# Towards Defect Prediction for Quantum Software*

Xuan Mao[§, †], Zijie Huang[†], Jianxin Ge[†], Chao Wang[†], Wuxu Wang[§, †], Lizhi Cai[§, †]
[§]*Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai, China*
[†]*Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai Development Center of Computer Software Technology, Shanghai, China*
maoxuan@mail.ecust.edu.cn, {huangzj, gjx, wc}@sscenter.sh.cn, wxw@mail.ecust.edu.cn, clz@sscenter.sh.cn

***Abstract*—Software defects are faults or bugs within a program that can lead to incorrect or unexpected outcomes. Efficiently allocating software quality assurance (SQA) resources to components with a higher likelihood of defects, based on software defect prediction (SDP) models, can save significant effort. While SDP has been extensively studied in classical software, its applicability to quantum software remains unexplored. Defect prediction for quantum software presents unique challenges, including the susceptibility to quantum-specific defects arising from quantum coding conventions and the limited size of available datasets. To address these issues, we propose QDP-FSL, an SDP model using a pre-trained code model to capture the semantics of quantum software code, and applying few-shot learning (FSL) to learn effectively from a small number of defective samples. Results show that QDP-FSL outperforms baseline methods that rely on static analysis. This work lays a foundation for future research in defect prediction for quantum software engineering and outlines potential directions for further improvement.**

***Index Terms*—quantum software engineering, software defect prediction, few shot learning, software quality assurance**

## I. Introduction

Software defects, defined as faults or bugs within a program, can result in incorrect or unexpected outcomes in the final product [1]. Software for quantum computing (i.e., quantum software) is receiving increasing attention due to its recent advancements in cryptography, optimization, and artificial intelligence. As a subset of software, quantum software is not immune from defects. Due its potential to revolutionize various domains, reliable software quality assurance (SQA) practices are essential to identify and resolve these defects.

Software defect prediction (SDP) approaches are utilized to identify defective code components for more efficient SQA [2]. Without SDP, identifying defects in quantum software can be expensive, even when automated. For instance, the differential testing tool QDiff [3] requires running programs in parallel across various backends. However, some proprietary backends impose strict limitations and pricing models, and generating program variants for testing can also be time-consuming. This process could be more efficient if focused specifically on the most error-prone components of the code. Meanwhile, SDP can serve as a valuable guide for code reviews, especially when resources are limited. In such cases, the most defective code should be prioritized for reviewing.

However, quantum software defects are different from defects of classifical software. A recent study [4] mentioned several quantum-specific defects, such as incorrect usage of qubits and quantum gates, improper measurement positioning, and others. Furthermore, an empirical study [5] found that existing defect prediction practices may not be effective in domains with unique coding conventions. Therefore, we intend to investigate whether defect prediction approaches are feasible in predicting quantum software defects.

Achieving effective defect prediction for quantum software presents several known challenges. The two primary categories of defect prediction approaches are classical machine learning-based and deep learning-based methods. Classical machine learning approaches require feature engineering, with commonly used features being generic software product metrics, such as lines of code and method cohesion [6]. Since quantum software specific defects are less related to code structure (e.g., [4], [7] do not mention defect with such causes), we turn to deep learning approaches instead. Deep learning methods [8] automatically capture semantic and structural characteristics of code by representing them as code embeddings, enabling automatic feature engineering for defect prediction. However, many deep learning approaches are data-hungry, making it difficult for them to learn effectively from small quantum software defect datasets. To address this issue, we could either balance the dataset or implement more efficient learning strategies. We choose not to balance our dataset through data synthesis or sampling because (1) synthesized data may not accurately reflect real-world scenarios, potentially introducing model bias [9], and (2) there is no empirical basis to suggest that any code component should be balanced with respect to the others [10]. Therefore, we propose QDP-FSL, an approach using few-shot learning (FSL) [11] that learns from limited samples effectively for defect prediction in quantum software.

The main contributions of our paper are as follows:

(1) To the best of our knowledge, this is the first study to conduct defect prediction for quantum software. Additionally, it is the first research to apply a pre-trained code model and FSL to quantum software engineering tasks.

(2) We provide an assessment of QDP-FSL, comparing its performance against static analysis baselines. QDP-FSL demonstrates strong performance and outperforms the baseline. We also analyze the prediction results to identify potential directions for improvement in future research.

(3) We present a replication package[1] that contains the model construction and prediction process.

*Corresponding author: Lizhi Cai

[1]https://github.com/backordinary/QDP-FSL

## II. Related Work

The following sections provide an overview of studies in SDP, quantum software defect analysis, and FSL for defect prediction related to the scope of our study.

### A. Software Defect Prediction

SDP on classical software rely on machine learning, and deep learning techniques to predict potential defects. These models analyze code metrics and historical data to find patterns linked to defects. Machine learning techniques are conducted based on product and process features [6] using models such as Random Forest, Decision Tree, Logistic Regression, and so on. Deep learning based techniques are emerging since they could automatically extract features and achieve good performance, for example, Wang et al. [8] utilized a convolutional neural network (CNN) with an optimized AST node granularity for cross-project defect prediction, while Ardimento et al. [12] employed a deep neural network for just-in-time defect prediction. Farid et al. [13] combined long short-term memory networks with CNN in a hybrid model to improve prediction accuracy, and Wang et al. [14] used a deep belief network to extract semantic features for defect prediction. Recent study indicated the unique coding convention of domain specific software may hinder the feasibility of defect prediction [5]. This leads to our motivation of studying the feasibility of conducting SDP to quantum software.

### B. Few-Shot Learning in Software Defect Prediction

In SDP tasks, the scarcity of defect data presents a major challenge. Boehm and Basili [15] observed that defect data distribution follows the Pareto principle, i.e., 20% of modules often account for 80% of software defects, which means that defective data is both limited and unevenly distributed. Such a distribution hinders SDP models from accurate prediction. FSL could addresses this issue by effectively learning from a small set of labeled samples, leveraging prior knowledge to make reliable predictions even with limited data [16]. FSL is originally introduced by Li et al. [17], which mimics human learning relying on only a few examples.

In terms of FSL application in defect prediction, Zhao et al. [18] applied a Siamese parallel fully-connected neural network (SPFCNN) to tackle the issue of limited defect data in SDP. This model combines the strengths of Siamese networks and deep learning, allowing it to better identify defect patterns with minimal labeled data. SPFCNN is trained using the AdamW algorithm to optimize weight adjustments, which improves its prediction accuracy across imbalanced datasets. In another study, Wang et al. [11] proposed an FSL approach based on balanced distribution adaptation (FSLBDA). This method addresses both class imbalance and dataset heterogeneity by first minimizing the differences in marginal and conditional distributions between source and target datasets and then adaptively assigning weights to these distributions. Through these techniques, FSLBDA effectively enhances defect prediction accuracy across various datasets.

### C. Quantum Software Defect Analysis

In an empirical study, Zhao et al. [19] found that 28% of the bug patterns in quantum software are quantum computing specific, involving issues such as incorrect unitary matrix implementation in qubit manipulation. Threfore, researchers developed several specialized tools to localize specific types of defect in quantum software. QChecker [20] is a static analysis tool for Qiskit quantum program bug detection which extracts program information by static analysis of abstract syntax trees and uses pattern-based detection to find defects caused by resource allocation errors or incorrect quantum gate usage. QSmell [21], in contrast, mainly uses dynamic analysis to detect "code smells" in quantum programs, such as long circuits and redundant quantum gates. Although by definition, code smells are software maintenance issues that do not necessarily lead to bugs, QSmell still detects buggy implementation. The study showed that about 73% of quantum programs contain one or more code smells, which may lead to performance and reliability issues. LintQ [22] is a more comprehensive quantum code analysis tool that captures quantum-specific concepts by introducing high-level concepts of quantum circuits, gates, and qubits. Based on these concepts, it detects 10 types of programming problems, including potential quantum state errors, redundant measurements, and incorrect subcircuit use. LintQ achieves 91% precision in a selected subset of problems, which is superior to traditional defect localization tools.

## III. Dataset Construction

The section describes how we construct a dataset containing defective and clean samples for defect prediction.

### A. Collecting Defective Samples

To obtain defective samples, we selected Bugs4Q as the primary source. Bugs4Q [7] is a defect benchmark specifically designed for quantum programming, containing 42 manually validated Qiskit defects from GitHub, StackOverflow, and Stack Exchange, submitted by Qiskit users, focusing on quantum-specific defects that from real-world usage scenarios. Moreover, it also ensures that each defect is reproducible, even when probabilistic outputs make exact reproducibility challenging. Moreover, it guarantees isolated fixes, excluding unrelated refactoring or modifications, and thus preserving the independency of each defect samples. Lastly, it provides complete pre- and post-fix code, supporting test case construction and verification of resolution effectiveness. With these characteristics, Bugs4Q stands as a high-quality, quantum-specific, and reproducible dataset.

### B. Collecting Non-Defective (Clean) Samples

We select the LintQ dataset [22], the largest available collection of 7,568 real-world Qiskit programs, as a foundational source for clean samples. According to LintQ's evaluation, the original dataset contains some code with implementational problems. From our understanding, these "problems" are not identical to bugs, i.e., some problematic code components may not be buggy. To ensure the cleanness of our dataset, these

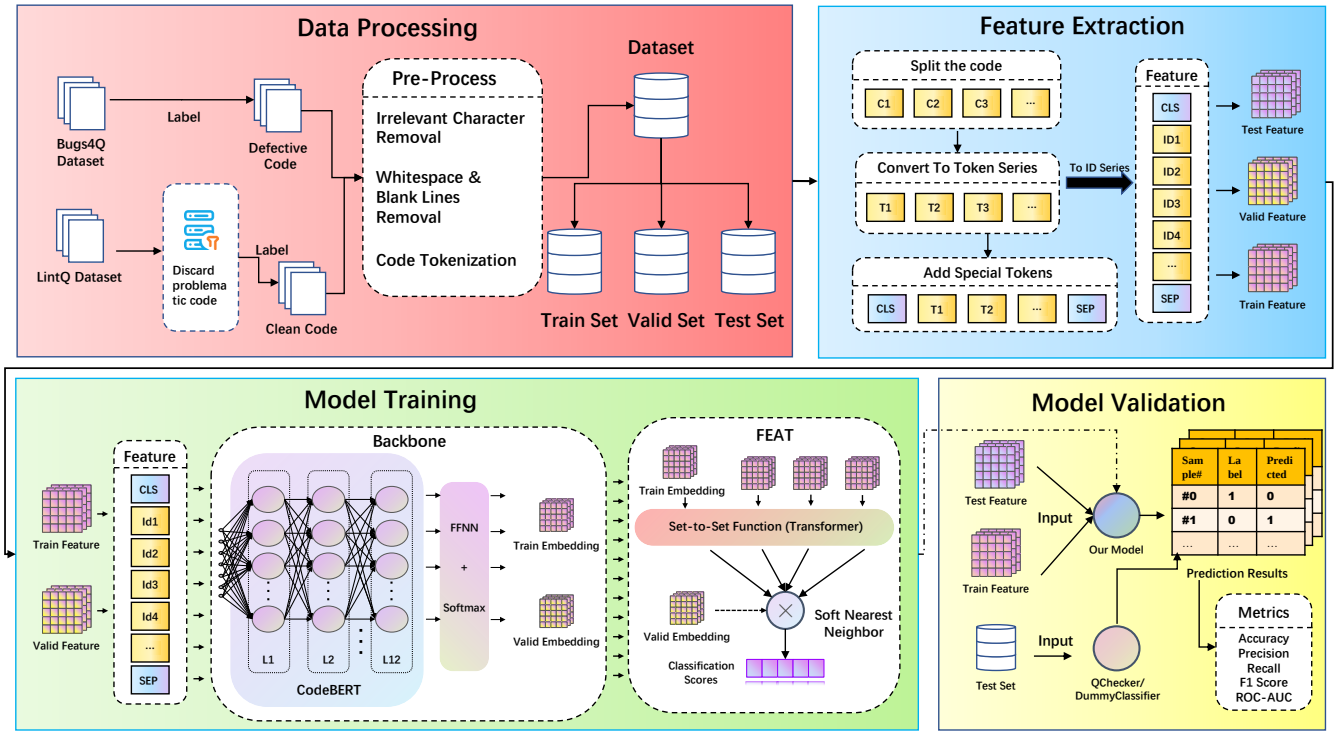


Fig. 1. The process of training and validating QDP-FSL.

samples should be discarded. First, we check LintQ's detection report and exclude samples marked with having potential problems. Second, we review the manual annotation and tool generated information of all remaining samples to remove any samples reported problematic, e.g., we exploit CodeQL [23], LintQ and Qsmell [21] to identify and exclude problematic samples. Finally, although we do not have enough resource to manually investigate every code snipet, we still randomly select 100 programs to verify the effect of this process. The 1st, 2nd, and 5th authors (2 Master's students and 1 Post-doc) with quantum software development experience manually check if they contain any noticeable defects, and we do not find any defective samples. Indeed, it is still possible that the dataset contain defective samples, however, this process filters out suspectable samples by most currently available tools, and our validation shows it is promising as a clean dataset.

Finally, our dataset contains 6,294 samples, of which 42 are labeled as defective, while the remainders are labeled as clean.

### C. Data Preprocessing

First, we remove blank lines and excessive whitespace in codes. Then, we use a tokenizer that converts each code sample into a fixed-length sequence of tokens. Each code snippet is tokenized and then padded or truncated to be fit into a specified block size (set to 256 in our implementation). Lastly, we add special tokens `[CLS]` and `[SEP]` at the beginning and the end of each sequence to mark the start and end positions.

## IV. Experimental Design

Fig. 1 shows the process of our experiment. We undergo data processing and feature extracting procedures to generate model input, and train QDP-FSL for further validation. This section describes these processes in detail.

### A. Research Questions

The goal of our study is to investigate whether it is feasible to predict defect for quantum software, with the purpose of localizing the most error-prone code components to save SQA efforts. To these ends, we propose the following 3 research questions, and design the experiment accordingly.

**RQ1.** How well can QDP-FSL predict defect for quantum software?

The motivation of this part of study is that since defect prediction has proven beneficial in classical software engineering, we intend to verify if quantum software SQA could also benefit from such methods.

**RQ2.** Compared with existing approaches, can QDP-FSL achieve better prediction performance?

In this RQ, we verify whether QDP-FSL is superior to the available related methods designed for defect localization based on static program analysis.

**RQ3.** How can we further improve defect prediction for quantum software?

We will further conduct case studies and look into the predictions, which could foster the understanding of QDP-FSL, and provide empirical insights for future model improvement.

### B. Prediction Model Architecture

First, we employ CodeBERT [24] as the pre-trained code model to extract code representation embeddings and capture code semantics. CodeBERT, known for its strong generalizability [25], has been trained on a multi-language dataset from GitHub, covering major languages used to implement quantum software such as Python and C#, making it transferrable across various software engineering tasks. This pretraining enables CodeBERT to handle a wide range of programming languages and capture diverse code patterns effectively. CodeBERT utilizes a bidirectional transformer encoder architecture to model dependencies and contextual relationships within code, allowing it to represent variable and function interactions with high granularity. This comprehensive representation is particularly advantageous in FSL scenarios, as it provides informative embeddings that facilitate effective classification and defect prediction with limited data.

Next, we employ FEAT (Few-Shot Embedding Adaptation with Transformer) [26] for FSL implementation. FSL models are trained to use the examples in the support set to predict the class of a query sample. Thus, an FSL task could be represented as an $N$-way, $M$-shot classification problem [27], where there are $N$ classes in the support set, and each class has $M$ labeled examples. We use a fixed $N = 2$ (which refers to clean and defective classes), and after tuning, we set $M = 7$ with a single query sample per task. The introduction of FEAT is available in the next section.

Finally, to implement the model, we use CodeBERT as the feature extraction backbone, where its output is a vector with a dimension equal to 768 (i.e., the hidden size). This vector is then fed into an 8-head multi-head attention module, producing a concatenated set of transformed embeddings, which are subsequently passed into the main FEAT network.

### C. Few-Shot Embedding Adaptation with Transformer (FEAT)

Traditional FSL models rely on fixed embedding functions for generalization, while FEAT adaptively refines class centers in the support set, making them more representative for accurate classification of query samples.

We aim to classify samples into either clean class (labeled 0) or defective class (labeled 1). To make the class centers in the embedding space more distinguishable, FEAT introduces a set-to-set transformation function $T(\cdot)$, which transforms a set of original class centers $C = (c_0, c_1)$ into a set of adapted class centers $C' = (c'_0, c'_1)$, where $C' = T(C)$. Transformer [28] architecture is used for this transformation function [26].

First, we define the transformed query, key, and value representations. For the original class centers $C$, we obtain the transformed query, key, and value as follows:

$$\tilde{Q} = W_Q C, \quad \tilde{K} = W_K C, \quad \tilde{V} = W_V C, \tag{1}$$

where $W_Q$, $W_K$, and $W_V \in \mathbb{R}^{m \times hm}$ are learnable linear transformation matrices, $m$ is the embedding dimension, and $h$ is the number of attention heads. In the context of defect prediction, $C$ represents the centers of the clean and defective classes. These transformation matrices allow the model to encode the features of the class centers, which are then processed in the attention mechanism.

Next, multi-head self-attention is used to compute attention weights and outputs:

$$V = \tilde{V}\,\text{softmax}\left(\frac{\tilde{Q}^T \tilde{K}}{\sqrt{m}}\right)^T, \tag{2}$$

This self-attention mechanism captures contextual relationships between class centers without considering their order, which is particularly suitable for adjusting class prototypes in FSL tasks. The final adapted class centers are calculated as:

$$C' = \text{LayerNorm}(\text{Dropout}(W_{FC} V) + C), \tag{3}$$

where $W_{FC} \in \mathbb{R}^{hm \times m}$ is a fully connected, trainable weight matrix. The combination of LayerNorm and Dropout enhances the stability and generalizability of the embeddings.

With these adapted class centers $C'$, the model can better distinguish between clean and defective classes in the embedding space. The classification probability for a given sample is then defined as:

$$f(x, C') = \Pr(y = s|x, C') = \frac{e^{-\text{dist}(E(x), c'_s)}}{\sum_{c'_i \in C'} e^{-\text{dist}(E(x), c'_i)}}, \tag{4}$$

where $c'_s \in C'$ represents the adapted center of class $s$, and $\text{dist}(\cdot)$ is a distance metric measuring the similarity between the sample and class centers. This formulation ensures that the query sample aligns with the correct class center in the support set (either clean or defective).

To maintain intra-class compactness, an embedding adaptation loss is applied to the query set $Q_{\text{seen}}$:

$$\mathcal{L}_{\text{query}} = \sum_{(x,y) \in Q_{\text{seen}}} \mathcal{L}_{\text{CE}}(y, f(x, C')), \tag{5}$$

where $\mathcal{L}_{\text{CE}}$ denotes the cross-entropy loss, measuring the discrepancy between predictions and true labels. This loss encourages query samples to be close to their respective class centers and distant from other class centers.

To further maintain consistency within each class, FEAT incorporates a contrastive loss. Both support and query samples $Q_{\text{seen}} \cup S_{\text{seen}}$ are passed through the embedding adaptation module to recalibrate the class centers $C$. The contrastive loss is then defined as:

$$\mathcal{L}_{\text{contrastive}} = \sum_{(x,y) \in Q_{\text{seen}} \cup S_{\text{seen}}} \mathcal{L}_{\text{CE}}(y, f(x, C)), \tag{6}$$

where $C$ includes recalibrated class centers based on the adapted embeddings. This forces the adapted instance embeddings to remain close to their own class center and far away from other centers, enhancing inter-class separation.

The overall embedding adaptation loss in the FEAT model combines the query loss and the contrastive loss as follows:

$$\mathcal{L}_{\text{FEAT}} = \mathcal{L}_{\text{query}} + \alpha \mathcal{L}_{\text{contrastive}}, \tag{7}$$

where $\alpha$ is a hyperparameter that balances the two loss components. This structure enables the FEAT model to learn compact embeddings within each class and well-separated embeddings between classes, resulting in improved generalization to new samples in the defect prediction task.

### D. Parameter Tuning

We employ grid search to systematically optimize QDP-FSL's hyperparameters. This approach allow us to explore a predefined range of values for each parameter, helping us identify the optimal configuration for our specific task. The primary hyperparameters[2] and their ranges in this tuning process are as follows:

(1) **Learning Rate** is explored within $[1 \times 10^{-7}, 1 \times 10^{-1}]$ to fine-tune the model's convergence speed and stability.

(2) **Weight Decay** is tuned across $[1 \times 10^{-6}, 1 \times 10^{-2}]$ to regulate the L2 regularization to prevent overfitting.

(3) **Scheduler Gamma** is adjusted within $[1 \times 10^{-3}, 0.9]$, controlling learning rate decay and stablizing convergence.

(4) **Max Gradient Norm** is set between $[0.1, 5]$ to clip gradients, mitigating gradient explosion and stabilize training.

(5) **Number of Tasks per Epoch** is tuned from $[10, 1000]$ to balance between computational efficiency and robustness.

(6) **Number of Validation Tasks per Epoch** is configured between 10 and 700 to provide sufficient validation data per epoch for performance assessment.

### E. Model Validation

We conduct 10 rounds of 4-fold cross-validation to maximize the use of the dataset and ensure the reliability of the results, where each round involves a random split into training and test sets, with the test set always representing 20% of the total data. In each round, we divide the training set into 4 folds, using 3 for actual training and 1 for validation. This setup maximizes training data use while allowing us to monitor model performance on the validation set after each epoch. At each epoch's end, we evaluate the model on the validation set and save the parameters yielding the best performance based on validation loss or accuracy. Saving the best model configuration not only helps prevent overfitting but also ensures that the final model performs consistently across different validation sets. To clarify, the test set remains unseen by the model during training, used only for final evaluation.

### F. Performance Assessment

We select multiple performance metrics to provide a comprehensive evaluation of QDP-FSL's effectiveness. Considering the inherent class imbalance in defect prediction datasets, we specifically emphasize ROC-AUC (Receiver Operating Characteristic-Area Under Curve) as our primary metric, as it effectively evaluates model performance across varying classification thresholds and is less influenced by class distribution disparities, which is recommended by recent defect prediction study [29]. ROC-AUC assesses the model's ability to distinguish between defect and clean samples by calculating the area under the ROC curve, which plots the true positive rate against the false positive rate at different thresholds. This metric is particularly valuable in imbalanced data scenarios, where metrics such as accuracy or F1 Score may not fully capture a model's true discriminative power.

By convention, we also report Accuracy, Precision, Recall, and F1 Score as supplementary metrics of model performance. Accuracy provides an initial overview of the model's overall correct predictions across all samples. Precision measures the proportion of true positives among all positive predictions, helping us understand the model's tendency to avoid false positives. Recall, on the other hand, evaluates the model's effectiveness in identifying all true positive defect instances. Finally, F1 Score combines Precision and Recall to provide a balanced view of the model's defect prediction capability, especially in handling class imbalance.

Calculated as the area under the $TPR - FPR$ curve, ROC-AUC ranges from 0 to 1, where a value of 0.50 indicates a model's performance is equivalent to random guessing, and a value greater than 0.70 indicates good performance [29]. Other metrics range from 0 to 1. The definition of TPR, FPR, Precision, Recall, Accuracy, and F1 are listed as follows:

$$TPR = \frac{TP}{TP + FN}, \tag{8}$$

$$FPR = \frac{TN}{FP + TN}, \tag{9}$$

$$Precision = \frac{TP}{TP + FP}, \tag{10}$$

$$Recall = \frac{TP}{TP + FN}, \tag{11}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{12}$$

$$F1 = \frac{2 \times TP}{2 \times TP + FP + FN}, \tag{13}$$

where *TP* is for true positive (positive sample predicted as positive), *FN* is for false negative (positive sample falsely predicted as negative), *TN* is for true negative, and *FP* is for false positive.

## V. Result and Discussion

### A. RQ1. Model Performance

We evaluate QDP-FSL's performance using 10 rounds of 4-fold cross-validation to ensure robustness across different data splits. Overall, QDP-FSL demonstrates strong performance in quantum software defect prediction, with ROC-AUC performance of 0.798, QDP-FSL significantly outperforms other

[2]https://github.com/Sha-Lab/FEAT

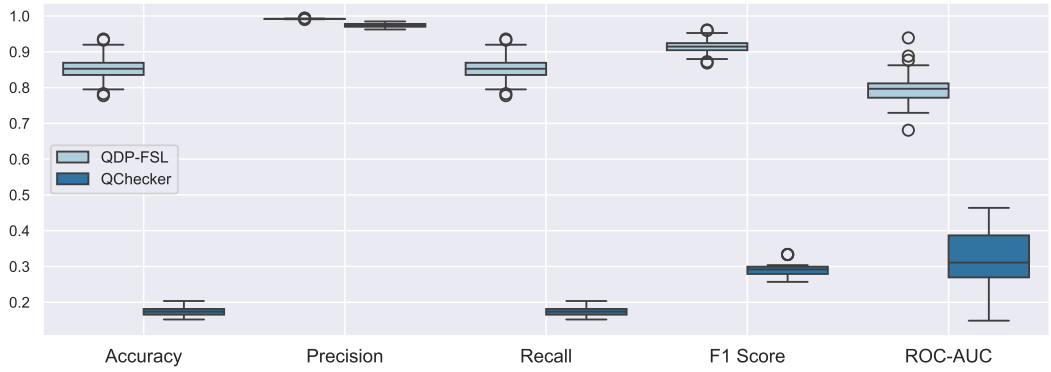


Fig. 2. Performance boxplots of QDP-FSL and QChecker.

TABLE I
PERFORMANCE OF QDP-FSL AND BASELINE MODELS

| Model | Accuracy | Precision | Recall | F1 Score | ROC-AUC |
|---|---|---|---|---|---|
| **QDP-FSL** | 0.855 | **0.992** | **0.855** | **0.916** | **0.798** |
| QChecker | 0.174 | 0.974 | 0.174 | 0.291 | 0.311 |
| DummyClassifier | **0.993** | 0.000 | 0.000 | 0.000 | 0.500 |

baselines. The F1 Score of 0.916 further shows QDP-FSL's reliability in balancing Precision and Recall.

**Finding 1.** QDP-FSL achieves good defect prediction performance for quantum software, showing high ROC-AUC scores, effectively balancing precision and recall.

### B. RQ2. Comparing with Baselines

We primarily compare QDP-FSL's defect prediction performance with the existing quantum defect detection tool, QChecker [20], and a dummy classification model that make only predictions of the majority class (the clean prediction).

- **QChecker** [20] is one of the most adavanced static analysis tool specifically designed for quantum software defect localization, focusing on detecting common defects in Qiskit programs. It addresses not only quantum-specific defects but also a broader range of defect types, and it relies on rules that capture predefined quantum-specific patterns. Since no SDP approach presently exists for quantum defect prediction, we select QChecker as a baseline to assess QDP-FSL's effectiveness compared with currently available solution. To clarify, all code components detected as buggy, regardless of their bug type, are marked as defective.
- **DummyClassifier** is involved to simulate the models that are not able to deal with extremely imbalanced dataset. For example, our early implementation based on a example classifier[3] of the CodeBERT repository has similar behavior as the DummyClassifier since it outputs the clean prediction for almost every instance.

[3]https://github.com/microsoft/CodeBERT/tree/master/CodeBERT/codesearch

Table 1 lists the mean performance of the baselines and QDP-FSL, and better performance is bolded. Fig. 2 demonstrates the model performance boxplots of QDP-FSL and QChecker. The DummyClassifier is not included in the figure because its performance is consistent in each fold. In terms of the comparison with QChecker, although QChecker performs well in Precision, it shows limited coverage in identifying defect samples, with low Recall and F1. Meanwhile, the prediction of the DummyClassifier outputs a high Accuracy of 0.993, but its Precision, Recall, and F1 Score values are 0, and the ROC-AUC performance is no better than random guessing. These results indicate that the approaches that cannot deal with extremely imbalanced dataset will be ineffective.

**Finding 2.** In defect prediction, QDP-FSL achieves better performance than the static analysis baseline.

### C. RQ3. Discussion

Although our quantum software defect prediction model demonstrate good performance, it outputs erroneous results in some cases. This section discusses these cases and propose potential improvement directions.

```
from qiskit.circuit import Parameter
from qiskit.circuit.library import
    RealAmplitudes
from qiskit.opflow import CircuitStateFn
from qiskit.opflow.gradients import Gradient

# The correction is: ansatz = RealAmplitudes(
    num_qubits=1, reps=1).decompose()

ansatz = RealAmplitudes(num_qubits=1, reps=1)

for method in ['param_shift', 'fin_diff', '
    lin_comb']:
    grad = Gradient(method).convert(
    CircuitStateFn(ansatz))
    print(f"{method} is ok")
```

Listing 1. An example of wrong API usage.

**Involving Knowledge of Quantum Software API Usage.** Some bugs can only be detected by executing the program, as they may result in exceptions or incorrect outputs (e.g., as discussed in [7]). These bugs are typically harder to predict

because they often do not involve obvious logical errors or are caused by simple misuse of APIs, and fixing them generally requires only minor adjustments. Defect predictors often classify such issues as clean samples. For example, the buggy statement `RealAmplitudes(num_qubits=1, reps=1)` should be appended with a `decompose()` method call (see Listing 1) to prevent an exception from being raised. This example highlights the challenge of using a general pre-trained model without domain-specific adaptation. We believe that incorporating more quantum programs and their documentations about API usages into the fine-tuning process for these pre-trained models will help them better capture the correct usage patterns of common and critical APIs.

```
import os
from kaleidoscope.errors import
    KaleidoscopeError
try:
    from .version import version as
    __version__
except ImportError:
    __version__ = '0.0.0'
from kaleidoscope.interactive import *
try:
    from qiskit import QuantumCircuit
    from qiskit.providers.aer import Aer
    from qiskit.providers.ibmq import IBMQ
except ImportError:
    HAS_QISKIT = False
else:
    HAS_QISKIT = True
```

Listing 2. An example containing error-related keywords.

```
import os
from kaleidoscope.errors import
    KaleidoscopeError
from .version import version as __version__
from kaleidoscope.interactive import *
from qiskit import QuantumCircuit
from qiskit.providers.aer import Aer
from qiskit.providers.ibmq import IBMQ
HAS_QISKIT = True
```

Listing 3. Modified code of Listing 2 that lead to a clean prediction.

**Enhancing Model Robustness for Specific Semantics.** In contrast, certain code components that are clearly correct are consistently classified as defective. We discovered that the model is highly sensitive to specific tokens, particularly keywords related to errors and bugs. For instance, consider the code snippet in Listing 2, which imports quantum software dependencies and is obviously clean. However, after removing the `try-catch` clauses in the same code (see Listing 3), the model classifies it as clean. This result is misleading, as Listing 3 is actually more erroneous than Listing 2 because it fails to verify the existence of the required dependencies. This observation suggests that the model's sensitivity to error-related keywords may contribute to misclassification, highlighting an area for further improvement in model robustness.

**Mitigating Bias in the Dataset.** We also observe that code components involving index operations and method calls with integer values are more frequently classified as defective (e.g., Listing 4 is misclassified as defective). We attribute this to dataset bias, since the defective samples in our dataset primarily focus on qubit-related issues, and many of these defective instances involve such operations, which likely contributes to the model's biased predictions. To address this issue, we propose incorporating a broader range of defect types and more diverse normal defect samples across various quantum bug categories, which could enhance the model's ability to generalize and reduce bias.

```
from qiskit import QuantumRegister,
    ClassicalRegister, QuantumCircuit
qrx = QuantumRegister(3, 'x')
qry = QuantumRegister(2, 'y')
qrz = QuantumRegister(1, 'z')
cr = ClassicalRegister(4, 'c')
qc = QuantumCircuit(qrx,qry,qrz,cr)
qc.measure([qrx[1], qrx[2]], [cr[0], cr[1]])
qc.measure([4, 5], [2, 3])
qc.draw()
```

Listing 4. Code containing index and method call operations with integers.

**Finding 3.** We infer that involving more quantum software API knowledge, improving model robustness and mitigating bias dataset are potential directions to improve QDP-FSL.

## VI. Threats to Validity

In this section we present the threats to validity. Construct validity refers to the relationship between theory and observation. Conclusion validity is related to treatment and outcome. External validity is about the generalizability of the results.

### A. Construct Validity

The most significant construct validity is that whether our dataset and method could capture real defect and their characteristics in quantum software. In terms of the dataset, the positive samples are derived from Bugs4Q, which includes 42 reproducible buggy program, and thus these samples are reliable. The negative samples may contain threats to validity, they come from the LintQ dataset, which utilizes the GitHub search API to identify code with Qiskit imports. We discard the code components marked as clean but affected by potential factors that may lead to bugs in the dataset. To minimize the threat, we perform a sampled analysis on a set of results and find no bugs in the clean dataset. In terms of QDP-FSL, although the applied model is significantly different from static analysis based approach that use heuristic- and rule-based approaches that locate precisely the defects, our embeddings extracted from CodeBERT also represent semantics of quantum software code components, which is proved effective in SDP [25].

### B. Conclusion Validity

The major conclusion validity of our study is the small size of our dataset, particularly the few defective samples. However, these samples come from manual annotation based on a large scale dataset, and their corrections are also included. To our knowledge, there are few existing datasets of good quality. For example, the LintQ paper [22] detects "problems" in code, but they do not confirm whether they are actual bugs. Since software for quantum computing is still emerging, we expect more data to become available in the future, e.g., the

Quantum HumanEval dataset to be released by IBM [30], which will help us better evaluate QDP-FSL.

### C. External Validity

Our dataset only includes Qiskit programs. Qiskit is one of the leading frameworks for creating quantum software. However, we cannot guarantee that our method will work well in other contexts, such as predicting defects in Q# code. However, since the pre-trained model we use to generate code embedding is generalizable and performs well in C# and Java downstream tasks such as code translation [31], we believe that QDP-FSL is transferrable to Q# defect prediction.

## VII. Conclusion and Future Work

We proposed QDP-FSL to investigat the feasibility of predicting defects for Qiskit-based quantum software. CodeBERT is used as the pre-trained code model to represent code components, and an FSL approach called FEAT is used to address the issues brought by the imbalanced dataset. Result showed QDP-FSL outperforms static analysis approach.

In the future, QDP-FSL may be further improved and assessed by shifting more focus to quantum software specific implementations, improving model robustness and explainability, and testing QDP-FSL in quantum software implemented in other frameworks and programming languages.

## Acknowledgment

This work was supported in part by the China Postdoctoral Science Foundation (2024M761927), the Shanghai "Science and Technology Innovation Action Plan" Star Cultivation Program (Sailing Program, 24YF2719900, 24YF2720000), and the Science and Technology Project of State Grid Hebei Energy Technology Service Co., Ltd., China (TSS2020-10).

## References

[1] M. McDonald, R. Musson, and R. Smith, *The practical guide to defect prevention*. Microsoft Press, 2007.

[2] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[3] J. Wang, Q. Zhang, G. H. Xu, and M. Kim, "Qdiff: Differential testing of quantum software stacks," in *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 692–704, 2021.

[4] P. Zhao, J. Zhao, and L. Ma, "Identifying bug patterns in quantum programs," in *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pp. 16–21, 2021.

[5] G. Catolino, "Just-in-time bug prediction in mobile applications: The domain matters!," in *IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 201–202, 2017.

[6] C. Chai, G. Fan, H. Yu, Z. Huang, J. Ding, and Y. Guan, "Exploring better alternatives to size metrics for explainable software defect prediction," *Software Quality Journal*, vol. 32, no. 2, pp. 459–486, 2024.

[7] P. Zhao, Z. Miao, S. Lan, and J. Zhao, "Bugs4Q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs," *Journal of Systems and Software*, vol. 205, p. 111805, 2023.

[8] J. Deng, L. Lu, S. Qiu, and Y. Ou, "A suitable ast node granularity and multi-kernel transfer convolutional neural network for cross-project defect prediction," *IEEE Access*, vol. 8, pp. 66647–66661, 2020.

[9] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, "The impact of class rebalancing techniques on the performance and interpretation of defect prediction models," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1200–1219, 2020.

[10] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," in *17th International Conference on Mining Software Repositories (MSR)*, pp. 220–231, 2020.

[11] A. Wang, Y. Zhang, H. Wu, K. Jiang, and M. Wang, "Few-shot learning based balanced distribution adaptation for heterogeneous defect prediction," *IEEE Access*, vol. 8, pp. 32989–33001, 2020.

[12] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, "Just-in-time software defect prediction using deep temporal convolutional networks," *Neural Computing and Applications*, vol. 34, no. 5, pp. 3981–4001, 2022.

[13] A. B. Farid, E. M. Fathy, A. S. Eldin, and L. A. Abd-Elmegid, "Software defect prediction using hybrid model (cbil) of convolutional neural network (cnn) and bidirectional long short-term memory (bi-lstm)," *PeerJ Computer Science*, vol. 7, 2021.

[14] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2020.

[15] B. Boehm and V. Basili, "Top 10 list [software development]," *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[16] X. Li, Z. Sun, J.-H. Xue, and Z. Ma, "A concise review of recent few-shot meta-learning methods," *Neurocomputing*, vol. 456, pp. 463–468, 2021.

[17] L. Fei-Fei, R. Fergus, and P. Perona, "One-shot learning of object categories," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 594–611, 2006.

[18] L. Zhao, Z. Shang, L. Zhao, T. Zhang, and Y. Y. Tang, "Software defect prediction via cost-sensitive siamese parallel fully-connected neural networks," *Neurocomputing*, vol. 352, pp. 64–74, 2019.

[19] P. Zhao, X. Wu, J. Luo, Z. Li, and J. Zhao, "An empirical study of bugs in quantum machine learning frameworks," in *2023 IEEE International Conference on Quantum Software (QSW)*, pp. 68–75, 2023.

[20] P. Zhao, X. Wu, Z. Li, and J. Zhao, "QChecker: Detecting bugs in quantum programs via static analysis," in *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*, pp. 50–57, 2023.

[21] Q. Chen, R. Câmara, J. Campos, A. Souto, and I. Ahmed, "The smelly eight: An empirical study on the prevalence of code smells in quantum computing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 358–370, 2023.

[22] M. Paltenghi and M. Pradel, "Analyzing quantum programs with lintq: A static analysis framework for qiskit," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 95:1–95:23, July 2024.

[23] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented queries on relational data," in *30th European Conference on Object-Oriented Programming (ECOOP)*, vol. 56, pp. 2:1–2:25, 2016.

[24] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

[25] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 425–436, 2021.

[26] H.-J. Ye, H. Hu, D.-C. Zhan, and F. Sha, "Few-shot learning via embedding adaptation with set-to-set functions," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8805–8814, 2020.

[27] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *34th International Conference on Machine Learning (ICML)*, p. 1126–1135, 2017.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *31st International Conference on Neural Information Processing Systems (NIPS)*, pp. 6000–6010, 2017.

[29] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "The impact of automated feature selection techniques on the interpretation of defect models," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3590–3638, 2020.

[30] N. Dupuis, L. Buratti, S. Vishwakarma, A. V. Forrat, D. Kremer, I. Faro, R. Puri, and J. Cruz-Benito, "Qiskit code assistant: Training llms for generating quantum computing code," *CoRR*, vol. abs/2405.19495, 2024.

[31] S. Lu, D. Guo, S. Ren, and et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Neural Information Processing Systems Track on Datasets and Benchmarks*, vol. 1, 2021. https://tinyurl.com/5n8akszu.