Understanding the Nature of System-Related Issues in Machine Learning Frameworks: An Exploratory Study

Yang Ren
University of South Carolina
USA

Gregory Gay
Chalmers and the University of Gutenberg
Sweden

Christian Kästner
Carnegie Mellon University
USA

Pooyan Jamshidi
University of South Carolina
USA

ABSTRACT

Background: Modern systems are built using *development frameworks*. These frameworks have a major impact on how the resulting system executes, how configurations are managed, how it is tested, and how and where it is deployed. Machine learning (ML) frameworks and the systems developed using them differ greatly from traditional frameworks. Naturally, the issues that manifest in such frameworks may differ as well—as may the behavior of developers addressing those issues.

Aims: We are interested in characterizing the system-related issues—issues impacting performance, memory and resource usage, and other quality attributes—that emerge in ML frameworks, and how they differ from those in traditional frameworks.

Method: We have conducted a moderate-scale exploratory study analyzing real-world system-related issues from 10 popular machine learning frameworks.

Results: Our findings offer implications for the development of machine learning systems, including differences in the frequency of occurrence of certain issue types, observations regarding the impact of debate and time on issue correction, and differences in the specialization of developers.

Conclusions: We hope that this exploratory study will enable developers to improve their expectations, plan for risk, and allocate resources accordingly when making use of the tools provided by these frameworks to develop ML-based systems.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Open source model**; **Software evolution**.

KEYWORDS

Machine Learning Systems, Deep Learning Systems, Software Infrastructure, Empirical Study

Technical Report, May 2020

ACM ISBN AAA-B-CCCC-DDDD-E/FF/GG…$15.00
https://doi.org/10.1145/AAA.BBB

1 INTRODUCTION

A new paradigm of software systems have emerged, called machine learning (ML) systems [7, 16]. Traditional software systems consist of explicit instructions to a computer written by the programmer, whereas ML systems learn behavior from data. These systems are designed as a skeletal code architecture—specifying high-level behavioral goals—layered over highly-optimized models [16]. ML systems have revolutionized business intelligence, health care, finance, and other industries that power society.

Modern systems, both traditional and ML-based, are often powered by underlying *frameworks*—libraries of services that are used for providing higher-level functionality. TensorFlow, for example, offers a library for developing, training, or deploying models for use in ML systems. In traditional domains, we can look to examples like React or Flutter—frameworks for building user interfaces—or Rancher—a framework that provides container services.

Some argue that *best practices* for the development and quality assurance of traditional software systems are still largely based on ad-hoc experience, and often more closely represent an art than an established science [21]. ML systems and frameworks are so different that many of the lessons learned from traditional software development may no longer apply. ML systems differ in how they are developed, how they execute, how configurations are managed, how systems are tested, and how and where those systems are deployed. Naturally, then, the faults that developers create and the failures that manifest as a result may differ as well—as may how communities of developers behave in correcting those issues. We expect many differences and that those differences can be attributed to the underlying frameworks powering such systems. Therefore, we wish to better understand the types of issues that tend to occur in machine learning frameworks, and how they compare and contrast to the issues that impact frameworks in traditional paradigms.

Specifically, we are interested in characterizing and contrasting the types of *system-related issues* that emerge in the infrastructure code provided by ML frameworks (e.g., TensorFlow) versus frameworks for more traditional tasks (i.e., React). With system-related issues, we refer to issues affecting quality attributes, such as performance, configuration, component interaction, and memory usage. System-related issues are common in all types of systems [29], but it is not yet understood how they impact ML frameworks—and the systems built using these frameworks—and how the characteristics of such issues differ between system paradigms. Based on our experience in developing several ML and non-ML systems as well as systematic mining of issues in open source frameworks

in GitHub, we hypothesize that the system-related issues in ML frameworks differ from traditional software frameworks in three key areas: (1) *their types and frequencies of occurrence*, (2) *the difficulty of fixing particular issue types*, and (3) *the composition and behavior of the teams of developers engaged in fixing these issues*. A better understanding of system-related issues across domains will enable developers to improve their expectations, plan for risk, and better allocate resources.

In order to explore this topic, we have conducted an *exploratory study of bug reports* in ML and non-ML frameworks. Guided by a number of research questions, we compare data gathered from ML frameworks with data from traditional software frameworks. Specifically, we manually analyzed and classified **121** system-related issues from 10 popular ML frameworks and a further **332** system-related issues from 10 traditional software frameworks, collected from the GitHub project of each system. Furthermore, we approximate the difficulty of fixing an issue, including time, discussion, and patch size indicators, and examined the behavior of the community, as it relates to correcting issues—including the number of people involved in issue discussion and the level of activity involved in issue correction, among others. We explore the collected data for trends and differences. Our findings include a number of observations:

- Incorrect memory allocation, memory leaks, multi-threading errors, and performance regression occur more commonly in ML frameworks—possibly due to the need to manage large quantities of data in parallel and the rapid pace of system enhancement. Increased dependence on hardware selection, like the GPU, can also lead to issues. Configuration errors are very common in traditional software frameworks, but rarely occur in ML frameworks.
- Issues in ML do not appear to be significantly more difficult to address. API mismatch issues require significant time and discussion to fix, reflecting rapidly evolving communities debating how to best evolve their systems. Incorrect memory allocation issues also attract significantly more participants in discussing potential fixes. The most contentious issues reflect an evolving field and an active community. Memory leaks attract less participation at the pull request level, indicating an area where issue commonality leads to quick acceptance of solutions.
- Users of ML frameworks provide more detailed issue descriptions. Issues are not necessarily harder to reproduce. Instead, users may be more knowledgeable, have more development experience, and may be more prepared to offer background on the issue being reported than in traditional frameworks.
- Many ML frameworks developers identify as a combination of Engineer and Researcher, while many traditional framework developers identify solely as an Engineer. ML framework developers also tend to be more popular than the developers of traditional frameworks. There is little consistency in how long developers have had GitHub accounts.
- There is little we can say categorically about community activity level for ML versus software frameworks. ML issues do not attract significantly more non-developer users to take part in discussion than software frameworks. Overall, the two categories show similar levels of member participation. Software frameworks members contribute more to open source software. However, there are significant differences between individual systems.

Table 1: Issue categories and their definitions.

Category (Short Title)	Definition
API Mismatch (API)	Change to API version or mixed usage of APIs leading to performance degradation.
Compilation Error (Compl)	Failure to compile the source code.
Configuration Error (Config)	Configuration settings lead to performance degradation or error.
Connection Error (Conn)	Unexpected or wrongly-formatted connection request leads to error.
Data Race (Race)	Two or more threads access the same memory location concurrently.
Execution Error (Exec)	Unexpected error leads to the execution process crashing.
Hardware-Architecture Mismatch (HA)	Unfit hardware architecture leads to performance degradation or compilation error.
Memory Allocation (MA)	Memory allocation leads to performance degradation.
I/O Slowdown (I/O)	Issues with I/O processes lead to performance degradation.
Memory Leak (ML)	A failure in a program to release memory.
Model Conversion (Conv)	Performance degradation due to type conversion/cast.
Multi-Threading Error (MT)	Performance degradation due to thread interaction.
Performance Regression (PR)	Performance degradation after a change to the system.
Slow Synchronization (SYNC)	Synchronization between components leads to performance degradation.
Unexpected Resource Usage (RU)	Unusual system resource usage or requests leading to error or performance degradation.

To summarize, we make the following contributions:

- A moderate-scale exploratory study of system-related issues and their root causes as well as cross-comparison on ten widely used ML and traditional software frameworks.
- An in-depth analysis, characterization, and classification of **453** system-related issues and their related patches.
- A quantitative comparison of the difficulty of fixing issues, community behavior, and the issue fixing process.
- Actionable recommendations to the developers of ML frameworks, as well as systems that make use of these frameworks.
- A replication package[1] containing all data gathered in the process of performing this study. We hope that this exploratory study will offer assistance to the developers and researchers building ML systems and forming the best practices for ML-based fields.

2 SYSTEM-RELATED ISSUES

We define a *system-related issue* as a fault in the software that impacts quality attributes (non-functional properties) of the system, rather than functional issues, which result in the software producing the incorrect output. System-related issues tend to lead to performance degradation, loss of security, inappropriate usage of disc resources, or reduction in service [11]. System-related issues are significant, as they are a critical in determining system reliability and user experience. They are also useful in characterizing categories of systems, as unlike functional issues, system-related issued are not typically tied to system-specific requirements [3].

In this study, we have manually classified sampled issues into fifteen categories. Those categories are listed in Table 1. We derived these categories through manual coding [31]. More specifically, we used open coding to transform the initial structure into unstructured text by abstracting from large amounts of textual descriptions of issues and assigning codes to single textual description. One of the authors read the description of a new issue and if there exist

[1] https://doi.org/10.5281/zenodo.3786191

an existing code in the taxonomy, he will then assign it to the issue, otherwise, he would create a new code for the new issue. One of the other authors then review the codes and refine the name and check whether the assignment was done correctly. In case of disagreement, then they have discussed the details of each issue to come to a resolution by renaming the issue code, assigning to another category, or simply creating a new code for the issue.

These categories reflect the root causes of all of the sampled issues. To help illustrate the core concept, we present here examples of system-related issues in the studied ML frameworks:

Unexpected Resource Usage: A PyTorch user complained of "too many resources requested" errors shortly after the release of JetPack 3.2[2]. The developers found that the compiler lacked knowledge of how many threads the user wished to launch with, and the kernel was compiled to request more registers than is available on NVIDIA TX2. The patch added launch bounds that point out the maximum number of threads, so the kernel would not overuse registers.

Performance Regression: A Keras user reported that version 2.0.9 was extremely slow compared to 2.0.8[3]. For example, training a model went from 1-2 seconds to 10+ seconds, despite no environmental changes. After examining a variety of component interactions, the developers discovered the source of the slowdown—a method counting individual parameters. A simple check on the number of weights could be used without impacting functional correctness of the code, and without incurring slowdown.

Memory Leak: A TensorFlow user reported that a simple code fragment, creating a queue structure, would consume 10GB of memory[4]. A contributor found that the root cause was heap fragmentation, resulting from input being copied into new arrays on each step. This led to rapid changes to the memory heap, which were not handled well by `malloc`. The patch fixing this issue reduced the number of unnecessary array allocations by using a function that pulls values directly instead of copying them to a new array first.

API Mismatch: A TensorFlow user reported a crash following the use of a method from the `dataset` API on a dataset containing nested elements[5]. The issue was with a function used to group input by variable length. The API was updated to correctly unpack input with nested arguments.

Incorrect Memory Allocation: A MXNet user reported that they were unable to use multiple GPUs for model training, while a single GPU worked[6]. A contributor discovered that the issue was due to an inability to use pinned memory for those GPUs. The patch counts the number of GPUs and ensures that their pinned memory is used during training.

These examples illustrate how system-related issues affect ML frameworks, illustrating how different hardware configurations, memory and resource constraints, and limited testing of APIs can hinder the use of ML-based systems.

3 METHODOLOGY FOR ISSUE SAMPLING

To study differences in system-related issues between ML and traditional frameworks, we sample frameworks and their issues, classify them, and collect additional data.

[2]https://github.com/pytorch/pytorch/issues/7680
[3]https://github.com/keras-team/keras/issues/8381
[4]https://github.com/tensorflow/tensorflow/issues/2942
[5]https://github.com/tensorflow/tensorflow/issues/17932
[6]https://github.com/apache/incubator-mxnet/issues/7000

Table 2: The selected frameworks and indicators.

Framework	Domain	Watches	Stars	Forks	Commits	Contributors	Issues
TensorFlow	Machine Learning	8585	127514	74602	55724	1987	28
Torch7	Scientific Computing	664	8292	2331	1337	131	10
Caffe2	Deep Learning	559	8446	2130	3680	194	7
PyTorch	Machine Learning	1244	27999	6667	17915	1039	11
Theano	Scientific Computing	590	8786	2483	28080	332	10
OpenCV	Computer Vision	2444	34673	25213	26492	999	12
Keras	Deep Learning	2013	41115	15652	5110	794	10
Chainer	Deep Learning	321	4778	1263	26356	227	10
CNTK	Deep Learning	1386	16110	4267	16090	199	10
MxNet	Deep Learning	1173	16856	6017	9585	690	13
React	UI	6632	129558	23817	10955	1295	40
ETCD	Database	1268	24937	5051	15102	495	25
Flutter	Mobile	2269	64674	7241	14264	393	60
Rancher	Container	602	11549	1275	2686	57	70
iPython	Notebook	832	13576	3812	23811	592	24
Babel	Compiler	858	33145	3511	12405	726	33
AWS-CLI	Cloud	564	8029	1718	6963	197	11
Drone	DevOps	585	18344	1800	3436	241	9
OSQuery	Operating System	707	14187	1721	5005	264	10
Grafana	Log System	1212	28857	5410	21934	868	50

3.1 System Selection

In this study, we target frameworks—rather than individual systems—because the functionality offered by frameworks will be utilized by many systems, and will subsequently impact the behavior of such systems. Further, individual systems typically are developed by a smaller team of developers, have a smaller community of users, and will have fewer reported issues.

We selected ten open-source ML frameworks and ten open-source traditional software frameworks, as shown in Table 2. We selected ML frameworks based on their popularity and maturity. The popularity of each system in GitHub can be assessed from the number of stars of a repository [5]. We sought ML frameworks with a reasonable level of maturity. The selected frameworks have 1k-55k commits and more than a thousand forks. For contrast, we selected a matching set of 10 traditional (non-ML) frameworks that (a) come from a variety of different fields, and (b) are reasonably matched to the ML frameworks in terms of their activity and popularity. We devised a series of categories, and chose the most popular systems in each field and collect the indicators listed above for each system's repository. We normalized the collected values for each indicator. Then, we compared indicators for each traditional software framework with those for the ML frameworks. In line with propensity score matching [26], we choose one software framework from each category that was the most similar to one of the ML frameworks. For instance, React and TensorFlow are considered reasonably similar, as judged by the collected indicators. Table 2 lists all frameworks, values for the indicators used in pairing, and the number of issues sampled.

3.2 Sampling Issues

To understand the nature of system-related issues in ML frameworks, we need to collect enough data to investigate different types of issues. As we want to ensure sufficient information on each issue and how it was fixed, we focused on closed issues—those already fixed. Studying all system-related issues would be prohibitively costly, so instead we sampled issues from all studied systems.

In order to avoid selection bias, we *randomly* sample issues for each system to generate the sample set. We created a Python program based on the REST API provided by GitHub to randomly collect closed issues from each system's repository to generate the data set. In this data set, the data for each issue includes the issue title, the issue description, issue timeline, number of participants

in the issue discussion and the discussion of the corresponding pull request, and the number of comments on the issue and the corresponding pull request. We also collect information on the patch that fixes the issue, including the number of code lines changed and the number of files changed. To assess community behavior, we collect the number of participants for each issue that are members of the development team, the members' number of contributions to the project, and the number of contributions made by the creator of the issue-closing pull request.

After that, to avoid meaningless issues, we set inclusion and exclusion criteria to filter the data set. Issues are **excluded** if *there are less than three comments, a patch is not included, the issue status is open, the item is not an issue (i.e., a pull request in the issue list, a question, or enhancement suggestion)*, or *the issues has been closed due to lack of activity*. The issue is **included** if it is a system-related issue (not a functional issue), the patch is valid, and if the description has enough information to classify the issue type.

3.3 Determining Sample Size

We used power statistics to compute an appropriate sample size across all ML frameworks and all traditional software frameworks. At confidence level of 95 percent, we set the margin of error—how much we can expect our analysis result to reflect the view of the overall population—to 10% for ML frameworks and 5% for traditional software frameworks (because the traditional frameworks represent a diverse set of domains, and we, therefore, need more observations to understand them). We use the following formula to calculate the sample size [15]:

$$\frac{\frac{z^2 * p(1-p)}{e^2}}{1 + \frac{z^2 * p(1-p)}{e^2 * N}}, \tag{1}$$

where N is the population size, e is the margin of error and z is the z-score—the number of standard deviations a given proportion is away from the mean, resulting in 121 samples for ML frameworks and 332 samples for traditional software frameworks. Finally, we allocate the total sample size to each system in the group based on the percentage of the population of closed issues that belongs to each system. The formula for each system's sample size is $SS = GS * (\frac{SI}{GI})$, where SS is the system's sample size, GS is equal to the group sample size, SI represents the total number of closed issues for the system, and GI is the total number of closed issues for the group, resulting in the sample sizes listed in Table 2.

4 EXPLORATORY HYPOTHESES

Our research design is exploratory, but we guide our research using research questions and hypotheses (conjectures) shaped by personal experience in developing large-scale ML systems, interacting with ML developers in industry, and a literature and open source issue review of how ML systems differ from traditional software systems. We will explain our expectations and use them to guide our analysis.

RQ1: What differences can be seen in the types and distribution of issues in ML versus traditional frameworks?

This research question allows us to better understand whether particular types of issues are unique to ML frameworks, or differ in frequency of occurrence. This helps us understand whether system-related issues affect ML frameworks differently than traditional software frameworks, and what types of issues developers can expect to see. This allows better risk planning and allocation of resources. In this question, we examine a hypothesis about the distribution of issues.

H1: There are categories of system-related issues that occur more frequently or uniquely in ML frameworks, and categories that occur more frequently or uniquely in traditional software frameworks.

ML-based systems differ in many aspects from traditional software, and proper execution relies on choosing a model, training it, tuning parameters, and correctly executing prediction processes. We suspect that ML frameworks will suffer from system-related issues that occur rarely, if at all, in traditional software. Likewise, certain issues in traditional frameworks may occur rarely or be irrelevant to ML frameworks.

RQ2: Are system-related issues more difficult to fix in ML frameworks than in traditional frameworks?

In some ways, the development of ML frameworks is more complex and less mature than traditional software frameworks. This, in turn, may affect the difficulty of addressing system-related issues. In this question, we examine two hypotheses about the difficulty of issue correction and reproduction.

H2: There are categories of system-related issues in ML frameworks that are more difficult to fix than in traditional software frameworks.

ML frameworks have a large volume of input data, complicated algorithms, are built on complex models, and require configuration. These characteristics may impact the difficulty of fixing issues. We also wish to understand whether differences in difficulty are categorical—ML vs traditional frameworks—or dependent on framework-specific factors. For example, TensorFlow supports multiple GPUs, while Theano is bound to a single GPU by default.

H3: System-related issues are easier to reproduce in traditional frameworks than in ML frameworks. Issue reports in ML frameworks require that the reporter offer additional information in order to reproduce and debug the issue.

Issues in ML frameworks may arise from a more diverse pool of configurations, hardware platforms, and deployment environments than in traditional frameworks. To reproduce and debug issues, developers may require additional information from the reporter.

RQ3: Are there differences in how communities behave when identifying and fixing system-related issues between ML frameworks and traditional frameworks?

We investigated the behavior of the open-source communities building the studied frameworks. To answer RQ3, we investigate three hypotheses about community behavior, examining participant specialization and experience (H4), discussion activity (H5), and the impact of activity level on the issue-fixing process (H6).

Table 3: Percentage and number of issues in each category. Bolded cells (in all tables) show significantly differing distribution between groups (P-Value < 0.05, One-Way ANOVA).

Category	ML	Traditional
API Mismatch (API)	13% (16)	15% (56)
Configuration Error (Config)	**2% (2)**	**41% (148)**
Compilation Error (Compl)	2% (2)	0% (0)
Connection Error (Conn)	0% (0)	1% (4)
Data Race (Race)	1% (1)	0% (0)
Execution Error (Exec)	1% (1)	0% (0)
Hardware-Architecture Mismatch (HA)	1% (1)	0% (0)
Incorrect Memory Allocation (MA)	**5% (6)**	**2% (8)**
I/O Slowdown (I/O)	9% (11)	5% (17)
Memory Leak (ML)	**30% (36)**	**14% (50)**
Model/Data Conversion (Conv)	1% (1)	0% (0)
Multi-Threading Error (MT)	**13% (16)**	**4% (13)**
Performance Regression (PR)	**20% (24)**	**12% (42)**
Slow Synchronization (SYNC)	3% (4)	7% (24)

H4: The participants in issue discussion in ML frameworks are more experienced, are more specialized in their knowledge, and attract more popularity than participants in discussions in traditional frameworks.

As ML frameworks incorporate complex algorithms, the developers of such systems require appropriate specialization in their expertise. Solving system-related issues requires a deep understanding of the complex underlying algorithms (e.g., distributed training). This means that active developers of such systems may be more senior than developers of generic systems and may have certain specific areas of expertise. Because ML represents a new paradigm, users may—in turn—pay more attention to the developers of the systems and their contributions to ML frameworks.

H5: Discussion of system-related issues attracts a greater number of non-developer users in ML frameworks.

As ML frameworks are currently attracting a lot of attention, there exists the possibility that issues discussion also attracts a higher level of participation from users who are not part of the development team. Issues may affect a greater number of users, who in turn may experience the same or similar issues in a greater variety of contexts.

H6: ML frameworks require a more active developer community than traditional software to fix system-related issues.

The complexity of underlying ML algorithms, increase in need for specialized knowledge, and variety of deployment environments for ML may, in turn, require a more active community of developers in order to address system-related issues. More developers may need to take part in discussion, contribute to the project, and make pull requests in order to maintain a healthy, functioning system.

5 RQ1 — ISSUE CHARACTERIZATION

Our first research questions asks about the differences in the types and distribution of issues between ML and traditional software frameworks. More specifically, hypothesis 1 speculates whether the exist statistically significant differences in types and distribution of issues. Table 3 show the percentage of the total issue pool and raw number of issues for each category for the two paradigms. Figure 1 shows boxplots of the percentage of issues belonging to each category for each system in the two paradigms. For clarity, only common categories are shown.

From Figure 1, we can see that there are a number of issue types that occur quite often in both paradigms—including memory leaks,

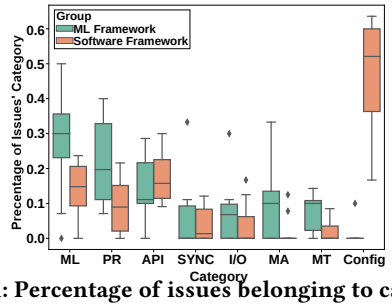

Figure 1: Percentage of issues belonging to categories.

performance regression, API mismatch, slow synchronization, and multi-threading errors. However, there are categories for which there are obvious differences in frequency. Configuration errors are quite common for traditional frameworks, making up 41% of the total issue pool, with a median of close to 50% on a per-system basis. They are vanishingly rare for ML frameworks, only making up 2% of the total pool. On a per-system basis, incorrect memory allocation makes up a median of 10% of the issue pool for ML frameworks, and is rarer for traditional frameworks.

To more clearly understand the major areas of difference between the system paradigms, we used a one-way analysis of variance (ANOVA) to compare the distributions of fault types between groups of systems. In Table 3, we bold the categories where significant difference was shown between the groups with p-value < 0.05. We find that configuration errors occur significantly more often in traditional frameworks. Systems of both categories are "configured", in the sense that their execution depends on can vary depending on certain adjustable factors. However, in traditional software, configuration tends to be explicit, based on providing values in a file or through the command line. For example, a user of AWS-CLI reported an issue that occurs when a space character appears in a provided profile name[7]. Such issues are more rare in ML frameworks, where a user rarely directly adjusts values in a file. In ML frameworks, "configuration" tends to be more implicit, where—for example—behavior varies based on a chosen hardware platform or training data. This leads to other issues, as we will discuss, but reduced the potential for explicit configuration issues.

We also find that incorrect memory allocation, memory leaks, multi-threading errors, and performance regressions occur significantly more often in ML frameworks. ML systems must process, manage, and make decisions using massive sets of data. Such algorithms must be multi-threaded, in order to rapidly process subsets of the dataset in parallel [9]. Likewise, the volume of data and need to store and access it efficiently requires careful management of memory. As a result, threading and memory errors will likely occur more often. Our observations bear this out. The field of ML evolves rapidly, and the popularity of such systems has led to an ever-expanding userbase. The need for rapid evolution may also explain the increased frequency of performance regressions.

In our random sample, there were several types of issues uniquely observed in ML—as can be seen in Table 3. These include data races, execution errors, hardware-architecture mismatch, model/-data conversion, and unexpected resource usage. None of these types were common, and most of these can—without doubt—occur

[7] https://github.com/aws/aws-cli/issues/2806

in traditional frameworks as well. However, several of these issue categories closely relate to important facets of ML systems, and may occur more commonly as a result. For example, hardware-architecture mismatch can occur because of the variety of hardware configurations being used in ML. Many ML platforms can use GPUs for efficient data processing, particularly using NVIDIA's CUDA platform [27]. As an example, an issue encountered in this study, from OpenCV, occurred because the system did not support the version of CUDA used by the GPU in the user's configuration[8].

Summary: Incorrect memory allocation, memory leaks, multi-threading errors, and performance regression occur more commonly in ML frameworks—likely due to the need to manage large quantities of data in memory and the rapid pace of system enhancement. Increased dependence on hardware selection, like the GPU, can also lead to issues. Configuration errors are very common in traditional frameworks, but rarely occur in ML, as such frameworks tend to offer fewer explicit user-defined configuration options.

6 RQ2 — ISSUE DIFFICULTY

Our second research question asks whether system-related issues are more difficult to fix in ML frameworks than in traditional software frameworks. We focus on the six categories of issues with a reasonable number of samples for both traditional software and ML frameworks: memory leaks, performance regressions, API mismatch, I/O slowdown, incorrect memory allocation, synchronization, and multi-threading errors. We guide our analysis using two exploratory hypotheses: (H2) that there are categories of issues that are more difficult to fix in ML, and (H3), that issue reporters must provide additional information to developers of ML frameworks.

6.1 H2—Issues are More Difficult to Fix

Hypothesis 2 speculates that certain categories of issues are more difficult to fix in ML than in software frameworks. We gathered seven indicators that, together, present an approximation of the effort required to fix an issue. These indicators include the number of days between issue creation and closure, the number of comments on the issue report, the number of participants in the issue report discussion, the number of comments on the pull request closing the issue, the number of participants involved in discussion of the pull request, the number of lines of code changes in the patch fixing the issue, and the number of files changed. Table 4 lists the median values for each indicator for six issue types and a summary across all types of issues. We provide details about the distribution of these indicators in Figures 2, 3, and 4.

From Table 4, we can see that—overall—issues seem to take slightly longer to be fixed in ML frameworks, with a median of 11 days versus 8 days. They also tend to require slightly larger patches (26 LOC versus 23). However, neither of these indicators show a significant difference according to the ANOVA test, and many of the other indicators—comments on the report, participants in the PR, and number of files changed—have the same median. Therefore, there is little we can conclude about issue difficulty overall. *ML framework issues, overall, are not more difficult to fix.* It is, however, worth looking more deeply at individual categories of issues. API mismatch issues do take longer to fix in ML, taking a median of 57 days (compared to 7.5 days in traditional frameworks) and demonstrating significant difference in the ANOVA test. Likewise, there tends to be more discussion on the issue report, from more participants. From this, we can speculate that API issues may not actually be more difficult to fix in terms of traditional code changes. Rather, they may be more difficult because the APIs themselves are evolving rapidly following debate in an active, opinionated community. The long median time to fix, and the larger number of comments on issue reports, suggest that API mismatch issues require debate and community deliberation to determine if they are, in fact, actual problems or misuse of the framework. Multiple sampled issues show debate between contributors before consensus is reached on whether there is an issue[9] [10] [11]. Once developers agree that there is a bug, changes to the API—which have the potential to affect a large number of users—require further debate.

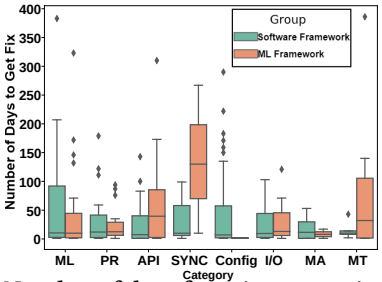

Figure 2: Number of days from issue creation to close.

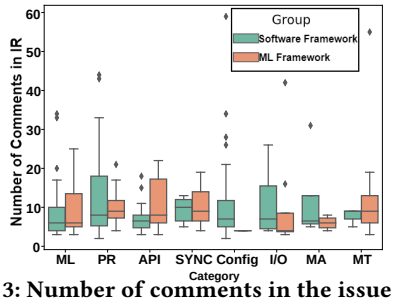

Figure 3: Number of comments in the issue report.

This is also suggested in Figures 2 and 3, where there is a large variance in ML frameworks for number of days and number of comments. This variance suggests some contention in the discussion of API mismatch issues. By contrast, Figure 4 shows less variance for ML than traditional frameworks in terms of the number of comments in the pull request. By the time a pull request is filed, it tends to be rapidly accepted.

Incorrect memory allocation issues are more common in ML frameworks, but do not necessarily appear to be more difficult to solve. However, pull requests fixing such issues seem to attract some debate, with significantly more participants involved at the pull request level (confirmed by ANOVA).

Memory leaks are fixed in approximately the same amount of time, with the same quantity of issue discussion. In fact, despite

[8] https://github.com/opencv/opencv/issues/7375

[9] https://github.com/torch/torch7/issues/281

[10] https://github.com/opencv/opencv/issues/6081

[11] https://github.com/tensorflow/tensorflow/issues/25882

Table 4: Median values for each collected data regarding issue difficulty for Hypothesis 2.

Category	Days		Comments-Rep		Comments—PR		Participants—Rep		Participants—PR		LOC		Files	
	ML	Trad	ML	Trad	ML	Trad	ML	Trad	ML	Trad	ML	Trad	ML	Trad
API Mismatch (API)	**47.0**	**7.5**	**9.0**	**7.0**	1.0	3.0	**4.0**	**3.0**	2.0	2.5	24.0	36.0	2.0	2.0
Incorrect Memory Allocation (MA)	8.0	11.5	6.0	6.5	1.5	0.0	3.0	2.5	**3.0**	**0.0**	8.0	23.5	1.5	2.5
I/O Slowdown (I/O)	13.0	7.0	4.0	7.0	1.0	5.0	3.5	3.0	3.5	3.0	30.5	58.0	1.0	2.0
Memory Leak (ML)	10.0	10.0	6.0	6.0	0.0	6.0	5.0	3.0	**2.0**	**4.0**	28.0	34.0	2.0	3.0
Multi-Threading Error (MT)	32.0	10.0	9.0	9.0	6.0	1.0	3.0	2.0	3.0	1.0	45.0	16.0	2.0	1.0
Performance Regression (PR)	12.5	12.0	9.0	8.0	6.0	3.0	4.5	4.0	5.0	2.0	25.5	37.5	2.0	2.0
Overall	11.0	8.0	7.0	7.0	1.0	3.0	4.0	3.0	3.0	3.0	26.0	23.0	2.0	2.0

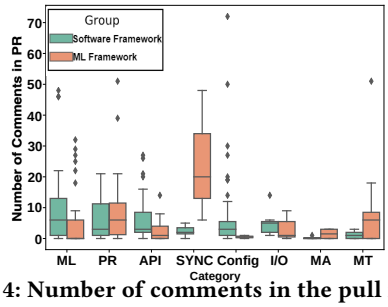

Figure 4: Number of comments in the pull request.

Table 5: Median values for Hypothesis 3.

Category	Comments		Words		Files Attached		Code Attached	
	ML	Trad	ML	Trad	ML	Trad	ML	Trad
API	2	1	127	104.5	**0.5**	**0**	21	14
MA	0.5	2	167	70	0.5	1	0	7
I/O	0.5	1	136.5	90	0	0.5	4.5	1.5
ML	0	1	**131**	**110**	0	0	18	17
MT	2	2	143	115	2	1	36	1.5
PR	0	1	186	124	1	0	0	12
Overall	2	1	**113**	**93.5**	0	0	21	12.5

happening more frequently in ML frameworks, memory leaks may be slightly *easier* to fix, with significantly fewer participants in the pull request. Given increased frequency of memory leaks, developers may have a more immediate understanding of how to solve such issues using automated tools, and the fixes for such issues may be accepted with little need for community debate.

Summary: Broadly, issues in ML do not appear to be significantly more difficult to address. API mismatch issues require significant time and discussion to fix, reflecting rapidly evolving communities debating how to best evolve their systems. Incorrect memory allocation issues also attract significantly more participants in discussing potential fixes. The most contentious issues reflect an evolving field and an active community. Memory leaks attract less participation at the pull request level, indicating an area where issue commonality leads to quick acceptance of solutions.

6.2 H3—Information Quantity

Our third guiding hypothesis states that more information will be required for developers to reproduce reported system-related issues. We hypothesize this for multiple reasons. ML systems are often somewhat stochastic in nature, behavior is often influenced by subtle environmental factors, and understanding an issue may require specialized understanding of the underlying statistical algorithms. Therefore, we suspect that the reporting user may need to provide detailed information on both the issue and their deployment environment. This could, in turn, contribute to the difficulty of correcting an issue.

We measure several indicators of the information content that a user must provide. These indicators include the number of comments in the discussion thread before the issue is reproduced. This is determined manually for each sampled issue. We also collect the number of words in the issue description, the number of files attached to the issue report, and the number of lines of code attached to the issue report. Table 5 lists the median values for each indicator both for the six issue types with a reasonable number of samples for both system categories and over the full pool of issues.

Overall, as shown in Table 5, ML frameworks require a higher median number of comments before issues are reproduced (2 to 1), number of lines of code attached to the report (21 to 12.5), and number of words in the issue report (113 to 93.5). However, of those, only the number of words shows statistical significance—as demonstrated using the ANOVA test. Therefore, an increase in the amount of information that a user has to provide primarily manifests in terms of the number of words in the issue description. *Users of ML systems provide detailed descriptions of issues to the development community.* This does not necessarily suggest that issues are harder to reproduce or solve in ML, but may instead suggest that the users of such systems are knowledgeable, have more development experience, and may be prepared to offer more background on the issue being reported than the average issue reporter in a software system.

Memory leaks, in particular, require a significantly larger number of words in the issue description. Memory leak issues are not necessarily harder to reproduce, but do require that the user provide a detailed account. This may not reflect the *difficulty* of fixing memory leaks, but rather that the increased frequency of memory leaks in ML better prepares users to report such problems. It is possible that the descriptive initial bug reports help ease acceptance of the pull request, as indicated in the previous section.

API mismatch also requires a significantly higher number of attached files with the issue description. This further suggests that API mismatch issues are difficult to address, and can require debate in the development community—for instance, requiring a higher median number of comments before being confirmed as an issue. The variance between systems is low in terms of the number of words in the description, suggesting along with the higher median that users—up front—provide more information on these issue in ML frameworks.

The remaining categories offer little in the way of clear trends. While medians may differ in various ways, the differences are not significant according to the one-way ANOVA tests.

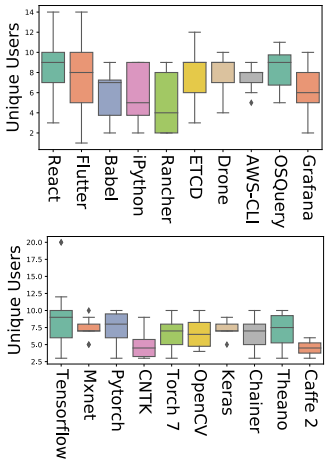

Figure 5: The participants account history.

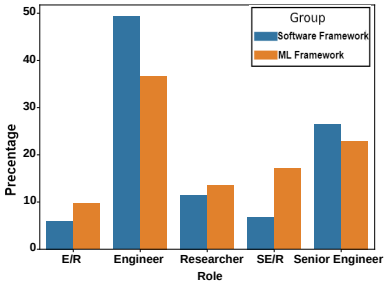

Figure 6: The real-world role of participants.

Summary: Users of ML frameworks provide more detailed issue descriptions. Issues are not necessarily harder to reproduce. Instead, users may be more knowledgeable, have more development experience, and may be more prepared to offer background on the issue being reported than in traditional frameworks.

7 RQ3 — COMMUNITY BEHAVIOR

Our third research question revolves around the behavior of the open-source communities building the studied systems. To answer RQ3, we investigate three hypotheses, examining participant specialization and experience (H4), discussion activity (H5), and the impact of the community activity level on the issue-fixing process (H6). Rather than discussing particular issue types in this question, we focus on the differences between systems.

7.1 H4—Participant Experience

Our fourth hypothesis states that the participants in issue discussion in ML frameworks are more experienced and are more specialized in their experience. We measure three indicators for this hypothesis—the number of years that participants in issue discussion have owned their GitHub account, the role of the participants, and the number of followers participants have.

The number of years a user has owned their account partially indicates their development experience. We did not identify any statistically significant results. Our results, shown in Figure 5, indicate quite a bit of variance between systems.

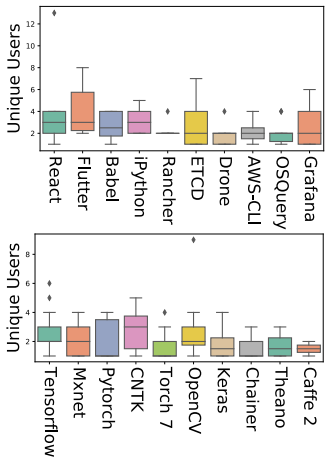

Figure 7: Number of unique users involved in discussion.

Our second indicator revolves around the job role of the participants in issue discussion. For this indicator, we manually gathered the job title for each participant from their GitHub profile or linked personal and LinkedIn pages. Figure 6 shows the results of this process. The categories we assigned include Engineer, Researcher, Senior Engineer, and combinations of (Senior) Engineer and Researcher. We were able to identify the role of more participants in ML. This potentially suggests that the participants in such projects tend to have a more distinct online identity that ties open-source development into their work role, or where their work role explicitly involves open-source development. The clearest difference we see between ML and traditional frameworks, as shown in Figure 6, is the increased importance of the role of "Researcher" in the participant pool. More participants work as a combination of Engineer and Researcher in ML than in traditional frameworks.

Our third indicator is the number of followers that participants in the issue fixing process have on GitHub. This indicates user popularity. We have removed 13 outliers, who had more than 40,000 followers. Overall, ML participants have a higher median number of followers—62 to 54. ANOVA confirms statistical significance.

Summary: Many ML frameworks developers identify as a combination of Engineer and Researcher, while many traditional framework developers identify solely as an Engineer. ML framework developers also tend to be more popular than the developers of traditional frameworks. There is little consistency in how long developers have had GitHub accounts.

7.2 H5—Non-Developer Users

Our fifth hypothesis states that the discussion of issues attracts a greater number of non-developer users in ML than in software systems. As such systems are currently quite popular, they may have a more active and varied discussion community. To measure the quantity, we collect the number of users from the participant list for each issue. We then omit any users that are listed members of the project development team (who have "write" access). An ANOVA test fails to indicate significant differences between the two paradigms. Figure 7 shows that the number of users that take part in discussion varies quite a bit between frameworks.

Table 6: Community behavior data.

System	# Members	% Participate	# Contributions	# PR Contributions
TensorFlow	2	33.33%	184	387
Torch7	1	33.33%	2503	974
Caffe2	0	0%	0	448
PyTorch	1	25.00%	1465	787
Theano	2	58.33%	1114	653
OpenCV	0	0%	0	480
Keras	0.5	16.67%	211	211
Chainer	3	75.00%	1465	1558
CNTK	0.5	12.5%	0	92
MxNet	1	14.29%	241	208
Overall	1	33.33%	366	344
React	1	25.00%	704	704
ETCD	1	33.33%	704	1954
Flutter	1	33.33%	483	809
Rancher	3	66.67%	133	133
iPython	2	66.67%	1649	1695
Babel	2	42.22%	496	356
AWS-CLI	1	25.00%	598	633
Drone	1	25.00%	68	57
OSQuery	1	33.33%	254	254
Grafana	2	33.33%	1965	1965
Overall	1	33.33%	754	704

Summary: ML issues do not attract significantly more non-developer users to take part in discussion than software frameworks. There is a large amount of variance in the makeup of discussion participants between systems.

7.3 H6—Activity Level

Our final hypothesis is that ML frameworks require a more active developer community than traditional frameworks in order to fix system-related issues. We speculate that more developers may need to take part in discussion, contribute to the project, and make pull requests in order to maintain a healthy, functioning system.

To measure the activity level, we focus on *members* of each project. The members are people who are part of the organization that owns a project, and that have "write" access to the project repository. This list is made available as part of a GitHub project[12]. We collect three indicators, including: (1) the percentage of members that take part in issue discussion, (2) the number of contributions made by a member during the year that each issue was reported, and (3) the number of contributions (commits, pull requests) made by the issue's corresponding pull request creator during the year that each pull request was created.

Table 6 shows the median values for each indicator for each system that we studied. Immediately, we see quite a bit of variance between systems in ML. Compared to traditional projects, ML projects vary wildly in terms of the percent of members that participate in issue discussion—from 12.50% to 75.00% of members taking part in discussions. In comparison, traditional frameworks show a narrower range of percentages, with medians of 25-66.67% of members taking part in issue discussion. Overall, however, the median percentage of members taking part in issue discussion in ML and traditional frameworks are quite similar.

There is quite a bit of variance in the number of contributions made by project members. Members of the PyTorch and Chainer communities contribute quite a lot each year, while members of the TensorFlow community contribute very little in comparison, possibly due to a larger community. There is quite a bit of variance for traditional frameworks as well. Overall, members of traditional projects make more contributions on a yearly basis.

Again, there is significant variance between individual systems in terms of the number of contributions made by the issue-fixing pull request creator during the year that each pull request was created. Compare with ML, pull request creators in traditional frameworks contribute more overall.

Summary: There is little we can say categorically about community activity level for ML versus software frameworks. Overall, the two categories show similar levels of member participation. Software frameworks members contribute more to open source software. However, there are significant differences between individual systems.

8 THREATS TO VALIDITY

Internal Validity: First, our study involves manual inspections on system-related issues in machine learning frameworks. These subjective steps can be biased due to interpretation of intent based on limited code comments and issue description. In order to reduce this threat, one author analyzed the issues separately and discussed inconsistent issues with a second author until an agreement was reached. Second, our study investigated 453 issues from Github for 10 machine learning frameworks and 10 traditional systems. It is not clear how much our findings can or will generalize beyond our dataset, especially considering the fact that machine learning systems are evolving rapidly. However, it is not easy to expand this dataset. First, the manual efforts required to analyze the issues were large. We could automate the labeling process, but it would then introduce noise in how we categorise issues. To collect and analyze the issues, we spent approximately 960 person-hours, leading to an average 2.11 person-hours per issue. However, we believe that this process lead to stable conclusions for this exploratory analysis.

External Validity: The main threat to the external validity is generalisation beyond the considered frameworks and selected issues. We selected the frameworks based on their popularity. To make the issue taxonomy as comprehensive as possible, we labeled a large number of issues from GitHub until we reached saturation of the categories. Since both ML and traditional frameworks are rapidly changing, the observations and relative numbers may change for each corresponding category. However, due to a large sample set, we believe that it is unlikely that the answers to the research questions would be impacted by sampling additional systems.

9 ACTIONABLE RECOMMENDATIONS

Based on our findings, we can make several recommendations to the developers of ML frameworks, as well as systems that make use of these frameworks. First, our results indicate that incorrect memory allocation, memory leaks, multi-threading errors, and performance regression occur more commonly in ML frameworks. Increased dependence on hardware selection, like the GPU, can also lead to issues. Developers should plan for handling these types of issues. It would be reasonable to actively recruit or advertise for developers

[12] https://github.com/orgs/tensorflow/people

who specialize in areas such as memory management, concurrency, or software product lines. Recruitment of developers with expertise in these topics could lead to better development, and faster response when an issue occurs.

We found that many ML frameworks developers identify as researchers or some combination of engineer and researcher. This has both positive and negative implications. Researchers have specialized knowledge in their area of focus. This can be utilized to great benefit in developing ML frameworks. By taking advantage of this expertise, frameworks can deliver sophisticated, highly effective features. At the same time, it is important that overall development of a framework can proceed without losing sight of the "big picture". Developers should not focus solely on their own areas of expertise and ignore features outside of their focus area. The overall architecture of a framework, as well as its usability, are extremely important and require consensus and conversation across the team as a whole. It is important that the development community of a project crafts compatible API designs, coding standards, and testing standards that are followed across the project, and that developers have some knowledge of how their work influences the system as a whole.

We also found that the users of ML frameworks tend to provide more detailed issue descriptions than those of traditional systems, perhaps reflecting the complex, specialized nature of such systems. This can be good, as more information can help developers reproduce and correct issues more easily. However, more text does not necessarily imply a greater quantity of useful information. It is important that users be given structure and guidance when reporting issues. ML framework developers should make use of issue report templates to ensure that important information is provided by reporters. TensorFlow and PyTorch communities are using templates for reporting the issues. Past experience can be quite useful in helping users file reasonable reports. Detailed issue reports, filed for past issues, can be used to provide examples to users filing new issue reports. Well-crafted issue reports should be retained and pointed to in order to help ensure that relevant details are included in new reports.

Finally, we found that some issues such as API mismatches or incorrect memory allocation required more time, more discussion, and a greater number of involved users to come to a conclusion on whether there was an issue or how to fix it. The most contentious issue types reflect an evolving field and an active community. This is not necessarily a negative finding. In fact, it can be quite positive—a healthy culture where developers share ideas, debate the merits of them, and come to a consensus on a solution will often lead to rapid, sustainable improvement to a framework. Development communities should encourage and expect debate. This requires, however, the creation of moderation standards within a community to keep discussion on-target and civil.

10 RELATED WORK

Others have tried to investigate the differences between the two system paradigms from various perspectives. A previous empirical study analyzed issue reports for three open source ML systems including Apache Mahout, Lucene, and OpenNLP [32]. Programs bugs developed in TensorFlow have also been studied empirically [44]. However, these studies focused on particular frameworks (e.g., TensorFlow) and collected all types of program issues (not necessarily systems-related issues) to the extent that they concluded "the small number of performance inefficiency issues suggests either performance issues rarely occur or these issues are difficult to detect." We mainly focus on system-related issues, and we found that performance regressions are actually common symptoms in machine learning systems. While many prior studies exist on understanding the nature of system-related issues in the traditional software stack [1, 4, 12, 20, 38, 40], our study explores a wider range of systems and issues, and offers a detailed comparison of machine learning with traditional systems.

The findings of studies on traditional software systems may not apply in ML for multiple reasons, including the fact that in the ML stack, programming is done differently than in the traditional software stack. For example, when the network fails in a handful of rare cases in ML, we do not correct those predictions by correcting the code. Rather, those predictions are fixed by including more labeled examples of those rare cases in order to regularize the learning process [16].

Differences between the two paradigms have been investigated from the perspective of software engineering practices as well. For example, a case study at Microsoft [2] details differences of developing in the AI domain versus traditional application domains, and how team processes and practices change. They identified three distinguishable aspects of ML: (i) data accumulating, massaging and cleaning is much more complex, (ii) model customization require very different skill sets, and (iii) components are more difficult to handle as distinct modules. The testing process is also different in ML [8, 10, 22–25, 30, 30, 33, 35, 35, 43]. Prior studies have also identified unique technical debt concerns for machine learning systems [6, 28].

Many prior studies have examined performance-related issues in traditional software [13, 17, 19, 34, 39, 40, 42]. Each of these has informed our issue classification process. Configuration-related issues are also a significant concern in our research. A number of studies have been conducted on performance-related issues in software, systems, and cloud that informed our approach [14, 18, 36, 37, 41].

11 CONCLUSION

Frameworks offer services that can be used to build software. Issues in frameworks will impact the software built using those frameworks. ML systems differ from traditional systems in how they execute, how configurations are managed, how systems are tested, and how and where they are deployed. Naturally, the issues that manifest will differ as well—as will how communities of developers behave in correcting those issues. We have conducted a moderate-scale study contrasting the differences in the system-related issues between popular ML and traditional frameworks. Our findings offer a number of interesting observations, with implications for the development of ML frameworks and systems that make use of these frameworks. We hope that this exploratory study as well as the recommendations will offer assistance to the "machine learning systems" community forming the best practices for this new paradigm.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 421–432.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 291–300.

[3] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

[4] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 133–143.

[5] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Predicting the popularity of GitHub repositories. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 9.

[6] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. 2017. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. In *Proceedings of IEEE Big Data*.

[7] Michael Carbin. 2019. Overparameterization: A Connection Between Software 1.0 and Software 2.0. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[8] Dawei Cheng, Chun Cao, Chang Xu, and Xiaoxing Ma. 2018. Manifesting bugs in machine learning code: An explorative study with mutation testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 313–324.

[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[10] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 118–128.

[11] Brendan Gregg. 2013. *Systems performance: enterprise and the cloud*. Pearson Education.

[12] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[13] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 17–28.

[14] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin Van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L Abad, et al. 2018. Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)* 3, 4 (2018), 1–36.

[15] Prashant Kadam and Supriya Bhalerao. 2010. Sample size calculation. *International journal of Ayurveda research* 1, 1 (2010), 55.

[16] Andrej Karpathy. 2017. *Software 2.0*. https://medium.com/@karpathy/software-2-0-a64152b37c35

[17] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. 2010. Finding latent performance bugs in systems implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 17–26.

[18] Philipp Leitner and Jürgen Cito. 2016. Patterns in the chaosâĂŤa study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)* 16, 3 (2016), 1–23.

[19] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1013–1024.

[20] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 329–339.

[21] S. McConnell. 1998. The art, science, and engineering of software development. *IEEE Software* 15, 1 (Jan 1998), 120–119. https://doi.org/10.1109/52.646892

[22] Christian Murphy, Gail E Kaiser, and Marta Arias. 2007. An approach to software testing of machine learning applications. (2007).

[23] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 785–796.

[24] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[25] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1027–1038.

[26] Paul R Rosenbaum and Donald B Rubin. 1983. The central role of the propensity score in observational studies for causal effects. *Biometrika* 70, 1 (1983), 41–55.

[27] Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming* (1st ed.). Addison-Wesley Professional.

[28] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

[29] 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. In *Advances in neural information processing systems*. 2503–2511.

[30] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-implementation testing of supervised learning software. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

[31] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*. 120–131.

[32] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.

[33] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. ACM, 303–314.

[34] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 154–168.

[35] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558.

[36] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 307–319.

[37] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 619–634.

[38] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. 2017. How do system administrators resolve access-denied issues in the real world?. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 348–361.

[39] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 1299–1308.

[40] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. How not to structure your database-backed web applications: a study of performance bugs in the wild. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 800–810.

[41] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 159–172.

[42] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2012. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 199–208.

[43] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine Learning Testing: Survey, Landscapes and Horizons. *arXiv preprint arXiv:1906.10742* (2019).

[44] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 129–140.

A APPENDIX

A.1 Patch Code for Section 2

Resource Usage (Pytorch 7680 - Patch)

```
@@ -7,6 +7,7 @@
aten/src/THCUNN/im2col.h
template <typename Dtype>
+ __launch_bounds__(CUDA_NUM_THREADS)
__global__ void im2col_kernel(const int n,
            const Dtype* data_im,
            const int height, const int width,
            const int ksize_h, const int ksize_w,

@@ -58,6 +59,7 @@ void im2col(cudaStream_t
stream, const Dtype* data_im, const int chan-
nels, _consistency(self):
aten/src/THCUNN/im2col.h
template <typename Dtype, typename Acctype>
+ __launch_bounds__(CUDA_NUM_THREADS)
```

Performance Regression (Keras 8381 - Patch)

```
@@ -22,7 +22,6 @@
keras/engine/training.py
from .. import metrics as metrics_module
from ..utils.generic_utils import Progbar
- from ..utils.layer_utils import count_params

@@ -967,8 +966,8 @@ def _check_trainable_weights
_consistency(self):
keras/engine/training.py
- if (count_params(self.trainable_weights) !=
-     count_params(self._collected_trainable_
-     weights)):
+ if (len(self.trainable_weights) !=
+     len(self._collected_trainable_weights)):
 warnings.warn(UserWarning(
      'Discrepancy between trainable weights
      and collected trainable'
      ' weights, did you set `model.trainable`
      without calling'
```

Memory Leak (Tensorflow 2942 - Patch)

```
@@ -616,7 +616,7 @@ def _feed_fn:
' to a larger type (e.g. int64).')
- np_val = np.array(subfeed_val,
-                   dtype=subfeed_dtype)
+ np_val = np.asarray(subfeed_val,
+                   dtype=subfeed_dtype)
```

API (Tensorflow 17932 - Patch)

```
tensorflow/contrib/data/python/ops/grouping.py
@@ -140,9 +140,9 @@ def bucket_by_sequence_length:
' Try explicitly setting the type of the feed tensor'
' to a larger type (e.g. int64).')
- def element_to_bucket_id(element):
+ def element_to_bucket_id(*args):
- seq_length = element_length_func(element)
+ seq_length = element_length_func(*args)
boundaries = list(bucket_boundaries)
buckets_min = [np.iinfo(np.int32).min]
            + boundaries
```

Memory Allocation (Incubator-Mxnet 7000 - Patch)

```
tensorflow/contrib/data/python/ops/grouping.py
@@ -140,9 +140,9 @@
#if MXNET_USE_CUDA
+ CUDA_CALL(cudaGetDeviceCount(&num_gpu_device));
+ CHECK_GT(num_gpu_device, 0) <<
+          "GPU usage requires at least 1 GPU";
  ptr = new storage::GPUPooledStorageManager();
#else
  LOG(FATAL) << "Compile with USE_CUDA=1 to
                enable GPU usage";
```