Challenges in Migrating Imperative Deep Learning Programs to Graph Execution: An Empirical Study

Tatiana Castro Vélez  
City University of New York (CUNY) Graduate Center  
New York, NY, USA  
tcastrovelez@gradcenter.cuny.edu

Mehdi Bagherzadeh  
Oakland University  
Rochester, MI, USA  
mbagherzadeh@oakland.edu

Raffi Khatchadourian  
City University of New York (CUNY) Hunter College  
New York, NY, USA  
raffi.khatchadourian@hunter.cuny.edu

Anita Raja  
City University of New York (CUNY) Hunter College  
New York, NY, USA  
anita.raja@hunter.cuny.edu

ABSTRACT

Efficiency is essential to support responsiveness w.r.t. ever-growing datasets, especially for Deep Learning (DL) systems. DL frameworks have traditionally embraced deferred execution-style DL code that supports symbolic, graph-based Deep Neural Network (DNN) computation. While scalable, such development tends to produce DL code that is error-prone, non-intuitive, and difficult to debug. Consequently, more natural, less error-prone imperative DL frameworks encouraging eager execution have emerged at the expense of run-time performance. While hybrid approaches aim for the “best of both worlds,” the challenges in applying them in the real world are largely unknown. We conduct a data-driven analysis of challenges—and resultant bugs—involved in writing reliable yet performant imperative DL code by studying 250 open-source projects, consisting of 19.7 MLOC, along with 470 and 446 manually examined code patches and bug reports, respectively. The results indicate that hybridization: (i) is prone to API misuse, (ii) can result in performance degradation—the opposite of its intention, and (iii) has limited application due to execution mode incompatibility. We put forth several recommendations, best practices, and anti-patterns for effectively hybridizing imperative DL code, potentially benefiting DL practitioners, API designers, tool developers, and educators.

CCS CONCEPTS

• General and reference → Empirical studies; • Computing methodologies → Machine learning; • Software and its engineering → Language features; Software evolution.

KEYWORDS

empirical studies, deep learning, imperative programs, hybrid programming paradigms, graph-based execution, software evolution

1 INTRODUCTION

Machine Learning (ML), including Deep Learning (DL), systems are pervasive in society. Central to such systems are dynamic models, whose behavior is ultimately defined by input data. However, as datasets grow, efficiency becomes essential to support responsiveness [103]. For industrial applications, DL frameworks—pillars of DL systems [56,58,68,100]—must quickly execute complex computations on large datasets while supporting easy-to-use programming paradigms [60]. For efficiency, DL frameworks have traditionally embraced a deferred execution-style that supports symbolic, graph-based Deep Neural Network (DNN) computation [25,46]. While scalable, development is error-prone, cumbersome, and produces programs that are difficult to debug [56,57,99,100]. Furthermore, because graph computation executes statements in a non-imperative order, traditional Software Engineering (SE) tools cannot help troubleshoot bugs [9]. Contrarily, more natural, less error-prone, and easier-to-debug imperative DL frameworks [3,27,79] encouraging eager execution have emerged. Though ubiquitous, eagerly-executed imperative DL programs are less efficient and scalable as their deferred-execution counterparts [25,37,43,60,72,79]. Executing (imperative) DL programs eagerly “makes tensor [matrix-like data structures central to DL] evaluation trivial but at the cost of lower performance” [30].¹ Thus, hybrid approaches [6,37,72]—integrated into mainstream DL frameworks—execute imperative DL programs as static graphs at run-time. For example, in TensorFlow [1]—a popular [54,100] DL framework—AutoGraph [72] can potentially enhance performance by decorating (annotating)—with optional yet influential decorator arguments—appropriate Python function(s) with @tf.function. Decorating functions with such hybridization Application Programming Interfaces (APIs) can increase imperative DL code performance without explicit modification. Though promising, hybrid approaches necessitate non-trivial specialized metadata [60] and exhibit limitations and known issues [42] with native program constructs. Subtle considerations are required to make code amenable to safe, accurate, and efficient graph execution—avoiding performance bottlenecks and semantically inequivalent results. Therefore, developers are burdened with making their code compatible with the underlying execution model conversion, as well manually specifying which functions should be converted. While alternatives [60] exist, they impose custom Python interpreters, which may be impractical for industry, and support only specific Python constructs. Thus, there is a knowledge gap in how hybridization is used in real-world DL applications, leading to the challenges in successfully applying it underexplored. Without such insight, DL systems may be inefficient, fallible, and difficult to maintain. Moreover, advances in DL are likely to be futile if they cannot be effectively used.

¹Performance is this paper refers to run-time performance (speed), not model accuracy.
we aim to answer the following research questions: (RQ1) what bug patterns and corresponding challenges are involved in writing reliable yet performant imperative DL code, and (RQ2) which best practices and anti-patterns can be extracted from (RQ1)? Such knowledge can help drive new automated migration techniques, IDE code completion, and automated (data science-specific [11,32,33]) refactoring mining approaches [95]. The results: (i) advance knowledge of this emerging yet pervasive hybrid paradigm, (ii) provide feedback to language and API designers for future API versions, (iii) help tool designers comprehend difficulties with writing performant imperative DL code, (iv) include preliminary recommendations, best practices, and anti-patterns for practitioners in using hybridization effectively, and (v) assist educators in teaching hybridization APIs.

Our study involves analyzing occurrences of tf.function in 250 projects, consisting of 19.7 MLOC, along with 470 and 446 manually examined code patches (Git commits) and bug reports (GitHub issues), respectively. Challenges—along with their causes, symptoms, and fix patterns—are taxonomized using manual processes aided by automated software repository mining. Due to its popularity and extensive analysis by previous work [26,55,56,58,68,74,99,100], we focus on hybridization in TensorFlow. Our study indicates that: (i) tf.function is widely used, (ii) misusing tf.function was a major theme in migrating imperative DL programs to graph execution, (iii) subtle bugs in using tf.function can result in performance degradation—the opposite of its intention, and (iv) tf.function is commonly incompatible in a given context—limiting its application.

Our contributions can be summarized as follows:

**Hybridization bug hierarchical taxonomy** From 470 and 446 patches and bug reports, respectively, of 250 projects manually examined, we build a rich hierarchical taxonomy of common hybridization challenges.

**Recommendations, best practices, & anti-patterns** We propose preliminary recommendations, best practices, and anti-patterns for effectively hybridizing imperative DL code from our statistical results, as well as an in-depth analysis. Complete results of our study are available in our dataset [24].

## 2 MOTIVATING EXAMPLES & BACKGROUND

Popular DL frameworks have historically embraced deferred execution-style (low-level) APIs, making DNNs straight-forward to execute as symbolic graphs that enable various run-time optimizations. For example, during graph building (lines 2–4 of Listing 1), line 4 does not execute until the Session created on line 6 is run on line 8. While efficient, legacy code using such APIs are cumbersome, error-prone, and difficult to debug and maintain [56,57,99,100]. Such APIs also do not natively support common imperative program constructs, e.g., iteration [5]. Contrarily, eager execution-style DL APIs [3,79] facilitate higher-level, imperative, and Object-Oriented (OO) [27] (Python) programs that are easier-to-debug, less error-prone, and more extensible have emerged. For instance, with eager execution, line 4 of listing 1 would execute and immediately evaluate tensor c, foregoing the need of a session. In many DL frameworks, eager execution is now the default. Despite the benefits, executing (imperative) DL programs eagerly comes at the cost of run-time performance [30]. Thus, hybridization approaches [6,37,72] that execute imperative DL programs as graphs at run-time have been integrated into mainstream DL frameworks. For example, listing 2 portrays TensorFlow imperative (OO) DL code representing a modestly-sized model for classifying images. On line 11, `AutoGraph` is used to potentially improve performance by decorating the model’s call() method with `@tf.function`, possibly providing optional yet influential decorator arguments. At run-time, call()’s execution will be “traced” and an equivalent graph will be generated [42]. In this case, a speedup (runtime\_old/runtime\_new) of ∼9.22, averaged over five runs, ensues [63].

As noted in Section 1, while promising, hybridization presents unique challenges [42,60] in ensuring that programs run reliably and efficiently. If used incorrectly, hybridization may yield programs that result in unexpected run-time behavior. Decorating the right functions, supplying the correct decorator arguments, using the appropriate API, and properly structuring imperative DL code so that it is amenable to graph execution can be daunting, especially for developers (data scientists) lacking SE expertise.

**Python Side-effects**. Side-effect producing, native Python statements, e.g., printing, list appending, global variable mutation [42], are problematic for tf.function-decorated functions. Because they are traced, a function’s behavior is “etched” into its corresponding graph and thus can have unexpected results, executing side-effects multiple times or not at all. Side-effects occur when tf.functions are called the first time; subsequent calls with similar arguments execute the graph instead. For example, on line 3 of listing 3, f() outputs x. On line 1, f() is decorated with @tf.function, which migrates it to a graph at run-time. Then, f() is invoked three times, the first two with the argument 1 and the last with 2. In the output on the right, the first invocation of f() on line 4 results in a graph being built (through tracing) that—due to a similar argument—is
Challenges in Migrating Imperative Deep Learning Programs to Graph Execution

Listing 4: Imperative TensorFlow code using a counter [42].

```
1 class Model(tf.Module):
2   def __init__(self):
3     self.v = tf.Variable(0)
4     self.counter = 0
5     @tf.function
6     def __call__(self):
7         if self.counter == 0:
8             self.counter += 1
9         self.v.assign_add(1)
10        return self.v

Output (expecting 1, 1, 1):
1 1
2 2
3 3
```

Listing 5: DL model (listing 2) client code using varying datasets [42].

```
1 model = SequentialModel()
2 res1 = model(tf.constant([[1, 2, 3]]))
3 res2 = model(tf.constant([[1, 2, 3, 4, 5]]))
```

later used on line 5. Consequently, the side-effecting code on line 3 is not exercised. In contrast, line 3 is exercised as a result of the call on line 6 due to a different argument being supplied.

Although listing 3 is simple, unexpected behavior can generally be difficult to notice. Consider listing 4, where a model uses a counter to safeguard a variable incrementation. The initial value of counter, however, is captured during tracing upon the first model invocation (line 14). The overall effect is that the value of v is incremented unconditionally (line 10) each time the model is invoked. Such problems are common in migrating deferred-execution-style DL code (e.g., listing 1) to an imperative style (e.g., listing 2). Worse yet, developers only realize such errors after observing suspicious numerical results or significantly lower performance than expected (e.g., when guarded operations are costly) [42].

**When To Use Hybridization?** Besides ensuring that DL code is amenable hybridization [36], developers must also know when and where to use it to avoid performance bottlenecks and other undesired behavior. For instance, confusion exists on how often `@tf.function` should be applied [87], and calling `tf.functions` recursively could cause infinite loops [42]. Even if a recursion seems to work, the `tf.function` will be traced multiple times (“retracing”), potentially impacting performance. Also, using `@tf.function` on small computations can be dominated by graph creation overhead [43].

**Using Hybridization Parameters.** Decorating the correct function but with incorrect decorator arguments may result in performance degradation. For instance, retracing helps ensure that the correct graphs are generated for each set of inputs; however, excessive retracing may cause code to run more slowly had `tf.function` not been used [42,80,81]. Listing 5 depicts code that invokes the model declared in listing 2 multiple times using different (hypothetical) datasets, producing the warning on the right. To limit retracing, an `input_signature` can be specified on line 11, listing 2 as follows:

```
@tf.function(input_signature=([tf.TensorSpec(shape=()), dtype=tf.int32]))
```

A `None` dimension in the `tf.TensorSpec` allows for flexibility in trace (graph) reuse. Since tensors are matched on their shape, a `None` wild card allows `tf.functions` to reuse traces for variably-sized input—occurring when sequences or images are of different lengths or sizes, respectively. Since each call no longer produces a trace, the warning disappears—averting any performance bottlenecks.

| Table 1: Studied subjects. |
|-----------------------------|
| sub | KLOC | studied periods | commits/iss | kws | exe |
|-----|------|----------------|-------------|-----|-----|
| fixes | 122 | 10,879 | 2015-11-06 to 2021-01-14 | 199,140 | 470 | 470 |
| reports | 167 | 17,378 | 2012-05-07 to 2021-08-11 | 237,232 | 704 | 446 |
| Total | 250 | 19,677 | 2012-05-07 to 2021-08-11 | 436,372 | 1,174 | 916 |

Represents unique totals due to subject overlap between the study portions.

These simplified examples demonstrate that effectively using hybridization is not always straight-forward, potentially requiring complex analyses and a thorough understanding of API intricacies—a compounding problem in more extensive programs. As imperative DL programming becomes more widespread, statistical insight into how such programs are best written efficiently and how to avoid common bugs would be extremely valuable to developers.

### 3 METHODOLOGY

**Subjects.** We examined Git commit changesets (code patches; row fixes, Table 1) representing bug fixes involving `tf.function` and GitHub issues (row reports) mentioning `tf.function`. Our study encompassed 250 open-source DL systems (column subj), comprising ~19.7 million lines of source code (column KLOC), 199,140 Git commits (column commits for commits), 237,232 GitHub issues (column iss for bug reports), and 460.21 years of combined project history, averaging 1.86 years per subject. Subject details may be found in our dataset [24]; subjects sources are publicly available on GitHub. While we focus `tf.function` client usages, we include TensorFlow as developers often file GitHub issues against it to discuss `tf.function` usage challenges and potential bugs. Subjects include those used in previous studies [26,32,55,56,58,59,68,99,100] and appearing in data science-specimen datasets [17]. To determine if a project represents a DL system, i.e., one with at least one DL module, we searched repositories for specific keywords, e.g., “keras,” “layer,” “net,” “neural network,” “deep learning.” The keywords have also been used in related work [59] for a similar purpose; the keywords were only used to ensure that subjects were DL systems, not for finding hybridization bugs. We then verified the code to ensure that the keywords represented DL contexts.

For changesets (bug fixes), subject criteria consists of having at least one commit whose changeset contains `tf.function`. For issues, subjects must have at least one GitHub issue mentioning “tf.function.” Subjects were mostly written in Python, which is popular for DL [16]. While the subjects include popular open-source repositories from well-known and reputable organizations, e.g., Apache [7], Apple [8], Google [45], NVIDIA [75], they also include lesser-known repositories to understand hybridization challenges facing the DL community-at-large. Furthermore, hybridization is relatively new—`tf.function` was released on September 30, 2019.

**Mining.** To find changesets (patches) representing hybridization bug fixes, we mined repositories for commits referencing `tf.function` using gitcproc [23], a tool for classifying Git commits used by previous work [12,65,92,94]. Row fixes, column kws of Table 1 is the commits containing `tf.function` in their changesets. We manually examined all 470 commits, portrayed by row fixes, column exe. To find issues related to hybridization, we mined repositories for GitHub issues mentioning “tf.function” by first filtering out
issues containing only irrelevant discussion (e.g., “social conversation”) using a pre-trained classification model [10] used by previous work [76, 98, 102]. We then invoked the GitHub Search API [41] to select (open and closed) issues that included “tf.function” using several different criteria, e.g., “best match,” “most commented.” To reduce false positives, since the API ignores punctuation, we further filtered the results to ensure that they included the period. Row **reports**, column **kws** of Table 1 is the issues containing “tf.function” in either their title or body (description and conversations). We randomly selected a subset of these to examine manually (details below), portrayed by row **reports**, column **exe**. The aforementioned tools [10, 23, 41] were only used to narrow the search space and not for classification, which was done manually. The GitHub search was performed in a (standard) manor consistent with previous work.

**Identification.** We used a gitcproc feature that leverages heuristics based on log messages to identify bug fix commits. Natural language processing (NLP) is internally used by gitcproc to determine the commits that fall into this category. Doing so helps us to focus on likely bug fix commits for further manual examination. Random matching issues—with ones containing code being favored—were chosen for manual inspection. Next, the authors manually examined the commits and issues to ascertain if they indeed relate to hybridization bugs. Two authors are SE and PL professors with extensive expertise in software evolution, system performance, and empirical SE. Another author is a data mining and ML professor with substantial proficiency in AI and SE. Three authors have several years of industrial SE experience.

Although the researchers did not converse during the initial identification and classification process to avoid bias, this mix of expertise is effective in studying SE tasks in DL systems. The researchers convened regularly during the study, as well as at the end for finalization, to solidify the results. Cohen’s Kappa coefficients [96] for identification and classification were 0.80 and 0.57, respectively. As the authors did not always have detailed knowledge of the particular systems, only changes where a bug fix was extremely likely were marked as such. The authors also used commit comments and referenced bug databases to ascertain whether a change was a bug fix. GitHub issues tags were also considered.

**Classification.** For commits, once bug fixes were identified, the authors studied the code changes to determine the category of bug fixes and whether the category relates to hybridization. For issues, the authors examined issue descriptions and discussions, paying attention to the *tf.function* challenges being described and their possible solutions and workarounds. Particular attention was paid to code snippets. No scripts were involved in the classification—only manual examination. Categories were then formed into a hierarchy, in part by using the TensorFlow documentation [42]. On several occasions, developers were contacted for clarification using the GitHub line comment mechanism and via email.

4 RESULTS

This section answers (RQ1) by summarizing our results, noting trends, exceptions, and unexpected outcomes. Contrarily, Section 5 consolidates, comments on, and connects the main findings. Related discussion in Section 5 is referenced where appropriate.

4.1 Quantitative Analysis

From the 470 commits and 446 GitHub issues (totaling 916) manually examined (column **exe**, Table 1), we found 157 and 123 (totaling 280) *tf.function* bug fixes and developer challenges depicted in columns **cmts** (commits) and **iss** (GitHub issues) of Table 2, respectively. Finding these bugs and understanding their relevance required a significant amount of manual labor that may not be feasible in more large-scale, automated studies. Python, being a dynamic language, can be difficult to analyze, particularly w.r.t. inheritance relationships; subclassing Keras models is a common way to write imperative DL code in TensorFlow (cf. line 1, listing 2). Furthermore, our number of findings (280) is comparable with previous studies involving manual inspection (e.g., Tang et al. [91] found 285, Zhang et al. [100] found 175, Khatchadourian et al. [65] found 61). Nevertheless, as *tf.function* becomes more popular, we expect its usage and number of related bugs to grow.

4.1.1 Problem Categories. We group bug fixes and GitHub issues into common problem categories, shown in Fig. 1 and Table 2 (column **abbr** is the category abbreviation). The former includes combined data (commits and issues), while the latter separates the two. Figure 1 presents a hierarchical categorization—with varying levels of detail—of the 280 discovered *tf.function*-related challenges in our subjects. Challenges are represented by their problem category name and are followed by their counts. Categories without instances are abstract, i.e., they only group together other categories. Table 2 portrays a nonhierarchical, top-level view of Fig. 1; the innermost (top) layers of Fig. 1 represent the rows of Table 2.

Challenges are grouped into several (top-level) problem categories. Categories include performance (PRF, 90; further discussed later), API misuse (APM, 25; further discussed later), and incompatibility between execution modes, i.e., eager and deferred, where *tf.function* is used in a context not amenable to graph conversion (INC, 48; further discussed later). An example of the latter is where particular loss functions cannot be used in graph mode or there is an AutoGraph limitation that prevents graph conversion.

Other problem categories include dealing with or working around open bugs related to *tf.function* in TensorFlow (TFB, 20; further discussed later) and “other” (OTH, 16), which involves syntactic corrections,
Challenges in Migrating Imperative Deep Learning Programs to Graph Execution

Figure 1: Discovered problem categories (hierarchical).

Table 3: Performance fixes.

| fix category                              | count |
|-------------------------------------------|-------|
| Add tf.function decorator                 | 61    |
| Change tf.function argument               | 20    |
| Add input_signature argument to tf.function | 9    |
| Remove tf.function decorator              | 8     |
| Upgrade to new library version            | 4     |
| Relocate tf.function (use on different function) | 5    |
| Re-add tf.function decorator              | 2     |
| Unsolved (open)                           | 2     |
| Total                                     | 111   |

Numerical errors (NME, 1) involve possible numeric overflow. Autograph compilation errors (CMP, 1) surface when tf.functions are compiled and subsequently result in compilation errors. This problem may arise when certain dynamic Python features, e.g., lexical scoping, are utilized (cf. Section 4.2.2). Segmentation fault (SEG, 1) is when using tf.function causes a program crash. While compilation and numerical errors and segmentation faults may be considered symptoms, we focus on tf.function usage; these categories represent problems from a client perspective. Their underlying causes are bugs within the framework.

Performance. As the main purpose to hybridization is to improve general cleanup, and refactorings—a category similar to that used by previous work [65,94]. "Unknown" (UKN, 10) represents situations where the problem category was indeterminable without further domain knowledge or developer input. Only 3.57% of problems had unknown categories. Code changes involving tf.function appearing in tests were categorized as “Test” (TST, 8).

Debuggability. Debuggability (DBG, 6) represent situations where using tf.function to improve performance of DL code may, in turn, reduce a developer’s ability to easily debug it. “In general, debugging code is easier in eager mode than inside tf.function” [42]. In such situations, developers may not understand that using tf.function is the reason why they are not able to debug their code, e.g., intermediate variable values may be missing. Or, tf.function may temporarily be removed (via a commit) to facilitate debugging, but developers inadvertently neglect to replace it (cf. Section 4.2.5). This latter situation is unfortunate as, to assist in the debugging process, a flag can be used to globally (temporarily) toggle tf.function [42].

Other Categories. Other (top-level) categories were more minor in terms of their counts, yet have potentially significant consequences. For example, exposed variable state (EVS, 2) occurs when saving (exposed) program state (variables) is problematic during tf.function conversion at run-time, e.g., variables becoming undefined [71].
the performance of imperative style DL code by building a bridge to graph-based execution, it was not surprising that performance—at 39.64% (111/280)—was the largest category:

**Finding 1:** At 39.64% (111/280), performance was the largest problem category encompassing `tf.function` usage.

Performance problems represent a spectrum of situations, stemming from using `tf.function` to solve a DL code performance bug to not observing the expected speedup from using `tf.function` to exhibiting worse performance that not using `tf.function`. Table 3 portrays the various fixes used to solve performance problems. Though the majority of times it was used to enhance performance of imperative DL code, we found that in 7.21% (8/111) of cases, `tf.function` was removed to alleviate performance problems:

**Finding 2:** Despite intent to improve performance, `tf.function` caused performance degradation in 7.21% (8/111) of cases.

Moreover, only 54.95% of imperative DL code performance problems were fixed by adding `tf.function`. Thus, the remaining 45.05% of cases were due to existing hybridization:

**Finding 3:** Only 54.95% (61/111) of imperative DL code performance problems were fixed by adding `tf.function`. The remaining 45.05% were due to using `tf.function`.

In fact, 25.23% of performances fixes involved altering `tf.function` arguments:

**Finding 4:** Performance fixes entailed altering developer-supplied `tf.function(...)` arguments at a rate of 25.23%.

Performance problems are further categorized into those related to “input shapes,” which make up 18.92% of all such problems:

**Finding 5:** Performance problems involved incorrect input tensor shape specifications at a rate of 18.92%.

Tensors are heavily used DL programs, and accurately matching tensor shapes (dimensions) is often required to write reliable DL code.

In hybridization, since `tf.functions` are being traced and thus converted into graphs, the underlying framework (by default) attempts to build specialized graphs for each kind of input. However, when tensors are involved, graphs may be specialized to particular input shapes, creating a situation where function retracing is excessive. Retracing can lead to significant performance degradation [43].

To curb this problem, an (`input_signature`) argument may be supplied to `tf.function` that specifies an expected range of shapes. In effect, developers provide contextual information to the framework about how `tf.functions` will be used. For instance, setting `experimental_relax_shapes to True` may cause `tf.functions` to generate fewer graphs that are less specialized on input shapes. However, this may not match reality, especially when dealing with dynamic shapes. As such, we further divide “input shape” challenges:

**Graph overly specified on input shapes** (11) Generated graphs are too specific for the context where a `tf.function` is being used, which can occur when either:

(i) `experimental_relax_shapes` is incorrectly set to `False`.

(ii) `input_signature` is unnecessarily specified. Either it should be either removed or set to `None` (the default).

**Underspecified input signature** (4) The `input_signature` parameter lacks proper arguments to avoid excessive retracing.

**Unspecified input signature** (6) The `input_signature` is missing in contexts that are advantageous to graph specialization.

API Misuse. API Misuse—the second largest problem category at 18.93%—involves situations where `tf.function` is not used in a way recommended by the API documentation:

**Finding 6:** At 18.93%, API misuse—using `tf.function` inconsistent to documentation—was the second largest category.

Misusing APIs typically results in either run-time errors or unexpected behavior. Violating DL API constraints may lead to crashes and poor performance [56,58]. In high-level, e.g., imperative DL, bugs are commonly due to misunderstandings of the guarantees offered and obligations imposed by increasingly layered software, e.g., those written against the TensorFlow API [66]. TensorFlow documentation contains a prominent sections regarding `tf.function` and AutoGraph usage constraints and limitations. If such constraints, e.g., w.r.t. control-flow, side-effects, global variables, are violated, AutoGraph will not properly generate graphs from Python code. Despite the vast documentation, at 37.74%, API confusion was the largest cause of API misuse (Table 4):

**Finding 7:** API misuse was caused by developers not understanding hybridization APIs at a rate of 37.74% (20/53).

Regarding potential category overlap, recall that API misuse is defined above as a violation of intended API usage per the documentation. Consider changing a `tf.function` argument. Performance degradation can occur when parameter usage is consistent with the documentation; it can be a tuning issue, e.g., shape-related. In such a case, according to the earlier definition, shape mismatches would not be considered API misuse as they are dependent on context.

We found that the most common way (28.30% or 15/53) to fix API misuse was to remove `@tf.function`. Of these, in 46.67% of cases (7/15), the problem cause was that `@tf.function` was used to decorate an `inner` function called by an `already decorated outer` function. As `tf.function` applies to the decorated function and all

| Table 4: API misuse causes. |
|-----------------------------|
| cause                        | count |
| API confusion                | 20    |
| Use of graph mode            | 14    |
| Decorated outer function calls unnecessarily decorated inner function | 8     |
| Incorrect tf. function argument | 7     |
| Use of eager mode            | 2     |
| Lost variable state due to graph conversion | 1     |
| Lack of static shape specifications | 1     |
| None                         | 20.5% |

![](image.png)

Figure 2: API misuse symptoms.
other functions it calls and since the inner function cannot be called from any other function besides the outer function, the inner function decorator is unnecessary and can thus be safely removed [43]. However, another 46.67% (7/15) of cases were caused by API confusion. Thus, in these cases, unfortunately, developers abandoned @tf.function—along with its potential to enhance performance—due to their confusion over how to use it. Most likely, developers were doing so to avoid run-time errors, which occurred in 62.50% (5/8) of tf.function removals not caused by unnecessary inner function decoration and 52.83% (28/53) overall (Fig. 2):

**Finding 8**: To fix API misuse, tf.function was removed 28.30% of the time. In 46.67% of these, hybridization was abandoned due to API confusion, with 62.50% causing run-time errors.

API misuse is further divided into several categories, the largest of which involves creating tf.Variables within tf.functions (10). A tf.Variable represents a tensor whose value is mutable [44]. Currently, tf.function only supports singleton tf.Variables; creating multiple tf.Variables within the scope of a tf.function results in a run-time exception [42]. Redundant decoration (8) is where multiple functions on a call path are unnecessarily decorated with @tf.function; all functions called from a tf.function are also automatically migrated to graphs. Accurately approximating such paths statically—especially in the context of a dynamic language such as Python—may be difficult, and there is ample confusion among developers on where to apply @tf.function [51] (cf. Section 2).

Executing Python side-effects (2) refers to the situation where tf.functions contain side-effect producing Python statements. As described in Section 2, executing such statements within migrated graphs can have unexpectant results, sometimes executing twice or not at all. A specific pattern of side-effects were those involving the use of iterators and generators (2), a common looping mechanism in Python code. Random number generation (RNG, 3) problems occur when developers do not use RNG facilities consistently with the documentation, commonly resulting in unexpected behavior under graph mode. For example, RNG creation inside a tf.function can only happen during the first run of the function [47]. Seeding may also not work as expected in graph mode (e.g., [89])—“when [a] global seed is set but [TensorFlow] operation seeds are not, the sequence of random numbers are the same for each tf.function” [51]. “Graph inadequately specified on input shapes” (2) involves an API misuse that is opposite to the “graph overly specified on input shapes” performance problem category described earlier. Such problems may be fixed by setting experimental_relax_shapes to False (the default). In other words, the shape specification is too general, which may result in a situation that is not amenable to graph migration [38]. For example, an input_signature may be supplied using a wild card shape to improve performance (q.v. Section 2) but results in a run-time error due to a tensor dimension mismatch [42,83]. Conversion to TFLite (1) represents problems with an alternate use case of tf.function to convert a DL model to a portable format.

**Execution Mode Incompatibility.** At 17.50%, incompatibility is the third largest problem category:

**Finding 9**: Execution mode incompatibility, at 17.50% (49/280), was the 3rd largest problem category, meaning that seamlessly using similar constructs in different modes was problematic.

Developers seemingly struggle with seamlessly using imperative DL program constructs, e.g., particular loss functions, across execution modes. Ideally, developers could toggle between eager and graph execution modes—with AutoGraph simply enhancing performance—without making code changes. In other words, incompatibility problems prevent developers from focusing on the correctness of their DL code—thinking of performance as an afterthought. Instead, to use hybridization effectively, developers must be cognizant of its internal structure, i.e., how their DL code is being migrated to graphs. Moreover, developers must (manually) be aware of which constructs are amenable to graph conversion, how best to write code that works in either mode, and how to interact with code that may be executed in a different mode.

Execution mode incompatibility problems have dire consequences. As shown in Fig. 3, 81.63% of symptoms resulting from incompatibility involve run-time errors or unexpected behavior. Such problems that only occur at run-time are difficult to uncover and, if found, may be found after deployment:

**Finding 10**: Incompatibility problems led to run-time errors or unexpected results, which do not surface until after running the code, 81.63% of the time.

**TensorFlow Bugs.** TensorFlow bugs (TFB) made up 7.86% of bugs:

**Finding 11**: TensorFlow bugs, where developers were offered workarounds or awaited new framework versions, made up 7.86% of problems. Of these, 9.09% involve deadlocks.

Such bugs involve dealing with or working around open TensorFlow bugs related to tf.function. As hybridization is relatively new, the tf.function API is under active development. Thus, it was not uncommon for such bugs to be reported to TensorFlow by filing issues against its GitHub repository; 81.82% of TFBs appear as GitHub issues (see Table 2 and Fig. 4). We categorize bugs as TFB if they were in fact real bugs with TensorFlow that required a workaround—often suggested by TensorFlow contributors—or a new TensorFlow library version to solve. If the reported bugs were not resolved to be the result of problems with TensorFlow, such bugs were not categorized as TFB but perhaps other categories.

TFB is further categorized into deadlock (2). Situations leading to the execution of a tf.function being deadlocked include using tensors as stopping condition of a recursive tf.function [29]. Deadlock may also occur as a result of other, specific tf.function code
Figure 4: Top-level problem category comparison.

(1) Favor @tf.function on Python functions containing imperative, otherwise eagerly-executed, DL code to improve performance.
(2) If possible, supply an input_signature argument to tf.function with the intended shape and types of any input tensors to avert retracing—a practice similar to that of providing type annotations to variables in dynamic languages to assist with type inferencing.
(3) When an operation is deemed incompatible with hybridization, check the documentation to see if additional steps are required to make the imperative DL code more amenable to graph conversion.
(4) Framework limitations may impede performance enhancements. Check for potential workarounds of (unresolved) TensorFlow bugs.
(5) Use tf.config.run_functions_eagerly(True) to temporarily disable tf.function to facilitate debugging.

Figure 5: Preliminary hybridization best practices.

4.2 Qualitative Analysis

This section answers (RQ2) by highlighting bug patterns with examples, summarizing causes, symptoms, and fixes, and proposing preliminary best practices and anti-patterns.

4.2.1 Performance. In listing 6, pm() is decorated with @tf.function (line 1). Using tf.function "ensure[s] that the graph for [a] function is compiled once and not every time it is called, thus gaining in speed and performance" [18], leading to best practice 1, Fig. 5.

(1) Hybridizing nested functions may cause performance degradation.
Sacrifice some modularity by either hybridizing the top-level function or refactoring the nested function to a top-level function [84].
(2) Since shared variables must be singleton, using tf.Variables in tf.functions, either directly or indirectly, may cause run-time exceptions. Either rewrite the function or do not hybridize it.
(3) Since tf.functions are compiled, using dynamic language features, e.g., lexical scoping, either directly or indirectly, may lead to run-time exceptions. Avoid such features in tf.functions where possible.

Figure 6: Preliminary hybridization anti-patterns.

Listing 6: Commit af166467 in galference: bug boxsize-nc

```python
@tf.function
@tf.function(input_signature=[]
1 2 4 5 6
state = lpt_init(linear, a0=0.1, order=1)
final_state = nbody(state, stages, nc)
6 return tfinal_field
```

Listing 7: Commit 02a3f297 in DDPG-tf2: Fixed all...this should work

While hybridization can enhance the performance of their imperative, otherwise eagerly-executed, DL code, we found that developers struggled to use it correctly. Some distrusted it, stating, e.g., that “it does far too much hidden magic” [2]. Others [82] struggled with uncontrolled retracing (q.v. Sections 2 and 4.1.1), which actually results in worse performance—speedup of 0.13 in this case—by using tf.function with only using it: “tfa.image.equalize() uses an internal scale_channel() function[,] which triggers excessive retracing ...” The problem is related to hybridizing inner functions: “I ... tried using @tf.function at the top-level of equalize(), which made it run ~25%–40% faster ....” The root cause is that, "using [embedded] functions ([i.e.,] defining functions inside function) will retrace the graph multiple times [as] the[ir] scope is not [publicly] visible, and the graphs cannot be cached” [84]. As a modularity mechanism, embedding (nesting) function definitions is a common idiom in Python, yet, currently, TensorFlow documentation does not mention this problem. Developers are left to consider the internals of AutoGraph in writing performative imperative DL code, leading to anti-pattern 1, Fig. 6.

Input Signatures. Arguments to tf.function(), particularly involving input tensor shapes, may also influence performance (q.v. Section 2). Listing 7 portrays an underspecified input signature (q.v. Section 4.1.1)—one of the most used tf.function parameters that we observed. On lines 2–6, a performance regression was fixed by adding an input_signature to a weight distribution tf.function to “make sure it does not recreate graph, which will slow down training significantly” [40]. The sequence of tf.TensorSpecs specifies the intended tensor shapes and data types (dtypes) that will be supplied to update_weights(). Otherwise, a separate (concrete) function (graph) is instantiated for each inferred input signature, which may result in retracing, leading to best practice 2, Fig. 5.
def unify(f_list):  # Stack a list of means/vars into a full block.
    self._data_dep_init(inputs)
    @tf.function
    if not self._initialized:
        return tf.reshape(tf.concat([tf.reshape(f, (-1, 1)) for f in f_list],
                                    axis=1), (-1, 1, Din))
    self._initialize_weights(inputs)
    @tf.function
    return tf.reshape(tf.concat([tf.reshape(f, (-1, 1)) for f in f_list],
                                axis=1), (-1, 1, Din))

Fmu, Fvar = map(unify, [Fmu, Fvar])  # both [N, 1, Din]

Listing 8: Commit b65848a2 in GPFlow: fix compilation issue...

Listing 9: Commit 8bab3226 in tensorflow/addons: remove tf.func

Listing 10: Commit 8bab3226 in neuro-art: Multiple request bug-fix...

4.2.2 Compilation Errors. Consider unify() originally defined on lines 3–5, listing 8 that accesses Din on line 5. This variable, however, is defined after the function definition on line 7—legal due to Python’s lexical scoping rules. In other words, the value of Din will come from the calling context. In this case, Din on line 5 is replaced with the value defined on line 7 due to unify() being accessed on line 11. The code, though, results in the following (run-time) NameError on line 5: free variable ‘Din’ referenced before assignment in enclosing scope [19]. The problem is that, while it itself is not a tf.function, ndiagquad() is called by a tf.function elsewhere—it will also be compiled into a (static) graph (cf. Section 4.1.1). Thus, dynamic language features like lexical scoping are not available in static contexts. As a result, unify() is moved to line 8, where Din is its declaration scope. Although Python is a dynamic language, developers must be aware that certain code will be compiled to static graphs, leading to anti-pattern 3, Fig. 6.

4.2.3 API Misuse. On line 1 in listing 9, tf.function is removed to fix a bug that is causing flaky tests [31]. The problem is deemed to be that @tf.function and the assert statement on lines 3–4 is incompatible. The developers express that “removing the decorator is not ideal, but stability is more important than the [speedup] we [would] get with [it]” [39]. However, this code likely causes a race condition because of a missing control dependency following the assertion. To use the assertion within a tf.function, a control dependency is required “to block follow-up computation[s] until the check has executed” as a result of the function being converted to a (static) graph [49]. This leads to best practice 3, Fig. 5.

4.2.4 TensorFlow Bugs. On line 1, listing 10, tf.function is once again removed. The problem is that—with tf.function—the application “can only process one image before” needing to restarted [20], terminating with the message: ValueError: tf.function-decorated function tried to create variables on non-first call. Recall from Section 4.1.1 that shared variables inside a tf.function must be singleton; a run-time exception ensues otherwise [42]. However, it is not obvious from listing 10 where the variable creation occurs—there are no explicit tf.Variables. The developer expresses that “removing the ... decorator is a viable workaround but not [a] best practice,” and that the root cause is an (unresolved) TensorFlow bug [88]. In terms of listing 10, the problematic line is 8, as “calling apply_gradients() on an optimizer for the first time will create its internal variables” [78]. In terms of the framework, it transpires to be related to software layering, as, “sadly[,] there [is] currently no public API to just initialize the optimizer state but not [apply it].” While several developers found workarounds for their particular situations, imperative DL code such as in listing 10 have foregone any potential performance gains from using @tf.function, leading to best practice 4 and anti-pattern 2 in Figs. 5 and 6, respectively.

4.2.5 Debuggability. To improve debuggability, @tf.function is removed on line 1, listing 11 (cf. Section 4.1.1). However, in the latest file version, @tf.function has not been replaced. Thus, the developer may have inadvertently sacrificed permanent performance gains for temporary debuggability, leading to best practice 5, Fig. 5.

5 DISCUSSION

We summarize and comment on our main findings while connecting them to other research. To help solve the problems, we put forth preliminary recommendations for practitioners, tool developers, and researchers. Though our hope is that the findings will shed light on future tool challenges and that the aforementioned descriptions and real-world examples will provide sufficient, generalizable, and actionable contexts, we nonetheless outline potential solutions.

Performance. It is not surprising that performance is our largest category (finding 1) since hybridization is centrally related to performance enhancement. The volume of performance problems is a testament to the struggles developers have in writing performant, imperative DL code. However, 45.05% of performance problems (finding 3) were due to existing tf.function usages, suggesting that developers also struggle with using hybridization effectively to achieve the performance they desire. A feasible explanation is that developers must manually decide: (i) where and when to use tf.function, (ii) the arguments to supply tf.function for their code
developers are writing imperative DL code, there exist situations where they must nevertheless be cognizant of hybridization limitations, and (iv) error messages may not be helpful. Due to Item (i), learning how to use DL APIs effectively necessitates a steep learning curve, especially considering that hybridization is relatively new. As ML systems have a quick time-to-market [86], developers may be not have the luxury of time to thoroughly understand the documentation. This is especially evident in finding 7, with 37.74% of misuses caused by API confusion. Item (ii) has been recognized by other ML/DL software studies (e.g., [91]). We conjecture that Item (iii) can also be alleviated with more tool-support, however, such tool-support in this context may require (e.g., design-by-contract) formalization of DL API specifications (e.g., modeling operation limitations in particular contexts), leading to recommendation 3, Fig. 7. A potential downside to recommendation 3 is the rapid change of ML APIs [32]. For Item (iv), developers often expressed frustration with error messages, e.g., "the main complexity in [TensorFlow] 2 is in @tf.function(), . . . error messages should be as clear as possible, especially for common problems" [28].

Zhang et al. [99] likewise observed broader API misuse in DL systems. Nadi et al. [73] also found API misuse despite ample documentation in the context of cryptography—developers prefer higher-level documentation. Current hybridization documentation tends to focus on lower-level details—future research may explore whether a similar concept will work for DL APIs. Furthermore, our findings coincide with Jin et al. [61] that many performance bugs are due performance implication misunderstandings of certain functions.

Incompatibility. Execution incompatibility of particular Python constructs was also a major theme (q.v. finding 9). Zhang et al. [99] found a similar problem in DL systems w.r.t. CPU/GPU compatibility. We again advocate for more automation to circumvent such problems. To use hybridization effectively, developers must understand which constructs are amenable to both eager and graph execution and make appropriate considerations. Tool-support, e.g., IDE recommendations, may be helpful here. To alleviate run-time errors and unexpected results, we also advocate for more testing (dynamic analysis) of (imperative) DL code that runs the same code under multiple execution modes. Testing of DL systems is an emerging yet promising area, and testing focusing on (imperative) DL code hybridization may help to shed light on: (i) where developers struggle to write performant yet reliable (imperative) DL code and (ii) potential areas of where hybridization technologies can be improved. This leads to recommendation 4, Fig. 7.

Commits vs. GitHub Issues. Performance bugs appeared more in commits than GitHub issues. The reason may be that enhancing performance typically requires a code change, which can be benchmarked. Contrarily, "incompatibility" is more difficult to quantify, often resulting in unexpected behavior or run-time errors (q.v. Fig. 3). Therefore, developers may be more likely to seek external assistance. Developers commonly file GitHub issues against TensorFlow; 93.75% of TFB issues are against the TensorFlow subject. That all UKN and TST bugs appeared in commits may be due to GitHub issues being easier to categorize than changesets and DL testing remains an emerging area, respectively.
6 THREATS TO VALIDITY

Subjects may not be representative of DL systems. To mitigate this, subjects encompass diverse domains and sizes, have been used in previous studies, and are from a data science-specific dataset (q.v. Section 3). Various GitHub metrics and DL-related keywords were used in choosing subjects. Also, hybridization is relatively new; we expect a larger selection of subjects as it grows in popularity.

Our study involved many hours of manual validation to understand and categorize bugs. To mitigate bias, we investigated referenced resources and comments made by developers to help more fully understand the challenges faced. The NLP of gitcproc may have missed bug fix changesets. Nevertheless, using it, we were still able to find 157 bugs (280 overall) that contributed to a rich bug categorization, best practices, and anti-patterns. Furthermore, gitcproc has been used previously in other studies (q.v. Section 3).

Hybridization in comparable DL frameworks may have yielded different challenges. Nevertheless, focusing on TensorFlow enables us to more thoroughly understand the intricacies involved in using hybridization effectively. Moreover, TensorFlow is a widely-studied and popular (industrial) DL framework (q.v. Section 1).

7 RELATED WORK

Cao et al. [22] characterizing performance bugs in DL systems. During their analysis of general performance bugs, they also find that developers often struggle with knowing where to add `tf.function` and how to implement decorated functions for optimal performance. Beyond performance bugs, our study includes a rich, hierarchical taxonomy of varying hybridization bug types, including input shape mismatches, API misuse, and construct incompatibility, whose results include run-time errors, unexpected behavior, and deadlock. Tambon et al. [90] examine (silent) behavioral bugs within DL frameworks and their impact on client code. Their work is reminiscent of our TFB problem category (q.v. Section 4.1) and also note that performance degradation may lead to significant problems at run-time. While they do not explicitly mention hybridization performance bugs, some of their performance bugs in imperative DL code may be alleviated by using `tf.function`. Zhang et al. [101] study API change trends in TensorFlow and for which reasons; our focus is on client code modifications involving hybridization. Baker et al. [15] extract 11 common TensorFlow API misuse patterns. Only one of the patterns (and corresponding fix suggestion) involves (a specific use case of) `tf.function`. In contrast, our study goes beyond API misuse and entails 12 top-level problem categories—24 overall—encompassing hybridization challenges.

Zhang et al. [99] present a large-scale empirical study of general DL questions on Stack Overflow. Particularly, their “CPU/GPU incompatibility” problem category resembles our execution mode incompatibility category. Concerning hybridization, whether the migrated graph executes on a GPU is typically decided by the underlying DL framework; our focus is on conversion itself. Islam et al. [56] and Zhang et al. [100] study general DL bug characteristics and present anti-patterns to avoid bugs. Islam et al. [58] study patterns in which such bugs are fixed. Chen et al. [26] explore faults in deploying DL models to mobile applications. Nikanjam and Khomh [74] catalog various design smells in DL systems and recommend suitable refactoring. Jebnoun et al. [59] correlate code smells with bugs in DL code. Liu et al. [68] characterize technical debt in DL frameworks, while Humbatova et al. [55] taxonomize (functional) faults in DL systems. Arpteg et al. [9] categorize general SE challenges in DL systems into three areas—development, production, and organizational. Liu et al. [67] study failed TensorFlow industrial jobs and propose a constraint-based approach for detecting shape-related errors. Amershi et al. [4] conduct a study at Microsoft, observing software teams as they developed AI applications. Lwakatere et al. [69] also classify SE challenges for ML systems at six different companies, focusing mainly on deployment issues. Thung et al. [93] examine bugs in three general ML systems, finding that nonfunctional bugs, of which performance problems may be categorized, require the most involved fixes. Dilhara et al. [32] study ML library evolution and its resulting client-code modifications. And, Dilhara et al. [33] and Tang et al. [91] analyze repetitive code changes and refactorings made in ML systems, respectively. While valuable, these studies do not deal with challenges faced in migrating imperative DL code to graph execution.

Several studies involve performance in other contexts. Han and Yu [52] study configurability and performance. Future work entails correlating their findings with `tf.function` arguments. Jin et al. [62] study performance slowdowns caused by system side inefficiencies. Bagherzadeh et al. [13] investigate performance in Actor-based systems. Others study language features. Farin et al. [77] study Java generics adoption. Dyer et al. [34] study language feature evolution. Khatchadourian and Masu [64] empirically assess default methods. There are many general empirical studies. Makhshari and Mesbah [70] taxonomize development challenges of IoT systems. Bagherzadeh and Khatchadourian [14] investigate common questions asked by big data developers, and Khatchadourian et al. [65] examine the use and misuse of Java streams. Engler et al. [35] and Tian and Ray [94] study errors in systems code.

8 CONCLUSION & FUTURE WORK

This study advances knowledge of the development challenges involved in migrating imperative DL code to graph execution via hybridization. A hierarchical taxonomy of common hybridization challenges was formulated and preliminary recommendations, best practices, and anti-patterns were proposed. In the future, we will explore analyzing alternative developer resources, e.g., Stack Overflow, and integrating our results into automated bug finders and refactoring detection approaches [11,95].

REFERENCES

[1] Martin Abadi et al. 2016. TensorFlow: a system for large-scale Machine Learning. In Symposium on Operating Systems Design and Implementation.
[2] 2020. Added jitted ncon. Pull request #623. google/TensorNetwork. Xanadu. (May 26, 2020). Retrieved 01/10/2022 from https://git.io/36Mx.
[3] Akshay Agrawal et al. 2019. TensorFlow Eager: a multi-stage, Python-embedded DSL for Machine Learning. (2019). arXiv: 1903.01855 [cs.PL].
[4] Sameera Amerishi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nauhapan Nagappan, Bemira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: a case study. In Internation Conference on Software Engineering. Software Engineering in Practice. IEEE. IEEE, (May 2019), 291–300. doi: 10.1109/ICSE-SEIP.2019.00042.
[5] Apache. 2018. Customer layers (beginners). Apache MXNet documentation. Retrieved 07/23/2021 from https://mxnet.apache.orgversions/1.7/api/python/docs/tutorials/packages/gluonblocks/custom_layer_beginners.html.
[6] Apache. 2021. Hybridize: Apache MXNet documentation. (April 8, 2021). Retrieved 04/08/2021 from https://mxnet.apache.orgversions/1.8.0/api/python/docs/tutorials/packages/gluonblocks/hybridize.html.
