Characterizing the Complexity and Its Impact on Testing in ML-Enabled Systems

A Case Sutdy on Rasa

Junming Cao*, Bihuan Chen*, Longjie Hu*, Jie Gao†, Kaifeng Huang*, Xuezhi Song*, Xin Peng*
*School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
†Singapore University of Technology and Design, Singapore

***Abstract*—Machine learning (ML) enabled systems are emerging with recent breakthroughs in ML. A model-centric view is widely taken by the literature to focus only on the analysis of ML models. However, only a small body of work takes a system view that looks at how ML components work with the system and how they affect software engineering for ML-enabled systems. In this paper, we adopt this system view, and conduct a case study on Rasa 3.0, an industrial dialogue system that has been widely adopted by various companies around the world. Our goal is to characterize the complexity of such a large-scale ML-enabled system and to understand the impact of the complexity on testing. Our study reveals practical implications for software engineering for ML-enabled systems.**

I. INTRODUCTION

The recent advances in machine learning (ML) have attracted an increasing interest in applying ML across a breadth of business domains, e.g., self-driving cars, virtual assistants, robotics, and health care. According to the Global AI Adoption Index by IBM [36], 35% of companies around the world have deployed AI in their business, while 42% of companies are exploring AI. Such a trend has caused the emergence of ML-enabled systems which are composed of ML and non-ML components. ML components are often important, but usually only a part of many components in ML-enabled systems [41].

The previous research on software engineering for machine learning often takes a model-centric view that focuses only on the analysis of ML models [41, 58]. For example, many advances have been made for DL model testing (e.g., [4, 16, 20, 26, 43, 62, 77, 81, 85, 86, 93, 95]), verification (e.g., [9, 60, 61, 75, 82]) and debugging (e.g., [46, 52, 57, 80]). Only a small body of work takes a holistic system view, e.g., architectural design [73, 91], technical debt [72, 79], ML component entanglement [56, 89, 94], feature interaction [1, 2, 8], and model interactions in Apollo [63]. However, the lack of system-level understanding of ML-enabled systems may hide problems in engineering ML-enabled systems and hinder practical solutions.

In this paper, we adopt this system view, and conduct a case study on Rasa 3.0 [11] to characterize the complexity of such a large-scale ML-enabled system as well as to understand the impact of the complexity on testing. Rasa is a task-oriented industrial dialogue system that has been widely used by various companies and researchers around the world [28, 64]. Therefore, we believe Rasa is a good representative of real-world ML-enabled systems.

We first investigate the complexity of Rasa at three levels. At the system level, we explore how ML components are adopted across the modules in Rasa. We find that there are 23 ML models in 15 ML components across 6 modules. At the interaction level, we analyze how ML components interact with other components in Rasa. We find that there are 43 interaction patterns and 230 interaction instances across 4 major categories and 8 inner categories. At the component level, we investigate how the code of ML components is composed by what kinds of code. We find that 57.1% of the code inside components are data processing code, and there are 8 composition patterns between data processing code and model usage code.

We then explore the impact of the complexity on testing from two perspectives. From the testing practice perspective, we analyze how is the characteristic of test cases, and how well they cope with the complexity. We find that the test coverage of component interactions is low because of the complexity from huge configuration space and from hidden component interactions. From the mutation testing perspective, we study how is the bug-finding capability of test cases and test data (i.e., the data for testing models), and how well they cope with the complexity. We find that there may be many potential bugs in data processing code that can only be detected by test cases, due to the complexity from data processing code. The capability of test data to kill mutants is limited because of the complexity from huge configuration space.

Based on our case study, we highlight practical implications to improve software engineering for ML-enabled systems. For example, the configuration space of ML-enabled systems should be tested adequately, and configuration suggestions should be provided to developers. A general taxonomy of data processing code should be constructed, and then the maintaining and testing tools for it can be developed. More integration-level test cases should be created to cover component interactions. Test cases and test data should be used in combination to detect both non-ML specific and ML-specific bugs.

In summary, this paper makes the following contributions.

- We conduct an in-depth case study on Rasa to characterize its complexity and the impact of its complexity on testing.
- We highlight practical implications to improve software engineering for ML-enabled systems.

II. Background and Study Design

We present the architecture of a typical task-oriented dialogue systems, an overview of Rasa, and our study design.

A. Architecture of a Typical Task-Oriented Dialogue System

A task-oriented dialogue system (TDS) aims to assist users in performing specific tasks, such as restaurant booking and flight booking [15]. A pipeline-based TDS consists of four parts, i.e., natural language understanding (NLU), dialogue state tracking (DST), dialogue policy (Policy) and natural language generation (NLG) [97]. NLU parses a user utterance into a structured semantic representation, including intent and slot-values. The intent is a high-level classification of the user utterance, such as `Inform` and `Request`. Slot-values are task-specific entities that are mentioned in the user utterance, such as restaurant price range and location preference. After tokenization and featurization of the user utterance, NLU applies classification models to recognize intent, and named entity extraction models to extract slot-values. DST takes the entire history of the conversation, including both user utterances with predicted intents and slot-values and system responses, to estimate the current dialogue state, which is usually formulated as a sequential prediction task [88]. Dialogue state is typically the probability distribution of user intent and slot-values till the current timestamp. Given the estimated dialogue state, Policy generates the next system action, such as `Query Database` and `Utter Question`. As Policy determines a series of actions sequentially, sequential models such as Recurrent Neural Network (RNN) are applied. For actions that require a response, NLG converts the action into a natural language utterance, which is often considered as a sequence generation task [87].

B. An Overview of Rasa 3.0

Rasa is a popular open-source ML-enabled TDS, which is fully implemented with Python and used by many well-known companies in customer service for real users, including Adobe, Airbus, and N26 [11]. An architecture overview of Rasa 3.0 is shown in Fig. 1. Each module consists of one single component or multiple semantically similar components. Apart from the modules in a typical TDS, Rasa proposes the Selector module to select candidate intents and responses for FAQ questions [17]. We present some concepts in Rasa 3.0 to ease our presentation.

Components in Rasa. There are two types of components in Rasa, namely *ML components* and *rule-based components*. We define ML components as components with an optimization objective (e.g., cross-entropy loss function), which use training data for optimization. Although some components construct internal data structures based on training data, they lack explicit optimization objectives and are therefore considered as rule-based components. For example, *CountVectorizer* [71] constructs a map of indices to tokens from training data, which can be utilized to convert test samples to a matrix of token counts. Different from ML components with optimization objectives, the training and evaluation of rule-based components will always yield identical results with the same data. We consider rule-based components in this paper, because there may exist interactions between them and ML components. General utils code in Rasa is not considered in this paper, such as command line and database access code, because we only focus on ML-related code in this paper.

Configuration File and Component Graph in Rasa. As there are multiple available components in each module, developers need to choose components that are actually used in the Rasa pipeline with a *configuration file* to build a chatbot. Parameters of each component are specified in the configuration file (e.g., ML model used by a component and hyperparamers of a ML model). Rasa applies Dask [6] to compile a configuration file into a component graph. Each node in the component graph denotes a component, and the edges connected with it denote upstream and downstream components with input and output data dependency. Execution of components obeys the topological order specified by edges. These components interact with each other through fields in shared `Message` class instances. An upstream component stores outputs to `Message` instances, and a downstream component retrieves them for further processing.

ML Stages in Rasa. Different from Apollo, which uses trained model files from external systems, and therefore only contains the inference stage of ML models [63], the training, evaluation and inference stages of ML models are all present in Rasa. Given a configuration file, Rasa separately compiles it to a training component graph and an inference component graph. In training stage, the trainable upstream components are first trained, and then process the training data used by downstream components. In evaluation stage, only the performance metrics of *IntentClassifier*, *EntityExtractor* and *Policy* are reported, as there is no ground truth for evaluation data in other modules.

C. Study Design

Our goal is to understand the complexity and its impact on testing in Rasa. To achieve this goal, we propose five RQs.

- **RQ1 System Complexity Analysis**: how ML components are adopted across the modules in Rasa?
- **RQ2 Interaction Complexity Analysis**: how ML components interact with other components in Rasa?
- **RQ3 Component Complexity Analysis**: how the code of ML components is composed by what kinds of code?
- **RQ4 Testing Practice Analysis**: how is the characteristic of test cases, and how well they cope with the complexity?
- **RQ5 Mutation Testing Analysis**: how is the bug-finding capability of test cases and test data (i.e., the data for testing models), and how well they cope with the complexity?

RQ1 aims to identify ML components in Rasa and broadly view them from the perspective of dependent libraries and ML models. **RQ2** aims to summarize a comprehensive taxonomy of component interaction patterns. **RQ3** aims to inspect the source code inside every component to characterize the statistics and composition patterns of different code types, including data processing code, model usage code, etc. Our findings from **RQ1**, **RQ2** and **RQ3** could reveal how the complexity originates and manifests in real world large-scale ML-enabled systems, which provide both practitioners and researchers with

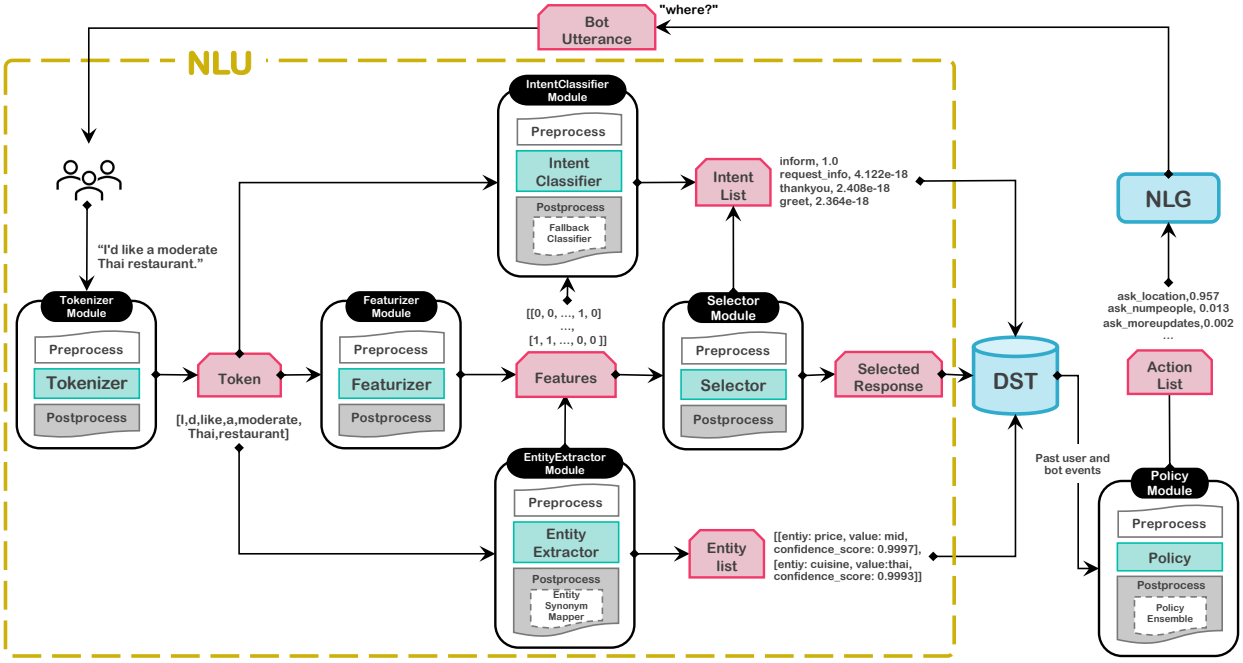

Fig. 1: The Modules and Workflow of Rasa

insights to overcome the complexities involved in implementing, maintaining, debugging and testing such complex systems.

RQ4 aims to quantitatively assess Rasa's test cases from code coverage, test case statistics (i.e., granularity levels, oracle types, and ML stages), and component interaction coverage perspectives. **RQ5** aims to generate mutants (i.e., artificial bugs) and check whether these mutants can be killed (i.e., detected) by test cases. Further, for the survived mutants, we train Rasa with 3 default configuration files on *MutiWoz* [15], a widely used multi-domain TDS dataset. We calculate the statistical significance between the performance metrics from mutated Rasa code with metrics from pipelines trained with clean code. Our findings from **RQ4** and **RQ5** evaluate the testing practice in Rasa, and shed light on automated test generation, bug localization and bug repairing techniques for complex ML-enabled systems. It took about 5, 10, 10 and 8 people-days for performing manual analysis in RQ1, RQ2, RQ3 and RQ4 respectively.

III. RQ1: System Complexity Analysis

A. Methodology

To answer **RQ1**, we identified ML and rule-based components in Rasa and characterized them through a detailed examination of Rasa's source code and documentation. We excluded DST and NLG as they are fully implemented with rule-based code logic in Rasa without ML components.

All the modules we identified are listed in Table I, except for a special module, *Shared*, as it contains general data processing code and ML model definition code (e.g., Transformer), while does not contain any independent components. We will include it in the last three RQs. Specifically, for each component, we recursively tracked methods within it to manually extract the model or rule definition code. We examined implementation details of APIs in ML libraries by reading the documentations and source code of external libraries, including ML model type and number of candidate models.

In particular, we analyzed whether ML components are implemented by using external direct libraries or indirect libraries, whether Rasa implements components with its own code and provides built-in model and rule definition code, the number of ML model parameters, and the lines of code (LoC) of each component excluding blank lines, code comments and import statements.

B. Results

The results are summarized in Table I. Components shown in gray color are ML components, and others are rule-based components. There are 6 modules in total, including 15 ML and 14 rule-based components. These components contain 23 ML models and are implemented with 7 directly dependent external ML libraries and 3 indirectly dependent external ML libraries. In particular, all ML components in *Tokenizer* and *Featurizer* are not trainable because pre-trained language models are applied. All components in *Policy* are implemented in Rasa's own code, because there are no ready-to-use Policy models provided by existing libraries. There are a total of 5348 LoC in ML components and 2980 LoC in rule-based components. In addition, the general module *Shared* contains 5375 LoC, which is not listed in the table.

Notably, we find that classical machine learning models (e.g., Support Vector Machine and Conditional Random Field) together with deep learning models (e.g., Convolutional Neural Networks and Transformer) play an important role in Rasa. This is different from the previous study [63] on Apollo, which is focused on deep learning models. Compared to DL models, classical ML models typically have significantly fewer parameters. For instance, the *CCA* used in *MiteFeaturizer* only

TABLE I: System Complexity Analysis of Rasa

Module	Component	Direct Lib.	Indirect Lib.	Model Type	No. Model	Rasa Imp.	No. Parameter	LoC
Tokenizer	JiebaTokenizer	Jieba	N/A	HMM	1	False	140K	85
	SpacyTokenizer	Spacy	Thinc	MLP	1	False	4M	39
	MitieTokenizer	Mitie	N/A	N/A	N/A	False	N/A	43
	WhitespaceTokenizer	N/A	N/A	N/A	N/A	False	N/A	52
Featurizer	ConveRTFeaturizer	TensorFlow	N/A	Transformer	1	False	110M	269
	LanguageModelFeaturizer	Transformers	TensorFlow	Transformer	6	False	82.8M	378
	MitieFeaturizer	Mitie	Dlib	CCA	1	False	500	98
	SpacyFeaturizer	Spacy	Thinc	CNN	2	False	4M	66
	CountVectorsFeaturizer	Scikit-learn	N/A	N/A	N/A	True	N/A	520
	LexicalSyntacticFeaturizer	N/A	N/A	N/A	N/A	True	N/A	319
	RegexFeaturizer	N/A	N/A	N/A	N/A	True	N/A	151
IntentClassifier	DIETClassifier	TensorFlow	N/A	Transformer	1	True	9.3M	1217
	MitieIntentClassifier	Mitie	Dlib	SVM	1	True	818	89
	SklearnIntentClassifier	Scikit-learn	N/A	SVM	1	True	2.5M	173
	FallbackClassifier	N/A	N/A	N/A	N/A	False	N/A	91
	KeywordIntentClassifier	N/A	N/A	N/A	N/A	True	N/A	132
EntityExtractor	DIETClassifier	TensorFlow	N/A	Transformer	1	True	9.3M	1217
	CRFEntityExtractor	Scikit-learn	N/A	CRF	1	True	1.6K	438
	MitieEntityExtractor	Mitie	Dlib	SVM	1	True	768	164
	SpacyEntityExtractor	Spacy	Thinc	MLP	1	False	4M	52
	DucklingEntityExtractor	N/A	N/A	N/A	N/A	False	N/A	134
	RegexEntityExtractor	N/A	N/A	N/A	N/A	True	N/A	124
	EntitySynonymMapper	N/A	N/A	N/A	N/A	True	N/A	102
Selector	ResponseSelector	TensorFlow	N/A	Transformer	2	True	110M	560
Policy	TEDPolicy	TensorFlow	N/A	Transformer+CRF	1	True	300K	1262
	UnexpecTEDIntentPolicy	TensorFlow	N/A	Transformer+CRF	1	True	300K	458
	MemoizationPolicy	N/A	N/A	N/A	N/A	True	N/A	207
	AugmentedMemoizationPolicy	N/A	N/A	N/A	N/A	True	N/A	65
	RulePolicy	N/A	N/A	N/A	N/A	True	N/A	818
	PolicyEnsemble	N/A	N/A	N/A	N/A	False	N/A	150

has 500 parameters, whereas the parameter numbers for the other three DL-based featurizers range from tens to hundreds of millions.

Next, we introduce components used in each module.

Tokenizer. Tokenizer splits the user utterance into tokens with component specific split symbols (e.g., whitespace and punctuation). (1) *SpacyTokenizer* provides the richest token information, including splitting tokens with rules, lemmatizing tokens with a look-up table, and performing part-of-speech tagging with a multi-layer perceptron (MLP). (2) *JiebaTokenizer* is the only component that tokenizes non-English sentences using Hidden Markov Model (HMM) [21]. (3) *MitieTokenizer* and *WhitespaceTokenizer* toeknize text with predefined rules.

Featurizer. As shown in Fig. 1, Featurizer converts tokens into features for downstream module inference. (1) *ConveRTFeaturizer* loads TFHub's [29] pre-trained ConveRT (Conversational Representations from Transformers) TensorFlow model to featurize tokens [31]. (2) *LanguageModelFeaturizer* loads pre-trained language models from Hugging Face Transformers [23], including BERT [18], GPT [33], XLNet [90], Roberta [49], XLM [44] and GPT2 [59]. (3) *MitieFeaturizer* combines Canonical Correlation Analysis (CCA) feature and word morphology features together. (4) *SpacyFeaturizer* applies HashEmbedCNN or Roberta to convert tokens to features, depending on the pre-trained Spacy pipeline specified in the configuration file. (5) *CountVectorsFeaturizer*, *LexicalSyntacticFeaturizer* and *RegexFeaturizer* create sparse features with n-grams, sliding window and regex patterns, respectively.

IntentClassifier. IntentClassifier generates a predicted intent list ordered by confidence scores based on tokens and features from upstream modules. (1) *DIETClassifier* implements Dual Intent and Entity Transformer (DIET) to perform intent classification and entity recognition simultaneously, and is therefore included in both **IntentClassifier** and **EntityExtractor** modules. (2) *MitieIntentClassifier* and *SklearnIntentClassifier* apply a multi-class Support Vector Machine (SVM) [74] with a sparse linear kernel using Scikit-learn and Mitie, respectively. (3) *KeywordIntentClassifier* classifies user intent with keywords extracted from training data. (4) *FallbackClassifier* is a post-processing component to check the results of other components in *DIETClassifier*. It identifies a user utterance with the intent `nlu_fallback` if the confidence scores are not greater than `threshold`, or the score difference of the two highest ranked intents is less than the `ambiguity_threshold`.

EntityExtractor. EntityExtractor extract entities such as the restaurant's location and price. (1) *DIETClassifier* also serves as an EntityExtractor. (2) *CRFEntityExtractor*, *MitieEntityExtractor* and *SpacyEntityExtractor* utilize a conditional random fields (CRF) model, a multi-class linear SVM, and a MLP to predict entities, respectively. (3) *DucklingEntityExtractor* and *RegexEntityExtractor* extract entities using a duckling server [24] and regex patterns. (4) *EntitySynonymMapper* is a post-processing component to convert synonymous entity values into a same value. As Fig. 1 shows, the value of "price" entity, "moderate", is coverted to "mid" by *EntitySynonymMapper*.

Selector. *ResponseSelector* aims to directly select the response from a set of candidate responses, which is also known as response selection task in the literature [17]. It embeds user

inputs and candidate responses in the same vector space, using the same neural network architecture as *DIETClassifier*.

Policy. Policy decides the action the system takes on each conversation based on dialogue states. (1) *TEDPolicy* proposes a Transformer Embedding Dialogue (TED) model to embed dialogue states and system actions into a single semantic vector space, and select the action with the max similarity score with the current dialogue states [84]. (2) *MemoizationPolicy*, *AugmentedMemoizationPolicy* and *RulePolicy* match the current conversation history with examples in the training data and predefined rules to predict system actions. (3) *UnexpecTEDIntentPolicy* decides on the possibility of the intent predicted by IntentClassifier given current dialogue states, which follows the same model architecture as *TEDPolicy*. (4) *PolicyEnsemble* is a post-processing component to select the proper system action from output actions of different policies.

C. Implications

The system complexity of Rasa poses challenges for developers using Rasa (i.e., application developers) and developers creating Rasa (i.e., system developers).

Complexity from ML supply chain. Rasa depends on 10 external ML libraries directly or indirectly. Less than 100 (0.03%) projects out of 355392 projects using TensorFlow on GitHub depend on 10 more DL libraries [78]. It could be inferred that relying on 10 more ML libraries is also less common. For application developers, it may be difficult to understand the implementation details of components that rely on external ML libraries, not to mention selecting proper components and parameters. For example, due to the lack of documentations of *MitieFeaturizer* in Rasa, application developers need to inspect Mitie's source code to learn that it implements CCA using Dlib APIs. For system developers, vulnerabilities [92] and dependency bugs [34] may arise because of outdated or incompatible library versions. Therefore, future work should provide supports for the management of components and corresponding dependent ML libraries for ML-enabled systems, similar to traditional software component analysis [27].

Complexity from configurations. It could be extremely complex to configure Rasa with 29 components and hundreds of parameters, making it easy to misconfigure and thus affect functionality and performance. This kind of misconfiguration is similar to what happens in traditional configurable software systems [83]. Additionally, finding optimal configurations for application developers' specific TDS scenarios may be difficult, also known as configuration debt [72]. Although AutoML has been extensively studied to select appropriate ML models and parameters for specific tasks, they all focus on selecting a single ML model without considering the combination of multiple ML models and rules [30]. Another challenge is to detect potential bugs by testing a huge set of configuration settings. Existing studies on ML model testing mainly focus on testing a single ML model with predefined hyperparameters [93].

IV. RQ2: Interaction Complexity Analysis

A. Methodology

The workflow in Fig. 1 only shows the general flow among different modules, leaving the details of interactions among components uncovered. We consider interactions among two or more components, with at least one ML component involved. An interaction pattern contains a module placeholder, which could be instantiated with components in the module to generate interaction instances. For example, the pattern (*PolicyEnsemble*, [*Policy*]) could be instantiated as (*PolicyEnsemble*, *TEDPolicy*) or (*PolicyEnsemble*, (*TEDPolicy*, *RulePolicy*)). To answer **RQ2**, we conducted a qualitative and quantitative analysis of component interaction patterns and instances of components.

Step 1: Extract interaction patterns. The interaction can be divided into two categories: inter-module interaction and intra-module interaction. (1) Inter-module interaction: the interaction between two adjacent modules (e.g., Featurizer with Tokenizer) was considered. We analyzed the usages of `Message` in the component code, as components use the `Message` class to transfer data. Specifically, we extracted all interaction patterns of its upstream and downstream components. We also considered the interaction between post-processing components (i.e., *FallbackClassifier*, *EntitySynonymMapper* and *PolicyEnsemble*) and other components in their residing modules as inter-module interaction. (2) Intra-module interaction: we identified interaction patterns for components within each module.

Step 2: Generate interaction instances. For each inter-module interaction pattern, we instantiated the module placeholder with every component in the module. For every intra-module interaction, we derived the Cartesian product of all components in each module as interaction instances. We then filtered out interaction instances that either do not contain ML components or are indicated impossible in Rasa documentation. For example, *CRFEntityExtractor* could not use features of *SparseFeaturizer* other than *RegexFeaturizer*.

Step 3: Summarize the interaction taxonomy. For generated component patterns and instances, we analyzed their semantics and summarized a component interaction taxonomy.

B. Results

The component interaction taxonomy is shown in Fig. 2. It is divided into 4 high-level categories (i.e. *Inter-Module*, *Intra-Module*, *Component Instantiation* and *Component Inheritance*) and 8 inner categories. Note that only *Inter-Module* interactions contain components with direct data dependencies, while other categories contain components with indirect interactions (e.g., two featurizers are used together). The number of interaction patterns and interaction instances in each category is listed as *pattern_count*/*instance_count* in Fig. 2. There are a total of 43 interaction patterns and 230 interaction instances. Nearly all categories include both ML to ML components and ML to rule-based components interactions. On the contrary, previous work on Apollo [63] also presented 4 of the 8 inner categories, but did not provide a taxonomy and quantitative analysis.

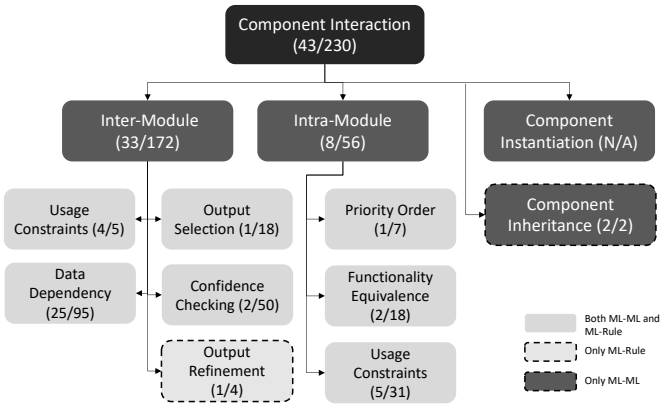

Fig. 2: Taxonomy of Component Interactions

Inter-Module. Components from multiple modules interact through data transfers. In particular, *Output Selection* means that the downstream component selects the proper output from multiple upstream outputs based on configurable criteria, e.g., *PolicyEnsemble* with policies. *Output Refinement* denotes that the downstream component complements the imperfect outputs of upstream components with rules, e.g., *EntitySynonymMapper* with entity extractors. *Confidence Checking* means that the downstream component checks reliability of the output from upstream components using ML models (e.g., *UnexpecTEDIntentPolicy* with intent classifiers) or rules (e.g., *FallbackClassifier* with IntentClassifiers). If the outputs are marked as not reliable, fallback behaviors such as the `fall_back` system action are triggered. *Usage Constraints* defines components that should or should not be used together under certain circumstances. For example, *SpacyTokenizer* is required by *CountVectorsFeaturizer* when applying `use_lemma` option and *LexicalSyntacticFeaturizer* when applying `pos_tag` option. *Data Dependency* includes the rest of inter-module interaction patterns that do not fall into any of the above categories, which are relatively "trivial" interactions with no specific semantics. For instance, *SpacyFeaturizer* uses the output of *SpaceTokenizer* as its input.

Intra-Module. The interaction mode of components within a module differs from **Inter-Module**. These components interact indirectly when used together. *Priority Order* means that the outputs of components within a module are selected according to priority order, e.g., the priority order of policies. *Usage Constraints* is similar to *Usage Constraints* in the inter-module category. For example, only one component in any of *Tokenizer*, *IntentClassifier* and *EntityExtractor* should be used in each configuration file, otherwise outputs of additional components will be overwritten. *Functionality Equivalence* includes all intra-module interaction patterns that do not belong to any of the above categories, which are relatively "trivial" interactions involving components used together with no specific semantics. For instance, both *LanguageModelFeaturizer* and *SpacyFeaturizer* can serve as a featurizer in a Rasa pipeline.

Component Instantiation. Rasa supports creating multiple instances of a component within a configuration setting. For example, multiple *CountVectorFeaturizers* instances with different ngram settings, and multiple *LanguageModelFeaturizer* instances with different language models could be used together. We did not count this category of interaction patterns and instances, since developers could specify infinite instances of a component within a configuration setting.

Component Inheritance. The class inheritance mechanism allows ML models to be shared among components. For example, ML model definition class in *UnexpecTEDIntentPolicy* is a subclass of the ML model definition class in *TEDPolicy*.

C. Implications

Lack of specifications for interactions. The outputs of ML components for specific inputs cannot be predicted due to the stochastic nature of ML models [8]. As a result, formulating interaction semantics in ML-enabled systems is more challenging than in traditional systems. When testing samples are predicted incorrectly, localizing the exact faulty component becomes difficult. Furthermore, even if the faulty component has been fixed and performance of it has been improved, the overall performance of the entire system may degrade [89]. Consequently, training and evaluation should be extended from component-level to system-level to consider interactions among components. In summary, we need to pay more attention to addressing the challenges caused by the lack of specifications in bug localization and repairing for ML-enabled systems.

Hidden interactions. Identifying all interactions is non-trivial, even for Rasa system developers. For example, the *Data Dependency* interaction between *RegexFeaturizer* and *CRFEntityExtractor* is not documented and can only be identified through source code analysis. Application developers may misuse components and get confused with the poor performance of the system without understanding the hidden interactions, especially for interaction categories like *Usage Constraints*, *Output Selection* and *Priority Order*. Techniques like data flow analysis can be explored to automatically reveal component interactions in ML-enabled systems [70].

Furthermore, our results on component interaction complexity could be helpful to guide developers to build better ML-enabled systems. For example, developers can follow interaction patterns *Output Selection* and *Output Refinement* to improve the outputs of components at system level, as well as utilizing *Confidence Checking* to detect cases that ML models can not handle, and then triggering fallback rules, which is particularly important in safety-critical systems like self-driving systems [63].

V. RQ3: Component Complexity Analysis

A. Methodology

To answer **RQ3**, we classified categories of code snippets in each component and explored their composition patterns.

Step 1: Label code snippets. We segmented each source code file into code snippets according to semantic meaning, and then classified them into 6 categories: (1) model definition, the definition code of ML models; (2) rule definition, the definition code of rules in rule-based components; (3) model usage, the

TABLE II: LoC of Different Code Categories

Module	Data		Model		Rule	
	Pre.	Post.	Usage	Definition	Usage	Definition
Tokenizer	8	80	27	0	25	25
Featurizer	390	323	92	0	162	119
IntentClassifier	441	131	113	298	3	69
EntityExtractor	746	311	120	298	24	30
Selector	48	55	9	16	0	0
Policy	1332	540	64	543	167	283
Shared	996	314	112	1673	0	43
Total	3961	1754	537	2828	381	569

usage code of ML models; (4) rule usage, the usage code of rules; (5) data pre-processing, the input data processing code before model or rule usages; (6) data post-processing, the output data processing code after model or rule usages. Two of the authors labeled code snippets independently, and the third author was involved to resolve disagreements. The Cohen's Kappa coefficient of the two authors was 0.830.

Step 2: Summarize composition patterns of code snippets. Based on labeled code snippets, we summarized the composition patterns of data processing code, and model or rule usage code in each component.

B. Results

The statistics of different code categories are shown in Table II. We only considered the LoC of labeled code snippets, while ignoring general utils code such as class initialization. Data processing code contributes a total of 5715 (57.1%) LoC, while model usage&definition code and rule usage&definition code contribute 3365 (33.5%) and 950 (9.4%) LoC, respectively. 1673 (59.2%) of the 2828 LoC of model definition code is in *Shared* module, which shows that the reuse of model definition code between different components is quite common. There is no model definition code in *Tokenizer* and *Featurizer*, because ML components are all built on top of external ML libraries.

We classified data pre-processing and data post-processing categories into more specific types, due to the dominant proportion of data processing code in Rasa. Specifically, *Validation* code intends to validate the input or output data of components. *Format Transformation* code transforms data format, such as constructing vectors from Python arrays and reshaping vectors. *Component Input/Output Filter* code filters data that does not meet the specified criteria, such as the absence of certain attributes. *Data Scale/Padding/Encoding/Decoding* code changes the value of data, while *Data Split/Shuffle/Balance/Batch/Rank* code changes the organization of data for better training and inference of components. We provide the complete codebook and statistics of data processing types at our website [7].

Moreover, we find that composition patterns of code snippets include sequential code composition pattern and various non-sequential composition patterns. In a typical sequential composition pattern, data is first pre-processed, and then processed by model or rule usage code, and finally post-processed. The non-sequential code composition patterns are summarized in Fig. 3. The black box is a data processing code snippet, the red box is a model or rule usage code snippet, and the green diamond means to select one or multiple downstream code snippets. The first 5 patterns consist of multiple model or rule usages in one component. The last 3 patterns consist of a single model or rule usage with multiple possible data processing snippets, decided by configurations or input data.

C. Implications

Data processing. Data processing code is scattered at different granularity levels, unlike the well-documented and structured code of ML models and rules. In detail, data processing code includes data processing components (e.g., *PolicyEnsemble*), general data processing classes and functions in *Shared* module, and specific data processing snippets in components entangled with model or rule usages. On the one hand, it could become troublesome for application developers to identify and understand the semantics of all data processing code. A specific example is that data pre-processing code also exists in model definition class of `TransformerRasaModel`, including *Formant Conversion* and *Data Batch* code. It could be explicitly helpful to automatically extract and analyze the semantics of data processing code with techniques like program analysis [69]. On the other hand, it would be challenging for system developers to maintain and test data processing code, possibly resulting in severe consequences with ML development paradigm shift from model-centric to data-centric [48]. In general, building a taxonomy of data processing code would be helpful for maintaining and testing of data processing code.

Code composition patterns. These non-sequential composition patterns could introduce additional dynamic complexity for ML-enabled systems, e.g., it is too expensive to capture all possible run-time compositions of code snippets with static analysis. Although dynamic testing is widely adopted to complement the limitations of static analysis in traditional software [25], most existing testing techniques tailored for ML only target the ML model level [93]. It would be beneficial to extend them to include data processing code and composition patterns.

VI. RQ4: Testing Practice Analysis

A. Methodology

To answer **RQ4**, test cases were inspected in three steps.

Step 1: Label test cases. We manually labeled the granularity level, oracle type and ML stage of each test case. There are three different granularity levels of test cases: (1) Method-level: testing single or multiple methods; (2) Component-level: testing the complete process of a component in training or inference stage; (3) Integration-level: testing the current component with upstream components. There are four test oracle types: (1) Given input-output pairs: the input and expected output data are given; (2) Component-specific constraints: the constraints must be satisfied according to the implementation of a component, i.e., the sum of confidence scores of the intent list generated by classifiers should equal to 1; (3) Differential executions: outputs of executions under different settings should change or remain the same, i.e., given the same input, the outputs of an original ML model and its loaded version from disk should remain the same; (4) Exception: whether or not the test case throws exceptions for certain inputs and configurations. Finally, the ML stages covered by test cases include training, inference and

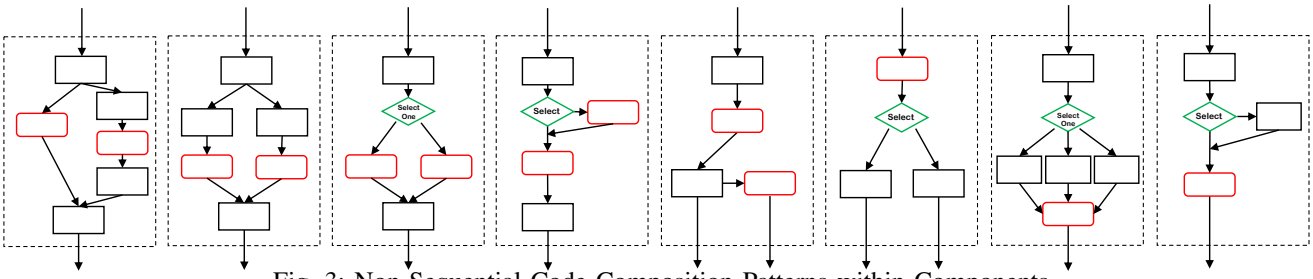

Fig. 3: Non-Sequential Code Composition Patterns within Components

TABLE III: Code Coverage and Labeled Statistics of Test Cases

Module	Total	Test Case Type			Test Case Stage			Oracle Type				Code Coverage	
		Meth.	Comp.	Integ.	Infer.	Train	Eval.	I-O	C-S	Diff.	Exception	Stat. Cov.	Bran. Cov.
Tokenizer	27	7	20	0	25	14	0	24	1	0	3	97.4%	96.8%
Featurizer	62	13	14	35	56	40	0	46	5	3	8	95.7%	94.9%
IntentClassifier	36	7	15	14	29	30	0	18	11	7	1	92.5%	89.5%
EntityExtractor	41	6	19	16	34	31	0	18	14	8	1	92.3%	90.1%
Selector	13	4	6	3	9	12	0	6	5	1	1	68.3%	66.4%
Policy	165	77	88	0	105	127	0	117	51	0	20	95.7%	94.5%
Shared	138	129	2	7	90	84	47	89	42	2	16	92.3%	91.4%
Total	461	240	156	65	331	317	47	312	123	15	49	93.2%	92.0%

evaluation stages. To test the training stage of a component, test cases must first train it and check whether any test oracle is violated. Note that there may exist several oracle types and ML stages but only one granularity level for each test case in Rasa. Two of the authors labeled test cases independently, and the third author was involved to resolve disagreements. The Cohen's Kappa coefficient of granularity level, test oracle and ML stage is 0.907, 0.908, and 0.854, respectively.

Step 2: Collect code coverage of test cases. We collected the statement coverage and branch coverage of code via *pytest-cov* [65], because Rasa uses *pytest* to run test cases.

Step 3: Collect interaction pattern coverage of test cases. We injected logging statements into methods of every component, and then executed test cases to collect the co-executed component sets of each test case. All component interaction instances, except *Model Inheritance* instances and some *Usage Constraints* instances that cannot be used together, were tried to be matched with these component sets. The matched interaction patterns and instances were considered as covered by test cases.

B. Results

The code coverage and labeled statistics of test cases are shown in Table III. (1) The total statement coverage and branch coverage of code reach 93.2% and 92.0%, which is much higher than 21.5% and 13.3% in Apollo [63]. One potential reason for Apollo's low test coverage is that, different from Rasa, Apollo encompasses hardware-related code and is a multi-language project, making the creation of high-coverage test cases more challenging. (2) The coverage of *Selector* is only 68.3% and 66.4%, because *Selector* has two candidate ML models while only one of them was tested. (3) There are 240 (52.0%) method-level, 156 (33.8%) component-level and 65 (14.1%) integration-level test cases. (4) There is no integration-level test cases in *Policy*, because *Policy* was tested with given intents and entities input from developers without the dependency of NLU modules. (5) Inference and training stages have similar test case quantities. (6) Only test cases in *Shared* module cover evaluation stage, because *Shared* module provides the evaluation code for all components. (7) There are 312 (67.7%), 123 (26.3%), 15 (3.3%), and 49 (10.6%) test cases with given input-output pairs, component-specific constraints, differential executions and exception test oracles.

TABLE IV: Test Coverage of Component Interactions

Category	Sub-Category	Cov. Patterns	Cov. Instances
Inter-module	Data Dependency	9/25	17/95
	Confidence Checking	0/2	0/50
	Output Selection	0/1	0/18
	Output Refinement	1/1	1/4
	Usage Constraints	3/3	3/3
Intra-module	Functionality Equivalence	2/2	3/18
	Prioriy Order	1/1	4/7
	Usage Constraints	2/2	2/4
Total		18/37	30/199

As Table IV shows, the test coverage of interactions is relatively low, i.e., 18 (48.6%) of 37 patterns and 30 (15.1%) of 199 instances are covered. This is because only integration-level tests cover components interactions. In particular, *Confidence Checking* and *Output Selection* are not covered.

C. Implications

Low test coverage of component interactions. It is difficult to achieve a high test coverage of component interactions due to the complexity caused by vast configuration space and hidden interactions. The only test cases that cover component interactions (i.e., integration-level test cases) contribute no more than 15% test cases. Yet, integration-level test cases can cover and kill more mutants than component-level and method-level test cases, as many mutants do not manifest in non-integration-level test cases [45]. Therefore, it is crucial to generate integration-level test cases for ML-enabled systems.

Limited test oracle types. It is challenging and time-consuming to write test cases with given input-output pairs and component-specific constraints oracles, due to the complexity brought by the lack of specification for interactions. As a result,

those test cases without the need of specification of interactions, that is, differential executions and exception test oracles, have been widely utilized to tackle the oracle problem in test case generation techniques for traditional software, such as differential testing [22], fuzzing [47] and search-based testing [53]. Besides, we find that test cases with these two oracles have a similar capability to kill mutants similar to component-specific constraints oracle (see **RQ5**). In spite of this, only 13.9% test cases in Rasa are written with differential executions and exception test oracles, implying that there could be a big room to apply these two test oracle types in test case generation techniques for ML-enabled systems.

VII. RQ5: Mutation Testing Analysis

A. Methodology

To answer **RQ5**, we performed an analysis of mutation testing [40]. It applies mutators to generate versions of faulty code, i.e., mutants. Every mutator was applied independently, and we did not consider multiple mutators at the same time. For every mutant, test cases and test data were applied to collect the testing results to decide whether the mutant was killed. As Rasa contains both ML components and rule-based components, we considered both mutators for traditional software (i.e., syntactic mutators) and ML-specific mutators.

We assessed both test cases and test data for their detection capabilities on these two different mutant types, rather than using test cases only for syntactic mutants and test data only for ML-specific mutants. Test cases for ML components typically use specifically designed toy training data and toy test data to verify whether ML components can be trained and tested properly, and whether they satisfy specific constraints. For example, the test case *test_softmax_normalization* verifies that the *DIETClassifier* outputs confidence scores that sum up to 1 after softmax normalization under different conditions [67]. It utilizes two distinct training data files [66, 68] with seven configurations to cover various corner cases. Therefore, these test cases can also detect ML-specific mutants (e.g., generated by mutating activation functions). Besides, syntactic mutants' impact on the accuracy of the system in real-world scenarios needs to be evaluated with test data. Therefore, both test cases and test data can kill two different types of mutants. Running a test case does not require whole model training and evaluation process, and failures can assist developers in identifying locations of issues. In contrast, test data can evaluate the system's real-world performance with the whole training and evaluation process. These two methods complement each other.

As Table V shows, we used 9 syntactic mutators from Jia et al. [39] and 11 ML-specific mutators from DeepCrime [35].

We list steps of mutation analysis in the following.

Step 1: Generate mutants. We generated syntactic mutants using *mutmut* [54]. We used two groups of syntactic mutators, i.e., *Logic* and *Value*, which mutate the logic flow and variable value. Besides, we generated ML-specific mutants with DeepCrime [35]. We used 4 of 8 mutation groups in DeepCrime (*Activation*, *Regularization*, *Weights* and *Optimization*). For others, mutators in *Training Data* and *Validation* groups are not included because this paper focuses on potential bugs in code rather than data; *Hyperparameters* group is not included, as hyperparameters in Rasa are specified with configuration files by developers; and *Loss Function* group is not applicable, as the loss functions in Rasa are all implemented from scratch, while the mutators provided by DeepCrime are only to replace the Keras loss function API with another one.

Besides, we generated syntactic mutants for 6 labeled code cateogries in **RQ3**, excluding general utils code, and generated ML-specific mutants only for model definition code. We generated no more than 30 mutants for every Python class to reduce potential bias. We also only modified one AST node for every mutant.

Step 2: Perform mutation testing analysis with test cases. For every mutant, only test cases that cover the mutated line were collected (from test coverage data in **RQ4**) and executed to save running time. If any test case fails on a mutant, the mutant is considered as killed by the test case. Otherwise, the mutant is considered as survived. A test case could fail with three symptoms: (1) an assertion fails; (2) an execution or runtime error manifests; and (3) the test case times out. The maximum time for a test case to run is 10 times of its running time in the original clean code version. Besides, test cases were executed 3 times for every mutant to avoid flaky tests. We found that all test case statuses remain same for three runs.

We did not apply statistical killing [35, 38] for test cases, becasue test cases must be deterministic. Otherwise, it results in an undesirable flaky test. We executed the test cases provided by Rasa three times and did not encounter any flaky tests. Thus, when a syntactic or ML-specific mutant is killed by a test case here, it indicates that the test case encountered assertion errors, runtime exceptions, or timeouts, with these issues recurring upon repetition.

Step 3: Perform mutation testing analysis with test data. For those survived mutants in Step 2, we assessed the impact of them with 3 default configuration files and the restaurant domain data in *Multiwoz* [15], which is a widely used multi-domain dataset to evaluate the performance of TDS. Given a configuration file, only components specified in it are included in the Rasa pipeline, thus mutated nodes of some survived mutants from Step 2 will not be executed as they are not *impacted* by the configuration. Due to the stochastic nature of machine learning programs, we trained both the mutated program and original program for 5 times with 80/20 data split into train/test data randomly, and decided whether the performance metrics of two versions are statistically significant with non-negligible and non-small effect size. We followed the same statistical killing method to decide whether a mutant is killed with the test data as previous works [35, 38], with the threshold of significance value is 0.05 and of effect size is 0.5. We adopted F1 scores of *IntentClassifier*, *EntityExtractor* and *Policy* as performance metrics, i.e., if the F1 score in any of the three modules is statistically different between two code versions, the mutant is marked as killed by test data.

TABLE V: Mutation Testing Results

Mutation Group	Operator	Total	Test Case		Test Data	
			Killed	Survived	Impacted	Killed
Logic	ArOR	109	86	23	10	1
	ComOR	109	88	21	6	0
	LogOR	145	112	33	14	0
	AsOR	20	19	1	0	0
	MemOR	32	30	2	0	0
	KVR	12	7	5	1	0
Value	BVR	64	32	32	9	0
	NVR	224	180	64	18	2
	AsVR	582	525	67	10	0
Activation	ACH	22	3	19	18	6
	ARM	2	0	2	1	0
	AAL	22	11	11	11	2
Regularization	RAW	6	0	6	3	3
	RCW	10	0	10	10	0
	RRW	5	0	5	5	0
Weights	WCI	24	10	14	13	1
	WAB	4	0	4	3	1
	WRB	3	1	2	2	0
Optimization	OCH	24	2	22	9	6
	OCG	8	0	8	3	0
Total		1447	1106	341	146	22

TABLE VI: Mutant Location Results

Location	Total	Test Case Result		Test Data Result	
		Killed	Survived	Impacted	Killed
Data Prep.	385	326	59	23	0
Data Post.	271	222	49	5	0
Model Usage	307	243	64	4	0
Model Def.	364	224	140	99	22
Rule Usage	4	4	0	0	0
Rule Def.	115	101	14	4	0

B. Results

The mutation testing results by each mutator are shown in Table V. There are 1447 mutants generated, 1106 (76.4%) mutants killed by test cases, 341 (23.6%) mutants survived, 146 (10.1%) mutants impacted, and 22 (1.5%) mutants killed by test data. Only 146 mutants from 341 survived mutants impact the default 3 Rasa pipelines, which shows that the huge configuration space is challenging to be tested adequately. 81.3% syntactic mutants and 20.0% ML-specific mutants are killed by test cases, while 4.4% syntactic mutants and 24.4% ML-specific mutants from impacted mutants are killed by test data. It shows that test case is much more effective to detect syntactic mutants and slightly less effective to detect ML-specific mutants than test data. The killed syntactic mutants and ML-specific mutants by test data cause the F1 score degradation of *IntentClassifier*, *EntityExtractor* and *Policy* by 20.8%, 0.8%, 3.6% and 11.1%, 13.4%, 5.7% on average.

The mutation testing results w.r.t. the location of mutants are shown in Table VI. 224 (61.5%) of 364 mutants in model definition code, and 896 (82.8%) of 1082 mutants in other code categories are killed by test cases. In particular, few mutants in code categories except model definition are impacted and killed by test data, which implies that test data is only effective to kill mutants in model definition code.

We investigated the capability to detect mutants w.r.t. different categories of test cases, by calculating the ratio of *strong test case number* to *all test case number*, and the ratio of *killed mutants* to *covered mutants* of them. We define *strong test case* as the test case that kills equal or more than 75% of its covered mutants. As Table VII shows, test cases in integration level have the highest ratio of strong test case (44.6%) and highest ratio of killed mutants (67.9%) among three granularity levels. Test cases with given input-output test oracle have the highest ratio of strong test case (31.4%) and highest ratio of killed mutants (73.7%) among four oracle types, while test cases with other three oracle types have similar ratios.

TABLE VII: Test Case Mutation Results

Category	Type	Test Num.	Strong Test Num.	Covered	Killed
Granularity	Method	240	59	947	635
	Component	156	31	1121	709
	Integration	65	29	903	613
Stage	Infer.	331	86	1358	995
	Training	317	75	1184	847
	Evaluation	47	11	772	476
Oracle Type	I-O	312	98	1298	956
	C-S	123	19	1103	707
	Diff	15	3	625	338
	Exception	49	6	686	352

C. Implications

Non-ML-specific bugs and test cases in ML-enabled systems. Complexity from data processing code causes that non-ML-specific bugs are prone to be introduced. Compared with test data, test case is more effective to detect syntactic mutants, i.e., non-ML-specific bugs. Moreover, it is notorious for developers to analyze, localize and fix bugs in ML programs according to test data, thus interpreting [96], debugging [3] and repairing [76] techniques have been developed for ML models. It is easier for developers to localize and fix bugs with failed test cases by analyzing violated test oracles. Thus, we claim that non-ML-specific bugs and test cases in ML-enbaled systems should be paid more attention to. Although there is a rich set of test cases in Rasa that achieve high code coverage, the kill ratio of mutants remains to be improved (76.4%), especially of ML-specific mutants (29.8%). The applicability and limitations of existing test case generation, selection and quality assurance techniques in ML-enabled systems are worthwhile to be explored [19, 42].

Challenges of test data to kill mutants. Existing researches on mutation testing for ML programs only evaluated mutants with test data to decide whether they can be killed [32, 35, 38, 39, 51]. However, the capability of test data to kill mutants in large-scale ML-enabled systems is limited for two reasons. First, due to complexity from configurations, only part of mutants will impact the components of actual configured systems. Second, the amount and distributions of training data and test data affect the results a lot. For example, we tried to train the clean code version and mutated version with 75% of original training data, the number of killed mutants changed from 22 to 83, which means some bugs may only manifest under specific training data settings. Therefore, system developers should evaluate and test ML-enabled systems under more possible configurations and data settings that may be used by application developers to detect potential bugs.

VIII. THREATS

First, our study conducts a case study on Rasa, a widely used task-oriented industrial dialogue system. It is not clear whether our results can be generalized to other ML-enabled systems. However, we believe it is a good start to take a system view for ML-enabled systems. Second, our study involves a lot of manual analyses of Rasa source code and documentations, which may incur biases. To reduce them, two of the authors conduct manual analysis separately, and a third author is involved to resolve disagreements. Third, the mutators that we adopt may not simulate real-world bugs. To mitigate it, we decide to use mutators from DeepCrime [35], whose mutators are actually summarized from real word ML bugs.

IX. RELATED WORK

Study of ML-Enabled Systems. While much of the attention has been on ML models, less attention has been paid on system-level analysis [41]. Peng et al. [63] investigated the integration of ML models in Apollo by analyzing how ML models interact with the system and how is the current testing effort. Besides, Nahar et al. [55] explored collaboration challenges between data scientists and software engineers through interviews. Amershi et al. [5] and Bernardi et al. [10] reported challenges and practices of MLOps (from model requirement to model monitoring) at Microsoft and Booking.com. Although they still take a model-centric view, they emphasize that models can be complexly entangled to cause non-monotonic errors [5] and model quality improvement does not necessarily indicate system value gain [10]. Further, Yokoyama [91] developed an architectural pattern to separate ML and non-ML components, while Serban and Visser [73] surveyed architectural challenges for ML-enabled systems. Sculley et al. [72] identified ML-specific technical debt in ML-enabled systems, while Tang et al. [79] further derived new ones from real-world code refactorings. In addition, some attempts were made on the problem of ML component entanglement [5], e.g., performing metamorphic testing on a system with two ML components [94], troubleshooting failures in a system with three ML components by human intellect [56], and decomposing errors in a system with two or three ML components [89]. These studies explore the interaction among models but only on simple systems. Moreover, Abdessalem et al. [1, 2] studied the feature interaction failures in self-driving systems, and proposed testing and repairing approaches to automatically detect and fix them. Apel et al. [8] also discussed feature interactions in ML-enabled systems, and suggested strategies to cope with them.

The main difference from the previous work is that we take a large-scale complex ML-enabled system, explore its complexity at three levels, and analyze the impact of its complexity on testing. The closest work is Peng et al.'s [63], but we report a deeper complexity analysis and also conduct a testing impact analysis.

Mutation Testing for DL Models. Jia et al. [39] used syntactic mutators for traditional programs to DL models. DeepMutation [51] and DeepMutation++ [32] defined DL-specific mutators. DeepCrime [35] derived DL-specific mutators based on real DL bugs. Jahangirova and Tonella [38] evaluated syntactic and DL-specific mutators. These studies are focused on model-level mutation, while we target at system-level mutation. As far as we know, only Jahangirova et al. [37] performed system-level mutation analysis for autonomous vehicles.

Testing for Dialogue Systems. Bozic and Wotawa [13] proposed a security testing approach for chatbots to prevent cross-site scripting and SQL injection. Bozic et al. [12] tested a hotel booking chatbot via planning. Bozic and Wotawa [14] introduced a metamorphic testing approach for chatbots. Similarly, Liu et al. [50] used semantic metamorphic relations to test the NLU module in dialogue systems. Despite the effort, less attention has been paid on system-level testing of dialogue systems.

X. CONCLUSION

We present a comprehensive study on Rasa to characterize its complexity at three levels and the impact of its complexity on testing from two perspectives. Furthermore, we highlight practical implications to improve software engineering for ML-enabled systems. All study data and source code used in this paper are available at https://rasasystemcomplexity.github.io/.

ACKNOWLEDGEMENT

This work was supported by the National Key R&D Program of China (2021ZD0112903). Bihuan Chen is the corresponding author of this paper.

REFERENCES

[1] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, p. 143–154.

[2] ——, "Automated repair of feature interaction failures in automated driving systems," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 88–100.

[3] A. Abid, M. Yuksekgonul, and J. Zou, "Meaningfully debugging model mistakes using conceptual counterfactual explanations," in *Proceedings of the International Conference on Machine Learning*, 2022, pp. 66–88.

[4] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, p. 625–635.

[5] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 291–300.

[6] Anaconda. (2022) Dask. [Online]. Available: https://docs.dask.org/en/stable/

[7] Anonymous. (2023) Understanding the complexity and its impact on testing in ml-enabled systems. [Online]. Available: https://rasasystemcomplexity.github.io/

[8] S. Apel, C. Kästner, and E. Kang, "Feature interactions on steroids: On the composition of ml models," *IEEE Software*, vol. 39, no. 3, pp. 120–124, 2022.

[9] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena, "Scalable quantitative verification for deep neural networks," in *Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings*, 2021, p. 248–249.

[10] L. Bernardi, T. Mavridis, and P. Estevez, "150 successful machine learning models: 6 lessons learned at booking.com," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, p. 1743–1751.

[11] T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, "Rasa: Open source language understanding and dialogue management," *CoRR*, vol. abs/1712.05181, 2017.

[12] J. Bozic, O. A. Tazl, and F. Wotawa, "Chatbot testing using ai planning," in *Proceedings of the IEEE International Conference On Artificial Intelligence Testing*, 2019, pp. 37–44.

[13] J. Bozic and F. Wotawa, "Security testing for chatbots," in *Proceedings of the IFIP International Conference on Testing Software and Systems*, 2018, pp. 33–38.

[14] ——, "Testing chatbots using metamorphic relations," in *Proceedings of the IFIP International Conference on Testing Software and Systems*, 2019, pp. 41–55.

[15] P. Budzianowski, T.-H. Wen, B.-H. Tseng, I. Casanueva, S. Ultes, O. Ramadan, and M. Gašić, "Multiwoz - a large-scale multi-domain wizard-of-oz dataset for task-oriented dialogue modelling," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 5016–5026.

[16] J. Cao, B. Chen, C. Sun, L. Hu, S. Wu, and X. Peng, "Understanding performance problems in deep learning systems," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 357–369.

[17] D. Chaudhuri, A. Kristiadi, J. Lehmann, and A. Fischer, "Improving response selection in multi-turn dialogue systems by incorporating domain knowledge," in *Proceedings of the 22nd Conference on Computational Natural Language Learning*, 2018, pp. 497–507.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[19] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 302–311.

[20] S. Dola, M. B. Dwyer, and M. L. Soffa, "Distribution-aware testing of neural networks using generative models," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, 2021, pp. 226–237.

[21] S. R. Eddy, "Hidden markov models," *Current opinion in structural biology*, vol. 6, no. 3, pp. 361–365, 1996.

[22] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007, pp. 549–552.

[23] H. Face. (2022) Transformers. [Online]. Available: https://huggingface.co/docs/transformers/index

[24] Facebook. (2022) Duckling. [Online]. Available: https://github.com/facebook/duckling/

[25] R. E. Fairley, "Tutorial: Static analysis and dynamic testing of computer software," *Computer*, vol. 11, no. 4, pp. 14–23, 1978.

[26] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, p. 177–188.

[27] D. Foo, J. Yeo, H. Xiao, and A. Sharma, "The dynamics of software composition analysis," *CoRR*, vol. abs/1909.00973, 2019.

[28] J. Gao, K. T. W. Choo, J. Cao, R. K. W. Lee, and S. Perrault, "Feasibility, opportunities, and challenges of utilizing ai for collaborative qualitative analysis," *arXiv preprint arXiv:2304.05560*, 2023.

[29] Google. (2022) Tensorhub. [Online]. Available: https://tensorflow.google.cn/hub

[30] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge Based Systems*, vol. 212, 2021.

[31] M. Henderson, I. Casanueva, N. Mrkšić, P.-H. Su, T.-H. Wen, and I. Vulić, "Convert: Efficient and accurate conversational representations from transformers," *CoRR*, 2019.

[32] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1158–1161.

[33] Z. Hu, Y. Dong, K. Wang, K.-W. Chang, and Y. Sun, "Gpt-gnn: Generative pre-training of graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1857–1867.

[34] K. Huang, B. Chen, S. Wu, J. Cao, L. Ma, and X. Peng, "Demystifying dependency bugs in deep learning stack," *CoRR*, vol. abs/2207.10347, 2022.

[35] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: Mutation testing of deep learning systems based on real faults," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, p. 67–78.

[36] IBM. (2022) Ibm global ai adoption index 2022. [Online]. Available: https://www.ibm.com/watson/resources/ai-adoption

[37] G. Jahangirova, A. Stocco, and P. Tonella, "Quality metrics and oracles for autonomous vehicles testing," in *2021 14th IEEE conference on software testing, verification and validation (ICST)*. IEEE, 2021, pp. 194–204.

[38] G. Jahangirova and P. Tonella, "An empirical evaluation of mutation operators for deep learning systems," in *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification*, 2020, pp. 74–84.

[39] L. Jia, H. Zhong, X. Wang, L. Huang, and Z. Li, "How do injected bugs affect deep learning?" in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 793–804.

[40] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[41] C. Kästner. (2022) Machine learning in production: From models to systems. [Online]. Available: https://ckaestne.medium.com/machine-learning-in-production-from-models-to-systems-e1422ec7cd65

[42] R. Kazmi, D. N. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–32, 2017.

[43] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, p. 1039–1049.

[44] G. Lample and A. Conneau, "Cross-lingual language model pretraining," 2019. [Online]. Available: https://arxiv.org/abs/1901.07291

[45] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*, 1990, pp. 290–301.

[46] Z. Li, X. Ma, C. Xu, J. Xu, C. Cao, and J. Lü, "Operational calibration: Debugging confidence errors for dnns in the field," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 901–913.

[47] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[48] W. Liang, G. A. Tadesse, D. Ho, F.-F. Li, M. Zaharia, C. Zhang, and J. Zou, "Advances, challenges and opportunities in creating data for trustworthy AI," *Nature Machine Intelligence*, 2022.

[49] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[50] Z. Liu, Y. Feng, and Z. Chen, "Dialtest: automated testing for recurrent-neural-network-driven dialogue systems," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 115–126.

[51] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," in *Proceedings of the IEEE 29th International Symposium on Software Reliability Engineering*, 2018, pp. 100–111.

[52] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: Automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, p. 175–186.

[53] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163.

[54] Mutmut. (2022) Mutmut. [Online]. Available: https://pypi.org/project/mutmut/

[55] N. Nahar, S. Zhou, G. Lewis, and C. Kästner, "Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process," in *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 413–425.

[56] B. Nushi, E. Kamar, E. Horvitz, and D. Kossmann, "On human intellect and machine failures: Troubleshooting integrative machine learning systems," in *Proceedings of the Thirty-First AAAI Conference on Artificial*

Intelligence, 2017, pp. 1017–1025.
[57] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," in *Proceedings of the 36th International Conference on Machine Learning*, 2019, pp. 4901–4911.
[58] K. O'Leary and M. Uchida, "Common problems with creating machine learning pipelines from existing code," in *Proceedings of the Third Conference on Machine Learning and Systems*, 2020.
[59] OpenAI. (2022) Gpt2. [Online]. Available: https://openai.com/blog/tags/gpt-2/
[60] B. Paulsen, J. Wang, and C. Wang, "Reludiff: Differential verification of deep neural networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, p. 714–726.
[61] B. Paulsen, J. Wang, J. Wang, and C. Wang, "Neurodiff: Scalable differential verification of neural networks using fine-grained approximation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, p. 784–796.
[62] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, p. 1–18.
[63] Z. Peng, J. Yang, T.-H. P. Chen, and L. Ma, "A first look at the integration of machine learning models in complex autonomous driving systems: A case study on apollo," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1240–1250.
[64] D. Porfirio, E. Fisher, A. Sauppé, A. Albarghouthi, and B. Mutlu, "Bodystorming human-robot interactions," in *proceedings of the 32nd annual ACM symposium on user Interface software and technology*, 2019, pp. 479–491.
[65] Pytest. (2023) pytest_cov. [Online]. Available: https://pypi.org/project/pytest-cov/
[66] Rasa. (2023) test_data_nlu. [Online]. Available: https://github.com/RasaHQ/rasa/blob/main/data/test_moodbot/data/nlu.yml
[67] ——. (2023) test_diet_classifier. [Online]. Available: https://github.com/RasaHQ/rasa/blob/524a6d67d40a265e131eadbdf6b865d1196e6374/tests/nlu/classifiers/test_diet_classifier.py#L443
[68] ——. (2023) test_many_intents. [Online]. Available: https://github.com/RasaHQ/rasa/blob/main/data/test/many_intents.yml
[69] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021, p. 1646–1657.
[70] F. Sattler, A. von Rhein, T. Berger, N. S. Johansson, M. M. Hardø, and S. Apel, "Lifting inter-app data-flow analysis to large app sets," *Automated Software Engineering*, vol. 25, no. 2, pp. 315–346, 2017.
[71] scikit learn. (2023) count_vectorizer. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[72] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, 2015, p. 2503–2511.
[73] A. Serban and J. Visser, "Adapting software architectures to machine learning challenges," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 152–163.
[74] A. Shmilovici and L. Rokach, *Support Vector Machines*, 2005, pp. 257–276.
[75] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–30, 2019.
[76] B. Sun, J. Sun, L. H. Pham, and J. Shi, "Causality-based neural network repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 338–349.
[77] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, p. 109–119.
[78] X. Tan, K. Gao, M. Zhou, and L. Zhang, "An exploratory study of deep learning supply chain," in *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 86–98.
[79] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, "An empirical study of refactorings and technical debt in machine learning systems," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, 2021, pp. 238–250.
[80] G. Tao, S. Ma, Y. Liu, Q. Xu, and X. Zhang, "Trader: Trace divergence analysis and embedding regulation for debugging recurrent neural networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, p. 986–998.
[81] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, p. 303–314.
[82] F. Toledo, D. Shriver, S. Elbaum, and M. B. Dwyer, "Distribution models for falsification and verification of dnns," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021, p. 317–329.
[83] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "On debugging the performance of configurable software systems: Developer needs and tailored tool support," in *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 1571–1583.
[84] V. Vlasov, J. E. M. Mosig, and A. Nichol, "Dialogue transformers," *CoRR*, vol. abs/1910.00486, 2019.
[85] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are machine learning cloud apis used correctly?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 125–137.
[86] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Automated testing of software that uses machine learning apis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 212–224.
[87] T.-H. Wen, M. Gašić, N. Mrkšić, P.-H. Su, D. Vandyke, and S. Young, "Semantically conditioned LSTM-based natural language generation for spoken dialogue systems," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1711–1721.
[88] J. Williams, A. Raux, and M. Henderson, "The dialog state tracking challenge series: A review," *Dialogue & Discourse*, vol. 7, no. 3, pp. 4–33, 2016.
[89] R. Wu, C. Guo, A. Y. Hannun, and L. van der Maaten, "Fixes that fail: Self-defeating improvements in machine-learning systems," in *Proceedings of the 35th Conference on Neural Information Processing Systems*, 2021, pp. 11 745–11 756.
[90] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.
[91] H. Yokoyama, "Machine learning system architectural pattern for improving operational stability," in *Proceedings of the IEEE International Conference on Software Architecture Companion*, 2019, pp. 267–274.
[92] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *Proceedings of the 17th International Conference on Software Reuse*, 2018, pp. 95–110.
[93] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2022.
[94] J. Zhang, X. Jing, W. Zhang, H. Wang, and Y. Dong, "Improve the quality of arc systems based on the metamorphic testing," in *Proceedings of the International Symposium on System and Software Reliability*, 2016, pp. 137–141.
[95] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai, "White-box fairness testing through adversarial sampling," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, p. 949–960.
[96] Y. Zhang, P. Tiňo, A. Leonardis, and K. Tang, "A survey on neural network interpretability," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 5, no. 5, pp. 726–742, 2021.
[97] Z. Zhang, R. Takanobu, Q. Zhu, M. Huang, and X. Zhu, "Recent advances and challenges in task-oriented dialog systems," *Science China Technological Sciences*, vol. 63, no. 10, pp. 2011–2027, 2020.