Table of Contents

Background

There has been existing work on automating tests e.g., Git [Figure 6] and Jenkins. While the existing tools are great at tracking and managing source code change and enabling teamwork on a project, they aren't effective at setting benchmarks and conduct tests for the Machine Learning code. Machine Learning has additional data requirements because it consists of training and testing code.

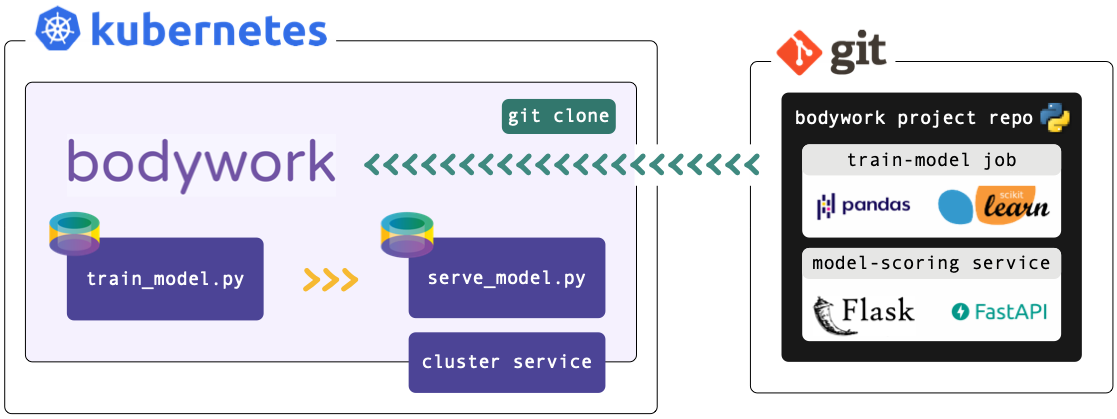
When applying the project, the system should be able to train the datasets based on the training and the benchmarks before moving onto the testing data to determine the model's performance.

Establishing the goals of your project

What is your project fundamentally about?

When building machine learning projects, additional functionalities of training and benchmarking are needed for testing during the code changes. In my project, the goal is to automate the training of the ML code to select the best ML model [1]. My project is divided into two sections: the input containing the ML (training and benchmark) code, and the data. Whenever there is a change made to the repository, the ML system will be retrained to compare with the previous tests. The comparison will help to determine which version of code to become the updated version. Through automation, the project aims to create an iterative process to improve the input machine learning model.

Implement a pipeline to train and test the model with predictions.



What are you intending to design/build/investigate?

The target is to build a machine learning continuous integration system. The project input will be divided into the code (training and benchmark) and the data sections. I will fit the data onto the model to determine its performance. Then I will produce the visualised performance data e.g., model accuracy, sensitivity, and whether they follow the user's requirements.

Whenever there are changes within the dataset or the ML (training and benchmark code), the program must retrain the existing ML model. Consequently, the project simplifies the testing process and ensure that the code will take less time to reach the deployment stage. Figure 5 visualises the MLCI repository branch when two users attempt concurrent edits. After the project has met its objectives, the code will be moved onto the stage and production stage where the code will be deployed.

What do you intend to deliver as the project results?

Figure 4 is the project architectural diagram providing an overview of the requirements. Using the figure, I aim to combine existing version control system with automated training on the machine learning system. In additional to the backend programming, I intend to host the programs on a browser platform, which will allow the user to conduct the testing from different platforms. The browser will contain a user-friendly menu that leads to training and displaying the model score, which will help the user to effectively apply new changes to the ML repository. If the program is examining large datasets, then I will develop or use a suitable existing data version control system to track the changes in the repository.

What would constitute, in your own and your supervisor's eyes, a 100% satisfactory solution?

A perfect solution would be a web-based platform running the machine learning continuous integration tool. Web browser is portable and can be applied for development on different devices from computer to tablet.

Resolve commit conflicts made by different users as show in Figure 1.

Design

Objective	Stage
Outline the software development cycle.	⏳

Front-end

Objective	Stage
Produce a menu for the different functionalities	⏳
For each function, produce a separate webpage for displaying the outputs.	⏳
Develop a UI for configuring the code for testing.	⏳

Back-end

Objective	Stage
Develop version control and suitably handle the commit conflicts.	⏳
Merge function runs tests on the training and the datasets.	⏳
Include external servers to run and test the program in different environment. [Figure 3]	⏳
Create an iterative process to train and benchmark the ML code.	⏳
Calculate a prediction accuracy score on the training and testing datasets.	⏳

Model

Objective	Stage
Automate pushing code	⏳
Automate the ML models and the data	⏳
Train the Machine learning system in a way that the code ca be run on different machines.	⏳

Data Control

Objective	Stage
Design my own data version control system if there are no suitable existing data version control systems.	⏳
Allowing the user to determining the training and validation sample size.	⏳

Display

Objective	Stage
Using the merge results, generate and display performance statistics.	⏳

Testing

Objective	Stage
Set up a test system to compare the new model with the existing model using a common test score.	⏳

Benchmarking

Objective	Stage
Use the benchmark to perform other operations e.g., push code, model, and data onto the repository.	⏳

In the worst case what is the minimum that needs to be completed to achieve a pass?

My goal is developing the basic frameworks for the program as the minimal requirement. The minimal program will be a working version of the testing and running of the machine learning code on a command prompt system. My further purpose would be to produce a visual framework to run the code and will be finished when the time constraints permit the addition of the supplementary functions.

Planning

Objective	Stage
Build the diagram outlining the program structure.	⏳

Front-end

Objective	Stage
Create a web interface for the continuous integration platform	⏳

Back-end

Data

Objective	Stage
Load the datasets into the repository.	⏳
Apply existing data version control system to keep the data up to date.	⏳

Data Processing

Objective	Stage
Split the data into training and testing groups.	⏳

Code function

Objective	Stage
Detect changes within the repository.	⏳
Link existing source control systems with the developed code.	⏳
Enables the ML code to run different versions of local code.	⏳

Testing

Objective	Stage

For each failed build and test, identify the bugs for the users.	⏳
Provide an outcome for each training process.	⏳
Identify and select a version control system to ensure that the program is up to date.	⏳
I can successfully run the back-end code using the command prompt platform.	⏳
The user can run the ML code (training and benchmarking) as inputs to the software.	⏳

Display

Objective	Stage
Present the benchmark statistics in a visually aesthetic format.	⏳
Generate a probabilistic result (probability in which a result is valid). [3]	⏳

External Libraries

Connect the ML-CI tool with an existing source control (e.g., GitHub or Jenkins)	⏳

What are your personal aims that you hope to achieve?

Design

1. Learn and apply the ML development cycle into the current CI system. 🎯

Technical

1. Improve and demonstrate Python skills at creating the coded solutions. 🎯
2. Advance understanding in continuous integration for helping with future teamwork. 🎯
2. Improve upon the existing automated testing strategies and improving automated training and benchmarking mechanisms for the ML system. 🎯

Testing

1. Automated testing aims to improve the understanding of the software development cycle [Figure 2]. Within the cycle, improve the training for the testing stage and identify the differences between the traditional software testing with testing ML software. 🎯
2. Determine a score (probability) for the validity of the test and a confidence interval. 🎯

Error calculation

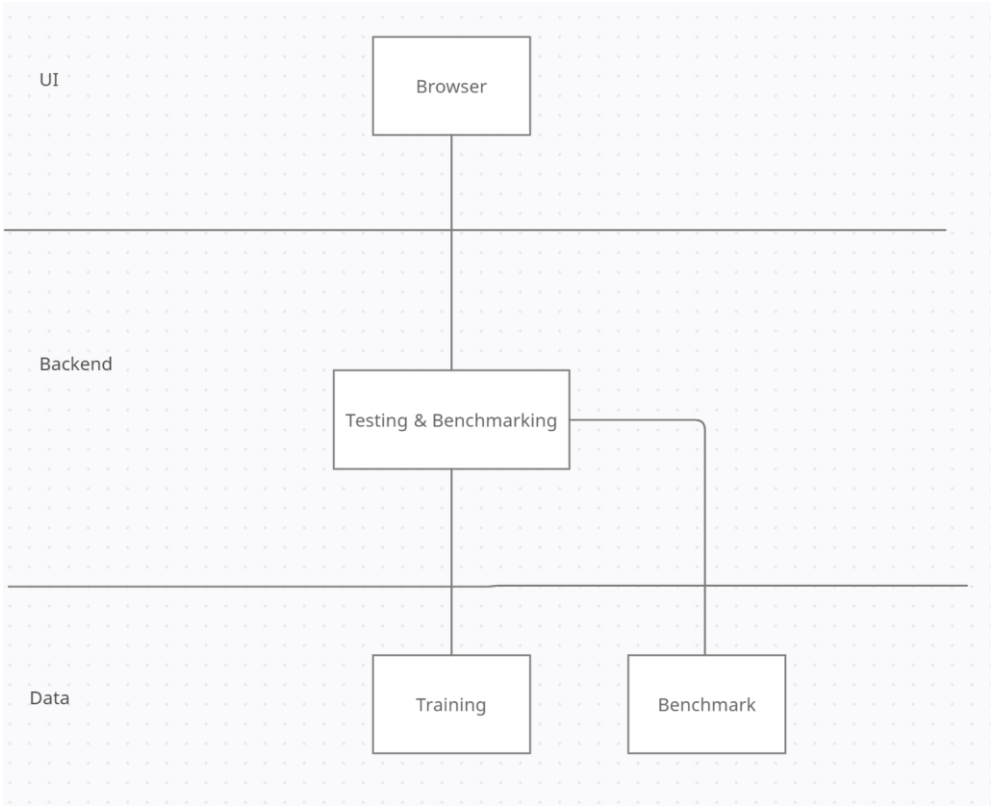
1. Demonstrate building machine learning to minimize the errors from overfitting. 🎯
2. Make some progress within the CI development for the machine learning type of code. 🎯

Planning

1. Apply my research skill into planning and coding the project. 🎯

Optimisation

1. Perform optimisation operations on the machine learning code testing conditions. 🎯


UI
Browser
Backend
Testing & Benchmarking
Data
Training
Benchmark

Figures

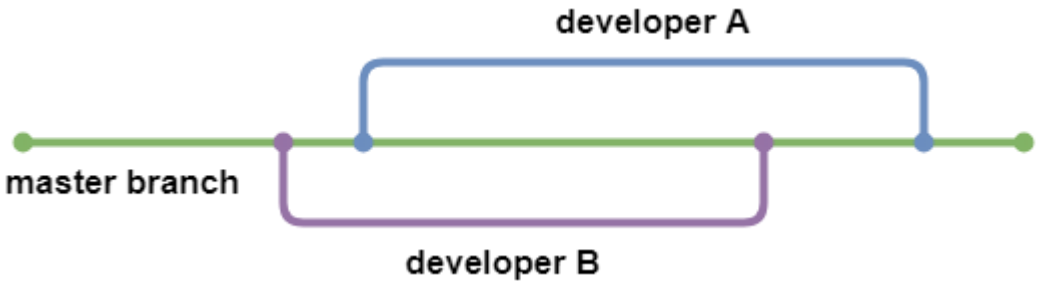


Figure 1

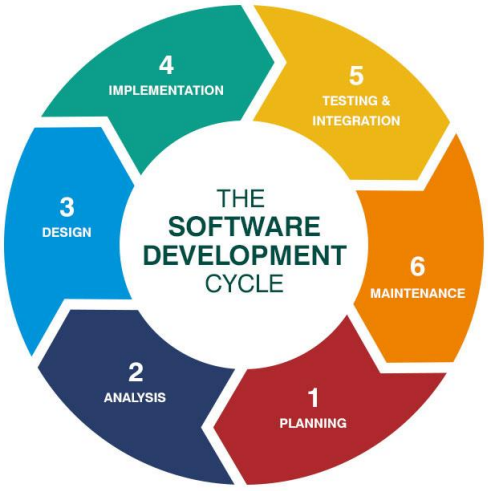


Figure 2

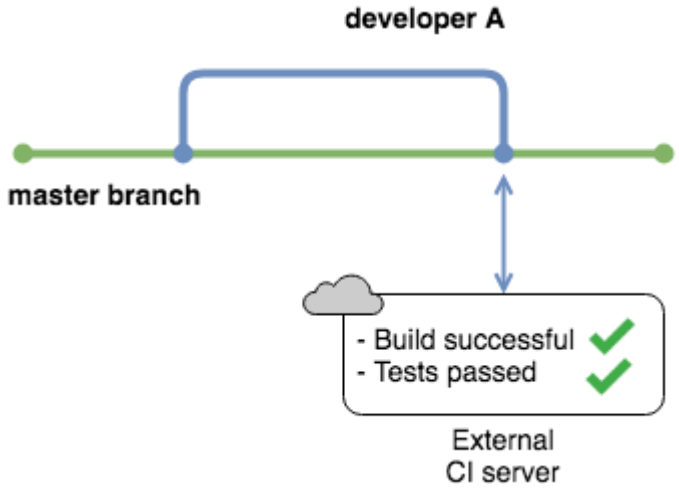


Figure 3

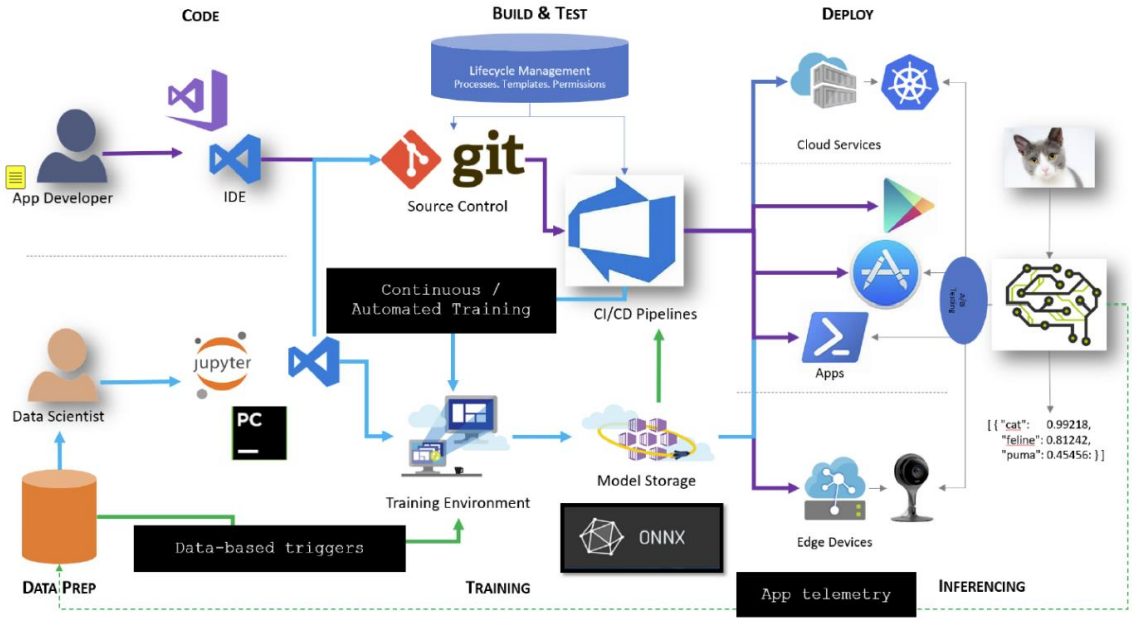


Figure 4

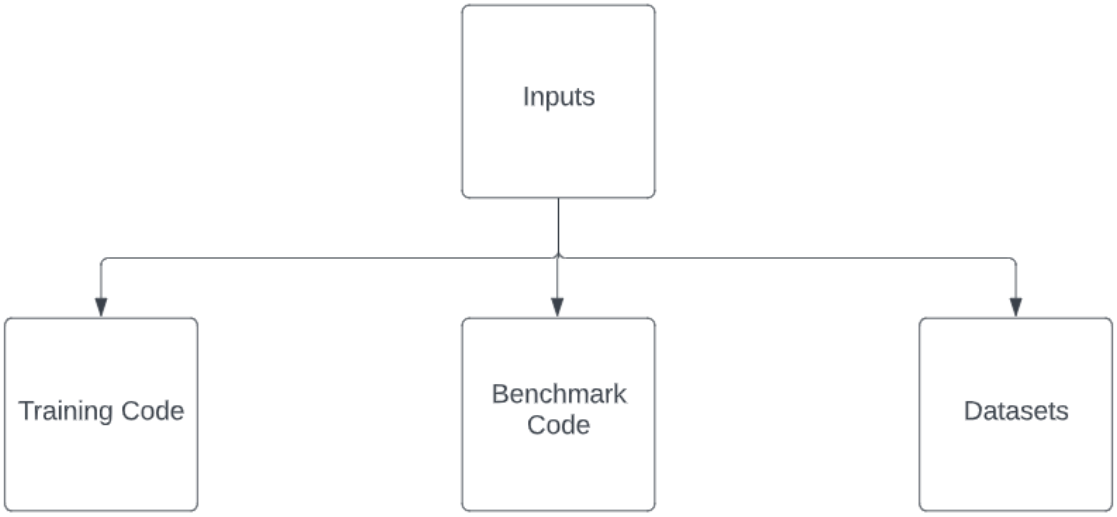


Figure 5

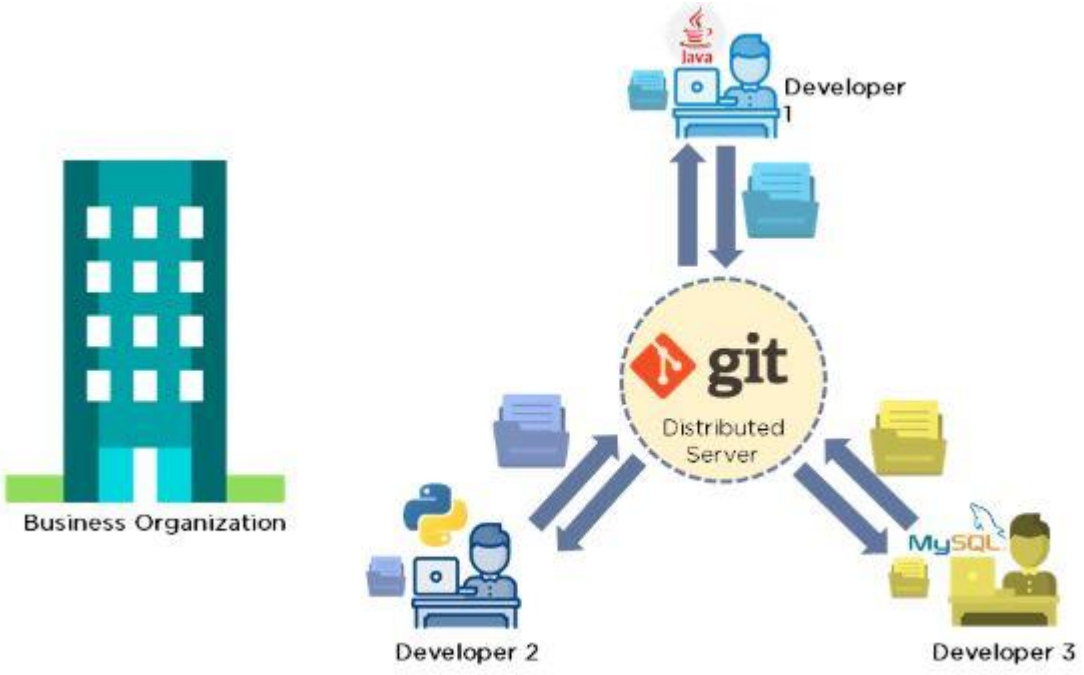


Figure 6

Sources

1. Karlaš, Bojan, et al. "Building continuous integration services for machine learning." Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020.
2. Danglot, Benjamin, et al. "An approach and benchmark to detect behavioral changes of commits in continuous integration." Empirical Software Engineering 25.4 (2020): 2379-2415.
3. Renggli, Cedric, et al. "Continuous integration of machine learning models with ease. ml/ci: Towards a rigorous yet practical treatment." Proceedings of Machine Learning and Systems 1 (2019): 322-333.

"Continuous delivery is the ability to get changes of all types – including new features, configuration changes, bug fixes, and experiments – into production, or into the hands of users, safely and quickly, in a sustainable way."

Jez Humble

Traditional Delivery Process

Traditional code development divides the responsibilities into three teams: the development 💻, quality assurance 🧪, and release teams ☑️. The development team focuses on coding 😺 💻 for passing the code onto the quality assurance team. Using the technical documentation, the quality team conducts user acceptance tests 🧪 on the code from debugging to checking whether the code is following the requirements. After passing the quality tests, the production team determines whether it is possible to stage the updates 👍.

The traditional release cycle can take up to several months to ensure that the code is in production 🏭. The delay ⏳ may result in the code being not up to date 🗓️ and not fitting the purpose of the customer. If the customer discovered a bug, it has been too long for the development team to recall and identify the problem. Consequently, this makes debugging more challenging after deploying the program. With individuals working on the different code sections, it weakens the communication 🎥 between the different teams, which makes it challenging to form more coherent and consistent code 🖥️. Therefore, we need an alternative approach to improve the software delivery experiences.

Continuous Delivery

Continuous Delivery uses a pipeline to automate the changes to ensure a faster deployment. A case study examines the differences between Yahoo! and Flickr's code management 🧑 💼 culture in 2005. While Yahoo focused on traditional coding approaches. Flickr implemented continuous delivery to produce regular updates each day 📆.

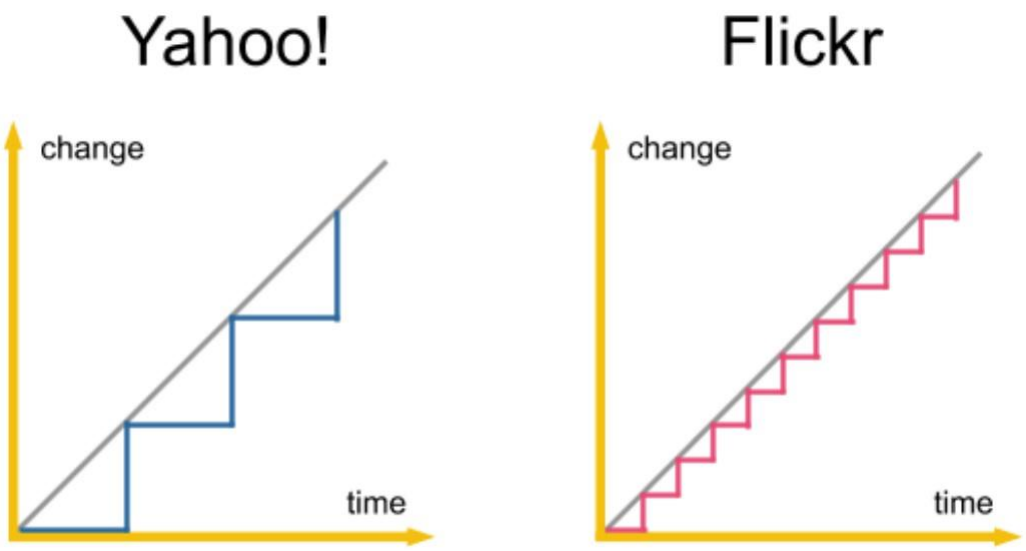


Figure 1.2 – Comparison of the release cycles of Yahoo! and Flickr

The exchange allowed Yahoo! to realise the benefits of continuous integration, reducing downtime. The minor changes in code are reversible and have less risk than implementing a lot of changes in one update. In each update, there will be fewer bugs for the programmer to go through and the client will experience fewer changes.

Today's focus has been on how to construct the program's main structure. I decided to explore the foundations for the program through the Machine Learning Life Cycle.

Define the input for the program.

Previous work has been focused on establishing a pipeline and implementing a function to track the changes. The next step will be to evaluate upon whether the current procedures are in the correct direction and to implement code for tracking the repository into the integration system.

Client working on machine learning repository.

Watches the repo for changes: GitHub

GitHub: check the last time of the updates. Use the Id to check whether there have been any changes.

Tracking code: examine the commit ids.

Create local copies of the data for comparison.

Git computes differences and stores the code.

How to focus on tracking the data in the repository?

Timestamp to track the data changes (simple)

Examine the changes on the single repository.

Version Control : use the existing version control on the sources.

Optional: Design a source control for the data.

Installed git-python to use Python to track the Git repository versions.

I have added comments for the files pipeline.py and pipeline_example.py.

Career: PHD research project for funding and advertised PHD project roles.

Application process: interview.

Institutes with programs: Biology and Welcome Trust Program: Interview + Selection Process.

https://wellcome.org/grant-funding/schemes/four-year-phd-programmes-studentships-basic-scientists

Key identification: Department + Research Topics.

Topics: science and machine learning.

https://learn.microsoft.com/en-us/azure/devops/pipelines/customize-pipeline?view=azure-devops

Website to determine the platform for the YAML file.

https://www.youtube.com/watch?v=9BgIDqAzfuA&list=PL7WG7YrwYcnDBDuCkFbcyjnZQrdskFsBz&index=4

Tutorial series for implementing a machine learning continuous integration system.

Apply DVC to help with managing the larger datasets.

I've followed the tutorial and completed the steps from the first two videos.

1. Changed load data methods to loading the data from the csv and splitting them into training and the testing sets.
2. Next step: follow the 3rd step video tutorial to extend the application functionalities.
3. I added the wget library to the requirements because it is needed for downloading a zip file.
4. I created the farmers.csv file and now am encountering some difficulties loading the data from the csv file. I managed to load the data headings and am uncertain about the next steps.
5. Identified the useless columns in the table for the future operation to drop the useless columns from the table.

Monthly updates on the application.

Datasets: background using various sources.

Is it possible to automate each part in the server?

Create small versions of the application

Basic: appearance – divide the program into the front end and the back end.

https://neptune.ai/blog/ways-ml-teams-use-ci-cd-in-production#:~:text=Continuous%20integration%20(CI)%20is%20the,(to%20build%20the%20application).

Backend: run preliminary work before the machine learning program.

Design an automated system: run a sequence of instructions.

Predictor: job

Each job is a workflow with a configured engine to run the workflows.

Input: machine, types, and outputs.

Executable: the directory to run the application.

Example:

Job name and runnable determines whether it is possible to run.

Custom data validation methods, there is drop-down.

An administrator can write down the drop-down items.

Set the tasks into order and save the tasks.

Is there going to be overwrite conflicts between the standard outputs and the outputs?

If there are some commands, files outputs are written on system, standard output, and standard error. Outputs are sent to standard out and written to others e.g. png.

Standard output -> file

Out glob: check whether the file has been examined.

https://www.commonwl.org/user_guide/topics/yaml-guide.html

To save work from writing a workflow language

User configures and determines the benchmarks for the machine learning algorithm

Gain performance statistics on the changes

Record the previous running statistics to make decisions.

Writing own or pre-existing data storage systems.

Example input: mnist, consider the scale of the project for the engineering

Example output: performance number from the classification.

Congressional maps

: clustering for grouping different districts

Continuous integration

Adds model to the repository and tests the program functionalities.

Source control

Examine continuous integration and data tracking tools: GitHub, Alien Brain

Other source control systems (GitHub, alienbrain). Source control for data sets?

What other CI tools are out there (Jenkins)

Automated build systems for software.

User interfaces for automation and build systems (i.e., Jenkins)

What languages or technologies to use to complete the project?

Protein Web Server

Strength

1. It will be an excellent exercise for building a new web project.
2. A challenge to learn and apply interfaculty skills, gain a better understanding of protein structures and new algorithms.

World Conqueror 4 mod development

Strengths

1. The application is fun to play and has excellent layout and strategies.

Limitations

1. There are inadequate number of tutorials for how to implement the ai algorithms and creating new levels.

Machine Learning Continuous integration tool

Strengths

1. Plenty of potentials because it is helpful for programmers to improve their code.
2. Python is a common program language.
3. Python's libraries and functionalities made it easier for processing the machine learning functionalities.

Redistricting Tool

Python : Programming language because it has the geopandas library and the plotlines for plotting the data from each county.

Establishing the goals of your project

What is your project fundamentally about?

Recently, there have been an increase in the machine learning development. When building machine learning projects, the common challenge is overfitting or underfitting the model. In our project, the goal is to automate the building, testing, and the deploying of the new ML code [1]. Through the automation, the project aims to create an iterative process to improve the existing machine learning model. Consequently, through the programs, a working machine learning code will be built.

What are you intending to design/build/investigate?

The target is to build a machine learning continuous integration system. While there are existing systems to automate tests on repository code, there remains plenty of areas of development for the machine learning coding aspects.

The project will be divided into the training and the data sections. Based on the training and the testing data, the program automates error analysis, which allows the program to continue the iterations. Using the tests, the project aims to simplify the process to validate the tests and ensure that the product transitions to the deployment stage within a shorter timeframe.

Figure 5 provides a visual representation for the inputs. The inputs will be divided into two sub sections: the code (ML training and benchmark) and the datasets to train the ML algorithms.

What do you intend to deliver as the project results?

Figure 4 provides an overview of the project diagram, in which we allow connections with the existing source control systems. Using the figure, I aim to deliver an interactive training program on ML code. Using each test result and the performance statistics, we aim to retrain the model through automation.

In ML testing, compare the previous and the current test accuracies to determine which version of code to become the updated version. Applying the methods help to consolidate knowledge of the cost of testing and apply the tests on different data sample sizes.

Use the difference between the test set results and the expected results to calculate the test score. Test score will help to compare the machine learning model with the expected result. After a series of continuous iteration testing the code and assess the model with the desired model, the test score will be expected to converge.

What would constitute, in your own and your supervisor's eyes, a 100% satisfactory solution?

Resolve commit conflicts made by different users as show in Figure 1.

Front-end

Objective	Stage
Provide different interfaces with each functionality.	⏳

Back-end

Objective	Stage
Enables the ML code to run different versions of local code.	⏳
Develop version control and suitably handle the commit conflicts.	⏳
Provide a platform to build and run the code.	⏳
Merge function runs tests on the training and the datasets.	⏳
Using the merge results, generate performance statistics.	⏳
Include external servers to run and test the program in different environment. [Figure 3]	⏳
Create an iterative process to train and benchmark the ML code.	⏳
Create a benchmark. The benchmark will provide a report on the repository stats e.g., accuracy and sensitivity.	⏳
Calculate a prediction accuracy score on the training and testing datasets.	⏳

Data Control

Objective	Stage
For large datasets, conduct research into different data version controls and select a suitable method to process the datasets.	⏳
Allowing the user to determining the training and validation sample size.	⏳

Testing

Objective	Stage
Set up a test system to compare the new model with the existing model using a common test score.	⏳

In the worst case what is the minimum that needs to be completed to achieve a pass?

Planning

Objective	Stage
Build the diagram outlining the program structure.	⏳

Front-end

Objective	Stage
Create a web interface for the continuous integration platform	⏳

Back-end

Objective	Stage
Provide unit testing for each functionality	⏳
Split the data into training and testing groups.	⏳
Present the benchmark statistics in a visually aesthetic format.	⏳
For each failed build and test, identify the bugs for the users.	⏳
Provide an outcome for each training process.	⏳
Identify and select a version control system to ensure that the program is up to date.	⏳
Connect the ML-CI tool with an existing source control (e.g., GitHub or Jenkins)	⏳
I can successfully run the back-end code using the command prompt platform.	⏳
Integrate the system with an existing ML service.	⏳

What are your personal aims that you hope to achieve?

1. Improve and demonstrate Python skills at creating the coded solutions. 🎯
2. Automated testing aims to improve the understanding of the software development cycle [Figure 2]. Within the cycle, improve the training for the testing stage and identify the differences between the traditional software testing with testing ML software. 🎯
3. Learn and apply the ML development cycle into the current CI system. 🎯
4. Demonstrate building machine learning to minimize the errors from overfitting. 🎯
5. Improve upon the existing automated testing strategies and improving automated training and benchmarking mechanisms for the ML system. 🎯
6. Apply my research skill into planning and coding the project. 🎯
7. Make some progress within the CI development for the machine learning type of code. 🎯

Figures

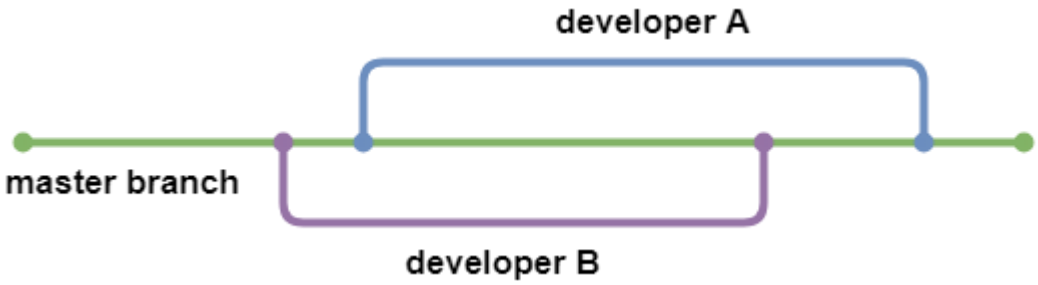


Figure 1

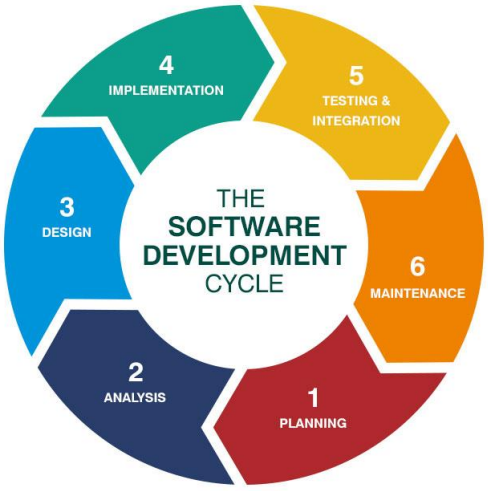


Figure 2

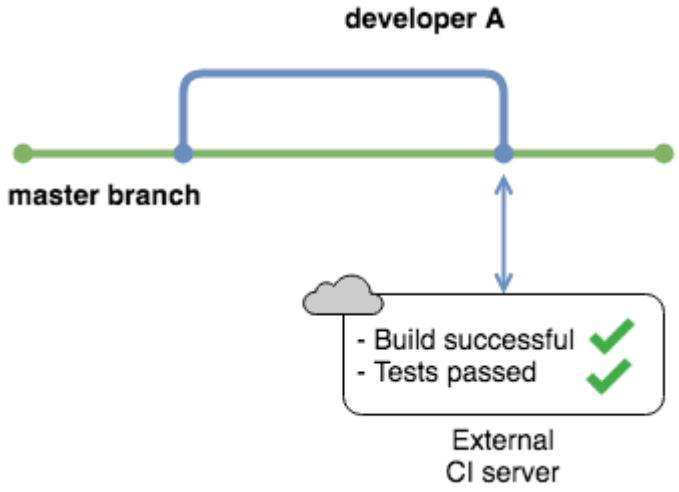


Figure 3

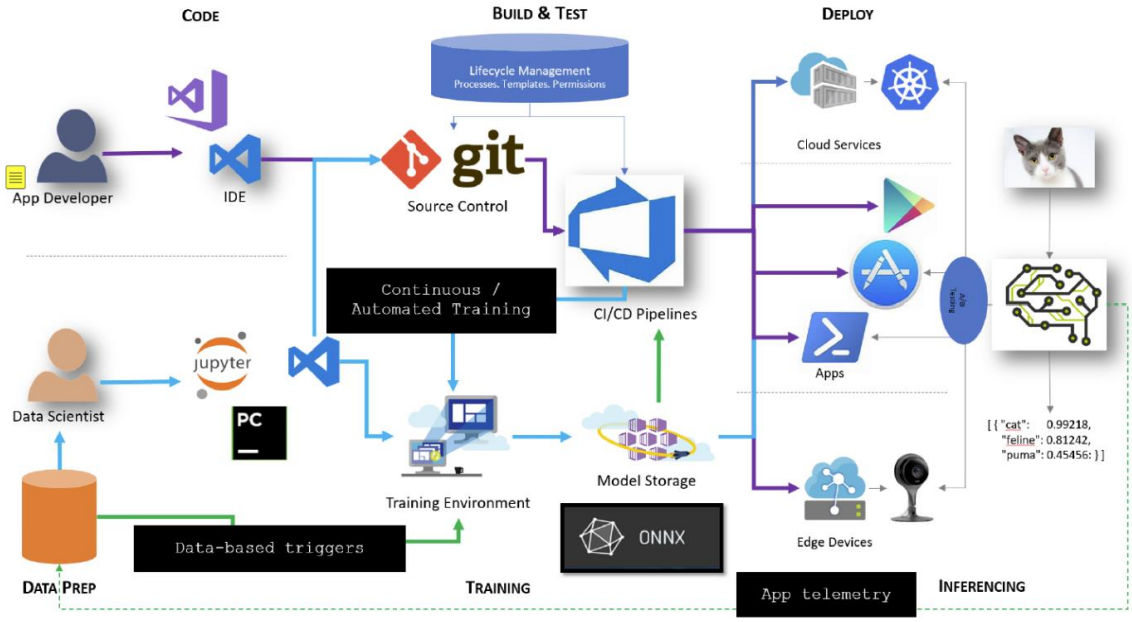


Figure 4

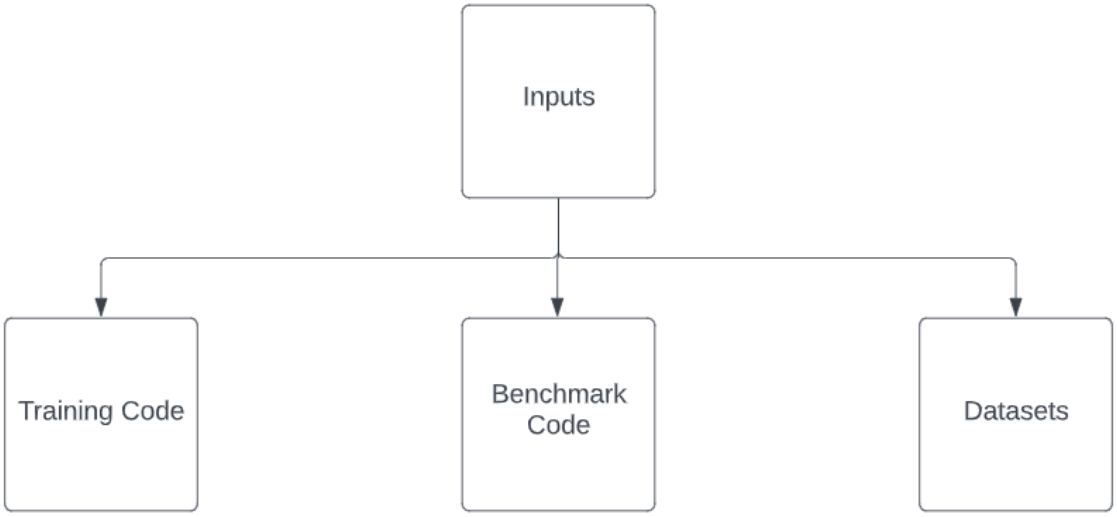
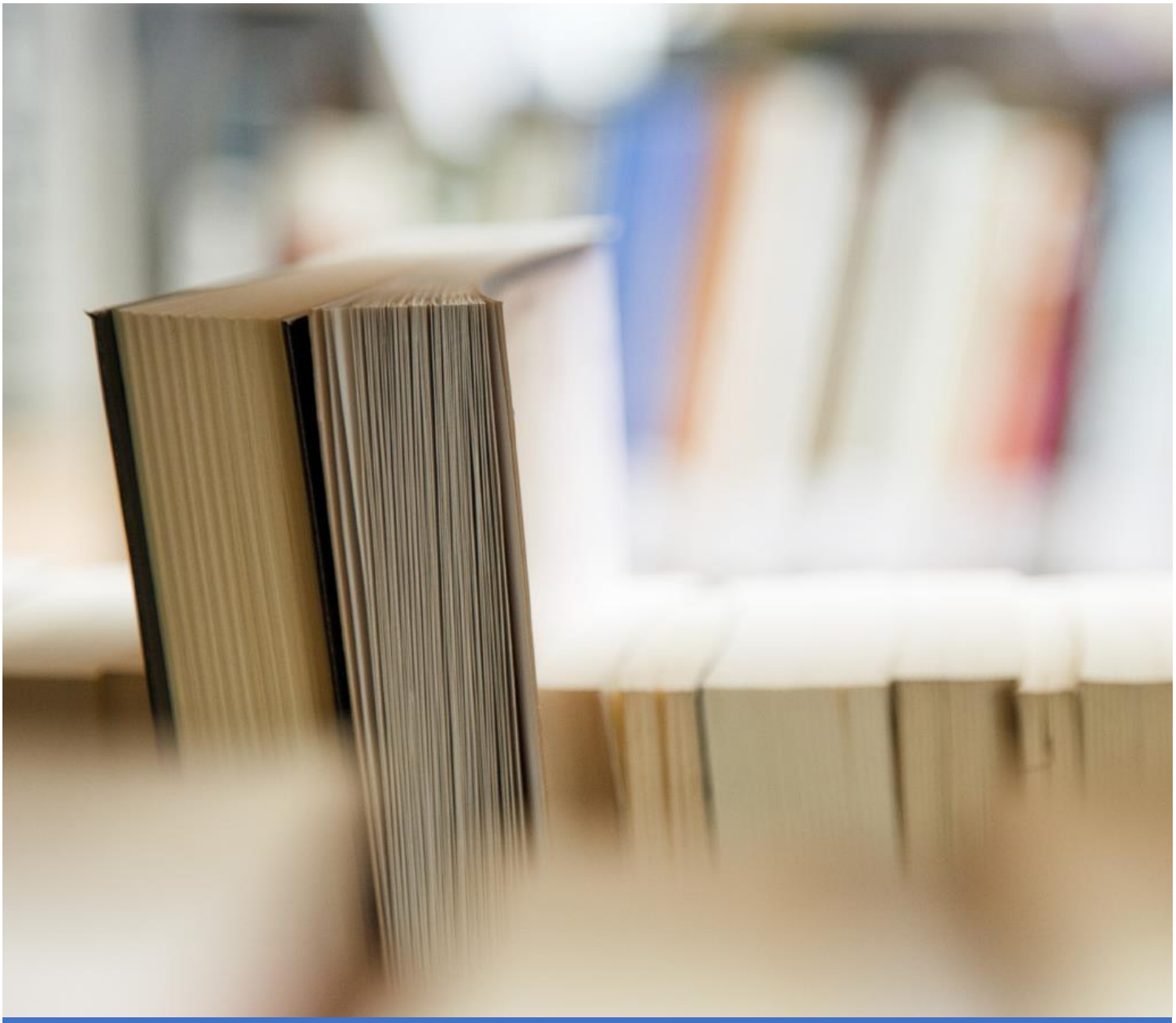


Figure 5

Sources

1. Karlaš, Bojan, et al. "Building continuous integration services for machine learning." Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020.
2. Danglot, Benjamin, et al. "An approach and benchmark to detect behavioral changes of commits in continuous integration." Empirical Software Engineering 25.4 (2020): 2379-2415.



ML-CI System Specification

Haocheng

8/22/22

Computer Science

Table of Contents

Background

There has been existing work on automating tests e.g., Git [Figure 6] and Jenkins. While the existing tools are great at tracking and managing source code change and enabling teamwork on a project, they aren't effective at setting benchmarks and conduct tests for the Machine Learning code. Machine Learning has additional data requirements because it consists of training and testing code.

When applying the project, the system should be able to train the datasets based on the training and the benchmarks before moving onto the testing data to determine the model's performance.

Establishing the goals of your project

What is your project fundamentally about?

When building machine learning projects, the common challenge is overfitting or underfitting the model. In our project, the goal is to automate the building, testing, and the deploying of the new ML code [1]. My project is divided into two sections: the input containing the ML training and benchmark code, and the data. Through the automation, the project aims to create an iterative process to improve the existing machine learning model. Consequently, through the programs, a working machine learning code will be built.

What are you intending to design/build/investigate?

The target is to build a machine learning continuous integration system. While there are existing systems to automate tests on repository code, there remains plenty of areas of development for the machine learning coding aspects.

The project will be divided into the training and the data sections. Based on the training and the testing data, the program automates error analysis, which allows the program to continue the iterations. Using the tests, the project aims to simplify the process to validate the tests and ensure that the product transitions to the deployment stage within a shorter timeframe.

Figure 5 provides a visual representation for the inputs. The inputs will be divided into two sub sections: the code (ML training and benchmark) and the datasets to train the ML algorithms.

What do you intend to deliver as the project results?

Figure 4 provides an overview of the project diagram, in which we allow connections with the existing source control systems. Using the figure, I aim to deliver an interactive training program on ML code. Using each test result and the performance statistics, we aim to retrain the model through automation.

In ML testing, compare the previous and the current test accuracies to determine which version of code to become the updated version. Applying the methods help to consolidate knowledge of the cost of testing and apply the tests on different data sample sizes.

Use the difference between the test set results and the expected results to calculate the test score. Test score will help to compare the machine learning model with the expected result. After a series of continuous iteration testing the code and assess the model with the desired model, the test score will be expected to converge.

What would constitute, in your own and your supervisor's eyes, a 100% satisfactory solution?

Resolve commit conflicts made by different users as show in Figure 1.

Design

Objective	Stage
Implement a full software development cycle.	⏳

Front-end

Objective	Stage
Provide different interfaces with working functionalities.	⏳

Model

Objective	Stage
Automate pushing code	⏳
Automate the ML models and the data	⏳

Back-end

Objective	Stage
Develop version control and suitably handle the commit conflicts.	⏳
Merge function runs tests on the training and the datasets.	⏳
Include external servers to run and test the program in different environment. [Figure 3]	⏳
Create an iterative process to train and benchmark the ML code.	⏳
Calculate a prediction accuracy score on the training and testing datasets.	⏳

Data Control

Objective	Stage
For large datasets, conduct research into different data version controls and select a suitable method to process the datasets.	⏳
Allowing the user to determining the training and validation sample size.	⏳

Testing

Objective	Stage
Set up a test system to compare the new model with the existing model using a common test score.	⏳

Display

Objective	Stage
Using the merge results, generate and display performance statistics.	⏳

Benchmarking

Objective	Stage
Use the benchmark to perform other operations e.g., push code, model, and data onto the repository.	⏳

In the worst case what is the minimum that needs to be completed to achieve a pass?

Planning

Objective	Stage
Build the diagram outlining the program structure.	⏳

Front-end

Objective	Stage
Create a web interface for the continuous integration platform	⏳

Back-end

Data

Objective	Stage
Load the datasets into the repository.	⏳

Data Processing

Objective	Stage
Split the data into training and testing groups.	⏳

Code function

Objective	Stage
Detect changes within the repository.	⏳
Link existing source control systems with the developed code.	⏳

Enables the ML code to run different versions of local code.	⏳

Testing

Objective	Stage
For each failed build and test, identify the bugs for the users.	⏳
Provide an outcome for each training process.	⏳
Identify and select a version control system to ensure that the program is up to date.	⏳
I can successfully run the back-end code using the command prompt platform.	⏳
Integrate the system with an existing ML service.	⏳

Display

Objective	Stage
Present the benchmark statistics in a visually aesthetic format.	⏳
Generate a probabilistic result (probability in which a result is valid). [3]	⏳

External Libraries

Connect the ML-CI tool with an existing source control (e.g., GitHub or Jenkins)	⏳

What are your personal aims that you hope to achieve?

---

1. Improve and demonstrate Python skills at creating the coded solutions. 🎯
2. Automated testing aims to improve the understanding of the software development cycle [Figure 2]. Within the cycle, improve the training for the testing stage and identify the differences between the traditional software testing with testing ML software. 🎯
3. Learn and apply the ML development cycle into the current CI system. 🎯
4. Demonstrate building machine learning to minimize the errors from overfitting. 🎯
5. Improve upon the existing automated testing strategies and improving automated training and benchmarking mechanisms for the ML system. 🎯
6. Apply my research skill into planning and coding the project. 🎯
7. Make some progress within the CI development for the machine learning type of code. 🎯
8. Determine a score (probability) for the validity of the test and a confidence interval. 🎯

Figures

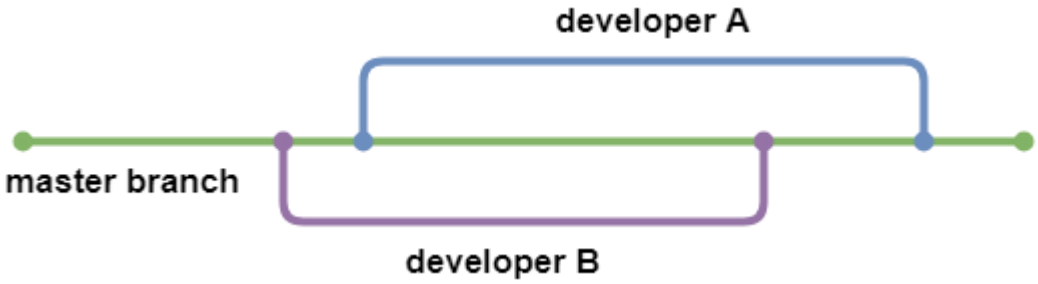


Figure 1

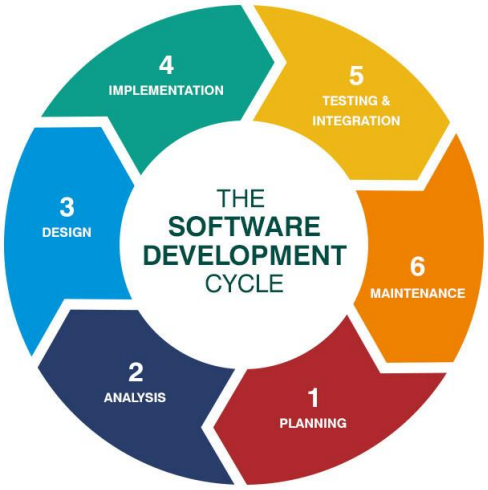


Figure 2

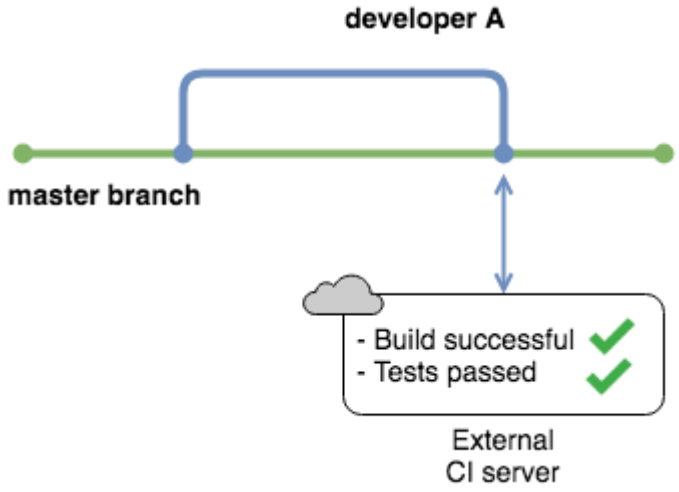


Figure 3

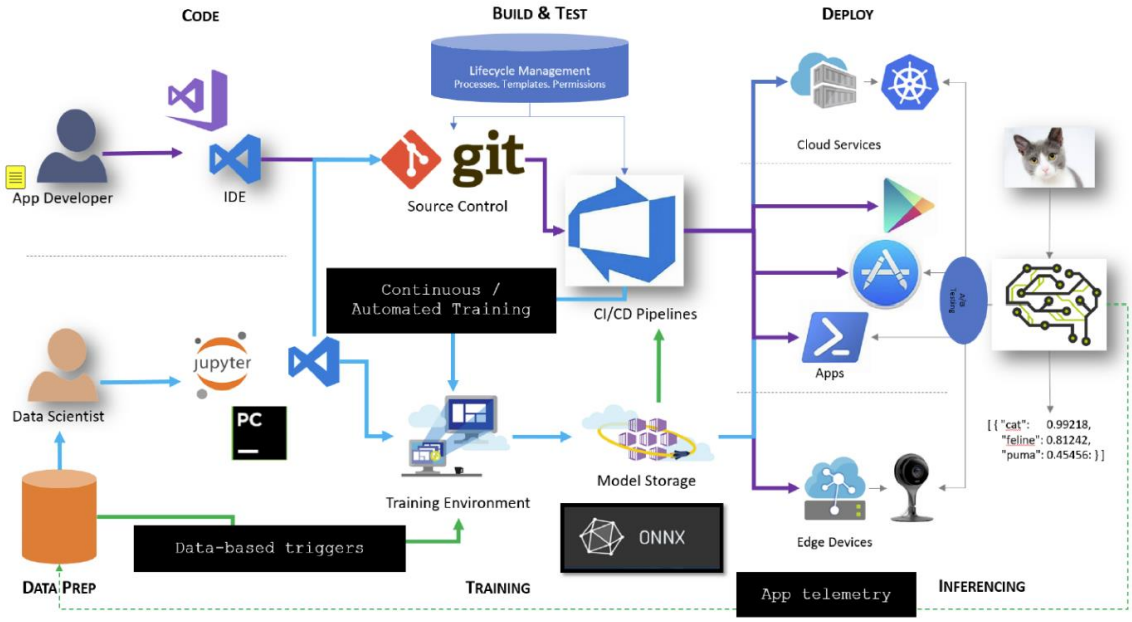


Figure 4

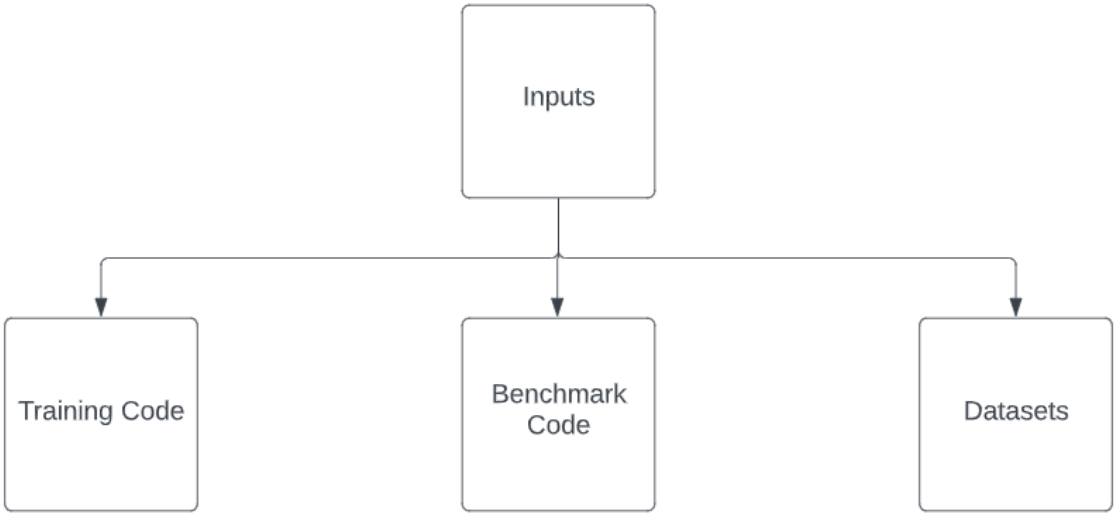


Figure 5

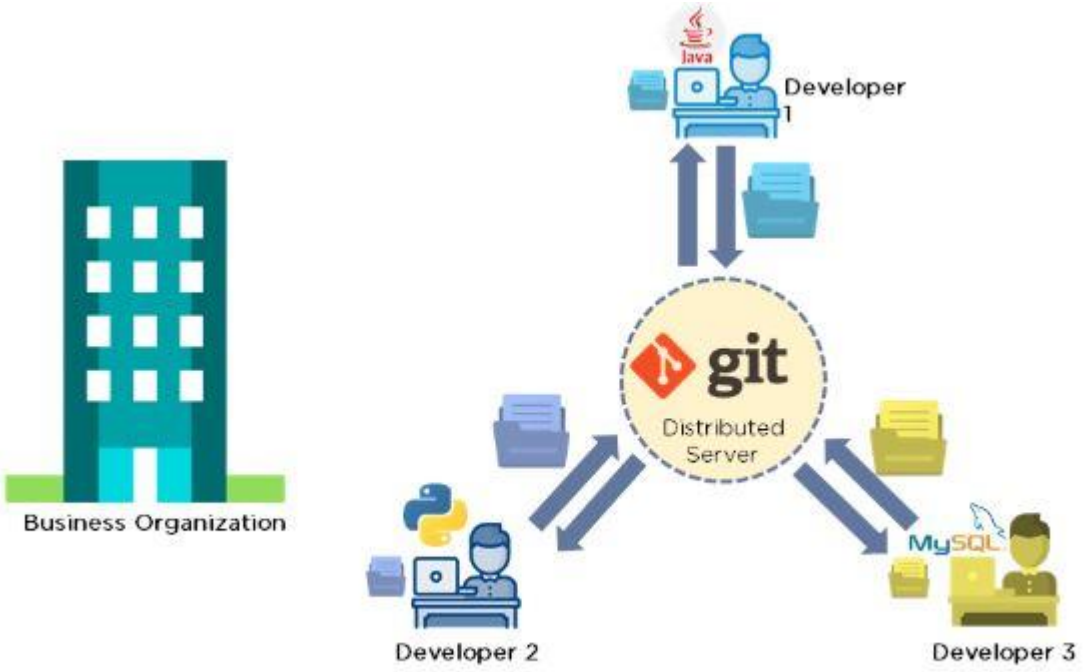

Developer 1
Business Organization
git
Distributed Server
Developer 2
Developer 3

Figure 6

Sources

1. Karlaš, Bojan, et al. "Building continuous integration services for machine learning." Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020.
2. Danglot, Benjamin, et al. "An approach and benchmark to detect behavioral changes of commits in continuous integration." Empirical Software Engineering 25.4 (2020): 2379-2415.
3. Renggli, Cedric, et al. "Continuous integration of machine learning models with ease. ml/ci: Towards a rigorous yet practical treatment." *Proceedings of Machine Learning and Systems* 1 (2019): 322-333.

ML Continuous Integration System

Haocheng Lin

UCL

Author Note

Supervisor: Dr. Daniel Buchan

ML Continuous Integration System

Aims and Objectives

Aim: Create a platform for taking Machine Learning Code and Data as inputs for running automated tests. We will add the ML code into the repository if it passes the benchmark tests.

Objectives

1. Review and configure the ML pipelines for training.
2. Integrate GitHub to monitor the code changes.
3. Implement a tracking method on the datasets.
4. Any changes to the code or dataset will rebuild the ML pipeline.
5. Optimize for the best ML model by comparing new scores with historical ML models' scores.

Work Plan

July – October: Planning

1. Selected the topic.
2. Conducted research on the ML Continuous Integration System.

October – November: Write-up a preliminary specification outlining my progresses.

November: Design a small-scale prototype with basic functionalities.

November – January: Extend upon the prototype into the full system.

January – February:

1. Design Tests for evaluating each task.
2. Interim report for checking the progress.

February – March: Record a video preview outlining the project.

March: Finalize the Project Report for submission.

Expected Outcomes

Minimum Requirements

Code monitoring

Function	Stage
Integrate GitHub to track the code changes.	✅
Run the ML Code whenever there are new changes made to the repository.	✅

Configuration

Function	Stage
Configure the application via YAML text file	✅

Execute ML Code

Function	Stage
Can run a ML pipeline on the local computer.	✅

Results Presentation

Function	Stage
Present a visualized exhibition of the ML results and its scores via email.	✅

Perfect Solution

Code monitoring

Function	Stage
Allow the user to monitor the changes across several repositories.	⏳

Configuration

Function	Stage
Provide a GUI application to help with configuring the code.	⏳

Execute ML Code

Function	Stage
Enables running the pipeline on the different computational platforms.	⏳

Results Presentation

Function	Stage
Present the results on a web interface alongside an email for the user.	⏳

https://realpython.com/python-continuous-integration/#what-is-continuous-integration

Website on continuous integration tutorials.

> *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."*

Building Continuous Integration Services for Machine Learning

Bojan Karlaš[1,2], Matteo Interlandi[1], Cedric Renggli[2], Wentao Wu[1], Ce Zhang[2], Deepak Mukunthu Iyappan Babu[1], Jordan Edwards[1], Chris Lauren[1], Andy Xu[1], Markus Weimer[1]
[1]Microsoft, [2]ETH Zurich

ABSTRACT

Continuous integration (CI) has been a *de facto* standard for building industrial-strength software. Yet, there is little attention towards applying CI to the development of machine learning (ML) applications until the very recent effort on the theoretical side. In this paper, we take a step forward to bring the theory into practice.

We develop the first CI system for ML, to the best of our knowledge, that integrates seamlessly with existing ML development tools. We present its design and implementation details.

ACM Reference Format:
Bojan Karlaš[1,2], Matteo Interlandi[1], Cedric Renggli[2], Wentao Wu[1], Ce Zhang[2], Deepak Mukunthu Iyappan Babu[1], Jordan Edwards[1], Chris Lauren[1], Andy Xu[1], Markus Weimer[1] . 2020. Building Continuous Integration Services for Machine Learning. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394486.3403290

1 INTRODUCTION

Recent decades have witnessed an increasingly frequent usage of machine learning for a wide range of mission critical applications. However, one challenge in building such applications, especially from the perspective of practitioners and domain scientists, is *overfitting*. In our experience in supporting a range of industrial and academic users [9, 14, 22–25], it is not uncommon for users of modern ML platforms to suffer from this problem, as illustrated in Figure 1(a), that the gap between the estimated test/validation accuracy and true test accuracy increases during the development process. *Can we provide tools to help practitioners tackle this problem?*

In this paper, we draw our inspiration from continuous integration (CI), which has been part of the industry standard of modern software development [12], evidenced by the recent surge of cloud-hosted software development services such as Azure DevOps [3] and AWS CodePipeline [2]. CI services lift the burden of managing the software development lifecycle from the developers by providing a variety of tools for building, testing, and deploying software applications in an automated and iterative manner. Development of machine learning (ML) applications is not much different in this regard from regular software systems – it typically requires many iterations as developers try to continuously improve the quality of their ML models. *Can we build continuous integration services for ML to give users constant feedbacks and signals about overfitting?*

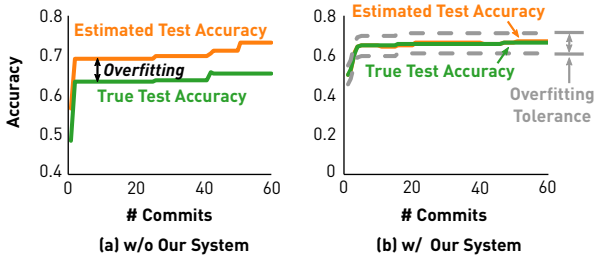


Figure 1: (a) The challenge of building a CI system for ML is that, if not being careful, one might overfit the test set when committing multiple models during the CI process; (b) The goal of our system is to provide rigorous guarantees on the overfitting behavior by, intuitively, measuring the "information leakage" from the test set during the CI process.

Challenges. None of the existing CI services are sufficient when it comes to ML applications [20]. One major issue lies in testing: In traditional software testing, test cases can be reused infinitely to evaluate test assertions, and simply return *deterministic* binary `true`/`false` signals. *In ML testing, every pass/fail signal leaks information about the test set itself and thus may lead to overfitting, resulting in false positive/false negative outcomes* (see Figure 1).

(Example) Consider the test assertion `n - o > 0.01`, where `n` and `o` represent the accuracy of the new and old version of an ML model in consecutive development iterations. The semantics of the test assertion is clear — the test will pass only if the new model improves accuracy by at least 0.01. To evaluate this test assertion, however, one needs to estimate the accuracy of both models using a test set, which is a finite set of i.i.d. samples from the underlying data distribution. Such estimates are inherently uncertain and so is the `true`/`false` evaluation result of the test assertion. Therefore, one has to interpret the evaluation result from a *probabilistic* view: The result holds with high probability if the test set is sufficiently large. Ensuring such probabilistic guarantees for ML test assertions, when the same test set is repeatedly used for evaluation, is one major challenge that "CI for ML" services need to address.

In this paper, we develop the first "CI for ML" service. As illustrated in Figure 1(b), our system controls the size of test/validation set such that the gap between the estimated accuracy and the true test accuracy is guaranteed to be smaller than a small constant specified by the user. To the best of our knowledge, this is the first such system that integrates seamlessly with existing ML development tools. Our service provides a framework for testing ML models that is based on strict theoretical bounds and enables a principled way to avoid over-fitting the test set, which is a common problem that is easily overlooked. Our service is also seamlessly integrable with existing CI frameworks such as TravisCI, Microsoft Azure DevOps,

KDD '20, August 23–27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403290

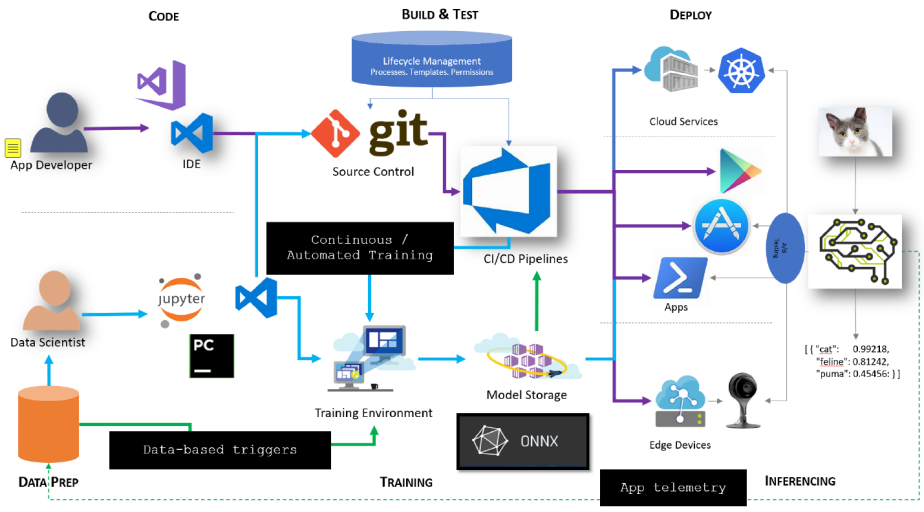


Figure 2: A macroscopic view of where a CI/CD service stands in modern ML application development lifecycle.

and GitHub Actions. We have released an open-source version on GitHub [7] and we have ongoing efforts conducted with Microsoft internal partners to integrate our service into Azure ML Services. In this paper, we present the design and implementation details of our CI service.

1.1 Production Requirements

Making a theoretical framework like `ease.ml/ci` into an industrial-strength tool requires us to revisit the requirements of a real production environment. The first step in our work is to define such requirements from our experience in building real-world ML platforms and applications.

ML Life Cycle. An ML application development lifecycle typically involves multiple iterations. In each iteration, developers try to come up with an ML model with the best quality (e.g., prediction accuracy) on the *training* or the *validation* dataset. This model is then evaluated against a holdout, *test* dataset that is drawn independently from the data distribution that governs the underlying data generation. Based on the quality of the model observed on the test dataset, developers then perform error analysis (if possible) and enter the next iteration. This iterative procedure ends whenever developers are satisfied with the model quality (over the test dataset) or the model quality cannot be improved any more.

CI/CD for ML. A CI/CD solution for iterative ML model development requires supporting numerous types of data sources, a variety of training tools, a validation solution to analyze and validate models (for functionality and performance) and supporting deployment to the infrastructure used to serve models in production. This becomes particularly challenging when data changes over time and fresh models need to be produced regularly, as is the case in many large-scale, AI infused systems. Complexity only grows as models need to be deployed to a hybrid of the Intelligent Edge + Intelligent Cloud. Figure 2 provides an overview of where a "CI for ML" service stands in the whole ML application development lifecycle.

Our Ultimate Goal. Our primary goals here are to (1) standardize the components leveraged for model lifecycle management – model training, model validation, model storage/versioning, model and health monitoring; (2) provide lightweight process and templates to simplify the data scientist/app developer collaboration; (3) reduce the time from model creation to production deployment from the order of months to weeks to days.

1.2 Theoretical Foundation

Recent progress on *adaptive analysis* [13] reveals the fact that the fidelity of the test dataset may fade away when it being accessed again and again. The intuition behind this observation is that, developers may be able to gain insights upon seeing the test accuracy, and then customize their next version of the ML model towards improving the accuracy on this particular test dataset. This obviously may result in *overfitting* — while the prediction error over the given test dataset seems small, the generalization error (which can be estimated using an independently generated test dataset) can be potentially large.

One obvious solution to this problem is to draw an independent test dataset each time when a new version of model is developed. However, this will result in significant overhead in terms of *sample complexity*, i.e., the amount of test data being required. This poses a problem because test data (especially labelled data) is not cheap to obtain. Fortunately, as was shown in [11, 20], it is possible to avoid paying that price as the sample complexity can be reduced to the level that is feasible in practice. This lays the theoretical foundation of developing CI systems for ML applications.

1.3 Data Management

The risk of overfitting due to adaptive analysis requires refreshing the test dataset as CI proceeds, which leads to natural data management problems in terms of effectively maintaining the test data. In addition to standard database operations such as insertion, update, and deletion, users of the CI service may also want to query historical data as well as telemetry information about performance of models that have been submitted in the past. Moreover, users may even wish to keep track of the entire development history and "roll back" to any point in the development trace to "restart"

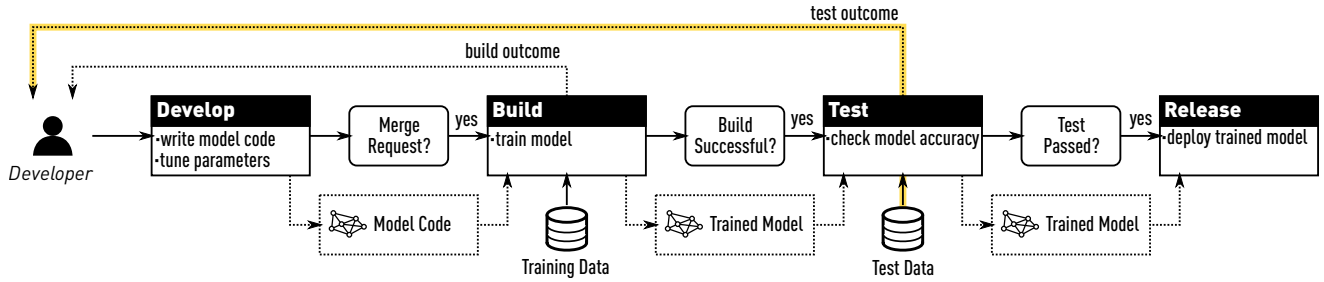


Figure 3: The development lifecycle of a machine learning model in the framework of traditional software development. The shaded yellow line depicts the information leakage pathway that our method tries to solve.

from there. All these requirements need careful design of the data management layer of the CI service to incorporate data versioning and version control mechanisms.

1.4 Paper Organization

We start by presenting the design of the core component, the `mltest` tool, of our "CI for ML" service in Section 2. We then present its implementation details in Section 3. We discuss the new challenges raised by adaptive analysis and our solutions in Section 4. In Section 5 we further present evaluation results that showcase both the necessity and effectiveness of our solutions. We summarize related work in Section 6 and conclude the paper in Section 7.

2 SYSTEM DESIGN

In this section we present an overview of the CI system we have developed, including the interaction model, key design considerations, as well as a walkthrough over the major components it contains.

2.1 An Overview of ML Development Lifecycle

Figure 3 presents an overview of the ML development lifecycle under our CI system. Like the development of regular software, the entire lifecycle consists of four stages (akin to a GitHub or Azure DevOps kind of development scenario):

- *Develop* – the developer writes code for featurizing data, selecting an appropriate algorithm with efficient implementation, as well as basic parameter tuning; the entire ML-related software artifact produced by this stage (including the feature extraction code) is what we refer to as a *model*.
- *Build* – the developer requests merging the code into the master branch (a.k.a., a *pull request*); this automatically triggers the build process of the codebase, which trains the new model over the training data.
- *Test* – the test phase follows if the build process succeeds; the final model returned is evualuated against the test dataset, after which the test accuracy is reported to the developer.
- *Release* – if all test cases are passed and the developer is satisfied with the test accuracy, the model can then be promoted to a release environment for upstream consumption, potentially replacing an old model that was already released.

The key difference of our "CI for ML" system from a classic CI system lies in the *Test* stage:

- *Probabilistic evaluation of test conditions* – unlike test conditions in regular software test that have outcomes which are deterministic, test conditions in our CI system are probabilistic in terms of their semantics.
- *Automatic refresh of test dataset* – whenever the test dataset loses its power as a representative of the underlying data distribution due to repeated accesses, a new test dataset is generated and an automatic swap occurs behind the scenes.

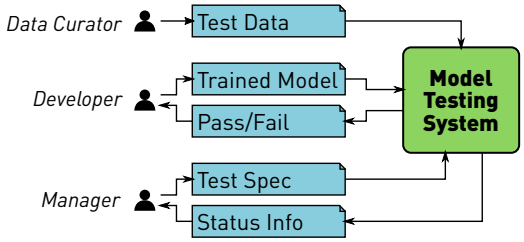


Figure 4: Roles and their interactions with our system.

We will discuss these two respects in more detail soon. Before that, below we give more details regarding the interaction model between our system and the developer.

2.2 Interaction Model

In order to justify various design choices we made, it is important to review the user-facing interface of the system. We define three different types of users (or *roles*). Figure 4 presents the interactions between different roles and our system.

The **manager** role is in charge of defining test conditions. The manager is aware of the broader architecture and all components that make up the user's system, only one of which may be the ML model itself. She is also aware of all tests that have been developed, which gives her the ability to determine which quality standards the model needs to fulfil in order for the overall system to function correctly. Her main point of interfacing with our CI system is the *test spec*. This enables her to control all aspects of model testing, the most important of which is the ***test condition*** that determines whether a new model that was committed will be accepted or rejected. Moreover, the manager also has access to all monitoring tools provided by our system. Mainly, the manager is able to monitor the amount of available test data and the number of test runs that the system can perform before a new test set would need to get staged.

The **data curator** role is in charge of providing fresh test data to the system. The data curator may perform various data preprocessing steps before ***depositing the test data***, which is her main point

of interfacing with our system. Depending on the original source of data and the storage medium, the data curator might benefit from some integration capabilities such as being able to pull data directly from a SQL Database. To achieve this, the data curator may find it useful to implement custom data acquisition adapters.

The **developer** role is in charge of building and improving ML models. The developer may write model code, train the model, and tune its hyperparameters, either manually or by using existing tools and overseeing the process. Most importantly, the developer is in charge of submitting new ML models to our system for *testing* and, potentially in the end, for deployment. Each model that the developer submits triggers a ***test run*** that determines if the model passes the test conditions defined by the manager and is ready for deployment. Depending on the way that our system is configured, the developer may get informed of test outcomes in the form of a binary `pass`/`fail` signal (with built-in probabilistic semantics that we shall discuss later). This signal, albeit a necessary element of the development lifecycle, is the main source of information leakage that our system is trying to control.

2.3 Interfaces

One of the most important design goals of our system is to be easily integrable with as many existing systems and engineering practices as possible. We recognize three prominent interfacing methods that would cover the needs of the vast majority of our users: (1) interacting directly through the command line (CLI), (2) serving Web-based requests (REST), as well as (3) integrating with custom testing code. We implement our solution in Python with support for all three mentioned interfacing methods.

In the following, we present individual interfaces for various functionalities required by all three roles defined in Section 2.2.

2.3.1 Data Management. As we described earlier, the test dataset cannot remain static over the long-term development lifecycle, so it has to be managed as part of the test workflow. Our system achieves this by maintaining a pool of fresh test data where new data can be asynchronously ***deposited*** as soon as it becomes available. New data is added by invoking the `deposit` command.

A subset of that test data gets ***staged***. This is a separate pool of data that is ready to be used for immediate test runs. Each staged dataset has a unique *stage key*. Any staged dataset can be ***loaded*** with the `load` command by using the stage key. If the stage key is omitted, then the latest staged dataset is retrieved.

As our system keeps counting the number of test runs performed over a staged pool of test data, it is able to determine when the maximum allowed number of test runs has been reached (ref. Section 4.1). Once this occurs, the current staged test pool has to be set aside and a new test dataset has to be staged. This is most commonly done automatically, but it is also possible to stage a new test dataset by running a `stage` command.

There is currently no universally accepted interface or storage format for managing ML datasets. What ML datasets have in common is that they are a set of *independent and identically distributed* (i.i.d.) data examples that can be treated individually. In the supervised learning setting that our system focuses on, there is also the concept of *feature* as the input to an ML model and the concept of *label* as its target output. In principle, these concepts are the bare minimum that a data access interface needs to support in order to work with our system.

Our implementation represents datasets by using the Pandas `DataFrame` abstraction, a very popular choice among data scientists. It represents data in tabular form where named columns hold values of the same type. Individual data examples are represented by rows of this table. We expect there to be a single column for target labels and one or more columns for input features, all of which can be specified by name.

2.3.2 Test Condition Specification. As described in Section 2.2, test conditions are used by the manager to define a predicate that needs to be satisfied in order for a model to *pass* the test. This is a necessary step in order to guarantee a certain quality standard of ML models. In typical ML settings, the quality of a model is tested by invoking a *scoring function* over the predictions that the model generated by taking input features from a test dataset. Our implementation currently only supports classification tasks and the model *accuracy* as a scoring function that returns the proportion of labels that the model predicted correctly, as judged by their equality to the true labels that are part of the test dataset.[1]

The result of a scoring function is a single real number from the $[0, 1]$ interval that we call a ***score***. Since any test set is randomly sampled from a (theoretically infinite) pool of test data, the score that we measure is a *random variable*. A test condition is composed of *test clauses*. Test clauses are specified as *inequalities* defined over the ***score*** variables. Each clause is also associated with a *confidence interval* ϵ that enables probabilistic treatment of those clauses. The following is an example test clause:

```
n - o > 0.01 +/- 0.005.
```

Here, the variable `n` represents the score of a newly submitted model that we are currently testing, and `o` is the score of the last model that got accepted by the system. On the right-hand side of the `>` symbol, we have a comparison with a constant `0.01` and a confidence interval `0.005`. Our sample-size estimation method (described in Section 4) ensures that we have enough samples to control the variance of all random variables and correctly evaluate a clause with (high) probability $1-\delta$, where $0 < \delta < 1$ is configurable.

Test conditions represent a conjunction of one or more test clauses. An example test condition made up of two test clauses is:

```
n - o > 0.01 +/- 0.005 and d < 0.01 +/- 0.005.
```

Here, the variable `d` represents the fraction of predicted labels that are different between the newly submitted model currently being tested and the latest model that got accepted.

2.3.3 Test Run. We define a single `run` command that internally orchestrates other system components in order to run a submitted model and evaluate it against the specified test conditions. The command dynamically assembles other components of the system and exposes their configuration options. All options are assigned with sensible defaults, which can be overridden by providing a key-value based configuration file that maps assigned values to configuration options referenced by their keys. Among other things, these options permit specifying user-defined test conditions as well as invoking the `predict` method of the submitted model. Here is

[1] https://en.wikipedia.org/wiki/Precision_and_recall

an example of using the `run` command (we have named our system the `mltest` tool internally):

```
mltest run \
  --config mltest.yml \
  --condition-statement \
    "n - o > 0.01 +/- 0.005" \
  --model-run-command \
   "./predict --data-in {{input}} --data-out {{output}}"
```

It explicitly specifies a test condition and the command used to generate predictions from the trained model. The `{{input}}` and `{{output}}` placeholders are dynamically replaced at runtime by locations of the test data and the temporary output directory, both of which are managed internally by our system.

3 IMPLEMENTATION

The key functional elements the `mltest` tool offers are: (1) compute the number of required test runs given a test condition, and (2) evaluate those test conditions given accuracy score estimates. However, in order to achieve better usability of the system, as showcased in Section 2.3.3 we extend this very narrow range of capabilities by enabling the running of a test in a single-line command, with everything else taken care of transparently. The broader functionality of the `mltest` tool revolves around managing access to data and models, keeping track of all test runs, running models, and estimating accuracy scores. We now present the implementation details of the `mltest` tool.

3.1 Architecture

Figure 5 presents an architectural overview of the `mltest` tool. The `TestRunner` class implements the `run` command and thus hosts the main loop of the system. We define three other interfaces that host major functional components: `DataSource`, `RunLogger`, and `ModelRunner`. This design enables us to host states that represent data, run logs, and models in various locations. Adding a new storage endpoint (for any of these) therefore comes down to providing a concrete implementation of the corresponding interface. Given this separation of interfaces, we are able to mix and match various storage endpoints. For each of these interfaces, we also provide one default implementation that uses `git` as an endpoint for storing data, models, and logs. Git is a convenient system for storing the state because it enables the state to live and evolve together with the main code. Each `git`-based endpoint stores its entire state in a separate branch on the same git repository. Even though (at least a portion of) this state should be inaccessible to the developer, we assume the developers are well-behaving and will not peek into the hidden branch. If extra security is required, the content of this branch could also be encrypted. To increase the modularity and adaptability of our system, we also define the (test) `Condition` as an interface to enable modifying the way of processing test conditions.

3.2 Components

In this section we walk through the four main components of the `mltest` tool mentioned above and discuss the most interesting details (of the default implementations).

3.2.1 Data Source. As mentioned in Section 2.3, our assumptions about a dataset include it being a collection of i.i.d. data examples and that each data example contains one or more *features* and a single target *label*. For simplicity, we restrict ourselves to tabular data and we use Pandas `DataFrame` as a data interface.

We define the `DataSource` interface to provide an abstraction over an arbitrary endpoint that hosts the datasets we use for testing. This endpoint needs to provide the ability for the pool of data to incrementally grow in size by depositing new test examples, to prepare (or stage) a subset of that pool for testing and to load the staged subset. Our current implementation assumes the schema of the data remains constant over time. We expect this to be sufficient for a lot of scenarios because it is still possible to add new features to the model by including them in the feature extraction code. We define the following operations in this interface:

- `deposit` – Feeds the test data pool by adding one incremental batch of data to it. We expect this method to be invoked each time the **data curator** prepares new test data. All deposited data becomes available for staging.
- `stage` – Removes a batch of data examples from the test data pool and produces from it a new ***staged dataset*** that can be used for running tests. All stages have unique keys and represent distinct sets of data examples. The `stage` operation takes an optional argument *size* that specifies the maximal number of data examples taken from the pool of deposited data to form a new stage. If omitted, all available unstaged data will be added to the new stage. A stage can be loaded by using its unique stage key.
- `load` – Returns a staged dataset identified by a stage key. If the stage key is omitted, the last staged dataset is returned.
- `get_size, get_staged_size` – Returns the number of available examples in the unstaged data pool, and the number of examples in a stage identified by the stage key (or the latest stage if the key is omitted).
- `get_keys` – Returns the keys of all stages ever created on a given data source, in chronological order.

We provide a `git`-based implementation of the `DataSource` interface, named `GitDataSource`. It assumes that all test data is stored on an arbitrary branch of a `git` repository. By default this would be a separate branch on the main repository that hosts all other code, but this is configurable. To store large files, we can configure it with `git` Large File Storage (LFS) [4]. The `git` data source keeps all data under a single directory, and creates one subdirectory *per stage* where the name corresponds to the stage key. The data files are stored as JSON-serialized Pandas `DataFrame` instances.

3.2.2 Model Runner. Our system works with trained models that take input features and predict corresponding target labels. They are assumed to be able to contain additional feature extraction components that preprocess the input features before feeding them into an actual trainable model. These models can be hosted on an arbitrary endpoint, which is why we use the `ModelRunner` interface to abstract them away and expose a minimal interface.

A model runner exposes a collection of models, each of which can be identified by a unique model *key*. In the context of CI, each different version of a model will have a unique key. We assume that all versions belong to the same ML task and have been trained on datasets following the same underlying data distribution. We define the following operations in the `ModelRunner` interface:

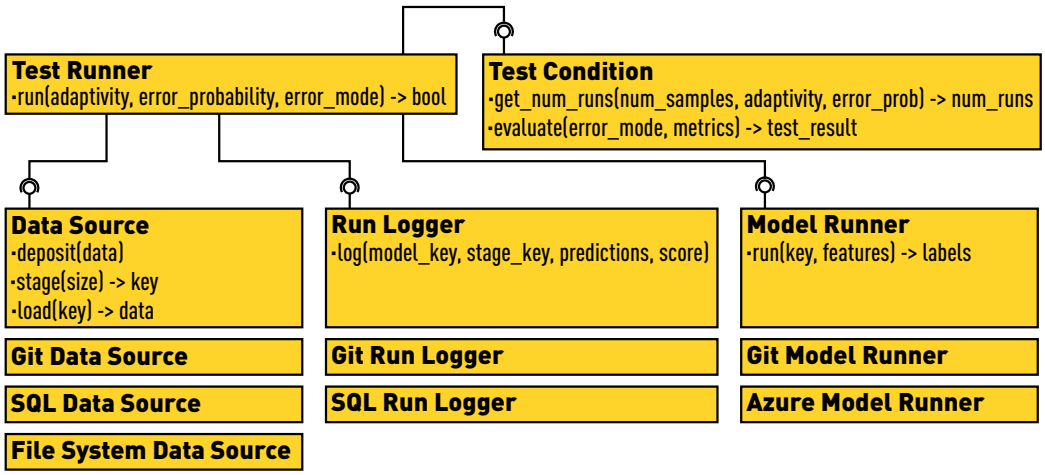


Figure 5: An architectural overview of the implementation of the `mltest` tool.

- `run` – Invokes the `predict` function of a model identified by the key. If the key is omitted, the latest version of the model is targeted. We pass the features of a batch of data examples and the targeted model returns the predicted labels.
- `get_keys` – Returns the keys of all (versions of) models that are available on a given model storage endpoint, in chronological order.

We again provide a `git`-based implementation of the `ModelRunner` interface, named `GitModelRunner`. It assumes that all models are committed to an arbitrary branch of a single `git` repository. The model keys thus correspond to the hashes of all commits that contain updates to the model — we want to avoid treating updates to non-model files as new model versions. To run a model with a specific key, the `git` model runner checks out the commit based on the hash and invokes its `predict` command. This command is an input parameter of the `git` model runner.

3.2.3 Run Logger. We use the `RunLogger` interface to maintain a unified record of all test runs that took place for a given data source and a given model runner. A test run is implemented by the `Run` class which has a unique *run key*, as well as the corresponding *model key* and *stage key*. It also holds all predictions generated by a model, the accuracy score of the model, and the test outcome which is a Boolean `pass/fail` indicator.

The pool of test runs can be queried by model key and/or by stage key. The query can either return serialized run descriptors or simply return the count of runs that satisfy a predicate. We define the following operations in the `RunLogger` interface:

- `log` – Submits a new run to the log. This method accepts the targeted model key and stage key, as well as the predictions, the score, and the test outcome of the corresponding model. Once a run is created, its run key is returned. Runs can be queried either individually or collectively.
- `get_run` – Returns a run identified by its run key.
- `get_runs, count_runs` – Queries the whole pool of runs by using a specific model key or stage key, or both, as a search criterion. The `get_runs` operation returns the run instances, whereas the `count_runs` simply counts the number of runs satisfying the search criterion.

The most important operation is `count_runs` that returns the number of runs for a given stage, because it enables our system to impose limits on test runs over a single test dataset.

Once again we provide an implementation of the `RunLogger` interface with a `git`-based approach, named `GitRunLogger`. Just like the `GitDataSource` implementation, it stores all runs in some branch of a specified `git` repository. By default, this would be a separate branch of the `git` repository that hosts the model. Serialized runs are stored as JSON files in a specified directory.

3.2.4 Test Condition. We implement the `Condition` class in order to encapsulate the functionality required to work with test conditions. We need to be able to parse them from a string representation, use them to compute the number of permitted test runs for a given number of samples, as well as use accuracy estimates that are treated as random variables and evaluate a test condition to compute a `pass/fail` outcome.

Following `ease.ml/ci`, we define a *domain specific language* (DSL) that allows expressing test conditions in a compact way. The DSL is simple but is able to encode a large number of test clauses that are interesting in practice. The top-level literal of the DSL is a ***condition***, which is simply a conjunction of clauses:

```
condition := clause "and" condition | clause.
```

The building blocks of each condition is a ***clause*** that is a simple inequality with an expression on the left-hand side and a constant real value on the right-hand side, along with an error margin ϵ:

```
clause := expression (">"|"<") constant "+/-" constant.
```

Here, an ***expression*** is a summation of factors:

```
expression := factor ("+"|"-") expression | factor,
```

where each ***factor*** is made up of a variable multiplied by one or more constants:

```
factor := constant "*" factor | variable.
```

Finally, a ***constant*** can be any real value, and a ***variable*** is one of:

- `n` – the accuracy of the *new* model, i.e., the newly submitted model that we are currently testing;

- `o` – the accuracy of the *old* model, i.e., the previously submitted model that we last tested before the new one;
- `d` – the fractions of predictions that are different between the *new* and the *old* model, used to control model stability.

After we parse a test condition expressed using the above DSL, we are able to perform several useful operations on it:

- `evaluate` – Computes a Boolean `pass`/`fail` outcome of a test condition given the estimated values of the variables, in terms of an *error mode* that defines how to deal with type I and type II statistical errors (details in Section 4.2).
- `get_num_runs` – Computes the *number of test runs* permitted on the (staged) test dataset in order to protect it from overfitting, given an *error probability* that defines the plausibility of the test outcome (since the outcome itself is a random variable), an *adaptivity mode* that prescribes the regime by which information will be released to the user, and the *number of samples* we have in a (staged) test set (details in Section 4.1).
- `get_num_samples` – Works similarly as `get_num_runs`, with the only difference that it returns the *number of samples* needed to support a given *number of test runs* instead of the other way around.

One of our goals is to enable users to get instant feedback while trying out different settings for their test conditions. For this purpose, all functionalities of the `Condition` class are exposed through all three interfacing methods defined in Section 2.3.

4 EVALUATION OF TEST CONDITIONS

We present details of the evaluation of test conditions in this section. Specifically, we provide solutions to the following two problems:

- *Test run estimation* – given the size of the test dataset, estimate the number of runs/submissions that it can support;
- *Probabilistic semantics* – what kind of probabilistic guarantees we can provide for the evaluation outcomes.

We note that the theoretical foundation of the techniques we present here has already been laid out by Renggli et al. [20]. We therefore focus on their implementation in the `mltest` tool.

4.1 The Number of Test Runs

As shown in [20], the number of test examples required for an (ϵ, δ)-guarantee for a single test condition (e.g., that makes an assertion of the model accuracy) is:

$$n(v, \epsilon, \delta) = \frac{-\ln \delta}{2\epsilon^2}. \tag{1}$$

We use v to represent the random variable we are estimating, e.g., the model accuracy. If we have the number of test examples indicated by Equation 1, we can then guarantee that $\mathbb{P}(|\hat{v} - v| \geq \epsilon) \leq \exp\left(-2n\epsilon^2\right) \leq \delta$, where $\hat{v}$ is the estimated value of v.

To support more complex test conditions, however, we need to deal with expressions elaborated in Section 3.2.4. The simplest one is multiplication with a constant:

$$n(c \cdot v, \epsilon, \delta) = n(v, \epsilon/c, \delta). \tag{2}$$

For a summation or difference we have:

$$\begin{aligned} n(v_1 \pm v_2, \epsilon, \delta) = &\max\{n(v_1, \epsilon_1, \delta/2), n(v_2, \epsilon_2, \delta/2)\} \\ &\text{s.t. } \epsilon = \epsilon_1 + \epsilon_2. \end{aligned} \tag{3}$$

In both cases we need to solve an optimization problem with the constraint $\epsilon = \epsilon_1 + \epsilon_2$. This permits us to construct arbitrary clauses defined in Section 3.2. Finally, we want to handle a test condition that is a conjunction over multiple clauses C_1, ..., C_k. The number of samples we need for the conjunction is equal to the number of samples to evaluate the hardest clause in the conjunction:

$$n(C_1 \wedge \ldots \wedge C_k, \epsilon, \delta) = \max_i n(C_i, \epsilon, \delta/k). \tag{4}$$

4.1.1 Adaptivity Modes. So far, we have been focusing on estimating the number of samples for a single test run. If we wish to use the same test dataset to support multiple test runs, then it may lose its statistical power (as a representative of the true data distribution) due to the presence of adaptivity. In our system, we provide two *adaptivity mode*s: the **non-adaptive** mode and the **full-adaptive** mode. The number of test runs that can be supported depends on the mode our system operates in.

***Non-adaptive** Mode.* In this mode, no information is ever revealed, i.e., the outcome of the test is completely hidden. This mode is rarely useful in practice based on conversation with our partners. Nonetheless, it serves as a special (and the unique) case where we can safely treat all submitted models as *independent*, which significantly increases the number of test runs (supported by a certain test data size). Suppose that we want to run N independent models over a single staged test set. By the union bound, it follows that

$$\delta = \bigcup_{i=1}^{N} \delta_i \leq \sum_{i=1}^{N} \exp\left(-2n\epsilon^2\right) = N \cdot \exp\left(-2n\epsilon^2\right). \tag{5}$$

This means that the required test dataset size for N models is:

$$K = n(v, \epsilon, \delta/N) = \frac{-\ln(\delta/N)}{2\epsilon^2}. \tag{6}$$

For a given K it thus follows that $N = \delta e^{K \cdot 2\epsilon^2}$.

***Fully-adaptive** Mode.* In this mode, after every evaluation the test outcome is released, which is typical in CI scenarios. By Figure 3, such information leakage creates a dependency between models submitted over the same staged test set. As a result, the union bound and thus the test set size derived in Equation 6 do not apply.

As was shown in [20], by using an idea similar to the Ladder mechanism [11], we can, however, apply the union bound over all 2^N possible (binary) sequences that represent the whole space of possible test outcomes from N submitted models:

$$\delta = \bigcup_{i=1}^{2^N} \delta_i \leq \sum_{i=1}^{2^N} \exp\left(-2n\epsilon^2\right) = 2^N \cdot \exp\left(-2n\epsilon^2\right). \tag{7}$$

As a result, the required test data size is

$$K = n\left(v, \epsilon, \delta/2^N\right) = \frac{-\ln(\delta/2^N)}{2\epsilon^2}, \tag{8}$$

which gives $N = \frac{\ln \delta + K \cdot 2\epsilon^2}{\ln 2}$.

4.2 Probabilistic Semantics

The major functionality of test conditions is to produce a reliable `pass`/`fail` signal. In our context, all values that we pass to our three variables `n`, `o`, and `d` are random variables, which implies that the evaluation outcomes should also be treated as random variables. How do we, then, interpret the semantics (i.e., probabilistic guarantees) of the evaluation outcomes?

Consider, for example, the test condition `n > 0.6 +/- 0.05`. If we measured `n` to be 0.7, should this clause be evaluated as `true`? The answer is yes, but only if we used at least the amount of samples prescribed by Equation 1. That bound provides us with a statistical guarantee that, with probability δ, our estimate of `n` will *not* be more than ϵ-away from the true `n`. Since in this case $\epsilon = 0.05$, it is safe to evaluate this clause as `true`.

Now, consider the case if we measured `n` to be 0.61. Since this is *within* ϵ-distance from 0.6, we cannot rely on our statistical guarantees. When these situations occur, it is not possible to avoid sometimes returning a wrong result and it is not possible to know when the result is wrong. In such cases, the best we can do is to consider a trade-off and choose between false positives and false-negatives. In other words, we can choose to either be *false positive free* and always treat these unknown situations by returning `false`, or be *false negative free* and always return `true`. Accordingly, we further provide two *error mode* settings in the `mltest` tool, which can be either `fp-free` or `fn-free`.

5 EVALUATION

We try to answer two main questions in our evaluation:

(1) What is the danger of using a static test set in CI and how can our method help?
(2) Given a test condition and our methods presented in Section 4, how many test runs can be performed against a single staged test set, with what guarantees and at what price?

5.1 Battle Against Overfitting

We simulate a *develop-commit-test* scenario in this experiment. We set up a test system that accepts a new model only if its estimated test score is greater than the current best model. We take a real dataset that represents a classification task regarding the presence of Higgs bosons in a physical process [10]. We split this dataset into a training set (with 20k data examples), a running test set of size N that is used by our test system, and a large test set with *one million* data examples that we use to estimate the "true test score." Each time the developer submits a model, we run a test procedure and send the `pass/fail` outcome back to the developer.

We simulate the developer's submission activities as a random process that first randomly picks a model from a set of available classifiers and then randomly picks hyperparameter values from predefined ranges. The set of candidate classifiers includes *random forest*, *extra trees*, *decision tree*, *k-nearest neighbors*, and *linear SVM*.

As a baseline approach for comparison, we simulate a scenario where the dataset used for testing remains unchanged over the entire duration of a development lifecycle. We use a static test set of size 500. As depicted in Figure 6, it becomes clear that, as time goes by, the estimated test score begins to *diverge* slowly from the true test score. We thus conclude that, if we use a static test set in a "CI for ML" system, in certain scenarios we may end up in a situation where developers can overfit the test set even though only a single bit of information is disclosed per test run.

In Figure 7 we examine how our approach manages to deal with the overfitting problem. The test condition and the development cycle remain the same. The only difference is that now we use our stage-based method for estimating test scores. We conduct three

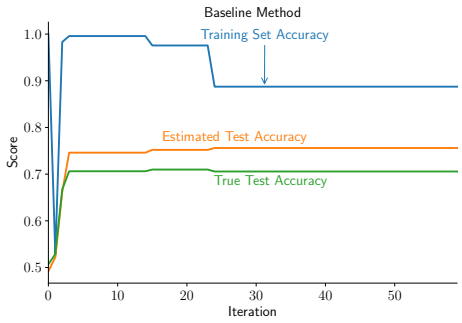


Figure 6: A scenario where the developer overfits the test set.

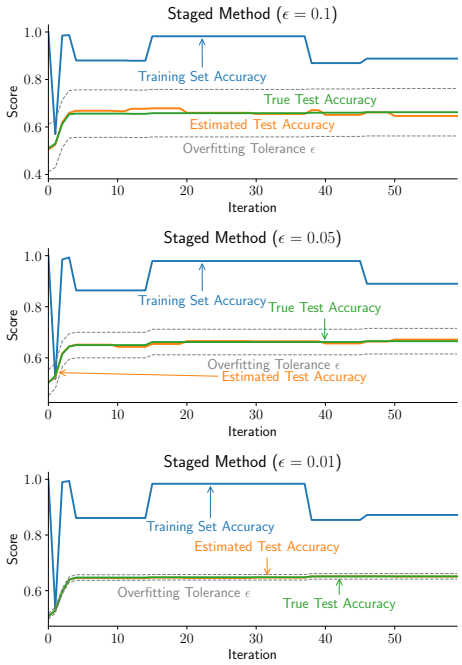


Figure 7: Our strategy ensures that the estimated test error will be within the ϵ-distance of the true test error.

experimental runs, each time ensuring that the estimated test score remains within ϵ-distance from the true test score (with probability $1 - \delta = 0.99$). We try out three values of ϵ: 0.1, 0.05, and 0.01. Since we run 10 tests for each staged test set, by Equation 8 the staged test sets need to contain 577, 2307, and 57683 data examples respectively, under the ***full-adaptive*** mode. It is easy to observe that the estimates do indeed always stay within bounds of our ϵ-margin (shown by the dotted lines).

5.2 The Cost of Running Staged Tests

We present an evaluation of our method with the goal of gaining intuition about the cost of running staged tests. We measure how many runs can be performed on a test set of a given size with required quality guarantees.

We study the impact of the input parameters to the method we use for computing the number of runs (described in 4.1). We try out three test set sizes with 50k, 100k, and 500k data examples. We try two representative test condition categories: One that checks if

	test condition:		n - o > 0.1 +/- 0.1	n > 0.5 +/- 0.1
test set size	ϵ	δ	# of runs	# of runs
$50k$	0.01	0.0001	n/a	2 ($208)
		0.001	n/a	5 ($83)
		0.01	n/a	8 ($52)
	0.025	0.0001	8 ($52)	76 ($5)
		0.001	11 ($37)	80 ($5)
		0.01	14 ($30)	84 ($5)
$100k$	0.01	0.0001	n/a	16 ($46)
		0.001	n/a	18 ($36)
		0.01	n/a	23 ($5)
	0.025	0.0001	30 ($28)	168 ($5)
		0.001	34 ($24)	170 ($5)
		0.01	38 ($22)	174 ($5)
$500k$	0.01	0.0001	21 ($198)	130 ($32)
		0.001	25 ($166)	134 ($31)
		0.01	29 ($143)	138 ($30)
	0.025	0.0001	211 ($20)	889 ($5)
		0.001	215 ($19)	891 ($5)
		0.01	217 ($19)	895 ($5)

Table 1: Number of runs and (in braces) a price estimate for a single run for various test set sizes, and (ϵ, δ) values. Cells marked as n/a correspond to insufficient test set sizes.

the test score is above a certain threshold, and another that checks if there is an improvement in the test score when comparing two models. For each test condition, we try out different combinations of the error margin ϵ (taken to be either 0.025 or 0.01) and the error probability δ (taken to be either 1%, 0.1%, or 0.01%).

We associate each test run with a hypothetical "dollar price" as follows. We assume a labelling effort that is conducted at the speed of 5 seconds per label, and at the price of $6 per hour (number taken from `labellbox.com`). At this rate, 720 data examples can be labelled per hour. We do not include the price of acquiring features, though it is not negligible. We argue that the overall price is dominated by the labelling cost since each data example requires human effort. With this setting, we can then compute the price of a test set (and thus the test run). Table 1 presents the results.

6 RELATED WORK

CI has been an industrial standard in practice and the literature on classic CI in software engineering is overwhelming [12]. However, so far little work has been done towards "CI for ML," although there have been emerging discussions in online communities regarding such requirements [8, 17, 18, 26]. The recent effort by Renggli et al. [20, 21] is the first work along this line, as far as we know. It lays out theoretical foundations as well as building a proof-of-concept system to demonstrate the feasibility of "CI for ML." The current paper, meanwhile, takes one step further by addressing the design, implementation, data management, integration challenges for developing an industrial-strength "CI for ML" service.

The "CI for ML" idea also fits well into the broader scope of building AutoML systems and services. Users of AutoML systems only need to provide their data and high-level specifications of their ML tasks (e.g., loss functions to be minimized), and the system will take over the rest of the job, such as automatic pipeline execution, resource allocation, and performance monitoring. Typical AutoML systems include industrial offerings from major cloud service providers such as Amazon SageMaker [1], Microsoft Azure Machine Learning [6], and Google Cloud AutoML [5], as well as ones from academic institutions such as the Northstar system developed at MIT [16], and the `ease.ml` service [15, 19, 27] by ETH Zurich, among others. It remains interesting to see how to incorporate "CI for ML" into these existing AutoML services.

7 CONCLUSION

We have presented our efforts and experiences with building an industrial-strength "CI for ML" service. We discussed the details of its design, implementation, data management, and integration, as well as evaluation over real datasets. We showcased the risk of overfitting a static test set in the context of "CI for ML" that motivated us to come up with the "staged test set" solution, and demonstrated its affordable cost in practice.

REFERENCES

[1] Amazon sage maker. https://aws.amazon.com/sagemaker/.
[2] Aws codepipeline. https://aws.amazon.com/codepipeline/.
[3] Azure devops services. https://azure.microsoft.com/en-us/services/devops/.
[4] Git LFS. https://git-lfs.github.com/.
[5] Google cloud automl. https://cloud.google.com/automl/.
[6] Microsoft azure machine learning. https://azure.microsoft.com/en-us/services/machine-learning/.
[7] `mltest` tool open-source repository on github. https://aka.ms/gsl-ml-test.
[8] Continuous integration for machine learning. https://medium.com/@rstojnic/continuous-integration-for-machine-learning-6893aa867002, April 2018.
[9] S. Ackermann et al. Using transfer learning to detect galaxy mergers. *MNRAS*, 2018.
[10] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.
[11] A. Blum and M. Hardt. The ladder: A reliable leaderboard for machine learning competitions. In *ICML*, pages 1006–1014, 2015.
[12] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[13] C. Dwork et al. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015.
[14] I. Girardi et al. Patient risk assessment and warning symptom detection using deep attention-based neural networks. *LOUHI*, 2018.
[15] B. Karlas, J. Liu, W. Wu, and C. Zhang. Ease.ml in action: Towards multi-tenant declarative learning services. *PVLDB*, 11(12):2054–2057, 2018.
[16] T. Kraska. Northstar: An interactive data science system. *PVLDB*, 11(12):2150–2164, 2018.
[17] A. F. Lara. Continuous integration for ml projects. https://medium.com/onfido-tech/continuous-integration-for-ml-projects-e11bc1a4d34f, October 2017.
[18] A. F. Lara. Continuous delivery for ml models. https://medium.com/onfido-tech/continuous-delivery-for-ml-models-c1f9283aa971, July 2018.
[19] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *PVLDB*, 11(5):607–620, 2018.
[20] C. Renggli et al. Continuous integration of machine learning models with ease.ml/ci: Towards a rigorous yet practical treatment. In *SysML*, 2019.
[21] C. Renggli, F. A. Hubis, B. Karlas, K. Schawinski, W. Wu, and C. Zhang. Ease.ml/ci and ease.ml/meter in action: Towards data management for statistical generalization. *PVLDB*, 12(12):1962–1965, 2019.
[22] K. Schawinski et al. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. *MNRAS*, 2017.
[23] K. Schawinski et al. Exploring galaxy evolution with generative models. *Astronomy & Astrophysics*, 2018.
[24] D. Stark et al. PSFGAN: a generative adversarial network system for separating quasar point sources and host galaxy light. *MNRAS*, 2018.
[25] M. Su et al. Generative adversarial networks as a tool to recover structural information from cryo-electron microscopy data. *BioRxiv*, 2018.
[26] D. Tran. Continuous integration for data science. http://engineering.pivotal.io/post/continuous-integration-for-data-science/, February 2017.
[27] C. Zhang, W. Wu, and T. Li. An overreaction to the broken machine learning abstraction: The ease.ml vision. In *HILDA@SIGMOD 2017*, pages 3:1–3:6, 2017.

Empirical Software Engineering (2020) 25:2379–2415
https://doi.org/10.1007/s10664-019-09794-7

An approach and benchmark to detect behavioral changes of commits in continuous integration

Benjamin Danglot[1] · **Martin Monperrus**[2] · **Walter Rudametkin**[3] · **Benoit Baudry**[2]

Published online: 5 March 2020

Abstract

When a developer pushes a change to an application's codebase, a good practice is to have a test case specifying this behavioral change. Thanks to continuous integration (CI), the test is run on subsequent commits to check that they do no introduce a regression for that behavior. In this paper, we propose an approach that detects behavioral changes in commits. As input, it takes a program, its test suite, and a commit. Its output is a set of test methods that capture the behavioral difference between the pre-commit and post-commit versions of the program. We call our approach DCI (Detecting behavioral changes in CI). It works by generating variations of the existing test cases through (i) assertion amplification and (ii) a search-based exploration of the input space. We evaluate our approach on a curated set of 60 commits from 6 open source Java projects. To our knowledge, this is the first ever curated dataset of real-world behavioral changes. Our evaluation shows that DCI is able to generate test methods that detect behavioral changes. Our approach is fully automated and can be integrated into current development processes. The main limitations are that it targets unit tests and works on a relatively small fraction of commits. More specifically, DCI works on commits that have a unit test that already executes the modified code. In practice, from our benchmark projects, we found 15.29% of commits to meet the conditions required by DCI.

Keywords Continuous Integration · Test amplification · Behavioral change detection

Communicated by: Tao Yue

✉ Benjamin Danglot
bdanglot@gmail.com

Martin Monperrus
martin.monperrus@csc.kth.se

Walter Rudametkin
walter.rudametkin@inria.fr

Benoit Baudry
benoit.baudry@kth.se

[1] INRIA, Lille-Nord Europe, 40 Avenue Halley, Villeneuve d'Ascq, 59650, France

[2] KTH Royal Institute of Technology, Brinellvägen 8, 114 28 Stockholm, Sweden

[3] Université de Lille, 42 rue Paul Duez, 59000 Lille, France

1 Introduction

In collaborative software projects, developers work in parallel on the same code base. Every time a developer integrates her changes, she submits them in the form of a *commit* to a version control system. The *Continuous Integration* (CI) server (Fowler and Foemmel 2006) merges the commit with the master branch, compiles and automatically runs the test suite to check that the commit behaves as expected. Its ability to detect bugs early makes CI an essential contribution to quality assurance (Hilton et al. 2016; Duvall et al. 2007).

However, the effectiveness of Continuous Integration depends on one key property: each commit should include at least one test case t_{new} that specifies the intended change. For instance, assume one wants to integrate a bug fix. In this case, the developer is expected to include a new test method, t_{new}, that specifies the program's desired behavior after the bug fix is applied. This can be mechanically verified: t_{new} should fail on the version of the code that does not include the fix (the *pre-commit* version), and pass on the version that includes the fix (the *post-commit* version). However, many commits either do not include a t_{new} or t_{new} does not meet this fail/pass criterion. The reason is that developers sometimes cut corners because of lack of time, expertise or discipline. This is the problem we address in this paper.

In this paper, we aim to automatically generate test methods for each commit that is submitted to the CI. In particular, we generate a test case t_{gen} that specifies the behavioral change of each commit. We consider a generated test case t_{gen} to be relevant if it satisfies the following property: t_{gen} *passes* on the pre-commit version and *fails* on the post-commit version. To do so, we developed a new approach, called DCI, that works in two steps. First, we analyze the test cases of the pre-commit version and select the ones that exercise the parts of the code modified by the commit. Second, our test generation techniques produce variant test cases that either add assertions (Xie 2006) to existing tests or explore new inputs following a search-based test input generation approach (Tonella 2004). This process of automatic generation of t_{gen} from existing tests is called *test amplification* (Zhang and Elbaum 2012). We evaluate our approach on a benchmark of 60 commits selected from 6 open source Java projects, constructed with a novel and systematic methodology. We analyzed 1576 commits and selected those that introduce a behavioral change (*e.g.*, we do not want to generate tests for commits that only change comments). We also make sure that all selected commits contain a developer-written test case that detects the behavioral change. In our protocol, the developer's test case acts as a ground-truth to analyze the tests generated by DCI. Overall, we found 60 commits that satisfy the two essential properties we are looking for: 1) the commit introduces a behavioral change; 2) the commit has a human written test we can use for ground truth. This corresponds to 15.3% of commits in average. While this may appear to be a low proportion of commits, our approach is fully automated and developers can still benefit from its output without any manual intervention.

To sum up, our contributions are:

- DCI (**D**etecting behavioral changes in **CI**), an approach based on *test amplification* to generate new tests that detect the behavioral change introduced by a commit.
- An open-source implementation of DCI for Java.
- A curated benchmark of 60 commits that introduce a behavioral change and include a test case to detect it, selected from 6 notable open source Java projects[1].

[1] https://github.com/STAMP-project/dspot-experiments

```
@@ -260,7 +260,8 @@ public boolean equals(Object object)
} else {
    if (object instanceof FilterStreamType) {
        result = Objects.equals(getType(), ((FilterStreamType) object).getType()
            )
-       && Objects.equals(getDataFormat(),
-                         ((FilterStreamType) object).getDataFormat());
+       && Objects.equals(getDataFormat(),
+                         ((FilterStreamType) object).getDataFormat())
+       && Objects.equals(getVersion(),
+                         ((FilterStreamType) object).getVersion());
    } else {
        result = false;
    }
```

Listing 1 Commit 7E79F77 on XWiki-Commons that changes the behavior without a test

– A comprehensive evaluation based on 4 research questions that combines quantitative and qualitative analysis with manual assessment.

In Section 2 we motivate the need to have commits include a test case that specifies the behavioral change. In Section 3 we introduce our technical contribution: an approach for commit-based test selection and amplification. Section 4 introduces our benchmark of commits, the evaluation protocol and the results of our experiments on 60 real commits. Section 5 discusses the exact applicability scope of our approach. Section 6 presents the threats validity and actions that have been taken to overcome them. In Section 7, we expose the related work, their evaluation and the differences with our work and eventually we conclude in Section 8.

2 Motivation & Background

In this section, we introduce an example to motivate the need to generate new tests that specifically target the behavioral change introduced by a commit. Then we introduce the key concepts on which we elaborate our solution to address this challenging test generation task.

2.1 Motivating Example

On August 10, a developer pushed a commit to the master branch of the XWiki-commons project. The change[2], displayed in Listing 1, adds a comparison to ensure the equality of the objects returned by `getVersion()`. The developer did not write a test method nor modify an existing one.

In this commit, the intent is to take into account the `version` (from method `getVersion`) in the `equals` method. This change impacts the behavior of all methods that use it, `equals` being a highly used method. Such a central behavioral change may impact the whole program, and the lack of a test case for this new behavior may have dramatic consequences in the future. Without a test case, this change could be reverted and go undetected by the test suite and the Continuous Integration server, *i.e.* the build would still pass. Yet, a user of this program would encounter new errors, because of the changed behavior. The developer took a risk when committing this change without a test case.

[2]https://github.com/xwiki/xwiki-commons/commit/7e79f77

Our work on automatic test amplification in continuous integration aims at mitigating such risk: test amplification aims at ensuring that every commit include a new test method or a modification of an existing test method. In this paper, we study how to automatically obtain a test method that highlights the behavioral change introduced by a commit. This test method allows to identify the behavioral difference between the two versions of the program. Our goal is to use this new test method to ensure that any changed behavior can be caught in the future.

What we propose is as follows: when Continuous Integration is triggered, rather than just executing the test suite to find regressions, it could also run an analysis of the commit to know if it contains a behavioral change, in the form of a new method or the modification of an existing one. If there is no appropriate test case to detect a behavioral change, our approach would provide one. DCI would take as input the commit and a test suite, and generate a new test case that detects the behavioral change.

2.2 Practibility

We describe a complete scenario to sum up the vision of our approach's usage.

A developer commits a change into the program. The Continuous Integration service is triggered; the CI analyzes the commit. There are two potential outcomes:

1) the developer provided a new test case or a modification to an existing one. In this case, the CI runs as usual, *e.g.* it executes the test suite;
2) the developer did not provide a new test nor the modification of an existing one, the CI runs DCI on the commit to obtain a test method that detects the behavioral change and present it to the developer.

The developer can then validate the new test method that detects the behavioral change. Following our definition, the new test method passes on the pre-commit version but fails on the post-commit version. The current amplified test method cannot be added to the test suite, since it fails. However, this test method is still useful, since one has only to negate the failing assertions, *e.g.* change an `assertTrue` into an `assertFalse`, to obtain a valid and passing test method that explicitly executes the new behavior. This can be done manually or automatically with approaches such as ReAssert (Daniel et al. 2009).

DCI could apply to any kind of test: unit-level or system-level. However, from our experience, unit tests (vs integration tests) are the best target for DCI, for two reasons. First, they have a small scope, which allows DCI to intensify its search, while an integration test, that contains a lot of code, would make DCI explore the neighborhood in different ways. Second, that is a consequence of the first, the unit tests are fast to execute compared to integration tests.

Since DCI needs to execute the tests 5 times under amplification, it means that DCI would be executed faster when it amplifies unit tests than when it amplified integration tests.

DCI has been designed to be easy to use. The only cost of DCI is the time to set it up: in the ideal, happy-path case, it is meant to be a single command line through Maven goals. Once DCI is set up in continuous integration, it automatically runs at each commit and developers directly benefit from amplified test methods that strengthen the existing test suite.

2.3 Behavioral Change

A *behavioral change* is a source-code modification that triggers a new state for some inputs (Saff and Ernst 2004). Considering the pre-commit version P and the post-commit version P' of a program, the commit introduces a behavioral change if it is possible to implement a test case that can trigger and observe the change, *i.e.*, it passes on P and fails on P', or the opposite. In short, the behavioral change must have an impact on the observable behavior of the program.

2.4 Behavioral Change Detection

Behavioral change detection is the task of identifying or generating a test or an input that distinguishes a behavioral change between two versions of the same program. In this paper, we propose a novel approach to detect behavioral changes based on test amplification.

2.5 Test Amplification

Test amplification is the idea of improving existing tests with respect to a specific test criterion (Zhang and Elbaum 2012). We start from an existing test suite and create variant tests that improve a given test objective. For instance, a test amplification tool may improve the code coverage of the test suite. In this paper, our test objective is to improve the test suite's detection of behavioral changes introduced by commits.

3 Behavioral Change Detection Approach

We propose an approach to produce test methods that detect the behavioral changes introduced by commits. We call our approach DCI (**D**etecting behavioral changes in **CI**), and propose to use it during continuous integration.

3.1 Overview of DCI

DCI takes as input a program, its test suite, and a commit modifying the program. The commit, as done in version control systems, is basically the diff between two consecutive versions of the program.

DCI outputs new test methods that detect the behavioral difference between the pre- and post-commit versions of the program. The new tests pass on a given version, but fail on the other, demonstrating the presence of a behavioral change captured.

DCI computes the code coverage of the diff and selects test methods accordingly. Then, it applies two kinds of test amplification to generate new test methods that detect the behavioral change. Figure 1 sums up the different phases of the approach:

1) Compute the diff coverage and select the test methods to be amplified;
2) Amplify the selected tests based on the pre-commit version;
3) Execute amplified test methods against the post-commit version, and keep the failing test methods.

This process produces test methods that pass on the pre-commit version, fail on the post-commit version, hence they detect at least one behavioral change introduced by a given commit.

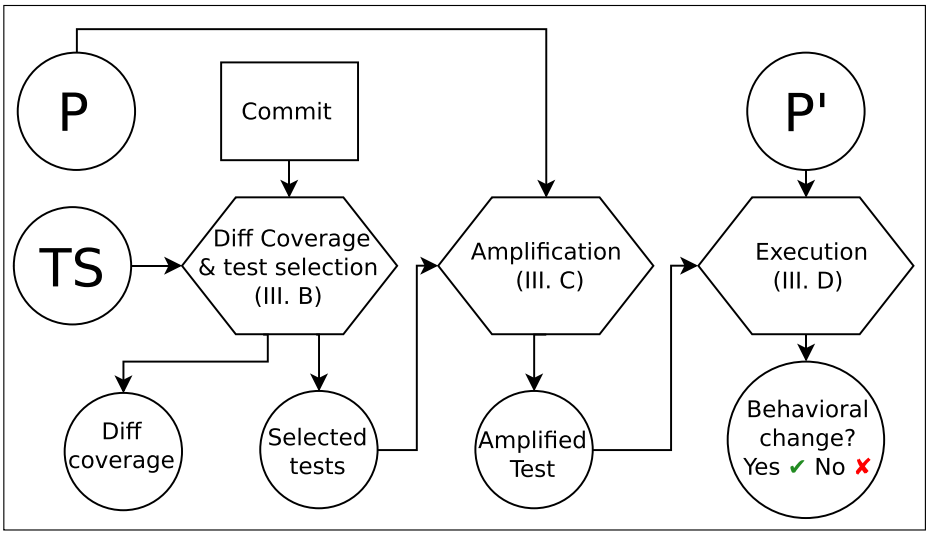


Fig. 1 Overview of our approach to detect behavioral changes in commits

3.2 Test Selection and Diff Coverage

DCI implements a feature that: 1. reports the diff coverage of a commit, and 2. selects the set of unit tests that execute the diff.

To do so, DCI first computes the code coverage for the whole test suite.

Second, it identifies the test methods that hit the statements modified by the diff.

Third, it produces the two outcomes elicited earlier: the diff coverage, computed as the ratio of statements in the diff covered by the test suite over the total number of statements in the diff and the list of test methods that cover the diff.

Then, we select only test methods that are present in pre-commit version (*i.e.*, we ignore the test methods added in the commit, if any). The final list of test methods that cover the diff is then used to seed the amplification process.

3.3 Test Amplification

Once we have the initial tests that cover the diff, we want to make them detect the behavioral change and assess the new behavior. This process of extending the scope of a test case is called test amplification (Zhang and Elbaum 2012). In DCI, we build upon Xie's technique (Xie 2006) and Tonella's evolutionary algorithm (Tonella 2004) to perform test amplification.

3.3.1 Assertion Amplification

A test method consists of a setup and assertions. The former is responsible for putting the program under test into a specific state; the latter is responsible for verifying that the actual state of the program at the end of the test is the expected one. To do this, assertions compare actual values against expected values: if the assertion holds, the program is considered correct, if not, the test case has revealed the presence of a bug.

Assertion amplification (AAMPL) has been proposed by Xie (2006). It takes as input a program and its test suite, and it synthesizes new assertions on public methods that capture the program state. The targeted public methods are those that take no parameter, return a result, and match a Java naming convention of getters, *e.g.* the method starts with *get* or *is*. The standard method *toString()* is also used. If a method used returns a complex Java Object, AAMPL recursively uses getters on this object to generate deeper assertions.

In case the test method sets the program into an incorrect state and an exception is thrown, AAMPL generates a test for this exception by wrapping the test method body in a *try/catch* block. It also inserts a *fail* statement at the end of the body of the *try*, *i.e.* it means that if the exception is not thrown the test method fails.

Algorithm 1 AAMPL: Assertion amplification algorithm.

Require: Program P
Require: Test Suite TS
Ensure: An Amplified Test Suite ATS
1: $ATS \leftarrow \emptyset$
2: **for** $Test$ in TS **do**
3: $NoAssertTest \leftarrow removeAssertions(Test)$
4: $InstrTest \leftarrow instrument(NoAssertTest)$
5: $execute(InstrTest)$
6: $AmplTest \leftarrow NoAssertTest.clone()$
7: **for** $Observ$ in $InstrTest.observations()$ **do**
8: $Assert \leftarrow generateAssertion(Observ)$
9: $AmplTest \leftarrow AmplTest.add(Assert)$
10: **end for**
11: $ATS.add(select(AmplTest))$
12: $ATS.add(AmplTest)$
13: **end for**
14: **return** ATS

We present AAMPL's pseudo-code in Algorithm 1. First, it initializes an empty set of tests ATS (Line 1). For each $Test$ method in the test suite TS (Line 2), it removes the existing assertions to obtain $NoAssertTest$ (Line 3). Then, it instruments $NoAssertTest$ with observation points (Line 4) that allow retrieving values from the program at runtime, which results in $InstrTest$. In order to collect the values, it executes $InstrTest$ (Line 5). Eventually, for each observation $Observ$ of the set of observations from $InstrTest$ (Line 7 to 10), it generates an assertion (Line 8) and adds it to the amplified tests $AmplTest$ (Line 9). At the end, it selects amplified test according to a specific test criterion using the method $select()$ (Line 11) and add selected amplified test methods to the set of test methods $AmplTest$, in other words, an amplified test suite (Line 13).

To sum up, AAMPL increases the number of assertions. By construction, it specifies more behaviors than the original test suite. DCI_{AAMPL} is the AAMPL mode for DCI.

3.3.2 Search-based Amplification

Search-based test amplification consists in running stochastic transformations on test code (Tonella 2004).

For DCI_{SBAMPL}, this process consists in

a) generating a set of original test methods by applying code transformations;
b) executing AAMPL to synthesize new assertions for these test methods for which the input has been modified at the previous step;
c) repeating this process nb times[3], each time seeding with the previously amplified test methods.

This final step allows the search-based algorithm to explore more inputs, and thus improve the chance of triggering new behaviors.

Algorithm 2 SBAMPL: Search based amplification algorithm.

Require: Program P
Require: Program P'
Require: Test Suite TS
Require: Iterations number Nb
Ensure: An Amplified Test Suite ATS

```
1:  ATS ← ∅
2:  TmpTests ← ∅
3:  for Test in TS do
4:      TmpTests ← Test
5:      for i ← 0, i < Nb do
6:          TransformedTests ← transform(TmpTests)
7:          AmplifiedTests ← aampl(TransformedTests)
8:          ATS.add(select(AmplifiedTests))
9:          TmpTests ← AmplifiedTests
10:     end for
11: end for
12: return ATS
```

We present the search-based amplification algorithm in Algorithm 2. This algorithm is a basic Hill Climbing algorithm. It takes as input a program with two distinct versions P and P', its test suite TS and a number of iterations nb, (in our case $nb = 3$). It produces an amplified test suite that contains test methods that pass on P but fail on P'. To do so, it initializes an empty set of amplified test methods ATS (Line 1), which will be the final output, and $TmpTests$ (Line 2) which is a temporary set. Then, for each test method in the test suite TS (Line 3), it applies the following operations:

1) transform the current set of test methods (Line 6) to obtain $TransformedTests$;
2) apply AAMPL on $TransformedTests$ (Line 7, see Algorithm 2) to obtain $AmplifiedTests$;
3) select amplified test methods using the method $select()$, and add them to ATS (the method $select()$ executes the amplified tests on P' and keeps only tests that fail, *i.e.* that detect a behavioral change);

and Finally, 4) affects $AmplifiedTests$ to $TmpTests$ in order to stack transformations. In our study, we consider the test transformations in Table 1.

[3]by default, $nb = 3$

Table 1 Test transformations considered in our study

Types	Operators
Number	add 1 to an integer minus 1 to an integer replace an integer by zero replace an integer by the maximum value (Integer.MAX_VALUE in Java) replace an integer by the minimum value (`Integer.MIN_VALUE` in Java).
Boolean	negate the value.
String	replace a string with another existing string. replace a string with white space, or a system path separator, or a system file separator. add 1 random character to the string. remove 1 random character from the string. replace 1 random character in the string by another random character. replace the string with a random string of the same size. replace the string with the `null` value.

DCI_{SBAMPL} is the search-based amplification mode for DCI.

3.4 Execution and Change Detection

The final step performed by DCI consists in checking whether that the amplified test methods detect behavioral changes. Because DCI amplifies test methods using the pre-commit version, all amplified test methods pass on this version, by construction. Consequently, for the last step, DCI runs the amplified test methods only on the post-commit version. Every test that fails is in fact detecting a behavioral change introduced by the commit, and is a success. DCI keeps the tests that successfully detect behavioral changes.

3.5 Implementation

DCI is implemented in Java and is built on top of the OpenClover and Gumtree (Falleri et al. 2014) libraries. It computes the global coverage of the test suite with OpenClover, which instruments and executes the test suite. Then, it uses Gumtree to have an AST representation of the diff. DCI matches the diff with the test that executes those lines. Through its Maven plugin, DCI can be seamlessly implemented into continuous integration. DCI is publicly available on GitHub.[4]

4 Evaluation

To evaluate the DCI approach, we design an experimental protocol to answer the following research questions:

- RQ1: To what extent are DCI_{AAMPL} and DCI_{SBAMPL} able to produce amplified test methods that detect the behavioral changes?
- RQ2: What is the impact of the number of iteration performed by DCI_{SBAMPL}?
- RQ3: What is the effectiveness of our test selection method?
- RQ4: How do human and generated tests that detect behavioral changes differ?

[4] https://github.com/STAMP-project/dspot.git

4.1 Benchmark

To the best of our knowledge, there is no benchmark of commits in Java with real behavioral changes in the literature. Consequently, we devise a project and commit selection procedure in order to construct a benchmark for our approach.

Project selection We need software projects that are

1) publicly-available,
2) written in Java,
3) and use continuous integration.

We pick the projects from the dataset in Vera-Pérez et al. (2018) and Danglot et al. (2019), which is composed of mature Java projects from GitHub.

Commit selection We take commits in inverse chronological order, from newest to oldest. On September 10 2018, we selected the first 10 commits that match the following criteria:

- The commit modifies Java files (most behavioral changes are source code changes.[5]).
- The changes of the commit must be covered by the pre-commit test suite. To do so, we compute the diff coverage. If the coverage is 0%, we discard the commit. We do this because if the change is not covered, we cannot select any test methods to be amplified, which is what we want to evaluate.
- The commit provides or modifies a manually written test that detects a behavioral change. To verify this property, we execute the test on the pre-commit version. If it fails, it means that the test detects at least 1 behavioral change. We will use this test as a *ground-truth test* in **RQ4**.

Together, these criteria ensure that all selected commits:

1) modify java files,
2) that there is at least 1 test in the pre-commit version of the program that executes the diff and can be used to seed the amplification process
3) provide or modify a manually written test case that detects a behavioral change (which will be used as ground-truth for comparing generated tests), and
4) There is no structural change in the commit between both versions, *e.g.* no change in method signature and deletion of classes (this is ensured since the pre-commit test suite compiles and runs against the post-commit version of the program and vice-versa.)

Final benchmark Table 2 shows the main descriptive statistics of the benchmark dataset. The *project* column is the name of the project. The *LOC* column is the number of lines of code computed with *cloc*. The *start date* column is the date of the project's oldest commit. The *end date* column is the date of the project's newest commit. The *#total commit* column is the total number of commits we analyzed.

#Matching commits is the number of commits that match our first two criteria to run DCI but might not provide a test in the post-commit version that fails on the pre-commit version of the program. We could potentially apply DCI on all *#matching commits*, but for this paper, we cannot validate DCI with them because they might not provide a ground-truth

[5]We are aware that behavioral changes can be introduced in other ways, such as modifying dependencies or configuration files (Hilton et al. 2018).

Table 2 Considered period for selecting commits

Project	LOC	start date	end date	#total commits	#matching commits	#selected commits
commons-io	59607	9/10/2015	9/29/2018	385	49 / 12.73%	10
commons-lang	77410	11/22/2017	10/9/2018	227	40 / 17.62%	10
gson	49766	6/14/2016	10/9/2018	159	56 / 35.22%	10
jsoup	20088	12/21/2017	10/10/2018	50	42 / 84.00%	10
mustache.java	10289	7/6/2016	04/18/2019	68	28 / 41.18%	10
xwiki-commons	87289	10/31/2017	9/29/2018	687	26 / 3.78%	10
summary	304449	9/10/2015	04/18/2019	avg(262.67)	avg(40.17 / 15.29%)	60

test. The *#selected commits* column shows the number of commits we select for evaluation. It is a subset of *#matching commits* from which we searched for the first 10 commits per project that match all criteria, including a ground-truth test to evaluate DCI.

The bottom row reports a summary of the benchmark dataset with the total number of lines of code, the oldest and the newest commit dates, the average number of commits analyzed, the average number of commits matching all the criteria but the third: there is a test in the post-commit version of the program that detect the behavioral change, and the total number of selected commits. The percentage in parenthesis next to the averages are percentage of averages, *e.g.* $\frac{\#matching}{\#total}$. We note that our benchmark is only composed of recent commits from notable open-source projects and is available on GitHub at https://github.com/STAMP-project/dspot-experiments.

4.2 Protocol

To answer **RQ1**, we run DCI_{AAMPL} and DCI_{SBAMPL} on the benchmark projects. We then report the total number of behavioral changes successfully detected by DCI, *i.e.* the number of commits for which DCI generates at least 1 test method that passes on the pre-commit version but fails on the post-commit version. We also discuss 1 case study of a successful behavioral change detection.

To answer **RQ2**, we run DCI_{SBAMPL} for 1, 2 and 3 iterations on the benchmark projects. We report the number of behavioral changes successfully detected for each number of iterations in the main loop. In addition, we want to have a proper understanding of the impact of randomness as follows. We consider the case of $n = 1$ iteration. For "$n = 1$, we run DCI for each commit for 10 different seeds in addition to the reference run with the default seed, totalling 11 runs.. From those runs, we compute the confidence interval on the number of successes, *i.e.* the number of time DCI generates at least one amplified test method that detects the behavioral change, in order to measure the uncertainty of the result. To do this, we use Python libraries *scipy* and *numpy*, and we consider a confidence level of 95%. Per our open-science approach, the interested reader has access to both the raw data and the script computing the confidence interval.[6]

For **RQ3**, the test selection method is considered effective if the tests selected for amplification semantically relate to the code changed by the commit. To assess this, we perform a manual analysis. We randomly select 1 commit per project in the benchmark, and we manually analyze whether the automatically selected tests for this commit are semantically related to the behavioral changes in the commit.

To answer **RQ4**, we use the ground-truth tests written or modified by developers in the selected commits. We manually compare the amplified test methods that detect behavioral changes to the human tests, for 1 commit per project.

4.3 Results

The overall results are reported in Table 3. This table can be read as follows: the first column is the name of the project; the second column is the shortened commit id; the third column is the commit date; the fourth column column is the total number of test methods executed when building that version of the project; the fifth and sixth columns are respectively the number of tests modified or added by the commit, and the size of the diff in terms of line

[6] https://github.com/STAMP-project/dspot-experiments/tree/master/src/main/python/april-2019

Table 3 Performance evaluation of DCI on 60 commits from 6 large open-source projects

	id	date	#Test	#Modified tests	+ / -	Cov	#Selected tests	#AAMPL tests	Time	#SBAMPL tests	Time
commons-io	c6b8a38	6/12/18	1348	2	104 / 3	100.0	3	0	10.0s	0	98.0s
	2736b6f	12/21/17	1343	2	164 / 1	1.79	8	0	19.0s	✓(12)	76.3m
	a4705cc	4/29/18	1328	1	37 / 0	100.0	2	0	10.0s	0	38.1m
	f00d97a	5/2/17	1316	10	244 / 25	100.0	2	✓(1)	10.0s	✓(39)	27.0s
	3378280	4/25/17	1309	2	5 / 5	100.0	1	✓(1)	9.0s	✓(11)	24.0s
	703228a	12/2/16	1309	1	6 / 0	50.0	8	0	19.0s	0	71.0m
	a7bd568	9/24/16	1163	1	91 / 83	50.0	8	0	20.0s	0	65.2m
	81210eb	6/2/16	1160	1	10 / 2	100.0	1	0	8.0s	✓(8)	23.0s
	57f493a	11/19/15	1153	1	15 / 1	100.0	8	0	7.0s	0	54.0s
	5d072ef	9/10/15	1125	12	74 / 34	68.42	25	✓(6)	29.0s	✓(1538)	2.2h
	total						66	8	2.4m	1608	6.5h
	average						6.60	0.80	14.5s	160.80	38.8m
commons-lang	f56931c	7/2/18	4105	1	30 / 4	25.0	42	0	2.4m	0	8.5m
	87937b2	5/22/18	4101	1	114 / 0	77.78	16	0	35.0s	0	18.1m
	09ef69c	5/18/18	4100	1	10 / 1	100.0	4	0	16.0s	0	98.8m
	3fadfdd	5/10/18	4089	1	7 / 1	100.0	9	0	17.0s	✓(4)	17.2m
	e7d16c2	5/9/18	4088	1	13 / 1	33.33	7	0	16.0s	✓(2)	15.1m
	50ce8c4	3/8/18	4084	4	40 / 1	90.91	2	✓(1)	28.0s	✓(135)	2.0m
	2e9f3a8	2/11/18	4084	2	79 / 4	30.0	47	0	79.0s	0	66.5m
	c8e61af	2/10/18	4082	1	8 / 1	100.0	10	0	17.0s	0	16.0s
	d8ec011	11/12/17	4074	1	11 / 1	100.0	5	0	31.0s	0	2.3m
	7d061e3	11/22/17	4073	1	16 / 1	100.0	8	0	17.0s	0	11.4m
	total						150	1	6.7m	141	4.0h
	average						15.00	0.10	40.5s	14.10	24.0m

Table 3 (continued)

	id	date	#Test	#Modified tests	+ / -	Cov	#Selected tests	#AAMPL tests	Time	#SBAMPL tests	Time
gson	b1fb9ca	9/22/17	1035	1	23 / 0	50.0	166	0	4.2m	0	92.5m
	7a9fd59	9/18/17	1033	2	21 / 2	83.33	14	0	15.0s	✓(108)	2.1m
	03a72e7	8/1/17	1031	2	43 / 11	68.75	371	0	7.7m	0	3.2h
	74e3711	6/20/17	1029	1	68 / 5	8.0	1	0	4.0s	0	16.0s
	ada597e	5/31/17	1029	2	28 / 3	100.0	5	0	8.0s	0	8.7m
	a300148	5/31/17	1027	7	103 / 2	18.18	665	0	9.2m	✓(6)	4.9h
	9a24219	4/19/17	1019	1	13 / 1	100.0	36	0	2.2m	0	48.9m
	9e6f2ba	2/16/17	1018	2	56 / 2	50.0	9	0	32.0s	✓(2)	8.5m
	44cad04	11/26/16	1015	1	6 / 0	100.0	2	0	15.0s	✓(37)	40.0s
	b2c00a3	6/14/16	1012	4	242 / 29	60.71	383	0	7.9m	0	3.6h
	total						1652	0	32.4m	153	14.4h
	average						165.20	0.00	3.2m	15.30	86.5m
jsoup	426ffe7	5/11/18	668	4	27 / 46	64.71	27	✓(2)	42.0s	✓(198)	33.6m
	a810d2e	4/29/18	666	1	27 / 1	80.0	5	0	10.0s	0	26.6m
	6be19a6	4/29/18	664	1	23 / 1	50.0	50	0	69.0s	0	67.7m
	e38dfd4	4/28/18	659	1	66 / 15	90.0	18	0	35.0s	0	12.5m
	e9feec9	4/15/18	654	1	15 / 3	100.0	4	0	9.0s	0	95.0s
	0f7e0cc	4/14/18	653	2	56 / 15	84.62	330	0	6.5m	✓(36)	11.8h
	2c4e79b	4/14/18	650	2	82 / 2	50.0	44	0	67.0s	0	4.7h
	e5210d1	12/22/17	647	1	3 / 3	100.0	14	0	9.0s	0	4.9m
	df272b7	12/22/17	647	2	17 / 1	100.0	13	0	9.0s	0	4.6m
	3676b13	12/21/17	648	6	104 / 12	38.46	239	0	6.2m	✓(52)	6.8h
	total						744	2	16.8m	286	25.8h
	average						74.40	0.20	101.0s	28.60	2.6h

Table 3 (continued)

	id	date	#Test	#Modified tests	+ / -	Cov	#Selected tests	#AAMPL tests	Time	#SBAMPL tests	Time
mustache.java	a1197f7	1/25/18	228	1	43 / 57	77.78	131	0	11.8m	✓(204)	10.1h
	8877027	11/19/17	227	1	22 / 2	33.33	47	0	7.3m	0	100.2m
	d8936b4	2/1/17	219	2	46 / 6	60.0	168	0	12.7m	0	84.2m
	88718bc	1/25/17	216	2	29 / 1	100.0	1	✓(1)	7.0s	✓(149)	3.7m
	339161f	9/23/16	214	2	32 / 10	77.78	123	0	8.6m	✓(1312)	5.8h
	774ae7a	8/10/16	214	2	17 / 2	100.0	11	0	66.0s	✓(124)	6.8m
	94847cc	7/29/16	214	2	17 / 2	100.0	95	0	11.5m	✓(2509)	21.4h
	eca08ca	7/14/16	212	4	47 / 10	80.0	18	0	87.0s	0	41.8m
	6d7225c	7/7/16	212	2	42 / 4	80.0	18	0	87.0s	0	40.1m
	8ac71b7	7/6/16	210	10	167 / 31	40.0	20	0	2.1m	✓(124)	5.6m
	total						632	1	58.1m	4422	42.0h
	average						63.20	0.10	5.8m	442.20	4.2h
xwiki-commons	ffc3997	7/27/18	1081	0	125 / 18	21.05	1	0	29.0s	0	18.0s
	ced2635	8/13/18	1081	1	21 / 14	60.0	5	0	93.0s	0	2.5h
	10841b1	8/1/18	1061	1	107 / 19	30.0	51	0	5.7m	0	3.4h
	848c984	7/6/18	1074	1	154 / 111	17.65	1	0	28.0s	0	18.0s
	adfefec	6/27/18	1073	1	17 / 14	40.0	22	✓(1)	76.0s	✓(3)	14.9m
	d3101ae	1/18/18	1062	2	71 / 9	20.0	4	✓(1)	72.0s	✓(31)	41.4m
	a0e8b77	1/18/18	1062	2	51 / 8	42.86	4	✓(1)	72.0s	✓(60)	42.1m
	78ff099	12/19/17	1061	1	16 / 0	33.33	2	0	68.0s	✓(4)	6.6m
	1b79714	11/13/17	1060	1	20 / 5	60.0	22	0	78.0s	0	17.9m
	6dc9059	10/31/17	1060	1	4 / 14	88.89	22	0	79.0s	0	20.5m
	total						134	3	15.7m	98	8.2h
	average						13.40	0.30	94.3s	9.80	49.5m
	total						3378	9(15)	2.2h	25(6708)	100.9h

additions (in green) and deletions (in red); the seventh and eighth columns are respectively the diff coverage and the number of tests DCI selected; the ninth column provides the amplification results for DCI_{AAMPL}, and it is either a ✓ with the number of amplified tests that detect a behavioral change or a - if DCI did not succeed in generating a test that detects a change; the tenth column displays the time spent on the amplification phase; The eleventh and the twelfth are respectively a ✓ with the number of amplified tests for DCI_{SBAMPL} (or - if a change is not detected) for 3 iterations. The last row reports the total over the 6 projects. For the tenth and the twelfth columns of the last row, the first number is the number of successes, *i.e.* the number of times DCI produced at least one amplified test method that detects the behavioral change, for DCI_{AAMPL} and DCI_{SBAMPL} respectively. The numbers between brackets correspond to the total number of amplified test methods that DCI produces in each mode.

4.3.1 Characteristics of commits with behavioral changes in the context of continuous integration

In this section, we describe the characteristics of commits introducing behavioral changes in the context of continuous integration The first five columns in Table 3 describe the characteristics of our benchmark.

The commit dates show that the benchmark is only composed of recent commits. The most recent is GSON#B1FB9CA, authored 9/22/18, and the oldest is COMMONS-IO#5D072EF, authored 9/10/15.

The number of test methods at the time of the commit shows two aspects of our benchmark:

1) we only have strongly tested projects;
2) we see that the number of tests evolve over time due to test evolution.

Every commit in the benchmark comes with test modifications (new tests or updated tests), and commit sizes are quite diverse.

The three smallest commits are COMMONS-IO#703228A, GSON#44CAD04 and JSOUP#E5210D1 with 6 modifications, and the largest is GSON#45511FD with 334 modifications.

Finally, on average, commits have 66.11% coverage. The distribution of diff coverage is reported graphically by Fig. 2: in commons-io all selected commits have more than 75% coverage. In XWiki-Commons, only 50% of commits have more than 75% coverage. Overall, 31 / 60 commits have at least 75% of the changed lines covered. This validates the correct implementation of our selection criteria that ensures the presence of a test specifying the behavioral change.

Thanks to our selection criteria, we have a curated benchmark of 60 commits with a behavioral change, coming from notable open-source projects, and covering a diversity of commit sizes. The benchmark is publicly available and documented for future research on this topic.

4.3.2 RQ1: To what extent are DCI_{AAMPL} and DCI_{SBAMPL} able to produce amplified test methods that detect the behavioral changes?

We now focus on the last 4 columns of Table 3. For instance, for COMMONS-IO#F00D97A (4^{th} row), DCI_{AAMPL} generated 39 amplified tests that detect the behavioral change. For COMMONS-IO#81210EB (8^{th} row), only the SBAMPL version of DCI detects the change.

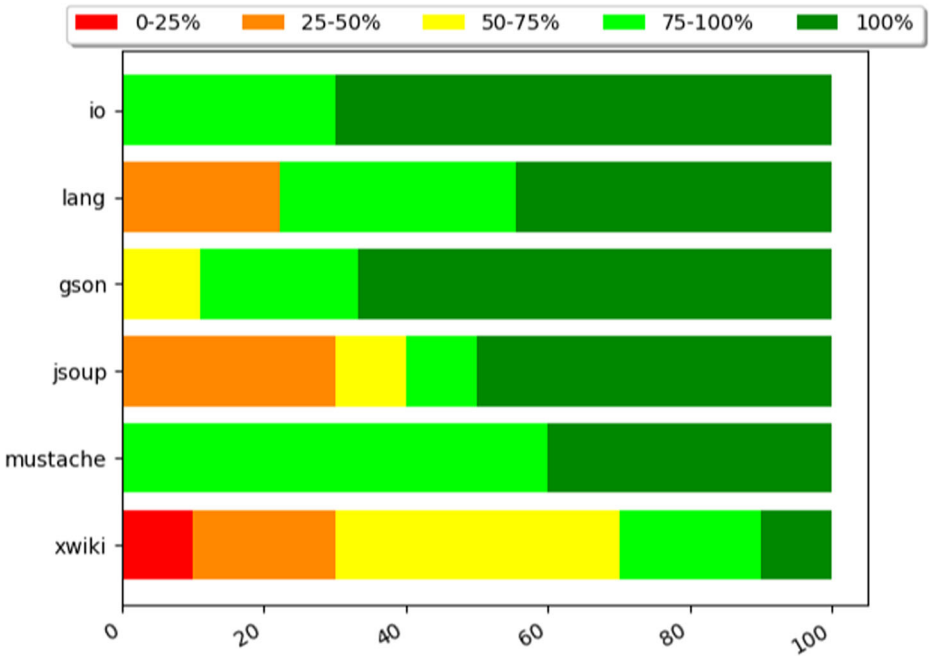


Fig. 2 Distribution of diff coverage per project of our benchmark

Overall, using only AAMPL, DCI generates amplified tests that detect 9 out of 60 behavioral changes. Meanwhile, using SBAMPL only, DCI generates amplified tests that detect 28 out of 60 behavioral changes.

Regarding the number of generated tests. DCI_{SBAMPL} generates a large number of test cases, compared to DCI_{AAMPL} only (15 versus 6708, see column "total" at the bottom of the table). Both DCI_{AAMPL} and DCI_{SBAMPL} can generate amplified tests, however, since DCI_{AAMPL} does not produce a large amount of test methods, the developers do not have to triage a large set of test cases. Also, since DCI_{AAMPL} only adds assertions, the amplified tests are easier to understand than the ones generated by DCI_{SBAMPL}.

DCI_{SBAMPL} takes more time than DCI_{AAMPL} (for successful cases 38.7 seconds versus 3.3 hours on average). The difference comes from the time consumed during the exploration of the input space in the case of DCI_{SBAMPL}, while DCI_{AAMPL} focuses on the amplification of assertions only, which represents a much smaller space of solutions.

Overall, DCI successfully generates amplified tests that detect a behavioral change in 42% of the commits in our benchmark (25 out of 60). Recall that the 60 commits that we analyze are real changes that fix bugs in complex code bases. They represent modifications, sometimes deep in the code, that represent challenges with respect to testability (Voas and Miller 1995). Consequently, the fact that DCI can generate test cases that detect behavioral changes, is considered as an achievement. The commits for which DCI fails to detect the change can be considered as a target for future research on this topic.

```
1 @@ -2619,7 +2619,7 @@ protected void appendFieldStart(final StringBuffer buffer,
      final String fieldNam
2
3 -         super.appendFieldStart(buffer, FIELD_NAME_QUOTE + fieldName
4 +         super.appendFieldStart(buffer, FIELD_NAME_QUOTE +
5 +             StringEscapeUtils.escapeJson(fieldName) + FIELD_NAME_QUOTE);
6     }
```

Listing 2 Diff of commit 3FADFDD from commons-lang

Now, we manually analyze a successful case where DCI detects the behavioral change. We select commit 3FADFDD[7] from commons-lang, which is succinct enough to be discussed in the paper. The diff is shown in Listing 2.

The developer added a method call to a method that escapes specials characters in a string. The changes come with a new test method that specifies the new behavior.

DCI starts the amplification from the `testNestingPerson` test method defined in `JsonToStringStyleTest`, showed in Listing 3.

This test is selected for amplification because it triggers the execution of the changed line.

We show in Listing 4 the resulting amplified test method. In this generated test, DCI_{SBAMPL} applies 2 input transformations: 1 duplication of method call and 1 character replacement in an existing String literal. The latter transformation is the key transformation: DCI replaced an 'a' inside "name" by '/' resulting in "n/me" where "/" is a special character that must be escaped (Line 8). Then, DCI generated 11 assertions, based on the modified inputs. The amplified test the behavioral change: in the pre-commit version, the expected value is:

`{\"n/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}`

while in the post-commit version it is

`{\"n\/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}` (Line 9).

> Answer to **RQ1**: Overall, DCI is capable of detecting the behavioral changes for 25/60 commits. DCI_{SBAMPL} finds behavioral changes in 25/60 commits, while DCI_{AAMPL} finds some in 9/60 commits. Since DCI_{SBAMPL} also uses AAMPL to generate assertions, all DCI_{AAMPL}'s commits are contained in DCI_{SBAMPL}'s. The search-based algorithm of input exploration finds many more behavioral changes, at the cost of execution time.

4.3.3 RQ2: What is the impact of the number of iteration performed by DCI_{SBAMPL}?

The results are reported in Table 4.

This table can be read as follow: the first column is the name of the project; the second column is the commit identifier; then, the third, fourth, fifth, sixth, seventh and eighth provide the amplification results and execution time for each number of iteration 1, 2, and 3. A ✓ indicates the number of amplified tests that detect a behavioral change and a - denotes that DCI did not succeed in generating a test that detects a change. The last row reports the total over the 6 projects. For the third, fifth and the seventh columns of the last row, the first number is the number of successes, *i.e.* the number of times that DCI produced at least

[7] https://github.com/apache/commons-lang/commit/3fadfdd

```
@Test
public void testPerson() {
    final Person p = new Person();
    p.name = "Jane Doe";
    p.age = 25;
    p.smoker = true;

    assertEquals(
            "{\"name\":\"Jane Doe\",\"age\":25,\"smoker\":true}",
            new ToStringBuilder(p).append("name", p.name)
                    .append("age", p.age).append("smoker", p.smoker)
                    .toString());
}
```

Listing 3 Selected test method as a seed to be amplified for commit 3FADFDD from commons-lang

one amplified test method that detect the behavioral change, for respectively$iteration = 1$, $iteration = 2$ and $iteration = 3$. The numbers in parentheses are the total number of amplified test methods that DCI produces with each number of iteration.

Overall, DCI$_{SBAMPL}$ generates amplified tests that detect 23, 24, and 25 out of 60 behavioral changes for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The more iteration DCI$_{SBAMPL}$ does, the more it explores, the more it generates amplified tests that detect the behavioral changes but the more it takes time also.

When DCI$_{SBAMPL}$ is used with $iteration = 3$, it generates amplified test methods that detect 2 more behavioral changes than when it is used with $iteration = 1$ and 1 then when it is used with $iteration = 2$.

On average, DCI$_{SBAMPL}$ generates 18, 53, and 116 amplified tests for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. This number increases by 544% from $iteration = 1$ to $iteration = 3$. This increase is explained by the fact that DCI$_{SBAMPL}$ explores more with more iteration and thus is able to generate more amplified test methods that detect the behavioral changes.

In average DCI$_{SBAMPL}$ takes 23, 64, and 105 minutes to perform the amplification for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. This number increases by 356% from $iteration = 1$ to $iteration = 3$.

```
@Test(timeout = 10000)
public void testPerson_literalMutationString85602() throws Exception {
    final ToStringStyleTest.Person p = new ToStringStyleTest.Person();
    p.name = "Jane Doe";
    Assert.assertEquals("Jane Doe", p.name);
    p.age = 25;
    p.smoker = true;
    String o_testPerson_literalMutationString85602__6 = new
        ToStringBuilder(p).append("n/me", p.name).append("age", p.age)
        .append("smoker", p.smoker).toString();
    Assert.assertEquals(
        "{\"n/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}",
        o_testPerson_literalMutationString85602__6
    );
    Assert.assertEquals("Jane Doe", p.name);
}
```

Listing 4 Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang

Table 4 Evaluation of the impact of the number of iteration done by DCI_{SBAMPL} on 60 commits from 6 open-source projects

	id	$it = 1$	Time	$it = 2$	Time	$it = 3$	Time
commons-io	c6b8a38	0	25.0s	0	62.0s	0	98.0s
	2736b6f	✓(1)	26.1m	✓(2)	44.2m	✓(12)	76.3m
	a4705cc	0	4.1m	0	21.1m	0	38.1m
	f00d97a	✓(7)	13.0s	✓(28)	19.0s	✓(39)	27.0s
	3378280	✓(6)	15.0s	✓(10)	20.0s	✓(11)	24.0s
	703228a	0	30.3m	0	55.1m	0	71.0m
	a7bd568	0	28.6m	0	52.0m	0	65.2m
	81210eb	✓(2)	14.0s	✓(4)	18.0s	✓(8)	23.0s
	57f493a	0	20.0s	0	32.0s	0	54.0s
	5d072ef	✓(461)	32.2m	✓(1014)	65.5m	✓(1538)	2.2h
	total	477	2.0h	1058	4.0h	1608	6.5h
	average	47.70	12.3m	105.80	24.0m	160.80	38.8m
commons-lang	f56931c	0	0.0s	0	3.7m	0	8.5m
	87937b2	0	3.5m	0	10.5m	0	18.1m
	09ef69c	0	97.0s	0	21.0m	0	98.8m
	3fadfdd	✓(1)	2.0m	✓(1)	9.3m	✓(4)	17.2m
	e7d16c2	✓(3)	111.0s	✓(2)	8.4m	✓(2)	15.1m
	50ce8c4	✓(61)	38.0s	✓(97)	78.0s	✓(135)	2.0m
	2e9f3a8	0	11.4m	0	35.0m	0	66.5m
	c8e61af	0	16.0s	0	16.0s	0	16.0s
	d8ec011	0	36.0s	0	68.0s	0	2.3m
	7d061e3	0	79.0s	0	5.8m	0	11.4m
	total	65	23.3m	100	96.4m	141	4.0h
	average	6.50	2.3m	10.00	9.6m	14.10	24.0m
gson	b1fb9ca	0	14.6m	0	51.0m	0	92.5m
	7a9fd59	✓(7)	33.0s	✓(48)	73.0s	✓(108)	2.1m
	03a72e7	0	30.2m	0	102.3m	0	3.2h
	74e3711	0	6.0s	0	11.0s	0	16.0s
	ada597e	0	61.0s	0	4.9m	0	8.7m
	a300148	0	45.2m	✓(4)	2.6h	✓(6)	4.9h
	9a24219	0	10.8m	0	28.4m	0	48.9m
	9e6f2ba	0	79.0s	0	4.5m	✓(2)	8.5m
	44cad04	✓(4)	21.0s	✓(21)	30.0s	✓(37)	40.0s
	b2c00a3	0	31.5m	0	111.8m	0	3.6h
	total	11	2.3h	73	7.7h	153	14.4h
	average	1.10	13.6m	7.30	46.0m	15.30	86.5m

Table 4 (continued)

	id	$it = 1$	Time	$it = 2$	Time	$it = 3$	Time
jsoup	426ffe7	✓(126)	5.4m	✓(172)	19.2m	✓(198)	33.6m
	a810d2e	0	90.0s	0	13.9m	0	26.6m
	6be19a6	0	8.1m	0	39.7m	0	67.7m
	e38dfd4	0	117.0s	0	6.3m	0	12.5m
	e9feec9	0	20.0s	0	50.0s	0	95.0s
	0f7e0cc	✓(1)	2.4h	✓(7)	6.8h	✓(36)	11.8h
	2c4e79b	0	7.1m	0	34.1m	0	4.7h
	e5210d1	0	45.0s	0	2.3m	0	4.9m
	df272b7	0	43.0s	0	2.2m	0	4.6m
	3676b13	✓(6)	21.4m	✓(35)	2.9h	✓(52)	6.8h
	total	133	3.2h	214	11.6h	286	25.8h
	average	13.30	19.4m	21.40	69.8m	28.60	2.6h
mustache.java	a1197f7	✓(28)	5.9h	✓(124)	8.4h	✓(204)	10.1h
	8877027	0	30.5m	0	58.4m	0	100.2m
	d8936b4	0	3.2m	0	4.8m	0	84.2m
	88718bc	✓(13)	78.0s	✓(85)	2.5m	✓(149)	3.7m
	339161f	✓(143)	115.9m	✓(699)	4.1h	✓(1312)	5.8h
	774ae7a	✓(18)	2.7m	✓(65)	4.7m	✓(124)	6.8m
	94847cc	✓(122)	5.3h	✓(580)	10.4h	✓(2509)	21.4h
	eca08ca	0	8.1m	0	24.3m	0	41.8m
	6d7225c	0	7.9m	0	26.8m	0	40.1m
	8ac71b7	✓(2)	2.7m	✓(48)	3.8m	✓(124)	5.6m
	total	326	14.0h	1601	25.0h	4422	42.0h
	average	32.60	84.3m	160.10	2.5h	442.20	4.2h
xwiki-commons	ffc3997	0	19.0s	0	18.0s	0	18.0s
	ced2635	0	8.0m	0	31.8m	0	2.5h
	10841b1	0	56.2m	0	2.9h	0	3.4h
	848c984	0	18.0s	0	17.0s	0	18.0s
	adfefec	✓(22)	3.5m	✓(57)	9.9m	✓(3)	14.9m
	d3101ae	✓(9)	11.6m	✓(12)	28.2m	✓(31)	41.4m
	a0e8b77	✓(10)	12.0m	✓(17)	28.2m	✓(60)	42.1m
	78ff099	✓(4)	2.6m	✓(4)	4.6m	✓(4)	6.6m
	1b79714	0	4.0m	0	10.7m	0	17.9m
	6dc9059	0	4.0m	0	10.8m	0	20.5m
	total	45	102.8m	90	4.9h	98	8.2h
	average	4.50	10.3m	9.00	29.7m	9.80	49.5m
	total	23(1057)	23.7h	24(3136)	54.9h	25(6708)	100.9h

Table 5 Number of successes, *i.e.* DCI produced at least one amplified test method that detects the behavioral changes, for 10 different seeds

Seed	ref	1	2	3	4	5	6	7	8	9
#Success	23	18	17	17	17	19	21	18	21	18

Impact of the randomness The number of amplified test methods obtained by the different seeds are reported in Table 5.

This table can be read as follow: the first column is the id of the commit. the second column is the result obtained with the default seed, used during the evaluation for $\mathbf{RQ}_1$. the ten following columns are the results obtained for the 10 different seeds.

The computed confidence interval is [20.34, 17.66] It means that, from our samples, with probability 0.95, the real value of the number of successes lies in this interval.

Answer to **RQ2**: DCI_{SBAMPL} detects 23, 24, and 25 behavioral changes out of 60 commits for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The number of iterations performed by DCI_{SBAMPL} impacts the number of behavioral changes detected, the number of amplified test methods obtained and the execution time.

4.3.4 RQ3: What is the effectiveness of our test selection method?

To answer **RQ3**, there is no quantitative approach to take, because there is no ground truth data or metrics to optimize. Per our protocol described in Section 4.2, we answer this question based on manual analysis: we randomly selected 1 commit per project, and we analyzed the relevance of the selected tests for amplification.

In order to give an intuition of what we consider as a relevant test selection for amplification, let us look at an example. If `TestX` is selected for amplification, following a change to method `X`, we consider this as relevant. The key is that DCI will generate an amplified test `TestX'` that is a variant of `TestX`, and, consequently, the developer will directly get the intention of the new test `TestX'` and what behavioral change it detects.

COMMONS-IO#C6B8A38[8]: our test selection returns 3 test methods: `testContentEquals`, `testCopyURLToFileWithTimeout` and `testCopyURLToFile` from the same test class: `FileUtilsTestCase`. The considered commit modifies the method `copyToFile` from `FileUtils`. Two test methods out of 3 (`testCopyURLToFileWithTimeout` and `testCopyURLToFile`) have an intention related to the changed file. The selection is thus considered relevant.

COMMONS-LANG#F56931C[9]: our test selection returns 39 test methods from 5 test classes: `FastDateFormat_ParserTest`, `FastDateParserTest`, `DateUtilsTest`, `FastDateParser_TimeZoneStrategyTest` and `FastDateParser_MoreOrLessTest`. This commit modifies the behavior of two methods: `simpleQuote` and `setCalendar` of class `FastDateParser`. Our manual analysis reveals two intentions: 1) test behaviors related to parsing, 1) test behaviors related

[8] https://github.com/apache/commons-io/commit/c6b8a38
[9] https://github.com/apache/commons-lang/commit/f56931c

to dates. While this is meaningful, a set of 39 methods is not a focused selection. It is considered as an half-success.

GSON#9E6F2BA[10]: our test selection returns 9 test methods from 5 different test classes. Three out of those five classes `JsonElementReaderTest`, `JsonReaderPathTest` and `JsonParserTest` relate to the class modified in the commit(`JsonTreeReader`). The selection is thus considered relevant but unfocused.

JSOUP#E9FEEC9[11], our test selection returns the 4 test methods defined in the `XmlTreeBuilderTest` class : `caseSensitiveDeclaration`, `handlesXmlDeclarationAsDeclaration`, `testDetectCharsetEncodingDeclaration` and `testParseDeclarationAttributes`. The commit modifies the behavior of the class `XmlTreeBuilder`. Here, the test selection is relevant. Actually, the ground-truth, manually written test added in the commit is also in the `XmlTreeBuilderTest` class. If DCI proposes a new test there to capture the behavioral change, the developer will understand its relevance and its relation to the change.

MUSTACHE.JAVA#88718BC[12], our test selection returns the `testInvalidDelimiters` test method defined in the `com.github.mustachejava.InterpreterTest` test class. The commit improves an error message when an invalid delimiter is used. Here, the test selection is relevant since it selected `testInvalidDelimiters` which is the dedicated test to the usage of the test invalid delimiters. This ground-truth test method is also in the test class `com.github.mustachejava.InterpreterTest`.

XWIKI-COMMONS#848C984[13] our test selection returns a single test method `createReference` from test class `XWikiDocumentTest`. The main modification of this commit is on class `XWikiDocument`. Since `XWikiDocumentTest` is the test class dedicated to `XWikiDocument`, this is considered as a success.

> Answer to **RQ3**: In 4 out of the 6 manually analyzed cases, the tests selected to be amplified are semantically related to the modified application code. In the 2 remaining cases, DCI selects tests whose intention is semantically relevant to the change, but also tests that are not. DCI's test selection provides developers with important and targeted context to better understand the behavioral change at hand.

4.3.5 RQ4: How do human and generated tests that detect behavioral changes differ?

When DCI generates an amplified test method that detects the behavioral change, we can compare it to the ground truth version (the test added in the commit) to see whether it captures the same behavioral change. For each project, we select 1 successful application of DCI, and we compare the DCI test against the human test.[14] If they capture the same behavioral change, it means they have the same intention and we consider the amplification a success.

[10] https://github.com/google/gson/commit/9e6f2ba
[11] https://github.com/jhy/jsoup/commit/e9feec9
[12] https://github.com/spullara/mustache.java/commit/88718bc
[13] https://github.com/xwiki/xwiki-commons/commit/848c984
[14] For a side-by-side comparison, see https://danglotb.github.io/resources/dci/index.html

```
1  @Test(timeout = 10000)
2  public void readMulti_literalMutationNumber3() {
3      BoundedReader mr = new BoundedReader(sr, 0);
4      char[] cbuf = new char[4];
5      for (int i = 0; i < (cbuf.length); i++) {
6          cbuf[i] = 'X';
7      }
8      final int read = mr.read(cbuf, 0, 4);
9      Assert.assertEquals(0, ((int) (read)));
10 }
```

Listing 5 Test generated by DCI_{SBAMPL} that detects the behavioral change introduced by commit 81210EB in commons-io

COMMONS-IO#81210EB[15]: This commit modifies the behavior of the `read()` method in `BoundedReader`. Listing 5 shows the test generated by DCI_{SBAMPL}. This test is amplified from the existing `readMulti` test, which indicates that the intention is to test the read functionality. The first line of the test is the construction of a `BoundedReader` object (Line 3) which is also the class modified by the commit. DCI_{SBAMPL} modified the second parameter of the constructor call (transformed 3 into a 0) and generated two assertions (only 1 is shown). The first assertion, associated to the new test input, captures the behavioral difference. Overall, this can be considered as a successful amplication.

Displayed Fx fig

Now, let us look at the human test contained in the commit, shown in Listing 6. It captures the behavioral change with the timeout (the test timeouts on the pre-commit version and goes fast enough on the post-commit version). Furthermore, it only indirectly calls the changed method through a call to `readLine`.

In this case, the DCI test can be considered better than the developer test because 1) it relies on assertions and not on timeouts, and 2) it directly calls the changed method (`read`) instead of indirectly.

COMMONS-LANG#E7D16C2[16]: this commit escapes special characters before adding them to a `StringBuffer`. Listing 7 shows the amplified test method obtained by DCI_{SBAMPL}. The assertion at the bottom of the excerpt is the one that detects the behavioral change. This assertion compares the content of the `StringBuilder` against an expected string. In the pre-commit version, no special character is escaped, *e.g.* '\—'. In the post-commit version, the DCI test fails since the code now escapes the special character \.

Let's have a look ar the human test method shown in Listing 8. Here, the developer specified the new escaping mechanism with 5 different inputs.

The main difference between the human test and the amplified test is that the human test is more readable and uses 5 different inputs. However, the amplified test generated by DCI is valid since it detects the behavioral change correctly.

GSON#44CAD04[17]: This commit allows Gson to deserialize a number represented as a string. Listing 9 shows the relevant part of the test generated by DCI_{SBAMPL}, based on `testNumberDeserialization` of `PrimitiveTest` as a seed. First, we see that the test selected as a seed is indeed related to the change in the deserialization feature. The DCI test detects the behavioral change at lines 3 and 4. On the pre-commit version,

[15] https://github.com/apache/commons-io/commit/81210eb

[16] https://github.com/apache/commons-lang/commit/e7d16c2

[17] https://github.com/google/gson/commit/44cad04

```
1 @Test(timeout = 5000)
2 public void testReadBytesEOF() {
3     BoundedReader mr = new BoundedReader( sr, 3 );
4     BufferedReader br = new BufferedReader( mr );
5     br.readLine();
6     br.readLine();
7 }
```

Listing 6 Developer test for commit 81210EB of commons-io

line 4 throws a `JsonSyntaxException`. On the post-commit version, line 5 throws a `NumberFormatException`. In other words, the behavioral change is detected by a different exception (different type and not thrown at the same line).[18].

We compare it against the developer-written ground-truth method, shown in Listing 10. This short test verifies that the program handles a number-as-string correctly. For this example, the DCI test does indeed detect the behavioral change, but in an indirect way. On the contrary, the developer test is shorter and directly targets the changed behavior, which is better.

JSOUP#3676B13[19]: This change is a pull request (*i.e.* a set of commits) and introduces 5 new behavioral changes. There are two improvements: skip the first new lines in pre tags and support deflate encoding, and three bug fixes: throw exception when parsing some urls, add spacing when output text, and no collapsing of attribute with empty values. Listing 11 shows an amplified test obtained using DCI_{SBAMPL}. This amplified test has 15 assertions and a duplication of method call. Thanks to this duplication and assertion generated on the `toString()` method, this test is able to capture the behavioral change introduced by the commit.

As before, we compare it to the developer's test. The developer uses the `Element` and `outerHtml()` methods rather than `Attribute` and `toString()`. However, the method `outerHtml()` in `Element` will call the `toString()` method of `Attribute`. For this behavioral change, it concerns the `Attribute` and not the `Element`. So, the amplified test is arguably better, since it is closer to the change than the developer's test. But, DCI_{SBAMPL} generates amplified tests that detect 2 of 5 behavioral changes: adding spacing when output text and no collapsing of attribute with empty values only, so regarding the quantity of changes, the human tests are more complete (Listing 12).

MUSTACHE.JAVA#774AE7A[20]: This commit fixes an issue with the usage of a dot in a relative path on Window in the method `getReader` of class `ClasspathResolver`. The test method `getReaderNullRootDoesNotFindFileWithAbsolutePath` has been used as seed by DCI. It modifies the existing string literal with another string used somewhere else in the test class and generates 3 new assertions (Listing 13). The behavioral change is detected thanks to the modified strings: it produces the right test case containing a space.

The developer proposed two tests that verify that the object reader is not null when getting it with dots in the path. There are shown in Listing 14. These tests invoke the method `getReader` which is the modified method in the commit.

[18]Interestingly, the number is parsed lazily, only when needed. Consequently, the exception is thrown when invoking the `longValue()` method and not when invoking `parse()`

[19]https://github.com/jhy/jsoup/commit/3676b13

[20]https://github.com/spullara/mustache.java/commit/774ae7a

```
@Test(timeout = 10000)
public void testAppendSuper_literalMutationString64() {
    String o_testAppendSuper_literalMutationString64__15 =
        new ToStringBuilder(base)
            .appendSuper(((((("Integer@8888[" + (System.lineSeparator()
                )) + "  null")
                + (System.lineSeparator())) + "]"))
            .append("a", "b0/|]")
            .toString();
    Assert.assertEquals("{\"a\":\"b0/|]\"}",
        o_testAppendSuper_literalMutationString64__15);
}
```

Listing 7 Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of E7D16C2 in commons-lang

```
@Test
public void testLANG1395() {
    assertEquals("{\"name\":\"value\"}",
        new ToStringBuilder(base).append("name","value").toString());
    assertEquals("{\"name\":\"\"}",
        new ToStringBuilder(base).append("name","").toString());
    assertEquals("{\"name\":\"\\\"\"}",
        new ToStringBuilder(base).append("name",'"').toString());
    assertEquals("{\"name\":\"\\\\\"}",
        new ToStringBuilder(base).append("name",'\\').toString());
    assertEquals("{\"name\":\"Let's \\\"quote\\\" this\"}",
        new ToStringBuilder(base).append("name","Let's \"quote\" this"
            ).toString());
}
```

Listing 8 Developer test for E7D16C2 of commons-lang

```
public void
    testNumberDeserialization_literalMutationString8_failAssert0()
    throws Exception {
    try {
        String json = "dhs";
        actual = gson.fromJson(json, Number.class);
        actual.longValue();
        junit.framework.TestCase.fail(
        "testNumberDeserialization_literalMutationString8 should have
            thrown JsonSyntaxException");
    } catch (JsonSyntaxException expected) {
        TestCase.assertEquals("Expecting number, got: STRING",
            expected.getMessage());
    }
}
```

Listing 9 Test generated by DCI that detects the behavioral change of commit 44CAD04 in Gson

```
public void testNumberAsStringDeserialization() {
    Number value = gson.fromJson("\"18\"", Number.class);
    assertEquals(18, value.intValue());
}
```

Listing 10 Provided test by the developer for 44CAD04 of Gson

```
@Test(timeout = 10000)
public void parsesBooleanAttributes_add4942() {
    String html = "<a normal=\"123\" boolean empty=\"\"></a>";
    Element el = Jsoup.parse(html).select("a").first();
    List<Attribute> attributes = el.attributes().asList();
    Attribute o_parsesBooleanAttributes_add4942__15 =
    attributes.get(1);
    Assert.assertEquals("boolean=\"\"",
        ((BooleanAttribute) (o_parsesBooleanAttributes_add4942__15)).
            toString());
}
```

Listing 11 Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of 3676B13 of Jsoup

```
@Test
public void booleanAttributeOutput() {
    Document doc = Jsoup.parse("<img src=foo noshade='' nohref async=
        async autofocus=false>");
    Element img = doc.selectFirst("img");

    assertEquals("<img src=\"foo\" noshade nohref async autofocus=\"
        false\">", img.outerHtml());
}
```

Listing 12 Provided test by the developer for 3676B13 of Jsoup

```
@Test(timeout = 10000)
public void getReaderNullRootDoesNotFindFileWithAbsolutePath_litStr4()
     {
    ClasspathResolver underTest = new ClasspathResolver();
    Reader reader = underTest.getReader(" does not exist");
        Assert.assertNull(reader);
    Matcher<Object>
        o_getReaderNullRootDoesNotFindFileWithAbsolutePath_litStr4__5
            =
            Is.is(CoreMatchers.nullValue());
    Assert.assertEquals("is null",
        ((Is) (
            o_getReaderNullRootDoesNotFindFileWithAbsolutePath_litStr4__5
            ))
            .toString());
    Assert.assertNull(reader);
}
```

Listing 13 Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of 774AE7A of Mustache.java

```
@Test
public void getReaderWithRootAndResourceHasDoubleDotRelativePath()
    throws Exception {
    ClasspathResolver underTest = new ClasspathResolver("templates");
    Reader reader = underTest.getReader("absolute/../
        absolute_partials_template.html");
    assertThat(reader, is(notNullValue()));
}

@Test
public void getReaderWithRootAndResourceHasDotRelativePath() throws
    Exception {
    ClasspathResolver underTest = new ClasspathResolver("templates");
    Reader reader = underTest.getReader("absolute/./
        nested_partials_sub.html");
    assertThat(reader, is(notNullValue()));
}
```

Listing 14 Developer test for 774AE7A of Mustache.java

The difference is that the DCI_{SBAMPL}'s amplified test method provides a non longer valid input for the method `getReader`. However, providing such inputs produce errors afterward which signal the behavioral change. In this case, the amplified test is complementary to the human test since it verifies that the wrong inputs are no longer supported and that the system immediately throws an error.

XWIKI-COMMONS#D3101AE[21]: This commit fixes a bug in the `merge` method of class `DefaultDiffManager`. Listing 15 shows the amplified test method obtained by DCI_{AAMPL}. DCI used `testMergeCharList` as a seed for the amplification process, and generates 549 new assertions. Among them, 1 assertion captures the behavioral change between the two versions of the program: "assertEquals(0, result.getLog().getLogs(LogLevel.ERROR).size());". The behavioral change that is detected is the presence of a new logging statement in the diff. After verification, there is indeed such a behavioral change in the diff, with the addition of a call to "logConflict" in the newly handled case.

The developer's test is shown in Listing 16. This test method directly calls method `merge`, which is the method that has been changed. What is striking in this test is the level of clarity: the variable names, the explanatory comments and even the vertical space formatting are impossible to achieve with DCI_{AAMPL} and makes the human test clearly of better quality but also longer to write. Yet, DCI_{AAMPL}'s amplified tests capture a behavioral change that was not specified in the human test. In this case, amplified tests can be complementary.

Answer to **RQ4**: In 3 out of 6 cases, the DCI test is complementary to the human test. In 1 case, the DCI test can be considered better than the human test. In 2 cases, the human test is better than the DCI test. Even though human tests can be better, DCI can be complementary and catch missed cases, and provide added-value when developers do not have the time to add a test.

[21] https://github.com/xwiki/xwiki-commons/commit/d3101ae

```
@Test(timeout = 10000)
public void testMergeCharList() throws Exception {
    MergeResult<Character> result;
    result = this.mocker.getComponentUnderTest().merge(
        AmplDefaultDiffManagerTest.toCharacters("a"),
        AmplDefaultDiffManagerTest.toCharacters(""),
        AmplDefaultDiffManagerTest.toCharacters("b"), null);
    int o_testMergeCharList__9 = result.getLog().getLogs(LogLevel.
        ERROR).size();
    Assert.assertEquals(1, ((int) (o_testMergeCharList__9)));
    List<Character> o_testMergeCharList__12 =
        AmplDefaultDiffManagerTest.toCharacters("b");
    Assert.assertTrue(o_testMergeCharList__12.contains('b'));
    result.getMerged();
    result = this.mocker.getComponentUnderTest().merge(
        AmplDefaultDiffManagerTest.toCharacters("bc"),
        AmplDefaultDiffManagerTest.toCharacters("abc"),
        AmplDefaultDiffManagerTest.toCharacters("bc"), null);
    int o_testMergeCharList__21 = result.getLog().getLogs(LogLevel.
        ERROR).size();
    Assert.assertEquals(0, ((int) (o_testMergeCharList__21)));
}
```

Listing 15 Test generated by DCI$_{AAMPL}$ that detects the behavioral change of D3101AE of XWiki

```
@Test
public void testMergeWhenUserHasChangedAllContent() throws Exception
{
    MergeResult<String> result;

    // Test 1: All content has changed between previous and current
    result = mocker.getComponentUnderTest().merge(Arrays.asList("Line
        1", "Line 2", "Line 3"),
    Arrays.asList("Line 1", "Line 2 modified", "Line 3", "Line 4 Added
        "),
    Arrays.asList("New content", "That is completely different"), null
        );

    Assert.assertEquals(Arrays.asList("New content", "That is
        completely different"), result.getMerged());

    // Test 2: All content has been deleted
    // between previous and current
    result = mocker.getComponentUnderTest().merge(Arrays.asList("Line
        1", "Line 2", "Line 3"),
    Arrays.asList("Line 1", "Line 2 modified", "Line 3", "Line 4 Added
        "),
    Collections.emptyList(), null);

    Assert.assertEquals(Collections.emptyList(), result.getMerged());
}
```

Listing 16 Developer test for D3101AE of XWiki

5 Discussion About the Scope of DCI

In this section, we overview the current scope of DCI and the key challenges that limit DCI.

Focused applicability From our benchmark, we see that DCI is applicable to a limited proportion of commits: on average 15.29% of the commits analyzed. This low proportion is the first limit of DCI usage. However, once DCI is setup, it is fully automated, there is no manual overhead. Even if DCI is not used at each commit, it costs little more.

Adoption Our evaluation showed that DCI is able to obtain amplified test methods that detect behavioral changes. But, it does not provide any evidence on the fact that developers would exploit such test methods. However, from our past evaluation (Danglot et al. 2019), we know that software developers value amplified test methods. This provides strong evidence of the potential adoption of DCI.

Performance From our experiments, we see that the time to complete the amplification is the main limitation of DCI. For example DCI took almost 5 hours on JSOUP#2C4E79B, with no result. For the sake of our experimentation, we choose to use a pre-defined number of iterations to bound the exploration. In practice, we recommend to set a time budget (*e.g.* at most one hour per pull-request).

Importance of test seeds By construction, DCI's effectiveness is correlated to the test methods used as seeds. For example, see the row of `commons-lang#c8e61af` in Table 4, where one can observe that whatever the number of iterations, DCI takes the same time to complete the amplification. The reason is that the seed tests are only composed of assertions statements. Such tests are bad seeds for DCI, and they prevent any good input amplification. Also, DCI requires to have at least one test method that executes the code changes. If the project is poorly tested and does not have any test method that execute the code changes, DCI cannot be applied.

False positives The risk of false positives is a potential limitation of our approach. A false positive would be an amplified test method that passes or fails on both versions, which means that the amplified test method does not detect the behavioral difference between both versions. We manually analyzed 6 commits and none of them are false positives. This increases our confidence that DCI produces a limited number of such confusing test methods.

6 Threats to Validity

An internal threat is the potential bugs in the implementation of DCI. However, we heavily tested our prototype with JUnit test cases to mitigate this threat.

In our benchmark, there are 60 commits. Our result may be not be generalizable to all programs. But we carefully selected real and diverse applications from GitHub, all having a strong test suite. We believe that the benchmark reflects real programs, and we have good confidence in the results.

Last but not least, there is a potential flakiness to generated test methods. However we take care that our approach does not produce flaky test methods, and we make sure to observe a stable and different state of the program between different executions. To do this,

we execute each amplified test 3 times in order to check weather or not there are stable. If the outcome of at least one execution is different than the others, we discard the amplified test.

Our experiments are stochastic, and randomness is a threat accordingly. To mitigate this threat, we have computed a confidence interval that estimates the number of successes that DCI would obtain.

7 Related Work

7.1 Commit-based Test Generation

Person et al. (2008) present differential symbolic execution (DSE). DSE combines symbolic execution and a new approximation technique to highlight behavioral changes. They use symbolic execution summary to find equivalences and difference and generate a set of inputs that trigger different behavior. This is done in three steps: 1) they execute both versions of the modified method; 2) they find equivalences and differences, thanks to the analysis of symbolic execution summary; 3) they generate a set of inputs that trigger the different behaviors in both versions. The main difference with our work is that they have the strong assumption to have a program whose semantics is fully handled by the symbolic execution engine. In the context of Java, to our knowledge, no symbolic execution engine works on arbitrary Java program. Symbolic execution engines do not scale to the size and complexity of the programs we targeted. On the contrary, our approach, being more lightweight, is meant to work on all Java programs.

Marinescu and Cadar (2013) present Katch, a system that aims at covering the code included in a patch. This approach first determine[17.66 ; 20.34s the differences of a program and its previous version. It targets modified and not executed by the existing test suite lines. Then, it selects the closest input to each target from existing tests using a static minimum distance over the control flow graph. The proposal is evaluated on Unix tools. They examine patches from a period of 3 years. In average, they automatically increase coverage from 35% to 52% with respect to the manually written test suite. Contrary to our work, they only aim at increasing the coverage, not at detecting behavioral changes.

A posterior work of the same group (Palikareva et al. 2016; Kuchta et al. 2018) focuses on finding test inputs that execute different behaviors in two program versions. They devise a technique, named ShaddowKlee, built on top of Klee (Cadar et al. 2008). They require the code to be annotated at changed places. Then they select from the test suite those test cases that cover the changed code. The unified program is used in a two stage dynamic symbolic execution guided by the selected test cases. They first look for branch points where the conditions are evaluated in both program versions. Then, a constraint solver generates new test inputs for divergent scenarios. The program versions are then normally executed with the generated inputs and the result is validated to check the presence of a bug or of an intended difference. The evaluation of the proposed method is based on the CoREBench (Böhme and Roychoudhury 2014) data set that contains documented regression bugs of the GNU Coreutils program suite.

Noller et al. (2018) aim at detecting regression bugs. They apply shadow symbolic execution, originally from Palikevera (Person et al. 2008; Palikareva et al. 2016) that has been discussed in the previous paragraph, on Java programs. Their approach has been implemented as an extension of Java Path Finder Symbolic (jpf-symbc) (Anand et al. 2007),

named jpf-shadow. Shadow symbolic execution generate test inputs that trigger the new program behavior. They use a merged version of both version of the same program, i.e. the previous version, so called old, and the changed version, called new This is done by instrumenting the code with method calls "change()". The method change() takes two inputs: the old statement and the new one. [17.66 ; 20.34 Then, a first step collects divergence points, i.e. conditional where the old version and the new version do not take the same branch. On small examples, they show that jpf-shadow generates less unit test cases yet cover the same number of path. Jpf-shadow only aims at covering the changes and not at detecting the behavioral change with an assertion.

Menarini et al. (2017) proposes a tool, GETTY, based on invariants mined by Daikon. GETTY provides to code reviewers a summary of the behavioral changes, based on the difference of invariants for various combinations of programs and test suites. They evaluate GETTY on 6 open source project, and showed that their behavioral change summaries can detect bugs earlier than with normal code review. While they provide a summary, DCI provides a concrete test method with assertions that detect the behavioral changes.

Lahiri et al. (2013) propose differential assertion checking (DAC): checking two versions of a program with respect to a set of assertions. DAC is based on filtering false alarms of verification analysis. They evaluate DAC on a set of small example. The main difference is that DAC requires to manually write specifications, while DCI is completely automated with normal code as input.

Yang et al. (2014) introduce IProperty, a way to annotate correctness properties of programs. They evaluate their approach on the triangle problem. The key novelty of our work is to perform an evaluation on real commits from large scale open source software.

Campos et al. (2014) extended EvoSuite to adapt test generation techniques to continuous integration. Their contribution is the design of a time budget allocation strategy: it allocates more time budget to specific classes that are involved in the changes. They evaluated their approach on 10 projects from the SF100 corpus, on 8 of the most popular open-source projects from GitHub, and on 5 industrial projects. They limit their evaluation to the 100 last consecutive commits. They observe an increase of +58% branch coverage, +69% thrown undeclared exceptions, while reducing the time consumption by up to 83% compared to the baseline. The major difference compared to our approach, they do not aim at specifically obtaining test methods that detect the behavioral changes but rather obtain better branch coverage and detect undeclared exceptions. They also do not generate any assertions. However, from the point of view practitioners, integrating a time budget strategy into DCI would increase its usability, practicability and potential adoption.

7.2 Behavioral Change Detection

Evans and Savoia (2007) devise the differential testing. This approach aims at alleviating the test repair problem and detects more changes than regression testing alone. They use an automated characterization test generator (ACTG) to generate test suite for both version of the program. They then categorizes the tests of these 2 test suites into 3 groups: 1) $T_{preserved}$ which are the tests that pass on the both versions; 2) $T_{regressed}$ which are the tests that pass on the previous version but not on the new one; 3) $T_{progressed}$ which are the tests that pass on the new version but not on the previous one; Then, they define also $T_{different}$ which is the union of both $T_{regressed}$ and $T_{progressed}$. The approach is to execute $T_{different}$ on both versions and observe progressed and regressed behaviors. They evaluate their approach on a small use case from the SIR dataset on 38 diffrent changes, for version of the program. They showed that their approach detects 21%, 34%, and 21% more behavior changes than

regression testing alone for respectively version 1, version 2 and version 3. In DCI, the amplified test methods obtained would lie into the $T_{regressed}$ group. However, we could also amplified test methods using the new version of the program and obtain a $T_{progressed}$. We would obtain a $T_{different}$ of amplified test methods and it might improve the performance of DCI. About the evaluation, we run experimentation of 60 commits which the double than their dataset, and on real projects and real commits from GitHub.

Jin et al. (2010) propose BEhavioral Regression Testing BERT. BERT aims at assisting practitioners during development to identify potential regression. It has been implemented as a plugin for the IDE Eclipse. Each time a developer make a change in their code base and Eclipse compiles, BERT is triggered. BERT works in 3 phases: 1) it analyzes what are the classes modified and runs a test generation tools, such as Randoop, to create new test input for these classes. 2) it executes the generated tests on both version of the program and collect multiples values such as the values of the fields of objects, the returned values by methods, etc. 3) it produces a report containing all the differences of behaviors based on the collected values. Then the developer used this report to decide whether or not the changes are correct. They evaluated BERT on a small and artificial project, showing that about 60% of the automatically generated test inputs were able to reveal the behavioral difference that indicates the regression fault In addition to this proof-of-concept, they evaluated in on JODA-time, which is a mature and widely used library. They evaluated on 54 pairs of versions. They reported 36 behavioral differences. However, they could establish only for one of them was a regression fault. There are two major differences with DCI: 1) DCI works at commit level and not to the class changes level. 2) DCI produces real and actionable test methods.

Taneja and Xie (2008) present DiffGen, a tool that generate regression tests for two version of the same class. Their approach works as follow: First, they detect the changes between the two version of the class. It is done using the textual representation and at method level. Second, they generate what they call a test driver, which is a class that contains a method for each modified method. These methods takes as input an instance of the old version of the class and the inputs required by the modified method. They also make all the field public to compare their values between the old version and the new one. These comparison have the form of branches. The intuition is if the test generator engine is able to cover these branches, it will reveal the behavioral differences. Third, they generate test using a test generator and the test driver. Eventually, they execute the generated tests to see whether or not there is a behavioral difference. They evaluated DiffGen on 8 artificial classes from the state of the art. They compared the mutation score of their generated test suite to an existing method from the state of the art. They showed that that DiffGen has an Improvement Factor If2 varying from 23.4% to 100% for all the subjects. They also performed an evaluation on larger subjects from the SIR dataset. They detected 5 more faults than the state of the art. DiffGen must modify the application code to be efficient while DCI does not required any modification of it. Thus, is makes generated tests by DiffGen unused by developers since they must expose all the fields of their classes.

Madeiral et al. (2019) built a benchmark of bugs for evaluating automatic program repair tools. This benchmark has been built using behavioral change detection such as we d do in this paper. However, this benchmark includes a different kind of behavioral change: bug fixes. Also, they have different criteria to select the commits than ours, and their procedure is similar in different ways. Their approach used continuous integration to build automatically and enrich their benchmark, and it would be fruitful to automate our process as well.

7.3 Test Amplification

Yoo and Harman (2012) devise Test Data Regeneration(TDR). They use hill climbing on existing test data (set of input) that meets a test objective (*e.g.* cover all branch of a function). The algorithm is based on *neighborhood* and a *fitness* functions as the classical hill climbing algorithm. The key difference with DCI is that they at fulfilling a test criterion, such as branch coverage, while we aim at obtaining test methods that detect the behavioral changes.

It can be noted that several test generation techniques start from a seed and evolve it to produce a good test suite. This is the case for techniques such as concolic test generation (Godefroid et al. 2005), search-based test generation (Fraser and Arcuri 2012), or random test generation (Groce et al. 2007). The key novelty of DCI relies in the very nature of the tests we used as seed. DCI uses complete program, which creates objects, manipulates the state of these objects, calls methods on these objects and asserts properties on their behavior. That is to say real and complex object-oriented tests as seed

7.4 Continuous Integration

Hilton et al. (2016) conduct a study on the usage, costs and benefits of CI. To do this, they use three sources: open-source code, builds from Travis, and they surveyed 442 engineers. Their studies show that the usage of CI services such as Travis is widely used and became the trend. The fact that CI is widely used shows that relevance of behavioral change detection.

Zampetti et al. (2017) investigate the usage of Automated Static Code Analysis Tools (ASCAT) in CI. There investigation is done on 20 projects on GitHub. According to their findings, coding guideline checkers are the most used static analysis tools in CI. This paper shows that dynamic analysis, such as DCI, is the next step for getting more added-value from CI.

Spieker et al. (2017) elaborate a new approach for test case prioritization in continuous integration based on reinforcement learning. Test case prioritization is different from behavioral change detection.

Waller et al. (2015) study the portability of performance tests in continuous integration. They show little variations of performance tests between runs (every night) and claim that the performance tests must be integrated in the CI, early as possible in the development of Software. Performance testing is also one kind of dynamic analysis for the CI, but different in nature from behavioral change detection.

8 Conclusion

In this paper, we have studied the problem of behavioral change detection for continuous integration. We have proposed a novel technique called DCI, which uses assertion generation and search-based transformation of test code to generate tests that automatically detect behavioral changes in commits. We analyzed 1576 commits from 6 projects. On average, our approach is applicable to 15.29% of commits per-project. We built a curated set of 60 commits coming from real-world, large open-source Java projects to evaluate our technique. We show that our approach is able to detect the behavioral differences of 25 of the 60 commits.

We plan to work on an automated continuous integration bot for behavioral change detection that will: 1) check if a behavioral change is already specified in a commit (*i.e.* a test

case that correctly detects the behavioral change is provided); 2) if not, execute behavioral change detection and test generation; 3) propose the synthesized test method to the developers to complement the commit. Such a bot can work in concert with other continuous integration bots, such as bots for automated program repair (Urli et al. 2018).

References

Anand S, Pasareanu CS, Visser W (2007) Jpf-se: A symbolic execution extension to java pathfinder 03

Böhme M, Roychoudhury A (2014) Corebench: Studying complexity of regression errors. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, pp 105–115

Cadar C, Dunbar D, Engler D (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08. USENIX Association, Berkeley, pp 209–224

Campos J, Arcuri A, Fraser G, Abreu R (2014) Continuous test generation: Enhancing continuous integration with automated test generation. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14. ACM, pp 55–66

Danglot B, Vera-Pérez OL, Baudry B, Monperrus M (2019) Automatic test improvement with dspot: a study with ten mature open-source projects. Empirical Software Engineering

Daniel B, Jagannath V, Dig D, Marinov D (2009) Reassert: Suggesting repairs for broken unit tests. In: 2009 IEEE/ACM International conference on automated software engineering, pp 433–444

Duvall PM, Matyas S, Glover A (2007) Continuous integration: improving software quality and reducing risk. Pearson Education

Evans RB, Savoia A (2007) Differential testing: a new approach to change detection. In: The 6th joint meeting on european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: Companion papers. ACM, pp 549–552

Falleri J-R, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and Accurate Source Code Differencing. In: Proceedings of the International Conference on Automated Software Engineering, pp 313–324

Fowler M, Foemmel M (2006) Continuous integration. Thought-Works https://www.thoughtworks.com/continuous-integration, pp 122:14

Fraser G, Arcuri A (2012) The seed is strong: Seeding strategies in search-based software testing. In: 2012 IEEE fifth international conference on Software testing, verification and validation (ICST). IEEE, pp 121–130

Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. In: ACM Sigplan notices. ACM, vol 40, pp 213–223

Groce A, Holzmann G, Joshi R (2007) Randomized differential testing as a prelude to formal verification. In: Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, pp 621–631

Hilton M, Tunnell T, Huang K, Marinov D, Dig D (2016) Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016 .ACM, New York, pp 426–437

Hilton M, Bell J, Marinov D (2018) A large-scale study of test coverage evolution. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018. ACM, New York, pp 53–63

Jin W, Orso A, Xie T (2010) Automated behavioral regression testing. In: 2010 Third international conference on software testing, verification and validation, pp 137–146

Kuchta T, Palikareva H, Cadar C (2018) Shadow symbolic execution for testing software patches. ACM Trans Softw Eng Methodol 27(3):10:1–10:32

Lahiri S, McMillan K, Hawblitzel C (2013) Differential assertion checking. Technical report

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)

Marinescu PD, Cadar C (2013) KATCH: high-coverage testing of software patches. ACM Press, pp 235

Menarini M, Yan Y, Griswold WG (2017) Semantics-assisted code review: an efficient tool chain and a user study. In: 2017 32Nd IEEE/ACM international conference on automated software engineering (ASE), pp 554–565

Noller Y, Nguyen HL, Tang M, Kehrer T (2018) Shadow symbolic execution with java pathfinder. SIGSOFT Softw. Eng. Notes 42(4):1–5
Palikareva H, Kuchta T, Cadar C (2016) Shadow of a doubt: testing for divergences between software versions. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp 1181–1192
Person S, Dwyer MB, Elbaum S, Păsăreanu CS (2008) Differential symbolic execution. In: sProceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16. ACM, New York, pp 226–237, NY
Saff D, Ernst MD (2004) An experimental evaluation of continuous testing during development. In: ACM SIGSOFT Software engineering notes. ACM, vol 29, pp 76–85
Spieker H, Gotlieb A, Marijan D, Mossige M (2017) Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017. ACM, New York, pp 12–22
Taneja K, Xie T (2008) Diffgen: Automated regression unit-test generation. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08. IEEE Computer Society, Washington, pp 407–410
Tonella P (2004) Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04. ACM, New York, pp 119–128
Urli S, Yu Z, Seinturier L, Monperrus M (2018) How to Design a Program Repair Bot? Insights from the Repairnator Project. In: ICSE 2018 - 40Th international conference on software engineering, track software engineering in practice (SEIP), pp 1–10
Vera-Pérez OL, Danglot B, Monperrus M, Baudry B (2018) A comprehensive study of pseudo-tested methods. Empirical Software Engineering
Voas JM, Miller KW (1995) Software testability: the new verification. IEEE Softw 12(3):17–28
Waller J, Ehmke NC, Hasselbring W (2015) Including performance benchmarks into continuous integration to enable devops. SIGSOFT Softw Eng Notes 40(2):1–4
Xie T (2006) Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas D (ed) ECOOP 2006 – Object-Oriented Programming. Springer, Berlin, pp 380–403
Yang G, Khurshid S, Person S, Rungta N (2014) Property differencing for incremental checking. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, New York, pp 1059–1070
Yoo S, Harman M (2012) Test data regeneration: Generating new test data from existing test data. Softw Test Verif Reliab 22(3):171–201
Zampetti F, Scalabrino S, Oliveto R, Canfora G, Penta MD (2017) How open source projects use static code analysis tools in continuous integration pipelines. In: 2017 IEEE/ACM 14Th international conference on mining software repositories (MSR), pp 334–344
Zhang P, Elbaum S (2012) Amplifying tests to validate exception handling code. In: Proc. of int. Conf. on software engineering (ICSE). IEEE Press, pp 595–605

Benjamin Danglot is a researcher in the following fields: test suite amplification in the DevOps context and chaos engineering. His work took part in the Horizon 2020 European project called STAMP: Software Testing AMPlification.

Walter Rudametkin is an associate professor at the University of Lille and part of the Spirals team, a joint team between the CRIStAL laboratory and Inria. He received his Ph.D. in 2013 from the University of Grenoble, focused on dynamic updates on large software software systems. His work currently focuses on applying software engineering to cross-cutting concerns, such as privacy and security.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology, Sweden. In 2011-2017, he was associate professor at the University of Lille, France and adjunct researcher at Inria. He received a Ph.D. from the University of Rennes, and a Master's degree from the Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, self-healing software and chaos engineering.

Benoit Baudry is a Professor in Software Technology at the Royal Institute of Technology (KTH) in Stockholm, Sweden. He received his PhD in 2003 from the University of Rennes and was a research scientist at INRIA from 2004 to 2017. His research is in the area of software testing, code analysis and automatic diversification. He has led the largest research group in software engineering at INRIA, as well as collaborative projects funded by the European Union, and software companies

Link to install the transformer library: https://huggingface.co/docs/transformers/installation