On the Variability of AI-based Software Systems Due to Environment Configurations

Musfiqur Rahman[a,*], SayedHassan Khatoonabadi[a], Ahmad Abdellatif[b], Haya Samaana[c], Emad Shihab[a]

[a] *Concordia University, Montreal, Canada*
[b] *University of Calgary, Calgary, Canada*
[c] *An-Najah National University, Nablus, Palestine*

Abstract

[Context] Nowadays, many software systems include Artificial Intelligence (AI) components and changes in the development environment have been known to induce variability in an AI-based system. [Objective] However, how an environment configuration impacts the variability of these systems is yet to be explored. Understanding and quantifying the degree of variability due to such configurations can help practitioners decide the best environment configuration for the most stable AI products. [Method] To achieve this goal, we performed experiments with eight different combinations of three key environment variables (operating system, Python version, and CPU architecture) on 30 open-source AI-based systems using the Travis CI platform. We evaluate variability using three metrics: the output of an AI component like an ML model (performance), the time required to build and run a system (processing time), and the cost associated with building and running a system (expense). [Results] Our results indicate that variability exists in all three metrics; however, it is observed more frequently with respect to processing time and expense than performance. For example, between Linux and MacOS, variabilities are observed in 23%, 96.67%, and 100% of the studied projects in performance, processing time, and expense, respectively.

[*] Corresponding author.

Email addresses: `musfiqur.rahman@mail.concordia.ca` (Musfiqur Rahman), `sayedhassan.khatoonabadi@concordia.ca` (SayedHassan Khatoonabadi), `ahmad.abdellatif@ucalgary.ca` (Ahmad Abdellatif), `hayasam@najah.edu` (Haya Samaana), `emad.shihab@concordia.ca` (Emad Shihab)

[Conclusion] Our findings underscore the importance of identifying the optimal combination of configuration settings to mitigate performance drops and reduce retraining time and cost before deploying an AI-based system.

Keywords: Continuous integration, SE4AI, variability, AI-based systems

1. Introduction

With the recent advancement and popularity in the field of Artificial Intelligence (AI) — more specifically Machine Learning (ML) models — in solving numerous real-life problems, more and more software is integrating such models as part of the pipeline [1]. Software systems are inherently complex. In addition, any ML model is, at its core, probabilistic in nature and hence, suffers from the challenge of uncertainty [2, 3, 4]. The complexity of a software system combined with the non-deterministic nature of an ML model can introduce variability – the phenomenon where a piece of software behaves differently when the development or the runtime environment changes although the internal software artifacts such as code, and input data are exactly the same.

In practice it is very likely that development and deployment environments are different, hence, understanding how an ML model may behave differently after deployment compared to how it behaved in the development environment is a crucial aspect of AI-based software development. For example, an arbitrary face recognition system achieving an F1-score of, say 0.9, in the development environment does not guarantee that it will on average achieve a similar F1-score once deployed in a different environment configuration. Therefore, running the system under different configuration settings needs to be an additional step before deployment of the system to determine if the performance significantly varies from configuration to configuration. As demonstrated by the previous example, the probabilistic and uncertain nature of ML models can introduce novel challenges affecting different stages of the software development life cycle. The software engineering research community has recently started investigating the challenges that come with the uncertain nature of AI-enabled software systems in various aspects of the development life cycle such as requirement elicitation [5], software testing and quality assurance [6], and deployment [7].

As discussed above, the environment settings can vary from one stage to another in the development life cycle. The choices made by the developers regarding development environment variables, such as operating systems, versions of a programming language and associated libraries, and hardware configurations, can depend on many factors including developers' experience, business needs, and existing environment configurations of legacy systems. However, these choices may potentially induce variability in the performance of AI/ML models as "practitioners' degrees of freedom" [8, 9], which is a known issue in the field of applied statistics. However, in the domain of software engineering, no existing work studies the potential sources of variability in AI-based software from an environment configuration perspective. In this paper, we aim to address this issue by answering the following three research questions:

RQ1: **(Operating System) To what extent does the operating system induce variability in AI-based systems?** We analyze whether variations in operating systems make AI-based systems behave differently. We find that variability in performance is observed in 23% of the projects between Linux and MacOS whereas 20% of the projects show such variability between Linux and Windows. Almost all projects show significant variability in processing time and expense between different operating systems.

RQ2: **(Python Version) How does the Python version contribute to the variability in AI-based systems?** We investigate the effect of Python versions on the behavior of AI-based systems. We found that Python 3.6 and Python 3.7 consistently produce identical results in all three metrics. However, between Python 3.7 and Python 3.8, variability can be observed in about 17% of the projects in performance and 80% of the projects in both processing time and expense.

RQ3: **(CPU Architecture) How does CPU architecture affect the variability in AI-based systems?** We turn our focus from software-level configuration to hardware-level configuration. We compare two CPU architectures and find that over 93% of the projects show variability in processing time and expense while only 20% of them vary in performance between AMD64 and ARM64 architectures.

Overall, our findings show that configuration settings induce variability although the degree varies from project to project. Variability in processing time and expense is more frequently observed than variability in performance.

Determining the best configuration settings for a project is an iterative process, and developers should build and run their systems on different settings to find the most optimized environment configuration for the product.

This paper makes the following contributions:

- To the best of our knowledge, this is the first empirical study on the variability of AI-based systems from the environment configuration point of view.
- We provide empirical evidence on the effect of different environment configurations on the variability of AI-based systems.
- We make our data and scripts available for reproducibility and future research [10].

The rest of the paper is structured as follows. Background and research methodology are described in Section 2. In Sections 3, 4, and 5, we present the results and findings of our analysis for each research question. Sections 6, 7 present discussion and threats to validity respectively followed by Section 8 in which we talk about existing works. Finally, in Section 9, we summarize our findings and conclude the paper by describing potential future directions of research.

2. Methodology and Background

To be able to conduct our experiments in different development environment configurations, we use Travis CI — a widely used Continuous Integration (CI) platform [11]. The reason for choosing Travis CI is that a recent study on the usage of CI tools in ML projects reports that Travis CI is the most popular among open-source software (OSS) developers for building AI-based systems [12].

2.1. Environment Configurations in Travis CI

In this study, the three configuration variables we experiment with are *Operating System*, *CPU Architecture*, and *Python Version*. We choose to experiment with these three variables because the operating system is the core of any development environment where a product is built and run, CPU is the core of the hardware on which a product is run, and the programming language is at the core of development tech stack used for building a product.

In our experiment, we use the following list of options for each configuration variable:

- *Operating System:* Linux (version Ubuntu-Xenial 16.04), MacOS (version 10.14.4), and Windows (version 10). We chose these three operating systems because they are the most common operating systems used in development stacks across the globe [13]. Within Linux, we experiment with three different distributions, which are Ubuntu-Xenial 16.04, Ubuntu-Bionic 18.04, and Ubuntu-Focal 20.04. We chose these three distributions because, during the time we were running our experiments, these three distributions were the latest Long Term Support (LTS) versions of the top three most recent Ubuntu distributions. For brevity, we drop the term 'Ubuntu' and the version number from the rest of the paper.

- *CPU Architecture:* AMD64 and ARM64. We chose these two architectures because existing works in the domain of microprocessors reveal that these two CPU architectures have been being compared against each other for a very long time [14, 15, 16] from a variety of points of view. However, no work exists that compares these two CPU architectures from the variable nature of an AI-based software perspective. Furthermore, a recent study shows that ARM64 architecture is considered an alternative to traditional AMD64 architectures which is gaining interest among software developers [17].

- *Python Version:* 3.6, 3.7, and 3.8. We chose these three versions because, during the time of our experiments, Python 3.7 was the oldest version of Python that was being maintained [18]. We compare Python 3.7 against one earlier (Python 3.6) and one later version (Python 3.8) so that features of different versions are still comparable and not significantly different from one another.

To limit the complexity of our analysis we opt to compare all configuration settings against a baseline configuration as opposed to performing pairwise comparison across all settings. We define a baseline configuration to quantify the variability by comparing other environment configurations against this baseline configuration. The baseline configuration is defined as Linux with Xenial distribution for the operating system, AMD64 for CPU architecture, and Python 3.7 for the development programming language. The reason

Table 1: Environment configurations compared against the baseline configuration: `os:linux`, `dist:xenial`, `arch:amd64`, `python:3.7`. In each row, underlined is the variable that is different from the baseline.

Purpose	Comparison	Configuration	Total Configurations
Effect of operating system	*Linux vs MacOS*	`os:osx`, `arch:amd64`, `python:3.7`	2
	Linux vs Windows	`os:windows`, `arch:amd64`, `python:3.7`	
Effect of distribution	*Linux-Xenial vs Linux-Bionic*	`os:linux`, `dist:bionic`, `arch:amd64`, `python:3.7`	2
	Linux-Xenial vs Linux-Focal	`os:linux`, `dist:focal`, `arch:amd64`, `python:3.7`	
Effect of Python version	*Python 3.6 vs Python 3.7*	`os:linux`, `arch:amd64`, `python:3.6`	2
	Python 3.7 vs Python 3.8	`os:linux`, `arch:amd64`, `python:3.8`	
Effect of CPU architecture	*AMD64 vs ARM64*	`os:linux`, `arch:arm64`, `python:3.7`	1
Total configurations compared against baseline:			**7**

behind this choice is that these were the default values set by Travis CI. A total of seven environment configurations were compared with the baseline configuration as shown in Table 1.

The configurations are defined in a `.travis.yml` file which is written in YAML-based Domain Specific Language [19]. An example of a typical `.travis.yml` file is shown in Listing 1 which defines a *build* with two *jobs* each of which has three *phases*.

```
language: python
jobs:
  include:
    - name: Python 3.6 on Linux-Xenial
      python: 3.6
      os: linux
      dist: xenial
      arch: arm64
    - name: Python 3.7 on Linux-Bionic
      python: 3.7
      os: linux
      dist: bionic
      arch: amd64
install:
  - pip3 install --upgrade pip
  - pip3 install -r requirements.txt
script:
  - python3 src/train.py
  - python3 src/test.py
after_success:
```

```
21    - echo "Successful."
```

Listing 1: An example of a simple .travis.yml file

- *Job:* A *job* is defined as an automated process that clones a repository into a virtual environment (VM). A *job* carries out a series of *phases*.
- *Phase:* A *phase* is the sequential steps of a *job*. There are two main Travis CI *phases*, namely, `install` and `script`. Installation of any dependencies required to build a software project is performed in the `install` *phase* whereas the `script` *phase* runs the build scripts. Travis CI also supports three optional *deployment phases*: `before_deploy`, `deploy`, and `after_deploy`. Custom commands like `after_success` and `after_failure` can also be run as part of a *phase*.
- *Build:* A *build* is a group of *jobs*. By default, *jobs* in a build run in sequence, although depending on one's subscription plan, *jobs* can be run concurrently.

The configuration settings of a VM are described using a set of keywords. For instance, `OS` and `language` are two configuration-related keywords. `OS` sets the Operating System of a VM for a particular *job* whereas the `language` keyword is used to prepare the VM by setting up tools of a specific programming language. In Listing 1. Python is set as the programming language for the project in line 1. Line 2 marks the beginning of the `jobs` block. In this example, two independent *jobs* are defined. The first *job* (line 4–8) will run on a VM with an ARM64 CPU and Linux-Xenial distribution as the operating system. Python version 3.6 is installed to run the Python scripts. Similarly, the second *job* (line 9–13) will run on a VM where the operating system is Linux-Bionic and CPU architecture is AMD64. Python version 3.7 is used to run the Python scripts. In both *jobs*, after the VMs are spun up, `pip` is upgraded (line 15) and required libraries are installed (line 16). Once all the dependencies are installed, two Python scripts from the `src` folder are run sequentially: `train.py` (line 18) and `test.py` (line 19). After the successful execution of the `script` *phase*, a message "Successful." is displayed on the screen (line 21).

2.2. Dataset

In this study, we use a dataset of open-source AI applications from GitHub curated by Rzig *et al.* [12]. This dataset consists of 206 projects. We chose this dataset because all these projects use Travis CI and are primarily written in Python. We focus on Python projects only because it has been reported that Python is the most popular programming language for the development of AI and ML-based solutions [20, 21, 22]. We clone all 206 projects and build them in the Travis CI platform. Once built, a project has one of the following statuses:

- *Errored:* An *errored build* has one or more *errored job(s).* A *job* that encounters an issue during the `install` *phase* receives the *errored* status.
- *Failed:* A *failed build* has one or more *failed job(s).* A *job* that encounters an issue during the `script` *phase* receives the *failed* status.
- *Passed:* A *build* receives the *passed* status when all *jobs* receive the *passed* status.

Since the goal of our work is to study variabilities in these projects, it is required that all projects are successfully built under all configuration settings described in Section 2.1. For example, if a project only runs on Linux, but not on MacOS and/or Windows, then we cannot quantify variability in this project due to the change in operating system. Unfortunately, the majority of the projects are not developed with the aim of running them on all major operating systems, CPU architectures, or multiple versions of Python. For example, fer [23] is one of the projects in the dataset. The `.travis.yml` file in this project reveals that it was developed for and tested on Linux-Xenial, AMD64 CPU architecture, and Python 3.6. While we tried to edit the `.travis.yml` files of all the projects in our dataset to incorporate all the configuration settings from Section 2.1, for 176 projects we were unsuccessful in building them in all those settings. We move forward with the 30 projects that returned a *build* status of `passed` under all configuration settings under investigation. Table 2 gives an overview of the projects used in this study. The median values of the number of commits, number of forks, number of stars, and number of contributors across these 30 projects are 331.50, 43, 179.5, and 5 respectively. Building and running 30 projects on Travis CI took a total of 1185.87 build hours and cost us 1566775 build credits which is worth $940 excluding the monthly subscription fee of $260.

Table 2: Overview of the projects used in this study.

Project	**Commits**	**Forks**	**Stars**	**Contributors**
HDI-Project/ATM	775	140	523	13
duxuhao/Feature-Selection	98	202	669	4
eyounx/ZOOpt	364	100	392	8
lucasmaystre/choix	77	27	153	3
mackelab/delfi	120	28	71	7
ibrahimsharaf/doc2vec	29	43	104	5
raphaelvallat/entropy	106	47	153	2
gbolmier/funk-svd	50	65	206	4
HealthCatalyst/healthcareai-py	929	185	308	17
beringresearch/ivis	634	43	318	8
wittawatj/kernel-gof	128	16	65	2
zuoxingdong/lagom	703	31	373	5
Erotemic/netharn	366	9	39	4
pescadores/pescador	466	12	75	8
ParrotPrediction/pyalcs	279	15	9	8
glm-tools/pyglmnet	859	83	278	25
ealcobaca/pymfe	1361	27	121	5
jkoutsikakis/pytorch-wrapper	28	18	93	1
amoussawi/recoder	196	7	55	3
fdtomasi/regain	728	12	28	3
kLabUM/rrcf	266	111	488	7
scikit-optimize/scikit-optimize	1570	548	2732	67
scitime/scitime	514	12	123	3
dmbee/seglearn	283	64	567	13
RomanovMikeV/setka	402	6	19	3
sherpa-ai/sherpa	823	53	329	14
analyticalmindsltd/smote_variants	498	138	606	5
maciejkula/spotlight	299	421	2949	10
MaxHalford/starboost	13	10	26	1
titu1994/tfdiffeq	156	52	215	3

2.3. Analysis of Variability

Evaluation Metrics:

One of the reasons why the popularity of AI-based systems has shown consistent growth over the last few years is that these systems are becoming more and more accurate in solving real-life problems. With the increasing amount of good quality data, these systems are expected to perform better over time [24]. Therefore, the primary factor that determines if an AI-based system is practically useful or not is how well it performs in accomplishing a given task. The secondary factor that influences a system's practical usefulness is whether it can accomplish a task within a reasonable amount of time. This implies that like any other traditional software system, both performance and time are critical aspects of an AI-based system as well. However, that is not all. Because AI-based systems are trained on existing data, any changes in the data cause performance degradation over time [25]. This necessitates frequent retraining of a system within a reasonable amount of time. Research in less time-consuming training of AI-based systems has been an interesting topic for a while [26, 27, 28, 29]. In order to further facilitate this process of frequent improvement of a system by retraining it many online cloud platforms offer paid services that can be utilized. These services provide users with different computation resources such as high volumes of RAM, GPUs, and TPUs. Of course, these services are not free and usually, a user needs to pay at an hourly rate [30, 31]. This brings in the third most important factor which is the expense associated with building and running an AI-based system. In our investigation of variability, we also pay attention to these three factors as discussed below:

1. Performance: This metric is determined from the performance of the AI component of the system. For each project, we create a Python script named `example.py`. In this script, we implement an example use case of respective projects. Some projects, such as StarBoost [32], already have example scripts and/or notebooks that demo one or more key use cases of those projects. In other projects where no example scripts/notebooks are available (*e.g.,* PyALCS [33]), we go through the tutorial sections of their documentation and find example use cases. This is a crucial step in our experimental setup because the `example.py` scripts define and run ML tasks like regression and classification. The outputs of these scripts are some numeric measures like *F1-score* (for classification) and R^2 (for regression). This numeric measure is the performance-related metric.

2. Processing time: This metric is obtained from the total processing time (in minutes) taken to run a project in a given environment configuration. In other words, it is the time taken to complete a *job* in Travis CI. This includes spinning up the VM, installing required libraries and modules, building the project, and running the `example.py` script.

3. Expense: This metric is obtained from the amount of Travis CI credits spent on building and running each project. The number of credits associated with processing a project in Travis CI is calculated based on the amount of time it takes from spinning up the VM to executing the last *phase* in the `.travis.yml` file. In other words, the longer it takes to complete processing a project, the more credits are spent. The number of credits required to run a project on a VM in the Travis CI environment is determined only by the operating system of the VM and nothing else. This means that credits are deducted at different rates only when operating systems are different. The billing documentation from the official Travis CI website states that the number of credits spent per minute on running a VM with Linux, Windows, and MacOS are respectively 10, 20, and 50 [34]. We realize that processing time and expense are correlated and it may seem redundant to study expense as a separate metric. However, the scale of processing time and expense can be considerably different. Let us take an arbitrary example. If a project takes 120 minutes to complete on Linux and 121 minutes to complete on MacOS, the processing time differs only by one unit, and a one-unit difference may not be significant. However, when we consider the number of credits spent, these values are $120 \times 10 = 1200$ and $121 \times 50 = 6050$ for Linux and MacOS respectively. When we convert the number of credits to the equivalent dollar amounts at a rate of 0.0006 dollars per credit as calculated from [34], they are $1200 \times 0.0006 = 0.72$ and $6050 \times 0.0006 = 3.63$ dollars for Linux and MacOS respectively. As this arbitrary example demonstrates, even a small difference in processing time can trigger a much bigger difference in expense, which implies that there can potentially be cases where variability in processing time across different settings is not significant, but the associated variability in expense can still be significant. This is why we study processing time and expense as two separate metrics in this study.

Result Analysis:

We run each project 50 times under each configuration shown in Table 1. The purpose behind choosing to generate a distribution of 50 runs per configuration per project is to mitigate random and unaccounted-for fluctuations in the metrics. For example, let us assume that we aim to determine how the *performance* of a project varies due to CPU architecture. In this case, we generate a distribution of *performance* for a project by running it 50 times under the configuration of `os:linux`, `dist:xenial`, `arch:arm64` and `python:3.7`. This distribution is then compared against the distribution of *performance* generated from 50 runs of the same project under the baseline configuration which is `os:linux`, `dist:xenial`, `arch:amd64` and `python:3.7`. It is important to note that there is always one and only one environment variable that is different from the baseline configuration. In this example, the only variable that is different from the baseline configuration is `arch`, as underlined above. We apply this condition to make sure that if the generated distributions differ, it is due to the environment variable that differs between the two settings and nothing else.

Once these two distributions are generated, we then perform two steps of analysis of variability:

Step1: For each project, we calculate the percentage change as follows:

$$P = \frac{\overline{m_o} - \overline{m_b}}{\mid \overline{m_b} \mid} \times 100 \tag{1}$$

The variables in Equation 1 are defined as follows:

- $\overline{m_b}$ is the arithmetic mean of a metric (*performance*, *processing time*, or *expense*) obtained from 50 runs of a project under the **baseline** configuration.

- $\overline{m_o}$ is the arithmetic mean of the same metric (*performance*, *processing time*, or *expense*) obtained from 50 runs of the project under one of the **other** configurations from Table 1.

- P is the percentage change between $\overline{m_b}$ and $\overline{m_o}$. **Any non-zero value of P indicates the existence of variability in a project.**

The purpose of this step is to determine, on average across 50 runs, how much variability can be observed in each project. This gives us a high-level overview of the variability patterns shown by the projects in our dataset.

Step 2: While *Step 1* of our analysis gives us an overall picture of variability for each project, in *Step 2* we set out to determine whether or not any observed variability is indeed statistically significant. To determine the statistical significance of any observed variability, we first perform Mann-Whitney U test [35] to compare two distributions. We choose Mann-Whitney U test as a nonparametric test of statistical significance because the distributions being compared are not guaranteed to follow a normal or a near-normal distribution. We set the level of significance, $\alpha = 0.05$ for this test. Next, we determine the degree of difference, also known as effect size, between the compared distributions with using Cliff's delta [36]. Cliff's delta, d, is bounded between -1 and 1. Based on the value of d, the effect size can have one of the following qualitative magnitudes [37]:

$$\text{Effect size} = \begin{cases} \text{Negligible}, & \text{if } |d| \leq 0.147 \\ \text{Small}, & \text{if } 0.147 < |d| \leq 0.33 \\ \text{Medium}, & \text{if } 0.33 < |d| \leq 0.474 \\ \text{Large}, & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

Following existing work [38], if the Mann-Whitney U test returns a *p-value* of less than 0.05 **and** the effect size obtained from Cliff's delta is not *negligible*, only then we consider the observed variability between the generated distributions as statistically significant.

Finally, we categorize the studied projects into three categories based on our two-step analysis previously described: (i) projects that show zero variability, (ii) projects that show non-zero variability which is statistically insignificant, and (iii) projects that show non-zero variability which is statistically significant. We categorize the projects in this way because the existence of variability does not necessarily mean the variability is statistically significant.

3. RQ1: (Operating System) To what extent does the operating system cause variability in AI-based systems?

As our first research question, we study variability with respect to operating systems. We perform a comparative analysis among three operating systems: Linux, MacOS, and Windows. Furthermore, we also investigate whether variability can be observed in different distributions of the same operating system. In this case, the comparative analysis is performed among three Linux LTS distributions: Xenial, Bionic, and Focal.

Table 3: Number of projects falling under different variability types due to differences in operating systems.

Metric	Variability Type	Linux vs MacOS	Linux vs Windows
Performance	Zero variability	4 (13.33%)	4 (13.33%)
	Non-zero but statistically insignificant	19 (63.33%)	20 (66.67%)
	Non-zero and statistically significant	7 (23.33%)	6 (20%)
Processing Time	Zero variability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	1 (3.33%)	0 (0%)
	Non-zero and statistically significant	29 (96.67%)	30 (100%)
Expense	Zero variability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	0 (0%)	0 (0%)
	Non-zero and statistically significant	30 (100%)	30 (100%)

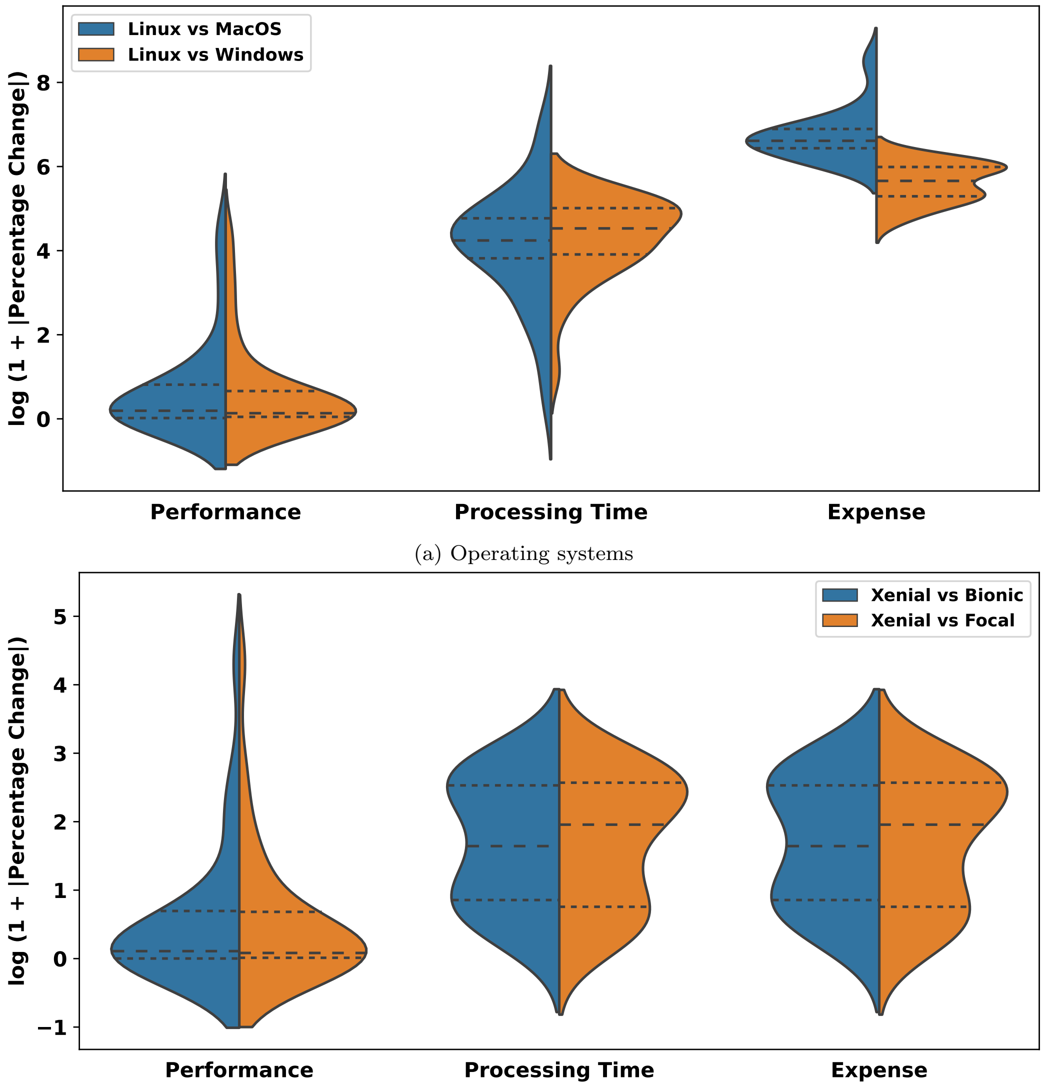
3.1. Variability with respect to Operating System

Setup:

To study the effect of operating systems on AI-based systems, we keep the CPU architecture and Python version constant to their default values and vary the choice of operating system only.

Findings:

Figure 1a shows the distributions of percentage change (P) due to different operating systems across the studied projects. Table 3 reports the number of projects falling under different variability categories defined in Section 2.3. We observe that a majority of the studied projects show changes in all three metrics due to changes in operating systems. However, only a few projects show variability in performance with statistical significance. On the other hand, almost all observed variability in processing time and expense is statistically significant. Paying closer attention to the breakdown of effect size for the projects with statistically significant variability, we find that in almost all cases the observed variability is large as shown in Table 4. We further find that there is a slight increase in performance (1.44%) on average when projects are run on MacOS compared to Linux. On the other hand, performance drops on average by 4.21% when projects are run on Windows. In processing time and expense, MacOS and Windows are always more expensive than Linux. Our findings implied that Linux is a faster and more cost-effective operating system than both MacOS and Windows. Although a slight increase in performance may be achieved on MacOS compared to Linux, this will require sacrifice in processing time and expense with MacOS taking 137% longer processing time and costing 1085.47% more money in comparison to Linux.



(a) Operating systems

(b) Linux distributions

Figure 1: Distributions of variability with respect to Operating Systems and Linux Distributions.

Table 4: Breakdown of effect size for projects with statistically significant variability due to different operating systems.

	Linux vs MacOS				**Linux vs Windows**			
	Small	Medium	Large	Total	Small	Meidum	Large	Total
Performance	1	0	6	7	0	0	6	6
Processing Time	0	0	29	29	1	0	29	30
Expense	0	0	30	30	0	0	30	30

Table 5: Number of projects falling under different variability types due to differences in Linux distributions.

Metric	**Variability Type**	**Xenial vs Bionic**	**Xenial vs Focal**
Performance	Zero variability	7 (23.33%)	6 (20%)
	Non-zero but statistically insignificant	23 (76.67%)	21 (70%)
	Non-zero and statistically significant	0 (0%)	3 (10%)
Processing Time	Zero Variability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	7 (23.33%)
	Non-zero and statistically significant	0 (0%)	23 (76.67%)
Expense	Zero Variability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	7 (23.33%)
	Non-zero and statistically significant	0 (0%)	23 (76.67%)

3.2. Variability with respect to Linux Distribution

Setup:

To study whether variability can be observed in different distributions of the same operating system, we vary only the distribution variable in the configuration settings and keep the operating system, CPU architecture, and Python version constant to the default values.

Findings:

Figure 1b shows the distribution of percentage change (P) caused because of changes in the Linux distribution across the studied projects. Table 5 reveals that the majority of the projects show some degree of variability between different distributions of Linux. Although none of the observed variability between Xenial and Bionic is statistically significant in any of the metrics, the observed variability between Xenial and Focal shows a different pattern. Between Xenial and Focal, three projects show a statistically significant variability in performance whereas 23 projects show a statistically significant variability in processing time and expense. Most of the observed statistically significant variability is large in terms of effect size as shown in Table 6. On average a slight performance gain of 2% can be achieved

Table 6: Breakdown of effect size for projects with statistically significant variability between different Linux distributions.

	Xenial vs Bionic				Xenial vs Focal			
	Small	Medium	Large	Total	Small	Medium	Large	Total
Performance	0	0	0	0	1	0	2	3
Processing Time	0	0	0	0	4	1	18	23
Expense	0	0	0	0	4	1	18	23

by choosing Focal over Xenial, however, this comes with a 7% increase in processing time and expense.

Our findings indicate that even though the choice of Linux distribution is unlikely to affect the performance of AI components significantly, it is very likely to affect the processing time and associated cost of building and running a system.

> A small portion of the studied projects show a variability in performance which is statistically significant between different operating systems. In the case of different distributions of the same operating system, the frequency of statistically significant performance-related variability is even less. However, the variability observed in processing time and expense is statistically significant in most projects with respect to both different operating systems and different distributions.

4. RQ2: (Python Version) How does the Python version contribute to the variability in AI-based systems?

Setup:

In RQ2, we investigate if variability can be observed when different versions of Python are used to run the same system. To run our experiments for this RQ, we keep all configuration variables constant except the Python version.

Findings:

Figure 2 shows a similar pattern to the observed variability in RQ1. Although a majority of the studied projects show some variability between Python versions, not all observed variability has statistical significance as shown in Table 7. Furthermore, Table 8 reveals that any variability observed

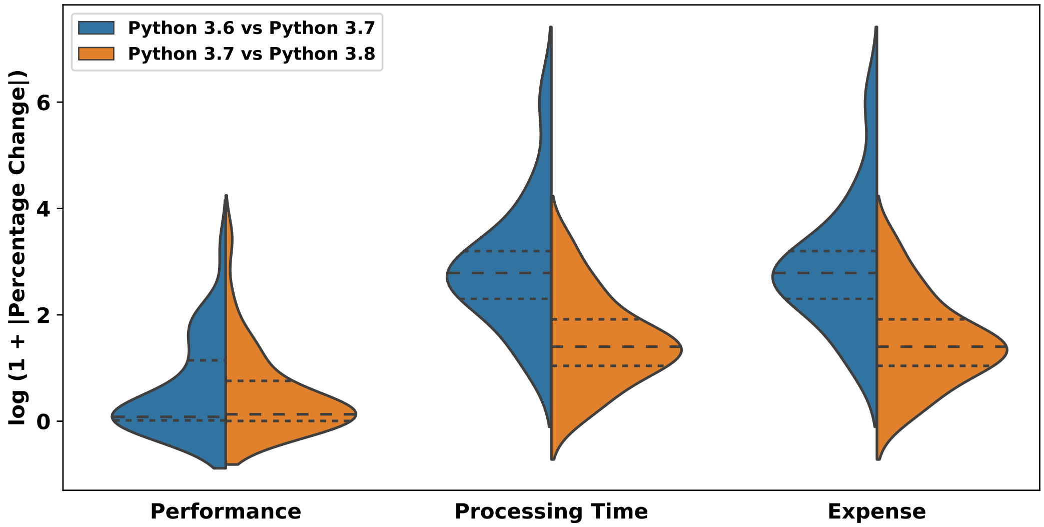


Figure 2: Distributions of variability with respect to Python versions.

Table 7: Number of projects falling under different variability types due to differences in Python versions.

Metric	Variability Type	Python 3.6 vs Python 3.7	Python 3.7 vs Python 3.8
Performance	Zero variability	6 (20%)	6 (20%)
	Non-zero but statistically insignificant	24 (80%)	19 (63.33%)
	Non-zero and statistically significant	0 (0%)	5 (16.67%)
Processing Time	Zero variability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	6 (20%)
	Non-zero and statistically significant	0 (0%)	24 (80%)
Expense	Zero variability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	6 (20%)
	Non-zero and statistically significant	0 (0%)	24 (80%)

Table 8: Number of projects out of 30 with statistically significant variability between different Python versions broken down by effect size.

	Python 3.6 vs Python 3.7				**Python 3.7 vs Python 3.8**			
	Small	Medium	Large	Total	Small	Meidum	Large	Total
Performance	0	0	0	0	1	0	4	5
Processing Time	0	0	0	0	3	2	19	24
Expense	0	0	0	0	3	2	19	24

between Python 3.6 and Python 3.7 is insignificant in all metrics. On the other hand, five projects with four large effect sizes and one small effect size show significant variability between Python 3.7 and Python 3.8. An even higher degree of variability can be observed in processing time and expense with a total of 24 projects showing significant variability with 19 large, two medium, and three small effect sizes. Moreover, choosing Python 3.6 over Python 3.7 causes a 0.52% drop in performance, and choosing Python 3.8 over Python 3.7 causes a 0.73% drop in performance on average. To build and run a project it takes 25% longer using Python 3.6 and 5.3% longer using Python 3.8. Expense follows the same pattern as processing time.

Findings from RQ2 indicate that the choice of Python version can induce variability. This could happen due to the fact that building a project requires many libraries to install and different versions of Python can cause outdated or newer versions of the libraries to install. This in turn may cause the performance, processing time, and expense to vary. For example, newer libraries may have additional features that may cause longer installation time, whereas, outdated libraries may cause performance drops and longer processing time due to their internal dependency on even more outdated libraries.

Python 3.6 and Python 3.7 produce an identical behavior of AI-based systems in all three metrics. Between Python 3.7 and Python 3.8, only a small fraction of projects show significant variability in performance, while the majority of the projects show significant variability in terms of processing time and expense.

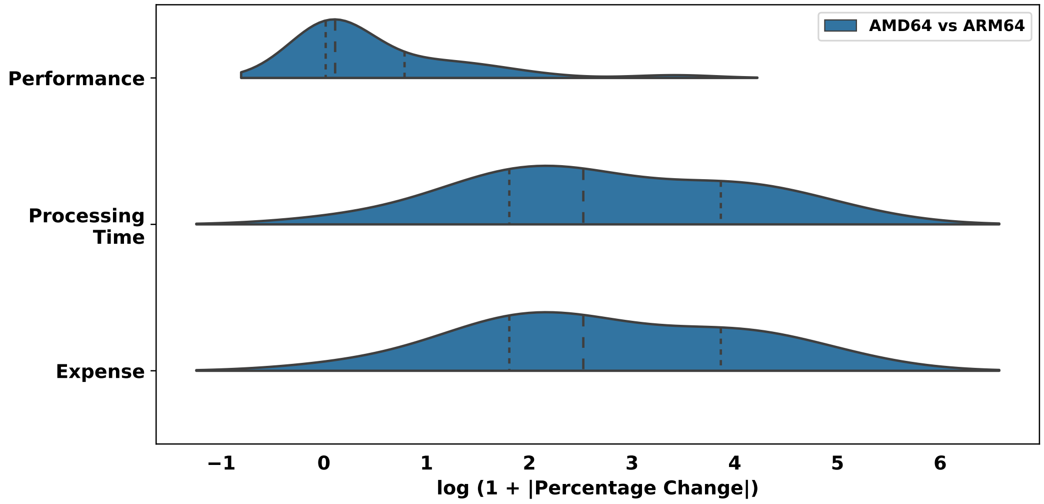


Figure 3: Distribution of percentage of average changes between AMD64 and ARM64 CPU architectures.

5. RQ3: (CPU Architecture) How does CPU architecture affect the variability in AI-based systems?

Setup:

We study the effect of CPU architecture on variability by keeping the operating system, distribution, and Python version constant to the default value and only varying CPU architecture configuration.

Findings:

Figure 3 summarize the variability pattern in terms of percentage changes between AMD64 and ARM64 CPU architectures. Similar to the findings from RQ1 and RQ2, most of the observed variability in performance is insignificant, whereas, in processing time and expense, the observed variability is significant in the majority of the cases. Table 10 shows that out of six projects with significant variability five show a large effect size and one shows a small effect size. In processing time and expense two, one and 25 projects show small, medium, and large effect sizes respectively among the 28 projects that differ significantly between AMD64 and ARM64 CPU architectures. In all three metrics, the ARM64 CPU performs poorly compared to the AMD64 CPU with a slight drop of 0.62% in performance costing 25% more time and money, on average. We conjecture that the observed variability between AMD64 and ARM64 CPU architectures may be happening

Table 9: Number of projects falling under different variability types due to differences in CPU architectures.

Metric	**Variability Type**	**AMD64 vs ARM64**
Performance	Zero variability	4 (13.33%)
	Non-zero but statistically insignificant	20 (66.67%)
	Non-zero and statistically significant	6 (20%)
Processing Time	Zero variability	0 (0%)
	Non-zero but statistically insignificant	2 (6.67%)
	Non-zero and statistically significant	28 (93.33%)
Expense	Zero variability	0 (0%)
	Non-zero but statistically insignificant	2 (6.67%)
	Non-zero and statistically significant	28 (93.33%)

Table 10: Number of projects showing statistically significant variability between AMD64 and ARM64 CPU architectures broken down by effect size.

	AMD64 vs ARM64			
	Small	Medium	Large	Total
Performance	1	0	5	6
Processing Time	2	1	25	28
Expense	2	1	25	28

due to design differences between them. ARM64 has a much smaller instruction set compared to AMD64 which might require ARM64 to take longer to perform more complex operations [39]. AMD64 being the most common CPU architecture has more software support compared to ARM64 CPUs. All these can negatively affect the performance, processing time, and expense of building and running AI-based systems on ARM64 CPUs.

We can draw a similar conclusion for RQ3 to what we observed and concluded in RQ1 and RQ2. Variability in performance is less common than variability in processing time and associated costs. Therefore, the most optimized hardware configuration can significantly reduce processing time and costs because in the majority of the cases, the observed statistically significant variability is large between AMD64 and ARM64 CPU architectures.

CPU architecture affects the processing time and associated expenses significantly in most studied projects. More frequently than not, the effect size in these two metrics is large. Although less likely, the choice of CPU architecture can still induce variability in performance in some projects.

6. Discussion

AI components are becoming a core part of almost all software systems nowadays. Our analysis shows that these systems suffer from variability in all three metrics (*performance*, *processing time*, and *expense*) we studied although this finding is not consistent across all projects under investigation. Although existing works reported the variable nature of AI-based systems due to different factors such as choice of frameworks [40, 41], underspecification [42] and CPU multithreading [43], to the best of our knowledge, ours is the first work to investigate the effect of environment configurations on variability. We find that the choice of operating system including the distribution of an operating system, version of Python, and CPU architecture indeed induce variability. The degree of variability differs from project to project which implies that to be able to determine the existence and degree of variability in a project, developers must build and run the project under various configuration settings. Furthermore, our findings indicate that not all metrics show an equal degree of variability. Variability is more prominent in processing time and associated costs than the performance of an AI component. This can have an adverse effect on a project's development lifecycle given that AI components need to be retrained frequently because they suffer from performance decay due to data drift [25] and concept drift [44] over time. If the (re)training of an AI component takes a very long time under a given configuration setting it can reduce the frequent update of the product and eventually can lead to a performance drop. Moreover, longer processing time usually implies higher costs.

While variability in processing time and associated expenses impacts only the project internally (such as longer development time, and unnecessary increase in development efforts), the variability in performance can impact the end-users of the project. This can potentially cause a financial burden for a company. Therefore, an AI-based system should be built and run on different configuration settings before deployment so that the developers can determine the most optimized configuration of the environment on which the product will be deployed. Based on existing literature [7], it is not yet a common practice in the industry. Our findings imply that determining the best configuration setting with respect to the metric(s) of interest should be an important step in the development workflow. We acknowledge that this additional step is likely to increase the overall development time, however, this step is crucial to save time and cost in the long run.

7. Threats to Validity

We discuss threats to validity of this paper in this section.

Internal Validity:

There are projects in our dataset that are developed for more than one task. However, we only run one example task as part of our example script for each project in order to perform an analysis of the outputs. It is entirely possible that the example tasks may not represent the actual degree of variability associated with the project. For example, running a classification model with a curated or noise-free dataset (such as the iris dataset) may result in a very accurate classification model regardless of development environment configurations. However, the same model trained on a more noisy as well as relatively 'harder to classify' dataset may produce a very different level of accuracy, hence a very different degree of variability. In order to mitigate this issue, we only choose an example task that is part of the official documentation of each project with a naive assumption that the developers would choose those examples in such a way that they are a true representation of the overall functionality and performance of the project.

External Validity:

We cannot guarantee the generalizability of our findings. We chose a finite set of configuration variables with a finite set of possible values for each of the variables under study. However, we acknowledge that there are other options for each of these variables that we do not investigate. For example, Linux has many distributions other than Xenial, Bionic, and Focal. Python has many other versions besides the ones we studied. Therefore, we do not claim that our findings can be generalized beyond what we investigated. The reason behind limiting our choices of options for the configuration variables is the amount of time and money required to run experiments in Travis CI.

Furthermore, for each new configuration setting, we would have had to run 50 iterations because of our experimental design. Doing so was not practically feasible due to constraints on time and money.

8. Related Work

Non-determinism in AI:

Uncertain nature of AI components has been a topic of research in the domain of AI for quite some time. It has gained more traction with the popularity of deep learning systems. Most of the existing works on the non-deterministic nature of AI components focus on deep learning systems. For example, Zhuang *et al.* [40] studied the uncertain nature of training deep learning models. They reported that the choice of tools can have an effect on the behavior of an AI component which can potentially affect AI safety. Guo *et al.* [41] performed an empirical study on the development and deployment of deep learning solutions. They reported that frameworks and platforms can cause the performance of a system to decline. Crane [45] studied the challenges in the reproducibility of published results. He reported that random consistent use of random seeds can help mitigate the issue with reproducibility. Xiao *et al.* [43] reported the impact of CPU multithreading and how it impacts the training of deep learning systems.

Variability in Software:

Variability in traditional software systems is not a new topic in the software engineering domain. There are prior works done on the variable nature of software systems in general. Coplien *et al.* [46] described how to perform domain engineering by identifying the commonalities and variabilities within a family of products. Anda *et al.* [47] performed the study on variability in software systems from a practical perspective and made a connection between reproducibility and variability. In [48], Bachmann *et al.* performed an extensive study on variability in software product lines. Thiel *et al.* [49] focuses on variability in autonomous systems. Jaring *et al.* [50] suggested a representation and normalization of variability.

AI-components in Software:

Many recent studies have investigated the pros and cons of having AI components embedded in software systems. Masuda *et al.* [51] described practices for the evaluation and improvement of the software quality of ML applications. Washizaki *et al.* [52] proposed architecture and design patterns for ML systems. An extensive study on testing ML applications was performed in [53] by Zhang *et al.*. Scully *et al.* [54] studied hidden technical debt in ML systems whereas Obrien *et al.* [55] studied self-admitted technical debts in ML software.

Our work is different from the above studies in that ours is the first study to quantify the degree of variability between different environment configuration settings.

9. Conclusion and Future Work

In this paper, we investigate how AI-based software shows variability in terms of three metrics: performance, processing time, and expense of building and running a system. We perform our study with respect to three environment variables, namely operating systems including the distributions of an operating system, Python version, and CPU architecture. Our study shows that although a majority of the projects show some degree of variability, the degrees vary from project to project. The variability is more statistically significant for processing time and expense than the performance of an AI component. Because the observed variability patterns vary from project to project, we conclude that in order to serve the end users the most accurate AI solutions, it is crucial to run and test the AI components in different environment configurations. At the same time, it is a common requirement of any AI-based solution to retrain the model(s) in a predefined interval. Therefore, processing time and expense are also something that should be taken into consideration as they also vary significantly from one configuration to another. To the best of our knowledge, the only way to gauge the degree of variability of a system is to run it on different configuration settings. Being able to predict the degree of variability without having to run the system on different configuration settings can save a lot of time and effort. Therefore, this can be an interesting topic for future research. Another interesting follow-up study can be on the reasons behind the observed variability. For example, in this study we find the existence of variability between Python 3.7 and Python 3.8, however, why this variability exists is beyond the scope of this paper. We leave this as a topic for future research.

References

[1] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, S. Wagner, Software engineering for ai-based systems: a survey, ACM Transactions on Software Engineering and Methodology (TOSEM) 31 (2) (2022) 1–59.

[2] M. Kläs, A. M. Vollmer, Uncertainty in machine learning applications: A practice-driven classification of uncertainty, in: Computer Safety, Reliability, and Security: SAFECOMP 2018 Workshops, ASSURE, DECSoS, SASSUR, STRIVE, and WAISE, Västerås, Sweden, September 18, 2018, Proceedings 37, Springer, 2018, pp. 431–438.

[3] B. Kompa, J. Snoek, A. L. Beam, Second opinion needed: communicating uncertainty in medical machine learning, NPJ Digital Medicine 4 (1) (2021) 4.

[4] B. Hammer, T. Villmann, How to process uncertainty in machine learning?, in: ESANN'2007 proceedings - European Symposium on Artificial Neural Networks, Bruges (Belgium), 25-27 April 2007, 2007, pp. 79–90.

[5] H. Belani, M. Vukovic, Ž. Car, Requirements engineering challenges in building ai-based complex systems, in: 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW), IEEE, 2019, pp. 252–255.

[6] M. Felderer, R. Ramler, Quality assurance for ai-based systems: Overview and challenges (introduction to interactive session), in: Software Quality: Future Perspectives on Software Engineering Quality: 13th International Conference, SWQD 2021, Vienna, Austria, January 19–21, 2021, Proceedings 13, Springer, 2021, pp. 33–42.

[7] M. M. John, H. Holmström Olsson, J. Bosch, Architecting ai deployment: A systematic review of state-of-the-art and state-of-practice literature, in: Software Business: 11th International Conference, ICSOB 2020, Karlskrona, Sweden, November 16–18, 2020, Proceedings 11, Springer, 2021, pp. 14–29.

[8] J. M. Wicherts, C. L. Veldkamp, H. E. Augusteijn, M. Bakker, R. Van Aert, M. A. Van Assen, Degrees of freedom in planning, running,

analyzing, and reporting psychological studies: A checklist to avoid p-hacking, Frontiers in psychology (2016) 1832.

[9] J. P. Simmons, L. D. Nelson, U. Simonsohn, False-positive psychology: Undisclosed flexibility in data collection and analysis allows presenting anything as significant, Psychological science 22 (11) (2011) 1359–1366.

[10] Replication package: scripts and data. (Jun 2024).
URL https://anonymous.4open.science/r/variability_replication_package-F175/

[11] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 426–437.

[12] D. E. Rzig, F. Hassan, C. Bansal, N. Nagappan, Characterizing the usage of ci tools in ml projects, in: Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2022, pp. 69–79.

[13] Desktop operating system market share worldwide — statcounter global stats, https://gs.statcounter.com/os-market-share/desktop/worldwide, (Accessed on 06/07/2024) (May 2024).

[14] E. Blem, J. Menon, K. Sankaralingam, Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures, in: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2013, pp. 1–12.

[15] D. Bhandarkar, Risc versus cisc: a tale of two chips, ACM SIGARCH Computer Architecture News 25 (1) (1997) 1–12.

[16] K. Sankaralingam, J. Menon, E. Blem, A detailed analysis of contemporary arm and x86 architectures, Tech. rep. (2013).

[17] X. Chen, L.-H. Hung, R. Cordingly, W. Lloyd, X86 vs. arm64: An investigation of factors influencing serverless performance, in: Proceedings of the 9th International Workshop on Serverless Computing, 2023, pp. 7–12.

[18] Status of python versions, https://devguide.python.org/versions/, (Accessed on 06/07/2024) (Jun 2024).

[19] Domain-specific language (Jun 2024).
URL https://en.wikipedia.org/wiki/Domain-specific_language

[20] S. Sultonov, Importance of python programming language in machine learning., International Bulletin of Engineering and Technology 3 (9) (2023) 28–30.

[21] S. Raschka, J. Patterson, C. Nolet, Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence, Information 11 (4) (2020) 193.

[22] D. Gonzalez, T. Zimmermann, N. Nagappan, The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github, in: Proceedings of the 17th International conference on mining software repositories, 2020, pp. 431–442.

[23] F. Developers, GitHub - JustinShenk/fer: Facial Expression Recognition with a deep neural network as a PyPI package — github.com, https://github.com/JustinShenk/fer, [Accessed 27-05-2024] (Sep 2021).

[24] P. Domingos, A few useful things to know about machine learning, Communications of the ACM 55 (10) (2012) 78–87.

[25] K. Nelson, G. Corbin, M. Anania, M. Kovacs, J. Tobias, M. Blowers, Evaluating model drift in machine learning algorithms, in: 2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), IEEE, 2015, pp. 1–8.

[26] A. Kavikondala, V. Muppalla, K. Krishna Prakasha, V. Acharya, Automated retraining of machine learning models, International Journal of Innovative Technology and Exploring Engineering 8 (12) (2019) 445–452.

[27] Y. Wu, E. Dobriban, S. Davidson, Deltagrad: Rapid retraining of machine learning models, in: International Conference on Machine Learning, PMLR, 2020, pp. 10355–10366.

[28] A. Mahadevan, M. Mathioudakis, Cost-aware retraining for machine learning, Knowledge-Based Systems 293 (2024) 111610.

[29] J. Kim, S. S. Woo, Efficient two-stage model retraining for machine unlearning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 4361–4369.

[30] Ec2 on-demand instance pricing – amazon web services, `https://aws.amazon.com/ec2/pricing/on-demand/`, (Accessed on 06/07/2024) (Jun 2024).

[31] Vm instance pricing — compute engine: Virtual machines (vms) — google cloud, `https://cloud.google.com/compute/vm-instance-pricing`, (Accessed on 06/07/2024) (Jun 2024).

[32] M. Halford, Starboost, `https://github.com/MaxHalford/starboost`, [Accessed 30-12-2023] (2018).

[33] P. Developers, Pyalcs: Anticipatory learning classifier systems in python, `https://github.com/ParrotPrediction/pyalcs`, [Accessed 30-12-2023] (2018).

[34] Travis ci documentation (Jun 2024).
URL `https://docs.travis-ci.com/user/billing-overview/`

[35] H. B. Mann, D. R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, The annals of mathematical statistics (1947) 50–60.

[36] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions., Psychological bulletin 114 (3) (1993) 494.

[37] M. R. Hess, J. D. Kromrey, Robust confidence intervals for effect sizes: A comparative study of cohen's d and cliff's delta under non-normality and heterogeneous variances, in: annual meeting of the American Educational Research Association, Vol. 1, Citeseer, 2004.

[38] S. Khatoonabadi, D. E. Costa, R. Abdalkareem, E. Shihab, On wasted contributions: understanding the dynamics of contributor-abandoned pull requests–a mixed-methods study of 10 large open-source projects, ACM Transactions on Software Engineering and Methodology 32 (1) (2023) 1–39.

[39] D. Bhandarkar, D. W. Clark, Performance from architecture: comparing a risc and a cisc with similar hardware organization, in: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, 1991, pp. 310–319.

[40] D. Zhuang, X. Zhang, S. Song, S. Hooker, Randomness in neural network training: Characterizing the impact of tooling, Proceedings of Machine Learning and Systems 4 (2022) 316–336.

[41] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, X. Li, An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 810–822.

[42] A. D'Amour, K. Heller, D. Moldovan, B. Adlam, B. Alipanahi, A. Beutel, C. Chen, J. Deaton, J. Eisenstein, M. D. Hoffman, et al., Underspecification presents challenges for credibility in modern machine learning, Journal of Machine Learning Research 23 (226) (2022) 1–61.

[43] G. Xiao, J. Liu, Z. Zheng, Y. Sui, Nondeterministic impact of cpu multithreading on training deep learning systems., in: ISSRE, 2021, pp. 557–568.

[44] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, G. Zhang, Learning under concept drift: A review, IEEE transactions on knowledge and data engineering 31 (12) (2018) 2346–2363.

[45] M. Crane, Questionable answers in question answering research: Reproducibility and variability of published results, Transactions of the Association for Computational Linguistics 6 (2018) 241–252.

[46] J. Coplien, D. Hoffman, D. Weiss, Commonality and variability in software engineering, IEEE software 15 (6) (1998) 37–45.

[47] B. C. Anda, D. I. Sjøberg, A. Mockus, Variability and reproducibility in software engineering: A study of four companies that developed the same system, IEEE Transactions on Software Engineering 35 (3) (2008) 407–429.

[48] F. Bachmann, P. Clements, Variability in software product lines, Carnegie Mellon University, Software Engineering Institute, 2005.

[49] S. Thiel, A. Hein, Modelling and using product line variability in automotive systems, IEEE software 19 (4) (2002) 66–72.

[50] M. Jaring, J. Bosch, Representing variability in software product lines: A case study, in: International Conference on Software Product Lines, Springer, 2002, pp. 15–36.

[51] S. Masuda, K. Ono, T. Yasue, N. Hosokawa, A survey of software quality for machine learning applications, in: 2018 IEEE International conference on software testing, verification and validation workshops (ICSTW), IEEE, 2018, pp. 279–284.

[52] H. Washizaki, H. Uchida, F. Khomh, Y.-G. Guéhéneuc, Studying software engineering patterns for designing machine learning systems, in: 2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP), IEEE, 2019, pp. 49–495.

[53] J. M. Zhang, M. Harman, L. Ma, Y. Liu, Machine learning testing: Survey, landscapes and horizons, IEEE Transactions on Software Engineering 48 (1) (2020) 1–36.

[54] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, D. Dennison, Hidden technical debt in machine learning systems, Advances in neural information processing systems 28 (2015).

[55] D. OBrien, S. Biswas, S. Imtiaz, R. Abdalkareem, E. Shihab, H. Rajan, 23 shades of self-admitted technical debt: an empirical study on machine learning software, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 734–746.