 1. The diversity of project types ensures that our dataset covers various aspects of Java development, from low-level networking libraries to high-level application frameworks and tools. Regarding the train/validation/test split, we utilize the predefined MCMD splits, which divide the projects into train, validation, and test sets. This split design ensures a fair train/test separation at the project level and avoids potential data leakage. The train set contains 16,693 commits, while the test set consists of 1521 commits.

### 3.3. Delta graph

An abstract syntax tree (AST) provides a detailed breakdown of source code into language constructs (Yamaguchi et al., 2014) giving a structure-rich representation of the code. Similarly, a control flow graph (CFG) exhibits the execution flow of the program (Yamaguchi et al., 2014). Furthermore, a program dependence graph (PDG) models data and control dependencies of a program. Previous works on commit message generation remained limited to the use of AST and do not explore alternative representation methods. Although ASTs have long been an efficient code modeling approach, other information such as data and control dependencies could be leveraged to enrich the representation. An information-rich code representation in this context is Code Property Graph (CPG) that combines AST, CFG, and PDG. This combined representation has been effective in various software engineering tasks such as vulnerability analysis (Zhou et al., 2019; Xiaomeng et al., 2018) and code refactoring suggestions (Cui et al., 2023)

A *delta graph* models changes through structural (*i.e.,* AST) and semantic (*i.e.,* CFG and PDG) aspects into a single graph representation to maximize the information gained from the changes made. In the *delta graph* representation, each node denotes a code token from AST nodes (such as function declarations, statements, and expressions), CFG nodes (such as basic blocks representing a sequence of statements), or PDG nodes (statements or expressions involved in data or control dependencies). The edges in the *delta graph* represent the relationships between these code tokens, such as parent–child relationships in AST, control flow transitions in CFG, or data and control dependencies in PDG. We utilize *Joern* (Joern Documentation Blog RSS, 2023), a widely used source-code analysis platform to generate CPGs for code snippets. We use the generated CPGs, manipulate them, and construct corresponding *delta graphs* as shown in Fig. 2.

We use a set difference approach based on the node identifiers present in the two consecutive CPGs to identify the added and deleted nodes. We first extract the set of node identifiers from each CPG and then compute the set difference between these identifier sets. The nodes corresponding to the identifiers present in the set difference results are considered as added or deleted nodes, respectively. This approach efficiently identifies the changes in the node sets between the two versions of the code snippet.

In order to create a *delta graph* that represents the code changes between successive commits, we transform each version of the program into a CPG. This results in two CPGs denoted as $G_{p1} = (E_{p1}, V_{p1})$ and $G_{p2} = (E_{p2}, V_{p2})$, where $G_{p1}$ and $G_{p2}$ correspond to $CPG_n$ and $CPG_{n+1}$ respectively as shown in Fig. 2. Subsequently, we follow the steps given below to build three sub-graphs, aggregate and manipulate them to obtain a *delta graph* representation($\Delta G$):

- The common edges and vertices in $G_{p1}\&G_{p2}$ are extracted to represent a common graph $G_c = (E_c, V_c)$. This sub-graph contains the parts of the code that remain *unchanged*.
- The edges and vertices that are present in $G_{p1}$ but not in $G_{p2}$ are combined to form $G_{del} = (E_{del}, V_{del})$. This sub-graph contains the *deleted* parts of code.
- The edges and vertices that are present in $G_{p2}$ but not in $G_{p1}$ are considered as *added* tokens. The extraction of these properties results in $G_{add} = (V_{add}, E_{add})$. This sub-graph contains code tokens added or, in some cases, updated in the code.

These three sub-graphs are then aggregated to form a graph union: $G_U = G_{add} \cup G_{del} \cup G_c$. Graph union is a complete representation of the code change since it considers all possible tokens involved in the code change. The core contributors to a commit message are the uncommon tokens between two consecutive commits. Retaining all tokens leads to redundant information that not only makes the learning expensive but also possibly hurts the performance of the model. In order to tackle this issue and to extract the relevant contextual tokens that are closely related to the added and deleted tokens, we further process the penultimate representation to only retain the following information:

- Added edges and vertices : $G_{add}$
- Deleted edges and vertices : $G_{del}$
- Direct edges of added ($E_{add}$) / deleted ($E_{del}$) edges : $G_{[c]}$, where $G_{[c]} \subseteq G_c$

The decision to restrict the context to direct edges is inspired by the notion of over-smoothing (Li et al., 2018) in graph neural networks. The objective is to pass the most relevant features as context and as we move away from the modified tokens, the significance diminishes, providing a rationale for this approach. Moreover, considering additional edges elongates the context length, thereby posing a risk of performance degradation in LLMS (Liu et al., 2023a). The retained representations are united to form a *delta graph* representation: $\Delta G = (\Delta E, \Delta V)$, where $\Delta E = E_{add} \cup E_{del} \cup E_{[c]}$ and $\Delta V = V_{add} \cup V_{del} \cup V_{[c]}$.

### *3.4. Generation module*

We leverage transformer (Vaswani et al., 2017) models for their powerful sequence generation ability to achieve the goals of the study. The input to transformers is usually a sequence of tokens that are mapped to their corresponding embeddings. We aggregate these embeddings with positional embeddings, forming an initial vector representation. We pass the initial vector representations through a self-attention layer and a feed-forward neural network. The encoder's final representation is then passed onto the decoder through cross-attention layers. The decoder follows a similar architecture as the encoder except for an additional layer with cross-attention input from the encoder. The decoder outputs the sequence regressively using masked-self attention, ensuring it can only attend to the past tokens to generate a new word in the sequence.

Pre-trained transformers have shown huge success across various fields due to their dominant capability in zero-shot or few-shot generalization (Zhang et al., 2022). Pre-trained transformer models have proven effective in various code-related tasks such as code summarization, code comment generation, and others (Sharma et al., 2024). Several studies on commit message generation (Shi et al., 2022; Dong et al., 2022; Liu et al., 2020) leveraged encoder–decoder-based architecture to generate commit messages and have achieved promising results proving to be effective in this domain.

Commit message generation is a *code-to-text* task that requires a strong understanding of both code and text. Transformer-based encoder–decoder models such as `CodeT5` (770M) (Wang et al., 2021a), and auto-regressive models in recent times such as `GPT-3` (Brown et al., 2020) and `BLOOM` (Scao et al., 2022), have demonstrated their ability in sequence-to-sequence generation tasks in both natural language and code. Fine-tuning is a common approach used to leverage the transformer's domain knowledge. Pre-trained models require minimal data to adapt to a given task and can thus significantly reduce the time and resources required to train a model specific to a downstream task.

COMET fine-tunes a pre-trained transformer on *delta graph* representation. As the transformer's encoder or decoder accepts a linear sequence of tokens as input, the *delta graph* representation is linearized in a sequence of added, deleted, and common tokens and passed into the transformer. The *delta graph* captures structural and semantic aspects of code changes, including added, deleted, and context tokens, which helps retain important information about the code. By leveraging the *delta graph* representation, the model effectively capture and utilize the structural information of the code changes.

Consider the following example illustrating the input format and tokenization process.

**Input format before tokenization**: "Generate commit message: <added_code_tokens><sep_token><deleted_code_tokens><sep_token><common_code_tokens>"

**Input (before tokenization)**: "Generate commit message: public HotSpotResolvedObjectType getDeclaringClass()LoweringTool tool </s> LoweringTool tool(HotSpotResolvedObjectType) method.getDeclaringClass()public ResolvedJavaType getDeclaringClass()</s> @OverrideRETmodifiersunsafeholderconfig toolgetDeclaringClass()this.codethis.compilerStoragethis.holderreturn holder; HotSpotResolvedObjectType"

Where:

- <added_code_tokens> represents the code tokens that were added in the commit, e.g., `public HotSpotResolvedObjectType getDeclaringClass()LoweringTool tool`
- <deleted_code_tokens> represents the code tokens that were deleted in the commit, e.g., `LoweringTool tool (HotSpotResolvedObjectType) method. getDeclaringClass()public ResolvedJavaType getDeclaringClass()`
- <common_code_tokens> represents the code tokens that remained unchanged, e.g., `@OverrideRETmodifiersunsafeholderconfig toolgetDeclaringClass() this.codethis.compilerStoragethis.holderreturn holder;HotSpotResolvedObjectType`
- <sep_token> is a special token `</s>` used to separate the different parts

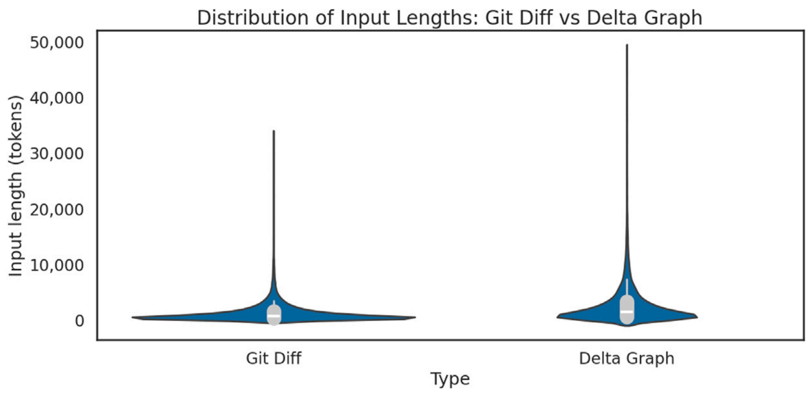

**Fig. 3.** Distribution of combined input lengths after dropping outliers (i.e., samples with length more than 50,000 tokens).

**After tokenization (tokens)**: ['<s>', 'Generate', 'Ġcommit', 'Ġmessage', ':', 'Ġpublic', 'ĠH', 'ot', 'Spot', 'Resolved', 'ObjectType', 'ĠgetDeclaringClass', '()', 'Lower', 'ing', 'Tool', 'Ġtool', '</s>', 'Lower', 'ing', 'Tool', 'Ġtool', '(', 'Hot', 'Spot', 'Resolved', 'ObjectType', ')', 'Ġmethod', '.', 'get', 'Decl', 'aringClass', '()', 'public', 'ĠResolved', 'JavaType', 'ĠgetDeclaringClass', '()', '</s>', '@', 'Override', 'RET', 'modifiers', 'un', 'safe', 'holder', 'config', 'tool', 'get', 'Decl', 'aringClass', '()', 'this', '.', 'cod', 'eth', 'is', '.', 'compiler', 'St', 'or', 'ag', 'eth', 'is', '.', 'holder', 'return', 'Ġholder', ';', 'Hot', 'Spot', 'Resolved', 'ObjectType', '</s>']

**After tokenization (input_ids)**: [1, 4625, 3294, 883, 30, 1071, 670, 352, 17292, 12793, 17610, 18401, 1435, 4070, 310, 6364, 5226, 2, 4070, 310, 6364, 5226, 12, 25270, 17292, 12793, 17610, 13, 707, 18, 588, 3456, 14682, 1435, 482, 22776, 31819, 18401, 1435, 2, 36, 6618, 10238, 15432, 318, 4626, 4505, 1425, 6738, 588, 3456, 14682, 1435, 2211, 18, 1559, 546, 291, 18, 9576, 510, 280, 346, 546, 291, 18, 4505, 2463, 10438, 31, 25270, 17292, 12793, 17610, 2]

**Output (commit message)**: "Refactor getDeclaringClass() method in HotSpotResolvedObjectType"

As shown above, the textual input to the model consists of the concatenated sequence of added, deleted, and common code tokens, separated by a special token and prefixed with a prompt. This input sequence undergoes tokenization prior to its ingestion by the model.

In our approach, the chosen pre-trained models have a maximum input sequence length, ranging from 512 to 2048 tokens depending on the specific model. When the aggregate length of the code diff and the surrounding context exceeds the maximum sequence length, we prioritize the code diff and truncate the surrounding context from the end until the input fits within the maximum sequence length. While this truncation approach allows us to process commits with extensive changes, it does come with a limitation. In cases where the code diff itself is extremely long and exceeds the maximum sequence length, potentially resulting in loss of information.

Fig. 3 shows the distribution of combined input lengths in our dataset. The figure shows a sparse distribution of input lengths where the majority of the inputs fall well within the maximum sequence lengths of the chosen models. The median input length is 247 tokens, indicating that half of the samples have an input length below this value. Furthermore, given that 75th percentile is at 543 tokens, the study is able to handle the majority of the samples in the dataset. The distribution has a few outliers, with the maximum input length of 220,464 tokens. The figure also illustrates that *delta graph* representations tend to have longer input on average compared to Git diffs. The mean length for *delta graphs* is about 102% larger than for Git diffs. This increased size in *delta graph* representations can be attributed to the additional contextual information included in the graph structure. While Git diffs primarily focus on the changed lines of code, *delta graph* captures the broader context of the changes, including relationships between code elements and surrounding code structures.

There are several pre-trained transformers trained on code that are publicly available for use such as `BLOOM` (Scao et al., 2022), `CodeT5` (Wang et al., 2021a), `GraphCodeBERT` (Guo et al., 2021), and `CodeBERT` (Feng et al., 2020). We select a set of pre-trained transformer-based models to experiment with, where either the encoder, or the decoder, or both are trained on a combination of code and natural language. A combination of `GraphCodeBERT` (Guo et al., 2021) & `GPT-2` (Radford et al., 2019), `Code-T5` (Wang et al., 2021a) and `BLOOM` (Scao et al., 2022) are the models satisfying the above criterion. The selection exercise is important as we intend to evaluate the selected pre-trained models to identify the best-performing model for Comet. We evaluate the selected models against three performance metrics: bleu, rouge-l, and meteor.

### *3.5. Quality assurance module*

Form and format of commit messages often depend on the organization and team culture. Comet proposes a Quality Assurance (QA) module that ensures the selected messages meet the organization or user-specific needs and criteria. The QA module is designed to be flexible and customizable, allowing users to tailor the commit message generation process to their specific preferences and conventions (see Fig. 4).

The QA module incorporates rich embeddings for both code and text, prioritizing the most relevant and informative messages for the user. To assess the generated messages from the Generation module, the QA module leverages a graph-based representation of code and text, formed from a Graph Convolutional Network (gcn) (Kipf and Welling, 2016) and passed onto a paired Convolutional Neural Network (cnn) for training. This approach enables the selection of customized, organization-specific commit messages among the messages generated by the Generator module, depending on the training data provided for the QA module.

Below, we elaborate on the steps involved in implementing the QA module.

- After acquiring the *delta graph* as described in Section 3.3, we pass code tokens corresponding to the edges of *delta graph* ($\Delta G$) into CodeBERT (Feng et al., 2020) and extract the `[CLS]` token embedding to act as initial representation for Edges ($\Delta E$). The *delta graph* is then passed into a Graph Convolution Network trained on node classification task for predicting the type of Edge (added ($E_a$), deleted ($E_d$)

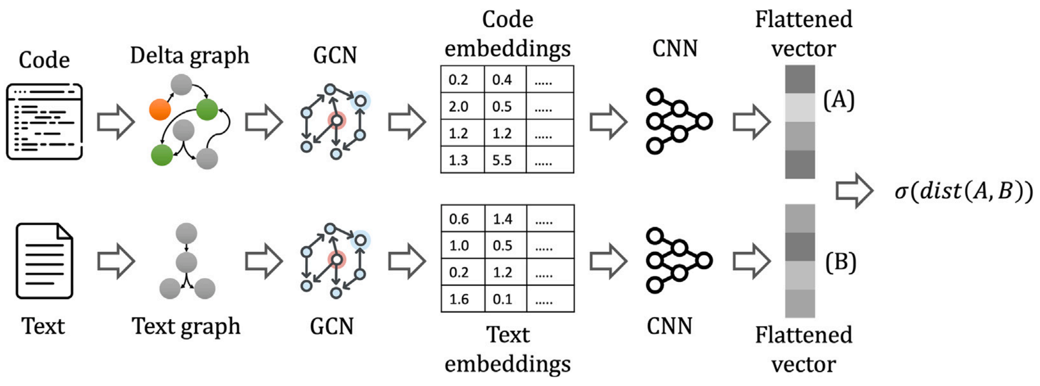

**Fig. 4.** Architecture of Quality Assurance module.

or common ($E_c$) to obtain the final representations.

- Words from a commit message are parsed into a dependency parse tree using the StanfordNLP parser (Manning et al., 2014) to leverage the grammar dependency property between words in a sentence.
- Word tokens corresponding to the edges of the dependency parse tree ($G_{text}$) are passed into BERT, and the [CLS] embeddings are extracted to serve as the initial edge ($E_{text}$) representations. The text graph is passed into a Graph Convolution Network trained on an incoming vertex ($V_{text}$) class prediction task to produce a final representation of the text graph.
- The final graph node representations are aggregated and are of size [len($E_{\Delta/text}$)×768], where len($E_{\Delta/text}$) is the number of Edges in the graph.

To train the QA module, we use a paired Convolutional Neural Network with shared weights. The training process involves labeling the *delta graph - message* pairs in binary notation based on user preferences for message quality (1: preferred and 0: not preferred). This approach treats the training as a binary classification problem. The paired CNN takes the aggregated representations from both the text and code Graph Neural Networks (GNNs) as input. The penultimate layer of the paired CNN is flattened and is passed through an additional neural network layer to obtain the final vector representations of code and text. We then calculate the Euclidean distance between these vector representations and pass the resulting scalar through a sigmoid activation to generate the probability of the code-message pair belonging to the positive class.

## 4. Experimental approach

In this section, we present the approach employed to address each of the research questions posed in this study.

### RQ1. Does capturing the code context improve the accuracy of the commit message generation model?

This research question aims to evaluate the impact of code context inclusion on the accuracy of commit message generation models. Additionally, it seeks to identify the optimal pre-trained model for this task, both with and without contextual information. We use the best-performing pre-trained model identified in RQ1 as the base model for Comet in subsequent research questions. All models considered in this research question use *delta graph* representations of the code changes. To evaluate the role of context tokens in boosting commit message generation accuracy, first, we choose a set of pre-trained models. Then, we fine-tune the models in two settings—with and without context tokens.

We use the following three pre-trained models based on the criteria mentioned in Section 3.4:

1. **GCB-GPT2:** The first model is formed using `GraphCodeBERT` (Guo et al., 2021) as the encoder and `GPT-2` (Radford et al., 2019) as the decoder. `GraphCodeBERT` is pre-trained on code with pre-training tasks such as edge prediction in data-flow graphs, equipping it with powerful programming knowledge. `GPT-2` is a popular open-source auto-regressive model trained on text data (English corpus majorly) and works well in text generation tasks favoring commit message generation.
2. **CodeT5:** It is a powerful encoder–decoder model with the architecture borrowed from T5 transformers. `CodeT5`'s challenging pre-training tasks make it a robust model that has demonstrated its effectiveness across various code-related tasks (Wang et al., 2021a).
3. **BLOOM:** The BLOOM model (Scao et al., 2022) has shown promising results across various zero-shot generalization tasks in code and natural language.

We fine-tune the models with two types of data: **AD** (containing added and deleted tokens) and **ADC** (containing added, deleted, and context tokens) to demonstrate the impact of incorporating context on the accuracy of the commit message generation model. All models use *delta graph* representations of the code changes as input.

We evaluate the models using the widely adopted metrics of BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004) scores. These metrics capture various aspects of the generated commit messages, such as n-gram overlap (BLEU), semantic similarity (METEOR), and longest common subsequence (ROUGE-L), providing a multi-faceted assessment of generation quality.

### RQ2. How does the proposed approach perform compared to existing state-of-the-art automated commit message generation approaches?

Through this research question, we compare our approach with existing commit message generation methods. We select the most relevant and latest commit message generation techniques as baselines, RACE (Shi et al., 2022), NNGen (Liu et al., 2018), ATOM (Liu et al., 2020), and Commit-BERT (Jung, 2021). RACE is a hybrid approach for commit message generation that utilizes a generation and retrieval module. NNGen is an information-retrieval-based model that retrieves the top-k commits and returns the one with the highest 4-gram precision match. ATOM utilizes

ASTs to represent code differences and leverages a *tree2seq* model to generate commit messages. However, we could not find their replication package; we reached out to the authors of the study and did not receive any response. Commit-BERT utilizes a pre-trained model CodeBERT to generate commit messages. The work first fine-tunes the model on CodeSearchNet (Husain et al., 2019) dataset to inject the domain knowledge into the model and then trains the model on the commit message generation dataset. We also considered FIRA (Dong et al., 2022) as one of our baseline approaches; however, we could not reproduce its implementation. Specifically, multiple implementation issues in their pre-processing scripts hindered our plans to include FIRA in our comparison.

These approaches utilize various techniques to encode commits, such as hybrid generation and retrieval (RACE), information retrieval (NNGen), and pre-trained models (Commit-BERT). However, none of these baselines use graph-based representations like our proposed approach. By comparing our method with these diverse baselines, we aim to demonstrate the effectiveness of our graph-based representation compared to representation chosen by other approaches and its impact on commit message generation performance.

We carry out our experiment on the selected baseline models along with our proposed approach Comet. We train the models on our dataset and document the reported performance metrics. We train the models on our dataset and document the reported performance metrics.

**RQ3. How does the proposed approach compare against the latest GPT models?**

ChatGPT (Huggingface, 2022)—a transformer-based model optimized for chat trained using Reinforcement Learning with Human Feedback (RLHF), has shown remarkable capability in performing code and text-related tasks (Laskar et al., 2023). This research question aims to compare the performance of our proposed approach against the *gpt-3.5-turbo-16k* and *gpt-4-1106-preview* models that power ChatGPT (Huggingface, 2022) in the commit message generation task. We conduct an extensive evaluation and present findings across zero-shot, one-shot, and multi-shot settings.

Prompt engineering has emerged as a new field of interest with the rise of LLM-powered chatbots. Previous research has identified various parameters that define a good prompt for LLM-based software such as ChatGPT, including factors such as prompt length, specificity, diversity, and relevance criteria (White et al., 2023; Tian et al., 2023). Following the recommended practices in existing research (White et al., 2023; Tian et al., 2023), we identified three key criteria for generating a prompt.

- **Completeness**: Prompts should clearly and concisely describe the task at hand.
- **Relevance**: Prompts should not contain any misleading or extraneous information.
- **Comprehensiveness**: Prompts should be sufficiently detailed to provide the necessary information.

To keep the prompt selection process unbiased, we conducted a survey with participants having software development experience to evaluate the effectiveness of diverse prompts generated following recommended practices (White et al., 2023; Tian et al., 2023; OpenAI Platform, 2023). The prompts were assessed based on the aforementioned criteria. The survey results helped identify the best prompts for each setting (zero-shot, one-shot, multi-shot) in our evaluation study.

For evaluation, we append code diffs from each commit in the test set to the selected prompts across the three scenarios. In one-shot and multi-shot settings, CodeBERT (Feng et al., 2020) is used to obtain vector representations of code diffs, and cosine similarity identifies the most similar code diffs to serve as examples. We implement chain-of-thought prompting (Wei et al., 2022) by including intermediate reasoning steps describing the input and expected output. The commit messages generated by the GPT APIs are evaluated using BLEU-NORM (Loyola et al., 2018), ROUGE-L (Lin, 2004), and METEOR (Banerjee and Lavie, 2005) metrics.

### 4.1. Experimental setup

After experimenting with various learning rates such as $5 \times 10^{-5}$, $1 \times 10^{-5}$, and $2 \times 10^{-5}$, we found that a learning rate of $5 \times 10^{-5}$ worked best for our model. This learning rate is also the default learning rate for the Adam optimizer, as shown in the official Hugging Face documentation (OpenAI, 2022). We implemented and trained the model using the Hugging Face implementation of transformers (Wolf et al., 2020), and used AdamW (Loshchilov and Hutter, 2017) optimizer with a learning rate of $5 \times 10^{-5}$ for fine-tuning the model over a maximum of 9 epochs, with a batch size of 16. We set the maximum input length to 512, and the maximum generated message length to 80, based on the data dimensions and the model's input limit. The models were trained on both a single NVIDIA P100 GPU and a single NVIDIA V100 GPU.

### 4.2. Metrics

We evaluate Comet on three popular metrics: BLEU (Papineni et al., 2002), ROUGE-L (Lin, 2004), and METEOR (Banerjee and Lavie, 2005) to demonstrate its efficiency. BLEU is a widely used precision-based metric that calculates the n-gram overlap between the predicted and reference candidates and averages them over the whole corpus. All the BLEU results reported in this paper are of BLEU-NORM (Loyola et al., 2018) variant. We evaluate and report up to 4 n-grams for BLEU. ROUGE is a recall-based metric that measures the overlap between the predicted and reference candidates in terms of n-gram matches. We use the version of ROUGE that calculates the score based on the longest common subsequence (LCS) between the predicted and reference candidates. METEOR is an F1-based metric that measures the quality of the generated text by mapping contiguous predicted unigrams to contiguous unigrams in the references and computing their harmonic mean with respect to precision and recall. These metrics have been commonly used in existing commit message generation techniques (Liu et al., 2020; Shi et al., 2022; Jung, 2021).

## 5. Results

In this section, we present our observations from the experiments for the considered research questions.

### 5.1. Results of RQ1: Does capturing the code context improve the accuracy of the commit message generation model?

In RQ1, we aim to evaluate the role of context tokens in boosting commit message generation accuracy. Comet uses *delta graph* to capture the context along with the code changes.

**Table 2**
RQ1: Comparison of models' performance.

| Model | Data | BLEU 1 | BLEU 2 | BLEU 3 | BLEU 4 | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|---|
| GCB-GPT2 | ADC | 10.09 | 3.9 | 1.9 | 1.0 | 10.3 | 8.2 |
| | AD | 9.6 | 3.7 | 1.76 | 0.91 | 9.84 | 7.76 |
| **CodeT5** | **ADC** | **17.33** | **8.96** | **5.4** | **3.53** | **12.26** | **13.45** |
| | AD | 15.58 | 7.83 | 4.73 | 3.11 | 11.49 | 11.85 |
| BLOOM | ADC | 6.12 | 1.9 | 0.8 | 0.4 | 5.49 | 5.9 |
| | AD | 8.85 | 3.1 | 1.3 | 0.6 | 6.87 | 6.88 |

**Table 3**
Manual evaluation results for RQ1.

| Model | Coverage | Conciseness |
|---|---|---|
| AD | 3.06 | 3.84 |
| ADC | **3.60** | **3.86** |
| Ref | 3.14 | 3.79 |

Table 2 presents results from the experiment. The findings demonstrate that `CodeT5` outperforms the other two models across all the metrics in both the data representations (AD and ADC) supplied as input. We have also observed that context tokens facilitate in 7% and 13.4% boost in performance on the BLEU metric for GCB-GPT2 and `CodeT5` respectively.

The results show that incorporating context leads to improvements in METEOR and ROUGE-L scores as well, indicating that the generated messages are more semantically similar to the references and contain longer shared subsequences when context is included. This supports the hypothesis that context tokens enhance the generated commit messages' relevance, completeness and coherence.

Although context tokens demonstrate a performance boost over all the metrics for the first two models, an interesting finding is that `BLOOM`, a decoder-only model shows an inferior performance with context tokens. One potential reason for the inferior performance of `BLOOM` with context tokens could be that the decoder-only architecture of `BLOOM` is not well-suited for capturing and incorporating context information. Unlike encoder–decoder models that have a separate component for encoding the input context, `BLOOM` solely relies on the decoder to generate the output tokens, which might not effectively utilize the contextual information. Further detailed exploration and experimentation is required to explain the performance degradation for the generative model. Even though `BLOOM` without context did better than with context, its performance is significantly lower than the other two models.

To complement the metrics-based analysis, we conduct a manual evaluation of the generated commit messages using the following criteria:

- **Coverage**: How well does the generated message cover all important aspects of the code difference and ground truth message?
- **Conciseness**: How clear and succinct is the message without losing essential information?

To conduct the manual assessment, we sought graduate-level software engineering student volunteers within our university. A total of 7 students (5 PhD and 2 master's) working on software engineering topics volunteered for the assessment. Each reviewer rates the messages, without knowing the source of the messages, on a scale of one to five on both assessment criteria, where a score of five indicates the message's high effectiveness in meeting the specific criterion.

The results, presented in Table 3, show COMET's ability to balance coverage and conciseness. Specifically, on coverage criterion, COMET with added, deleted, and common context tokens (ADC) achieved an average score of 3.60, outperforming both the version without context (AD: 3.06) and the reference messages (3.14). In terms of conciseness, both versions of COMET (AD: 3.84, ADC: 3.86) slightly outperformed the reference messages (3.79).

These results indicate that including context tokens enhances COMET's ability to capture important aspects of code changes without significantly compromising conciseness. The ADC version of COMET demonstrates an effective balance between comprehensive coverage and succinct expression, outperforming both the AD version and the reference messages.

Based on both automated metrics and manual evaluation results, we identify CodeT5 fine-tuned with ADC data as the best-performing model, which was used to answer subsequent research questions.

**Summary of RQ1:** Our findings suggest that incorporating contextual representation significantly improves the performance of encoder–decoder based models. We observe an improvement of 7% and 13.4% on the BLEU metric for considered Encoder-Decoder models.

### 5.2. Results of RQ2: How does the proposed approach perform compared to existing state-of-the-art automated commit message generation approaches?

We present the results of the experiment in Table 4 presents the comparative results of COMET and baseline approaches across various performance metrics. We also create a variant of COMET, referred to as $\text{COMET}_{gd}$, which uses git-diff as input instead of our *delta graph* representation while keeping all other aspects of the model architecture constant. The variant provides us an opportunity to isolate the impact of *delta graph* representation.

Between, COMET and $\text{COMET}_{gd}$, the results demonstrate that COMET significantly outperforms $\text{COMET}_{gd}$ across all metrics. This comparison highlights the substantial contribution of the *delta graph* representation to COMET's performance beyond the benefits of the overall model architecture. The improved results can be attributed to the graph structure's ability to capture complex code relationships and changes more effectively than a textual diff representation. Notably, COMETgd still outperforms some other baselines like Commit-BERT and NNGen, underscoring the strength of our overall model architecture. However, the significant performance boost provided by the *delta graph* representation is evident when comparing COMET to $\text{COMET}_{gd}$.

**Table 4**
Comparison of COMET with baseline approaches.

| Baseline | BLEU 1 | BLEU 2 | BLEU 3 | BLEU 4 | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| RACE | 13.5 | 7.3 | 4.7 | 3.3 | 11.9 | **13.6** |
| Commit-BERT | 1.70 | 0.70 | 0.30 | 0.20 | 1.74 | 1.20 |
| NNGen | 3.5 | 1.6 | 1.02 | 0.7 | 6.03 | 6.21 |
| **COMET** | **17.33** | **8.96** | **5.4** | **3.53** | **12.26** | 13.45 |
| $\text{COMET}_{gd}$ | 10.58 | 4.67 | 2.64 | 1.69 | 8.00 | 10.34 |

**Table 5**
Performance of COMET on the unfiltered dataset.

| Baseline | BLEU 1 | BLEU 2 | BLEU 3 | BLEU 4 | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| RACE | 6.3 | 2.8 | 1.3 | 0.5 | 12.1 | 7.5 |
| **COMET** | **8.7** | **4.7** | **3** | **1.96** | **14.4** | **12.3** |

The results indicate that COMET outperforms all the evaluated approaches in all metrics except for ROUGE-L, where RACE perform better than COMET. Our model improves 17.7% on BLEU metrics compared to the existing state-of-the-art approach.

We conduct a Wilcoxon signed-rank test to compare the performance of COMET and RACE to ensure the statistical significance. The Wilcoxon signed-rank test is a non-parametric test suitable for comparing paired samples. We set the significance level at $\alpha = 0.05$. The test reveals statistically significant differences between COMET and RACE for several metrics. While RACE outperform COMET with statistical significance in terms of the ROUGE metric (statistic=$7.18e+04$, p-value=$1.50e-02$), COMET demonstrate its superiority by outperforming RACE with high statistical significance in terms of the BLEU-2 (statistic=$5.541e+03$, p-value=$2.00e-04$), BLEU-3 (statistic=$5.985e+02$, p-value=$3.00e-04$), and BLEU-4 (statistic=$1.13e+02$, p-value=$8.10e-03$) metrics. These results highlight the effectiveness of our proposed approach in generating commit messages that align well with the reference messages, as evidenced by the significantly higher BLEU scores. For the BLEU-1 (statistic=$7.87e+04$, p-value=$2.17e-01$) and METEOR (statistic=$9.07e+04$, p-value=$1.93e-01$) metrics, the test did not show statistically significant differences between COMET and RACE, thus preventing definitive conclusions about their comparison.

The statistical test results confirm that COMET outperforms RACE with statistical significance in terms of the crucial BLEU-2, BLEU-3, and BLEU-4 metrics, which are widely used to assess the quality of generated text. These findings provide strong evidence for the effectiveness of our proposed approach in capturing the essential information and generating commit messages that closely resemble the reference messages. Although RACE outperforms COMET in terms of the ROUGE metric, it is important to note that the overall evaluation results presented in Table 4 demonstrate that COMET is highly competitive with RACE across all metrics. COMET achieves the highest scores in five out of six metrics (BLEU-1, BLEU-2, BLEU-3, BLEU-4, and METEOR), while being comparable to RACE in terms of the ROUGE-L metric. These results underscore the robustness and effectiveness of our proposed approach in generating high-quality commit messages.

To further compare COMET's performance with RACE, we evaluate them on an unfiltered dataset of randomly sampled 125 commits from MCMD dataset, encompassing both meaningful and non-meaningful scenarios. This assessment shed additional light to COMET's real-world performance.

Table 5 presents the result of this evaluation, demonstrating COMET's superior performance over RACE across all metrics on this unfiltered subset. These findings reinforce the effectiveness and generalizability of our proposed approach in generating informative commit messages across diverse commit scenarios.

Moreover, we analyze the size distribution of the ground-truth and COMET generated messages to assess their descriptiveness. Our analysis reveals that COMET messages tend to provide a more detailed description of the code changes compared to the ground-truth messages. On average, COMET messages contain 6.61 words, while the ground-truth messages have 6.52 words. In comparison, RACE messages have an average of 4.50 words, indicating that COMET generates more descriptive commit messages than RACE.

To gain more insight into the performance of the evaluated approaches, we discuss an example of generated commit messages for the code diff presented in Listing 2. The listing illustrates a code change where the condition in an if statement is modified from `root.waitThis().size() >= 0` to `root.waitThis().size() > 0`. While the ground truth message captures the essence of the change, it does not provide a clear explanation of the modified code. In contrast, the commit message generated by COMET not only captures the context of the code change successfully but also offers a more detailed and accurate description of the revised code. By explaining that the if condition is changed to check if the size of `root.waitThis()` is greater than zero, COMET's generated message provides a clearer understanding of the intended modification compared to the ground truth. This example highlights the effectiveness of our approach in generating informative commit messages that explain the changed code, even surpassing the ground truth in terms of clarity and completeness.

Commit messages generated by considered approaches are produced below.

- **Ground truth:** "fixed small issue with if-condition"
- **RACE:** "Lock graph manager fix"
- **NNGen:** "#315 Pure comment queries fix"
- **CommitBert:** "[hotfix][tests] Remove unused import"
- **COMET:** "LockGraph: only create root node if there are any locks."

Below, we also provide an example where RACE achieves a higher ROUGE score compared to COMET, even though the COMET-generated message can be argued more meaningful and informative.

- **Ground truth:** "Polish"
- **Predicted message by RACE:** "Polish OnEndpointElementCondition"
- **Predicted message by COMET:** "close cache if null is specified as a value"

```
diff --git a/LockGraphManager_old.java b/LockGraphManager_new.java
index 24c9111..a5835c5 100644
--- a/LockGraphManager_old.java
+++ b/LockGraphManager_new.java
@@ -174,7 +174,7 @@

public abstract class LockGraphManager<LOCK_TYPE extends DBAServerLock<?>, ID_TY
for(LOCK_TYPE root : roots) {
- if (root.waitThis().size() >= 0)

+ if (root.waitThis().size() > 0)
 createGraph(root);
 }
```

**Listing 2** Sample code change for commit message generation using baseline techniques.

**Table 6**
Commit message scoring criteria for manual assessment.

| Score | Description |
|---|---|
| 0 | Neither relevant in semantic nor having shared tokens |
| 1 | Irrelevant in semantic but with some shared tokens |
| 2 | Partially similar in semantic, but contains exclusive information |
| 3 | Highly similar but not semantically identical |
| 4 | Identical in semantic |

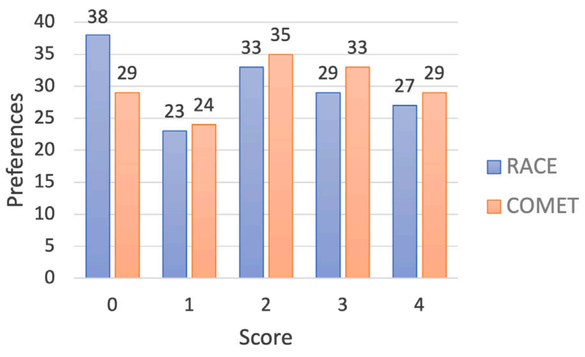

**Fig. 5.** Results of qualitative assessment.

In this example, the RACE-generated message, “Polish OnEndpointElementCondition”, aligns more closely with the ground truth message at a surface level, as it includes the keyword “Polish”. This results in a higher ROUGE score for the RACE prediction. However, the Comet-generated message, “close cache if null is specified as a value”, provides more actionable information about the specific code change being made.

Motivated by the above example and aiming to avoid demonstrating only the example where our method is giving best results, we conducted a qualitative assessment of our proposed approach in comparison to RACE (Shi et al., 2022). RACE is the state-of-the-art model for commit message generation. To evaluate and compare the performance of both the approaches, we created a survey inspired by similar studies (Liu et al., 2020; Dong et al., 2022). We randomly sampled 50 data points from the test set for the evaluation process. Every data point includes a ground truth message, along with two predicted messages generated separately by RACE and Comet. We invited graduate student volunteers to participate in this assessment from the Faculty of Computer Science of Dalhousie University. Three Ph.D. students with prior software development experience volunteered to assess the predicted messages against the ground truth data. To minimize bias, we ensured that the model responsible for each predicted message remained anonymous to the participants. The participants were provided with an existing scoring criteria from previous similar studies (Liu et al., 2018; Dong et al., 2022) as shown in Table 6.

Each participant assessed all the 50 samples and assigned a score based on the provided scoring criteria, considering the ground-truth message as the reference. Hence, we obtained a total of 150 observations. Fig. 5 presents the distribution of all the obtained observations for RACE and Comet. RACE messages scored an average of 2.56, while Comet achieved an average of 2.72. **Therefore, we conclude that Comet performed better than RACE in an anonymous qualitative assessment.**

Table 7 shows three samples randomly picked from the evaluation set. One may find their corresponding code diffs in our replication package (Smart lab Dalhousie university, 2023) in *Qualitative-Evaluation* folder. These examples show that Comet generates commit messages with more contextual information compared to the state-of-the-art approach or even the ground-truth message.

**Summary of RQ2:** The results show that Comet surpasses the state-of-the-art model by an average of 12.2% across all six metrics, establishing itself as an effective approach for generating high-quality commit messages. Furthermore, our manual comparative assessment shows a stronger preference for Comet compared to the state-of-the-art approach.

**Table 7**
Qualitative assessment samples.

| | |
|---|---|
| Ground-truth | RateLimiter fix (#2229) |
| RACE | refactoring |
| COMET | Update RateLimiterLongRunningUnitTest.java |
| Ground-truth | refactor demo |
| RACE | refactoring |
| COMET | apollo config demo improvements |
| Ground-truth | rename to assertGenerateKeyWithClockCallBack() |
| RACE | Update DefaultKeyGeneratorTest.java |
| COMET | Change test method name to use new method name |

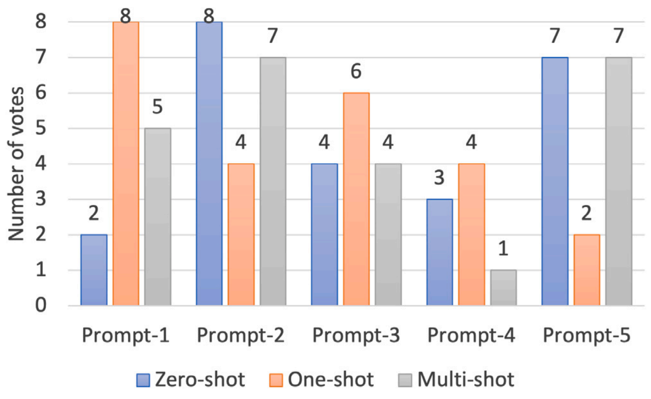

**Fig. 6.** Prompt selection survey.

## 5.3. Results of RQ3: How does the proposed approach compare against the latest GPT models?

After carefully considering the aforementioned criteria along with GPT best practices suggested by OpenAI (OpenAI Platform, 2023), we generated five diverse prompts that can be found in our replication package.[2] To keep the prompt selection process unbiased, we conducted a survey. We selected a diverse set of participants with software development experience to evaluate the effectiveness of the prompts. The participants were three Ph.D. students, four master's students, and one bachelor's student. We provided them with the code diff, reference message, and the message predicted by `gpt-3.5-turbo` for all three settings (*i.e.,* zero-shot, one-shot, and multi-shot) mentioned. The participants were asked to choose the top three prompts in each scenario following the criteria defined above, wherein the predicted messages are most similar to the reference message.

Fig. 6 shows the votes received for each prompt across previously mentioned cases from the participants. By using voting, we obtained a consensus view of the best prompts to use for each case in our evaluation study. It also helped ensure that the prompts were high quality and met the criteria for this study. The survey results, selected prompts and messages generated by `gpt-3.5-turbo` and `gpt-4` can be accessed online.[3]

For evaluation, we append code diffs from each commit in the test set to the selected prompts across three scenarios. Specifically, for one-shot and multi-shot settings, `CodeBERT` (Feng et al., 2020) is utilized to acquire vector representations of all the code diffs and cosine similarity is employed to identify the most similar code diffs, which serve as examples in this case. In order to utilize the model to the best of its capability, we implement the chain-of-thought prompting (Wei et al., 2022) technique. Instead of merely stacking code diffs and messages to be passed as input, we include intermediate reasoning steps that provide a description of the input and expected output. We used the `gpt-3.5-turbo` and `gpt-4` APIs provided by OpenAI to send each of these samples as individual API requests and collect the response for each of them. Once we had all the commit messages, we evaluated them using three metrics: BLEU-NORM (Loyola et al., 2018), ROUGE-L (Lin, 2004), and METEOR (Banerjee and Lavie, 2005). We present the findings from our experiments in Table 8.

Our results in Tables 8 and 9 clearly show that Comet outperform `gpt-3.5-turbo` model across five metrics and `gpt-4` model across four metrics by a significant margin. The performance of our approach is comparable with the gpt-3.5-turbo model's zero-shot prompt-2 setting and `gpt-4` model's zero-shot prompt-2 and multi-shot prompt-3 settings. Given that Comet uses `CodeT5`, which has 770 million parameters, while the ChatGPT models using `gpt-4` have hundreds of billions of parameters (Wikipedia Contributors, 2024), the performance gain is significant.

We extend our evaluation with a manual evaluation assessing the message quality based on two criteria: Coverage and Conciseness as defined in Section 5.1. To conduct the manual assessment, we engaged the same set of reviewers involved in RQ1 and followed the same review protocol. Each reviewer assessed individual messages using a 5-point scale, where a score of 5 indicates the message's high effectiveness in meeting a specific evaluation criterion.

Table 10 summarizes the results of the manual evaluation. **Comet achieves an average score of 3.95 across all criteria, outperforming the average GPT score of 3.57.**

These results highlight Comet's ability to generate concise yet informative commit messages. While some GPT prompts achieved higher scores in coverage, Comet's superior performance in conciseness (4.14) demonstrates its practical value in real-world software development scenarios,

[2] https://github.com/SMART-Dal/Comet/blob/main/ChatGPT/README.md
[3] https://github.com/SMART-Dal/Comet/tree/main/ChatGPT/RQ3

**Table 8**
GPT-3.5-turbo comparison results.

| Prompts | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| Zero-shot P1 | 6.49 | 2.69 | 1.39 | 0 | 11.01 | 9.48 |
| Zero-shot P2 | 10.21 | 4.2 | 2.17 | 1.25 | **12.3** | 10.24 |
| Zero-shot P3 | 8.67 | 3.7 | 1.93 | 0.11 | 10.22 | 11.8 |
| One-shot P1 | 12.64 | 6.02 | 3.66 | 2.55 | 11.1 | 11.46 |
| One-shot P2 | 11.11 | 4.88 | 2.67 | 1.67 | 10.92 | 11.3 |
| One-shot P3 | 11.2 | 5.03 | 2.82 | 1.8 | 11.36 | 10.8 |
| Multi-shot P1 | 8.68 | 4 | 2.28 | 1.47 | 10.95 | 11.6 |
| Multi-shot P2 | 4.99 | 2.18 | 1.2 | 0 | 11.42 | 10.5 |
| Multi-shot P3 | 6.71 | 2.9 | 1.57 | 0.96 | 11.27 | 11.44 |
| **Comet** | **17.33** | **8.96** | **5.4** | **3.53** | 12.26 | **13.45** |

**Table 9**
GPT-4 comparison results.

| Prompts | BLEU-1 | BLEU-2 | BLEU-3 | BLEU-4 | METEOR | ROUGE-L |
|---|---|---|---|---|---|---|
| Zero-shot P1 | 4.41 | 1.81 | 0.92 | 0.54 | 12.43 | 6.97 |
| Zero-shot P2 | 9.75 | 3.84 | 1.98 | 1.15 | **14.58** | 11.31 |
| Zero-shot P3 | 10.10 | 4.28 | 2.21 | 1.29 | 11.11 | 13.17 |
| One-shot P1 | 8.52 | 3.48 | 1.79 | 1.04 | 11.75 | 11.36 |
| One-shot P2 | 7.98 | 2.98 | 1.49 | 0.87 | 13.46 | 10.17 |
| One-shot P3 | 4.74 | 1.86 | 0.91 | 0.51 | 12.31 | 7.13 |
| Multi-shot P1 | 9.53 | 4.50 | 2.66 | 1.77 | 12.90 | 12.48 |
| Multi-shot P2 | 2.54 | 1.08 | 0.60 | 0.39 | 10.48 | 5.33 |
| Multi-shot P3 | 14.26 | 7.37 | 4.70 | 3.38 | 13.31 | **14.63** |
| **COMET** | **17.33** | **8.96** | **5.4** | **3.53** | 12.26 | 13.45 |

**Table 10**
Manual evaluation results for RQ3.

| Model/Prompt | Coverage | Conciseness |
|---|---|---|
| Zero-shot P1 | **4.17** | 3.34 |
| Zero-shot P2 | 3.86 | 4.05 |
| Zero-shot P3 | 3.27 | 3.92 |
| One-shot P1 | 3.34 | 3.68 |
| One-shot P2 | 3.19 | 2.97 |
| One-shot P3 | 3.08 | 3.69 |
| Multi-shot P1 | 3.58 | 3.53 |
| Multi-shot P2 | 3.58 | 3.97 |
| Multi-shot P3 | 3.98 | 3.15 |
| **COMET** | 3.76 | **4.14** |

where brevity is often crucial. Comet also maintains competitive coverage (3.76), outperforming most GPT variants. The combination of manual evaluation results and automated metrics validates Comet's effectiveness. It demonstrates its practical value in generating commit messages that balance informativeness and conciseness, which is crucial for software development workflows where developers need quick yet comprehensive insights into code changes.

**Summary of RQ3:** Based on our extensive evaluation, Comet demonstrates a competitive capability in generating high-quality commit messages compared to the most popular GPT 3.5 model and the most capable GPT 4 model.

## 5.4. Discussion

### 5.4.1. QA module

The effectiveness of the QA module relies on the training data and the specific quality criteria defined by the user or the organization. Therefore, the contribution of the QA module to the overall performance of Comet may vary depending on the specific use case and the desired commit message conventions.

To demonstrate the functionality and effectiveness of the QA module, we train the QA module to classify commit messages containing reference numbers. Reference numbers in commit messages (*e.g.,* "Fix bug #1234" or "Implement feature request JIRA-5678") are important as they link code changes to specific issues, bugs, or feature requests tracked in project management systems. This practice enhances traceability, facilitates code review, and improves project management.

We utilize the GPT-4o-mini model for identifying such messages in order to curate the dataset. Specifically, we use the following prompt: *"You are a commit message classifier. Respond with 1 if the message contains a reference number or issue, and 0 if it does not. Classify this commit message: {message}."* We collect negative samples for corresponding diffs without any reference numbers. This process yielded a curated dataset of 5,224 samples, which was subsequently split into training, validation, and test sets with ratios of 75%, 10%, and 15%, respectively.

To obtain the representations of diffs and corresponding message samples (both positive and negative), we passed the code diffs through `CodeBERT-base` (with a maximum token length of 512) and the commit messages through `BERT-base-uncased` (with a maximum length of 128) to extract embeddings. Using a similar architecture to the original QA module, the model was trained on this dataset. The resulting model achieved a precision of 96%, recall of 93%, and an accuracy of 95.28%. This evaluation demonstrates the QA module's ability to effectively classify commit messages based on the presence of reference numbers, showcasing its functionality and potential to adapt to specific quality criteria. The experiment outlined above assesses the QA module's ability to classify messages based on defined criteria, demonstrating its effectiveness in distinguishing between various message types. The experiment retains a structure similar to the original QA module, except for the method of representation. We use transformer-based encoding for this experiment. Though it is possible to use the same input representation as the origin QA module, this change demonstrates that QA module implementation can be tweaked without changing the expectation from the model to choose one of the options based on the preferences baked in the model.

To further illustrate the flexibility and customizability of the QA module, we present qualitative examples below showcasing how it can be adapted to different organization-specific commit message conventions.

- **Scenario 1**: Organization A follows a convention where commit messages should start with a verb in the imperative form, followed by a concise description of the changes. The QA module can be trained on commit messages adhering to this convention, ensuring that the generated messages align with the organization's preferred style. Example commit message:
  `Add user authentication endpoint`
- **Scenario 2**: Organization B emphasizes the inclusion of issue tracker references in commit messages to facilitate traceability. The QA module can be trained on commit messages that include issue tracker references, prioritizing the selection of messages that adhere to this convention. Example commit message:

  ```
  Fix rendering issue on mobile devices (Closes #123)
  ```

- **Scenario 3**: Organization C values detailed commit messages that explain the rationale behind the changes. The QA module can be trained on commit messages that provide comprehensive explanations, favoring messages that offer insights into the motivation and impact of the changes.
  Example commit message:

  ```
  Refactor data processing pipeline
  The existing data processing pipeline was causing performance bottlenecks
  due to inefficient algorithms and redundant operations. This refactoring aims
  to optimize the pipeline by:
  - Replacing the naive sorting algorithm with a more efficient one
  - Eliminating redundant data transformations
  - Introducing parallel processing for independent tasks

  These changes significantly improve the processing speed and reduce memory
  consumption, enabling the system to handle larger datasets efficiently.
  ```

These scenarios illustrate how the QA module can be customized to cater to different organization-specific commit message conventions. By training the QA module on relevant examples and quality criteria, it can effectively prioritize and select commit messages that align with the desired conventions.

The QA module receives multiple code-message pairs at the inference stage and calculates their probabilities. These probabilities are then used to rank the messages in descending order of their relevancy to the code change. The ranking of commit messages depends on the training samples and the chosen criteria. Our case study demonstrates the capabilities of the QA module, we invoked the QA module to rank the top **three** probable messages generated by the decoder for each data point. This was separate from our main experiments, where the QA module was not used.

The current version of Comet maintains a uniformity between the results generated by the decoder and the QA module's preference ranking as it is being offered as a flexible module to be moulded by the user itself. However, in the future, we plan to explore and implement different QA strategies that are tailored to the requirements of different organizations.

### *5.4.2. Input representation*

The use of structured representations such as asts and cpgs has shown promise in improving commit message generation. However, the scalability of this approach remains a concern due to the significant computational cost associated with building and manipulating trees or graphs. Existing studies tackle this issue by truncating the input, which results in loss of information. This issue calls for advanced techniques to capture relevant features and characteristics without bloating input representations.

The proposed *delta graph* representation, based on cpg (a combination of ast, pdg, and cfg), offers a unique approach to capturing code change information. By comparing our approach with existing state-of-the-art studies that employ various representations, such as RACE (Shi et al., 2022) (code diff encoder) and CommitBERT (Jung, 2021) (raw git diff), we effectively demonstrate the advantages of the *delta graph* representation. Our evaluation in RQ2 and RQ3 highlights the superiority of the *delta graph* approach over other representations, including the simple textual format used by ChatGPT.

### *5.4.3. Dataset quality*

Although there are several datasets available, the lack of high-quality commit message datasets is still a significant concern. Generating high-quality commit messages requires high-quality training data. The subpar quality of the ground truth dataset for commit messages can result in underestimating the quality of the generated commit messages even if they are of high quality, leading to unreliable evaluation metrics. Therefore, it is crucial to improve the quality of the ground truth dataset for commit messages for ensuring more accurate evaluation metrics and for achieving higher quality generated commit messages.

Our dataset, derived from the MCMD dataset after applying filtering criteria, covers a diverse range of Java projects, as shown in Table 1. However, we acknowledge that the majority of these projects are related to frameworks, libraries, and development tools, rather than specific end-user products such as web applications, company APIs, or open-source mobile applications. This predominance of certain project types could potentially impact the generalizability of our approach to a broader range of software projects.

It is important to note that our dataset does include some projects that are closer to end-user products, such as Android applications (e.g., ExoPlayer, Glide, Lottie) and various utility libraries (e.g., Guava, Gson, RxJava) that are often used in the development of end-user applications. These inclusions may help bridge the gap between the language used in frameworks/libraries and that used in end-user products. However, the commit messages for these different project types may vary in their use of technical language versus everyday vocabulary, which could affect the nature of the generated commit messages.

To address this limitation and enhance the generalizability of commit message generation models, we suggest several directions for future work. First, researchers could explore the development of models tailored to specific project types, taking into account their unique language and terminology. This could involve creating separate models or fine-tuning existing models for different categories of projects. Second, expanding the dataset to include a more diverse range of projects, particularly those related to end-user products, would provide a more comprehensive representation of real-world software development scenarios. Lastly, conducting comparative analyses of commit messages across different project types could yield valuable insights into how language and context vary, potentially informing more adaptive and versatile commit message generation approaches.

#### *5.4.4. Evaluation metrics*

Also, current accuracy metrics for evaluating generated commit messages have shown limitations in capturing the message's quality, as discussed in several recent studies, including Dey et al. (2022). Construct validity of the commonly used metrics often attract criticism. Therefore, more research is necessary to develop more effective metrics for evaluating the quality of generated commit messages.

## 6. Related work

Studies on generating commit messages automatically can be broadly classified into four categories: *machine learning-based*, *rule or template-based*, *information retrieval-based*, and *hybrid* approaches (Tao et al., 2021).

### *6.1. Machine learning-based methods*

The initial work in this domain was limited to classifying changes made to the source code using traditional machine learning techniques. Such efforts include classifying source code changes to categories of maintenance tasks (Hindle et al., 2009; Levin and Yehudai, 2017), and clustering commits to understand the intent of implementation (Yamauchi et al., 2014). Recent advancements in generation techniques, such as neural machine translation and attention mechanisms, have shown promising results in generating concise and meaningful commit messages by considering code change information. However, early studies, *e.g.,* neural machine translation (NMT)-based approaches (Jiang et al., 2017; Loyola et al., 2017; van Hal et al., 2019) treat code as a flat sequence of tokens, ignoring syntactic and semantic code change information. Such approaches suffer from a limited vocabulary of the most frequent words and fail to learn the semantics of the code changes.

To overcome these limitations, *CoDiSum* (Xu et al.) utilizes an attention mechanism and a multi-layer bidirectional GRU to consider both code structure and semantic information. Additionally, they employ a "copying" mechanism to address the out-of-vocabulary (OOV) problem by directly copying words from code diffs to the commit message. However, *CoDiSum* falls short in capturing the actual code structure as it treats code as a linear sequence of tokens. It also has a limited ability to capture long-term dependencies. Fira (Dong et al., 2022) is a graph-based model that uses fine-grained graphs to represent code changes, a graph neural network in the encoder to encode graph-structured inputs, and a transformer and dual copy mechanism to generate commit messages.

One common and major shortcoming of all of these early deep learning-based approaches is the need for their models to be trained from scratch. Due to this, these approaches are inefficient and exhibit a low capacity to capture the context between natural language (NL) and programming language (PL). This led to the development of large language model-based approaches that leverage pre-trained models trained on large corpora of NL-PL pairs. For example, CommitBERT (Jung, 2021) leverages CodeBERT (Feng et al., 2020), a code-based language model pre-trained on source code.

### *6.2. Rule-based approaches*

Buse and Weimer (2010) proposed *DeltaDoc*, an automatic technique that generates textual descriptions of source code modifications using symbolic execution and template-based summarization techniques. Similarly, Linares-Vásquez et al. (2015) proposed *ChangeScribe*, a technique to generate natural language commit messages by taking into account various factors, such as commit stereotypes, types of code changes (*e.g.,* file renames, property updates), and the impact of underlying code changes (*i.e.,* the relative number of methods impacted by a class in the commit). Another rule-based approach, *ARENA*, was proposed by Moreno et al. (2017) to address the problem of generating release notes, which involves summarizing changes made in the source code. *ARENA* extracts information about code changes, such as the files, classes, methods, and variables modified, as well as any deprecated classes, methods, or variables. It then uses predefined templates for the kind of artifact and kind of change to summarize code changes.

Despite the usefulness of these rule-based approaches, they have limited capability in capturing the intent of code changes, as they are only suitable for certain types of code changes. Additionally, they tend to generate verbose messages that lack specificity, making them unsuitable as commit messages without human intervention.

### *6.3. Information retrieval-based approaches*

Rastkar and Murphy (2013) proposed the use of multi-document summarization and information retrieval techniques to generate a concise natural language description of code changes. The approach captures the rationale of code changes, providing developers with better information about changes in commits. Liu et al. (2018) proposed *NNGen*, which relies on a bag-of-words model that calculates cosine similarity between the target code-diff and code-diffs in the training corpus. Out of the code-diffs with the top *k* scores, NNGen extracts commit messages based on the code-diff with the highest BLEU score. Huang et al. (2020) proposed *ChangeDoc*, a method for automatic commit message generation that utilizes existing commit messages in version control systems. ChangeDoc extracts similar messages by considering both semantic and syntactic information and uses this information to generate new commit messages. Wang et al. (2021b) took a different path with their proposed solution called *QAcom* where they automatically assess the quality of commit messages generated by various other approaches. This is achieved through retrieval techniques based on semantic relevance, which helps filter out poor-quality messages and retain high-quality ones.

Although promising, these approaches have limitations. They require a training dataset for reference, which can be limited in scope, leading to an out-of-vocabulary issue and potentially generating irrelevant or redundant messages. These limitations can result in poor quality and ineffective commit messages.

### *6.4. Hybrid approaches*

ATOM (Liu et al., 2020), incorporates an abstract syntax tree for representing code changes and integrates both retrieved and generated messages through hybrid ranking. Furthermore, Contextualized Code Representation Learning (CoreGen) (Nie et al., 2021) leverages contextualized code representation learning strategies to improve the quality of commit messages. However, both approaches suffer from *exposure bias* (Ranzato et al., 2016) issues. Wang et al. (2021b) proposed *CoRec*, a hybrid model that combines generation-based and retrieval-based techniques. CoRec addresses the limitations of generation-based models, such as ignoring low-frequency words and exposure bias, by using an information retrieval module and decay sampling mechanism. Shi et al. (2022) introduced RACE, a neural commit message generation model that incorporates retrieval-based techniques. RACE utilizes similar commits to guide the neural network model in generating informative and readable commit messages.

The hybrid approaches discussed above have demonstrated impressive performance. However, they fail to account for the impact of a code change on the source code's control flow and data dependency and often neglect the context of the entire source code. Furthermore, their reliance on a retrieval module requires a training dataset during inference, which limits their scalability in real-world scenarios. This paper aims to provide a solution to address these identified limitations.

### *6.5. Code change representation*

Yin et al. (2019) introduced the problem of learning distributed representations of code edits. By combining a "neural editor" with an "edit encoder", their models learn to represent the salient information of an edit and can be used to apply edits to new code inputs. Hoang et al. (2020) proposed CC2Vec, a neural network model that learns a distributed representation of code changes guided by their accompanying log messages, which represent the semantic intent of the changes. CC2Vec models the hierarchical structure of a code change using an attention mechanism and multiple comparison functions to identify the differences between the removed and added code. Liu et al. (2023b) proposed CCRep, a code change representation learning approach that encodes code changes as feature vectors for diverse downstream tasks. CCRep leverages a pre-trained code model to obtain high-quality contextual embeddings of code and uses a novel query back mechanism to extract and encode the changed code fragments, making them explicitly interact with the whole code change.

### *6.6. Quality assurance for commit message generation*

Ensuring the quality of automatically generated commit messages is crucial for their practical adoption. Jiang et al. (2017) proposed an approach to automatically generate commit messages from code changes and conducted a human evaluation to assess the quality of the generated messages. They found that the generated messages were of comparable quality to human-written ones in terms of content adequacy, conciseness, and expressiveness. Liu et al. (2018) investigated the correlation between automatic evaluation metrics (e.g., BLEU, ROUGE) and human judgments for commit message generation. They observed that while automatic metrics provide a reasonable approximation of human assessment, there is still room for improvement in capturing the semantic similarity between generated and reference messages. To further enhance the quality of generated commit messages, researchers have explored various techniques such as incorporating code structure information (Xu et al.), utilizing pre-trained language models (Liu et al., 2023b), and leveraging cross-project knowledge. These efforts aim to generate more accurate, informative, and human-like commit messages. However, a significant proportion of generated messages may still be semantically irrelevant, potentially misleading developers and hindering practical application. To address this issue, Wang et al. (2021b) proposed QAcom, an automated Quality Assurance framework for commit message generation. QAcom aims to assure the quality of generated messages by automatically filtering out semantically-irrelevant ones while preserving semantically-relevant messages.

### *6.7. Graph-based representations for code analysis*

Graph-based representations have proven to be highly effective in various code analysis and software engineering tasks. Ma et al. (2022) proposed GraphCode2Vec, a self-supervised pre-training approach that produces task-agnostic embeddings of lexical and program dependence features using Graph Neural Networks. This method has demonstrated superior performance compared to generic and task-specific baselines in a range of software engineering tasks, showcasing the power of combining code analysis with graph-based techniques.

Multivariate graphs play a crucial role in program comprehension, particularly in the context of software maintenance and evolution. Diehl and Telea (2013) highlight the importance of multivariate graphs and discuss various visualization solutions designed specifically for software engineering applications. Graph visualization techniques have long been recognized as valuable tools in software analysis, enabling the quick and attractive display of directed graphs to reveal important structural information that may not be apparent in abstract representations (Gansner et al., 1993).

Recent studies have shown that integrating semantic graphs, such as Control Flow Graph (CFG) and Program Dependence Graph (PDG), with Abstract Syntax Tree (AST) can lead to improved performance on software engineering tasks compared to using AST alone (Swarna et al., 2021; Vagavolu et al., 2021). Robillard and Murphy (2007) present an approach to representing concerns in source code using a concern graph representation, which is an abstract model that describes which parts of the source code are relevant to different concerns. Romanov et al. (2020) provides an overview of different approaches for representing software as graphs, which can then be used for machine learning applications. Horwitz and Reps (1992) describe a language-independent program representation called the program dependence graph and discuss how this representation, along with operations like program slicing, can be used to build powerful programming tools that address important software engineering problems.

Mining code change patterns using graph-based approaches has gained significant attention in recent years. Nguyen et al. (2019) introduced CPatMiner, a graph-based approach that detects semantic change patterns across various development activities, outperforming AST-based techniques. Similarly, Janke and Mäder (2022) proposed a graph-based method that captures context relations between edits, identifying meaningful patterns across projects. Negara et al. (2014) developed an algorithm to mine fine-grained code changes, revealing ten popular high-level program transformations. To facilitate pattern mining, Martinez and Martin (2018) introduced Coming, a tool that analyzes Git repositories to find instances of user-specified change patterns. These approaches demonstrate the value of fine-grained code change analysis in detecting unknown patterns, understanding software evolution, and potentially improving developer productivity through automated code assistance tools.

Our work builds upon these existing graph-based techniques while introducing several key innovations tailored specifically for commit message generation. Unlike previous approaches that use general-purpose graph representations, our *delta graph* is designed to capture the nuances of code changes more effectively. Our approach uniquely integrates structural information from Abstract Syntax Trees (ASTs) with semantic information from Control Flow Graphs (CFGs) and Program Dependence Graphs (PDGs) into a single, unified representation. This integration allows for a more comprehensive understanding of code changes, encompassing not just syntactic modifications but also alterations in control flow and data dependencies.

A distinguishing feature of our *delta graph* is its incorporation of contextual information from unchanged parts of the code that are semantically related to the changes. This context-aware representation enables the generation of more informative and relevant commit messages. Furthermore, our approach explicitly models the relationships between added, deleted, and unchanged code elements, providing a more nuanced view of code evolution compared to existing methods that often focus solely on the changed parts.

While many existing approaches use graph representations as intermediate steps in their analysis, we directly utilize the *delta graph* as input to our commit message generation model. This end-to-end approach allows the model to learn directly from the rich, structured representation of code changes. Consequently, our method has the potential to generate more accurate and contextually relevant commit messages compared to techniques that rely on flattened or simplified representations of code changes.

By addressing the specific challenges of commit message generation through our specialized *delta graph* representation, our work extends the application of graph-based techniques in software engineering to a new and important domain, potentially opening up new avenues for research in automated documentation of code changes.

## 7. Threats to validity

**Construct validity** concerns with the degree to which our analyses measure what we intend to analyze. To ensure that our implementation works as intended, we implemented automated tests and conducted manual code review for each module. Prompt selection process for `gpt-3.5-turbo` and `gpt-4` models can be seen as a threat to validity especially when the output of these models significantly depends on the provided prompts. To mitigate this threat, we sought participation from multiple participants with prior software development to help us select the prompts. During the survey, we provided the participants with the results of all prompts, alongside the corresponding ground truth message for a particular code diff. This helped to ensure that the prompts chosen were relevant, effective, and unbiased.

The automatic filtering of relevant commits could also be considered a threat to validity if no manual analysis is performed. To mitigate this threat, we randomly sampled a statistically significant subset of filtered commits and manually reviewed them to assess the accuracy of the filtering process. We found that the majority of filtered commits were correctly identified as relevant, demonstrating the high precision of the filtering mechanism.

**Internal validity** is associated with the ability to draw conclusions from the performed experiments and results. To minimize the associated threats with the implementation of compared techniques, we attempted to use the replication package provided by the respective authors. In some cases, we did not find replication packages or were outdated. In such cases, we contacted the authors of the paper for further clarification. Furthermore, we attempted to implement the approach ourselves if their replication package was not available or not reproducible. Specifically, we implement CommitBert based on the information provided in the original publication.

A potential threat relates to code analysis tools struggling with non-compiling commits, leading to inaccurate graph generation. This threat is mitigated in our study as we used the MCMD dataset, which generally contains compilable code. Additionally, Joern's fuzzy parsing capabilities can handle incomplete code, though we did not extensively rely on this feature. We encountered no significant issues with name resolution or incorrect graph generation, indicating that compilation-related challenges did not substantially impact our approach.

Another consideration is the computational overhead introduced by the generation of *delta graph*s. Our analysis shows that *delta graph* representations are generally a bit larger than Git diffs. This increased size is due to additional contextual information in the graph structure. While this richer representation can be beneficial for downstream tasks, it also introduces additional computational steps, including constructing Code Property Graphs (CPGs) and performing graph operations. This overhead could potentially impact the efficiency of our approach, particularly in scenarios requiring rapid processing of large numbers of commits. We acknowledge this limitation and suggest that future work could focus on optimizing the *delta graph* generation process to enhance overall efficiency.

**External validity** concerns with the ability to generalize the results. Our study focuses on code changes to Java programming language. However, *delta graph* and the rest of the implementation, by design, works independently from the programming language. A potential limitation of our study is the composition of our dataset, which is derived from the MCMD dataset. While it includes some projects that are closer to end-user products, such as Android applications (e.g., ExoPlayer, Glide, Lottie), many of the projects are related to frameworks, libraries, and development tools. This may limit the generalizability of our results to other types of software projects, such as web applications, company APIs, or other end-user products. Commit messages for these different project types may vary in their use of technical language versus everyday vocabulary. Future work could explore the development of commit message generation models tailored to specific project types, taking into account their unique language and terminology. To make our work reproducible, we make all the scripts, data, and developed tool available online (Smart lab Dalhousie university, 2023; COMET, 2023).

## 8. Conclusion and future work

In this work, we proposed a novel commit message generation approach viz. Comet. The proposed approach leverages a new code change representation technique, *delta graph*, and a quality assurance module combined with a Transformer-based generator to achieve the goal of effective commit message generation. Our results indicate that incorporating contextual information significantly improves the performance of encoder–decoder based models. Based on that, we implemented our approach that considers contextual information. We found that our approach achieves state-of-the-art performance compared to existing approaches. Furthermore, we conducted a comparison between our approach and the `gpt-3.5-turbo` and `gpt-4` api under zero-shot, one-shot, and multi-shot settings. The evaluation was performed against three prompts selected by participants in our survey. Our approach demonstrated a significant performance advantage over `gpt-3.5-turbo` and `gpt-4` model.

In the future, we would like to explore how the quality assurance module can be used to address other subjective aspects of commit messages beyond those addressed in this work. Also, we aim to conduct an extensive study of multiple quality aspects that facilitate leveraging the purpose of commit messages and modify the training approach of the QA module to fit the findings accordingly. Furthermore, our focus will extend to enhancing the scalability of the trained model, enabling it to handle larger codebases automatically, and maintaining real-time representations, which will result in faster inference times. We also plan to investigate the applicability of our approach to a wider range of project types, including end-user products, to address the potential limitations of our current dataset composition. Though we decided to include only the direct edges of the nodes in this study, we acknowledge that including more connected edges in the *delta graph* representation is an interesting research direction that warrants further exploration. In our future work, we intend to investigate the trade-offs between context richness and model performance by constructing *delta graph*s with varying levels of context and evaluating their impact on commit message generation.

## CRediT authorship contribution statement

**Abhinav Reddy Mandli:** Writing – review & editing, Writing – original draft, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Saurabhsingh Rajput:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Data curation, Conceptualization. **Tushar Sharma:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Funding acquisition, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Dataset is made available on Zenodo.

## References

Agrawal, Kapil, Amreen, Sadika, Mockus, Audris, 2015. Commit quality in five high performance computing projects. In: 2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science. IEEE, pp. 24–29.

Anon, 2019. Javadoc tool. URL https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html.

Banerjee, Satanjeev, Lavie, Alon, 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/Or Summarization. Association for Computational Linguistics, Ann Arbor, Michigan, pp. 65–72, URL https://aclanthology.org/W05-0909.

Barnett, Jacob G., Gathuru, Charles K., Soldano, Luke S., McIntosh, Shane, 2016. The relationship between commit message detail and defect proneness in java projects on github. In: Proceedings of the 13th International Conference on Mining Software Repositories. pp. 496–499.

Bhagat, Smriti, Cormode, Graham, Muthukrishnan, S., 2011. Node classification in social networks. arXiv preprint arXiv:1101.3291.

Brown, Tom, Mann, Benjamin, Ryder, Nick, Subbiah, Melanie, Kaplan, Jared D., Dhariwal, Prafulla, Neelakantan, Arvind, Shyam, Pranav, Sastry, Girish, Askell, Amanda, et al., 2020. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 33, 1877–1901.

Buse, Raymond P.L., Weimer, Westley R., 2010. Automatically documenting program changes. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. ASE '10, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450301169, pp. 33–42. http://dx.doi.org/10.1145/1858996.1859005.

COMET, 2023. Comet Models. Zenodo, http://dx.doi.org/10.5281/zenodo.7902315.

Cortés-Coy, Luis Fernando, Linares-Vásquez, Mario, Aponte, Jairo, Poshyvanyk, Denys, 2014. On automatically generating commit messages via summarization of source code changes. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 275–284.

Cui, Di, Wang, Qiangqiang, Wang, Siqi, Chi, Jianlei, Li, Jianan, Wang, Lu, Li, Qingshan, 2023. REMS: Recommending extract method refactoring opportunities via multi-view representation of code property graph. In: 2023 IEEE/ACM 31st International Conference on Program Comprehension. ICPC, pp. 191–202. http://dx.doi.org/10.1109/ICPC58990.2023.00034.

Dey, Samanta, Vinayakarao, Venkatesh, Gupta, Monika, Dechu, Sampath, 2022. Evaluating commit message generation: to BLEU or not to bleu? In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. pp. 31–35.

Diehl, Stephan, Telea, Alexandru Cristian, 2013. Multivariate graphs in software engineering. URL https://api.semanticscholar.org/CorpusID:13840902.

Dong, Jinhao, Lou, Yiling, Zhu, Qihao, Sun, Zeyu, Li, Zhilin, Zhang, Wenjie, Hao, Dan, 2022. FIRA: fine-grained graph-based code change representation for automated commit message generation. In: Proceedings of the 44th International Conference on Software Engineering. pp. 970–981.

Dwivedi, Yogesh K., Kshetri, Nir, Hughes, Laurie, Slade, Emma Louise, Jeyaraj, Anand, Kar, Arpan Kumar, Baabdullah, Abdullah M., Koohang, Alex, Raghavan, Vishnupriya, Ahuja, Manju, et al., 2023. "So what if ChatGPT wrote it?" Multidisciplinary perspectives on opportunities, challenges and implications of generative conversational ai for research, practice and policy. Int. J. Inf. Manage. 71, 102642.

Errica, Federico, Podda, Marco, Bacciu, Davide, Micheli, Alessio, 2019. A fair comparison of graph neural networks for graph classification. arXiv preprint arXiv:1912.09893.

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

Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.-P., 1993. A technique for drawing directed graphs. IEEE Trans. Softw. Eng. 19 (3), 214–230. http://dx.doi.org/10.1109/32.221135.

Gu, Xiaodong, Zhang, Hongyu, Zhang, Dongmei, Kim, Sunghun, 2016. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 631–642.

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

Hassan, Ahmed E., 2008. Automated classification of change messages in open source projects. In: Proceedings of the 2008 ACM Symposium on Applied Computing. pp. 837–841.

Hindle, Abram, German, Daniel M., Godfrey, Michael W., Holt, Richard C., 2009. Automatic classication of large changes into maintenance categories. In: 2009 IEEE 17th International Conference on Program Comprehension. pp. 30–39. http://dx.doi.org/10.1109/ICPC.2009.5090025.

Hoang, Thong, Kang, Hong Jin, Lo, David, Lawall, Julia, 2020. CC2vec: distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450371216, pp. 518–529. http://dx.doi.org/10.1145/3377811.3380361.

Horwitz, Susan, Reps, T., 1992. The use of program dependence graphs in software engineering. In: International Conference on Software Engineering. pp. 392–411, URL https://api.semanticscholar.org/CorpusID:2911262.

Huang, Yuan, Chen, Xiangping, Zou, Qiwen, Luo, Xiaonan, 2014. A probabilistic neural network-based approach for related software changes detection. In: 2014 21st Asia-Pacific Software Engineering Conference, vol. 1, IEEE, pp. 279–286.

Huang, Yuan, Jia, Nan, Zhou, Hao-Jie, Chen, Xiang-Ping, Zheng, Zi-Bin, Tang, Ming-Dong, 2020. Learning human-written commit messages to document code changes. J. Comput. Sci. Tech. 35, 1258–1277.

Huggingface, 2022. Trainer - transformers 4.2.0 documentation. https://huggingface.co/transformers/v4.2.2/main_classes/trainer.html. (Accessed 3 May 2023),

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

Janke, Mario, Mäder, Patrick, 2022. Graph based mining of code change patterns from version control commits. IEEE Trans. Softw. Eng. 48, 848–863, URL https://api.semanticscholar.org/CorpusID:226775935.

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

Joern Documentation Blog RSS, 2023. Code property graph: Joern documentation. https://docs.joern.io/code-property-graph. (Accessed 1 May 2023).

Jung, Tae-Hwan, 2021. Commitbert: Commit message generation using pre-trained programming language model. arXiv preprint arXiv:2105.14242.

Kipf, Thomas N., Welling, Max, 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

Laskar, Md Tahmid Rahman, Bari, M. Saiful, Rahman, Mizanur, Bhuiyan, Md Amran Hossen, Joty, Shafiq, Huang, Jimmy Xiangji, 2023. A systematic study and comprehensive evaluation of ChatGPT on benchmark datasets. arXiv preprint arXiv:2305.18486.

Levin, Stanislav, Yehudai, Amiram, 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 97–106.

Li, Qimai, Han, Zhichao, Wu, Xiao-Ming, 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, (no. 1).

Liben-Nowell, David, Kleinberg, Jon, 2003. The link prediction problem for social networks. In: Proceedings of the Twelfth International Conference on Information and Knowledge Management. pp. 556–559.

Lin, Chin-Yew, 2004. Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. pp. 74–81.

Linares-Vásquez, Mario, Cortés-Coy, Luis Fernando, Aponte, Jairo, Poshyvanyk, Denys, 2015. Changescribe: A tool for automatically generating commit messages. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, IEEE, pp. 709–712.

Liu, Shangqing, Gao, Cuiyun, Chen, Sen, Nie, Lun Yiu, Liu, Yang, 2020. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. IEEE Trans. Softw. Eng. 48 (5), 1800–1817.

Liu, Nelson F., Lin, Kevin, Hewitt, John, Paranjape, Ashwin, Bevilacqua, Michele, Petroni, Fabio, Liang, Percy, 2023a. Lost in the middle: How language models use long contexts. arXiv preprint arXiv:2307.03172.

Liu, Zhongxin, Tang, Zhijie, Xia, Xin, Yang, Xiaohu, 2023b. CCRep: Learning code change representations via pre-trained code model and query back. arXiv:2302.03924 [cs.SE], URL https://arxiv.org/abs/2302.03924.

Liu, Zhongxin, Xia, Xin, Hassan, Ahmed E., Lo, David, Xing, Zhenchang, Wang, Xinyu, 2018. Neural-machine-translation-based commit message generation: How far are we? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE '18, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450359375, pp. 373–384. http://dx.doi.org/10.1145/3238147.3238190.

Loshchilov, Ilya, Hutter, Frank, 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101.

Loyola, Pablo, Marrese-Taylor, Edison, Balazs, Jorge, Matsuo, Yutaka, Satoh, Fumiko, 2018. Content aware source code change description generation. In: Proceedings of the 11th International Conference on Natural Language Generation. pp. 119–128.

Loyola, Pablo, Marrese-Taylor, Edison, Matsuo, Yutaka, 2017. A neural architecture for generating natural language descriptions from source code changes. arXiv:1704.04856.

Ma, Wei, Zhao, Mengjie, Soremekun, Ezekiel, Hu, Qiang, Zhang, Jie, Papadakis, Mike, Cordy, Maxime, Xie, Xiaofei, Traon, Yves Le, 2022. GraphCode2Vec: Generic code embedding via lexical and program dependence analyses. arXiv:2112.01218 [cs.SE], URL https://arxiv.org/abs/2112.01218.

Manning, Christopher D., Surdeanu, Mihai, Bauer, John, Finkel, Jenny Rose, Bethard, Steven, McClosky, David, 2014. The stanford CoreNLP natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations. pp. 55–60.

Martinez, Matias, Martin, Monperrus, 2018. Coming: A tool for mining change pattern instances from git commits. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings. ICSE-Companion, pp. 79–82, URL https://api.semanticscholar.org/CorpusID:53046096.

Meng, Na, Jiang, Zijian, Zhong, Hao, 2021. Classifying code commits with convolutional neural networks. In: 2021 International Joint Conference on Neural Networks. IJCNN, pp. 1–8. http://dx.doi.org/10.1109/IJCNN52387.2021.9533534.

Moreno, Laura, Bavota, Gabriele, Penta, Massimiliano Di, Oliveto, Rocco, Marcus, Andrian, Canfora, Gerardo, 2017. ARENA: An approach for the automated generation of release notes. IEEE Trans. Softw. Eng. 43 (2), 106–127. http://dx.doi.org/10.1109/TSE.2016.2591536.

Negara, Stas, Codoban, Mihai, Dig, Danny, Johnson, Ralph E., 2014. Mining fine-grained code changes to detect unknown change patterns. In: Proceedings of the 36th International Conference on Software Engineering. URL https://api.semanticscholar.org/CorpusID:11601706.

Nguyen, Hoan Anh, Nguyen, Tien Nhut, Dig, Danny, Nguyen, S., Tran, Hieu Minh, Hilton, Michael C., 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, pp. 819–830, URL https://api.semanticscholar.org/CorpusID:174800424.

Nie, Lun Yiu, Gao, Cuiyun, Zhong, Zhicong, Lam, Wai, Liu, Yang, Xu, Zenglin, 2021. Coregen: contextualized code representation learning for commit message generation. Neurocomputing 459, 97–107.

OpenAI, 2022. Introducing chatgpt. https://openai.com/blog/chatgpt. (Accessed 3 May 2023),

OpenAI Platform, 2023. GPT best practices. URL https://platform.openai.com/docs/guides/gpt-best-practices.

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

qoomon, 2023. Git-conventional-commits. https://github.com/qoomon/git-conventional-commits. (Accessed 1 May 2023).

Radford, Alec, Wu, Jeffrey, Child, Rewon, Luan, David, Amodei, Dario, Sutskever, Ilya, et al., 2019. Language models are unsupervised multitask learners. OpenAI Blog 1 (8), 9.

Ranzato, Marc'Aurelio, Chopra, Sumit, Auli, Michael, Zaremba, Wojciech, 2016. Sequence level training with recurrent neural networks. arXiv:1511.06732.

Rastkar, Sarah, Murphy, Gail C., 2013. Why did this code change? In: 2013 35th International Conference on Software Engineering. ICSE, pp. 1193–1196. http://dx.doi.org/10.1109/ICSE.2013.6606676.

Rebai, Soumaya, Kessentini, Marouane, Alizadeh, Vahid, Sghaier, Oussama Ben, Kazman, Rick, 2020. Recommending refactorings via commit message analysis. Inf. Softw. Technol. 126, 106332.

Robillard, Martin P., Murphy, Gail C., 2007. Representing concerns in source code. ACM Trans. Softw. Eng. Methodol. 16, 3, URL https://api.semanticscholar.org/CorpusID:6453024.

Romanov, Vitaly, Ivanov, Vladimir, Succi, Giancarlo, 2020. Approaches for representing software as graphs for machine learning applications. In: 2020 International Computer Symposium (ICS). pp. 529–534, URL https://api.semanticscholar.org/CorpusID:232062298.

Sanders, Hillary, Saxe, Joshua, 2017. Garbage in, garbage out: how purportedly great ML models can be screwed up by bad data. Proc. Blackhat 2017.

Scao, Teven Le, Fan, Angela, Akiki, Christopher, Pavlick, Ellie, Ilić, Suzana, Hesslow, Daniel, Castagné, Roman, Luccioni, Alexandra Sasha, Yvon, François, Gallé, Matthias, et al., 2022. Bloom: A 176b-parameter open-access multilingual language model. arXiv preprint arXiv:2211.05100.

Scarselli, Franco, Gori, Marco, Tsoi, Ah Chung, Hagenbuchner, Markus, Monfardini, Gabriele, 2009. The graph neural network model. IEEE Trans. Neural Netw. 20 (1), 61–80. http://dx.doi.org/10.1109/TNN.2008.2005605.

Sharma, Tushar, Kechagia, Maria, Georgiou, Stefanos, Tiwari, Rohit, Vats, Indira, Moazen, Hadi, Sarro, Federica, 2024. A survey on machine learning techniques applied to source code. J. Syst. Softw. (ISSN: 0164-1212) 209, 111934. http://dx.doi.org/10.1016/j.jss.2023.111934, URL https://www.sciencedirect.com/science/article/pii/S0164121223003291.

Shi, Ensheng, Wang, Yanlin, Tao, Wei, Du, Lun, Zhang, Hongyu, Han, Shi, Zhang, Dongmei, Sun, Hongbin, 2022. RACE: Retrieval-augmented commit message generation. In: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. pp. 5520–5530.

Sillito, Jonathan, Murphy, Gail C., De Volder, Kris, 2008. Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. 34 (4), 434–451.

Smart lab Dalhousie university, 2023. COMET repository. URL https://github.com/SMART-Dal/Comet.

Spinellis, D., 2005. Version control systems. IEEE Softw. 22 (5), 108–109. http://dx.doi.org/10.1109/MS.2005.140.

Swarna, Karthik Chandra, Mathews, Noble Saji, Vagavolu, Dheeraj, Chimalakonda, Sridhar, 2021. On the impact of multiple source code representations on software engineering tasks - An empirical study. J. Syst. Softw. 210, 111941, URL https://api.semanticscholar.org/CorpusID:257767300.

Tao, Yida, Dang, Yingnong, Xie, Tao, Zhang, Dongmei, Kim, Sunghun, 2012. How do software engineers understand code changes? An exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.

Tao, Wei, Wang, Yanlin, Shi, Ensheng, Du, Lun, Han, Shi, Zhang, Hongyu, Zhang, Dongmei, Zhang, Wenqiang, 2021. On the evaluation of commit message generation models: an experimental study. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 126–136.

The Conventional Commits Specification, 2023. Conventional commits. https://www.conventionalcommits.org/. (Accessed 1 May 2023).

Tian, Haoye, Lu, Weiqi, Li, Tsz On, Tang, Xunzhu, Cheung, Shing-Chi, Klein, Jacques, Bissyandé, Tegawendé F., 2023. Is ChatGPT the ultimate programming assistant–how far is it? arXiv preprint arXiv:2304.11938.

Tian, Yingchen, Zhang, Yuxia, Stol, Klaas-Jan, Jiang, Lin, Liu, Hui, 2022. What makes a good commit message? In: Proceedings of the 44th International Conference on Software Engineering. pp. 2389–2401.

Vagavolu, Dheeraj, Swarna, Karthik Chandra, Chimalakonda, Sridhar, 2021. A mocktail of source code representations. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1296–1300, URL https://api.semanticscholar.org/CorpusID:235490356.

van Hal, S.R.P., Post, M., Wendel, K., 2019. Generating commit messages from git diffs. arXiv:1911.11690.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N., Kaiser, Łukasz, Polosukhin, Illia, 2017. Attention is all you need. In: Guyon, I., Luxburg, U. Von, Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (Eds.), In: Advances in Neural Information Processing Systems, vol. 30, Curran Associates, Inc., URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Wang, Yue, Wang, Weishi, Joty, Shafiq, Hoi, Steven C.H., 2021a. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, Bei, Yan, Meng, Liu, Zhongxin, Xu, Ling, Xia, Xin, Zhang, Xiaohong, Yang, Dan, 2021b. Quality assurance for automated commit message generation. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 260–271. http://dx.doi.org/10.1109/SANER50967.2021.00032.

Wei, Jason, Wang, Xuezhi, Schuurmans, Dale, Bosma, Maarten, Xia, Fei, Chi, Ed, Le, Quoc V, Zhou, Denny, et al., 2022. Chain-of-thought prompting elicits reasoning in large language models. Adv. Neural Inf. Process. Syst. 35, 24824–24837.

White, Jules, Fu, Quchen, Hays, Sam, Sandborn, Michael, Olea, Carlos, Gilbert, Henry, Elnashar, Ashraf, Spencer-Smith, Jesse, Schmidt, Douglas C., 2023. A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv:2302.11382.

Wikipedia Contributors, 2024. GPT-4. URL https://en.wikipedia.org/wiki/GPT-4.

Wolf, Thomas, Debut, Lysandre, Sanh, Victor, Chaumond, Julien, Delangue, Clement, Moi, Anthony, Cistac, Pierric, Rault, Tim, Louf, Rémi, Funtowicz, Morgan, Davison, Joe, Shleifer, Sam, von Platen, Patrick, Ma, Clara, Jernite, Yacine, Plu, Julien, Xu, Canwen, Scao, Teven Le, Gugger, Sylvain, Drame, Mariama, Lhoest, Quentin, Rush, Alexander M., 2020. HuggingFace's transformers: State-of-the-art natural language processing. arXiv:1910.03771.

Xiaomeng, Wang, Tao, Zhang, Runpu, Wu, Wei, Xin, Changyu, Hou, 2018. CPGVA: Code property graph based vulnerability analysis by deep learning. In: 2018 10th International Conference on Advanced Infocomm Technology. ICAIT, pp. 184–188. http://dx.doi.org/10.1109/ICAIT.2018.8686548.

Xu, Shengbin, Yao, Yuan, Xu, Feng, Gu, Tianxiao, Tong, Hanghang, Lu, Jian, Commit message generation for source code changes. IJCAI http://dx.doi.org/10.24963/ijcai.2019/552, URL https://par.nsf.gov/biblio/10159294.

Yamaguchi, Fabian, Golde, Nico, Arp, Daniel, Rieck, Konrad, 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Yamauchi, Kenji, Yang, Jiachen, Hotta, Keisuke, Higo, Yoshiki, Kusumoto, Shinji, 2014. Clustering commits for understanding the intents of implementation. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 406–410. http://dx.doi.org/10.1109/ICSME.2014.63.

Yan, Meng, Fu, Ying, Zhang, Xiaohong, Yang, Dan, Xu, Ling, Kymer, Jeffrey D., 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. J. Syst. Softw. 113, 296–308.

Yin, Pengcheng, Neubig, Graham, Allamanis, Miltiadis, Brockschmidt, Marc, Gaunt, Alexander L., 2019. Learning to represent edits. arXiv:1810.13337 [cs.LG], URL https://arxiv.org/abs/1810.13337.

Zhang, Zaixing, Liu, Liang, Chang, Jianming, Wang, Lulu, Liao, Li, 2023. Commit classification via diff-code GCN based on system dependency graph. In: 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security. QRS, IEEE, pp. 476–487.

Zhang, Susan, Roller, Stephen, Goyal, Naman, Artetxe, Mikel, Chen, Moya, Chen, Shuohui, Dewan, Christopher, Diab, Mona, Li, Xian, Lin, Xi Victoria, Mihaylov, Todor, Ott, Myle, Shleifer, Sam, Shuster, Kurt, Simig, Daniel, Koura, Punit Singh, Sridhar, Anjali, Wang, Tianlu, Zettlemoyer, Luke, 2022. OPT: Open pre-trained transformer language models. arXiv:2205.01068.

Zhou, Yaqin, Liu, Shangqing, Siow, Jingkai, Du, Xiaoning, Liu, Yang, 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.

**Abhinav Reddy Mandli** is a proficient professional currently pursuing a bachelor's degree at Dalhousie University. He has worked as a research engineer at Pythonic AI and as an ML engineer at Reel Data AI, focusing on the field of AI. His research interests lie in AI and its subdomains, specifically natural language processing (NLP) and computer vision. With hands-on experience and a strong educational foundation, Abhinav brings valuable insights and skills to his work, contributing to successful project outcomes and innovative solutions in the AI domain.

**Saurabhsingh Rajput** is a Ph.D. student at Dalhousie University, working towards his thesis in the discipline of Green AI and Sustainable Software Engineering. His research focuses on developing innovative techniques to enhance the energy and compute efficiency of deep learning systems, aiming to contribute to the creation of greener AI. He has published high-quality scholarly articles in top-tier software engineering venues such as ACM TOSEM, IEEE GREENS Workshop ICSA, and ACM MSR. He has also served as the Junior PC member for MSR'24 and been a subreviewer for conferences like ICSE, ASE, FSE, SANER, SCAM, and ICPC. Prior to his Ph.D., Saurabhsingh earned his Bachelor's in Computer Science and Engineering from the Visvesvaraya National Institute of Technology, India. He also worked with Fidelity Investments, gaining valuable experience and insight through his participation in the development of various enterprise-level applications and research projects.

**Tushar Sharma** is a tenure-track assistant professor at Dalhousie University, Canada. He leads the Software Maintenance and Analytics Research Team (SMART) lab in the Faculty of Computer Science. Topics related to software engineering, sustainable artificial intelligence, and machine learning for software engineering (ML4SE) define his career interests. He earned a Ph.D. from the Athens University of Economics and Business, Athens, Greece, specializing in software engineering in 2019. He has ten years of industry experience, mainly with Siemens Research, USA and India. He was the principal investigator for the MINDSIGHT team of the DARPA AMP program, consisting of researchers from Siemens, JHU/APL, BAE Systems, and UC Irvine. He co-authored Refactoring for Software Design Smells: Managing Technical Debt and two Oracle Java certification books. He founded Designite, offering code quality assessment tools that many practitioners and researchers use worldwide. He is an IEEE Senior Member.