The Impact of Environment Configurations on the Stability of AI-Enabled Systems

Musfiqur Rahman
Concordia University
Montréal, Canada
musfiqur.rahman@mail.concordia.ca

SayedHassan Khatoonabadi
Concordia University
Montréal, Canada
sayedhassan.khatoonabadi@concordia.ca

Ahmad Abdellatif
University of Calgary
Calgary, Canada
ahmad.abdellatif@ucalgary.ca

Haya Samaana
An-Najah National University
Nablus, Palestine
hayasam@najah.edu

Emad Shihab
Concordia University
Montréal, Canada
emad.shihab@concordia.ca

Abstract

Nowadays, software systems tend to include Artificial Intelligence (AI) components. Changes in the operational environment have been known to negatively impact the stability of AI-enabled software systems by causing unintended changes in behavior. However, how an environment configuration impacts the behavior of such systems has yet to be explored. Understanding and quantifying the degree of instability caused by different environment settings can help practitioners decide the best environment configuration for the most stable AI systems. 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-enabled systems using the Travis CI platform. We determine the existence and the degree of instability introduced by each configuration using three metrics: the output of an AI component of the system (model performance), the time required to build and run the system (processing time), and the cost associated with building and running the system (expense). Our results indicate that changes in environment configurations lead to instability across all three metrics; however, it is observed more frequently with respect to processing time and expense rather than model performance. For example, between Linux and MacOS, instability is observed in 23%, 96.67%, and 100% of the studied projects in model performance, processing time, and expense, respectively. Our findings underscore the importance of identifying the optimal combination of configuration settings to mitigate drops in model performance and reduce the processing time and expense before deploying an AI-enabled system.

CCS Concepts

• **Software and its engineering → Operational analysis**; *Empirical software validation.*

Keywords

Instability, Open-source Software, SE4AI, AI-enabled Systems, Empirical Software Engineering

ACM Reference Format:
Musfiqur Rahman, SayedHassan Khatoonabadi, Ahmad Abdellatif, Haya Samaana, and Emad Shihab. 2025. The Impact of Environment Configurations on the Stability of AI-Enabled Systems. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With the recent advances 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 systems are integrating such models as part of their pipeline [49]. Software systems are inherently complex and the issue of stability in software systems has been previously investigated [11, 47]. In addition, any ML model is, at its core, probabilistic and, as a result, suffers from uncertainty. [34, 43, 44]. The complexity of a software system, combined with the nondeterministic nature of an ML model, can make AI-enabled systems behave inconsistently across different operational environments. In this study, the term 'operational environment' refers to any environment where an AI-enabled system is built and/or served, such as development and deployment environments. This inconsistent behavior introduces instability—the phenomenon where a piece of software behaves differently when the operational environment changes, even though the internal software artifacts, such as code and input data, remain the same. Such instability indicates low adaptability [1] of a system, which is undesirable system behavior.

In practice, development and deployment environments may differ. The potential for a substantial difference between development and deployment behavior, termed *training/serving skew*, has been reported before [15, 59]. For example, an arbitrary face recognition system that achieves an F1 score of, say 0.9, in the development environment may not achieve a similar F1 score once deployed in a different environment configuration. Therefore, understanding how an ML model may behave differently after deployment compared to its behavior in the development environment is a crucial aspect of AI-enabled software development. Although the literature has previously investigated the instability in the behavior of ML

EASE 2025, Istanbul, Türkiye

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

models from the ML algorithm perspective [39, 62], the developers must also determine the degree of instability that can be introduced due to environment configurations as well. Therefore, running the system under different configuration settings should be an additional step before deployment of the system to determine whether model performance varies significantly across configurations. 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 begun investigating the challenges associated with the uncertain nature of AI-enabled software systems. These challenges affect various aspects of the development life cycle, including requirement elicitation [10], software testing and quality assurance [30], and deployment [37].

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 settings, such as operating systems, versions of a programming language, 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 introduce instability in the prediction quality of AI/ML models as "practitioners' degrees of freedom" [60, 65], which is a known issue in the field of applied statistics. However, in the domain of software engineering, there is no existing work studying the potential sources of instability in AI-enabled software from an environment configuration perspective. We use the term 'instability' as a quantitative measure throughout the study to assess the extent to which an AI-enabled system behaves differently when environment configurations change. We aim to achieve this goal by experimenting with eight combinations of three environment variables, namely operating system, Python version, and CPU architecture. We conduct experiments on 30 open-source AI-enabled projects using the Travis CI platform, measuring instability in model performance, processing time, and expense. Specifically, we aim to answer the following three research questions:

RQ1: (Operating System) How much instability is introduced by changing the operating system in AI-enabled systems? We analyze whether variations in operating systems make AI-enabled systems behave differently. We observed instability in model performance across 23% of the projects between Linux and MacOS whereas 20% of the projects show such instability between Linux and Windows. Almost all projects show significant instability in processing time and expense between different operating systems.

RQ2: (Python Version) How much does changing the Python version introduce instability in AI-enabled systems? Python is the most frequently used programming language for AI-enabled systems. Therefore, it is critical to investigate the effect of Python versions on the behavior of AI-enabled 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, instability can be observed in about 17% of the projects in model performance and 80% of the projects in both processing time and expense.

RQ3: (CPU Architecture) How much does changing the CPU architecture introduce instability in AI-enabled 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 instability in processing time and expense while only 20% of them vary in model performance between AMD64 and ARM64 architectures.

Our findings imply that changes in configuration settings are very likely to introduce significant instability in AI-enabled systems although the degree varies from project to project. Significant instability in AI-enabled systems in processing time and expense is more frequently observed than model 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 system.

In summary, this paper makes the following contributions:

- To the best of our knowledge, this is the first empirical study on the instability of AI-enabled systems from the environment configuration point of view.
- We provide empirical evidence behind the necessity of *dev/prod parity* principle where development and production environments are kept similar as much as possible [36].
- We make our data and scripts available for reproducibility and future research [5].

The rest of the paper is organized as follows. Section 2 covers background and methodology. Sections 3–5 present findings for each research question. Sections 6 and 7 discuss our results and threats to validity. Section 8 reviews related work, and Section 9 concludes with a summary and future directions.

2 Methodology and Background

We use Travis CI—a widely used Continuous Integration (CI) platform [35]—to run experiments across different operational configurations. We chose Travis CI because it's the most popular CI tool among OSS developers for AI-enabled systems [57].

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 system is built and run, the CPU is the core of the hardware on which a system is run, and the programming language is at the core of development tech stack used for building a system. 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 (10 version 1803). We chose these three operating systems because they are the most common operating systems used in development stacks across the globe [2]. 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 will only use the distribution name throughout the rest of the paper.

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 [6]. Furthermore, the majority of the projects in our dataset were developed using Python 3.7 or older versions. 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.

CPU Architecture: AMD64 and ARM64. We chose these two architectures because they are commonly compared against each other from a variety of points of view [12, 14, 58]. Furthermore, a recent study shows that ARM64 architecture is considered an alternative to traditional AMD64 architectures, which is gaining interest among software developers [17].

We compare all configuration settings against a baseline configuration to quantify the instability. The baseline configuration is defined as Linux with Xenial distribution for the operating system, AMD64 for the CPU architecture, and Python 3.7 for the programming language. The reason behind this choice is that these were the default values set by Travis CI at the time of conducting the experiments. It is important to note that there is always one and only one environment variable that is different from the baseline configuration. We apply this condition to make sure that if there is instability, it is due to the variable that is different from the baseline and nothing else. A total of seven environment configurations are selected to be 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 [3]. 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*.

```
1  language: python
2  jobs:
3    include:
4      - name: Python 3.6 on Linux-Xenial
5        python: 3.6
6        os: linux
7        dist: xenial
8        arch: arm64
9      - name: Python 3.7 on Linux-Bionic
10       python: 3.7
11       os: linux
12       dist: bionic
13       arch: amd64
14 install:
15   - pip3 install --upgrade pip
16   - pip3 install -r requirements.txt
17 script:
18   - python3 src/train.py
19   - python3 src/test.py
20 after_success:
21   - echo "Successful."
```

Listing 1: An example of a typical `.travis.yml` file

We define the three key Travis CI terminologies below:

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 one sequential step 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 the 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-enabled projects from GitHub curated by Rzig *et al.* [57]. 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 [31, 56, 61]. We clone all projects and build them in the Travis CI platform using the existing `.travis.yml` files. 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 instability 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

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

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

Table 2: Overview of the 30 projects used in this study.

	Avg.	Std.	Min.	Med.	Max.
Commits	437.33	393.24	13.00	331.50	1570.00
Forks	84.17	122.10	6.00	43.00	548.00
Stars	402.90	690.30	9.00	179.50	2949.00
Contributors	8.70	12.22	1.00	5.00	67.00

only runs on Linux, but not on MacOS and/or Windows, then we cannot quantify instability due to the change in operating system in this project. However, 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 [24] is one of the projects in the dataset. The `.travis.yml` file in this project shows 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 the dataset to incorporate all the configuration settings from Section 2.1, for the majority of the projects we were unsuccessful in building them in all those settings because open-source projects usually are developed and tested on a small subset of many possible configuration settings. For example, several projects were built using Python 3.6 and when we tried to build them with Python 3.8 they failed due to dependency issues and version mismatch between Python libraries. 30 projects returned a *build* status of `passed` under all configuration settings under investigation. We move forward with these 30 projects for further analysis. Table 2 provides an overview of the 30 projects used in this study. The full list of studied projects can be found in the replication package [5]. Building and running 30 projects on Travis CI took a total of 1185.87 build hours and cost us 1, 566, 775 build credits which is worth $940 excluding the monthly subscription fee of $260.

2.3 Analysis of Instability

2.3.1 Evaluation Metrics: One of the reasons why the popularity of AI-enabled 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 high-quality data, these systems are expected to perform better over time [29]. Therefore, the primary factor that determines if an AI-enabled 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 model performance and time are critical aspects of an AI-enabled system as well. However, that is not all. Because AI-enabled systems are trained on existing data, any changes in the data cause model performance degradation over time [52]. This necessitates frequent retraining of a system within a reasonable amount of time. Research in less time-consuming training of AI-enabled systems has been an interesting topic for a while [38, 42, 46, 67]. 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 [4, 8]. This brings in the third most important factor which is the expense associated with building and running an AI-enabled system. In our investigation of instability, we therefore pay attention to these three factors as discussed below:

Model performance: This metric is determined from the model 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 [33], 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 (such as PyALCS [25]), 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 model performance-related metric.

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.

Expense: This metric is obtained from the amount of Travis CI credits spent on building and running the example script for 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 [7] states that the number of credits spent per minute on running a VM with Linux, Windows, and MacOS are respectively 10, 20, and 50. 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 [7], they are $1200 \times 0.0006 = 0.72$ and $6050 \times 0.0006 = 3.63$ for Linux and MacOS, respectively. As this example demonstrates, there can be scenarios where the difference in processing time between different settings is small and insignificant, however, the difference in cost can still be big and significant. This is why we study processing time and expense as two separate metrics in this study.

2.3.2 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 *model performance* of a project varies due to CPU architecture. In this case, we generate a distribution of *model performance* for the 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 *model 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`. As previously mentioned, 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. Once these two distributions are generated, we then perform the following steps:

Step 1: 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 the Equation 1 are defined as follows:

- $\overline{m_b}$ is the arithmetic mean of a metric (*model 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 (*model 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 instability in the considering metric.

The purpose of this step is to determine, on average across 50 runs, how much instability can be observed in each of the considering metrics.

Step 2: While *Step 1* of our analysis gives us an overall view of instability for each project, in this step, we aim to determine whether or not any observed instability is statistically significant. To determine the statistical significance of any observed instability, we first perform the Mann-Whitney U test [48] to compare the two distributions. We choose the Mann-Whitney U test as the test of statistical significance because this nonparametric test does not assume the data to be normal. We set the level of significance, $\alpha = 0.05$ for this test which represents the traditional 95% confidence level [16]. Next, we determine the degree of difference, also known as effect size, between the compared distributions using Cliff's delta [18]. 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 as mentioned in [40, 41]:

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

Note that we consider the observed instability between the generated distributions as statistically significant 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*.

Finally, we categorize the studied projects into three categories based on our two-step analysis previously described: (i) projects that show zero instability, (ii) projects that show non-zero instability which is statistically insignificant, and (iii) projects that show non-zero instability which is statistically significant. While any instability is undesirable, statistically significant instability is even more concerning.

3 RQ1: (Operating System) How much instability is introduced by changing the operating system in AI-enabled systems?

As our first research question, we study instability with respect to operating systems. We perform a comparative analysis among three operating systems: Linux, MacOS, and Windows. Furthermore, we also investigate whether instability 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.

3.1 Instability with respect to Operating System

Setup: To study the effect of operating systems on the instability in AI-enabled systems, we keep the CPU architecture and Python

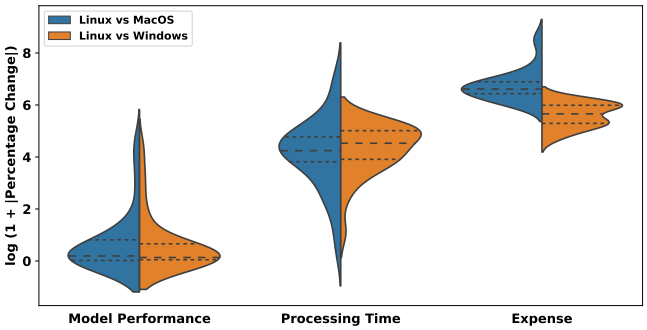

Figure 1: Distributions of instability with respect to Operating Systems.

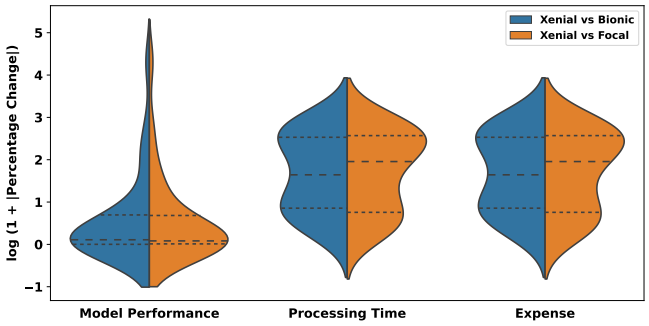

Figure 2: Distributions of instability with respect to Linux Distributions.

version constant to their baseline values and vary the choice of operating system only.

Findings: We find that the majority of the projects show instability across all three metrics due to operating systems. Figure 1 shows the distributions of percentage change (P) across all the studied projects. Table 3 reports the number of projects falling under different instability categories defined in Section 2.3. We observe that the majority of the projects show changes in all three metrics due to changes in operating systems. However, only a few projects show instability in model performance with statistical significance. On the other hand, almost all observed instability in processing time and expense is statistically significant. Paying closer attention to the breakdown of effect size for the projects with statistically significant instability, we find that in almost all cases the observed instability is large as shown in Table 4. We further find that there is a slight increase in model performance (1.44%) on average when projects are run on MacOS compared to Linux. On the other hand, model performance drops on average by 4.21% when projects are run on Windows. Although a slight increase in model 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.

Our findings imply that Linux is a faster and more cost-effective operating system than both MacOS and Windows although MacOS produces slightly better model performance.

3.2 Instability with respect to Linux Distribution

Setup: To study whether instability 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, Python version, and CPU architecture constant to their baseline values.

Findings: We find that, similar to operating systems, different Linux distributions also cause varying degrees of instability with respect to all three studied metrics. Figure 2 shows the distribution of percentage change (P) caused by changes in the Linux distribution across the studied projects. Table 5 reveals that the majority of the projects show some degree of instability between different distributions of Linux. Although none of the observed instability between Xenial and Bionic is statistically significant in any of the metrics, the observed instability between Xenial and Focal shows a different pattern. Between Xenial and Focal, three projects show a statistically significant instability in model performance whereas 23 projects show a statistically significant instability in processing time and expense. Most of the observed statistically significant instability is large in terms of effect size as shown in Table 6. On average a slight model performance gain of 2% can be achieved by choosing Focal over Xenial, however, this comes with a 7% increase in processing time and expense. This implies that newer versions of Linux have increased processing time and thus higher expense.

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

> **RQ1 Findings:** Most projects show significant instability in processing time and cost across operating systems, while only a few exhibit notable differences in model performance.

4 RQ2: (Python Version) How much does changing the Python version introduce instability in AI-enabled systems?

Setup: In this research question, we investigate if instability can be observed when different versions of Python are used to run the same system. We study the effect of Python version by keeping the operating system, distribution, and CPU architecture constant to their baseline values and only varying the Python version.

Findings: We find that most projects show some degree of instability between Python versions. Figure 3 shows a similar pattern to the observed instability in RQ1. That said, not all observed instability has statistical significance as shown in Table 7. Furthermore, Table 8 reveals that any instability observed 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 instability between Python 3.7 and Python 3.8.

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

Metric	Instability Type	Linux vs MacOS	Linux vs Windows
Model Performance	Zero instability	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 instability	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 instability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	0 (0%)	0 (0%)
	Non-zero and statistically significant	30 (100%)	30 (100%)

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

	Linux vs MacOS				Linux vs Windows			
	Small	Medium	Large	Total	Small	Medium	Large	Total
Model 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 instability types due to differences in Linux distributions.

Metric	Instability Type	Xenial vs Bionic	Xenial vs Focal
Model Performance	Zero instability	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 instability	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 instability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	7 (23.33%)
	Non-zero and statistically significant	0 (0%)	23 (76.67%)

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

	Xenial vs Bionic				Xenial vs Focal			
	Small	Medium	Large	Total	Small	Medium	Large	Total
Model 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

An even higher degree of instability can be observed in processing time and expense with a total of 24 projects showing significant instability 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 model performance, and choosing Python 3.8 over Python 3.7 causes a 0.73% drop in model 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.

Our findings show that the choice of Python version can cause instability, likely due to differences in library versions during installation. Newer Python versions may install updated libraries with added features, increasing installation time, while older versions may lead to performance drops and longer processing due to outdated dependencies.

RQ2 Findings: Python 3.6 and 3.7 produce identical results across all metrics in the studied projects. However, between Python 3.7 and 3.8, most projects experience significant instability in processing time and costs, with only a few showing instability in model performance.

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

Metric	Instability Type	Python 3.6 vs Python 3.7	Python 3.7 vs Python 3.8
Model Performance	Zero instability	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 instability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	6 (20%)
	Non-zero and statistically significant	0 (0%)	24 (80%)
Expense	Zero instability	0 (0%)	0 (0%)
	Non-zero but statistically insignificant	30 (100%)	6 (20%)
	Non-zero and statistically significant	0 (0%)	24 (80%)

Table 8: Breakdown of effect size for projects with statistically significant instability between different Python versions.

	Python 3.6 vs Python 3.7				Python 3.7 vs Python 3.8			
	Small	Medium	Large	Total	Small	Meidum	Large	Total
Model 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

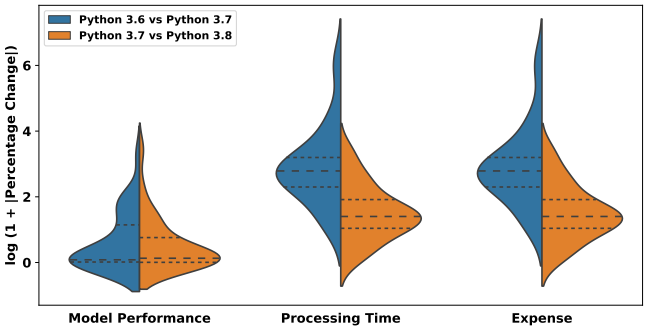

Figure 3: Distributions of instability with respect to Python versions.

5 RQ3: (CPU Architecture) How much does changing the CPU architecture introduce instability in AI-enabled systems?

Setup: We study the effect of CPU architecture on instability by keeping the operating system, distribution, and Python version constant to their baseline values and only varying CPU architecture configuration.

Findings: We find that CPU architectures also cause varying degrees of instability in all three metrics. Figure 4 summarizes the instability pattern in terms of percentage changes between AMD64 and ARM64 CPU architectures. Similar to the findings from RQ1 and RQ2, most of the observed instability in model performance is insignificant, whereas, in processing time and expense, the observed instability is significant in the majority of the studied projects. Table 10 shows that out of six projects with significant instability in model performance, 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 model performance costing 25% more time and money, on average. We conjecture that the observed instability between AMD64 and ARM64 CPU architectures may be due to design differences. ARM64 has a much smaller instruction set compared to AMD64 which might require ARM64 to take longer to perform more complex operations [13]. AMD64 being the most common CPU architecture has more software support compared to ARM64 CPUs. All these can negatively affect the model performance, processing time, and expense of building and running AI-enabled systems on ARM64 CPUs.

We can draw a similar conclusion for RQ3 to what we observed and concluded in RQ1 and RQ2. Instability in model performance is less common than instability 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 instability is large between AMD64 and ARM64 CPU architectures.

RQ3 Findings: CPU architecture has a major impact on processing time and costs in most projects, with a large effect size in these metrtics. While less common, the choice of CPU architecture can occasionally cause instability in model performance.

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

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

Table 10: Breakdown of effect size for projects with statistically significant instability between different CPU architectures.

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

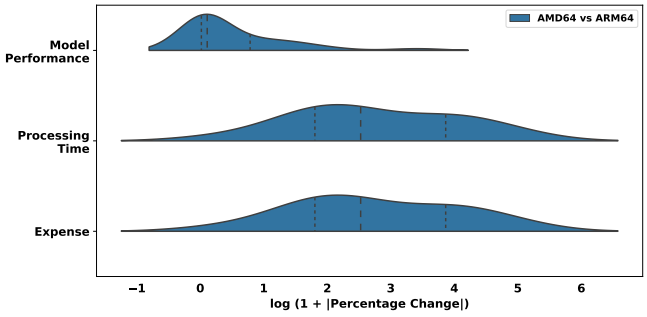

Figure 4: Distributions of instability with respect to CPU architectures.

6 Discussion

6.1 Interpretation

AI components are becoming a core part of almost all software systems nowadays. Our analysis shows that these systems suffer from instability in all three metrics we studied (*model performance*, *processing time*, and *expense*) although this finding is not consistent across all projects under investigation. Although existing work reported the existence of inconsistent and non-deterministic behavior of AI-enabled systems due to different factors such as choice of frameworks [32, 70], underspecification [20], and CPU multithreading [68], ours is the first work to investigate the effect of environment configurations on the systems' stability to the best of our knowledge. We find that the choice of operating system including the distribution of an operating system, version of Python, and CPU architecture indeed introduce instability. The degree of instability differs from project to project. Four projects, namely Seglearn [28], SMOTE-variants [63], pyGLMnet [55], and pymfe [26], consistently show stable behavior across all three studied metrics with respect to all the configuration settings under investigation. However, our qualitative analysis did not reveal any probable reason behind why this might be happening. Comparing these four projects against other 26 projects that show some degree of instability in at least one of the studied metrics did not show any distinct feature about them. While performing our qualitative analysis of the codebase of all projects, we did not notice anything different about the implementations of these four projects either, such as setting a value of the `random_state` setting internally within the implementations forcing an operation to repeatedly produce the same result. Therefore, we couldn't determine why the observed instability varies between projects without further investigation which is out of the scope of the paper and requires its dedicated study.

6.2 Implications

Firstly, our results imply that to be able to determine the existence and degree of instability in a project, developers must build and run the project under various configuration settings. Furthermore, our findings indicate that even within a project, not all metrics show an equal degree of instability. Instability is more prominent in processing time and associated costs than the model performance of an AI component. This can hurt a project's development lifecycle given that AI components like ML models need to be retrained frequently because they suffer from model performance decay due to data drift [52] and concept drift [45] 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 system and eventually can lead to a model performance drop. Moreover, a longer processing time usually implies higher costs.

Secondly, while instability in processing time and associated expenses impacts only the project internally (such as longer development time and an unnecessary increase in development efforts), the instability in model performance can impact the end-users of the project. This can potentially cause a financial burden for a company. The instability in model performance can be reduced by adhering to the practice of *dev/prod parity* which is an important principle of 12-factor apps [36, 66]. Therefore, our findings suggest that an AI-enabled system should be built and run on different configuration settings first so that the developers can determine

the environment configuration on which the most optimized system can be developed. This step should follow the *dev/prod* parity principle to ensure the stability of the system. Based on existing literature [37], it is not yet a common practice in the industry to build and run a system on multiple different configuration settings. 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.

7 Threats to Validity

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 to perform an analysis of the outputs. It is entirely possible that the example tasks may not represent the actual degree of instability associated with the project. 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 model performance of the project.

External Validity: Firstly, 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.

Secondly, we acknowledge that the dataset with 30 projects falls at the smaller end for quantitative experiments. As mentioned in Section 2.2, we started with a much bigger set of projects, however, the biggest requirement of our experimental design was that a project needed to successfully build and run on all configurations listed in Table 1. Only 30 projects met that requirement. That said, these 30 projects represent a diverse set of applications in terms of size, popularity, and activity as evident from Table 2. Furthermore, our manual analysis revealed that these projects represent the implementation of diverse ML algorithms and applications including natural language processing (such as Doc2Vec [23]), time-series analysis (such as Seglearn [28]), and recommender systems (such as Spotlight [27]).

8 Related Work

Nondeterminism 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.* [54, 70] studied the uncertain nature of training deep learning models. They reported that the choice of tools can affect the behavior of an AI component which can potentially affect AI safety. Guo *et al.* [32] performed an empirical study on the development and deployment of deep learning solutions. They reported that frameworks and platforms can cause the model performance of a system to decline. Crane [19] studied the challenges in the reproducibility of published results. This study reported that the consistent use of random seeds can help mitigate the issue of lack of reproducibility. Xiao *et al.* [68] reported the impact of CPU multithreading and how it impacts the training of deep learning systems.

Instability in Software: Instability in software systems has been explored in various contexts, including cloud infrastructure, system growth, and reproducibility. Some studies focused on detecting instability or proposing design practices to reduce it [21, 22]. Others found that instability can increase with codebase size [51], or that stable domain abstractions help maintain structural consistency [47]. Additionally, researchers have identified frequently modified system regions as unstable and proposed methods to prioritize them for restructuring [9, 11].

AI-components in Software: Many recent studies have investigated the pros and cons of having AI components embedded in software systems. Masuda *et al.* [50] described practices for the evaluation and improvement of the software quality of ML applications. Washizaki *et al.* [64] proposed architecture and design patterns for ML systems. An extensive study on testing ML applications was performed in [69] by Zhang *et al.*. Scully *et al.* [59] studied hidden technical debt in ML systems whereas Obrien *et al.* [53] 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 instability in AI-enabled systems in terms of model performance, processing time, and expense as a result of changes in configuration settings of three key environment variables (operating system, Python version, and CPU architecture).

9 Conclusion and Future Work

In this paper, we investigated how AI-enabled software systems show instability in terms of three metrics: model performance, processing time, and expense of building and running a system. We performed our study with respect to three environment variables, namely operating systems including the distributions of an operating system, Python version, and CPU architecture. Our findings indicate that although a majority of the projects show some degree of instability, the degrees vary from project to project. The instability is more statistically significant for processing time and expense than the model performance of an AI component. Because the observed instability patterns vary from project to project, we conclude that to serve the end users the most accurate AI solutions, it is crucial to run and test the AI components in different environment configurations. This practice can facilitate the identification of the environment where the **most optimized** system can be built which should follow adherence to the *dev/prod parity* principle to obtain the **most stable** system. We acknowledge that predicting instability without testing different configurations could save time and effort, making it a valuable topic for future research. Another potential study could explore the causes of observed instability, which we leave for future work.

References

[1] 2014. ISO/IEC 25000:2014 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE. https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-2:v1:en. [Accessed 09-01-2025].

[2] 2024. Desktop Operating System Market Share Worldwide | Statcounter Global Stats. https://gs.statcounter.com/os-market-share/desktop/worldwide. (Accessed on 06/07/2024).

[3] 2024. Domain-specific language. https://en.wikipedia.org/wiki/Domain-specific_language

[4] 2024. EC2 On-Demand Instance Pricing – Amazon Web Services. https://aws.amazon.com/ec2/pricing/on-demand/. (Accessed on 06/07/2024).

[5] 2024. Replication package: scripts and data. https://anonymous.4open.science/r/variability_replication_package-F175/

[6] 2024. Status of Python versions. https://devguide.python.org/versions/. (Accessed on 06/07/2024).

[7] 2024. Travis CI Documentation. https://docs.travis-ci.com/user/billing-overview/

[8] 2024. VM instance pricing | Compute Engine: Virtual Machines (VMs) | Google Cloud. https://cloud.google.com/compute/vm-instance-pricing. (Accessed on 06/07/2024).

[9] Bente CD Anda, Dag IK Sjøberg, and Audris Mockus. 2008. 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.

[10] Hrvoje Belani, Marin Vukovic, and Željka Car. 2019. Requirements engineering challenges in building AI-based complex systems. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*. IEEE, 252–255.

[11] Jennifer Bevan and E James Whitehead Jr. 2003. Identification of Software Instabilities.. In *WCRE*, Vol. 3. 134.

[12] Dileep Bhandarkar. 1997. RISC versus CISC: a tale of two chips. *ACM SIGARCH Computer Architecture News* 25, 1 (1997), 1–12.

[13] Dileep Bhandarkar and Douglas W Clark. 1991. 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*. 310–319.

[14] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. 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, 1–12.

[15] Justus Bogner, Roberto Verdecchia, and Ilias Gerostathopoulos. 2021. Characterizing technical debt and antipatterns in AI-based systems: A systematic mapping study. In *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 64–73.

[16] David Chavalarias, Joshua David Wallach, Alvin Ho Ting Li, and John PA Ioannidis. 2016. Evolution of reporting P values in the biomedical literature, 1990-2015. *Jama* 315, 11 (2016), 1141–1148.

[17] Xinghan Chen, Ling-Hong Hung, Robert Cordingly, and Wes Lloyd. 2023. X86 vs. ARM64: An Investigation of Factors Influencing Serverless Performance. In *Proceedings of the 9th International Workshop on Serverless Computing*. 7–12.

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

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

[20] Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. 2022. Underspecification presents challenges for credibility in modern machine learning. *Journal of Machine Learning Research* 23, 226 (2022), 1–61.

[21] Olga Vladimirovna Datskova. 2017. *Detection and Analysis of Operational Instability within Distributed Computing Environments*. Ph. D. Dissertation. University of Houston.

[22] Cornelia Davis. 2017. Realizing software reliability in the face of infrastructure instability. *IEEE Cloud Computing* 4, 5 (2017), 34–40.

[23] Doc2Vec Developers. 2018. Doc2Vec: Long(er) text representation and classification using Doc2Vec embeddings. https://github.com/ibrahimsharaf/doc2vec. [Accessed 22-01-2025].

[24] Fer Developers. 2024. FER: Facial Expression Recognition with a deep neural network as a PyPI package — github.com. https://github.com/JustinShenk/fer. [Accessed 27-05-2024].

[25] Pyalcs Developers. 2018. PyALCS: Anticipatory Learning Classifier Systems in Python. https://github.com/ParrotPrediction/pyalcs. [Accessed 30-12-2023].

[26] Pymfe Developers. 2019. Pymfe: Python Meta-Feature Extractor package. https://github.com/ealcobaca/pymfe. [Accessed 22-01-2025].

[27] Spotlight Developers. 2017. Spotlight: Deep recommender models using PyTorch. https://github.com/maciejkula/spotlight. [Accessed 22-01-2025].

[28] Seglearn Developers. 2018. Seglearn: Python module for machine learning time series. https://github.com/dmbee/seglearn. [Accessed 21-01-2025].

[29] Pedro Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM* 55, 10 (2012), 78–87.

[30] Michael Felderer and Rudolf Ramler. 2021. 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, 33–42.

[31] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. 2020. 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*. 431–442.

[32] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. 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, 810–822.

[33] Max Halford. 2018. StarBoost. https://github.com/MaxHalford/starboost. [Accessed 30-12-2023].

[34] Barbara Hammer and Thomas Villmann. 2007. How to process uncertainty in machine learning?. In *ESANN'2007 proceedings - European Symposium on Artificial Neural Networks, Bruges (Belgium), 25-27 April 2007*. 79–90.

[35] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 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*. 426–437.

[36] Kevin Hoffman. 2016. *Beyond the Twelve-Factor App*. O'Reilly Media, Inc.

[37] Meenu Mary John, Helena Holmström Olsson, and Jan Bosch. 2021. 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, 14–29.

[38] Akanksha Kavikondala, Vivek Muppalla, K Krishna Prakasha, and Vasundhara Acharya. 2019. Automated retraining of machine learning models. *International Journal of Innovative Technology and Exploring Engineering* 8, 12 (2019), 445–452.

[39] Jeremy Kedziora. 2024. Prediction Instability in Machine Learning Ensembles. *arXiv preprint arXiv:2407.03194* (2024).

[40] SayedHassan Khatoonabadi, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. 2023. 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.

[41] SayedHassan Khatoonabadi, Diego Elias Costa, Suhaib Mujahid, and Emad Shihab. 2023. Understanding the Helpfulness of Stale Bot for Pull-Based Development: An Empirical Study of 20 Large Open-Source Projects. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–43.

[42] Junyaup Kim and Simon S Woo. 2022. Efficient two-stage model retraining for machine unlearning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4361–4369.

[43] Michael Kläs and Anna Maria Vollmer. 2018. 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, 431–438.

[44] Benjamin Kompa, Jasper Snoek, and Andrew L Beam. 2021. Second opinion needed: communicating uncertainty in medical machine learning. *NPJ Digital Medicine* 4, 1 (2021), 4.

[45] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. Learning under concept drift: A review. *IEEE transactions on knowledge and data engineering* 31, 12 (2018), 2346–2363.

[46] Ananth Mahadevan and Michael Mathioudakis. 2024. Cost-aware retraining for machine learning. *Knowledge-Based Systems* 293 (2024), 111610.

[47] Sayyed G Maisikeli. 2018. Measuring architectural stability and instability in the evolution of software systems. In *2018 Fifth HCT Information Technology Trends (ITT)*. IEEE, 263–275.

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

[49] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software engineering for AI-based systems: a survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–59.

[50] Satoshi Masuda, Kohichi Ono, Toshiaki Yasue, and Nobuhiro Hosokawa. 2018. A survey of software quality for machine learning applications. In *2018 IEEE International conference on software testing, verification and validation workshops (ICSTW)*. IEEE, 279–284.

[51] Ali H Mresa and Abdussalam Nuri Baryun. 2021. Assessing Growth and Instability of Open Source Software Systems. In *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer*

Engineering MI-STA. IEEE, 398–406.

[52] Kevin Nelson, George Corbin, Mark Anania, Matthew Kovacs, Jeremy Tobias, and Misty Blowers. 2015. Evaluating model drift in machine learning algorithms. In *2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*. IEEE, 1–8.

[53] David OBrien, Sumon Biswas, Sayem Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. 2022. 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*. 734–746.

[54] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 771–783.

[55] pyGLMnet Developers. 2016. pyGLMnet: Python implementation of elastic-net regularized generalized linear models. https://github.com/glm-tools/pyglmnet. [Accessed 22-01-2025].

[56] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information* 11, 4 (2020), 193.

[57] Dhia Elhaq Rzig, Foyzul Hassan, Chetan Bansal, and Nachiappan Nagappan. 2022. Characterizing the Usage of CI Tools in ML Projects. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 69–79.

[58] Karthikeyan Sankaralingam, Jaikrishnan Menon, and Emily Blem. 2013. *A detailed analysis of contemporary arm and x86 architectures*. Technical Report.

[59] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).

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

[61] Sarvar Sultonov. 2023. IMPORTANCE OF PYTHON PROGRAMMING LANGUAGE IN MACHINE LEARNING. *International Bulletin of Engineering and Technology* 3, 9 (2023), 28–30.

[62] Cecilia Summers and Michael J Dinneen. 2021. Nondeterminism and instability in neural network optimization. In *International Conference on Machine Learning*. PMLR, 9913–9922.

[63] Smote variants Developers. 2018. Smote-variants: A collection of 85 minority oversampling techniques (SMOTE) for imbalanced learning with multi-class oversampling and model selection feature. https://github.com/analyticalmindsltd/smote_variants. [Accessed 21-01-2025].

[64] Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Studying software engineering patterns for designing machine learning systems. In *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 49–495.

[65] Jelte M Wicherts, Coosje LS Veldkamp, Hilde EM Augusteijn, Marjan Bakker, Robbie Van Aert, and Marcel ALM Van Assen. 2016. Degrees of freedom in planning, running, analyzing, and reporting psychological studies: A checklist to avoid p-hacking. *Frontiers in psychology* (2016), 1832.

[66] Adam Wiggins. 2017. *The twelve-factor app*.

[67] Yinjun Wu, Edgar Dobriban, and Susan Davidson. 2020. Deltagrad: Rapid retraining of machine learning models. In *International Conference on Machine Learning*. PMLR, 10355–10366.

[68] Guanping Xiao, Jun Liu, Zheng Zheng, and Yulei Sui. 2021. Nondeterministic Impact of CPU Multithreading on Training Deep Learning Systems.. In *ISSRE*. 557–568.

[69] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.

[70] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. 2022. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems* 4 (2022), 316–336.