Toward Automatically Completing GitHub Workflows

Antonio Mastropaolo
antonio.mastropaolo@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Fiorella Zampettti
f.zampetti@unisannio.it
Dept. of Engineering,
University of Sannio
Benevento, Italy, IT

Gabriele Bavota
gabriele.bavota@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Massimiliano Di Penta
dipenta@unisannio.it
Dept. of Engineering,
University of Sannio
Benevento, Italy, IT

ABSTRACT

Continuous integration and delivery (CI/CD) are nowadays at the core of software development. Their benefits come at the cost of setting up and maintaining the CI/CD pipeline, which requires knowledge and skills often orthogonal to those entailed in other software-related tasks. While several recommender systems have been proposed to support developers across a variety of tasks, little automated support is available when it comes to setting up and maintaining CI/CD pipelines. We present GH-WCOM (GitHub Workflow COMpletion), a Transformer-based approach supporting developers in writing a specific type of CI/CD pipelines, namely GitHub workflows.To deal with such a task, we designed an abstraction process to help the learning of the transformer while still making GH-WCOM able to recommend very peculiar workflow elements such as tool options and scripting elements. Our empirical study shows that GH-WCOM provides up to 34.23% correct predictions, and the model's confidence is a reliable proxy for the recommendations' correctness likelihood.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Continuous Integration and Delivery, GitHub workflows, Pre-Trained Models, Machine Learning on Code

ACM Reference Format:
Antonio Mastropaolo, Fiorella Zampettti, Gabriele Bavota, and Massimiliano Di Penta. 2024. Toward Automatically Completing GitHub Workflows. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Setting and maintaining a continuous integration and delivery (CI/CD) pipeline is crucial for any software project. Indeed, CI/CD contributes to enhancing software quality and developers' productivity [14], and to speed up release cycles [54]. Nevertheless, previous research has highlighted the challenges encountered by developers in setting up and maintaining CI/CD pipelines [13, 27, 46, 61, 62]. Despite the availability of modern CI/CD infrastructures and reusable assets (*e.g.,* GitHub actions), the intrinsic CI/CD requirements and underlying technology of a given project may still make this task hard [27, 61]. For example, this could be the case when a system needs to be deployed and tested on different operating systems or even embedded devices.

The aforementioned challenges entail the need for recommender systems helping developers in setting up and maintaining CI/CD pipelines. This is also supported by a study by Soroar *et al.* [46], reporting that ~60% of the 90 developers they surveyed encountered difficulties in automating workflows using GitHub actions.

It is worth mentioning that the possible solutions are somewhat similar to those related to automated code completion, where approaches have been defined either to provide suggestions about non-trivial, generic code elements (up to blocks) to be completed [18], or more specialized suggestions, *e.g.,* related to creating assertions [58], or repairing vulnerabilities [16, 22] and bugs [17, 33, 34].

That being said, helping developers in setting up a CI/CD pipeline poses unique challenges. Indeed, the structure a CI/CD pipeline mixes up very specific scripting elements (*e.g.,* related to configuring a server, downloading certain libraries, etc.) with some more recurring and regular reusable elements (*e.g.,* the actions in the case of GitHub), up to natural language elements. Also, CI/CD pipeline contain several extremely context-specific elements, such as paths of installation directories, or URLs of resources to download. This creates major challenges to the use of data-driven techniques for the automated recommendations of these elements.

This paper proposes GH-WCOM (GitHub Workflow COMpletion) an approach leveraging Transformer models [55] to provide automated completion of GitHub workflows. To develop (and train) GH-WCOM, we have leveraged the existing body of GitHub workflows starting from a dataset by Decan *et al.* [20].

To make a GitHub workflow completion possible, we have defined and implemented a multi-step pre-processing including an abstraction of the tokens for which their verbatim prediction would not be feasible (*e.g.,* a very specific path in a project) while still leaving to GH-WCOM the ability to recommend some very peculiar workflow elements such as tool options and other scripting elements. GH-WCOM can recommend GitHub workflow completions in different modes that mimic how a developer may implement the workflow, *i.e.,* (i) suggesting the next statement (a GitHub step), or (ii) helping to complete a job with implementation elements once the developer has defined, in plain English, what the job should do.

Summarizing, this paper makes the following contributions:

(1) We propose GH-WCOM, which, to the best of our knowledge, is the first approach to automatically complete CI/CD pipelines, and GitHub workflows in particular.
(2) We experiment with different pre-trainings, abstraction levels, and completion scenarios. Results indicate that pre-training at least on English text is required, and GH-WCOM's performance for correct prediction is ~34%. The correct prediction accuracy is correlated with the model's confidence.
(3) We report a qualitative analysis discussing the extent to which the recommendations provided by GH-WCOM could still be helpful also when the generated output is different from the target (expected) one. Also, we discuss how GH-WCOM is competitive with respect to recent, popular general-purpose recommenders based on large language models, *e.g.,* CoPilot [2] and ChatGPT [1].
(4) We made publicly available GH-WCOM scripts, checkpoints predictions, and the used datasets [6].

Example of GitHub Workflow

YAML Representation

```yaml
name: CBuild

on:
  push:
     branches: [ main ]
  pull_request:
     branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    container:
      image: gcc
  steps:
   - name: checking out the repository
     uses: actions/checkout@v2
   - name: Running makefile to compile the program
     run:  make
```

JSON-like Representation

```
{
  "name": "CBuild",
  "on": {
    "push": {
      "branches": [
        "main"
      ]
    },
    "pull_request": {
      "branches": [
        "main"
      ]
    }
  },
  "jobs": {
    "build": {
      "runs-on": "ubuntu-latest",
      "container": {
        "image": "gcc"
      }
    },
    "steps": [
      {
        "name": "checking out the repository",
        "uses": "actions/checkout@v2"
      }
    …
  }
}
```

Figure 1: GitHub workflow example

2 BACKGROUND

GitHub workflows integrate CI/CD in the GitHub infrastructure. A GitHub workflow (example in the top part of Fig. 1, while the bottom part will be described later in the paper) is a YAML file located under the `.github/workflows` (sub)directory of a repository. As specified by the `on:` clause, a workflow is triggered based on some events (*e.g.,* a push, a pull request) and executes a series of jobs, specified after the `jobs` keyword (as the job named `build` in the figure).

Jobs are units of execution of a CI/CD process and can run in parallel or sequentially (if dependencies between jobs are specified) on runners. Unless they use explicit ways to exchange information (*e.g.,* uploading and downloading artifacts in a storage area), jobs are independent of each other. Runners can be local or remote virtual machines or containers. Runners and containers are specified after the job name, using the `runs-on` clause, and, if containers are used, the `container:` and `image` clauses. The job in the example runs on an Ubuntu virtual machine and uses a container from an image bringing the *gcc* compiler. Each job consists of a sequence of steps. In Fig. 1, steps are all items preceded by a dash following the keyword `steps`. There are two ways to implement a step. The first (denoted by the keyword `uses`) is to leverage GitHub actions, *i.e.,* reusable applications available on GitHub that implement recurring tasks. For example, the `actions/checkout@v2` is version 2 of an action checking the content of the GitHub repository branch on which the workflow has been triggered.

The second (keyword `run`) consists of directly executing whatever application is available in the virtual machine/container (*e.g.,* `apt-get` to install components, `gradle` to run a Gradle build). Run steps are typically used for specific tasks for which an action is not available, or the task is so simple as to not require an action. Optionally, a step can be documented with a textual description of its action or run command, using the `name` keyword. Further information about GitHub workflows and actions is available on the GitHub documentation [4].

3 GH-WCOM

This section describes GH-WCOM, the proposed approach to recommend GitHub workflow completions. GH-WCOM leverages the Text-to-Text Transfer Transformer (T5) model by Raffel *et al.* [44]. First, we pre-train T5 by experimenting with different strategies. Then, we train the tokenizer needed by GH-WCOM and, after an hyperparameter calibration, we fine-tune T5 with instances specifically related to the actual prediction tasks. After that, we use the trained model for two different kinds of predictions, *i.e.,* (i) adding the next step in a workflow job, or (ii) completing a job whose steps have just been specified in terms of natural language text.

In the following, after overviewing the T5 model, we describe the different steps of the approach.

3.1 An overview of T5

T5 [44] is an encoder-decoder Transformer [55] designed to work in a text-to-text setting. Whatever the generation task is, T5 can be employed as long as both the input and the output can be represented as textual strings (*e.g.,* translating from English to Spanish, outputting the fixed version of a provided buggy code). We have chosen T5 given its successful application in several code completion/generation tasks [18, 39, 52, 57].

The training procedure of T5 is usually performed in two steps. First, the model is pre-trained on a large-scale dataset using self-supervised training. The pre-training provides T5 with general knowledge about the language(s) of interest. For example, assuming the will of building an English-to-Spanish translator, we could provide as an input to the model English and Spanish sentences having 15% of their tokens masked, with the model in charge of predicting them. That makes the pre-training fully self-supervised.

Subsequently, the model undergoes fine-tuning, which is supervised training (*e.g.,* providing pairs composed of an English sentence and its Spanish translation). Fine-tuning specializes the model for the task of interest.

Raffel *et al.* experimented with five T5 variants, differing in terms of the number of trainable parameters: small, base, large, 3 billion, and 11 billion. Considering our computational resources and recent successful application of $T5_{small}$ to automate code-related tasks [18, 39, 52, 57], we opted for the simplest architecture which still features 60M trainable parameters, consistently with large language models used in the literature. For additional architectural details, we point the reader to the work by Raffel *et al.* [44].

3.2 Abstraction

We conjecture (and will later experiment) that learning to autocomplete GitHub workflows on raw text (*i.e.,* with no preprocessing) is extremely challenging. This is mainly due to the presence of context-specific (and often unique, *i.e.,* they have not been seen before) elements in the workflows, such as `paths` and `urls`. For example, the left part of Fig. 2 shows a GitHub workflow featuring elements such as the `./vendor/bin/phpunit` path or the specific version of an action the user is using (*e.g., actions/checkout@v2*), which are likely to hinder the completion learning. These are some of the elements we aim at abstracting with special tokens (*e.g.,* replacing a path with the `<PATH>` tag), as it can be seen in the right part of Fig. 2.

Such an abstraction moves the definition of these context-specific elements from T5 (now only in charge of indicating the need for *e.g.,* a `<PATH>`) to the developer. We acknowledge that this might imply a slightly higher effort on the developer's side who needs to "fill the placeholders" (*i.e.,* the special tags) in the prediction.

To define the abstraction rules, we leverage the unique set of tokens extracted from the workflows of the projects listed in the GitHub actions dataset by Decan *et al.* [21]. The dataset features 67,870 GitHub repositories, 29,778 of which use GitHub workflows, and is the one we use to create our training and testing datasets as described in Section 3.3. Given the list in that dataset, we were able to clone 69,040 GitHub repositories, which is more than the 67,870 for which Decan *et al.* extracted workflow data. From those, we retrieved all GitHub workflows and extracted their "tokens". A token can be an action name, a command to run, the option of a command, a path, etc. Out of 10,188,342 unique tokens, 284,463 appear in one workflow, *i.e.,* are very specific, confirming our conjecture about the need for abstraction. We randomly selected 1,000 of those tokens for manual inspection. We clustered them based on their "type" (*e.g.,* `path`, `file`). Such a process has been performed by the first author, with the results checked by three other authors. Such a process led to the definition of five categories of context-specific tokens we aim at abstracting: `url` (*i.e.,* a reference to a web resource, such as an IP address), `file` (*i.e.,* a file name/path), `path` (*i.e.,* a path to a directory or to any other resource which cannot be identified as a file since lacking extension), `version number`, (*i.e.,* the specific version of a library, language, or other resources being used), and `action version` (*i.e.,* the specific version of an action that is used). For each category, we defined a special token acting as a placeholder during the abstraction. Note that we distinguish between `version number` and `action version` since we assume this could provide additional information to the model which might be useful for the learning.

The abstraction example reported in Fig. 2 shows how we replace the `action version` of the token `actions/checkout@v2` with the special `<PLH>` token, while `files` and `urls` such as `bin/install-wp-test.sh` and `127.0.0.1` are replaced with ⟨`FILE`⟩ and ⟨`URL`⟩, respectively. The code implementing our abstraction process is publicly available [6]. In a nutshell, we use regular expressions and heuristics to identify the token types of interest and abstract them. The identification of `files` leverages, besides a regular expression, a list of extensions we defined during the manual analysis of the tokens appearing in a single workflow. Such a list is also provided in our replication package [6].

To validate our choice of the specific tokens to abstract, we extracted all single-occurring tokens in our dataset, namely those certainly representing problematic cases for any data-driven technique. In total, we identified 23,273 distinct single-occurring tokens. Out of these: 8,226 (37%) are `paths`, 8,068 (35%) are `files`, 2,833 (12%) are `urls`, and 2,334 (10%) are `versions`. This means that ∼93% of single-occurring tokens are abstracted by our procedure. This indicates that the proposed abstraction strategy is suitable to abstract rarely-occurring tokens.

3.3 Training and Testing Datasets

3.3.1 Pre-training dataset. Since the goal of pre-training is to provide T5 with general knowledge about the language(s) of interest, we built a pre-training dataset featuring `YAML` files (*i.e.,* the language used in GitHub workflows), and in particular both general-purpose `YAML` files as well as those implementing GitHub actions. The former are used for various purposes, *e.g.,* CI/CD scripts for other infrastructures (*e.g.,* Travis-CI) or other configuration files.

GitHub actions feature a syntax closer to workflows and therefore would provide further knowledge during pre-training.

We collected general-purpose `YAML` files in two steps. First, we searched for `YAML` files in the 69,040 GitHub repositories we cloned, while excluding those implementing GitHub workflows that we will use to fine-tune the model (*i.e.,* those contained in the `./github/workflows/` directory). This resulted in 443,037 general-purpose `YAML` files.

Workflow Raw Tokens

```
name: PHPUnit

on:
  push:
    branches:
      - develop
      - trunk
    paths:
      - '**.php'
  pull_request:
    branches:
      - develop

jobs:
  phpunit:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - uses: getong/mariadb-action@v1.1

    - name: Set PHP version
      uses: shivammathur/setup-php@v2
      with:
        php-version: '7.4'
        coverage: none
        tools: composer:v1

    - name: Install dependencies
      run: composer install

    - name: Setup WP Tests
      run: bash bin/install-wp-tests.sh
           wordpress_test root '' 127.0.0.1

    - name: PHPUnit
      run: './vendor/bin/phpunit'
```

Workflow Abstracted Tokens

```
name: PHPUnit

on:
  push:
    branches:
      - develop
      - trunk
    paths:
      - '<FILE>'
  pull_request:
    branches:
      - develop

jobs:
  phpunit:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout<PLH>

    - uses: getong/mariadb-action<PLH>

    - name: Set PHP version
      uses: shivammathur/setup-php<PLH>
      with:
        php-version: '<V_NUMBER>'
        coverage: none
        tools: composer:v1

    - name: Install dependencies
      run: composer install

    - name: Setup WP Tests
      run: bash <FILE> wordpress_test
           root '' <URL>

    - name: PHPUnit
      run: '<PATH>'
```

Figure 2: Example of Raw and Abstracted Instance.

To further expand this dataset, we cloned all public non-forked repositories having at least 100 stars and 100 commits, and created in the time range that goes from 2022-25-01 (*i.e.,* the day after Decan *et al.* built their dataset) to 2022-30-09 (the day in which we performed the data collection). The identification of these repositories has been performed using the GitHub search platform by Dabić *et al.* [5].

We successfully cloned additional 1,124 GitHub repositories that are not in the dataset by Decan *et al.* nor are forks of those. To create the pre-training dataset, which counts a body of 146,006 general-purpose YAML files, we excluded duplicated instances as well as those including non-ASCII tokens and all those having $\#tokens \geq 1024$. Fixing an upper-bound in terms of the number of tokens for the model's input helps in taming the computational cost of training and is a common practice in the literature exploiting DL models to automate code-related tasks [18, 25, 37, 38, 53, 56].

Concerning the YAML files implementing GitHub actions, we collected 13,638 unique examples about the usage of actions from the GitHub Marketplace [3].

The pre-training dataset features 146,066 general-purpose YAML files and 13,638 YAML files implementing GitHub actions. Each instance in the dataset is a pair featuring (i) a YAML file with 15% of its tokens randomly masked, and (ii) the expected target, namely the tokens the model is expected to predict instead of the masked ones.

3.3.2 Fine-tuning dataset. Our fine-tuning dataset features 73,708 GitHub workflows from the whole body of GitHub projects made available by Decan *et al.* [21]. On top of those, we mined 733 workflows from the 1,124 GitHub repositories previously mentioned.

We removed duplicated workflows, and, as done before, all those having $\#tokens \geq 1024$, instances containing non-ASCII characters, and those which overlap with instances in the pre-training dataset. We were left with 17,935 unique workflows used to train and evaluate GH-WCOM. These workflows feature an average of 54 lines (median=41) and 120 tokens (median=84).

We split the dataset into training (80%), validation (10%), and test (10%), making sure that all the instances coming from the same project are assigned to the same subset, thus avoiding leakage of data among the three sets. We obtained 14,348 workflows to train the models, 1,793 for hyperparamenter tuning, and 1,794 to test the best configuration identified. Each workflow is represented as a JSON-like object preserving the structure of the original workflow file, as it can be seen in the bottom part of Fig. 1.

We then fine-tune GH-WCOM to support two workflow completion scenarios. In the first one, *next step* (NS_{task}), GH-WCOM is in charge of predicting the complete n^{th} step a developer is likely to write in a workflow given the preceding already written tokens. A step may or may not contain a textual description (*name*), and it can either consist of action invocations (*uses*) or commands (*run*). In the second scenario, job completion (JC_{task}), GH-WCOM gets as input an abstract job where only *names* are specified, and it is asked to complete it step by step. Fig. 3 helps in better understanding these two scenarios by depicting a fine-tuning instance from our dataset.

Since we experiment with both the raw workflow version (*i.e.,* no abstraction) and with its abstracted version, we report in Fig. 3 an example of "raw instance". The left part of the figure ❶ shows the original GitHub workflow, while ❷ depicts its version for fine-tuning the model for NS_{task}. In this case, we are simulating a scenario in which the developer already wrote the first 11 lines of the workflow (*i.e.,* up to `steps:`), and GH-WCOM is asked to predict the first step of the job (*i.e.,* `uses: actions/checkout@v2`). Note that we can extract multiple (5) training instances from this workflow. Indeed, we can ask the model to predict the first step of the job given just the preceding statements.

Then, we can ask the model to predict the second step also given the definition of the first step, etc. Fig. 3 ❸ depicts a fine-tuning instance for JC_{task}. In this case, we assume that the developer wrote the skeleton of a job by only defining, when available, the job's name it should feature (*e.g., Yarn install*). The model is in charge to predict the step masked with the `<TO_BE_PREDICTED>` token, while the `<FOR-LATER-USE>` token is used to indicate steps that are not yet implemented. Also in this case we can build multiple fine-tuning instances from the workflow in Fig. 3. We can start predicting the first step in a job using the following $n-1$ for which only the name is provided; then, we can predict the second step, providing the model with the full implementation of the first (as if the model already predicted it) and the following partially defined $n-2$ as context; etc.

Table 1 reports the number of instances in the training, validation, and test datasets for both completion scenarios.

Table 1: Number of instances in the used datasets

Dataset	train	eval	test
Pre-training	159,645	-	-
Fine-tuning: NS_{task}	108,900	13,009	13,630
Fine-tuning: JC_{task}	108,900	13,009	13,630

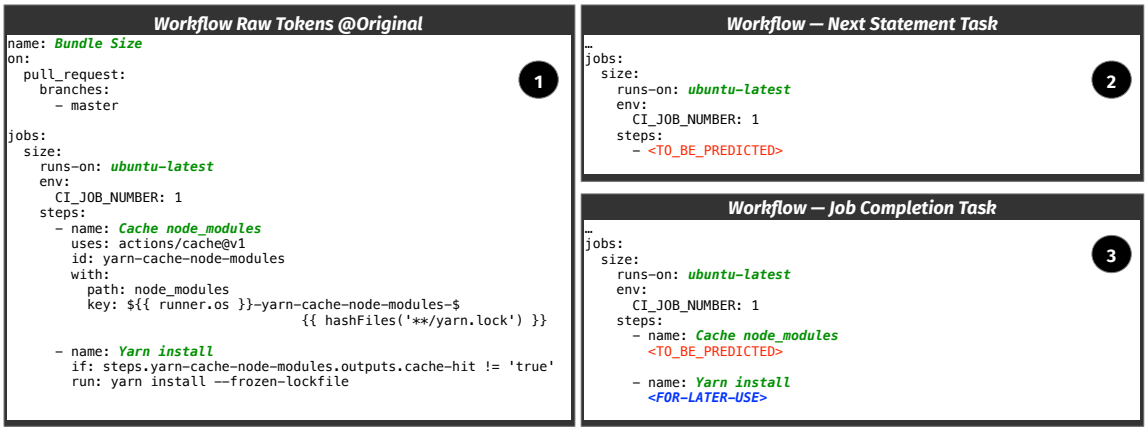

Figure 3: Example of instance for fine-tuning the T5 model on both tasks, namely NS_{task} and JC_{task}

3.4 Training and Hyperparameter Tuning

All the trainings we performed have been run using a Google Colab's 2x2, 8 cores TPU topology with a batch size of 32 and an input and target sequence length of 1,350 and 750 tokens, respectively.

3.4.1 Tokenizer Training. Since our task is characterized by the presence of natural language and human-readable data-serialization language (*i.e.,* YAML data), we trained a new tokenizer (*i.e.,* a SentencePiece model [31] with vocabulary size set to 32k word-pieces) to cope with context-specific elements. To this extent, we use the 159,645 YAML files included in our pre-training dataset and 712,634 English sentences from the C4 dataset [44]. The latter is a common practice in literature when developing DL-based models that are required to deal with multi-modal data such as code and technical natural language [39, 57]. We included English sentences due to the presence of technical English occurring within GitHub workflows.

3.4.2 Pre-training strategies. We assess GH-WCOM in four pre-training scenarios. The first is *No pre-training* ($T5_{NO-PT}$), in which the model is not pre-trained, but directly fine-tuned. This means that the model has no previous knowledge of any language and it is just trained to complete GitHub workflows with the available fine-tuning dataset composed by ~109k instances. The second is *YAML pre-training* ($T5_{YL}$), in which the model is first pre-trained for 300k steps on a total of 159,645 YAML files including 13,638 actions from the GitHub Marketplace [3] and then fine-tuned on the workflow completion task. Thus, in this case the model has knowledge of the general structure of YAML files before being then specialized on the completion task. The third is the *Natural Language Pre-training* ($T5_{NL}$), for which we fine-tune the publicly available checkpoint by Raffel *et al.* [7] which has been pre-trained for 1M steps on English sentences from the C4 dataset [44].

The fourth scenario is *Natural Language+YAML Pre-training* ($T5_{NL+YL}$) in which we further pre-trained the previously mentioned checkpoint for additional 300k steps on YAML files, reaching a total of 1,3M pre-training steps (1M on English sentences + 300k on YAML files).

3.4.3 Hyperparameter Tuning. Once pre-trained the models, we fine-tune the hyperparameters of the model following the same procedure employed by Mastropaolo *et al.* [40].

In particular, we assessed the performance of T5 when using four different learning rate schedulers: (i) *Constant Learning Rate* (C-LR): the learning rate is fixed during the whole training; (ii) *Inverse Square Root Learning Rate* (ISR-LR): the learning rate decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate [29]* (ST-LR): the learning rate first linearly increases and then linearly decays to the starting learning rate; and (iv) *Polynomial Decay Learning Rate* (PD-LR): the learning rate has a polynomial decay from an initial value to an ending value in the given decay steps. The exact configuration of all the parameters used for each scheduling strategy is reported in our replication package [6]. Such a procedure has been performed for each of the fine-tuning datasets previously described (*i.e.,* both tasks on raw and abstracted code).

Having four different training scenarios, four possible learning rates, two different completion contexts, and two versions of the fine-tuning dataset (*i.e.,* abstracted and raw tokens), the hyperparameter tuning required building and evaluating 64 models. We fine-tuned each model (*i.e.,* each configuration) for 100k steps. Then, we compute the percentage of correct predictions (*i.e.,* cases in which the model can correctly generate a recommendation) in the evaluation set. Table 2 reports the achieved results for each of the 64 models we fine-tuned to find the best-performing configuration (which is reported in boldface).

3.4.4 Fine-tuning. Once identified the best learning rates to use, we fine-tuned the final models using early stopping to avoid overfitting. In particular, we save checkpoints every 10k steps using a delta of 0.01, and a patience of 5. This means training the model on the fine-tuning dataset and evaluating its performance on the evaluation set every 10k. The training procedure stops if a gain smaller than the delta (0.01) is observed at each 50k step interval and the best-performing checkpoint up to that training step is selected. Complete data about this process is available in our replication package [6].

Table 2: Hyperparameters tuning results

	Raw		Abstracted	
No Pre-training				
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	11.06%	19.24%	13.27%	26.73%
Inverse Square Root (ISQ-LR)	**12.38%**	**21.13%**	**14.21%**	**27.86%**
Slanted Triangular (ST-LR)	10.13%	20.95%	12.81%	26.65%
Polynomial Decay (PD-LR)	10.86%	19.01%	13.78%	25.57%
YAML Pre-training				
	Raw		Abstracted	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	**16.26%**	25.92%	19.05%	32.35%
Inverse Square Root (ISQ-LR)	15.77%	25.47%	18.93%	31.22%
Slanted Triangular (ST-LR)	14.26%	24.73%	18.05%	30.96%
Polynomial Decay (PD-LR)	16.15%	**26.01%**	**19.24%**	**32.81%**
English Pre-training [44]				
	Raw		Abstracted	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	18.35%	27.18%	22.25%	34.02%
Inverse Square Root (ISQ-LR)	18.36%	27.10%	21.70%	33.91%
Slanted Triangular (ST-LR)	17.67%	26.61%	21.70%	33.25%
Polynomial Decay (PD-LR)	**18.46%**	**27.47%**	**22.30%**	**34.12%**
YAML+English Pre-training				
	Raw		Abstracted	
	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Constant (C-LR)	18.06%	27.40%	21.55%	32.91%
Inverse Square Root (ISQ-LR)	**18.36%**	**28.17%**	**21.84%**	**34.62%**
Slanted Triangular (ST-LR)	16.50%	25.90%	18.88%	32.11%
Polynomial Decay (PD-LR)	18.28%	27.33%	21.40%	33.36%

3.5 Generating Predictions

After the model has been trained, we can generate predictions for the task we aim at supporting using different decoding schema. To this end, we opted for a greedy decoding strategy [47] that generates the recommendation, by selecting at each decoding step the token with the highest probability of appearing in a specific position. Thus, a single prediction is generated for an input sequence.

4 STUDY DESIGN

The *goal* of our study is to evaluate GH-WCOM. The *quality focus* is GH-WCOM's ability to provide correct predictions, as well as predictions that, while differing from the ground truth, could still be valuable for developers. We focus on the two completion scenarios previously described: NS_{task} (mimicking a top-down coding adopted by the developer when writing the workflow statement by statement), and (ii) JC_{task} (helping the developer to complete a job with implementation elements given its textual description). The context consists of the test datasets summarized in Table 1.

The study aims at answering the following research questions:

RQ_1: *How does GH-WCOM perform with different pre-training strategies?* RQ_1 assesses the impact of using different pre-training strategies when completing workflows. We experiment with four pre-training strategies, including the lack of pre-training.

RQ_2: *How does GH-WCOM perform for different prediction scenarios?* RQ_2 tests GH-WCOM in different prediction scenarios, *i.e.,* next statement and job-level contextual completion with and without abstraction. We also implement a statistical language model used as a baseline for comparison.

RQ_3: *To what extent "wrong" recommendations provided by GH-WCOM can be leveraged by developers?* RQ_3 gauges the extent to which "wrong" predictions (*i.e.,* recommendations different from the expected output) can still be useful to developers and thus worth being integrated into CI/CD pipelines after minor changes.

4.1 Data Collection and Analysis

To address RQ_1, we run the best-performing configuration for each pre-training strategy and scenario (NS_{task} and JC_{task}) against the test sets (Table 1). Then, we compute the percentage of correct predictions, namely cases in which the models can synthesize completions identical to the expected target (*i.e.,* the code written by developers). We further assess the quality of the predictions generated using different pre-training strategies by relying on NLP (Natural Language Processing) metrics such as BLEU [43] and ROUGE [35].

BLEU score (Bilingual Evaluation Understudy) [43] measures how similar the candidate (predicted) and reference (oracle) texts are. Given a size n, the candidate and reference texts are broken into n-grams and the algorithm determines how many n-grams of the candidate text appear in the reference text. The BLEU score ranges between 0 (the sequences are completely different) and 1 (the sequences are identical). We use the BLEU-4 variant as did in previous software engineering papers [52, 56, 59].

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques [35]. ROUGE metrics compare an automatically generated summary or translation with a set of reference summaries (typically, human-produced). We use the ROUGE-L which computes the length of the longest common subsequence between a generated and a reference sentence.

To answer RQ_2, we first select the best-performing models when supporting the completion of GitHub workflow with and without abstraction in both predictions scenario (NS_{task} and JC_{task}). Later, we assess the quality of the predictions using the same set of metrics (*i.e.,* correct predictions, BLEU, and ROUGE score) adopted in RQ_1. As there is no previous approach to compare GH-WCOM against, we implemented a baseline leveraging an n-gram model which is a specific actualization of a large class of techniques that assign probabilities to sequences of tokens (*i.e.,* Statistical-Language-Model [23]). To train such a model we use the same set of instances used to fine-tune GH-WCOM without, however, any masked part. We experimented with three different values of n (*i.e.,* n=3, n=5, and n=7), with $n-1$ being the number of tokens on which the prediction of the next token is based upon. The best value for n ($n = 3$) has been found by running the models on the evaluation sets (results in our replication package [6]).

The best model has then been run on the same test sets used for GH-WCOM's assessment. We do not compare GH-WCOM against the n-gram when job-level information is provided (JC_{task}), since, by construction, such a technique would not leverage the additional knowledge provided (*i.e.,* it only "looks" at the tokens preceding the ones to predict). To explain how predictions are generated with the 3-gram model, let us assume we are completing a piece of workflow having five tokens T, of which the last two are masked (M): $\langle T1, T2, T3, M4, M5\rangle$. We provide, as input to the model, T2 and T3 to predict M4, obtaining the model prediction P4. Then, we use T3 and P4 to predict M5 obtaining the predicted sentence $\langle T1, T2, T3, P4, P5\rangle$. While GH-WCOM autonomously decides when to stop predicting tokens, this is not the case for the n-gram model in our usage scenario. We thus defined two heuristics to stop generating tokens. First, we stop when the n-gram model does not generate any output token given the preceding n-1.

Second, we rely on the format in which we represent the instances in our datasets: Each instance is a JSON object and we trained all models to generate as output `{target}`, where the two delimiting curly brackets are the result of our JSON-like representation. Thus, we stop generating tokens when we reach a fully-balanced (*i.e.,* valid) JSON object for the test instance to predict (*i.e.,* the *n*-gram generated the "closing" curly bracket and the latter does not close a curly bracket opened in the predicted code but the JSON-related one).

We complement the quantitative evaluation by performing statistical tests aimed at assessing whether GH-WCOM produces better recommendations as compared to the baseline. We use the McNemar's test [41] (with is a proportion test for dependent samples) and Odds Ratios (ORs) on the correct predictions both approaches (*i.e.,*GH-WCOM and *n*-gram) can generate when evaluated in the NS_{task} completion scenarios, working with both abstracted and raw tokens. We also statistically compare the distribution of the BLEU-4 (computed at statement level) and ROUGE, assuming a significance level of 95% and using the Wilcoxon signed-rank test [60]. The (paired) Cliff's Delta (d) is used as effect size [24] and it is considered: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [24]. Due to multiple comparisons for both statistical tests, we adjust p-values using Holm's correction procedure [28].

As for RQ_3, we perform a twofold analysis. We first assess whether the confidence of the model in the generated predictions can be used as a reliable proxy of their "quality". T5 provides a *score* for each generated prediction which represents the log-likelihood of the prediction. For example, having a log-likelihood of -2 means that the prediction has a likelihood of 0.69 ($ln(x) = -2 \implies x = 0.69$). The likelihood can be interpreted as the confidence of the model about the correctness of the prediction on a scale from 0.00 to 1.00 (the higher the better). We split the predictions generated by T5 into ten buckets at steps of 0.1 (*i.e.,* the lowest confidence scenario groups the predictions having confidence between 0.0 and 0.1, the highest from 0.9 to 1.0) and report the percentage of correct and wrong predictions within each bucket. Then, given the positive results we achieved (as we will show, the confidence values are representative of the prediction quality), we randomly sample 384 cases of wrong predictions having a confidence ≥0.70, with 384 representing a statistically significant sample with a confidence level of 95% and confidence interval of ±5%.

Each sample has been manually classified by two authors with one of the following labels:

(1) A minor change is required to make the suggestion usable, *e.g.,* change an option or a value;
(2) GH-WCOM has recommended the correct action/script command, yet with wrong arguments;
(3) GH-WCOM has recommended the correct action/script command, yet with the wrong name;
(4) The suggestion is completely wrong, *i.e.,*GH-WCOM recommendation is completely different from the ground truth.

In the labeling, the two involved authors achieved a Cohen's kappa [19] of 0.72, indicating a *substantial agreement* when measuring inter-rater reliability for categorical items.

Conflicts, which occurred for 17.97% of inspected samples, have been solved through open discussion among the authors.

We report the percentage of predictions assigned to each label and discuss qualitative examples of wrong predictions which, however, might still be valuable for developers.

5 STUDY RESULTS

RQ_1: How does GH-WCOM perform with different pre-training strategies? The results obtained by fine-tuning T5 using different pre-training strategies are presented in Table 3. The table shows the model's performance in terms of correct predictions, BLEU-4, and ROUGE-LCS (F-measure). The best model for a given combination of task (*i.e.,* NS_{task} and JC_{task}) and evaluation metrics is reported in boldface. As expected, the $T5_{NO-PT}$ is outperformed by all pre-trained models, with 11.23% and 19.74% correct predictions for the NS_{task} and JC_{task} task, respectively, when working on raw code. When abstracting the dataset, the correct predictions for the $T5_{NO-PT}$ model improve—14.14% for NS_{task} and 26.96% for JC_{task} —while remaining the worst configuration.

Table 3: Comparison among different pre-training strategies in terms of correct predictions, BLEU-4 and ROUGE-LCS (f-measure) computed at corpus level

Dataset	PT-Strategy	Correct predictions		BLEU 4		ROUGE-LCS	
		NS_{task}	JC_{task}	NS_{task}	JC_{task}	NS_{task}	JC_{task}
Raw	$T5_{NO-PT}$	11.23%	19.74%	13.70%	13.80%	44.0%	54.75%
	$T5_{YL}$	15.85%	24.51%	14.50%	24.10%	50.09%	61.20%
	$T5_{NL}$ [7]	**17.47%**	26.02%	**23.10%**	**29.60%**	**51.78%**	63.34%
	$T5_{NL+YL}$	17.33%	**26.35%**	16.40%	27.70%	51.74%	**63.58%**
Abstracted	$T5_{NO-PT}$	14.14%	26.98%	20.40%	24.20%	46.31%	59.92%
	$T5_{YL}$	19.81%	32.58%	13.80%	17.0%	53.30%	64.88%
	$T5_{NL}$ [7]	21.28%	33.84%	**28.40%**	**25.90%**	55.30%	66.51%
	$T5_{NL+YL}$	**21.36%**	**34.23%**	21.80%	18.40%	**55.37%**	**66.54%**

The results with pre-training (also) involving English documents ($T5_{NL}$ and $T5_{NL+YL}$) are always the best or the second-best in class, with performance very close to each other. Noteworthy, the usefulness of pre-training on English text when dealing with software-related tasks has been already documented in the literature [50] and is likely due to the vast presence of English terms in the code. Both $T5_{NL}$ and $T5_{NL+YL}$ models achieve the best performance on the abstracted workflows, with a percentage of correct predictions of around 21% for the NS_{task} task and 34% for the JC_{task} task.

Two observations can be made here. First, in the JC_{task} task, T5 is more successful thanks to the additional context provided before triggering the prediction (*i.e.,* the skeleton of the job defined by the developer—see Section 3.3.2).

Second, the abstraction seems to substantially boost the model's performance, with ~4% of additional correct predictions for the NS_{task} task and ~8% in the JC_{task} task.

Table 4 statistically compares the correct predictions achieved using the four different pre-training strategies for the two tasks and the two workflow representations (raw and abstract). Confirming what was said above, the performance of $T5_{NL}$ and $T5_{NL+YL}$ is always significantly better (adjusted p-value < 0.001) compared to the non-pre-trained models ($T5_{NO-PT}$) and to the ones pre-trained using `YAML` files only ($T5_{YL}$), with ORs going from 1.49 up to 4.88. The difference between $T5_{NL}$ and $T5_{NL+YL}$ is never statistically significant, showing that the two models are almost equivalent. This is an important finding because it means that an English pre-trained model can be simply fine-tuned to successfully accomplish the task (this is way less demanding than retraining the model).

Table 4: Effect of different pre-training strategies on performance: results of McNemar's test.

Dataset	Task	Comparison	*p*-value	OR
Raw Tokens	NS_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	4.88
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.95
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.50	1.05
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.96
	JC_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	3.60
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.59
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.10	0.88
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.74
Abstracted Tokens	NS_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	3.98
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.75
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.69	0.96
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.88
	JC_{task}	$T5_{NL}$ vs. $T5_{NO-PT}$	<0.001	3.78
		$T5_{NL}$ vs. $T5_{YL}$	<0.001	1.49
		$T5_{NL}$ vs. $T5_{NL+YL}$	0.05	0.86
		$T5_{NL+YL}$ vs $T5_{YL}$	<0.001	1.70

The analysis of the BLEU and ROUGE metrics shown in Table 3 confirms the above-described finding, *i.e.,* pre-training always helps, in particular when leveraging English sentences.

Answer to RQ$_1$. The pre-training boosts the performance of GH-WCOM. Pre-training with English text (possibly along with YAML files) helps to achieve the best performance.

In the following RQs we leverage the model pre-trained on English text and `YAML` files as the backbone of GH-WCOM.

Table 5: GH-WCOM vs 3-gram model when generating recommendations for the NS_{task}

Dataset	Comparison	Metric	*p*-value	d	OR
Raw tokens	GH-WCOM vs. *n*-gram	Correct Predictions	<0.001	-	17.69
		BLEU-4	<0.001	0.51 (L)	-
		ROUGE-LCS	<0.001	0.52 (L)	-
Abstracted tokens	GH-WCOM vs. *n*-gram	Correct Predictions	<0.001	-	13.76
		BLEU-4	<0.001	0.49 (L)	-
		ROUGE-LCS	<0.001	0.50 (L)	-

RQ$_2$: How does GH-WCOM perform for different prediction scenarios? Fig. 4 depicts the results achieved by GH-WCOM and the best-performing *n*-gram model (3-gram) in terms of correct predictions, BLEU-4 and ROUGE-LCS. Due to the technical limitations of the *n*-gram (*i.e.,* it only considers the $n-1$ preceding tokens when generating a prediction), such a comparison has been performed only for the NS_{task} task.

Table 5 reports the results of the statistical comparison between the two in terms of adjusted *p*-value and OR (for correct predictions) and effect size (for BLEU and ROUGE). On both datasets, GH-WCOM achieves statistically significant better results than the baseline for all metrics. When looking at the correct predictions the gap is of ~11% on the raw dataset (5.10% *vs* 17.33%) and ~12% on the abstracted dataset (9.28% *vs* 21.36%). The OR is 17.69 (raw) and 13.76 (abstract). An OR of 13.76 indicates ~13 times higher odds of obtaining a correct prediction using GH-WCOM. Even the comparisons in terms of BLEU and ROUGE show the superiority of GH-WCOM both visually (Fig. 4) and statistically (Table 5).

GH-WCOM achieves its best performance for the JC_{task} task, with 34.23% of correct predictions (see Table 3), benefiting from the additional contextual information provided as input. Truly, one may question the usefulness of an approach that fails 66% of the times. Nevertheless, as a term for comparison, the DL-based approach recently proposed by Ciniselli *et al.* [18] for block-level *Java* completion achieved ~27% of correct predictions.

Answer to RQ$_2$. GH-WCOM outperforms the *n*-gram baseline for the NS_{task} task on all the considered metrics. The gap in correct predictions is >11% on both the raw and the abstracted dataset. The best performances are achieved for the JC_{task} task (~34% of correct predictions) thanks to the additional contextual information provided as input.

RQ$_3$: To what extent "wrong" recommendations provided by GH-WCOM can be leveraged by developers? Fig. 5 depicts the relationship between the percentage of correct and wrong predictions when considering their confidence. Due to space limitations, we only focus our discussion on the most challenging scenario, namely NS_{task} , as the findings for JC_{task} are similar (complete results in [6]). The orange line shows the percentage of correct predictions within each confidence interval, *e.g.,* 68.45% of predictions having confidence between 0.8 and 0.9 are correct when working with the raw code. In contrast, the red line shows the percentage of wrong predictions within each confidence bucket. Fig. 5 shows a clear relationship between the confidence of the predictions and their likelihood of being correct. For example, out of the 1,076 predictions generated with confidence >0.9 in the abstracted dataset, 959 (89.13%) are correct.

This result has an important practical implication: By setting a threshold on confidence, it would be possible to filter out recommendations likely to be false positives and only notify the developer when the model is quite confident about the generated prediction. As previously said, the results for the JC_{task} are in line with those discussed for NS_{task} . For example, 89.03% of the 2,908 predictions having confidence >0.9 are correct in the abstracted dataset. A similar percentage is achieved on the raw dataset (89.13%).

Concerning the manual analysis of a sample of 384 completions "wrongly" predicted by GH-WCOM (*i.e.,* the prediction did not match the expected target), we found that: (i) 41.41% (159) are actually wrong, since the predicted code would implement a different behavior than the ground-truth; (ii) in 25.52% (98) of the cases, GH-WCOM suggested the correct action/script command yet with wrong arguments; (iii) 28.13% (108) of predictions would require minor changes, implying, on average, changing (*i.e.,* insertion and/or deletion) ~11 characters in the recommended output in order to align with the ground truth; and (iv) 4.95% (19) feature a wrong or missing action name, *i.e.,* just missing documentation. While the complete results of our manual inspection are available in our replication package [6], Fig. 6 shows two concrete examples of the instances we inspected. The left part of Fig. 6 ❶ shows an example in which the whole step is correctly predicted, with the exception of the name which is different from the expected one (`Set up Python` *vs* `Python`) but still meaningful.

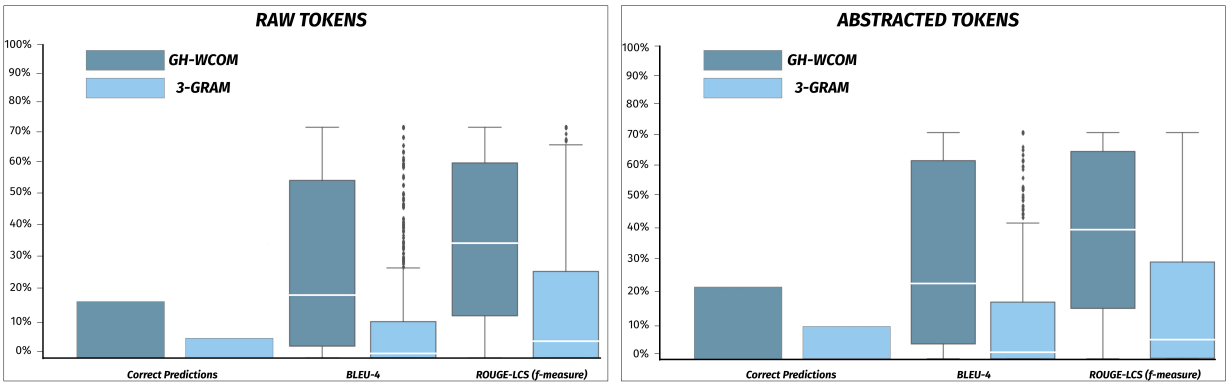

Figure 4: Results achieved by GH-WCOM and the n-gram model when predicting actions for NS_{task}

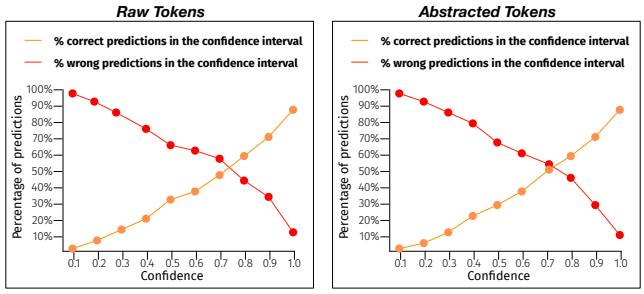

Figure 5: Correct and wrong predictions by the confidence of GH-WCOM when generating recommendations for NS_{task}

The right part ❷ depicts a case in which the only difference between the predicted and the expected step is the version of a specific action to use (`@v2` *vs* `@v3`). In both cases, the developer is still likely to benefit from the prediction.

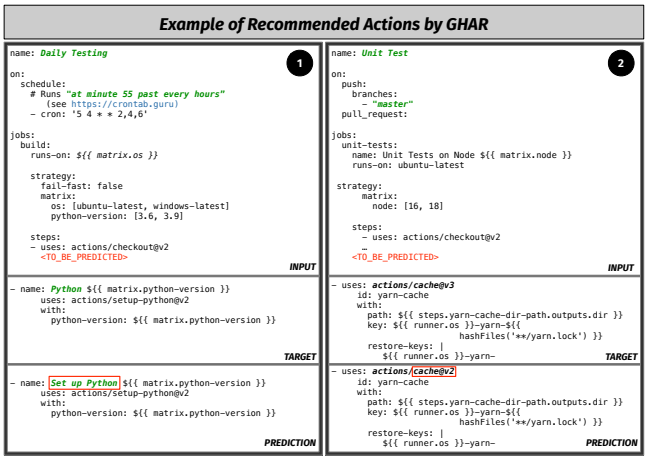

Figure 6: Examples of GH-WCOM's recommended actions extracted from the manual investigation we performed

Answer to RQ$_3$. The confidence of the predictions can serve as a trustworthy indicator of their correctness when auto-completing GitHub workflows; ~50% of predictions differing from the expected target but on which the model has high confidence could still be valuable for developers.

5.1 Why not just using a state-of-the-art chatbot or code recommender?

Large Language Models (LLMs) have opened up new possibilities even in the field of software engineering. One such application is GitHub Copilot [2], developed by Microsoft using the OpenAI Codex model. Copilot is a state-of-the-art tool for recommending code completion and generation tasks. Similarly, OpenAI's ChatGPT [1] showed remarkable performance in generating human-like text responses to prompts, even for code-related tasks.

We conducted a study to investigate the potential of these techniques for supporting auto-completion in GitHub workflows. We tested both tools on 60 instances in our test set by randomly selecting: (i) 15 workflows with the highest confidence score for which GH-WCOM provided correct predictions; (ii) 15 workflows with the highest confidence score for which GH-WCOM failed to provide meaningful recommendations; (iii) 15 workflows with the lowest confidence score for which GH-WCOM provided correct predictions; and (iv) 15 workflows with the lowest confidence score for which GH-WCOM failed to provide meaningful recommendations.

Concerning the high-confidence scenario, GitHub Copilot was able to provide correct recommendations for 7 of the 15 instances successfully predicted by GH-WCOM. For 2 instances, Copilot did not suggest any token, and for 6 instances, it provided incorrect recommendations. In contrast, when it came to the 15 instances for which GH-WCOM generated incorrect recommendations, Copilot correctly recommended only 2 of them and failed to provide meaningful recommendations for the remaining 13. Regarding ChatGPT, we observed that, out of the 15 instances correctly predicted by GH-WCOM, the chatbot can only suggest 4 meaningful GitHub workflow completions, while providing incorrect action elements/scripts for the remaining 11 instances.

We then tested ChatGPT on the instances where GH-WCOM failed, we found that for 13 out of 15 workflows, the recommended actions were incorrect, and, for 2 instances, ChatGPT was unable to respond to our query.

As for the GH-WCOM low-confidence instances, also Copilot and ChatGPT poorly performed on such instances. For the 15 successful predictions generated by GH-WCOM, Copilot succeeds in only 4 and ChatGPT in only 3 of them. Copilot and ChatGPT also fail in all 15 cases for which GH-WCOM provides a wrong output.

6 THREATS TO VALIDITY

Construct validity. One potential threat arises from the collection of our dataset, as we excluded workflows longer than 1,024 tokens. As mentioned earlier, it is a common practice to limit the input size of DL models to manage training complexity effectively. We recognize that using different thresholds could yield varying results, and we acknowledge this as a potential limitation.

Another concern involves the extent to which the masking is representative of what programmers do during their tasks [26]. We have simulated two scenarios, NS_{task} and JC_{task} , representative of when developers write steps sequentially or code them after sketching their documentation. To evaluate the quality of the predictions, we used consolidated measures such as the percentage of correct predictions, BLEU-4 [43, 45], and ROUGE score. Furthermore, we complemented such measures qualitative analyses.

In an attempt to help the model learning, we employed an abstraction schema in which five types of tokens are abstracted with special placeholders. The goal of our abstraction process was to identify a sort of upper-bound for the capabilities of our approach in a *best case* scenario, in which all tokes being *e.g.,* a path would be replaced with the same ⟨PATH⟩ placeholder. Such a simplification pushes more effort on the developer's side while, however, simplifying the learning, and thus representing an upper bound in terms of prediction performances (with the lower bound represented by the raw predictions). We acknowledge that alternative (and less extreme) solutions are possible; for example, distinct paths appearing within the same workflow could be abstracted with different placeholders (*e.g.,* ⟨PATH1⟩, ⟨PATH2⟩) with the model expected to use the same placeholder for related paths (*i.e.,* the same path appearing multiple times in the workflow). As part of our upcoming work agenda, we anticipate conducting user studies to assess different abstraction techniques as alternatives.

Internal validity. One key issue for DL models is the hyperparameter tuning, which we detailed in Section 3.3.2. We are aware that we could not consider all possible (combinations of) values for that. Also, the performances of a T5 model could largely depend on how it has been pre-trained. To mitigate this threat, we have shown how GH-WCOM works by leveraging different pre-trainings.

Conclusion validity. To address the RQs, wherever appropriate we use suitable statistical tests (McNemar's test and Wilcoxon signed rank test) as well as effect size measures (OR and Cliff's delta). In the qualitative analysis of RQ_4, we computed and reported Cohen's kappa inter-rater agreement.

External validity. We experiment GH-WCOM with a $T5_{small}$ model. We acknowledge that our choice of the specific model architecture to use could affect the generalizability of our findings.

For example, larger T5 versions [44] could lead to different performance. We performed a minimal check of how scaling up the model could affect our findings. To this aim, we trained a $T5_{base}$ model [44] using the $T5_{NL+YL}$ setting and the same training process used for $T5_{small}$: We further pre-trained the publicly released $T5_{base}$ checkpoint (pre-trained on natural language) for 300k steps on YAML files and then fine-tuned it on the GitHub workflows. We used the same learning rate scheduler used for $T5_{base}$ (*i.e.,* ISQ-LR). The achieved results show that scaling up the model size from 60M to 220M parameters yields negligible improvements in comparison to $T5_{small}$. When employing a $T5_{base}$ architecture to recommend actions in the most demanding scenario (NS_{task}), the difference in correct predictions is a +0.18% (21.54%) and a +0.47% (17.80%) for the raw and abstracted datasets, respectively. When incorporating contextual information into the model (JC_{task}), similar conclusions arise (up to +0.67% of correct predictions). Furthermore, while we applied GH-WCOM for GitHub workflow completion, with proper training/fine-tuning, GH-WCOM could be applied to CI/CD pipelines developed with different technologies, *e.g.,* Jenkins or GitLab.

7 RELATED WORK

We discuss literature on automated code completion (which has commonalities with GitHub workflow auto-completion. In particular, we discuss work about task-oriented and pre-trained models.

7.1 Task-Oriented models for Completing Code

Li *et al.* [32] introduce a pointer mixture network improving the accuracy of predicting Out-of-Vocabulary (OoV) words. The pointer mixture network can determine whether to create a word within the vocabulary using an RNN component or reconstruct an OoV word based on the local context using a pointer component.

Alon *et al.* [8] propose a language-agnostic approach for code completion which uses the syntax to model a code snippet as a tree. Their model predicts the next token in a partial expression represented by an AST, achieving an exact match accuracy of 18.04%.

Chen *et al.* [12] focus on recommendaing APIs. Their approach employs a DL technique integrating structural and textual code information with the use of an API context graph and code token network. Their model outperforms existing graph-based statistical and tree-based DL methods for API recommendation.

Avishkar *et al.* [10] propose a neural language model suggesting code in Python using a sparse pointer network to capture long-range relationships among identifiers. Aye and Kaiser [9] introduce a new language model that predicts the next top-k tokens while taking into account real-world constraints, including prediction latency, model size and memory usage, and suggestion validity. Svyatkovskiy *et al.* [49] propose a learning-to-rank approach for code completion, which is cheaper in terms of memory footprint than generative models.

7.2 Pre-trained Models for Code Completion

Svyatkovskiy *et al.* [48] introduce IntelliCode, a multilingual code completion tool that predicts sequences of arbitrary token types using subtokens to overcome the OoV problem [51].

Liu *et al.* [36] propose a pre-trained Transformer incorporating two tasks: (i) program understanding and (ii) code generation. The model has been fine-tuned to predict the next code token to write.

Kim *et al.* [30] use the Transformer architecture by incorporating the syntactic structure of the code to further advance the state-of-the-art next-token prediction by margins ranging from 14% to 18% when compared to previous techniques.

Ciniselli *et al.* [18] examine the effectiveness of Transformer-based models in completing code with varying degrees of complexity. T5 results to be the best model for recommending code completion across different complexities, with an accuracy of ~29% when predicting entire code blocks.

Our work shares with the aforementioned ones, and in particular with the one by Ciniselli *et al.* [18], the use of transformer architectures, and T5 in particular. That being said, unlike many source code artifacts, a GitHub workflow features several elements that are extremely project-specific, *e.g.*, dependencies, configuration files, hardware and software configurations to be tested. As detailed in Section 3.2, this has required a complex abstraction process. Last, but not least, the completion scenarios are different from the ones for the source code. For the former one mainly wants to generate the next statement, block, or code construct. For the latter, elements to generate are either job steps (combinations of natural language descriptions, actions, and script calls) or the implementation of a job specified in terms of its names.

LLMs such as GPT-3 [11] or GPT-4 [42] have propelled code completion techniques to new heights. GitHub Copilot [15] is a prime example of this advancement in the field. On a similar note, OpenAI in November 2022 released ChatGPT [1], which showcased remarkable abilities even when dealing with code-related tasks. While we did not use LLMs for feasibility and parsimony reasons, yet, we provide some evidence showing that GitHub workflow completion is a challenging task for them as well. Also, GH-WCOM can be evolved to replace T5 with LLMs featuring billions of parameters.

8 CONCLUSION AND FUTURE WORKS

This paper tackled the problem of automatically completing CI/CD pipeline scripts, and, in particular, GitHub workflows. We proposed GH-WCOM, an approach based on T5 [44] pre-trained models to automatically recommend workflow completions in different scenarios, *i.e.,* predicting the next step (NS_{task}), or filling a workflow job given its textual documentation, *i.e.,* the `names` (JC_{task}).

Our empirical analysis found that (i) leveraging a pre-training involving English text (possibly complemented by `YAML` files) always helps, (ii) the performance of best models range from 17.47% (NS_{task} task) and 26.35% (JC_{task} task) for raw correct predictions, to 21.36% (NS_{task}) and 34.23% (JC_{task}) for abstracted correct predictions; and (iii) the model confidence correlates with the likelihood of generating a correct prediction. Finally, GH-WCOM is competitive for context-sensitive completion tasks when compared to LLM-based tools such as CoPilot [2] and ChatGPT [1].

Future work aims to experiment with alternative DL models, and, possibly, incorporate developers' feedback in the GH-WCOM's learning (*e.g.*, by using reinforcement learning).

ACKNOWLEDGMENTS

Mastropaolo and Bavota are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720). Mastropaolo thanks CHOOSE for sponsoring his trip to the conference. Zampetti and Di Penta are supported by the Horizon 2020 (EU Commission) project COSMOS (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS.

REFERENCES

[1] [n. d.]. ChatGPT https://openai.com/blog/chatgpt.
[2] [n. d.]. GitHub Copilot https://copilot.github.com.
[3] [n. d.]. GitHub Marketplace https://github.com/marketplace?type=actions.
[4] [n. d.]. GitHub workflows. https://docs.github.com/en/actions/using-workflows Last accessed Feb 16, 2023.
[5] [n. d.]. MSR mining platform. https://seart-ghs.si.usi.ch.
[6] [n. d.]. Replication Package https://github.com/antonio-mastropaolo/GH-WCOM.
[7] [n. d.]. T5 public checkpoint gs://t5-data/pretrained_models/small.
[8] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International Conference on Machine Learning*. PMLR, 245–256.
[9] Gareth Ari Aye and Gail E Kaiser. 2020. Sequence Model Design for Code Completion in the Modern IDE. *arXiv preprint arXiv:2004.05249* (2020).
[10] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* (2016).
[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[12] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. 2021. Holistic combination of structural and textual code information for context based API recommendation. *IEEE Transactions on Software Engineering* (2021).
[13] L. Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software* 32, 2 (2015), 50–54.
[14] Lianping Chen. 2017. Continuous Delivery: Overcoming adoption challenges. *Journal of Systems and Software* 128 (2017), 72 – 86.
[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[16] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2023. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Trans. Software Eng.* 49, 1 (2023), 147–165.
[17] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959.
[18] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Trans. Software Eng.* 48, 12 (2022), 4818–4837. https://doi.org/10.1109/TSE.2021.3128234
[19] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
[20] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 235–245.
[21] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 235–245.
[22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 935–947.
[23] Yoav Goldberg. 2017. *Neural network methods in natural language processing*. Morgan & Claypool Publishers.
[24] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers.

[25] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 300–310.

[26] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When code completion fails: a case study on real-world completions. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 960–970.

[27] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017*.

[28] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[29] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).

[30] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[31] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[32] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[33] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[34] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[35] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[36] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*. Association for Computing Machinery.

[37] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2021. An Empirical Study on Code Comment Completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170.

[38] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2022. Automated Variable Renaming: Are We There Yet? *arXiv preprint arXiv:2212.05738* (2022).

[39] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using Deep Learning to Generate Complete Log Statements. *arXiv preprint arXiv:2201.04837* (2022).

[40] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[41] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[42] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[43] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[44] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[45] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020). https://arxiv.org/abs/2009.10297

[46] Sk Golam Saroar and Maleknaz Nayebi. 2023. Developers' Perception of GitHub Actions: A Survey Analysis. *arXiv preprint arXiv:2303.04084* (2023).

[47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[48] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[49] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. 2020. Fast and Memory-Efficient Neural Code Completion. arXiv:2004.13651 [cs.SE]

[50] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 54–64.

[51] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[52] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. *arXiv preprint arXiv:2201.06850* (2022).

[53] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174.

[54] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *ESEC/SIGSOFT FSE*. ACM, 805–816.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[56] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–30.

[57] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[58] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1398–1409.

[59] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[60] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[61] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. 2022. Problems and solutions in applying continuous integration and delivery to 20 open-source cyber-physical systems. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 646–657.

[62] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25 (2020), 1095–1135.