Computer Science: Faculty Publications and Other Works

Faculty Publications and Other Works by Department

9-11-2024

Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters

Purvish Jajal
Purdue University, pjajal@purdue.edu

Wenxin Jiang
Purdue University, jiang784@purdue.edu

Arav Tewari
Purdue University, tewari6@purdue.edu

Erik Kocinare
Purdue University, ekocinar@purdue.edu

Joseph Woo
Purdue University, woo56@purdue.edu

Follow this and additional works at: https://ecommons.luc.edu/cs_facpubs

See next page for additional authors

Part of the Artificial Intelligence and Robotics Commons, Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation

Jajal, P., Jiang, W., Tewari, A., Woo, J., Lu, Y., Thiruvathukal, G.K., & Davis, J.C. (2023). Analysis of Failures and Risks in Deep Learning Model Converters: A Case Study in the ONNX Ecosystem. ArXiv, abs/2303.17708.

Authors

Purvish Jajal, Wenxin Jiang, Arav Tewari, Erik Kocinare, Joseph Woo, Anusha Sarraf, Yung-Hsiang Lu, George Thiruvathukal, and James C. Davis

Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters

Purvish Jajal
Purdue University
West Lafayette, USA
pjajal@purdue.edu

Wenxin Jiang
Purdue University
West Lafayette, USA
jiang784@purdue.edu

Arav Tewari
Purdue University
West Lafayette, USA
tewari6@purdue.edu

Erik Kocinare
Purdue University
West Lafayette, USA
ekocinar@purdue.edu

Joseph Woo
Purdue University
West Lafayette, USA
woo56@purdue.edu

Anusha Sarraf
Purdue University
West Lafayette, USA
asarraf@purdue.edu

Yung-Hsiang Lu
Purdue University
West Lafayette, USA
yunglu@purdue.edu

George K. Thiruvathukal
Loyola University Chicago
Chicago, USA
gkt@cs.luc.edu

James C. Davis
Purdue University
West Lafayette, USA
davisjam@purdue.edu

Abstract

Software engineers develop, fine-tune, and deploy deep learning (DL) models using a variety of development frameworks and runtime environments. *DL model converters* move models between frameworks and to runtime environments. Conversion errors compromise model quality and disrupt deployment. However, the failure characteristics of DL model converters are unknown, adding risk when using DL interoperability technologies.

This paper analyzes failures in DL model converters. We survey software engineers about DL interoperability tools, use cases, and pain points (N=92). Then, we characterize failures in model converters associated with the main interoperability tool, ONNX (N=200 issues in PyTorch and TensorFlow). Finally, we formulate and test two hypotheses about structural causes for the failures we studied. We find that the node conversion stage of a model converter accounts for ~75% of the defects, and that 33% of reported failure are related to semantically incorrect models. The cause of semantically incorrect models is elusive, but models with behaviour inconsistencies share operator sequences. Our results motivate future research on making DL interoperability software simpler to maintain, extend, and validate. Research into behavioural tolerances and architectural coverage metrics could be fruitful.

CCS Concepts

• **General and reference** → **Empirical studies**; • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Software verification and validation**.

Keywords

ONNX, Machine learning, Deep neural networks, Interoperabilty, Empirical software engineering, Failure analysis, User survey

1 Introduction

Deep Learning (DL) achieves state-of-the-art performance in many domains [27, 38]. Software engineers engage in many activities for deep learning, including developing, re-using, fine-tuning, and deploying DL models [4, 30, 33, 34]. They use tools at each stage: DL frameworks for development (*e.g.,* PyTorch [56]); DL model registries for re-use (*e.g.,* HuggingFace [18]), and DL compilers for deployment platforms (*e.g.,* TVM [11]). Preferably, these tools would be *interoperable*, so that DL models can move seamlessly from one to another. Model conversion errors disrupt engineering workflows or compromise the resulting models [55]. High-quality model converters are crucial to the deep learning ecosystem.

Researchers have characterized failures in most of the DL ecosystem, but not in model converters. As depicted in Figure 1, previous works have considered DL development frameworks [32, 45, 46, 58, 65] and DL deployment compilers and runtimes [29, 63]. In contrast, prior research on DL model converters is limited to measuring conversions of 5 DL models [55]. We lack systematic knowledge of failure symptoms, causes, and patterns in DL model converters.

In this work, we analyze failures of DL model converters. We first survey software engineers (N=92), focusing on their experiences with DL interoperability tools, their use cases, and failure modes encountered. Then we analyze failures in DL model converters to the ONNX IR (Open Neural Network eXchange's Intermediate Representation), the most prominent interoperability target for DL models. We
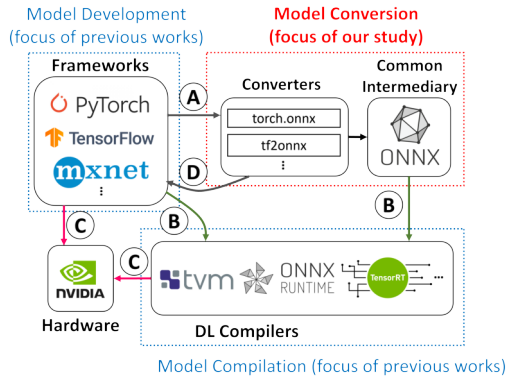
Figure 1: Paths from model development to deployment on hardware. Model interoperability facilitates reuse across frameworks and deployment environments [15]. Ⓐ represents *model conversion* to a common intermediary. Ⓑ represents *compilation*. Ⓒ represents *model deployment*. Ⓓ represents *model conversion* to a framework.

sample 200 closed GitHub issues (100 per converter) and determine failure symptoms, causes, and locations. Finally, we examine two possible root causes of model converter failures, testing hypotheses about specification updates and model types.

Our survey results show that ONNX is the most popular interoperability tool. It is primarily utilized for model deployment and framework conversion, with crashes and performance degradation being the most reported problems. Our failure analysis found that: common symptoms are crashes and incorrect model behaviors; common causes are incompatibility and type problems; and these issues tend to occur in a converter's graph translation and graph optimization components. Most results were consistent between both systems examined as well as with prior studies. Finally, in our root-cause examination of *why* converters fail, we describe some model characteristics that are correlated with converter failures.

Our contributions are:

- We survey 92 engineers and report common interoperability tools, use-cases, and pain point (§4).
- We analyze failures in two DL model converters: PyTorch and TensorFlow into ONNX. We taxonomize and measure the distribution of failure symptoms, causes, and locations (§5).
- We find that defective converter behavior is correlated with unusual models, but not with changes in the ONNX specification nor with the use of individual model layers or sequences (§6).

Significance for software engineering: DL interoperability tools, especially model converters, underpin many deployment pipelines. We conducted the first systematic study of DL interoperability tools and model converters through a user survey and failure analysis. Understanding *how* and *why* these tools fail will help software engineers make informed judgments about their robustness.

2 Background and Related Work

Here we define DL model converters via the concept of *interoperability*, and discuss prior failure studies of DL ecosystem components.

Table 1: Stages of a DL model converter to the ONNX intermediate representation, based on `torch.onnx`, `tf2onnx`, and `mxnet.onnx`.

| Component	Definition
Load Model	Framework representation → ONNX graph. Tracing used for dynamic graphs (*e.g.*, PyTorch).
Node conversion	Graph nodes replaced by ONNX equivalents.
Optimization	Nodes (*e.g.*, operator fusion), dataflows (*e.g.*, DCE).
Export	Model serialized into protocol buffer (protobuf).
Validate	Syntactic checks (compliance with spec) and semantic checks (behavioral changes).
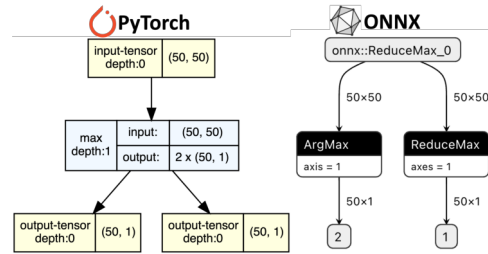
Figure 2: PyTorch model converted to ONNX Intermediate Representation. The PyTorch model calculates the per-row maximum using `torch.max`. In ONNX, this uses the operators `ArgMax` plus `ReduceMax`.

2.1 DL Model Conversion as Interoperability

In the context of DL, **interoperability** focuses on model reuse and is defined as *the ability of DL software to exchange DL models (i.e., deep neural networks/DNNs)* [44]. Wegner describes two patterns for interoperability: creating *pairwise mappings* between systems; and introducing a *common intermediary* understood by all participants [69]. These patterns are illustrated in Figure 1 in the context of DL. The two patterns trade-off customizability for scalability.

Two kinds of DL systems interoperate on DL models: *frameworks* for DL model development, and *runtimes* for DL model deployment. In the downward path of Figure 1, pairwise mappings occur in DL compilers such as TVM [11], which map from framework representations into internal compiler representations for each supported framework. Along the rightward path, common intermediaries such as ONNX [1] and MMDnn [44] give standard representations for DL models. In this pattern, each framework and runtime has a one- or two-way adapter to the common intermediary.

DL Model Converters: Model converters fill a purpose similar to compiler front-ends [42]. They transform a model from a DL framework into a high-level IR representing the model's computations and control flow. Graph-level optimizations are applied before further conversion to a low-level IR for hardware optimization and code generation. Table 1 summarizes a typical design.

Figure 2 illustrates a model converted from PyTorch to ONNX. Conversion is challenging, as noted by AirBus [25, 54] and others [13], because it maps between graphs expressed with different operators and different semantics. Model conversion can produce models that are incompatible with runtimes or have different behaviours. For a *compatibility* issue, in PyTorch #78721 a converted

ONNX model had a type mismatch [66]. For a *behavioral* issue, in PyTorch #74732 a converted ONNX model's prediction changed [73].

After conversion to an IR, models can be rendered back to DL frameworks or deployed to hardware. Framework-to-framework conversion can bypass issues in model reengineering [33]. Deployment from an IR allows DL runtimes to optimize against the IR rather than the many DL frameworks.

2.2 Failure Studies of DL Components

Software engineering failures inform development and maintenance [5, 6, 57]. Previous failure analyses of DL interoperability software focused on the "Development" and "Runtime" components of Figure 1. Chen *et al.* studied 800 defects from 4 DL frameworks to obtain testing guidelines [45]. Shen *et al.* studied DL compiler defects [63], and others have tested DL compilers [43, 72]. Some of this work has incidentally examined interoperability failures [43, 52]; for example, Shen *et al.*'s study of DL compilers included "front ends" offering interoperability through pairwise (DL framework specific) and common intermediary (ONNX) model loaders [63];

The two prior failure studies of DL model converters focused on testing and fault localization. Openja *et al.* converted 5 popular DL models [55]. They considered only the current state of model converters, not past failures. Louloudakis *et al.* studied behavioral issues resulting from framework-to-framework conversion [47]. They found failures in 10 out of 36 conversions. They created a fault localization and repair pipeline to localize and fix discrepancies [48].

In light of this literature, the main contribution of our work is the first systematic analysis of failures in DL model converters. As a conceptual contribution, we frame DL model converters as a class of interoperability software. We consider *how* and *why* they fail.

3 Research Questions & Study Design

Figure 3 shows our RQs (§3.1) and study design (§3.2), detailed next.

3.1 Research Questions

We summarize the related work in §2 as follows: The DL ecosystem is growing more complex (§2.1), motivating a shift to common intermediaries such as ONNX. Analyzing failures in this emerging pattern will inform ecosystem participants of risks and opportunities for improvement. Prior work examined failures in DL frameworks and runtimes, but has paid little attention to interoperability (§2.2).
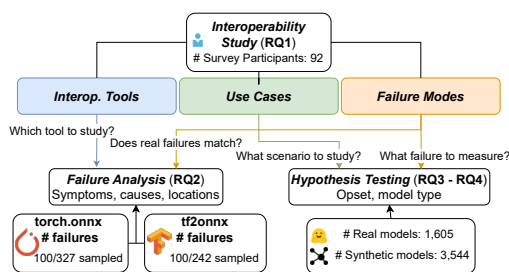

Figure 3: Goal, research questions, methods, and data sources.

Our study fills this knowledge gap by analyzing failures in DL interoperability software. We specifically focus on ONNX, the leading DL interoperability framework. Our research proceeds in three themes: (1) why and how engineers use interoperability tools; (2) a failure analysis of the most popular interoperability tool; and (3) evaluating hypotheses about the root causes of failures.

Theme 1: Interoperability User Survey (§4)
 RQ1 How and why do engineers use interoperability tools?
Theme 2: Failure Analysis (§5)
 RQ2 What are the failure characteristics in DL interoperability software — symptoms, causes, and locations?
Theme 3: Hypotheses on ONNX Failure Causes (§6)
 RQ3 Does ONNX evolution affect converter failure rates?
 RQ4 Do model types affect converter failure rates?

3.2 Study Design

Figure 3 relates research questions to methods. For RQ1 we use a user survey (§4). The survey results inform the remainder of the work. For RQ2 (§5) and RQ3-4 (§6) we apply methods from mining software repositories and software testing. We analyze GitHub issues (RQ2), correlate issue frequency to ONNX versions (RQ3), and evaluate ONNX on a range of model types (RQ4).

4 Theme 1: Interoperability Study

To understand the use of DL interoperability, we surveyed DL practitioners on DL development and deployment practices. We selected the survey methodology to gather insights across diverse practices and experiences [16]. Our method is given in §4.1, results in §4.2.

4.1 Methodology

4.1.1 Instrument Design. We followed Kitchenham & Pfleeger's guidelines to develop our survey instrument [39]. Table 2 illustrates the result. We asked general questions about DL interoperability, and specific questions about ONNX (interview data [62] and GitHub suggest it is most popular). We removed some demographic questions after the pilot study (e.g., ML/SE expertise, organization types) to reduce survey time in order to increase response rates, as is recommended by survey guidelines such as [26]. The pruning removed both demographic and technical questions, and reduced the survey time from 15-20 minutes (with which we had a 1% response rate from 130 initial invitations) to 5-8 minutes (yielding a 4% response rate from subsequent invitations). We made sure that we did not prune too far, so that the collected demographic data (reported in Table 3) is consistent with prior empirical software engineering works such as [17, 34, 51]. We iterated internally, piloting with 3 external participants to check instrument clarity.

We sent the survey out in batches to allow for iteration. We examined the results halfway to determine whether the instrument aligned with a larger sample of users' behavior. We observed an unexpectedly high incidence of one use case, and so in the second half we added an open-ended question to clarify this use case.

4.1.2 Population and Sampling. With approval from our Institutional Review Board (IRB), survey respondents were recruited from Hugging Face users. The Hugging Face ecosystem is the primary location for deep neural network-based model development and reuse [34, 36], so its users are a suitable population for questions about

Table 2: Survey excerpt. *Shown to second half of participants.

Topic	Example questions
Demographics	(1) How long have you worked on ML/DL projects? (2) What deployment environments are targeted?
Interoperability tool usage	(1) What do you consider when choosing between deploying from a DL framework vs. via a tool like ONNX? (2) Do you use ONNX as part of your model development and deployment process?
Use cases	(1) For what purpose do you use interoperability tools? (2*) If you use ONNX for framework-to-framework conversion, please describe your use case further?
Failure modes (ONNX-specific)	(1) Do you commonly encounter problems while working with ONNX models? (2) If you encounter such problems, how do you address them?

DL interoperability tools. To increase the likelihood of responses from experienced software engineers, we collected email addresses from users with PRO accounts (*i.e.,* paid), and from accounts in organizations marked as *company*, *community*, and *non-profit* (excluding types such as *education*). Participants received $10 gift cards.

We targeted a confidence level of 90% with a 10% margin of error. With an estimated Hugging Face user population of 1.2 million [22, 24], a sample size of 69 respondents was needed. We sent out surveys in batches until reaching our desired sample size. In total, we distributed our survey to 228 PRO users and 1,985 organization members. We received a total of 92 valid responses (4% response rate). All questions were optional to improve the response rate, so we do not have responses from all subjects on all questions.

Table 3 shows respondent demographic information.

Table 3: Participant demographics (N=92 but 3 skipped this).

Kind	Distribution of Responses
ML experience	<1 yr. (8) ; 1-2 yr. (17) ; 3-5 yr. (32) ; >5 yr. (32)
SE experience	<1 yr. (4) ; 1-2 yr. (12) ; 3-5 yr. (20) ; >5 yr. (53)
Org. size	Small, < 50 employees (48) ; Medium, < 250 employees (17) ; Large, > 250 employees (24)
Deployment environment	Web application (59) ; Cloud and data center (52) ; Desktop application (19) ; Mobile (14) ; IoT/embedded systems (14) ; Other (4)

4.1.3 Analysis. Most questions were closed-ended (multiple-choice, checkbox). Qualitative analysis was needed for 3 open-ended questions, about use cases, model deployments, and interoperability problems. Two authors reviewed the data and agreed that the responses to these questions were short and did not involve much subjectivity. Therefore, all data were analyzed by one author.

4.2 RQ1: Engineers and Interoperability Tools

> **Finding 1.** ONNX is the most popular interoperability tool, used by approximately 42% of respondents. Meanwhile, 41% of respondents do not use interoperability tools.
> **Finding 2.** Model deployment and framework-to-framework conversion are the primary use cases for interoperability tools, both used by over half of the interoperability-using respondents.
> **Finding 3.** Many respondents (59%) encounter ONNX problems. Crashes and performance differences are the most common failure modes. Each was encountered by ~35% of respondents.

4.2.1 Interoperability Tool Usage. The majority of respondents utilized the PyTorch or TensorFlow frameworks for model development. Most of the respondents utilized PyTorch (89%). TensorFlow is the second most used framework (37%). JAX, MLX, and other frameworks made up 20% responses. The data show that the respondents often use multiple frameworks or have moved from one framework to another (*e.g.,* moving from TensorFlow to PyTorch), perhaps motivating their use of interoperability tools.

Approximately 42% (39/92) of respondents reported using ONNX. Only 15 respondents used other interoperability tools such as MMdnn and NNEF. The rest (41%) do not use interoperability tools. In our survey, although we did not gather extensive data on participants' specific roles, insights from their deployment considerations suggest they are primarily model developers. They either lack model deployment responsibilities or deploy the models directly.

4.2.2 Use Cases. Interoperability tools are used equally to interoperate between frameworks and to deploy models. 69% (37/54) of respondents report using model conversion for deployment of DL models. Whereas 52% (28/54) of respondents report using framework-to-framework conversion (*e.g.,* PyTorch to TensorFlow).

For the respondents who use interoperability tools for framework-to-framework conversion, most of them use the tools to "*integrate the models in non-Python code*". For example, one respondent wants to "*interact with the base CUDA system*". Another respondent "*exports [pre-trained models] to ONNX for import into Axon, an Elixir/Nx based DL framework that often [lacks] native pre-trained models*".

Table 4 presents the deployment considerations of practitioners when using interoperability tools. Many practitioners report deploying directly from frameworks when they want easy deployment or expect to update models often. The broader organization's practices may play a role: "*if I have control over the entire training to deployment pipeline, it is easier to use PyTorch exports*".

Table 4: Induced themes for model deployment consideration when using interoperability tools, such as ONNX. Based on code-able responses from 32 participants.

Theme	# Participants (%)
Simplicity	8 / 32 (25%)
Deployment requirements	7 / 32 (22%)
Inference speed	5 / 32 (16%)
Portability	5 / 32 (16%)
Other (*e.g.,* maintenance, stability)	2 / 32 (6%)
Deploy directly from the DL frameworks	13 / 32 (35%)

Interoperability tools are preferred when respondents care about performance gains, portability, language compatibility, or if they require exotic configurations. Examples were: "*[PyTorch in browser is] too big*", and "*[ONNX offers] portability over performance*".

4.2.3 Failure Modes. We ask practitioners about their experiences using ONNX. 59% (32/54) of practitioners report encountering problems when using ONNX, while 41% (22/54) report that they do not commonly encounter problems with ONNX models. Crashes (19/32, 59%) and performance differences (19/32, 59%) are the most prominent. Often models do not convert, or there are performance differences between the original and converted models. There are also some other problems (6/32, 19%) mentioned by the respondents. Examples included "*ONNX doesn't support Fourier layers*" and "*driver problems during...deployment on user machines*".

Table 5 presents practitioners' strategies for resolving issues with the ONNX converter. Many (44%) turn to community resources, including GitHub issues and Stack Overflow posts, for help. One-third (31%) verify the conversion process through testing, while 19% explore solutions by changing ONNX versions. Some had experienced no good solution, resolving ONNX issues "*case-by-case*".

Table 5: Induced themes for strategies to address ONNX issues. Based on code-able responses from 16 participants.

Theme	# Participants (%)
Seeking help from the community	7 / 16 (44%)
Test with executions	5 / 16 (31%)
Version changes	3 / 16 (19%)
Other (*i.e.,* documentation, config)	3 / 16 (19%)

5 Theme 2: Failure Analysis (of ONNX)

The second theme of this work is a failure analysis of deep learning interoperability software, specifically, deep learning model converters. We apply the method of Failure Analysis [5, 6, 40, 57], which characterizes and reports the distributions of past failures to revise engineering methodologies and prioritize research targets.

Many model converters have been proposed, but our survey data show which converters are of practical interest (§4.2.1): those for the ONNX framework (~half of respondents use it); and specifically those associated with the PyTorch and TensorFlow model development frameworks (the two most common).

About ONNX: The ONNX (Open Neural Network eXchange) specification provides a common representation that DL frameworks and runtimes use to represent DL models [53]. The ONNX specification has three main components: (1) A definition of a computational graph; (2) definitions of standard data types; and (3) definitions of built-in operators such as ArgMax [53]. The ONNX specification changes regularly to keep up with DL framework evolution [64]. These changes consist of *adding* new types and operators, and *updating* the behavior of existing operators. These changes are versioned within *operator sets*. As of 2023 there have been 18 operator sets, totaling 149 additions and 217 updates to the available operators.

5.1 Methodology

5.1.1 Data Selection. Figure 4 depicts our data selection method. We describe the five main stages and rationales next.
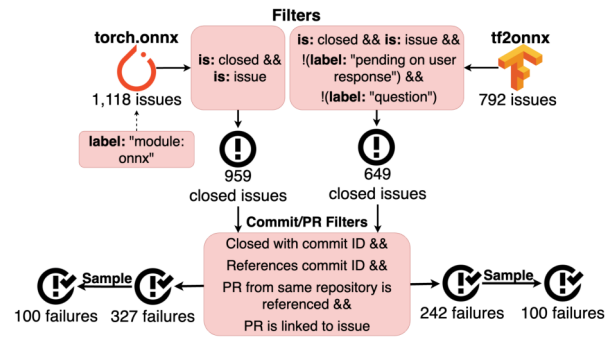

Figure 4: Filtering of issues for each repository studied. *Filters are using GitHub search predicates. Commit/PR filters are applied to issue timeline events. Data were collected on Jan. 6, 2023. 100 issues per repository were analyzed.*

Table 6: Popularity and data availability of ONNX converters. Framework→ONNX converters (top) have notably more activity than ONNX→Framework converters (bottom). *: torch.onnx is a component in the PyTorch repository, so stars are skewed. PyTorch issues are filtered for those in the converter. Data from April 11, 2024. Parenthesized numbers are issues at time of data collection (July 7, 2023).

Project	Input	Stars	Forks	Closed Issues
torch.onnx	PyTorch, Caffe2	*77,429	20,938	*1,286 (959)
tf2onnx	TensorFlow, Keras	2,200	425	848 (792)
Paddle2ONNX	PaddlePaddle	637	145	192
sklearn-onnx	Sci-kit Learn	506	94	328
onnx2torch	ONNX	370	35	32
onnx-tf	ONNX	291	25	129
onnx-coreml	ONNX	385	79	archived

(1) Repositories: For the reasons noted above, we studied the DL model converters from PyTorch and TensorFlow into the ONNX IR (torch.onnx and tf2onnx, respectively). We note that among ONNX model converters, those for PyTorch and TensorFlow have the most failure data available on GitHub (Table 6).

Using the PyTorch and TensorFlow converters covers two relevant differences within DL and interoperability. Within DL, these converters include both common representations of computational graphs: static (TensorFlow) and dynamic (PyTorch) [8, 31]. Within interoperability, these converters include both kinds of converter owners. Converters are naturally owned either by the upstream producer of the data or the downstream consumer of the data. In our case, torch.onnx gives an example of upstream ownership (it is owned by the PyTorch engineers), while tf2onnx gives an example of downstream ownership (it is owned by the ONNX engineers).

(2) Filters for Relevance: We follow prior work [10, 23, 61], and study closed issues because these issues typically contain greater information about failure causes and symptoms. For each repository, we collected closed GitHub issues related to ONNX conversion. For torch.onnx, the PyTorch repository marks ONNX converter issues with the label *module: onnx*. We collect all closed issues with this label, yielding the search filter: *is:issue label:"module: onnx" is:closed*. For tf2onnx, all issues are relevant. We remove issues labeled "pending on user response" and "question" because these issues may not be

failures. This yields in 959 issues in torch.onnx (from 20,782 issues, not all ONNX-related) and 649 issues in tf2onnx (from 792 issues).

(3) Filters for Data Availability: Following prior work [23, 32], we subsequently filtered for GitHub issues that contained enough information for failure analysis. We filter issues for sufficient information (*e.g.,* the issue is resolved with a commit and pull request). This filtering is conducted upon the timeline events for each GitHub issue and the filtering criteria are given in Figure 4. This filter yields 327 issues in torch.onnx and 242 issues in tf2onnx.

(4) Filter Validation: We piloted filters to ensure they captured relevant issues (recall) but not irrelevant issues (precision). We hand-labelled 50 issues per repository prior to filtering, applied the filter, and measured recall and precision. If recall and precision ≥ 0.8 we consider our filter to acceptably balance manual work against bias. For torch.onnx, we measured recall of 0.81 and precision of 0.94. For tf2onnx, we measured recall of 0.82 and precision of 0.93.

(5) Sampling: Given the similar number of issues after filtering (Figure 4), we randomly sampled and analyzed 100 issues per repository, or roughly one-third of relevant issues. This quantity is comparable to the proportion (~44%) analyzed in prior work [63].

5.1.2 Data Analysis. We describe converter failures in 3 dimensions: *location*, *symptoms*, and *causes*.

Location: We map failures to the converter stages in Table 1: Load Model, Node Convert, Optimization, Export, and Validate.

Symptoms & Causes: We adapted taxonomies from Shen *et al.* [63], the closest related work, as discussed in §2.2. Conceptually, we chose to reuse deep learning compiler taxonomies due to the expected similarity between compiler front-ends and converters. This choice also follows the recent advice of Amusuo *et al.*, who called for greater reuse of taxonomies in failure analysis research, where feasible [5]. To assess whether the taxonomies are applicable to converters, we trialed the taxonomies on a sample of 30 randomly chosen issues from each repository. Two researchers classified symptoms and causes. Computing inter-rater reliability via the Kappa coefficient [12], we found κ=0.90 (causes) and κ=0.95 (symptoms), *i.e.,* "strong agreement" [49] for both taxonomies.

The taxonomy we used for symptoms is: crash, wrong model, bad performance, build failure, and unknown/other. In a *crash*, conversion errors out. A *wrong model* is syntactically sound but semantically incorrect. *Bad performance* indicates unexpectedly high time or memory cost, *e.g.,* worse than on a previous version. *Build failure* means the user could not install the converter.

The taxonomy of failure causes is more complex — see Table 7.

Interrater Agreement: In our pilot study of applying Shen *et al.'s* taxonomies, we observed strong agreement between raters. Therefore, we relied on a single rating (one rater analyzing all issues) of the 200 sampled issues. To assess the risk of taxonomic drift [70], a second rater analyzed one tranche of randomly selected failures. In those 20 samples, the two raters had perfect agreement/κ=1.0. We conclude the single-rater analysis was sound.

Table 7: Taxonomy of failure causes, adapted from [63], with concise definitions. Italics and Strikeout indicate changed or deleted wording, respectively. Bold indicates additions.

Cause	Definition
Incompatibility - Internal	API compatib. issues in model converters.
Incompatibility - External	Compatibility issues with third-party libraries (e.g., ONNX, TensorFlow).
Type Problem - Node	Issues related to atomic DL operators in model conversion.
Type Problem - Tensor	Issues related to the types of tensors.
Type Problem - Conventional	Issues with types of conventional variables in software systems.
~~Tensor~~ Shape Problem	*Input/output tensor shape issues.*
Alg. Error - Optimization Error	Issues in model converter optimizations.
Alg. Error - *Tracing Issue*	Issues tracing dynamic models in computational graphs.
~~Alg. Error - Non-optim. Code Logic~~	~~Bugs outside DL compiler optimizations.~~
Testing	Test errors, incl. flaky or missing tests.

5.2 RQ2: The Characteristics of Failures

> **Finding 4.** Location: Most failures are in *Node Conversion* (74%).
> **Finding 5.** Symptom: The most common symptoms in DL model converters are *Crash* (56%) and *Wrong Model* (33%).
> **Finding 6.** Causes: *Crashes* are largely due to *Incompatibilities* and *Type Problems*. *Wrong models* are largely due to *Type Problems* and *Algorithmic Errors*.

5.2.1 Failure Locations. tf2onnx and torch.onnx have similar failure location distributions (Table 8). The most common location of failures is the *Node Conversion* stage: 148/200 failures occur in this stage, with a similar proportion in tf2onnx (70%) and torch.onnx (78%). The *Graph Optimization* stage is the second most common location of failures, with 19 in total. The distribution of failures in this stage differs between converters, with ~3x more in tf2onnx.

Table 8: Failure locations (cf. Table 1). The majority of failures occur during *Node Conversion* in each ONNX converter.

Location	TF	PT	Total
Load Model	5	6	11 (6%)
Node Conversion	**70**	**78**	**148 (74%)**
(Graph) Optimization	14	5	19 (10%)
Export (Protobuf)	1	0	1 (1%)
Validation	0	3	3 (2%)
(Not Distinguishable)	10	8	18 (9%)
Total	*100*	*100*	*200 (100%)*

5.2.2 Failure Symptoms. The distributions of symptoms for both tf2onnx and PyTorch are similar — see Table 9. The most common failure symptoms are *Crash* and *Wrong Model*, comprising ≥85% of failures in each converter. The *Bad Performance* and *Build Failure* make up around 5% of failure symptoms.

Our results show the similarity of the failure symptom distribution between DL model converters (our work) and DL compilers (Shen *et al.* [63]). In the last two columns of Table 9, we see the *Crash* and *Wrong Model* symptoms make up the majority of symptoms across both DL converters and DL compiler front-ends. The *Wrong Model* and *Crash* symptoms appear characteristic of interoperability in this context, regardless of implementation.

Table 9: Distribution of symptoms. *TF*: `tf2onnx`. *PT*: `torch.onnx`. *DL Comp.*: **Symptoms of DL compiler failures in compiler front-ends, per Shen *et al.* [63]. Percentages are rounded.**

Symptom	TF	PT	Total	DL Comp. [63]
Crash	**50**	**62**	**112 (56%)**	**226 (63%)**
Wrong Model	**35**	**30**	**65 (33%)**	**100 (28%)**
Build Failure	3	2	5 (3%)	3 (1%)
Bad Performance	2	1	3 (2%)	6 (2%)
Hang	0	0	0 (0%)	4 (1%)
Unreported	10	5	15 (8%)	20 (6%)
Total	*100*	*100*	*200 (100%)*	*359 (100%)*

Table 10: Joint distribution of primary causes and symptoms. The majority of *Crashes* result from *Incompatibilities* and *Type Problems*. *Algorithmic Errors* that result in *Wrong Models* occur more often in `tf2onnx`. The top-5 causes in terms of frequency are shown, with the rest binned as *Other*. Rare symptoms are likewise binned as *Other*.

Cause \ Symptom	Crash		Wrong Model		Other		Total	
	TF	PT	TF	PT	TF	PT	TF	PT
Incompatibility	**19**	**28**	4	3	2	1	*25*	*32*
Type Problem	**8**	**14**	17	13	0	2	*25*	*29*
Algorithmic Error	4	3	10	3	4	0	*18*	*6*
Shape Problem	5	4	4	7	0	1	*9*	*12*
API Misuse	6	5	0	1	0	0	*6*	*6*
Other	8	8	0	3	9	4	*17*	*15*
Total	50	62	35	30	15	9	100	100

5.2.3 Failure Causes. We report the joint distribution of failure causes by symptom in Table 10. The most common failure causes are the *Incompatibility* and *Type Problem*, with more than 50% of failures exhibiting these causes for both converters. *Algorithmic Errors* and *Shape Problems* each contribute ∼20% of cases. By symptom, crashes were caused by incompatibility and type problems, while wrong models were caused by type problems and algorithmic errors.

For most causes, we see a similar distribution in each studied converter. For *Algorithmic Error*, we observe that `tf2onnx` has three times as many as `torch.onnx`. The disparity in *Algorithmic Errors* varies by the subclass, with the majority in `torch.onnx` being related to the loading of models namely tracing. An example is PyTorch #84092: *trace not working on autocast*. In contrast, in `tf2onnx` the majority are related to optimizations, such as `tf2onnx` #226 (*incorrect reshape*) and `tf2onnx` #1719 (*incorrect folding*). We conjecture that this difference may have a deeper cause: recall that the PyTorch converter is owned by the PyTorch engineers, while the TensorFlow converter is owned by the ONNX team (§5). This difference in ownership may reduce the team's understanding of the implications of optimizations, leading to worse outcomes in TensorFlow.

6 Theme 3: Investigating Deeper Causes

In the final theme of this work, we investigate the possibility that ONNX converter errors have a shared *structural cause*. By this, we mean a latent cause beyond the code-level causes in the taxonomy of Table 7. If so, ONNX users could use this factor to better assess their risk for the associated failure modes. Based on our survey data

(§4) and failure analysis (§5), we formulate and test two hypotheses of structural failure causes. *The RQs and hypotheses are*:

RQ3: Does ONNX evolution affect converter failure rates? We hypothesize H_{RQ_3}: *Changes in ONNX operator sets are correlated with increased defects*. This hypothesis is based in data from our survey and failure analysis. In the survey, 19% of respondents reported changing ONNX versions as a possible solution (Table 5). This suggests that conversion defects (not just compatibility issues) may be localized by version. In the failure analysis, we found that crashes were largely due to *Incompatibilities* or *Type Problems* (Table 10). These failure modes relate to API compatibility and conversion correctness, which can be affected by changes in the ONNX opset.

RQ4: Do model types affect converter failure rates? We hypothesize H_{RQ_4}: *Failures are caused by model structure, i.e., models with particular layers (node) or layer sequences are more prone to defects*. This hypothesis is primarily based on our failure analysis. The majority of failures occurred during *Node Conversion* (Table 8), indicating that nodes and node sequences may be problematic. Survey data also had a relevant anecdote: one respondent solved conversion issues by re-implementing models in pure tensor operations, implying that conversion may fail due to the use of certain operations. In the remainder of this section, we describe the method and result for testing each hypothesis. Ultimately, we find evidence to support H_{RQ_4}. This result will help guide future validation efforts (§7.1).

6.1 RQ3: Does ONNX Evolution Lead to Failures?

> **Finding 7.** H_{RQ_3} is rejected: Changes in ONNX operator sets are ***not*** strongly correlated with increased defects.

Here we describe the method and results for testing H_{RQ_3}: That changes in ONNX operator sets are correlated with increased defects. The expected correlated failures in the model converters are incompatibility and type problems (defined in Table 7).

6.1.1 Methodology. We test the hypothesis by checking whether *larger* changes in the ONNX specification correlate with *greater* incidence of failures in DL model converters. To obtain the size of changes to the ONNX specification, we count the number of operator changes (additions or updates) per release in the ONNX release notes (Figure 5). We approximate the number of failures per release using the GitHub issue creation times of the failures sampled for RQ2. We then attribute each issue to the nearest previous ONNX release.

We note four ways in which this measurement approximates: (1) DL model converters lag behind ONNX releases (this might cause a failure to be mis-attributed to another release, *i.e.*, offset in time); (2) Failures might be in any ONNX available release, not just the most recent (possibly inflating the failure rate of a given release); (3) our failure analysis data were randomly sampled, possibly undersampling certain time windows (though we note that our sample comprises ∼30-50% of qualifying issues); (4) as noted in the user survey (§4.2.3), users may simply revert to a previous operator set without opening a bug report. For items 1 and 2, issues do not reliably include ONNX versions, so no more accurate data is available. For item 3, we use the data sampled during RQ2 so that we can check incidences by cause (incompatibility and type problems).

We test the hypothesis qualitatively and quantitatively. Qualitatively, we inspect a visualization of the hypothesized trend. We also
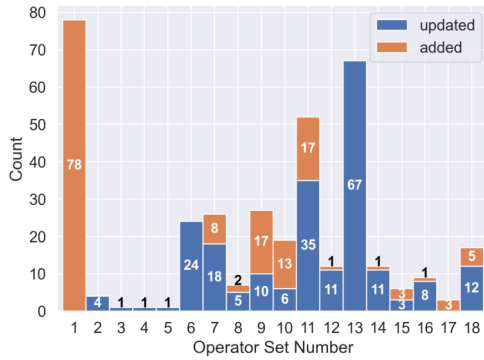
Figure 5: Additions and updates of operators by ONNX operator set version, from version 1 (2017)—version 18 (2022). Version size := sum of operator additions and updates.

measure the relationship, assessing the correlation in the number of changes in an ONNX release and the number of failures between its release and the next. We use the Spearman correlation, which is a commonly-used and robust metric for measuring a monotonic relationship between two variables [21].

6.1.2 Results. Figure 6 depicts the hypothesized correlation between ONNX releases and model converter failures from §5. After larger ONNX releases (Figure 5) we expect more failures (Figure 6).

Qualitatively, in Figure 6 we see no discernible increase in the number of failures following larger ONNX updates. Quantitatively (Spearman), results are similar. The test yields a weak positive correlation ($\rho = 0.34$). Similarly, Incompatibility and Type Problems are weakly positively correlated ($\rho = 0.33$). In the test, we discarded the first release (converters may be unstable) and the most recent release (insufficient data). We also merged releases 2–7 because they are too close together (released within a 1-month period).

Conclusion: Given the weak correlations, we do not find evidence to support H_{RQ_3}. There is a fairly steady rate of defects in ONNX model converters, whether the associated ONNX release is large or small.

6.2 RQ4: Do Model Types Affect Failure Rates?

> **Finding 8.** Real models convert well: converter crashes and incorrect behavior affected only 5% of models. Synthetic models show incorrect model behavior more often than Real models: 320/3544 (9%) of synthetic models vs. 20/1605 real models (1%).
> **Finding 9.** We find support for H_{RQ_4}. Though incorrect conversions are not directly attributable to the use of unusual operators, they may be attributable to operator sequences.

For this RQ we analyze models at two scales, *macro* and *micro*. In our *macro* scale analysis (§6.2.1, §6.2.2), we study entire models (all layers). For our *micro* scale analysis (§6.2.3, §6.2.4) we study individual layer and sequence patterns from the macro analysis.

6.2.1 Method for Macro Analysis. At the macro scale, we test DL converters using two types of models and then analyze the failure-inducing inputs. We used a differential testing approach (Figure 7). Differential tests need (1) inputs, and (2) a difference criterion [50].
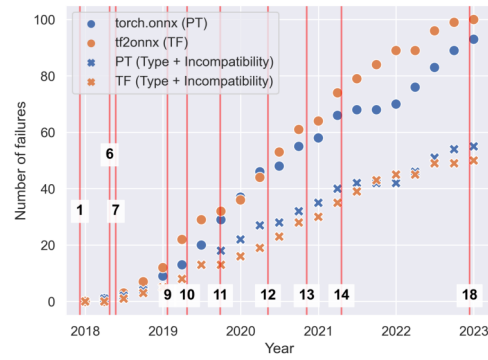

Figure 6: Cumulative number of failures in the `torch.onnx` and `tf2onnx` converters, plotted quarterly from 2018-2023. The gap between two points is the number of newly opened issues during that time period. The O's track all failures. The X's track the subset of Incompatibility and Type Problem failures. The annotated vertical lines indicate the release of ONNX operator sets with ≥ 5 changes.
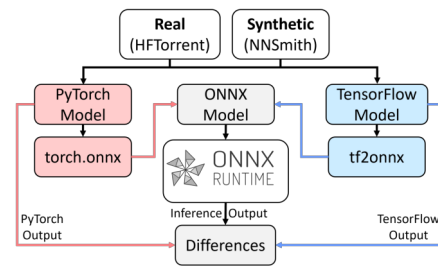

Figure 7: Macro analysis: *Real* and *Synthetic* models are converted and then differences are measured. For synthetic models, we used `tf2onnx` or `torch.onnx` directly. For real models we used HuggingFace's converter, which does preprocessing and then calls those converters.

(1) Inputs: For inputs, we converted both real models and synthetic models. Real models contain input patterns that the converter expects. Synthetic models are more diverse and should exercise edge cases on the converter. Real and synthetic models came from:

- *Real Models:* We used the HFTorrent dataset from Jiang *et al.* [34]. At time of experiment, this was the largest available set of real-world DL models, containing 63,182 pre-trained models collected from the HuggingFace model registry.[1] We filtered for the 1,761 models with both PyTorch and TensorFlow versions, allowing comparison between the two converters on similar inputs. These models represent 112 distinct full architectures (58 backbones).
- *Synthetic Models:* We systematically generated synthetic DNNs for conversion, using the NNSmith tool [43]. NNSmith generates random valid TensorFlow and PyTorch DNNs that use operators supported by ONNX. We added a parameter to NNSmith to generate DNNs of a given size (# nodes). We then *systematically generated DNNs containing between 15-100 nodes in increments of 5*. For each node count, we generated DNNs until two conditions were met:

---

[1]Future measurements could use the PeaTMOSS dataset [35] or its successors.

(1) All NNSmith-supported operators appeared at least once in the family; and (2) ≥ 100 distinct models were created.

In initial tests, we found the resulting models were often convertible yet unsupported by ONNX RunTime. To address common errors, we constrained the NNSmith synthesizer. Specifically, we removed 1 operator out of 71 for torch.onnx and 3 out of 56 for tf2onnx. Additionally, we confined the tensor data type to float32.

There are other DL model generation approaches, *e.g.,* MUFFIN [28], COMET [41], and LEMON [68]. These tools generate inputs by mutating seed models, typically real models. We viewed HFTorrent as a sufficient source of real models, and synthesized with NNSmith as a complementary approach.

(2) Difference Criterion: For each model, we attempt to convert it to ONNX using its respective converter. If conversion succeeds, we load it into the ONNX RunTime. For models that load successfully, we perform inference on both the original model (using PyTorch or TensorFlow) and the converted model (using ONNX Runtime). For *real* models, we use the inputs provided by the model owners to test the model. (These are also known as "dummy inputs" on the HuggingFace platform.) For *synthetic* models, we use 100 randomly generated inputs matching the model's input shape.

In both cases, to measure model misbehavior we used a simple community-accepted approach: the distance measure and threshold used by the PyTorch exporter. This rule is that *between the original and ONNX-converted models, in the inference result tensors, the maximum absolute element-wise difference should be* $< 10^{-7}$ [59].

We considered alternatives to measure behavioral differences. A more general notion of tolerance or "acceptable error" could be used, but the choice of tolerance is not well established — the PyTorch verification tool uses 10^{-7}, NNSmith uses 10^{-3} [43]. Openja *et al.* [55] proposed using model accuracy and robustness, but this method requires training each model on a suitable dataset. Training substantially increases the cost of the measurement, and the dataset requirement limits the types of models that can be generated.

6.2.2 Results for Macro Analysis. **Real Models:** Real models exhibited few converter failures and little incorrect model behavior. Most issues occurred within the HuggingFace-specific converter. Converter failures occurred for only 85/3,522 models (~2 %). Once models reached torch.onnx and tf2onnx, over 90 % could convert and had equivalent behavior in ONNX Runtime.

One interesting form of failure we observed was models with identical architectures but varying conversion issues. For example, in tf2onnx, various checkpoints of the t5 model had 65 HuggingFace conversion errors, 56 unsuccessful tf2onnx conversions, 1 unsuccessful ONNX Runtime load, 0 incorrect outputs, and 183 successes. We could not determine the cause(s) of this instability.

Synthetic Models: Synthetic models exhibit more difficulties and often show incorrect model behaviour.

Crashes: Synthetic models exhibit difficulties when converting to ONNX and loading converted models to the ONNX Runtime. We find 802/3,544 (~23%) *Unsuccessful Conversions*, almost entirely in synthetic models on tf2onnx. These crashes correspond to 4 unique crash locations. Of the 2,742/3,544 (~77%) synthetic models that successfully convert, only 1,244/2,742 (~45%) successfully load into ONNX Runtime. We observed 11 unique ONNX Runtime errors, of which 6 appear to correspond to open GitHub issues [3, 9, 20, 37, 60, 66]. We

Table 11: Results of conversion testing. Real models' conversion may fail in the HuggingFace wrapper. Both kinds of models may fail in DL model converter, or when the result is fed to ONNX Runtime (ORT). *Behavioural Difference:* **ORT inference results with difference** $> 10^{-7}$. **Real models fail rarely, synthetic models often.**

| Outcome | tf2onnx | | torch.onnx | |
	Real	Syn.	Real	Syn.
Start: Total models	1,761	1,820	1,761	1,724
Conv. Fail (HF)	456 (26%)	N/A	342 (20%)	N/A
Conv. Fail	65 (4%)	800 (44%)	20 (2%)	2 (1%)
ORT load Fail	19 (1%)	757 (42%)	27 (2%)	741 (42%)
Mismatch	9 (1%)	220 (12%)	11 (1%)	100 (6%)
Successful	1,212 (68%)	43 (2%)	1,361 (75%)	881 (51%)

disclosed the rest to the engineering teams. For most disclosures, we have not yet received a response. For one disclosure, which described a model that causes the libc++ abi to terminate unexpectedly during inference with ONNX Runtime, it had been (unintentionally) fixed in the development branch. The defect still affects older releases.

Behavioural Differences: We observed a large fraction of behavioural differences (incorrect output) with synthetic models. Compared to real models, which had 20 instances, synthetic models had 320 instances where the inference results exceeded the threshold. The majority of these instances were observed in the tf2onnx converter. For both converters, we disclosed such instances to the respective engineering teams (we have not heard a response yet). A summary of disclosed issues can be found in the artifact.

6.2.3 Method for Micro Analysis. To investigate the causes of the conversion failures we observed, we analyze mismatching models in terms of the operators used. We examine the individual operators and the sequences of operators. We compare these to non-failing models (for trends) as well as to the models in the converter test suites (for gaps in testing). Test suite models are collected from converters' CI/CD pipelines. Figure 8 illustrates our approach.

For operator types, we measure the operators present in each converted model, out of the set of available operators in the ONNX opset. For operator sequences, we extract the simple paths through each model. To illustrate, the ONNX model in Figure 2 shows three operators (ReduceMax_0, ArgMax, and ReduceMax) and two simple linear paths (ReduceMax_0→ArgMax and ReduceMax_0 → ReduceMax). Models with similar architectures may have simple paths that are largely identical, inflating the number of shared sequences. For example, the sequences aaaabd and aaaabc are identical except for the last element. To deal with cases we further reduce the common sequences to the smallest shared subsequences, *i.e.,* we recursively find the longest common subsequences for our sequences until we cannot find any smaller subsequences [14]. After reduction we find that sequences are between 3 to 5 operators long.

Evaluation Criteria for H_{RQ_4}: To evaluate the impact of *operator types*, we compare the operators of mismatched and correct models. If failing and correct models use different operators, we conclude that specific operators may cause failures. To evaluate the impact of *operator sequences*, we identify shared operator sequences belonging *only* to mismatched models, and their frequency. If mismatching models often share common simple paths of operators, that will support

H_{RQ_4}. H_{RQ_4} would be further strengthened if operator sequences are rarely shared with correct models.

6.2.4 Results for Micro Analysis. We focused on synthetic models that converted successfully but had mismatching behavior. There were 330 such models across TensorFlow and PyTorch. We compared them to correct models and models from converter test suites.

Incorrect conversions cannot be explained solely by operator types. Of the 154 ONNX operators, mismatched models contain 58 (`torch.onnx`) and 54 (`tf2onnx`) unique operators while correct models contain 59 (`torch.onnx`) and 52 (`tf2onnx`) unique operators. All but 1-2 operators are shared, indicating that for synthetic models, operator types do not predict mismatch.

In contrast, our analysis of shared operator sequences supports hypothesis H_{RQ_4}. As shown in Table 12 an overwhelming majority of mismatched models share unusual operator sequences. 312/320 of the mismatching models tested share at least one operator sequence that never occurred in the correct models. Further, we assess how often the sequences are shared by many models. We find that for `torch.onnx`, only 1 of the 131 sequences is shared by more than 3 models, and for `tf2onnx` no sequence is shared by more than 11 models. This indicates the failing models will often contain common operator patterns, suggesting families of sequences that cause errors. Finally, our comparison of test suite and mismatching models (③ in Table 12) shows that the failing models share few sequences with the models used in converter test suites (which are real models rather than synthetic ones), suggesting a gap in test coverage. Further analysis is left to future work.

7 Discussion and Future Work

7.1 Validating DL Model Converters

We find DL model converters are robust to new operator releases (§6.1). The weak correlation found between ONNX releases and issues implies that test suites are sufficient for catching issues that may
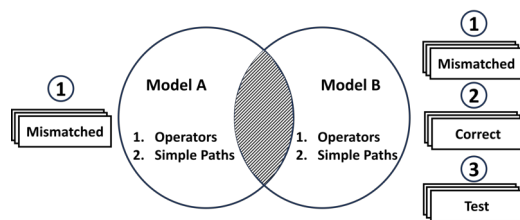

Figure 8: Micro analysis: Given two sets of models, we measure shared operators and simple paths between all model pairs (intersection). We compare three sets. First, the pairing ①×① compares mismatched models to themselves. Any operators and sequences shared by mismatched models may indicate shared causes of failure. Second, ①×② compares mismatched and correct models. Overlaps suggest a given operator or sequence is sometimes problematic, depending on context. Third, ①×③ compares mismatched models to those from converter test suites. Substantial *non*-overlap implies gaps in test coverage.

Table 12: Model sequence analysis results for the synthetic models that converted but mismatched ("Mismatch" in Table 11). ① : Sequences shared by mismatched models. ② : Sequences shared by mismatched and correct models; ③ : Sequences shared by mismatched and test suite models. ①\② : sequences present only in mismatched models. *Reduced*: smallest shared subsequences (§6.2.3).

Set	tf2onnx		torch.onnx	
	Unique	Reduced	Unique	Reduced
Total Models		220		100
① Mism. ∩ Mismatch	2,125	1,126	980	635
② Mism. ∩ Correct	1,050	508	4,243	2,988
③ Mism. ∩ Tests	35	35	2	2
①\②	1,527	862	176	131
# models with ①\②		216		96

come along with new ONNX releases — modulo the shortcomings of the approximations listed in §6.1.1.

Converter test suites share little in common with failing models, often missing critical model structures (§6.2.4). Specifically, **model converters need better testing techniques for behavioral changes**. Though this critical failure mode occurred in ~1/3 of historical failures (§5.2) and affected 6% of the models we tested (§6.2.2), the existing test suite models share little in common with models affected by this failure mode. For example, only 37 operator sequences are shared between mismatching models and test suite models (Table 12). Introducing **operator sequence coverage metrics may help improve the quality of test suites**, to promote the diversity of the models tested. These metrics could also be incorporated into fuzzers. Since some nodes or sequences are more likely to cause failures (Table 12), a coverage-guided fuzzing approach could help identify them.

Identifying behavioral changes after model conversion is critical, this is typically done with an end-to-end test. End-to-end testing carries with it the need to assess the outcome: is the converted model acceptable, and how much error is tolerable [19, 33]? Engineers currently use a variety of heuristic tolerances derived from experience (§6.2.1). Theoretical and empirical examination of the expected and acceptable tolerance would benefit DL model converter testing.

7.2 Comparison to Prior Studies

7.2.1 DL Model Converters. Openja *et al.* evaluated DL model converters by converting 5 models [55]. We extend their findings in two directions. First, we conducted a failure analysis (§5) to study *how* converters fail. Second, we convert models systematically (§6) and analyze them to understand *why* they fail. We considered both realistic and synthetic models. For realistic models, rather than picking 5 real-world models, we used 1,605 models from a large model corpus. For synthetic models, we adapted a DL model generation tool to conduct a bounded systematic exploration of converter behaviors. We omitted measurements of model size, adversarial robustness, and prediction accuracy, to focus instead on measuring the common failure modes (crashing and behavioral differences) identified in our failure analysis. Our analysis reveals that converters can successfully

convert many real model but synthetic models are more prone to failure, moreover analyzing the synthetic models reveals that particular operator sequences co-occur with failures.

7.2.2 DL Compilers.
Shen *et al.* studied failures in DL compilers [63], including the components that load models from DL frameworks and from common intermediary formats such as ONNX. Many of their results generalized to DL model converters: (1) we were able to adapt their taxonomies of failure symptoms and causes to DL model converters (§5.1.1, Table 7); and (2) as shown in Table 9, we found a similar distribution of failure symptoms for DL model converters. However, the causes of failures between the two contexts differed: as noted in §5.2.3, *Incompatibilities* were more common in DL model converters, while *Algorithmic Errors* were more common in DL compilers.

This causal asymmetry may be attributable to differences in the requirements of DL model converters and DL compilers. The purpose of DL model converters is *interoperability* (§2.1), making compatibility failures a focus and reducing the need for optimizations. Hardware-specific optimizations are left to the consumer of the model. Conversely, DL compilers must provide both compatibility and hardware-specific optimizations.

7.2.3 Methodological Innovations on the "Bug Study".
Our study began with a survey that offered valuable insights into the problem domain, guiding our design of the subsequent failure study, including the selection of the interoperability tool and the creation of a failure taxonomy (§5). This methodological progression, from survey to failure study, and then to hypothesis testing about the failures, diverges from the typical failure study in software engineering. Those studies typically focus on the "middle step" [5], understanding the root causes of defects and the conditions under which they manifest (*e.g.*, [32, 74]). Our hypothesis testing phase, in particular, adds understanding by providing an estimate of failure rates, an aspect seldom addressed in standard failure studies. We suggest that this holistic approach, integrating a survey, failure study, and hypothesis testing, may be a useful methodology for future failure studies.

8 Threats to Validity

We discuss three types of threats to validity [71]. Taking into consideration the criticisms of Verdecchia *et al.* [67], we focus on substantive threats that might influence our findings.

Construct Threats are potential limitations of how we operationalized concepts. We mitigated definitional concerns for RQ2 by adapting existing taxonomies in our failure analysis. We defined failures consistently with other works (GitHub issues closed with a repair), though we note there are other means of failure discovery [7]. To answer RQ3, we conservatively defined failure in terms of clear misbehaviors — when ONNX converter output is incompatible with ONNX Runtime, or the model's behavior changes. This may mask some "Crash" failures in the ONNX converter. In addition, different measures of behavioral difference are possible, such as the L1- or L2-distance. We used the measure recommended by PyTorch.

Internal threats are those that affect cause-effect relationships. We specifically allowed respondents to leave answers blank to improve the response rate. As a result, the results for interoperability tool use in RQ1 may be biased as only 48/92 respondents use an interoperability tool (§4.1.2). However, of the 48 respondents who reported using interoperability tools, 98% (47/48) gave analyzable data.

Our failure characterization in RQ2 was manual. We mitigated subjectivity via interrater agreement, on a pilot sample and a subsequent tranche during the full analysis (§5.1.2). In §6.1.1 we noted several approximations in our test of hypothesis H_{RQ_3}. We tested hypothesis H_{RQ_4} indirectly, using a frequency analysis of operators and sequences in mismatched models rather than directly testing the discovered subsequences.

External threats may impact generalizability. For RQ1, the surveyed population (HuggingFace users) may not fully reflect all ONNX users. However, it is an appropriate population, and we sampled at a confidence level of 90%. For RQ2-4, we examined two DL model converters for one interoperability framework (ONNX). Our results may not extend to other model converters nor other interoperability tasks. As mitigation, our data suggest that ONNX is the most popular DL interoperability tool. For ONNX, our results were similar across the two converters, suggested generalizability in two dimensions: DNN modeling approaches, and converter owners (§5.1.1). With regard to our methodology for RQ3-4, we generated synthetic models with the NNSmith tool. Other methods [28, 41, 68] might have different results.

9 Conclusion

DL model converters play a crucial role in providing interoperability between DL frameworks, registries, and runtimes. Understanding the nature of failures in DL model converters enables engineers to make informed decisions when relying on them. We conducted the first failure analysis of DL model converters, considering the PyTorch and TensorFlow converters for the popular ONNX intermediate representation. The most common symptoms of failure are crashes and, perhaps more concerningly, models that misbehave on certain inputs. Of the five stages of a typical model converter, one stage (node conversion) accounts for ~75% of the defects. A deductive description of the causes of erroneous converter behavior remains elusive — individual operators are not predictive of failure; sequences of operators may be correlated. Our findings suggest that in ONNX, engineers can rely on model converters but should validate the result for behavioral consistency. Through a mix of positive and negative results, we exposed several directions for further improvement of DL model converters. The main opportunities are new measurements: a new architectural coverage measure for a DL model converter, and a refined measure of tolerance after conversion.

10 Data-Availability Statement

This paper is accompanied by an artifact [2] to support replication and reproduction. The artifact includes the survey data for Theme 1, the failure analysis data for Theme 2, and the processed results of data associated with the hypothesis evaluation for Theme 3. The artifact also includes the modified NNSmith model generator, and the specific synthetic models used in the study. The source code for the artifact is available at https://github.com/PurdueDualityLab/issta24-onnx-artifact.

References

[1] 2019. ONNX | Home. https://onnx.ai/

[2] 2024. *Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters.* Zenodo. https://doi.org/10.5281/zenodo.12667479

[3] 12sf12. 2022. Name:'MatMul_32007' Status Message: matmul_helper.h:61 Compute MatMul dimension mismatch. https://github.com/microsoft/onnxruntime/issues/12594.

[4] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, and Harald Gall. 2019. Software Engineering for Machine Learning: A Case Study. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* https://doi.org/10.1109/ICSE-SEIP.2019.00042

[5] Paschal C. Amusuo, Aishwarya Sharma, Siddharth R. Rao, Abbey Vincent, and James C. Davis. 2022. Reflections on software failure analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* Association for Computing Machinery, New York, NY, USA, 1615–1620. https://doi.org/10.1145/3540250.3560879

[6] Dharun Anandayuvaraj and James C. Davis. 2023. Reflecting on Recurring Failures in IoT Development. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22).* Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3551349.3559545

[7] Jorge Aranda and Gina Venolia. 2009. The secret life of bugs: Going past the errors and omissions in software repositories. In *International Conference on Software Engineering (ICSE).* https://doi.org/10.1109/ICSE.2009.5070530

[8] Atılım Güneş Baydin, Barak A Pearlmutter, Alexey A Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of machine learning research* 18, 1 (2018), 5595–5637. http://jmlr.org/papers/v18/17-468.html

[9] BowenBao. 2023. [LocalFunction] Shape mismatch attempting to re-use buffer. https://github.com/microsoft/onnxruntime/issues/17061.

[10] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (oct 2021), 30 pages. https://doi.org/10.1145/3485500

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18).* USENIX Association, USA, 579–594.

[12] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104

[13] MLEAP Consortium. 2023. *EASA Research – Machine Learning Application Approval (MLEAP) interim technical report.* Horizon Europe research and innovation programme report. European Union Aviation Safety Agency.

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.

[15] James C Davis, Purvish Jajal, Wenxin Jiang, Taylor R Schorlemmer, Nicholas Synovic, and George K Thiruvathukal. 2023. Reusing deep learning models: Challenges and directions in software engineering. In *2023 IEEE John Vincent Atanasoff International Symposium on Modern Computing (JVA).* IEEE, 17–30.

[16] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. *Selecting Empirical Methods for Software Engineering Research.* Springer London, London, 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[17] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L Nord, and Ian Gorton. 2015. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 50–60.

[18] Hugging Face. 2021. Hugging Face – The AI community building the future. https://huggingface.co/.

[19] Hugging Face. 2023. Export to ONNX. https://huggingface.co/docs/transformers/serialization#validating-the-model-outputs.

[20] fatcat z. 2022. Resize op can't work well under Cubic mode with ORT 1.12. https://github.com/microsoft/onnxruntime/issues/12302.

[21] Norman Fenton and James Bieman. 2014. *Software Metrics: A Rigorous and Practical Approach, Third Edition* (3rd ed.). CRC Press, Inc., USA.

[22] Forbes. 2024. Hugging Face - Company Profile. https://www.forbes.com/companies/hugging-face/?sh=b2bcef56c38c.

[23] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20).* Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/3377811.3380397

[24] Gargi. 2024. The Power of Hugging Face AI. https://medium.com/@gargg/the-power-of-hugging-face-ai-4f6558ee0874.

[25] Adrien Gauffriau, Iryna De Albuquerque Silva, and Claire Pagetti. 2024. Formal description of ML models for unambiguous implementation. arXiv:2307.12713 [cs.NE] https://arxiv.org/abs/2307.12713

[26] Ahmad Nauman Ghazi, Kai Petersen, Sri Sai Vijay Raj Reddy, and Harini Nekkanti. 2019. Survey Research in Software Engineering: Problems and Mitigation Strategies. *IEEE Access* 7 (2019), 24703–24718. https://doi.org/10.1109/ACCESS.2018.2881041

[27] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386. https://doi.org/10.1002/rob.21918

[28] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22).* Association for Computing Machinery, New York, NY, USA, 1418–1430. https://doi.org/10.1145/3510003.3510092

[29] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2020. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19).* IEEE Press, 810–822. https://doi.org/10.1109/ASE.2019.00080

[30] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[31] Momoko Hattori, Shimpei Sawada, Shinichiro Hamaji, Masahiro Sakai, and Shunsuke Shimizu. 2020. Semi-static type, shape, and symbolic shape inference for dynamic computation graphs. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (London, UK) *(MAPL 2020).* Association for Computing Machinery, New York, NY, USA, 11–19. https://doi.org/10.1145/3394450.3397465

[32] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019).* Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[33] Wenxin Jiang, Vishnu Banna, Naveen Vivek, Abhinav Goel, Nicholas Synovic, George K. Thiruvathukal, and James C. Davis. 2024. Challenges and practices of deep learning model reengineering: A case study on computer vision. *Empirical Software Engineering* 29, 6 (Aug. 2024). https://doi.org/10.1007/s10664-024-10521-0

[34] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R. Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2023. An Empirical Study of Pre-Trained Model Reuse in the Hugging Face Deep Learning Model Registry. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23).* IEEE Press, 2463–2475. https://doi.org/10.1109/ICSE48619.2023.00206

[35] Wenxin Jiang, Jerin Yasmin, Jason Jones, Nicholas Synovic, Jiashen Kuo, Nathaniel Bielanski, Yuan Tian, George K Thiruvathukal, and James C Davis. 2024. Peatmoss: A dataset and initial analysis of pre-trained models in open-source software. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR).* IEEE, 431–443.

[36] J Jones, W Jiang, N Synovic, GK Thiruvathukal, and JC Davis. 2024. What do we know about Hugging Face? A systematic literature review and quantitative validation of qualitative claims. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) 2024.* ACM/IEEE.

[37] josephrocca. 2021. [wasm runtime] Could not find an implementation for ArgMax(12) node with name 'ArgMax_1382'. https://github.com/microsoft/onnxruntime/issues/9760.

[38] Mingyu Kim, Jihye Yun, Yongwon Cho, Keewon Shin, Ryoungwoo Jang, Hyun-jin Bae, and Namkug Kim. 2019. Deep Learning in Medical Imaging. *Neurospine* 16, 4 (Dec 2019), 657–668. https://doi.org/10.14245/ns.1938396.198

[39] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to advanced empirical software engineering.* Springer, 63–92. https://doi.org/10.1007/978-1-84800-044-5_3

[40] Nancy G. Leveson. 1995. *Safeware: System safety and computers.* ACM, New York, NY, USA.

[41] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen*, and Shing-Chi Cheung*. 2023. COMET: Coverage-guided Model Generation For Deep Learning Library Testing. *ACM Transactions on Software Engineering and Methodology* (Feb. 2023). https://doi.org/10.1145/3583566

[42] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727. https://doi.org/10.1109/TPDS.2020.3030548

[43] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep

Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[44] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In *Proceedings of the 28th ACM Joint Meeting on ESEC/FSE*. ACM, Virtual Event USA, 1320–1330. https://doi.org/10.1145/3368089.3417051

[45] Guoming Long and Tao Chen. 2022. On Reporting Performance and Accuracy Bugs for Deep Learning Frameworks: An Exploratory Study from GitHub. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering* (Gothenburg, Sweden) *(EASE '22)*. Association for Computing Machinery, New York, NY, USA, 90–99. https://doi.org/10.1145/3530019.3530029

[46] Guodong Long and Ting Chen. 2022. On Reporting Performance and Accuracy Bugs for Deep Learning Frameworks: An Exploratory Study from GitHub. *IEEE Transactions on Software Engineering* (Apr 2022). https://arxiv.org/abs/2204.04542

[47] Nikolaos Louloudakis, Perry Gibson, José Cano, and Ajitha Rajan. 2023. DeltaNN: Assessing the Impact of Computational Environment Parameters on the Performance of Image Recognition Models. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 414–424. https://doi.org/10.1109/ICSME58846.2023.00054

[48] N. Louloudakis, P. Gibson, J. Cano, and A. Rajan. 2023. Fault Localization for Buggy Deep Learning Framework Conversions in Image Recognition. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1795–1799. https://doi.org/10.1109/ASE56229.2023.00147

[49] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[50] William M McKeeman. 1998. Differential testing for software. https://www.cs.swarthmore.edu/~bylvisa1/cs97/f13/Papers/DifferentialTestingForSoftware.pdf. *Digital Technical Journal* 10, 1 (1998), 100–107.

[51] Louis G Michael, James Donohue, James C Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 415–426. https://doi.org/10.1109/ASE.2019.00047

[52] Mohammad Mehdi Morovati, Amin Nikanjam, Florian Tambon, Foutse Khomh, and Zhen Ming Jiang. 2024. Bug characterization in machine learning-based systems. *Empirical Software Engineering* 29, 1 (2024), 14.

[53] ONNX. 2022. Open Neural Network Exchange Intermediate Representation (ONNX IR) Specification . https://github.com/onnx/onnx/blob/ee7d2cdfa34b8b3c7e0b68b70daf72aaa48c23ac/docs/IR.md.

[54] ONNX. 2023. ONNX Meeting - Thursday, July 13th, 2023 at 9:00am PST. https://github.com/onnx/sigs/blob/main/operators/meetings/041-20230713.md.

[55] Moses Openja, Amin Nikanjam, Ahmed Haj Yahmed, Foutse Khomh, and Zhen Ming Jack Jiang. 2022. An Empirical Study of Challenges in Converting Deep Learning Models. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 13–23. https://doi.org/10.1109/ICSME55016.2022.00010

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[57] Henry Petroski et al. 1994. Design paradigms: Case histories of error and judgment in engineering. https://www.cambridge.org/core/books/design-paradigms/92832B6D5EF85B08B890DED83DDBAF57.

[58] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis

of Variance. In *International Conference on Automated Software Engineering (ASE)*. 771–783. https://doi.org/10.1145/3324884.3416545

[59] PyTorch. 2023. verification.py. https://github.com/pytorch/pytorch/blob/869e52e3dd211d4770ab38f621b906b23fae0132/torch/onnx/verification.py#L256.

[60] rafaelagrc. 2022. Incompatible dimensions for matrix multiplication Error in StarNet model when doing InferenceSession. https://github.com/microsoft/onnxruntime/issues/11846.

[61] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2022. An empirical study of bugs in webassembly compilers. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 42–54. https://doi.org/10.1109/ASE51524.2021.9678776

[62] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2024. "We Have No Idea How Models will Behave in Production until Production": How Engineers Operationalize Machine Learning. *Proceedings of the ACM on Human-Computer Interaction* 8, CSCW1 (April 2024), 1–34. https://doi.org/10.1145/3653697

[63] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[64] Ayush Shridhar, Phil Tomson, and Mike Innes. 2020. Interoperating Deep Learning models with ONNX. jl. In *Proceedings of the JuliaCon Conferences*, Vol. 1. 59.

[65] Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. 2022. An exploratory study of deep learning supply chain. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 86–98. https://doi.org/10.1145/3510003.3510199

[66] vbogach. 2022. [ONNX] Scripted reshape incorrect if shape is dynamically calculated. https://github.com/pytorch/pytorch/issues/78721.

[67] Roberto Verdecchia, Emelie Engström, Patricia Lago, Per Runeson, and Qunying Song. 2023. Threats to validity in software engineering research: A critical reflection. *Information and Software Technology* 164 (2023), 107329.

[68] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. https://doi.org/10.1145/3368089.3409761

[69] Peter Wegner. 1996. Interoperability. *ACM CSUR* 28, 1 (Mar 1996), 285–287. https://doi.org/10.1145/234313.234424

[70] David Wicks. 2017. The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal* 12, 2 (2017), 169–170.

[71] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[72] Dongwei Xiao, Zhibo LIU, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 15 (feb 2022), 28 pages. https://doi.org/10.1145/3508035

[73] YsYusaito. 2022. Inference result is different between Pytorch and ONNX model. https://github.com/pytorch/pytorch/issues/74732.

[74] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *International Symposium on Software Reliability Engineering (ISSRE)*.