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

Anonymous Author(s)

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 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 would be fruitful.

1 INTRODUCTION

Deep Learning (DL) achieves state-of-the-art performance in many domains [24, 36]. Software engineers engage in many activities for deep learning, including developing, re-using, fine-tuning, and deploying DL models [3, 27, 31, 33]. They use tools at each stage: DL frameworks for development (*e.g.*, PyTorch [53]); DL model registries for re-use (*e.g.*, HuggingFace [16]), 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 [52]. 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 [10, 29, 43, 55, 62] and DL deployment compilers and runtimes [26, 59]. In contrast, prior research on DL model converters is limited to measuring conversions of 5 DL models [52]. 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=108), 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 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.

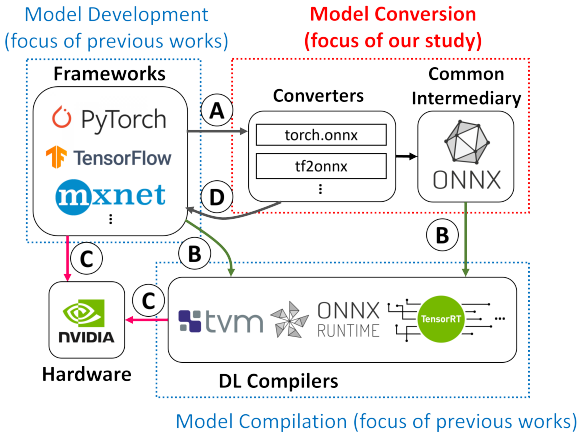

Figure 1: Paths from model development to deployment on hardware. Model interoperability facilitates reuse across frameworks and deployment environments. (A) represents *model conversion* to a common intermediary. (B) represents *compilation*. (C) represents *model deployment*. (D) represents *model conversion* to a framework.

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.

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).

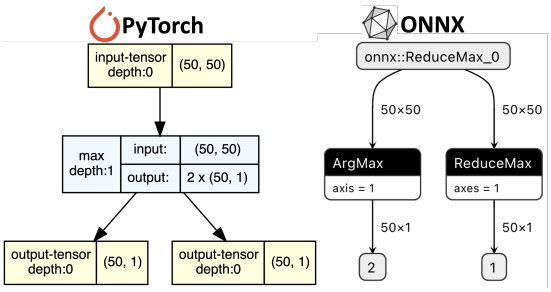

Figure 2: PyTorch model converted to ONNX Intermediate Representation. The PyTorch model calculates the per-row maximum using `torch.max`. In ONNX, this uses the operators `ArgMax` plus `ReduceMax`.

2 BACKGROUND AND RELATED WORK

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

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)* [42]. Wegner describes two patterns for interoperability: creating *pairwise mappings* between systems; and introducing a *common intermediary* understood by all participants [65]. 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 [42] 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 [39]. 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 [23, 51] 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 [63]. For a *behavioral* issue, in PyTorch #74732 a converted ONNX model's prediction changed [68].

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 [32]. 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 [4, 6, 54]. 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 [10]. Shen *et al.* studied DL compiler defects [59], and others have tested DL compilers [41, 67]. Some of this work has incidentally examined interoperability failures [41, 48]; 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 [59];

The two prior failure studies of DL model converters focused on testing and fault localization. Openja *et al.* converted 5 popular DL models [52]. They considered only the current state of model converters, not past failures. Louloudakis *et al.* studied behavioral issues resulting from framework-to-framework conversion [44]. They found failures in 10 out of 36 conversions. They created a fault localization and repair pipeline to localize and fix discrepancies [45].

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.

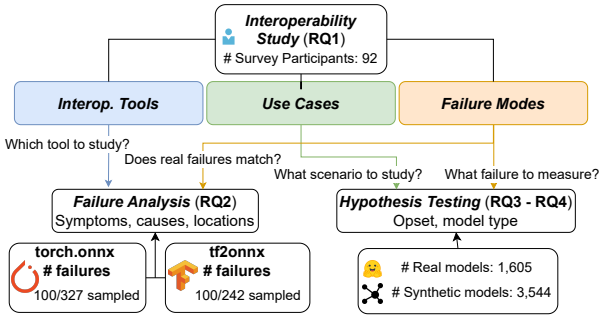

Figure 3: Goal, research questions, methods, and data sources.

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).

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 RQ3-5 (§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 [15]. Method is in §4.1, results in §4.2.

4.1 Methods

4.1.1 Instrument Design. We followed Kitchenham & Pfleeger's guidelines to develop our survey instrument [37]. Table 2 illustrates the result. We asked general questions about DL interoperability, and specific questions about ONNX (interview data [58] and GitHub suggest it is most popular). For higher response rate, the survey was short and mostly closed-ended. 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.

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?

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 re-use [34], so its users are a suitable population for questions about 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 [19, 22], 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).

Type	Break-down
ML Exp.	<1 yr. (8) ; 1-2 yr. (17) ; 3-5 yr. (32) ; >5 yr. (32)
SE Exp.	<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 Env.	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 Results

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%)

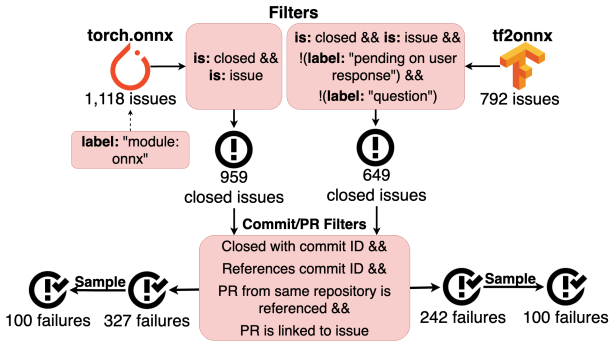

Figure 4: Filtering of issues for each repository studied. *Filters* are using GitHub search predicates. *Commit/PR filters* are applied to issue timeline events. Data were collected on Jan. 6, 2023. 100 issues per repository were analyzed.

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, 38, 54], 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 [50]. 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` [50]. The ONNX specification changes regularly to keep up with DL framework evolution [61]. 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 Methods

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

(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).

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

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, 28]. 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: 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 [20, 21, 30], 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 [59].

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.

Table 7: Taxonomy of failure causes, adapted from [60], 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.

Symptoms & Causes: We adapted taxonomies from Shen *et al.* [59], the closest related work. As that work concerns DL compilers rather than model 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" [46] 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 [66], 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.

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`.

5.2.2 Failure Symptoms. The distributions of symptoms for both `tf2onnx` and PyTorch are similar — see Table 9. The most common

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%)*

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.* [59]). 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.* [59]. Percentages are rounded.

Symptom	TF	PT	Total	DL Comp. [59]
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%)*

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.

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*

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*:

Does ONNX evolution affect converter failure rates? **RQ3:** Does ONNX evolution affect converter failure rates? We hypothesize $\boldsymbol{H}_{RQ_3}$*: 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 (§4, 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* (§5.2, 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 $\boldsymbol{H}_{RQ_4}$*: 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 using pure tensor operations, implying that conversion may fail due to the presence 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 $\boldsymbol{H}_{RQ_4}$. 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 $\mathbf{H}_{RQ_3}$: 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).

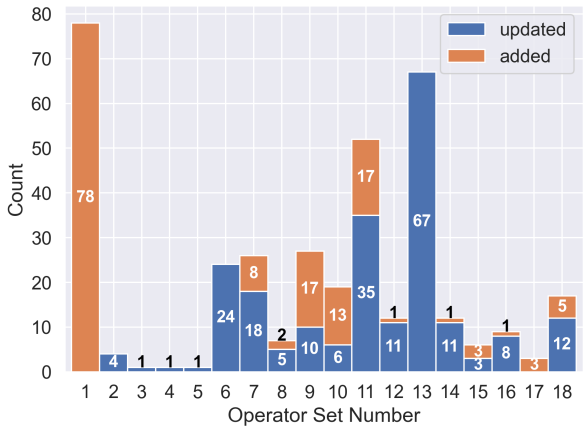

Figure 5: Additions and updates of operators by ONNX operator set version, from version 1 (2017)—version 18 (2022). Version size := sum.

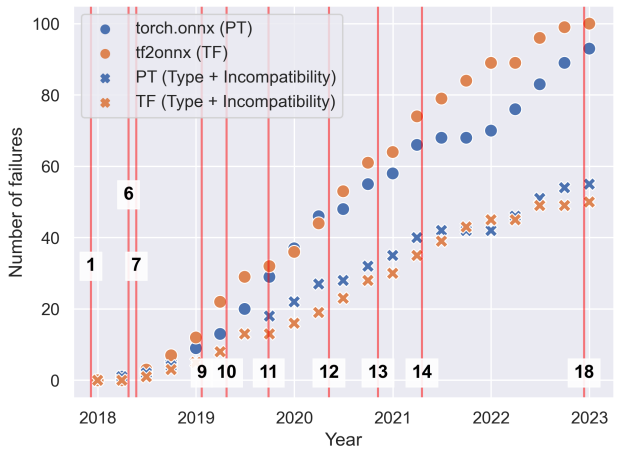

Figure 6: Cumulative number of failures in the `torch.onnx` and `tf2onnx` converters, plotted quarterly from 2018-2023. The gap between two points is the number of newly opened issues during that time period. The O's track all failures. The X's track the subset of Incompatibility and Type Problem failures.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

6.1.1 Method. 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 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 reuse, *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 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 [18].

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).

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 [47].

(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. We filtered for the 1,761 models with both PyTorch and TensorFlow versions, allowing

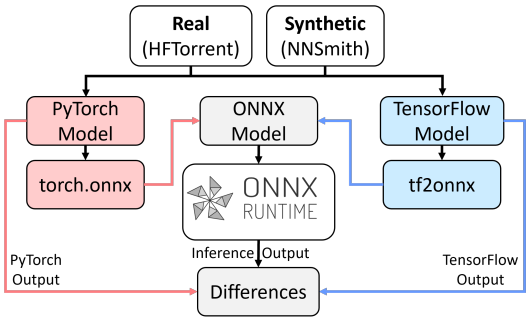

Figure 7: Macro analysis: *Real* and *Synthetic* models are converted and then differences are measured. For synthetic models, we used `tf2onnx` or `torch.onnx` directly. For real models we used HuggingFace's converter, which does preprocessing and then calls those converters.

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 [41]. `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) 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 [25], COMET [40], and LEMON [64]. 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* $< 10^{-7}$ [56].

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} [41]. Openja *et al.* [52] 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.

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%)

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 [2, 9, 35, 57, 63, 69]. We 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, the response was that it had been (unintentionally) fixed in the development branch. It 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

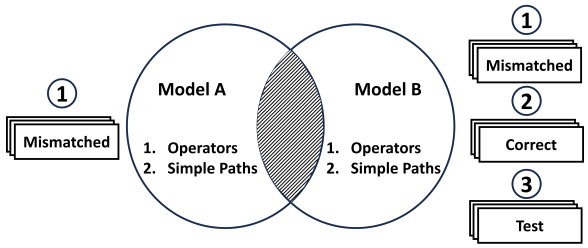

Figure 8: Micro analysis: Given two sets of models, we measure shared operators and simple paths between all model pairs (intersection). We compare three sets. First, the pairing ①×① compares mismatched models to themselves. Any operators and sequences shared by mismatched models may indicate shared 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.

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

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

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 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.

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 [17, 32]? 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 [52]. 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 [59], 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" [4], understanding the root causes of defects and the conditions under which they manifest (*e.g.*, [30, 70, 71]). 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

Construct: 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: 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 rather than directly testing the discovered subsequences.

External: 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 [25, 40, 64] 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

An anonymized artifact is at https://www.github.com/asub0/fonnx. It has survey data, the failure analysis data, and the processed results of data associated with the hypothesis evaluation. It also has some tables and figures omitted for space. It omits raw synthetic models due to storage limits. Data published on acceptance.

REFERENCES

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

[2] 12sf12 (2022) Name:'matmul_32007' status message: matmul_helper.h:61 compute matmul dimension mismatch. https://github.com/microsoft/onnxruntime/issues/12594

[3] Amershi S, Begel A, Bird C, DeLine R, Gall H (2019) Software Engineering for Machine Learning: A Case Study. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

[4] Amusuo P, Sharma A, Rao SR, Vincent A, Davis JC (2022) Reflections on software failure analysis. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering — Ideas, Visions, and Reflections track (ESEC/FSE-IVR)

[5] Amusuo PC, Sharma A, Rao SR, Vincent A, Davis JC (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, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 1615–1620, DOI 10.1145/3540250.3560879, URL https://dl.acm.org/doi/10.1145/3540250.3560879

[6] Anandayuvaraj D, Davis JC (2022) Reflecting on Recurring Failures in IoT Development. In: Automated Software Engineering: New Ideas and Emerging Results (ASE-NIER)

[7] Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: International Conference on Software Engineering (ICSE)

[8] Baydin AG, Pearlmutter BA, Radul AA, Siskind JM (2018) Automatic differentiation in machine learning: a survey. Journal of machine learning research 18(1):5595–5637

[9] BowenBao (2023) [localfunction] shape mismatch attempting to re-use buffer. https://github.com/microsoft/onnxruntime/issues/17061

[10] Chen J, Liang Y, Shen Q, Jiang J (2022) Toward Understanding Deep Learning Framework Bugs. URL http://arxiv.org/abs/2203.04026

[11] Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Cowan M, Shen H, Wang L, Hu Y, Ceze L, et al. (2018) Tvm: An automated end-to-end optimizing compiler for deep learning. arXiv preprint arXiv:180204799

[12] Cohen J (1960) A coefficient of agreement for nominal scales. Educational and psychological measurement 20(1):37–46

[13] Consortium M (2023) Easa research – machine learning application approval (mleap) interim technical report. Horizon europe research and innovation programme report, European Union Aviation Safety Agency

[14] Cormen TH, Leiserson CE, Rivest RL, Stein C (2022) Introduction to algorithms. MIT press

[15] Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. Guide to advanced empirical software engineering pp 285–311

[16] Face H (2021) Hugging Face – The AI community building the future. https://huggingface.co/

[17] Face H (2023) Export to onnx. https://huggingface.co/docs/transformers/serialization#validating-the-model-outputs

[18] Fenton N, Bieman J (2014) Software Metrics: A Rigorous and Practical Approach, Third Edition, 3rd edn. CRC Press, Inc., USA

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

[20] Franco AD, Guo H, Rubio-González C (2017) A Comprehensive Study of Real-World Numerical Bug Characteristics. Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering pp 509–519, URL https://ieeexplore.ieee.org/abstract/document/8115662, iSBN: 978-1-5386-2684-9

[21] Garcia J, Feng Y, Shen J, Almanee S, Xia Y, Chen aQA (2020) A comprehensive study of autonomous vehicle bugs. In: International Conference on Software Engineering (ICSE), IEEE, Seoul, Korea (South), pp 385–396, URL https://dl.acm.org/doi/10.1145/3377811.3380397

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

[23] Gauffriau A, Pagetti C (2023) Formal description of ml models for unambiguous implementation. arXiv preprint arXiv:230712713

[24] Grigorescu S, Trasnea B, Cocias T, Macesanu G (2020) A survey of deep learning techniques for autonomous driving. Journal of Field Robotics 37(3):362–386, DOI 10.1002/rob.21918

[25] Gu J, Luo X, Zhou Y, Wang X (2022) Muffin: testing deep learning libraries via neural architecture fuzzing. In: Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '22, p 1418–1430, DOI 10.1145/3510003.3510092, URL https://dl.acm.org/doi/10.1145/3510003.3510092

[26] Guo Q, Chen S, Xie X, Ma L, Hu Q, Liu H, Liu Y, Zhao J, Li X (2019) An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 810–822, DOI 10.1109/ASE.2019.00080

[27] Han X, Zhang Z, Ding N, Gu Y, Liu X, Huo Y, Qiu J, Yao Y, Zhang A, Zhang L, Han W, Huang M, Jin Q, Lan Y, Liu Y, Liu Z, Lu Z, Qiu X, Song R, Tang J, Wen JR, Yuan J, Zhao WX, Zhu J (2021) Pre-trained models: Past, present and future. AI Open 2:225–250

[28] Hattori M, Sawada S, Hamaji S, Sakai M, Shimizu S (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, pp 11–19

[29] Islam MJ, Nguyen G, Pan R, Rajan H (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, pp 510–520

[30] Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

[31] Jiang W, Banna V, Vivek N, Goel A, Synovic N, Thiruvathukal GK, Davis JC (2023) Challenges and practices of deep learning model reengineering: A case study on computer vision (arXiv:2303.07476), DOI 10.48550/arXiv.2303.07476, URL http://arxiv.org/abs/2303.07476, arXiv:2303.07476 [cs]

[32] Jiang W, Banna V, Vivek N, Goel A, Synovic N, Thiruvathukal GK, Davis JC (2023) Challenges and practices of deep learning model reengineering: A case study on computer vision. arXiv DOI 10.48550/arxiv.2303.07476, https://arxiv.org/abs/2303.07476

[33] Jiang W, Synovic N, Hyatt M, Schorlemmer TR, Sethi R, Lu YH, Thiruvathukal GK, Davis JC (2023) An Empirical Study of Pre-Trained Model Reuse in the Hugging Face Deep Learning Model Registry. In: International Conference on Software Engineering (ICSE)

[34] Jiang W, Synovic N, Hyatt M, Schorlemmer TR, Sethi R, Lu YH, Thiruvathukal GK, Davis JC (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 (ICSE'23)

[35] josephrocca (2021) [wasm runtime] could not find an implementation for argmax(12) node with name 'argmax_1382'. https://github.com/microsoft/onnxruntime/issues/9760

[36] Kim M, Yun J, Cho Y, Shin K, Jang R, Bae Hj, Kim N (2019) Deep learning in medical imaging. Neurospine 16(4):657–668, DOI 10.14245/ns.1938396.198

[37] Kitchenham BA, Pfleeger SL (2008) Personal opinion surveys. In: Guide to advanced empirical software engineering, Springer, pp 63–92

[38] Leveson NG (1995) Safeware: System safety and computers. ACM, New York, NY, USA

[39] Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Gan L, Yang G, Qian D (2020) The deep learning compiler: A comprehensive survey. IEEE Transactions on Parallel and Distributed Systems 32(3):708–727

[40] Li M, Cao J, Tian Y, Li TO, Wen* M, Cheung* SC (2023) Comet: Coverage-guided model generation for deep learning library testing. ACM Transactions on Software Engineering and Methodology DOI 10.1145/3583566, URL https://dl.acm.org/doi/10.1145/3583566, just Accepted

[41] Liu J, Lin J, Ruffy F, Tan C, Li J, Panda A, Zhang L (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, pp 530–543

[42] Liu Y, Chen C, Zhang R, Qin T, Ji X, Lin H, Yang M (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, p 1320–1330, DOI 10.1145/3368089.3417051, URL https://dl.acm.org/doi/10.1145/3368089.3417051

[43] Long G, Chen T (2022) On reporting performance and accuracy bugs for deep learning frameworks: An exploratory study from github. IEEE Transactions on Software Engineering URL https://arxiv.org/abs/2204.04542

[44] Louloudakis N, Gibson P, Cano J, Rajan A (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), IEEE, pp 414–424

[45] Louloudakis N, Gibson P, Cano J, Rajan A (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, pp 1795–1799

[46] McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia medica 22(3):276–282

[47] McKeeman WM (1998) Differential Testing for Software 10(1):8

[48] Morovati MM, Nikanjam A, Tambon F, Khomh F, Jiang ZM (2024) Bug characterization in machine learning-based systems. Empirical Software Engineering 29(1):14

[49] Norman D (2013) The design of everyday things: Revised and expanded edition. Basic Books, URL https://books.google.com/books?id=qBfRDQAAQBAJ, citation Key: norman2013design tex.lccn: 2013024417

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

[51] ONNX (2023) Onnx meeting - thursday, july 13th, 2023 at 9:00am pst. https://github.com/onnx/sigs/blob/main/operators/meetings/041-20230713.md

[52] Openja M, Nikanjam A, Yahmed AH, Khomh F, Ming Z, Jiang (2022) An Empirical Study of Challenges in Converting Deep Learning Models. IEEE Transactions on Software Engineering DOI 10.48550/arXiv.2206.14322

[53] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. (2019) Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems 32

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

[55] Pham HV, Qian S, Wang J, Lutellier T, Rosenthal J, Tan L, Yu Y, Nagappan N (2020) Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In: International Conference on Automated Software Engineering (ASE), pp 771–783, DOI 10.1145/3324884.3416545, URL https://dl.acm.org/doi/10.1145/3324884.3416545

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

[57] rafaelagrc (2022) Incompatible dimensions for matrix multiplication error in starnet model when doing inferencesession. https://github.com/microsoft/onnxruntime/issues/11846

[58] Shankar S, Garcia R, Hellerstein JM, Parameswaran AG (2024) " we have no idea how models will behave in production until production": How engineers operationalize machine learning. arXiv preprint arXiv:240316795

[59] Shen Q, Ma H, Chen J, Tian Y, Cheung SC, Chen X (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, ACM, Athens Greece, pp 968–980, DOI 10.1145/3468264.3468591

[60] Shen Q, Ma H, Chen J, Tian Y, Cheung SC, Chen X (2021) A comprehensive study of deep learning compiler bugs. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

[61] Shridhar A, Tomson P, Innes M (2020) Interoperating deep learning models with onnx. jl. In: Proceedings of the JuliaCon Conferences, vol 1, p 59

[62] Tan X, Gao K, Zhou M, Zhang L (2022) An exploratory study of deep learning supply chain. In: International Conference on Software Engineering (ICSE), Pittsburgh Pennsylvania

[63] vbogach (2022) [onnx] scripted reshape incorrect if shape is dynamically calculated. https://github.com/pytorch/pytorch/issues/78721

[64] Wang Z, Yan M, Chen J, Liu S, Zhang D (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, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2020, p 788–799, DOI 10.1145/3368089.3409761, URL https://dl.acm.org/doi/10.1145/3368089.3409761

[65] Wegner P (1996) Interoperability. ACM CSUR 28(1):285–287, DOI 10.1145/234313.234424, URL https://dl.acm.org/doi/10.1145/234313.234424

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

[67] Xiao D, Liu Z, Yuan Y, Pang Q, Wang S (2022) Metamorphic testing of deep learning compilers. Proceedings of the ACM on Measurement and Analysis of Computing Systems 6(1):1–28, DOI 10.1145/3508035

[68] YsYusaito (2022) Inference result is different between pytorch and onnx model. https://github.com/pytorch/pytorch/issues/74732

[69] fatcat z (2022) Resize op can't work well under cubic mode with ort 1.12. https://github.com/microsoft/onnxruntime/issues/12302

[70] Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An Empirical Study of Common Challenges in Developing Deep Learning Applications. In: International Symposium on Software Reliability Engineering (ISSRE)

[71] Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on TensorFlow program bugs. International Symposium on Software Testing and Analysis (ISSTA)

A SUMMARY OF APPENDICES

- Appendix A: Appendix Summary.
- Appendix B: Taxonomies and Extended Tables from §5.
- Appendix C: Additional Tables from §6.

B THEME 2 – FAILURE ANALYSIS (OF ONNX)

B.1 Taxonomies

- Table 13: Adapted taxonomy of failure symptoms.
- Table 17: Adapted taxonomy of failure causes.

B.2 Symptom and Cause Data

- Table 14: Distribution of causes in `tf2onnx` and `torch.onnx`.
- Table 16: Joint distribution of causes and symptoms (unabridged).

C THEME 3 – DEEPER CAUSES

Table 15 contains results of model operator analysis as described in §6.2.3.

Table 13: Adapted taxonomy of failure symptoms [59]. *Emphasis*: wording changed. *Hang* symptom omitted (unobserved). This is a tabular version of symptoms presented in §5.1.

Symptom	Definition
Crash	The *model converter* terminates unexpectedly during *conversion*, usually producing an error message.
Wrong *Model*	The *model converter* behaves unexpectedly without crashing, producing a wrong intermediate or final result.
Bad Performance	The time cost or memory consumption of the *model converter* is much larger than developer/user expectations (*e.g.*, the performance achieved on the previous version during regression testing or performance achieved by certain hardware).
Build Failure	Installation of *model converter* or dependencies fails.
Unreported	The symptom cannot be identified by analyzing the corresponding issues including code changes, discussions, and related issues.
Hang	This symptom means that the *converter* keeps running for a long period of time without producing the expected result.

Table 14: Distribution of causes in `tf2onnx` and `torch.onnx`. The top-5 causes in terms of frequency are shown. The *Others* are: *Concurrency*, *Documentation*, *Incorrect Numerical Computation*, *Misconfiguration*, *Testing*, *Incorrect Exception Handling*, *Typos*, and *Incorrect Assignment* (each has < 3% representation). The final column shows the cause distribution from Shen *et al.* [59]. Note the larger proportion of *Incompatibility* failures in DL model converters (first row).

Causes		TF	PT	Total	Comp. to DL compilers [59]
Incompatibility	External	**23**	**32**	**55 (28%)**	16 (4%)
	Internal	2	0	2 (1%)	5 (1%)
	Resource	0	0	0 (0%)	4 (1%)
Type Problem	Node	**21**	**25**	**46 (23%)**	13 (4%)
	Conventional	3	2	5 (3%)	20 (6%)
	Tensor	1	2	3 (2%)	42 (12%)
Algorithmic Error		**18**	**6**	**24 (12%)**	59 (16%)
Shape Problem		9	12	21 (11%)	67 (19%)
API Misuse		6	6	12 (6%)	35 (10%)
Others		17	15	32 (16%)	98 (27%)
Total		*100*	*100*	*200 (100%)*	*359 (100%)*

Table 15: Model Operator Analysis Results. As described in §6.2.3: Mismatched ①, all operators that mismatched models contain; Correct ② all operators that correct models contain; Test Suite ③ all operators that test suites contain. ①\② are all operators in ① but not in ②. ②\① are all operators in ② but not in ①. ①\③ are all operators in ① but not in ③. ③\① are all operators in ③ but not in ①.

Model Type	`tf2onnx`	`torch.onnx`
Mismatched ①	54	58
Correct ②	52	59
Test Suite ③	54	16
①\②	0	1
②\①	2	0
①\③	21	46
③\①	21	4

Table 16: Unabridged joint distribution of 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`. Table 10 is derived from this by binning the top-5 causes in terms of frequency, with the rest binned as *Other*. The *Others* are *Concurrency, Docs, Incorrect Numerical Computation, Misconfig, Testing, Incorrect Exception Handling, Typos*, and *Incorrect Assignment*. Rare symptoms are likewise binned as *Other*.

Causes \ Symptoms	Crash		Wrong Model		Bad Performance		Build Failure		Unreported		Total	
	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF	PT	TF
API Misuse	5	6	1	0	0	0	0	0	0	0	*6*	*6*
Incompatibility	**28**	**19**	3	4	0	0	0	0	1	2	*32*	*25*
Type Problem	**14**	**8**	**13**	**17**	1	0	0	0	1	0	*29*	*25*
Shape Problem	4	5	7	4	0	0	0	0	1	0	*12*	*9*
Incorrect Numerical Computation	0	0	0	0	0	0	0	0	0	0	*0*	*0*
Incorrect Assignment	2	0	0	0	0	0	0	0	0	0	*2*	*0*
Incorrect Exception Handling	1	1	1	0	0	0	0	0	0	0	*2*	*1*
Misconfiguration	0	1	0	0	0	0	2	2	0	1	*2*	*4*
Algorithmic Error	3	4	**3**	**10**	0	2	0	0	0	2	*6*	*18*
Typo	2	0	0	0	0	0	0	1	0	0	*2*	*1*
Testing	1	0	1	0	0	0	0	0	1	1	*3*	*1*
Others	2	6	1	0	0	0	0	0	1	4	*4*	*10*
Total	*62*	*50*	*30*	*35*	*1*	*2*	*2*	*3*	*5*	*10*	*100*	*100*

Table 17: Adapted taxonomy of failure causes [59]. *API Misuse, Type Problem*, and *Algorithmic Error* are high-level categories with sub-causes underneath them. This table is an unabridged version of Table 7.

Causes	Definition
API Misuse	High-level category that encapsulates *Wrong API*, *Condition missing/redundancy*, and *API missing/redundancy*.
Wrong API	The developer or user uses the wrong API or wrong arguments in an API.
Condition missing/redundancy	Developer fails to use (or redundantly uses) a condition check for an API.
API missing/redundancy	Developers fail to use (or redundantly use) an API.
Incompatibility	High-level category that encapsulates *Internal*, *External*, and *Resource*.
Internal	There are API compatibility issues within a model converter caused by API evolution.
External	There are API compatibility issues between a model converter and third-party libraries (e.g., ONNX, TensorFlow, ONNX Runtime).
Resource	There are compatibility issues between model converters and external resources. The characteristics of the target device are incompatible with the model converter.
Type Problem	
Node	A model converter works on a computational graph, where nodes represent the atomic DL operators (such as convolution and pooling) and edges represent the tensors. Each node takes zero or more tensors as input and produces a tensor as output.
Tensor	This relates to the types of tensors. Specifically, a tensor is a multi-dimensional matrix containing elements of a single data type.
Conventional	There are also conventional variables widely used in the development of traditional software systems. This subcategory refers to the problem involving the types of conventional variables.
Shape Problem	This cause covers, broadly, the shape of inputs and outputs a node can have. These occur during the operation of shape matching, shape transformation, shape inference, layout transformation, etc.
Tensor Shape Problem	This is related to tensor shape or layout. The tensor shape is the number of elements in each dimension. Layout describes how the shape is represented in memory.
Incorrect Numerical Computation	This involves incorrect numerical computations, values, or usage, such as incorrect operators or operands, dividing by 0, and missing or redundant operands.
Incorrect Assignment	This involves a variable being incorrectly initialized or assigned, or a variable lack initialization.
Incorrect Exception Handling	This category occurs due to incorrect exception handling. For example, an exception is not thrown when it should be, an example is thrown when it should be, or an incorrect/imprecise exception message is thrown.
Misconfiguration	This category occurs due to misconfiguration in the model converter. This includes dependency versioning and misconfigured environments.
Concurrency	This is a result of incorrect operations on concurrency-oriented structures (e.g., locks, threads, and critical regions).
Algorithmic Error	High-level category that encapsulates *Incorrect Optimization* and *Incorrect Tracing*.
Incorrect Optimization	There is an issue with optimizations that occur in the model converter. Examples: incorrect fusing, dead-code elimination, et cetera.
Incorrect Tracing	There is an issue with the tracing of DL models. This occurs in frameworks that use dynamic models. Dynamic models need to be "traced" to find the nodes in the computational graph prior to the conversion.
Typo	This is due to trivial mistakes (e.g. slips [49]) by developers, e.g., "default" is mistakenly written as "defualt".
Testing	This class contains issues that are a result of incorrect, outdated, or missing documentation.
Documentation	This class contains issues related to software tests such as unit tests. This covers "flaky" tests, missing tests, broken tests, or the addition of new tests.
Others	This cause is relegated to causes that occur infrequently and do not belong to the other causes.