COSMOS

DevOps for Complex Cyber-physical Systems

Project Number 957254

D3.4 Automated bad practice resolution recommender for CPS

Version 1.0
30 June 2023
Final

Public Distribution

University of Sannio

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

Project Partner Contact Information

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

Document Control

Version	Status	Date
0.1	Document outline	15 January 2023
0.2	Introduction and Design draft	20 February 2023
0.3	Results draft	15 March 2023
0.4	Tool Description	30 May 2023
1.0	Reviewed version	30 June 2023

Table of Contents

Executive Summary

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 require 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 a Transformer-based approach supporting developers in writing a specific type of CI/CD pipelines, namely GitHub or GitLab workflows, yet the tool can be trained and applied to other CI/CD pipelines.

As shown in this delivery, workflow completion is a *much harder* task than conventional code completion. This is mainly due to (i) the mix of specific scripting elements having completely different objectives (*e.g.,* configuring a server, downloading a library, etc.), and (ii) the presence of project-specific elements such as directory paths or URLs to download resources.

We deal with this complexity by designing an abstraction process to help the learning of the transformer while still making the proposed approach able to recommend very peculiar workflow elements such as tool options and scripting elements. Our empirical study conducted on GitHub actions shows that the proposed approach provides up to 34.23% correct predictions, and the model's confidence is a reliable proxy for the recommendations' correctness likelihood.

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 [15], and to speed up release cycles [61]. Nevertheless, previous research has highlighted the challenges encountered by developers in setting up and maintaining CI/CD pipelines [14, 32, 73]. Such challenges are exacerbated by the separation between the development and operation roles [25].

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 [32, 71]. Finally, challenges in setting up CI/CD pipelines have also been pointed in previous COSMOS deliverables, and in particular D3.2 [22] and a companion article [72].

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 aimed at helping developers in setting up and maintaining CI/CD pipelines. This need and its 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 [20], or more specialized suggestions, *e.g.,* related to creating assertions [67], or repairing vulnerabilities [17, 26] and bugs [18, 39, 40].

That being said, helping developers in setting up a CI/CD pipeline poses unique challenges. First, we conjecture that the structure and code of a CI/CD pipeline may be less regular and repetitive than conventional source code. This is because it 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. Second, a CI/CD pipeline contains several extremely context-specific elements. These are, for example, 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.

In this deliverable, we describe an approach leveraging Transformer models [63] to provide automated completion of GitHub and GitLab workflows. Given the availability of data, the experimentation has been conducted on GitHub data, yet the tool has been developed to also support GitLab workflows. To develop (and train) the tool, we have leveraged the existing body of GitHub workflows starting from a dataset by Decan *et al.* [23]. 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 the tool the ability to recommend some very peculiar workflow elements such as tool options and other scripting elements.

The proposed tool 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. Similarly, the tool can recommend GitLab completion by recommending how to complete (the next) scripts for a given job.

2 Background: GitHub and GitLab CI/CD Jobs

To better understand how the tool work, in the following we briefly outline the structure of the GitHub and GitLab CI/CD workflows.

2.1 Structure of GitHub Workflows

```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
```

Listing 1: GitHub workflow example

GitHub workflows integrate CI/CD in the GitHub infrastructure. A GitHub workflow (example in Listing 1) 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 Listing 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].

2.2 Structure of GitLab Workflows

Listing 2 shows an example of a GitLab CI/CD workflow. As the figure shows, GitLab workflows have significant differences with respect to GitHub workflows.

The first important difference is that, while GitHub provide reusable assets to execute commands (*i.e.,* actions), GitLab does not. Essentially, all commands are executed by invoking operating system commands or scripts.

```yaml
image: gradle:alpine

variables:
  GRADLE_OPTS: "-Dorg.gradle.daemon=false"

before_script:
  - export GRADLE_USER_HOME=`pwd`/.gradle

stages:
  - build
  - test

build-job:
  stage: build
  script:
  - gradle compileJava

unit-test-job:
  stage: test
  script:
  - echo "Running unit tests..."
  - gradle test
```

Listing 2: GitLab workflow example

This has also a consequence on the extent to which an automated completion tool for GitLab may effectively work, and the challenges that can be encountered in training a completion model for GitLab.

Other than that, a GitLab workflow has the following characteristics:

- It is based on a Docker image, specified at the beginning of the workflow using the `image:` keyword.
- Jobs are defined using arbitrary names. Jobs are recognized because they are top level elements in the workflow structure, and must contain a `script` clause. Unless they are contained in a staged build, jobs are executed in parallel, independently from each other.
- Script clauses specify the commands to be executed in a job. For example, in Listing 2, the script of the *build-job* job executes the command `gradle compileJava`, whereas the *unit-test-job* script has two commands in sequence, an `echo` and a `gradle test`.
- Jobs can be staged (*i.e.,* organized sequentially) by defining the stages through the `stages:` clause, where the stage names are defined. Such names must then be recalled inside each job using the `stage` keyword, to specify in what stage a job is being executed. For example, the *unit-test-job* job is executed in the *test* stage, that follows the *build* stage.

Further information about GitLab CI/CD workflows is available on the GitLab documentation [5].

3 Approach

This section describes the proposed approach to recommend CI/CD workflow completions. In the following, we will mainly describe the tool for what concerns GitHub workflow completions. That being said, as explained in the introduction, the tool has been developed to also support GitLab workflow completion. Finally, it can be easily adapt to work on other CI/CD workflow configuration scripts or, in principle, to support the completion of other configuration files.

The proposed tool leverages the Text-to-Text Transfer Transformer (T5) model by Raffel *et al.* [51]. First, we pre-train T5 by experimenting with different strategies. Then, we train the tokenizer needed by the proposed approach 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.

We start by overviewing the T5 model we exploit in the tool (Section 3.1). Then, we outline in Section 3.2 the abstraction schema we devised to help the model learning in the presence of context-specific elements to predict (*e.g.,* a file path used in a project). Section 3.3 details the procedure we used to build the datasets needed for training and testing the tool, as well as for the T5's hyperparameters tuning (Section 3.4). Finally, Section 3.5 explains how the model generates predictions once trained.

3.1 An overview of T5

T5 [51] is an encoder-decoder Transformer [63] 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 [45, 20, 59, 65].

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. Important to note is the self-supervised nature of the pre-training: We can automatically create as many pre-training instances as we wish by randomly masking tokens in sentences.

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 [45, 20, 59, 65], 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.* [51].

3.2 Abstraction

We conjecture (and will later experiment) that learning to autocomplete GitHub or GitLab workflows on raw text (*i.e.,* with no preprocessing) is extremely challenging, more than providing verbatim completions of source code. 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, Listing 3 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. Similar considerations (and similar abstraction heuristics) can be applied for GitLab, with the only difference that GitLab does not have actions.

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 Listing 4. Such an abstraction moves the definition of these context-specific elements from the T5 model (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.* [24]. 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 Listing 4 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. In a nutshell, we use regular expressions and heuristics to identify the token types of interest and abstract them. For example, to identify `path` tokens, we use the following regular expression:

```
^(.*/)?(?:$|(.+?)(?:(\.[^.]*$)|$))
```

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.

3.3 Training and Testing Datasets

We detail in this section the building of the pre-training and fine-tuning datasets used to train the tool for the GitHub experimentation.

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. 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.* [6]. 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 [29, 64, 43, 60, 44, 20].

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.* [24]. 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 that we use to train and evaluate the proposed approach. We split the dataset into training (80%), validation (10%), and test (10%), making sure that all the instances coming from the same

```yaml
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'
```

Listing 3: Example of Raw Instance for GitHub workflows.

```yaml
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>'
```

Listing 4: Example of Abstracted Instance for GitHub workflows.

```yaml
name: Bundle Size
on:
  pull_request:
    branches:
      - master
jobs:
  size:
    runs-on: ubuntu-latest
    env:
      CI_JOB_NUMBER: 1
  steps:
    - name: Cache node_modules
      uses: actions/cache@v1
      id: yarn-cache-node-modules
      with:
         path: node_modules
         key: ${{ runner.os }}-yarn-cache-node-modules-$
               {{ hashFiles('**/yarn.lock') }}
    - name: Yarn install
      if: steps.yarn-cache-node-modules.outputs.cache-hit
          != 'true'
      run: yarn install --frozen-lockfile
```

Listing 5: Example of instance for fine-tuning the T5 model: GitHub original workflow

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.

We then fine-tune the model to support two workflow completion scenarios. In the first one, namely *next step* (NS_{task}), the model 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, namely job completion (JC_{task}), the model gets as input an abstract job where only *names* are specified, and it is asked to complete it step by step. Listing 5, Listing 6, and Listing 7 help in better understanding these two scenarios by depicting a fine-tuning instance from our dataset.

Listing 5 and Listing 6 depict a fine-tuning scenario for NS_{task} . In this case, we are simulating a scenario in which the developer already wrote the first lines of the workflow (*i.e.,* up to `steps:`), and the proposed approach 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. Listing 5 and Listing 7 depict 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. In particular, 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

```
...
jobs:
  size:
    runs-on: ubuntu-latest
    env:
      CI_JOB_NUMBER: 1
    steps:
      - <TO_BE_PREDICTED>
```

Listing 6: Example of instance for fine-tuning the T5 model: next statement task for GitHub

```
...
jobs:
  runs-on: ubuntu-latest
  env:
    CI_JOB_NUMBER: 1
  steps:
    - name: Cache node_modules
      <TO_BE_PREDICTED>
    - name: Yarn install
      <FOR-LATER-USE>
```

Listing 7: Example of instance for fine-tuning the T5 model: job completion task for GitHub

```
unit-test-job:
  stage: test
  script:
    <TO_BE_PREDICTED>
```

Listing 8: Example of instance for fine-tuning the T5 model: job completion task for GitLab

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

3.3.3 Fine-tuning for GitLab

Similarly to the scenarios described in detail above for GitHub, it is possible to foresee some prediction scenarios for GitLab. For example, Listing 8 depicts a job completion scenario for the Gitlab workflow shown in Listing 2. In the case of GitLab, as the figure shows, the goal is to complete a `script:` environment in a job.

In this case, the training has been performed on jobs extracted from a dataset from a previous empirical work on GitLab workflows [62]. The dataset consists of 5,275 worflows, from which we generated a total of 116k training instances, *i.e.,* jobs to be completed.

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. Then, the models to be released with the tool—for both GitHub and GitLab—have been trained on a Lambda-VECTOR workstation, equipped with 3.50 GHz AMD Ryzen Threadripper PRO 3975WX 32-Cores, and four NVIDIA RTX A5000 GPUs.

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 [37] 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 [51]. 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 [45, 65]. We included English sentences due to the presence of technical English occurring within GitHub workflows.

3.4.2 Pre-training strategies

We assess the proposed approach in four pre-training scenarios:

(i) *No pre-training* ($T5_{NO-PT}$), in which the model is not pre-trained, but directly fine-tuned; (ii) *YAML pre-training* ($T5_{YL}$), the model is pre-trained for 300k steps on a total of 159,645 `YAML` files including 13,638 actions from the GitHub Marketplace [3]; (iii) *Natural Language Pre-training* ($T5_{NL}$); and (iv) *Natural Language+YAML Pre-training* ($T5_{NL+YL}$). For the former, we use the publicly available pre-trained model by Raffel *et al.* [51], thus, we do not perform any additional pre-training steps but we directly fine-tuned their latest available checkpoint [7]. As for the *English+YAML Pre-training* scenario, starting from the latest T5 public checkpoint [7], we further pre-train the model for 300k on the pre-training dataset we built, reaching 1,3M pre-training steps.

Once pre-trained, all models are fine-tuned and compared. This allows to assess the contribution to performance (if any) brought by the different pre-training strategies.

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.* [46]. 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 [34]* (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.

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.

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 [54] 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.

Table 2: Hyperparameters tuning results

No Pre-training				
	Raw		**Abstracted**	
	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 [51]				
	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%

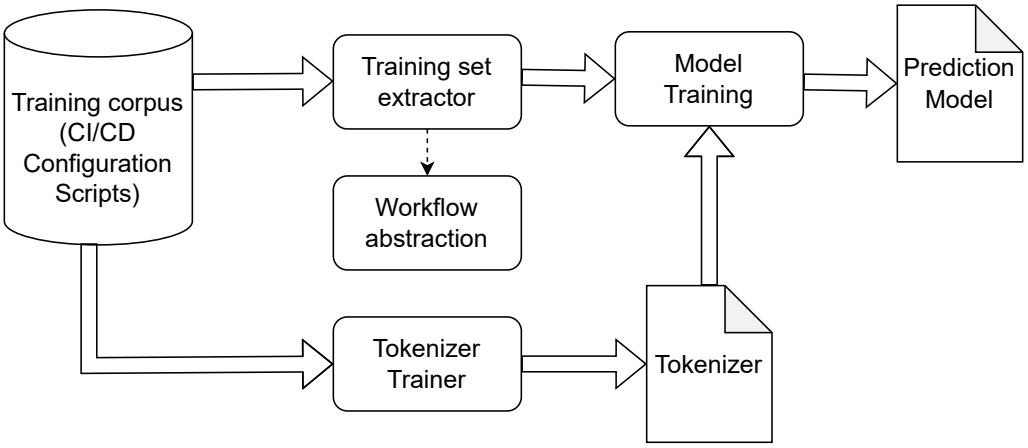

Figure 1: Training toolset infrastructure

4 Toolset Architecture

Fig. 1 and Fig. 2 depict the architecture of the model training toolset and of the inference API, described in the following. The whole infrastructure has been implemented in Python.

4.1 Model training toolset

Training set extraction. The first activity to be performed is the extraction of the training data. This is performed by the *Training set extractor*, which operates by searching for files with a given extension (*e.g.,* `.yml`) in a specified directory. If specified, the tool also leverages a workflow abstraction, which replaces

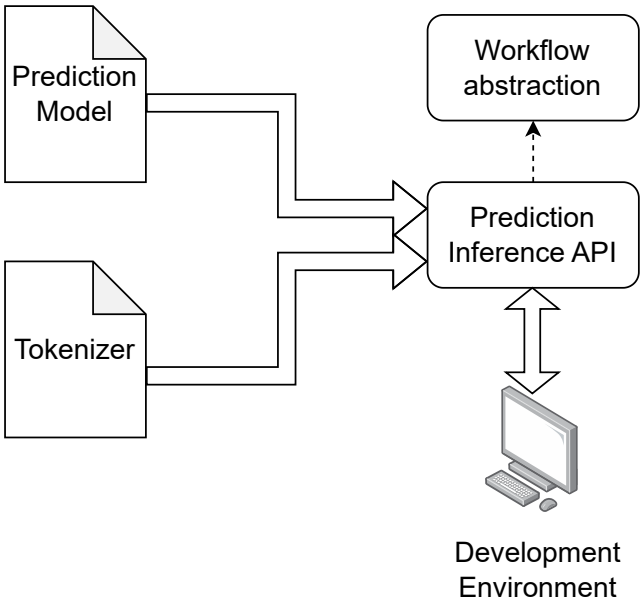

Figure 2: Inference API

context-specific (unlikely to be properly predicted) code elements with idiomized tokens. This can be done for several elements, of the workflows, namely:

- File names;
- URLs;
- Docker image names (for GitLab workflows);
- Variable names; and
- Action versions (for GitHub workflows).

The training set extraction and workflow abstraction leverage libraries such as *pyyaml* (for parsing YML files into Python data structures, and getting rid of comments), *json* (to convert the parsed YML into JSON documents), *BeautifulSoup* (to prune spurious HTML elements), other than the Python standard *re* library to leverage regular expressions during the abstraction phase. The training instances are saved in a JSON file, which can be then loaded by the *Model Training component*.

Tokenizer creation. To make the transformer model training working properly, it is adviceable to train a dictionary on a dataset that reflects the lexicon on the data on which the model is trained (and then used) as much as possible. The *Tokenizer Trainer* component takes as input either a plain textual file, or a directory containing files (*e.g.,* `.yml` files) to search, and creates a *Tokenizer* model that is then used by both the *Model Training* and by the inference component. The *Tokenizer Trainer* is based on the *sentencepiece* library, which allows to train tokenizers that can be then reused by deep learning models, and on the *T5TokenizerFast* from the Hugginface's *transformers* library [70].

Model Training. The *Model Training* component is the core of the training phase. It takes the training and evaluation set generated by the *Training set extractor*, and the *Tokenizer* to tokenize such data before seeding them into the deep learning model. Then, by using the hyperparameters configured by the user in a proper configuration file (see the user manual), the component trains the model, and, at the end of the training phases, saves it into a file, so that it can be deployed on a different machine for inference purposes. The training is based on the Hugginface's *transformers* library, and, above all, it uses the *AutoModelForSeq2SeqLM* component to create a transformer model. While we have performed our empirical evaluation using the T5-small model, it can be easily replaced with other (*e.g.,* bigger) models by simply modifying the configuration file.

It is important to note that the *Model Training* component is the only component for which it is extremely adviceable to leverage a server equipped with one or more GPUs. In principle, the tool adapts to the availability of GPUs or CPUs, yet it is unlikely that a CI/CD inference model could be realistically trained on CPUs.

4.2 Inference API

While we do not release a specific client for the tool (but just a running example that reads the text to be completed from the standard input and produces the recommendation on the standard output), we provide an *Inference API*, that can be simply imported and integrated in any program, *e.g.,* a development environment.

To properly work, the *Inference API* requires the *Tokenizer* and the *Prediction Model* produced in the training phase. Then, given a textual input (*i.e.,* a workflow to be completed), it produces a recommendation API. The Inference API can work on raw workflow source code (*i.e.,* by performing raw code completions), or by leveraging the *Workflow abstraction* component. Similarly to the *Model Training*, the inference API is based on the Huggingface's *transformers* [70] library, and, specifically, the *AutoModelForSeq2SeqLM* component. Differently than the *Model Training*, the inference does not require the availability of GPUs to properly work. Also in this case, however, the tool automatically adapts itself to the availability of GPUs, making the inference faster.

5 Empirical Evaluation

5.1 Study Design

The *goal* of our study is to evaluate the proposed approach for CI/CD workflow completion. The *quality focus* is the approach'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 difficult it is to automatically complete GitHub workflows as compared to other code completion tasks?* RQ_1 sheds light on the complexity of the GitHub actions completion task as compared to the classic task of code completion. The results of RQ_1 will help in interpreting our findings.

RQ_2: *How does the proposed approach perform with different pre-training strategies?* RQ_2 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_3: *How does the proposed approach perform for different prediction scenarios?* RQ_3 tests the proposed approach 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_4: *To what extent can "wrong" recommendations provided by the proposed approach be leveraged by developers?* RQ_4 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.

5.1.1 Data Collection and Analysis

To answer RQ_1 we use the token-level entropy [53] to compare the amount of information that two completion systems (one for code and one for workflows) are required to produce when recommending either code tokens

or workflow steps. Higher code entropy indicates a higher difficulty of the prediction task. Concerning the task of our interest (*i.e.,* workflow completion), we computed the token-level entropy from our dataset for all targets (*i.e.,* the steps that the model is expected to automatically generate).

To have a term of comparison that could help in interpreting the difficulty of the task, we also computed the entropy of the target predictions used in a recent DL-based code completion technique proposed by Ciniselli *et al.* [20] for block-level completion. This includes 636k instances in which the target prediction is a block of code (*i.e.,* a set of statements surrounded by curly brackets) which resemble our completion task on workflows.

We report how much information the proposed approach is asked to generate when working on both raw and abstracted tokens. In this way, we can also assess the complexity of our task in both scenarios (*i.e.,* with abstraction and without abstraction) and if the abstraction helps in reducing the task complexity.

To address RQ_2, 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 [50] and ROUGE [41].

BLEU score (Bilingual Evaluation Understudy) [50] 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 [64, 68, 59].

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques [41]. 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_3, 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_2. As there is no previous approach to compare the proposed approach 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 [27]). To train such a model we use the same set of instances used to fine-tune the proposed approach 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. The best model has then been run on the same test sets used for the approach's assessment. We do not compare the proposed approach against the n-gram model 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 the proposed approach 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. In the first, we stop when the n-gram model does not generate any output token given the preceding n-1. In the 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 experimented models to generate as output `{target}`, where the two delimiting curly brackets are the result of our JSON-like representation. Thus, in our second heuristic, 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 one related to the JSON).

We complement the quantitative evaluation by performing statistical tests aimed at assessing whether the proposed approach produces better recommendations as compared to the baseline. We use the McNemar's test [47] (with is a proportion test for dependent samples) and Odds Ratios (ORs) on the correct predictions both approaches (*i.e.,* the proposed approach and n-grams) 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 [69]. The (paired) Cliff's Delta (d) is used as effect size [28] 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 $|d| \geq 0.474$ [28]. Due to multiple comparisons for both statistical tests, we adjust p-values using Holm's correction procedure [33].

As for RQ_4, 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 $\geq$0.70, with 384 representing a statistically significant sample with a confidence level of 95% and confidence interval of $\pm 5\%$. 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. The proposed approach has recommended the correct action/script command, yet with wrong arguments;
3. The proposed approach has recommended the correct action/script command, yet with the wrong name;
4. The suggestion is completely wrong, *i.e.,* the approach's recommendation is completely different from the ground truth.

In the labeling, the two involved authors achieved a Cohen's kappa [21] 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.

5.2 Study Results

We start by answering RQ1-RQ4 (Section 5.1), presenting the complexity of generating meaningful GitHub actions as compared to *Java* statements. Then, we discuss the impact on performances of different pre-training strategies. Later, we outline the results that the proposed approach can achieve when used for completing workflow in a top-down manner (*i.e.,* NS_{task}) by contrasting its performances against an n-gram model we implement as a baseline. Finally, we discuss the reliability of the confidence as a proxy for the quality of the predictions while presenting at the same time the results of our manual investigation.

5.2.1 RQ_1: How difficult it is to automatically complete GitHub workflows as compared to other code completion tasks?

By computing the Shannon entropy [53] on the raw dataset (Table 1) we found that the proposed approach is required to produce 3.06 bits of information when recommending a complete GitHub step. As a comparison, predicting a block of *Java* code (*i.e.,* multiple contiguous statements surrounded by curly brackets) in the dataset by Ciniselli *et al.* [20] would require generating 2.07 bits of information. Thus, completing GitHub workflows is more challenging than completing *Java* code. Indeed, the lower entropy for the block-level *Java* completion indicates that the prediction outcome is more deterministic, *i.e.,* it comes with less "surprise", thus the lower entropy. As per the abstracted version of our dataset, we obtained an entropy value of 3.05 (-0.01 than for the raw dataset). This difference looks very small, but, as observed by Hellendorn *et al.* [30], given the dataset size, even minor differences in entropy can have a substantial impact on the prediction capabilities of language models.

Answer to RQ_1. Completing steps in GitHub workflows is more challenging than guessing *Java* code blocks. The abstraction process described in Section 3.2 helps to slightly reduce the prediction entropy, which however remains higher than for conventional *Java* code completion.

5.2.2 RQ_2: How does the proposed approach 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 [57] 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_2. The pre-training boosts the performance of the proposed approach. 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 the proposed approach.

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)	-

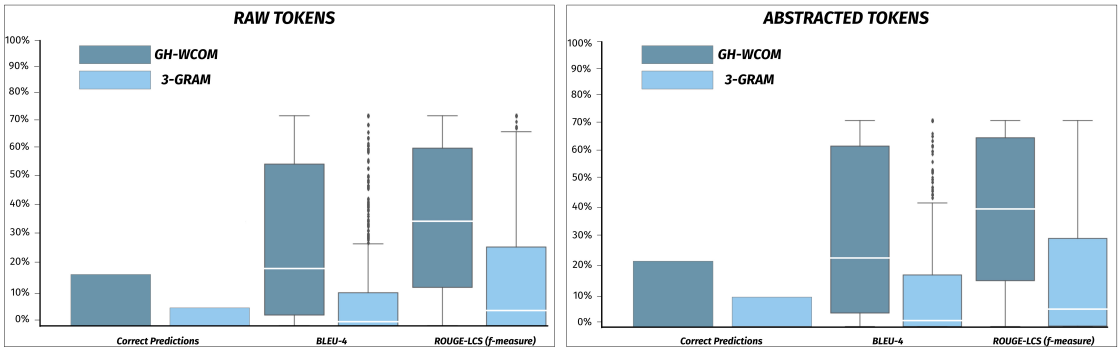

Figure 3: Results achieved by the proposed approach and the n-gram model when predicting actions for NS_{task}

5.2.3 RQ$_3$: How does the proposed approach perform for different prediction scenarios?

Fig. 3 depicts the results achieved by the proposed approach 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, the proposed approach achieves statistically significant better results than the baseline for all the considered metrics. When looking at the correct predictions the gap is of $\sim$11% on the raw dataset (5.10% *vs* 17.33%) and $\sim$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 $\sim$13 times higher odds of obtaining a correct prediction using the proposed approach. Even the comparisons in terms of BLEU and ROUGE show the superiority of the proposed approach both visually (Fig. 3) and statistically (Table 5).

The proposed approach 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 by Ciniselli *et al.* [20] for block-level *Java* completion achieved $\sim$27% of correct predictions and, as we showed in RQ$_1$, completing GitHub workflows is more difficult than code completion. Also, as we will show in RQ$_4$, it is still possible to build a reliable recommender system on top of the proposed approach by exploiting the confidence of its predictions.

Answer to RQ$_3$. The proposed approach outperforms the n-gram baseline for the NS_{task} task on all the considered metrics. The gap in correct predictions is $>$11% on both the raw and the abstracted dataset. The best performances are achieved for the JC_{task} task ($\sim$34% of correct predictions) thanks to the additional contextual information provided as input.

5.2.4 RQ$_4$: To what extent can "wrong" recommendations provided by the proposed approach be leveraged by developers?

Fig. 4 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. 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 (*e.g.,* 31.55% in the interval 0.8-0.9).

Fig. 4 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%).

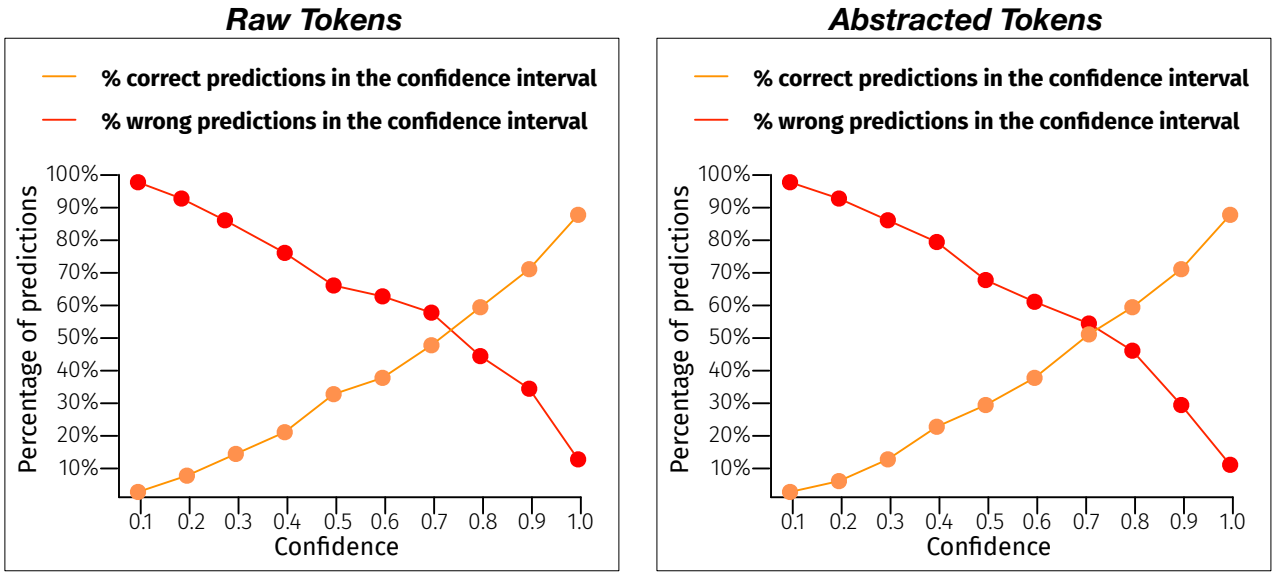

Figure 4: Correct and wrong predictions by the confidence of GH-WCOM when generating recommendations for NS_{task}

Concerning the manual analysis of a sample of 384 completions "wrongly" predicted by the proposed approach (*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, the proposed approach suggested the correct action/script command yet with wrong arguments, thus still providing "some" help to the developer; (iii) 28.13% (108) of predictions would require minor changes (*e.g.,* action version number); and (iv) 4.95% (19) feature a wrong or missing action name, *i.e.,* just missing documentation. Fig. 5 shows two concrete examples of the instances we inspected.

The left part of Fig. 5 (1) 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. The right part (2) depicts instead 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.

Answer to RQ$_4$. 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.

5.2.5 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

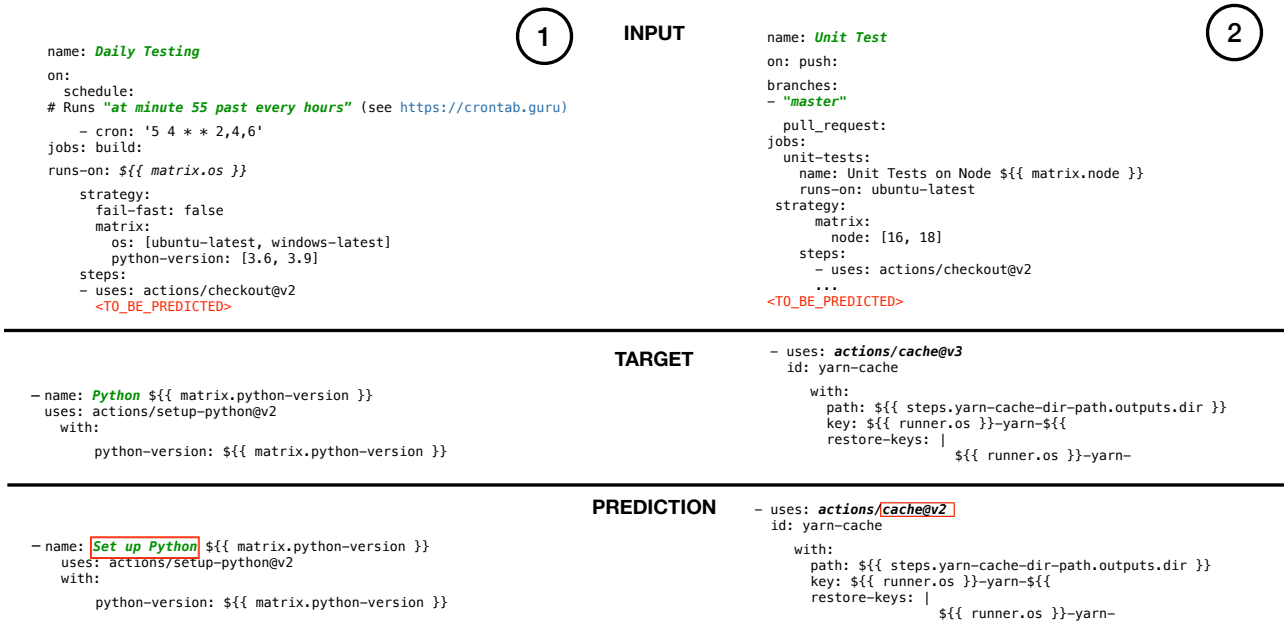

Figure 5: Examples of recommended actions extracted from the manual investigation we performed

tasks. We conducted a study to investigate the potential of these cutting-edge techniques for supporting auto-completion in GitHub workflows.

We tested both tools on 30 instances in our test set: We randomly selected 15 workflows with the highest confidence score for which the proposed approach provided correct predictions and 15 for which the proposed approach failed to provide meaningful recommendations.

The results indicate that out of the 15 correctly predicted actions by the proposed approach, Copilot was able to generate 7 correct recommendations. 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 the proposed approach 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 the proposed approach, the chatbot can only suggest 4 meaningful GitHub workflow completions, while providing incorrect action elements/scripts for the remaining 11 instances. When we tested ChatGPT on the instances where our approach 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.

5.3 Threats to Validity

Construct validity. One threat is the extent to which the masking is representative of what programmers do during their tasks [31]. 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 provide a proxy for the "difficulty" of the prediction task (and compare with other completion tasks), we used a consolidated metric used to assess the difficulty of an IR task *i.e.,* Shannon's entropy [53], already used in the past for similar purposes [48, 8, 19]. To evaluate the quality of the predictions, we used consolidated measures for code completion tasks, namely percentage of correct predictions, BLEU-4 [50, 52], and ROUGE score. Furthermore, we complemented such measures with a qualitative analysis of a statistically significant sample of wrong predictions having high confidence.

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 the proposed approach 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. Our work shows how the proposed approach performs with a T5 small model. However, it can be applied as is with larger transformer models (*e.g.,* GPT-3 [12] or GPT-4 [49]). Furthermore, while we fully experimented the proposed approach in the context of GitHub workflow completion, with proper training/fine-tuning, it is important ot note that:

1. As explained in Section 3.3.3 and as will also be detailed in the user manual of Section 6, the tool has been implemented to also support GitLab workflows;
2. The proposed approach could be applied to CI/CD pipelines developed with other technologies, *e.g.,* Jenkins;
3. In principle, the proposed approach can be used, with a proper training, to support the automated completion of other types of YAML files, *e.g.,* various forms of configuration files.

6 User Manual

In the following, we describe, one, by one, the syntax and the configuration files for the various components of the proposed toolset.

6.1 Tool Installation

The tool can be installed using the PyPi Python Installation Utility, by running the following command:

```
 pip install git+https://cosmos-devops.cloudlab.zhaw.ch
/cosmos-devops/cosmos-tools/bad-practice-
resolution-recommender.git#egg=ci-completion
```

Note: in this installation instruction we refer to the installation of version 1.0.0, but the same instructions apply for any released version.

6.2 Tokenizer Creator

Usage:

```
CreateTokenizer.py [-h] (-f FILENAME | -d DIRNAME) -o OUTDIR
```

options:

- `-h`, –help show this help message and exit.
- `-f FILENAME, -filename FILENAME` File with the dictionary corpus.
- `-d DIRNAME, -dirname DIRNAME` Directory where to search for files.

- `-o OUTDIR, -outdir OUTDIR` Directory where to store the resulting tokenizer.

The *Tokenizer Creator* can be configured through a configuration file named `tokenizerConfig.json`, where the following settings are specified:

- *extensions*: list of extensions for files to be indexed when creating the vocabulary (if using the `-d option`).
- *temp file*: temporary file used when creating the dictionary (will deleted at the end).
- *tokenizer prefix*: name of the dictionary model files (the tokenizer will produce a .model and a .vocab file, other than a directory with the pretrained tokenizer, than needs to be copied where the *Model Trainer* or the *Inference APIs* are used).
- *special tokens*: list of special tokens used in the abstraction phases. (usually to be left unchanged as it depends on the implementation of the abstractor).
- *vocab size*: maximum size of the vocabulary.
- *max sentence length*: maximum length of a sentence to be parsed.

6.3 Training Set Creator

Usage:

```
 CreateTrainingSet.py [-h] [-t github,gitlab] -d DIRNAME -o
OUTFILE [-a]
```

options:

- `-h, -help` show this help message and exit
- `-t {github,gitlab}, -type {github,gitlab}` Workflow type (gitlab or github)
- `-d DIRNAME, -dirname DIRNAME` Directory where to search for files for creating the training set
- `-o OUTFILE, -outfile OUTFILE` Output training set file
- `-a, -abstraction` Enables the workflow abstraction

The *Training Set Creator* can be configured through a configuration file named `trainingCreatorConfig.json`, where the following settings are specified:

- *extensions*: list of file extension to search for when creating the training set;
- *max file size*: maximum file size (in lines, longer will be skipped);
- *raw dump* : if enabled, the file content will not be represented in JSON format;
- *extended context*: if enabled, all lines preceeding a job to complete will constitute the context, otherwise, only the parent job will;
- *input label*: label used in the training file to name the input field (must be the same for the training script);
- *target label*: label used in the training file to name the predicted text field (must be the same for the training script);
- *github steps*: list of GitHub environments for which the completion will be enabled (*e.g.,* `steps`);
- *gitlab steps*: list of GitLab environments for which the completion will be enabled (*e.g.,* `script`);
- *min target tokens"*: minimum size (in tokens) of the prediction field (if less, the training entry will be skipped).

6.4 Model Training

Usage:

```
TrainModel.py [-h] -f FILENAME -m MODELNAME
```

options:

- `-h, -help` show this help message and exit
- `-f FILENAME, -filename FILENAME` File with the training set corpus
- `-m MODELNAME, -modelname MODELNAME` File where the model is saved

The *Model Training* script can be configured through a configuration file named `trainConfig.json`, where the following settings are specified:

- *validation percentage*: splitting percentage for the validation set;
- *tokenizer*: directory where to load the tokenizer;
- *input max length*: maximum length (in tokens) of the input (longer will be truncated);
- *output max length*: maximum length (in tokens) of the output ground truth (longer will be truncated);
- *context field name*: label used in the training file to name the input field (must be the same for the training creator script);
- *target field name*: label used in the training file to name the predicted text field (must be the same for the training creator script);
- *prompt*: prompt text used to train the predictor model;
- *model*: base pretrained model to use;
- *output dir*: directory where to store the checkpoints;
- *learning rate*: model learning rate (for T5, do not change it);
- *weight decay*: weight decay during training;
- *batch size*: batch size of the neural network training;
- *save total limit*: maximum number of checkpoints to save (older ones will be removed);
- *training steps*: number of steps in the training phase;
- *metric for best model*: metric used to evaluate the best model during training;
- *patience*: early stopping patience (training will stop if the metric does not improves for the specified number of epochs).

6.5 Abstractor

The *Abstractor* module is contained in a Python file named `Preprocessing.py`. Both the *Training Set Creator* and the *Inference API* depend on it. The *Abstractor* has its configuration file named , which simply specifies the name of files that contain information necessary to the *Abstractor* itself. These files must be brought with the *Abstractor e.g.,* when deploying the *Inference API* of a different machine.

Specifically, the configuration files contains the following fields:

- *domain names*: known domain names used to recognize URLs;
- *file extensions*: known file extensions used in the heuristic to recognize file names;
- *known commands*: known commands that are invoked from the current path (`./`), *e.g.,* Maven or Gradle wrappers.

6.6 Inference API

As for the Inference, we release a Python API that can be deployed on any machine and possibly integrated in development environments/tools. The API will be used as follows. All a client has to do is to import the `YmlInference` class with the code:

```
from YMLInference import YmlInference
```

and then use the following methods:

1. Constructor: `YmlInference(abstracting, jsonInput)`: where the parameters are Boolean values indicating whether (1) the input needs to be abstracted by the *Abstractor* (needed if we use an Abstracted inference model), and (2) if the input is already in JSON. Both parameters are False by default.
2. Once created an instance of the `YmlInference class`, one can use one of the following methods to generate the prediction:
   - `generateYmlRaw(text)`: performs the inference given the input *text* and returns the generated text as is, without post-processing it (the user has to interpret it);
   - `generateYml(text)`: performs the inference given the input *text* and attempts to produce a valid YML as output.

Note that both `generateYmlRaw` and `generateYml` produce a list of outputs, depending on the inference API configuration (see below).

The Inference API is configured through a configuration file named `inferenceConfig.json`, where the user can specify the following parameters:

- *base model*: based model for the inference (must be the same as of the training);
- *tokenizer*: directory containing the tokenizer (must be the same as of the training);
- *modelFile*: model file name
- *prompt*: prompt for the model (must be the same as of the training);
- *max new tokens*: maximum number of tokens to generate during the inference;
- *num beams*: beam size during the inference (more is better but also slower)
- *num return sequences*: number of results to return (must be $\leq$ *num beams*, if not, it will be set equal to *num beams*).

6.7 Simple Inference Client

While the inference is distributed as an API, we release a simple inference client that reads its input from a file or from the standard input. Its usage is described in the following.

Usage:

```
InferenceClient.py [-h] [-a] [-j] [-i INPUT]
```

options:

- `-h, -help` show this help message and exit;
- `-a, -abstract` Abstracts the input text before inference;
- `-j, -json` Takes JSON input directly;
- `-i INPUT, -input INPUT` Input file (default <STDIN>).

6.8 Restoring Configuration files

The script `CreateConfigFiles.py` generates/restores the JSON configuration files for the various components of the tool to their default factory values.

6.9 HOWTO: Creating a Training Model

To create a training model the steps, one should start from a directory containing YAML files (*e.g.,* GitLab or GitHub CI/CD workflow configuration scripts). Then, the steps to be followed are:

1. (optional) given the files, create a Tokenizer. Otherwise, the one distribute with the tool might just be good enough;
2. Create a training set from the files. The user may decide to create an abstracted training set or a raw one, depending on the type of inference to perform;
3. Perform the training. This step likely requires GPUs.

6.10 HOWTO: Inference

Once a training model is available, the following items must be deployed (*i.e.,* simply copied) on the inference machine:

- *Inference API* script (`YMLInference.py`) and *Abstractor* script (`Preprocessing.py`), along with their configuration files (`inferenceConfig.json` and `preprocessingConfig.json`) and the files needed by the *Abstractor* (see above);
- the Tokenizer (entire directory). It must be the same used during the training, and its path must be specified in the `inferenceConfig.json`;
- the model file produced by the training (its name must be specified in the `inferenceConfig.json`);
- (optional) the *Inference Client*, if one does not want to develop their own client.

Once the aforementioned files are deployed, the inference can be performed by following the instructions provided for the *Inference APIs* or for the *Inference Client*.

7 Related Work

CI/CD workflow auto-completion has commonalities with automated code completion. In the following, we discuss related literature about this task, emphasizing in particular (i) task-oriented models, and (ii) pre-trained models.

although it also has some specific peculiarities. In the following, we discuss some relevant related work about DL-based techniques for completing code (we do not discuss other techniques that have been outperformed by the latter). In this regard, we present information about two primary types of DL-based methods utilized in code completion: (i) task-oriented models and (ii) pre-trained models. While the former are specifically designed to facilitate code completion tasks, the latter are generally more versatile, leveraging the pre-training process to gain fundamental knowledge that can be subsequently applied to support code completion. For additional information regarding the applications of DL to software engineering, please refer to the systematic literature review conducted by Watson *et al.* [66].

7.1 Task-Oriented models for Completing Code

Li *et al.* [38] introduce a pointer mixture network that improves the accuracy of predicting Out-of-Vocabulary (OoV) words in code completion. 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. The effectiveness of the attention mechanism and pointer mixture network in code completion is shown through experiments on two established datasets.

Alon *et al.* [9] propose a language-agnostic approach called Structural Language Model 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.* [13] address the code completion task by focusing on the recommendation of 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.* [11] propose a neural language model suggesting code in Python using a sparse pointer network to capture long-range relationships among identifiers.

Aye and Kaiser [10] 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.* [56] propose a learning-to-rank approach for code completion, which is cheaper in terms of memory footprint than generative models.

In a separate work, Watson *et al.* [68] addressed an issue related to code completion by using a sequence-to-sequence model to suggest assert statements for a given *Java* test case. Their approach achieved a top-1 accuracy of 31% in generating a particular type of code statement.

7.2 Pre-trained Models for Code Completion

Svyatkovskiy *et al.* [55] introduce IntelliCode, a multilingual code completion tool that predicts sequences of arbitrary token types using subtokens to overcome the OoV problem [58].

Liu *et al.* [42] propose a pre-trained Transformer model 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.* [36] 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.* [20] examine the effectiveness of Transformer-based models, such as T5 and RoBERTa, 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 $\sim$29% when predicting entire code blocks.

Our work shares with the aforementioned ones, and in particular with the one by Ciniselli *et al.* [20], the use of transformer architectures, and T5 in particular. That being said, GitHub workflow completion has its peculiarities and challenges. As we have shown in RQ_1, it is intrinsically more challenging (*i.e.,* the text to be completed has higher entropy) than source code completion, even when performing abstractions. Second, 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.

Large Language Models (LLMs) such as GPT-3 [12] or GPT-4 [49] have propelled code completion techniques to new heights. GitHub Copilot [16] is a prime example of this advancement in the field, having undergone fine-tuning using open-source code from GitHub and exhibiting remarkable proficiency in numerous code-related tasks. 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, we provide some evidence showing that GitHub workflow completion is a challenging task for them as well. Also, GH-WCOM can be easily evolved to replace T5 with LLMs.

Kanade *et al.* [35] also demonstrated how code embeddings can assist with code-related tasks, such as identifying variable misuse and repair, specifically in the context of code completion for a single token.

8 Conclusion

In this deliverable described an approach and its toolkit implementation for automatically completing CI/CD pipeline scripts. The approach is based on T5 [51] pre-trained models to automatically recommend workflow completions in different scenarios, *i.e.,* predicting the next step (NS_{task}), or completing a workflow job. The tool has been implemented to support the completion of both GitHub and GitLab workflows.

Our empirical analysis conducted on GitHub data found that (i) recommending GitHub workflow completion is more difficult than recommending code blocks, (ii) leveraging a pre-training involving English text (possibly complemented by `YAML` files) always helps, (iii) 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 (iv) the model confidence correlates with the likelihood of generating a correct prediction.

Last, but not least, despite the recent advances in text and code generation thanks to LLMs [2, 1], the proposed still shows itself to be competitive for context-sensitive completion tasks.

References

[1] ChatGPT https://openai.com/blog/chatgpt.

[2] GitHub Copilot https://copilot.github.com.

[3] GitHub Marketplace https://github.com/marketplace?type=actions.

[4] GitHub workflows. Last accessed June 5, 2023.

[5] GitLab ci/cd. Last accessed June 5, 2023.

[6] MSR mining platform. https://seart-ghs.si.usi.ch.

[7] T5 public checkpoint gs://t5-data/pretrained_models/small.

[8] Ozgur Aktunc. Entropy metrics for agile development processes. pages 7–8, 11 2012.

[9] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR, 2020.

[10] Gareth Ari Aye and Gail E Kaiser. Sequence model design for code completion in the modern ide. *arXiv preprint arXiv:2004.05249*, 2020.

[11] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*, 2016.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[13] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. Holistic combination of structural and textual code information for context based api recommendation. *IEEE Transactions on Software Engineering*, 2021.

[14] L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[15] Lianping Chen. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72 – 86, 2017.

[16] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[17] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in C code. *IEEE Trans. Software Eng.*, 49(1):147–165, 2023.

[18] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021.

[19] Marcin Cholewa. Shannon information entropy as complexity metric of source code. In *2017 MIXDES-24th International Conference" Mixed Design of Integrated Circuits and Systems*, pages 468–471. IEEE, 2017.

[20] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of transformer models for code completion. *IEEE Trans. Software Eng.*, 48(12):4818–4837, 2022.

[21] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[22] COSMOS Project. Catolog of good and bad practices when applying ci/cd in cpsos development., 2021.

[23] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. 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*, pages 235–245. IEEE, 2022.

[24] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245. IEEE, 2022.

[25] Paul M. Duvall. Continuous delivery: Patterns and antipatterns in the software life cycle. *DZone refcard #145*, 2011.

[26] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 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*, pages 935–947. ACM, 2022.

[27] Yoav Goldberg. *Neural network methods in natural language processing*. Morgan & Claypool Publishers, 2017.

[28] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

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

[30] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.

[31] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 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*, pages 960–970, 2019.

[32] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 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*, 2017.

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

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

[35] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code, 2020.

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

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

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

[39] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 602–614, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 511–523, New York, NY, USA, 2022. Association for Computing Machinery.

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

[42] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 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, 2020.

[43] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. An empirical study on code comment completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170. IEEE, 2021.

[44] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. Automated variable renaming: Are we there yet? *arXiv preprint arXiv:2212.05738*, 2022.

[45] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. Using deep learning to generate complete log statements. *arXiv preprint arXiv:2201.04837*, 2022.

[46] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 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)*, pages 336–347. IEEE, 2021.

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

[48] Hector M Olague, Letha H Etzkorn, and Glenn W Cox. An entropy-based approach to assessing object-oriented software maintainability and degradation-a method and case study. In *Software Engineering Research and Practice*, pages 442–452. Citeseer, 2006.

[49] OpenAI. Gpt-4 technical report, 2023.

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

[51] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019.

[52] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[53] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

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

[55] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 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*, pages 1433–1443, 2020.

[56] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion, 2020.

[57] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 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*, pages 54–64, 2022.

[58] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 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):1–29, 2019.

[59] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. *arXiv preprint arXiv:2201.06850*, 2022.

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

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

[62] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 327–337, 2020.

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

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

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

[66] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

[67] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 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*, pages 1398–1409. ACM, 2020.

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

[69] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

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

[71] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. 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*, pages 646–657, 2022.

[72] Fiorella Zampetti, Damian Tamburri, Sebastiano Panichella, Annibale Panichella, Gerardo Canfora, and Massimiliano Di Penta. Continuous integration and delivery practices for cyber-physical systems: An interview-based study. *ACM Trans. Softw. Eng. Methodol.*, 32(3), apr 2023.

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