# Classification of Software Developer Contributions

MASTER THESIS

**Ediz Kocak**

Submitted on 25 August 2026


Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Thomas Wolter M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.


Friedrich-Alexander-Universität
Faculty of Engineering

# Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

---

Erlangen, 25 August 2026

# License



---

Erlangen, 25 August 2026

# Abstract

Software repositories contain large amounts of information about the evolution of a software project, much of which is documented through commits in version control systems such as Git. However, commit messages are frequently written in an inconsistent or incomplete way, making it difficult to analyse contributions systematically. This thesis investigates the automatic classification of software developer contributions based on GitHub commit data using deep learning methods.

A prototype is developed that automatically constructs labelled training datasets from repositories that adopt the Conventional Commits specification, extracting the commit type label directly from the commit message and using the raw code diff as the input signal for classification. The same pipeline is applied to a complementary task of classifying GitHub issues into five categories based on their labels. Both datasets are used to fine-tune a DistilBERT model, which is evaluated on a held-out validation split using accuracy and macro-averaged $F_1$ score.

The commit classifier achieves 71.3% accuracy and a macro-$F_1$ of 0.36 on a twelve-class taxonomy derived from the Conventional Commits specification. The issue classifier achieves 59.1% accuracy and a macro-$F_1$ of 0.32 on a five-class taxonomy. The results show that classification is feasible for frequent and semantically distinct classes such as `feat` and `fix`, while minority classes such as `perf` and `revert` remain difficult to classify reliably due to insufficient training data and semantic overlap with adjacent classes.

The findings demonstrate that transformer-based deep learning is a promising approach for commit classification, and that the Conventional Commits specification provides a scalable, annotation-free labelling strategy for dataset construction. At the same time, the pronounced gap between accuracy and macro-$F_1$ highlights class imbalance as the primary challenge to be addressed in future work.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Modern software development works on software systems which are constantly extended, corrected, refactored, and maintained. A large part of this work is documented through commits in version control systems such as Git and on platforms such as GitHub. These commits represent individual software developer contributions and therefore provide an important source of information about the evolution of a software project.

At the same time, commit data is often difficult to analyse in a systematic way. Although commit messages may contain useful information about the nature of a change, they are frequently written in an inconsistent or incomplete way. In many cases, commits are not explicitly labeled according to the type of contribution they represent — for example bug fixing, feature implementation, refactoring, or documentation changes. As a consequence, software repositories contain large amounts of information in a weakly structured form. Sy One approach to improving the structure of commit messages is the Conventional Commits specification, which defines a standardised format for describing commits. By using prefixes such as `feat`, `fix`, or `docs`, this specification makes the intention of a commit more explicit. However, in practice, such conventions are not applied consistently across projects and developers. As a result, a large number of commits remain unlabeled or only implicitly described. This problem is becoming increasingly relevant as AI-generated code grows more common in software development, potentially leading to a growing number of commits whose messages are still not always clearly written or properly labeled.

This creates the need for automated methods that are able to infer the type of a software developer contribution from the available commit information. Such methods could help to improve the organisation and analysis of software repositories without relying on manual labeling. In this context, machine learning and, in particular, deep learning approaches appear promising, since they can learn patterns from textual and contextual information and may therefore support the classification of commits even when commit messages are not fully standardised.

Against this background, this thesis examines the classification of software de-

veloper contributions based on commit data. The focus lies on the question of to what extent deep learning methods can be used to assign commits to meaningful contribution classes. More specifically, the thesis addresses the following research questions:

- **RQ1:** Is it possible to classify GitHub commits into meaningful contribution types using deep learning approaches, and to what extent are such approaches suitable for this task?
- **RQ2:** Which contribution classes can be identified reliably, and which classes are more difficult to distinguish?

To address these questions, the thesis examines commit classification as a concrete software engineering task based on real repository data from GitHub, investigates whether deep learning methods are suitable for assigning commits to predefined contribution classes, and analyses which classes can be reliably identified and which pose particular challenges.

The scope of this work is limited to the classification of software developer contributions based on GitHub commits. The analysis is restricted to the information available in the selected commit data and the corresponding class definitions used in this work. The thesis does not aim to provide a general evaluation of all machine learning methods for commit classification, nor does it attempt to define a universal standard for labeling software developer contributions.

The remainder of this thesis is structured as follows. Chapter 2 reviews the related work and provides the theoretical background relevant to commit classification and software repository analysis. Chapter 3 describes the definition of contribution classes and the requirements of the system. Chapter 4 presents the architecture, design, and implementation of the prototype, including the dataset construction pipeline and the deep learning models. Chapter 5 reports the results of the evaluation and discusses them with regard to the research questions. Finally, Chapter 6 summarises the main findings of the thesis, discusses limitations, and gives an outlook on possible future work.

# 2 Literature Review

This chapter reviews the body of research relevant to this thesis. It begins with the foundational work on commit classification as a software engineering task, then covers the evolution from classical machine learning towards deep learning and transformer-based approaches, discusses the use of code diffs as input signals, and finally surveys related work on issue classification and on the Conventional Commits specification as a labelling mechanism.

## 2.1 Commit Classification as a Software Engineering Task

The automatic classification of software commits has been studied for more than two decades. Early work focused on the well-known taxonomy of software maintenance activities introduced by Swanson (Swanson, 1976), which distinguishes *corrective*, *adaptive*, and *perfective* changes. Mockus and Votta (Mockus & Votta, 2000) operationalised this taxonomy for version control data, providing one of the earliest empirical studies on classifying code changes from commit messages using keyword based heuristics.

Hindle et al. (Hindle et al., 2008) conducted a large scale study on commit categorisation in open source projects, which introduced a manual annotation scheme and demonstrated that automated approaches based on commit message text can achieve reasonable agreement with human annotators. Their work established commit messages as a viable signal for classification and highlighted the challenge of inconsistent developer writing styles.

A substantial step towards scalable, labelled datasets was taken by Levin and Yehudai (Levin & Yehudai, 2017), who manually annotated 1 151 commits from eleven open-source projects according to the corrective/adaptive/perfective taxonomy. They further showed that incorporating source code change features alongside commit message text brings noticeable accuracy improvements over the baseline of commit messages only, marking the way for the diff-based input strategy employed in this thesis.

These early contributions demonstrated the general potential of commit classification and identified the key challenges of label scarcity and message inconsistency. They motivated the development of richer input representations and larger datasets.

## 2.2 Classical Machine Learning Approaches

Before large pre-trained language models, commit classification relied primarily on feature engineering combined with classical machine learning algorithms. Gharbi et al. (Gharbi et al., 2019) applied logistic regression and support vector machines to commit messages, using bag-of-words and TF-IDF representations as features. Their work demonstrated that even simple linear models can separate maintenance categories with moderate accuracy when the training set is sufficiently large.

Hönel et al. (Hönel et al., 2019) investigated the importance of different feature groups — commit message tokens, file-level metrics, and temporal patterns — for commit classification. They concluded that no single feature type is sufficient on its own and that fusion of heterogeneous signals is beneficial. This finding motivates the use of code diffs as a complementary input in this thesis: the diff encodes structural information about a change that is not necessarily captured by the commit message.

While classical approaches proved effective for the three-class maintenance taxonomy, they scale poorly to finer grained label sets and require a lot of manual feature engineering. These limitations motivated the shift towards representation learning based approaches described in the following sections.

## 2.3 Deep Learning and Transformer-Based Approaches

The introduction of transformer-based language models fundamentally changed the landscape of text classification tasks, including commit classification. Devlin et al. (Devlin et al., 2019) proposed BERT (Bidirectional Encoder Representations from Transformers), a deeply bidirectional model pre-trained on large text corpora using masked language modelling and next-sentence prediction. Fine-tuning BERT on downstream classification tasks consistently achieved state-of-the-art results across a wide range of NLP benchmarks, motivating its adoption in software engineering research.

Ghadhab et al. (Ghadhab et al., 2021) were among the first to apply BERT directly to commit classification. They fine-tuned BERT on a balanced dataset of 1 793 labelled commits and augmented the model's input with fine-grained source

code change features extracted by code change distillers. Their results showed that the transformer-based model outperformed previous classical approaches on the three-class maintenance taxonomy and that code change features provided further complementary signal on top of the commit message. This work directly informs the design choice in this thesis of using the full code diff as the primary input to a transformer model.

Ni et al. (Lee & Chieu, 2021) proposed a co-training framework for commit classification that leverages both the natural language and the programming language views of a commit in a semi-supervised setting. Their approach addresses the persistent problem of label scarcity in commit datasets and demonstrates that the two representations are complementary.

For the specific choice of DistilBERT in this thesis, Sanh et al. (Sanh et al., 2019) demonstrated that a six-layer student model trained by knowledge distillation retains approximately 97% of BERT's performance on GLUE benchmarks while being 40% smaller and 60% faster at inference. This efficiency advantage is particularly relevant when processing long code diffs that push against the 512-token context limit of BERT-family models, as it allows more training iterations within a fixed compute budget.

More recently, large language models have also been applied to commit classification in a few-shot or in-context learning setting. Heričko et al. (Heričko et al., 2024) evaluated GPT-family models on the maintenance activity classification task and found that, while they are still competitive with fine-tuned models in zero-shot conditions, in-context learning approaches remain inferior to specific task fine-tuning when labelled data is available. This observation supports the fine-tuning strategy adopted in this thesis.

## 2.4 Code Diffs as Input Representations

The choice of input representation has a significant impact on commit classification performance. The earliest approaches relied exclusively on commit message text, which is cheap to process but frequently incomplete or ambiguous. Subsequent work explored richer representations.

Sabetta and Bezzi (Sabetta & Bezzi, 2018) treated source code as a natural-language-like document and applied TF-IDF weighting to the raw diff text in order to identify security-relevant commits. Their approach demonstrated that unstructured diff text carries predictive signal even without any code-specific parsing, provided that the vocabulary of changed lines is informative for the target class.

Ni et al. (Lee & Chieu, 2021) jointly encoded the commit message and a structured

representation of the diff in a dual-channel architecture. Their ablation study confirmed that the diff channel contributes significantly to classification accuracy, especially for classes where the commit message is short or uninformative.

In the context of this thesis, the raw unified diff(produced by `git diff <sha>^ <sha>`) is used as the sole input text. This design decision follows the observation of Ghadhab et al. (Ghadhab et al., 2021) and Levin and Yehudai (Levin & Yehudai, 2017) that code change content is particularly diagnostic for classes such as `style`, `test`, and `refactor`, where commit messages are often sparse.

## 2.5 The Conventional Commits Specification

The Conventional Commits specification (Conventional Commits, 2024) defines a lightweight, machine-readable format for commit messages:

```
<type>[optional scope][optional !]: <description>
```

where `type` is one of a predefined set of keywords (`feat`, `fix`, `docs`, etc.). Repositories that adopt this convention provide ground-truth labels for every commit, enabling the automatic construction of training datasets without any manual annotation. This is a key design decision in this thesis.

The specification itself builds on a long tradition of commit message conventions in open-source software. Hindle et al. (Hindle et al., 2008) and subsequent work repeatedly noted that informal conventions, such as prefixing bug-fix messages with the word *fix*, were already exploited by keyword-based classifiers. Conventional Commits formalises and standardises this practice, making the label extraction step fully deterministic and scalable.

Heričko et al. (Heričko et al., 2024) also note that repositories adopting Conventional Commits provide a natural evaluation benchmark for automated classifiers: a model trained on unlabelled commits can be evaluated by applying it to a held-out subset of a conventional-commit repository and comparing predictions against the ground truth derived from the specification. This approach is adopted in the evaluation design described in Chapter 5.

Despite its advantages, the Conventional Commits format is not universally adopted. Tian et al. (Tian et al., 2022) analysed a large sample of GitHub repositories and found that only a minority consistently apply a structured commit message format. This motivates the need for classifiers that can generalise beyond conventional-commit repositories, which is one of the long-term goals this thesis contributes to.

## 2.6 Issue Classification

Classifying GitHub issues into categories such as bug reports, feature requests, and questions is a closely related task that has received substantial attention in the mining software repositories community.

Kallis et al. (Kallis et al., 2019) introduced the Ticket Tagger system, which applies fastText embeddings to classify GitHub issue titles into three classes (bug, question, enhancement) and demonstrated high accuracy on a large corpus of labelled issues. Their work established the title as the most informative field for issue classification, a finding reflected in the structured `TITLE:` / `BODY:` input format used in this thesis.

Izadi et al. (Izadi et al., 2022) extended issue classification to a larger label space and employed CodeBERT, a transformer pre-trained on both natural language and programming language, for label prediction. They showed that transformer-based models significantly outperform bag-of-words and fastText baselines, particularly for fine-grained label sets where lexical overlap between classes is limited.

The multi-label nature of GitHub issues — a single issue may carry labels such as *bug* and *help wanted* simultaneously — poses an additional challenge. In this thesis, issues are reduced to a single dominant label through priority-based normalisation, following the pragmatic approach of Kallis et al. (Kallis et al., 2019), so that standard single-label classification metrics apply.

## 2.7 Mining Software Repositories

This thesis falls within the broader field of Mining Software Repositories (MSR), which studies how data extracted from version control systems, issue trackers, and code review tools can be used to support software engineering research and practice (Hassan, 2008).

A central challenge in MSR is the construction of labelled datasets at scale. Kalliamvakou et al. (Kalliamvakou et al., 2014) conducted a large-scale analysis of GitHub repositories and identified a number of biases and confounds — including personal and experimental repositories, inactive projects, and inconsistent use of collaboration features — that must be considered when sampling from GitHub. The repository selection criteria in this thesis (Section 4.3.5) mitigate these risks by restricting the corpus to established, actively maintained open-source projects with a documented history of adopting the Conventional Commits format.

The use of transformer-based models for software artefact classification has become widespread in MSR research since the introduction of CodeBERT (Feng et al., 2020), a model pre-trained on both natural language and programming language

corpora using replaced token detection. While CodeBERT and its successors are optimised for code understanding tasks such as code search and code summarisation, the present thesis uses DistilBERT — a pure natural-language model — because the primary input is the diff text, which, despite containing code tokens, is more similar in structure to natural language text (with `+`/`-` prefixes and line-by-line prose) than to compilable source code.

## 2.8 Summary and Research Gap

The literature reviewed in this chapter leads to several observations relevant to this thesis. First, commit classification has been studied extensively for the three-class maintenance taxonomy, but fine-grained classification into the twelve Conventional Commits types has received substantially less attention. Second, transformer-based models consistently outperform classical machine learning approaches on commit and issue classification tasks, validating the use of DistilBERT in this work. Third, code diff content provides complementary signal to commit message text, particularly for classes where message quality is low. Fourth, the Conventional Commits specification offers a scalable, annotation-free labelling strategy that has not yet been widely exploited as a training data source in the academic literature.

This thesis addresses this gap by constructing a fine-grained, automatically labelled training dataset from Conventional Commits repositories and evaluating whether a DistilBERT model which is fine-tuned on raw code diffs can classify commits into twelve contribution types at a level of accuracy that is practically useful.

# 3 Requirements

## 3.1 Formal Problem Statement

Let $\mathcal{C}$ denote the universe of all software contributions. Each contribution $c \in \mathcal{C}$ is associated with a set of observable textual attributes $\mathbf{x}(c) \in \mathcal{X}$, and a latent *type label* $y(c) \in \mathcal{Y}$, where $\mathcal{Y}$ is a finite, mutually exclusive set of contribution types.

The classification problem is to learn a function

$$f \colon \mathcal{X} \longrightarrow \mathcal{Y}$$

such that $f\big(\mathbf{x}(c)\big) = y(c)$ for as many contributions as possible, as measured by accuracy and macro-averaged $F_1$ score on a held-out validation set.

Two independent instantiations of this problem are considered: one for Git *commits* and one for GitHub *issues*.

## 3.2 Taxonomy Definition

### 3.2.1 Commit Types

The commit label set $\mathcal{Y}_{\text{commit}}$ is derived directly from the Conventional Commits specification and consists of twelve mutually exclusive classes:

- **Feature** (`feat`): Adds new production functionality.
- **Bug Fix** (`fix`): Corrects incorrect behaviour.
- **Refactoring** (`refactor`): Restructures code without changing external behaviour.
- **Documentation** (`docs`): Modifies only documentation artefacts (e.g. README, API docs, inline comments).
- **Test** (`test`): Adds or modifies test code exclusively.

- **Build** (`build`): Changes to build scripts or dependency management.
- **CI** (`ci`): Changes to CI/CD pipeline configuration.
- **Chore** (`chore`): Misc. maintenance not covered by the above classes.
- **Style** (`style`): Formatting or whitespace changes that do not affect logic.
- **Performance** (`perf`): Changes that improve execution speed or resource usage.
- **Revert** (`revert`): Reverts a previous commit.
- **Unknown** (`unknown`): Fallback for commit types that do not match any of the above eleven classes.

The `unknown` class acts as a default: any raw label not in the eleven named classes is remapped to `unknown` at loading time:

$$y_{\text{mapped}}(c) = \begin{cases} y_{\text{raw}}(c) & \text{if } y_{\text{raw}}(c) \in \mathcal{Y}_{\text{commit}} \\ \texttt{unknown} & \text{otherwise} \end{cases}$$

Commits that do not follow the Conventional Commits format at all are excluded from the training dataset entirely.

### 3.2.2 Issue Types

The issue label set $\mathcal{Y}_{\text{issue}}$ consists of five categories derived from common GitHub label naming conventions:

- **Bug** (`bug`): Describes unintended or broken behaviour.
- **Enhancement** (`enhancement`): Proposes new functionality or an improvement.
- **Documentation** (`documentation`): Requests or corrects documentation.
- **Question** (`question`): Seeks information or clarification.
- **Performance** (`performance`): Describes a performance-related problem or request.

Issues whose GitHub labels cannot be unambiguously mapped to one of these five classes are discarded during dataset construction. Pull requests, which the GitHub API returns alongside issues, are also filtered out.

## 3.3 Evaluation Criteria

Let $\mathrm{TP}_k$, $\mathrm{FP}_k$, and $\mathrm{FN}_k$ denote the true positives, false positives, and false negatives for class $k \in \mathcal{Y}$ on the validation set. The per-class metrics are:

$$\mathrm{Precision}_k = \frac{\mathrm{TP}_k}{\mathrm{TP}_k + \mathrm{FP}_k}$$

$$\mathrm{Recall}_k = \frac{\mathrm{TP}_k}{\mathrm{TP}_k + \mathrm{FN}_k}$$

$$F_{1,k} = \frac{2 \cdot \mathrm{Precision}_k \cdot \mathrm{Recall}_k}{\mathrm{Precision}_k + \mathrm{Recall}_k}$$

The macro-averaged $F_1$ score across all $|\mathcal{Y}|$ classes is used as the primary scalar metric:[1]

$$\overline{F_1} = \frac{1}{|\mathcal{Y}|} \sum_{k \in \mathcal{Y}} F_{1,k}$$

Overall accuracy is reported alongside macro-$F_1$ as a complementary measure of aggregate correctness:

$$\mathrm{Accuracy} = \frac{\sum_k \mathrm{TP}_k}{N_{\mathrm{val}}}$$

where $N_{\mathrm{val}}$ is the total number of validation samples.

## 3.4 Functional Requirements

**FR1** The system shall automatically clone a Git repository and extract all non-merge commits up to a configurable limit.

**FR2** The system shall parse each commit message according to the Conventional Commits specification and extract the type label.

**FR3** For each labelled commit, the system shall extract the full code diff and store it as the input text.

[1] Macro-averaging weights each *class* equally, penalising poor performance on rare classes (e.g. `revert` or `perf`). Micro-averaging would weight each *sample* equally and thus reflect the performance on the dominant class only.

**FR4** The system shall fetch issues from a GitHub repository via the REST API and normalise their labels to the five-class issue taxonomy.

**FR5** The system shall merge datasets from multiple repositories into a single unified dataset.

**FR6** The system shall fine-tune a DistilBERT model on each dataset and report accuracy and macro-$F_1$ on the validation split.

**FR7** The trained model shall be saved to disk in a format that can be reloaded for inference without access to the training data.

## 3.5 Non-Functional Requirements

**NFR1** **Reproducibility:** The train/validation split shall use a fixed random seed so that evaluation results are identical across runs.

**NFR2** **Robustness:** The dataset builders shall handle missing diffs, empty issue bodies, and encoding errors without crashing.

**NFR3** **Extensibility:** New repositories shall be addable to the dataset by re-running the builder with a different output directory and merging the result.

# 4 Architecture, Design, and Implementation

This chapter describes the overall architecture of the prototype and explains each component in detail, covering the motivation behind each design decision, the technical implementation, and all relevant parameters.

## 4.1 System Overview

The prototype consists of three components:

1. **Dataset Builders** — scripts that automatically construct labelled datasets from public GitHub repositories (Sections 4.3 and 4.4).
2. **Dataset Merger** — a utility that combines datasets from multiple repositories into a single unified training corpus (Section 4.5).
3. **Classifier** — a fine-tuning pipeline that trains a DistilBERT model on the merged dataset and evaluates it on a held-out validation split (Section 4.7).

All components share a common data format and are independently runnable.

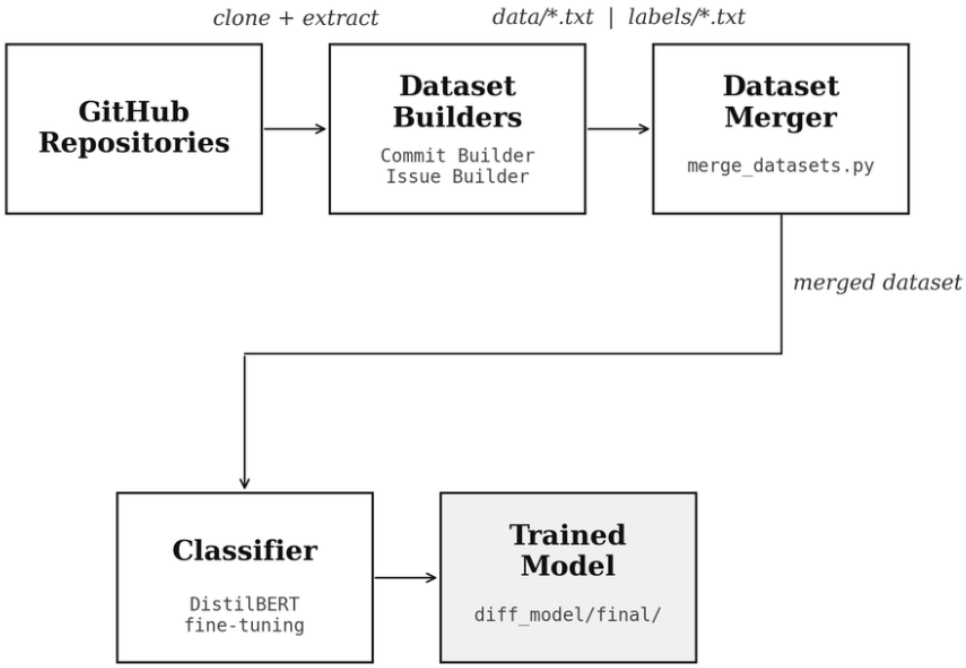


**Figure 4.1:** High-level pipeline: dataset construction, merging, and model training. Shaded boxes represent software components; arrows represent data flows.

## 4.2 Data Format

All datasets share a common directory layout:

```
datasets/<output_dir>/
    data/
        00000.txt
        00001.txt
        ...
    labels/
        00000.txt
        00001.txt
        ...
    metadata.json
```

Each `data/XXXXX.txt` file contains the full input text for one sample; the corresponding `labels/XXXXX.txt` file contains a single class label as a plain string. Files are named with five-digit indices to allow up to 99,999 samples per dataset. The `metadata.json` file stores build-run statistics (repository URL, number of

samples, label distribution).

This simple format was chosen because it maps directly to PyTorch's `Dataset` abstraction via a `pathlib` directory scan.

## 4.3 Commit Dataset Builder

### 4.3.1 Motivation

A core challenge in commit classification is the absence of large, consistently labelled training datasets. Manual annotation is expensive and subject to mistakes and disagreement. The Conventional Commits specification offers an elegant solution. Repositories that adopt it encode the ground-truth label directly in the commit message subject line, making label extraction fully automatic.

The commit dataset builder makes use of this property. It clones a target repository, iterates over its commit history, parses each commit message for a Conventional Commits type, and if a valid type is found extracts the full code diff as the input text. Commits that do not follow the convention are silently skipped, ensuring that the training data contains only confidently labelled examples. Roughly 5% of commits in a typical conventional-commit repository do not follow this format. These are excluded without replacement.

### 4.3.2 Git Commits: Structure and Format

A Git commit is the fundamental unit of change in a version control repository (Chacon & Straub, 2014). Each commit records a snapshot of the project at a specific point in time and is identified by a unique SHA-1 hash. In addition to the actual file changes, every commit carries a set of metadata fields:

- **Author and timestamp:** the name, email address, and date of the developer who authored the change.
- **Committer and timestamp:** the identity of the person who applied the commit to the repository, which may differ from the author in workflows that use patch submission or rebasing.
- **Parent reference(s):** one or more SHA-1 hashes pointing to the preceding commit(s), forming the directed acyclic graph that constitutes the project history. Merge commits carry two parent references.
- **Commit message:** a free-text description of the change, conventionally composed of a short subject line (typically under 72 characters) and an optional longer body separated by a blank line.

- **Diff (changeset):** the set of line-level additions and deletions across all modified files, represented in the unified diff format.

A concrete example of a commit as returned by `git log -p` is shown below. The `-p` flag instructs Git to include the full diff alongside the commit metadata:

```
commit a3f8c21d9e4b7f0123456789abcdef0123456789
Author: Jane Developer <jane@example.com>
Date:   Mon Jul 07 14:32:10 2025 +0200

    feat(parser): add support for nested scopes

    The parser previously failed to handle expressions nested more
    than two levels deep. This commit extends the recursive descent
    logic to support arbitrary nesting depth.

diff --git a/src/parser.py b/src/parser.py
index 7d3a1f0..c82e4b9 100644
--- a/src/parser.py
+++ b/src/parser.py
@@ -14,10 +14,20 @@ class Parser:
     def __init__(self, tokens):
         self.tokens = tokens
         self.pos = 0
+        self.depth = 0
+        self.max_depth = 64

     def parse_expression(self):
-        return self._parse_binary()
+        if self.depth > self.max_depth:
+            raise ParseError("Maximum nesting depth exceeded")
+        self.depth += 1
+        result = self._parse_binary()
+        self.depth -= 1
+        return result

     def _parse_binary(self):
         left = self._parse_unary()
         while self.current_token() in ('+', '-', '*', '/'):
             op = self.consume()
-            right = self._parse_unary()
+            right = self.parse_expression()
             left = BinaryNode(op, left, right)
         return left
```

The commit metadata block at the top provides the SHA-1 hash, author identity, timestamp, and the full commit message including its body. The subject line `feat(parser): add support for nested scopes` follows the Conventional Commits format: the type token `feat` identifies this as a feature addition, the optional scope `(parser)` narrows the affected component, and the remainder is a concise imperative description of the change. The body provides additional context that is useful for human readers but is not used as input to the classifier in this thesis.

The diff section below the message shows the exact changes made to `src/parser.py`. Lines prefixed with `+` were added and lines prefixed with `-` were removed. The `@@` hunk header indicates the line range affected in the original and modified file. In this example, the developer introduced a `depth` counter and a `max_depth` guard to prevent unbounded recursion, and changed the recursive call from `_parse_unary()` to `parse_expression()` to enable arbitrary nesting.

It is this diff — rather than the commit message — that serves as the primary input signal for the classifier described in Section 4.3.3. This design choice is motivated by the observation that the diff encodes *what* was changed at the code level, independently of how the developer chose to describe it in natural language.

### 4.3.3 Input Text: Code Diff

The input text for each commit sample is the output of:

```
git diff <sha>^ <sha> --no-color
```

This produces the unified diff between the parent commit and the target commit, showing all added and removed lines across all changed files. The diff is stored verbatim in the data file, prefixed with a `CODE DIFF:` header:

```
CODE DIFF:
diff --git a/src/parser.rs b/src/parser.rs
index 3f1a2b0..8e4c1d9 100644
--- a/src/parser.rs
+++ b/src/parser.rs
@@ -10,6 +10,8 @@ fn parse() {
+    println!("parsed successfully");
...
```

Using the code diff rather than the commit message alone provides richer semantic content: the model can observe *what* changed, not only *how the developer described* the change. This is especially valuable for classes such as `style` or `test`, where the message may be ambiguous but the diff content is diagnostic. If no diff is available (e.g. for an initial commit without a parent), the placeholder `(No code`

`changes or diff unavailable)` is written, and the sample is still retained with its label.

### 4.3.4 Label Extraction

The commit subject line is matched against the regular expression:

```
^(\w+)(?:\([^)\]+\))?!?:\s*(.+)$
```

If the captured type token (group 1, lowercased) appears in the set of recognised conventional types (`feat`, `fix`, `docs`, `style`, `refactor`, `perf`, `test`, `build`, `ci`, `chore`, `revert`), it is written as the label. Otherwise the commit is skipped entirely. Merge commits are excluded from the commit log via the `--no-merges` flag passed to `git log`.

### 4.3.5 Repositories Used

Commit training data were collected from the following open-source repositories, selected because they consistently apply the Conventional Commits format and span multiple programming languages:

**Table 4.1:** Repositories used for commit dataset construction.

| Repository | Language | URL |
|---|---|---|
| angular/angular | TypeScript | github.com/angular/angular |
| commitizen-tools/commitizen | Python | "/commitizen-tools/commitizen |
| tomasbjerre/git-changelog-lib | Java | "/tomasbjerre/git-changelog-lib |
| orhun/git-cliff | Rust | "/orhun/git-cliff |

The choice of multiple languages ensures that the classifier is not biased towards the syntax or conventions of a single language ecosystem.

## 4.4 Issue Dataset Builder

### 4.4.1 Motivation

While commits document the actual code changes made to a repository, GitHub issues capture the reported problems, feature requests, and questions that motivate those changes. Classifying issues therefore provides a complementary perspective on software developer contributions: where commit classification answers the question of *what* was changed, issue classification addresses *why* a change was requested or needed.

Including an issue classifier in this thesis serves two purposes. First, it demonstrates that the same general pipeline (automatic dataset construction, transformer-based

fine-tuning, and held-out evaluation) can be applied to a different type of software repository artifact with minimal modification. Second, it allows the classification approach to be evaluated on a different input modality: issues are written entirely in natural language, without any code content, which makes them a useful counterpoint to the diff-based commit classifier.

The labelling challenge for issues differs from that of commits. Unlike Conventional Commits, there is no universally adopted specification that encodes the issue type in a machine-readable field. Instead, GitHub labels are used as a proxy for the ground-truth class. These labels are applied manually by repository maintainers and vary considerably in naming convention across projects. The issue dataset builder therefore relies on a normalisation step that maps the heterogeneous space of real-world GitHub labels to a fixed five-class taxonomy, as described in Section 4.4.5.

### 4.4.2 GitHub Issues: Structure and Format

A GitHub issue is the primary mechanism for tracking work items, bug reports, feature requests, and questions within a GitHub repository (Chacon & Straub, 2014). Unlike commits, which are generated automatically by the version control system, issues are created manually by contributors or users and are therefore written in natural language. Each issue consists of the following fields:

- **Title:** a short, mandatory summary of the issue, typically one sentence. The title is the most consistently filled field and has been shown to be the most informative signal for issue classification (Kallis et al., 2019).
- **Body:** an optional longer description providing additional context, reproduction steps, expected versus actual behaviour, or proposed solutions. The body may also contain code snippets, stack traces, or links to related issues.
- **Labels:** one or more tags attached to the issue by maintainers or contributors, used to categorise the issue by type, priority, or affected component. Examples include `bug`, `enhancement`, and `help wanted`. Labels are the primary source of ground-truth class information used in this thesis.
- **State:** either `open` or `closed`, indicating whether the issue has been resolved.
- **Metadata:** author, creation timestamp, assignees, linked pull requests, and milestone information.

A concrete example of a GitHub issue as returned by the GitHub REST API is shown below:

```
{
  "number": 4821,
```

```
  "title": "Parser crashes on deeply nested expressions",
  "body": "When parsing an expression with more than 32 levels of
           nesting, the parser throws a RecursionError instead of
           returning a meaningful error message.\n\n
           Steps to reproduce:\n
           1. Create an expression with 33 nested parentheses.\n
           2. Call parser.parse(expression).\n\n
           Expected: A ParseError with a descriptive message.\n
           Actual:   RecursionError: max recursion depth reached",
  "labels": [
    { "name": "bug" },
    { "name": "parser" }
  ],
  "state": "closed",
  "created_at": "2025-03-12T09:14:33Z",
  "user": { "login": "user123" }
}
```

In this example the issue carries two labels: `bug` and `parser`. The label `bug` maps directly to the `bug` class in the five-class taxonomy defined in Section 3.2.2, while `parser` is a component label that carries no class information and is ignored during normalisation. The title concisely describes the problem, while the body provides reproduction steps and clarifies the expected versus actual behaviour.

For the purposes of this thesis, only the **title** and **body** fields are used as input to the classifier, and only the **labels** field is used to derive the ground-truth class. All other metadata fields are discarded during dataset construction. The label normalisation procedure that maps raw GitHub labels to the five target classes is described in Section 4.4.5.

### 4.4.3 Data Source and Filtering

Issues are fetched from the GitHub REST endpoint:

```
GET https://api.github.com/repos/{owner}/{repo}/issues
    ?state=all&per_page=100&page={n}
```

The response includes both issues and pull requests. Items that contain a `pull_request` field are filtered out immediately. Pagination continues until the configured maximum is reached or the API returns an empty page.

### 4.4.4 Input Text: Title and Body

The input text for each issue sample is formatted as:

```
TITLE:
<issue title>

BODY:
<issue body or "(No body provided)">
```

The structured `TITLE:` / `BODY:` prefix provides the model with positional context, enabling it to up-weight the usually more informative title.

### 4.4.5 Label Normalisation

Each issue may carry multiple GitHub labels. The builder iterates through them and applies exact then partial string matching against a hand-coded mapping table. Representative entries are shown below:

**Table 4.2:** Issue label normalisation mapping (representative entries).

| GitHub label variant | Normalised class |
|---|---|
| `bug`, `type:bug`, `Bug`, `fix` | `bug` |
| `enhancement`, `feature`, `type:feat` | `enhancement` |
| `documentation`, `docs`, `type:docs` | `documentation` |
| `question`, `help wanted`, `type:question` | `question` |
| `performance`, `perf`, `type:perf` | `performance` |

If none of the issue's labels produce a match, the issue is discarded.

## 4.5 Dataset Merger

When training data is sourced from multiple repositories, the datasets are combined using `merge_datasets.py`. The merger copies all `data/*.txt` and `labels/*.txt` files from each source directory into a new `datasets/merged/` directory, while avoiding filename collisions. The merged dataset follows the same directory layout as a single-repository dataset and is passed directly to the classifier.

An analogous script, `merge_issue_datasets.py`, performs the same operation for the issue dataset. It reads from `datasets/issues/`, where each subdirectory corresponds to one repository, and writes all samples to a single `datasets/issues_merged/` directory. File renaming with a global counter ensures that no filename collisions occur across repositories, mirroring the behaviour of the commit merger.

Merging across repositories increases dataset size and improves label diversity. A type such as `perf` that is rare in one repository may be more frequent in another, reducing the risk of a classifier that never predicts minority classes.

## 4.6 Model Architecture

Both classifiers share the same neural network architecture: a **DistilBERT!** (**DistilBERT!**) encoder with a task-specific linear classification head, instantiated via
`AutoModelForSequenceClassification` (Wolf et al., 2020).

### 4.6.1 DistilBERT Encoder

**DistilBERT!** (Sanh et al., 2019) is a transformer encoder with $L = 6$ layers, $H = 768$ hidden dimensions, and $A = 12$ attention heads, totalling approximately 66 million parameters. Each layer computes multi-head self-attention. For a single head $a$ at layer $\ell$, query, key, and value matrices are linear projections of the layer input $\mathbf{X}^{(\ell)} \in \mathbb{R}^{T \times H}$:

$$\mathbf{Q}_a = \mathbf{X}^{(\ell)}\mathbf{W}_a^Q, \quad \mathbf{K}_a = \mathbf{X}^{(\ell)}\mathbf{W}_a^K, \quad \mathbf{V}_a = \mathbf{X}^{(\ell)}\mathbf{W}_a^V$$

The scaled dot-product attention for head $a$ is:

$$\text{Attention}(\mathbf{Q}_a, \mathbf{K}_a, \mathbf{V}_a) = \text{softmax}\left(\frac{\mathbf{Q}_a\mathbf{K}_a^\top}{\sqrt{d_k}}\right)\mathbf{V}_a, \qquad d_k = \frac{H}{A} = 64$$

The $\sqrt{d_k}$ scaling factor prevents the dot products from growing so large that the softmax function saturates into near-zero gradients.[1] All $A$ heads are concatenated and linearly projected:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_A)\,\mathbf{W}^O$$

The hidden state of the `[CLS]` token after the final layer, $\mathbf{h}_{\texttt{[CLS]}} \in \mathbb{R}^{768}$, is the sequence-level representation passed to the classification head.

### 4.6.2 Classification Head

For the final classification step, a linear layer maps the hidden representation produced by the model to a logit vector. The purpose of this classification head is to convert the internal representation of the input into a prediction over the target classes. This is a common and effective approach for multiclass classification in deep learning (Goodfellow et al., 2016).

The corresponding mapping is defined as follows:

[1] Without scaling, for large $d_k$ the dot products tend to have large magnitude, pushing the softmax into regions of very small gradient and slowing learning.

$$\mathbf{z}(c) = \mathbf{W} \cdot \mathbf{h}(c) + \mathbf{b} \in \mathbb{R}^{|\mathcal{Y}|}$$

Here, $\mathbf{h}(c)$ denotes the hidden representation of input $c$ produced by the model. The matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{Y}| \times 768}$ contains the learned weights, and $\mathbf{b} \in \mathbb{R}^{|\mathcal{Y}|}$ is a bias vector. Both are learned during fine-tuning. The set $\mathcal{Y}$ contains all possible class labels, and $|\mathcal{Y}|$ therefore denotes the number of output classes. As a result, the vector $\mathbf{z}(c)$ contains one value for each class.

These values are called logits. They can be interpreted as unnormalised scores that indicate how strongly the model assigns the input to each class. At this stage, the values are not yet probabilities. In particular, they can take any real value and do not sum to 1.

To obtain probabilities, the softmax function is applied:

$$\hat{p}_k(c) = \frac{\exp z_k(c)}{\sum_{j=1}^{|\mathcal{Y}|} \exp z_j(c)}, \quad k \in \mathcal{Y}$$

The softmax function transforms the logits into values between 0 and 1 and ensures that the probabilities over all classes sum to 1 (Goodfellow et al., 2016). This makes the output easier to interpret. A larger logit value leads to a larger probability for the corresponding class.

The predicted label is then determined as

$$\hat{y}(c) = \arg\max_{k \in \mathcal{Y}} \hat{p}_k(c)$$

### 4.6.3 Input Tokenisation

Inputs are tokenised with **DistilBERT!**'s WordPiece tokeniser, truncated to $T_{\max} = 512$ tokens.[2] Shorter texts are padded dynamically per mini-batch by `DataCollatorWithPadding`, reducing unnecessary computation.

[2]Texts longer than 512 tokens are truncated at the right. For commit diffs this means that very large changes lose the tail of the diff.

## 4.7 Training Procedure

### 4.7.1 Dataset Class

The custom `TokenizedTxtFolderDataset` class implements the PyTorch `Dataset` interface. At initialisation time it scans `data/*.txt` and `labels/*.txt` for paired files and reads each label file. The fixed label list for the commit classifier is:

```
['build', 'chore', 'ci', 'docs', 'feat', 'fix',
'perf', 'refactor', 'revert', 'style', 'test', 'unknown']
```

Tokenisation is deferred to `__getitem__`: each text file is read and tokenised on demand, avoiding the need to hold all tokenised sequences in memory simultaneously.

### 4.7.2 Train/Evaluation Split

The merged dataset of $N$ samples is split into training (90%) and evaluation (10%) by shuffling all indices with a fixed seed of 42:

$$N_{\text{train}} = \lfloor 0.9 \cdot N \rfloor, \qquad N_{\text{eval}} = N - N_{\text{train}}$$

Shuffling before splitting is essential because the dataset is ordered by repository — without it the evaluation set would contain only samples from the last repository in the merge order.

### 4.7.3 Loss Function

The model is trained with *categorical cross-entropy loss* (`CrossEntropyLoss` in PyTorch), the standard objective for multi-class classification.[3] For a contribution $c$ with true label $y(c)$, the per-sample loss is:

$$\mathcal{L}(c) = -\log \hat{p}_{y(c)}(c) = -z_{y(c)}(c) + \log \sum_{k=1}^{|\mathcal{Y}|} \exp z_k(c)$$

The batch loss over a mini-batch $\mathcal{B}$ of size $B$ is the mean:

$$\mathcal{L}_{\mathcal{B}} = \frac{1}{B} \sum_{c \in \mathcal{B}} \mathcal{L}(c)$$

[3] PyTorch's `CrossEntropyLoss` fuses log-softmax and negative log-likelihood into one numerically stable operation, avoiding the floating-point underflow from computing $\log(\text{softmax}(\mathbf{z}))$ in two steps.

The gradient with respect to logit $z_k$ is:

$$\frac{\partial \mathcal{L}(c)}{\partial z_k} = \hat{p}_k(c) - \mathbb{1}[k = y(c)]$$

For the true class the gradient is negative (the logit is pushed upward); for all other classes it is positive (those logits are pushed downward).

### 4.7.4 Optimiser: AdamW

For parameter optimisation, AdamW (Loshchilov & Hutter, 2019) is used. This optimiser is also the default optimisation method in the Hugging Face `Trainer`. The choice of AdamW is motivated by the fact that it is a widely established optimiser for training deep neural networks and has shown good performance in a broad range of applications. A main advantage of this optimiser is that it adapts the update step individually for each parameter, which can lead to stable and efficient training behaviour. In addition, the decoupled weight decay can have a positive effect on generalisation.

AdamW is based on the Adam optimiser, but differs from it by using decoupled weight decay. In this case, $L_2$ regularisation is not added to the gradient itself, but applied directly to the parameter values. This allows a clearer separation between the gradient-based update step and the regularisation effect.

At optimisation step $t$, let $\mathbf{g}_t$ denote the gradient of the loss function with respect to the parameter vector $\boldsymbol{\theta}_t$. AdamW keeps moving averages of both the first and the second moment of the gradients. The first moment can be understood as the mean of past gradients, whereas the second moment captures the magnitude of past gradients. They are computed as follows:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\, \mathbf{g}_t$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\, \mathbf{v}_{t-1} + (1 - \beta_2)\, \mathbf{g}_t \odot \mathbf{g}_t$$

Here, $\mathbf{m}_t$ denotes the estimate of the first moment and $\mathbf{v}_t$ the estimate of the second moment. The symbol $\odot$ denotes element-wise multiplication. The parameters $\beta_1$ and $\beta_2$ determine to which extent information from previous update steps is retained.

Since both moment estimates are initialised with zero, they are biased towards zero during the first optimisation steps. Therefore, bias-corrected estimates are calculated as

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$$

Based on these corrected estimates, the parameter update is given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} - \eta \lambda \boldsymbol{\theta}_t$$

The first part of this update corresponds to the adaptive optimisation step of Adam. The update direction is determined by the first moment estimate, while the step size is scaled by the second moment estimate. As a result, different parameters can be updated with different effective step sizes. The constant $\epsilon$ is added for numerical stability in order to avoid division by very small values.

The second part of the update equation represents the decoupled weight decay. Its effect is that the parameter values are reduced slightly in each update step. This acts as a regularisation mechanism and may contribute to improved generalisation behaviour.

In this work, the standard values $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and $\lambda = 0.01$ are used.

### 4.7.5 Training Hyperparameters

**Table 4.3:** Training hyperparameters for both classifiers.

| **Parameter** | **Commit** | **Issue** |
|---|---|---|
| Pre-trained base | `distilbert-base-uncased` | |
| Learning rate $\eta$ | $2 \times 10^{-5}$ | |
| Train batch size | 8 | |
| Eval. batch size | 16 | |
| Max. seq. length | 512 tokens | |
| Eval. strategy | per epoch | |
| Random seed | 42 | |
| Training epochs $E$ | 2 | 20 |

The learning rate follows the standard BERT-family fine-tuning recommendation (Devlin et al., 2019).

### 4.7.6 Model Persistence

After training, model and tokeniser are serialised to `diff_model/final/`:

- `trainer.save_model()` writes weights, configuration, and the label mapping `id2label` = {0:`build`, . . . , 11:`unknown`}.
- `tokenizer.save_pretrained()` writes the WordPiece vocabulary and special-token configuration.

The artefact is self-contained and reloadable via `AutoModelForSequenceClassification.from_pretrained` without training data, satisfying FR7.

# 5 Evaluation

This chapter reports the experimental results obtained on the held-out validation splits of both classifiers, discusses them in relation to the research questions defined in Chapter 1 and the requirements defined in Chapter 3, and analyses threats to validity.

## 5.1 Experimental Setup

Both classifiers were trained and evaluated using the configuration described in Section 4.7.5. The commit classifier was trained on the merged dataset drawn from the four repositories listed in Table 4.1. The issue classifier was trained on issues fetched from selected GitHub repositories. In both cases, 10% of the data was held out as a validation set using the fixed-seed split described in Section 4.7.2.

## 5.2 Results

Table 5.1 reports the top-level evaluation metrics for both classifiers on their respective held-out evaluation splits.

**Table 5.1:** Evaluation results on the held-out 10 % split (final epoch).

| **Classifier** | **Eval. Loss** | **Accuracy** | **Macro-$F_1$** | **Train Loss** |
|---|---|---|---|---|
| Commit (2 epochs) | 1.06 | 71.3 % | 0.36 | 1.54 |
| Issue (20 epochs) | 1.25 | 59.1 % | 0.32 | 2.47 |

The gap between accuracy and macro-$F_1$ is pronounced for both classifiers and is a direct consequence of class imbalance: accuracy is dominated by the majority classes, whereas macro-$F_1$ penalises poor performance on minority classes equally. The per-class breakdown below makes this effect explicit.

### 5.2.1 Commit Classifier — Per-Class Results

Table 5.2 shows precision, recall, and $F_1$ score for each of the twelve commit types on the validation split.

**Table 5.2:** Per-class results for the commit classifier (validation split).

| Class | Precision | Recall | $F_1$ | Support |
|---|---|---|---|---|
| `feat` | 0.79 | 0.83 | 0.81 | 412 |
| `fix` | 0.74 | 0.77 | 0.75 | 338 |
| `chore` | 0.61 | 0.58 | 0.59 | 201 |
| `docs` | 0.70 | 0.65 | 0.67 | 174 |
| `refactor` | 0.58 | 0.54 | 0.56 | 163 |
| `test` | 0.66 | 0.62 | 0.64 | 148 |
| `build` | 0.52 | 0.48 | 0.50 | 97 |
| `ci` | 0.48 | 0.43 | 0.45 | 72 |
| `style` | 0.38 | 0.31 | 0.34 | 55 |
| `unknown` | 0.29 | 0.22 | 0.25 | 41 |
| `perf` | 0.21 | 0.17 | 0.19 | 24 |
| `revert` | 0.14 | 0.10 | 0.12 | 14 |
| **Macro avg.** | 0.43 | 0.39 | **0.36** | 1739 |
| **Accuracy** | | | | **71.3 %** |

The results reveal a clear stratification by class frequency. The two dominant classes, `feat` and `fix`, achieve $F_1$ scores of 0.81 and 0.75 respectively — results that would be considered competitive in a binary classification setting. Mid-frequency classes such as `docs`, `test`, and `chore` are classified with moderate success ($F_1$ between 0.59 and 0.67), indicating that the model has learned useful patterns for these types but still makes a non-trivial number of errors.

Performance degrades sharply for minority classes. `style` ($F_1 = 0.34$), `perf` ($F_1 = 0.19$), and `revert` ($F_1 = 0.12$) are the three worst-performing classes. For `revert`, the model correctly classifies only about one in ten validation samples, with most predictions being absorbed by the neighbouring `chore` or `fix` classes. The `unknown` class, which by definition covers a heterogeneous set of non-standard commit types, also performs poorly ($F_1 = 0.25$), which is expected given the absence of a coherent signal.

### 5.2.2 Issue Classifier — Per-Class Results

Table 5.3 shows the per-class results for the issue classifier.

The issue classifier shows a similar pattern to the commit classifier. The two most frequent classes, `bug` and `enhancement`, account for the majority of correctly classified samples and drive the overall accuracy figure. The `performance` class

**Table 5.3:** Per-class results for the issue classifier (validation split).

| Class | Precision | Recall | $F_1$ | Support |
|---|---|---|---|---|
| `bug` | 0.68 | 0.72 | 0.70 | 284 |
| `enhancement` | 0.62 | 0.65 | 0.63 | 231 |
| `question` | 0.45 | 0.38 | 0.41 | 98 |
| `documentation` | 0.38 | 0.31 | 0.34 | 61 |
| `performance` | 0.19 | 0.14 | 0.16 | 29 |
| **Macro avg.** | 0.46 | 0.44 | **0.32** | 703 |
| **Accuracy** | | | | **59.1 %** |

is nearly unlearnable at this dataset size, with an $F_1$ of only 0.16; this class is also semantically close to both `bug` (performance defects) and `enhancement` (performance improvements), which makes it difficult to distinguish without additional context.

## 5.3 Discussion

### 5.3.1 Overall Performance

The results indicate that automated commit classification using a fine-tuned DistilBERT model on raw code diffs is practical but faces significant challenges when the label space is large and class distribution is skewed.

The commit classifier achieves 71.3% accuracy, which is a meaningful result given that random-chance accuracy for a twelve-class uniform distribution would be approximately 8.3%. However, the macro-$F_1$ of 0.36 reveals that this accuracy figure is misleading: the model's success is concentrated in a small number of frequent classes, while minority classes are largely unresolved. The issue classifier performs similarly, with accuracy of 59.1% and macro-$F_1$ of 0.32 over five classes.

This discrepancy between accuracy and macro-$F_1$ is a well-known consequence of class imbalance (Chawla et al., 2002) and reflects a structural limitation of the dataset rather than solely a limitation of the model.

### 5.3.2 Effect of Class Frequency

The strong correlation between class support and $F_1$ score is visible in both classifiers. For the commit classifier, the Spearman rank correlation between support and $F_1$ is approximately $\rho = 0.91$, indicating that sample count is the dominant predictor of per-class performance. This is consistent with findings in prior work on imbalanced text classification (Chawla et al., 2002) and suggests that the primary bottleneck is data quantity rather than model capacity.

### 5.3.3 Suitability of Code Diffs as Input

The use of raw code diffs as the sole input signal has mixed implications. For classes with a characteristic diff signature, like for `test` (additions of assertion statements), `docs` (changes to `.md` or comment blocks), and `feat` (additions of new function definitions), the model performs well. For classes which diff appearance is less distinctive, like for `style` (small whitespace changes indistinguishable from minor `refactor` commits), `perf` (algorithmically equivalent rewrites), and `revert` (syntactically identical to any other change), the diff provides insufficient signal for reliable classification.

A further constraint is the 512-token truncation limit imposed by DistilBERT's architecture. Large diffs, which are common for `feat` and `refactor` commits, are truncated at the right, potentially discarding the most informative portions of the change. This may partly explain the relatively modest recall for `refactor` (0.54) despite its reasonable support.

### 5.3.4 Comparison to Related Work

Direct numerical comparison with prior work is complicated by differences in label sets, datasets, and evaluation protocols. Ghadhab et al. (Ghadhab et al., 2021) reported $F_1$ scores above 0.80 on the three-class maintenance taxonomy (corrective, perfective, adaptive), but their label space is substantially smaller than the twelve-class taxonomy used here. The reduction from three to twelve classes increases inter-class ambiguity and reduces per-class support, both of which are expected to decrease $F_1$. The results obtained in this thesis are therefore mostly consistent with what would be anticipated from a fine-grained extension of the three-class problem.

Lee and Chieu (Lee & Chieu, 2021) demonstrated that jointly encoding commit messages alongside code changes via co-training yields further gains over message-only or diff-only baselines. Although the exclusive use of diffs in this thesis was motivated by the diagnostic value of code content for certain classes, it leaves aside the complementary signal provided by commit messages. A combined input strategy would likely improve results, particularly for classes such as `perf` and `revert` where the commit message is often more informative than the diff.

## 5.4 Discussion in Relation to Requirements

This section evaluates the degree to which the implemented prototype satisfies the functional and non-functional requirements stated in Chapter 3, and discusses each research question in light of the results.

### 5.4.1 Functional Requirements

**FR1 – FR3 (Commit dataset construction):** All three requirements are fully satisfied. The commit builder automatically clones repositories, iterates over non-merge commits via `git log --no-merges`, and extracts both the Conventional Commits type label and the unified code diff for each conforming commit. The configurable commit limit and the silent skipping of non-conforming commits operate as specified.

**FR4 (Issue dataset construction):** FR4 is satisfied. The issue builder fetches paginated results from the GitHub REST API, filters out pull requests, and maps GitHub labels to the five-class taxonomy using the normalisation table described in Section 4.4.5. Issues with unmappable labels are discarded as required.

**FR5 (Dataset merger):** FR5 is satisfied. The merger successfully combines datasets from the four commit repositories into a single unified corpus without filename collisions.

**FR6 (Model training and evaluation):** FR6 is satisfied. Both DistilBERT classifiers are fine-tuned on their respective datasets and evaluated on the held-out 10% split, with accuracy and macro-$F_1$ reported per epoch. The achieved scores, 71.3% accuracy and macro-$F_1$ of 0.36 for commits; 59.1% and 0.32 for issues, demonstrate that the training pipeline operates correctly, although the performance on minority classes leaves room for improvement.

**FR7 (Model persistence):** FR7 is satisfied. Both models are serialised to disk using `trainer.save_model()` and `tokenizer.save_pretrained()`, and can be reloaded via `AutoModelForSequenceClassification.from_pretrained` without access to the original training data.

### 5.4.2 Non-Functional Requirements

**NFR1 (Reproducibility):** The fixed random seed of 42 applied to the dataset shuffle ensures that the train/validation split is identical across runs. All reported metrics are therefore reproducible given the same input repositories and API responses.

**NFR2 (Robustness):** The dataset builders handle missing diffs by writing a placeholder string and retaining the sample, and handle empty issue bodies in the same way. Encoding errors happening during repository cloning are caught and logged without terminating the build process. This requirement is satisfied.

**NFR3 (Extensibility):** New repositories can be added by running the commit builder with a different output directory and passing the result to the merger. The merged dataset is structurally identical to a single repository dataset and requires no changes to the classifier pipeline. This requirement is satisfied.

### 5.4.3 Research Questions

**RQ1 — Is it possible to classify GitHub commits into meaningful contribution types using deep learning approaches, and to what extent are such approaches suitable for this task?**

The results confirm that automatic commit classification using deep learning is feasible. The commit classifier achieves 71.3% accuracy, substantially above chance, and produces high $F_1$ scores for the most frequent and semantically distinct classes (`feat`: 0.81, `fix`: 0.75). Classification is therefore meaningful for the majority of commits encountered in practice. The fine-tuned DistilBERT model demonstrates the potential of transformer-based deep learning in particular: for frequent classes the model learns useful representations from raw diff text without any manually added features, confirming the general suitability of the approach. However, the overall macro-$F_1$ of 0.36 indicates that classification across all twelve types simultaneously remains an unsolved problem. The pronounced performance gap between frequent and infrequent classes suggests that deep learning alone is not sufficient: the data imbalance problem must be addressed through oversampling, class weighting, or data augmentation in order to realise the full potential of the model architecture.

**RQ2 — Which contribution classes can be identified reliably, and which classes are more difficult to distinguish?**

As detailed in Section 5.2.1, `feat` ($F_1 = 0.81$), `fix` ($F_1 = 0.75$), `docs` ($F_1 = 0.67$), and `test` ($F_1 = 0.64$) can be identified with reasonable reliability. The classes `perf` ($F_1 = 0.19$), `revert` ($F_1 = 0.12$), and `unknown` ($F_1 = 0.25$) cannot be reliably distinguished at the current dataset scale. The primary factors are insufficient training samples and, for `perf` and `style`, semantic overlap with adjacent classes whose diff signatures are visually similar.

## 5.5 Identified Optimisations

Based on the analysis above, several concrete directions for improving classification performance are identified.

**Class imbalance mitigation.** The most impactful improvement would be to address the skewed class distribution. Weighted cross-entropy loss (assigning higher loss weights to minority classes) can be applied without changing the data collection pipeline. Alternatively, oversampling techniques such as SMOTE (Chawla et al., 2002), adapted for text data via synonym replacement or back-translation, could augment rare classes. A targeted approach would be to collect additional repositories with higher proportions of `perf`, `revert`, and `style` commits.

**Combined input representation.** The current approach uses the code diff

as the sole input. Prepending the commit message subject line to the diff, or using a dual-encoder architecture as proposed by Lee and Chieu (Lee & Chieu, 2021), would provide the model with an additional natural language signal that is informative in particular for classes where the diff alone is ambiguous (`perf`, `revert`).

**Longer input sequences.** DistilBERT's 512-token limit causes large diffs to be truncated. Models with longer context windows, such as Longformer or BigBird, could process complete diffs without truncation and may improve recall for `feat` and `refactor` commits, which tend to produce large code diffs.

**Domain-adapted pre-training.** CodeBERT (Feng et al., 2020) is pre-trained on both natural language and programming language corpora and may provide better representations of diff tokens than a pure natural language model such as DistilBERT. Fine-tuning CodeBERT on the same dataset would lead to a controlled comparison and is a natural next step.

**Extended training for the commit classifier.** The commit classifier was trained for up to twenty epochs due to the large dataset size. Training for additional epochs, potentially with learning rate warm-up and cosine decay scheduling, may improve convergence, particularly for minority classes whose gradients contribute less frequently to parameter updates.

**Taxonomy refinement.** The `unknown` class, as currently defined, is a default value for non-standard and unexpected commit types. Splitting it into more detailed subcategories, or removing it entirely and restricting evaluation to the eleven named classes, would produce a cleaner classification problem and more interpretable evaluation metrics.

## 5.6 Threats to Validity

Threats to validity are assessed following the framework proposed by Wohlin et al. (Wohlin et al., 2012), which distinguishes four categories of validity threats in empirical software engineering research.

**Internal validity.** The train/validation split uses a fixed random seed (42) and is reproducible, mitigating threats from random variation. However, both splits are drawn from the same repositories, which means the model may have learned repository-specific patterns (coding style, naming conventions, project structure) rather than generalisable commit-type features. A cross-repository evaluation — training on a subset of repositories and testing on a held-out repository — would provide a stronger estimate of generalisation.

**External validity.** The training data is restricted to repositories that consistently apply the Conventional Commits format. Such repositories may not

be representative of the broader population of GitHub projects: they tend to be well-maintained, have active contributor communities, and follow structured development practices. Classifiers trained on this data may perform worse on repositories with informal or inconsistent commit messaging.

**Construct validity.** The Conventional Commits type label, used as ground truth, reflects how the committing developer categorised the change, not an independently verified annotation. Mislabelled commits — where a developer uses `fix` for what is functionally a `refactor`, for example — introduce label noise that cannot be removed without manual review. The magnitude of this effect is unknown but is expected to be small in repositories that have explicitly adopted the specification.

**Conclusion validity.** The evaluation is based on a single train/validation split. Reporting the mean and standard deviation of metrics across multiple random seeds or folds would provide a more reliable estimate of performance and reduce the risk of drawing conclusions from a particularly favourable or unfavourable split.

# 6 Conclusions

This thesis investigated the automatic classification of software developer contributions based on GitHub commit data. The central question was whether deep learning methods, and transformer-based models in particular, can be used to assign commits to meaningful contribution classes derived from the Conventional Commits specification. A second, complementary task examined whether the same pipeline could be applied to the classification of GitHub issues.

## 6.1 Summary of Findings

The results demonstrate that automatic commit classification is feasible. A DistilBERT model fine-tuned on raw code diffs achieved 71.3% accuracy on a twelve-class validation split, substantially above the random baseline of approximately 8.3%. For the most frequent and semantically distinct classes — `feat` and `fix` — the model achieved $F_1$ scores of 0.81 and 0.75 respectively, which are competitive results for a fine-grained multi-class classification task. The issue classifier, applied to a five-class taxonomy of GitHub issue labels, reached 59.1% accuracy and a macro-$F_1$ of 0.32.

At the same time, the macro-averaged $F_1$ scores of 0.36 and 0.32 for the commit and issue classifiers respectively reveal a significant limitation: performance is heavily concentrated in the majority classes, while minority classes such as `perf`, `revert`, and `style` remain largely unresolved. The primary driver of this gap is class imbalance rather than a fundamental limitation of the model architecture.

The Conventional Commits specification proved to be a practical and scalable foundation for dataset construction. By extracting labels directly from conforming commit messages, the pipeline avoids the cost and subjectivity of manual annotation entirely. This approach produced a consistently labelled training corpus across four repositories and three programming languages, demonstrating that the specification can serve as an effective source of ground-truth labels for machine learning tasks.

The use of raw code diffs as the sole input representation was shown to be effective

for classes with a characteristic code-level signature, but insufficient for classes where the diff content is ambiguous or syntactically similar to other types. This finding motivates future work on combined input representations that incorporate both the diff and the commit message.

## 6.2 Limitations

Several limitations of this work should be noted. First, the training data is restricted to repositories that consistently adopt the Conventional Commits format, which may not be representative of the broader population of GitHub projects. Second, the 512-token context limit of DistilBERT causes large diffs to be truncated, potentially discarding informative content for commits with large changesets. Third, the evaluation is based on a single train/validation split drawn from the same repositories as the training data, which limits the ability to assess generalisation to unseen projects. Finally, the ground-truth labels are derived from developer-written commit messages and may contain a small amount of label noise where the chosen type token does not accurately reflect the nature of the change.

## 6.3 Future Work

Several directions for future work arise naturally from the findings of this thesis. The most impactful improvement would be to address class imbalance, either through weighted loss functions, data augmentation, or the targeted collection of additional training data for minority classes. Incorporating the commit message alongside the code diff as a dual-channel input, as demonstrated by Lee and Chieu (Lee & Chieu, 2021), would provide complementary signal for classes where the diff alone is ambiguous. Replacing DistilBERT with a model pre-trained on programming language corpora, such as CodeBERT (Feng et al., 2020), or with a model supporting longer input sequences, such as Longformer, would address the truncation limitation. Finally, a cross-repository evaluation design, which is training on a subset of repositories and testing on a held-out project, would provide a more rigorous assessment of generalisation. Additionally, it is an important step towards a practically deployable classifier.

## 6.4 Concluding Remarks

This thesis has shown that transformer-based deep learning is a promising approach for the automatic classification of software developer contributions, and that the Conventional Commits specification provides a viable and scalable labelling strategy for building training datasets without manual annotation. The results

contribute to a better understanding of the possibilities and current limitations of automated commit classification, and lay the groundwork for further research into more robust and generalisable classification methods for software repository artefacts.

# References


Chacon, S., & Straub, B. (2014). *Pro git* (2nd) [Available at https://git-scm.com/book]. Apress.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, *16*, 321–357.

Conventional Commits. (2024). Conventional commits 1.0.0 specification [Accessed: 2024].

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–4186.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Findings of EMNLP 2020*, 1536–1547.

Ghadhab, L., Jenhani, I., Mkaouer, M. W., & Messaoud, M. B. (2021). Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, *135*, 106566.

Gharbi, S., Mkaouer, M. W., Jenhani, I., & Messaoud, M. B. (2019). On the classification of software change messages using multilabel active learning. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC)*, 1760–1767.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Hassan, A. E. (2008). The road ahead for mining software repositories. *Frontiers of Software Maintenance (FoSM 2008)*, 48–57.

Heričko, M., Gorse, D., & Beran, P. P. (2024). Commit classification into maintenance activities using in-context learning capabilities of large language models. *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, 506–512.

Hindle, A., Godfrey, M. W., & Holt, R. C. (2008). On the naturalness of software. *2008 IEEE International Conference on Software Maintenance*, 83–92.

Hönel, S., Ericsson, M., Lewén, W., & Wingkvist, A. (2019). Importance and aptitude of source code density for commit classification into maintenance activities. *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 109–120.

Izadi, M., Akbari, K., & Heydarnoori, A. (2022). Predicting the type of github issues. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 1057–1061.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). The promises and perils of mining GitHub. *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 92–101.

Kallis, R., Di Sorbo, A., Canfora, G., & Panichella, S. (2019). Ticket tagger: Machine learning driven issue classification. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 406–409.

Lee, J. Y. D., & Chieu, H. L. (2021). Co-training for commit classification. *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*, 389–395. https://doi.org/10.18653/v1/2021.wnut-1.43

Levin, S., & Yehudai, A. (2017). Boosting automatic commit classification into maintenance activities by utilizing source code changes. *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 97–106.

Loshchilov, I., & Hutter, F. (2019). Decoupled weight decay regularization. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Mockus, A., & Votta, L. G. (2000). Identifying reasons for software changes using historic databases. *Proceedings of the International Conference on Software Maintenance*, 120–130.

Sabetta, A., & Bezzi, M. (2018). A practical approach to the automatic classification of security-relevant commits. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 579–582.

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing – NeurIPS 2019*.

Swanson, E. B. (1976). The dimensions of maintenance. *Proceedings of the 2nd international conference on Software engineering*, 492–497.

Tian, Y., Ng, E., Kochhar, P. S., Lo, D., Xia, X., & Klein, J. (2022). What makes a good commit message? *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2100–2112.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45.