Received: 9 December 2022 | Revised: 24 March 2023 | Accepted: 10 April 2023

DOI: 10.1002/stvr.1845

SURVEY ARTICLE

WILEY

The integration of machine learning into automated test generation: A systematic mapping study

Afonso Fontes | **Gregory Gay**

Department of Computer Science and Engineering, Chalmers and the University of Gothenburg, Gothenburg, Sweden

Correspondence
Gregory Gay, Department of Computer Science and Engineering, Chalmers and the University of Gothenburg, Gothenburg, Sweden.
Email: greg@greggay.com

Funding information
Vetenskapsrådet, Grant/Award Number: 2019-05275

Abstract

Machine learning (ML) may enable effective automated test generation. We characterize emerging research, examining testing practices, researcher goals, ML techniques applied, evaluation, and challenges in this intersection by performing. We perform a systematic mapping study on a sample of 124 publications. ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-) Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function. The work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. Our findings can serve as a roadmap and inspiration for researchers in this field.

KEYWORDS

automated test generation, machine learning, test case generation, test input generation, test oracle generation

1 | INTRODUCTION

Software testing is invaluable in ensuring the reliability of the software that powers our society [1]. It is also notoriously difficult and expensive, with severe consequences for productivity, the environment, and human life if not conducted properly. New tools and methodologies are needed to control that cost without reducing the quality of the testing process.

Automation has a critical role in controlling costs and focusing developer attention [2]. Consider test generation—an effort-intensive task where sequences of program *input* and *oracles* that judge the correctness of the resulting execution are crafted for a system-under-test (SUT) [1]. Effective automated test generation could lead to immense effort and cost savings.

Automated test generation is a popular research topic, and outstanding achievements have been made in the area [2]. Still, there are critical limitations to current approaches. Major among these is that generation frameworks are applied in a *general* manner—techniques target simple universal heuristics, and those heuristics are applied in a static manner to all systems equally. Parameters of test generation can be tuned by a developer, but this requires advanced knowledge and is still based on the same universal heuristics. Current generation frameworks are largely unable to adapt their

approach to a particular SUT, even though such projects offer rich information content in their documentation, metadata, source code, or execution logs [3]. Such static application limits the potential effectiveness of automated test generation.

Machine learning (ML) algorithms make predictions by analysing and extrapolating from sets of observations [3]. Advances in ML have shown that automation can match or surpass human performance across many problem domains. ML has advanced the state-of-the-art in virtually every field. Automated test generation is no exception. Recently, researchers have begun to use ML either to *directly* generate input or oracles [4] or to *enhance* the effectiveness or efficiency of existing test generation frameworks [5]. ML offers the potential means to adapt test generation to a SUT, and to enable automation to optimize its approach without human intervention.

We are interested in understanding and characterizing emerging research around the integration of ML into automated test generation.[1] Specifically, we are interested in which testing practices have been addressed by integrating ML into test generation, the goals of the researchers using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how such techniques are trained and validated, and how the whole test generation process is evaluated. We are also interested in identifying the emerging field's limitations and open research challenges. To that end, we have performed a systematic mapping study. Following a search of relevant databases and a rigourous filtering process, we have examined 124 relevant studies, gathering the data needed to answer our research questions.

We observed that ML supports generation of input and oracles for a variety of testing practices (e.g., system or GUI testing) and oracle types (e.g., expected test verdicts and expected output values). During input generation, ML either directly generates input or improves the efficiency or effectiveness of existing generation methods. The most common types of ML are supervised and reinforcement learning. A small number of publications also employ unsupervised or semi-supervised/adversarial learning.

Supervised learning is the most common type for system testing, Combinatorial Interaction Testing, and all forms of oracle generation. Neural networks are the most common supervised techniques, and techniques are evaluated using both traditional testing metrics (e.g., coverage) and ML metrics (e.g., accuracy). Reinforcement learning is the most common ML type for GUI, unit, and performance testing. It is effective for practices with scoring functions and when testing requires a sequence of input steps. It is also effective at tuning generation tools. Reinforcement learning techniques are generally based on Q-Learning, and are generally evaluated using testing metrics tied to the reward function. Finally, unsupervised learning is effective for filtering tasks such as discarding similar test cases.

The publications show great promise, but there are significant open challenges. Learning is limited by the required quantity, quality, and contents of training data. Models should be retrained over time. Whether techniques will scale to real-world systems is not clear. Researchers rarely justify the choice of ML technique or compare alternatives. Research is limited by the overuse of simplistic examples, the lack of standard benchmarks, and the unavailability of code and data. Researchers should be encouraged to use common benchmarks and provide replication packages and code. In addition, new benchmarks could be created for ML challenges (e.g., oracle generation).

Our study is the first to thoroughly summarize and characterize this emerging research field[2] We hope that our findings will serve as a roadmap for both researchers and practitioners interested in the use of ML in test generation and that it will inspire new advances in the field.

2 | BACKGROUND AND RELATED WORK

2.1 | Software testing

It is essential to verify that software functions as intended. This verification process usually involves *testing*—the application of *input*, and analysis of the resulting *output*, to identify unexpected behaviours in the system-under-test (SUT) [1].

During testing, a *test suite* containing one or more *test cases* is applied to the SUT. A test case consists of a *test sequence (or procedure)*–a series of interactions with the SUT–with *test input* applied to some SUT component. Depending on the granularity of testing, the input can range from method calls, to API calls, to actions within a graphical interface. Then, the test case will validate the output against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails [1]. An oracle can be a pre-defined specification (e.g., an assertion), output from a past version, a model, or even manual inspection by humans [1].

[1]We focus specifically on the use of ML to enhance test generation, as part of the broader field of AI-for-Software Engineering (AI4SE). There has also been research in automated test generation for ML-based systems (SE4AI). These studies are out of the scope of our review.

[2]This publication extends an initial systematic literature review [6] that focused only on test oracle generation. Our extended study also includes publications on test input generation and an expanded set of publications for oracle generation. We also include additional and extended analyses and discussion.

```
@Test
public void testPrintMessage() {
    String str = "Test Message";
    TransformCase tCase = new TransformCase(str);
    String upperCaseStr = str.toUpperCase();
    assertEquals(upperCaseStr, tCase.getText());
}
```

FIGURE 1 Example of a unit test case written using the JUnit notation for Java.

An example of a test case, written in the JUnit notation, is shown in Figure 1. The test input is a string passed to the constructor of the `TransformCase` class, then a call to `getText()`. An assertion then checks whether the output matches the expected output—an upper-case version of the input.

Testing can be performed at different granularity levels, using tests written in code or applied by humans. The lowest granularity is unit testing, which focuses on isolated code modules (generally classes). Module interactions are tested during integration testing. Then, during system testing, the SUT is tested through one of its defined interfaces—a programmable interface, a command-line interface, a graphical user interface, or another external interface. Human-driven testing, such as exploratory testing, is out of the scope of this study, as it is often not amenable to automation.

2.2 | Machine learning

Machine learning (ML) constructs models from observations of data to make predictions [3]. Instead of being explicitly programmed like in traditional software, ML algorithms 'learn' from observations using statistical analyses, facilitating the automation of decision making. ML has enabled many new applications in the past decade. As computational power and data availability increase, such approaches will increase in their capabilities and accuracy.

ML approaches largely fall into four categories—supervised, semi-supervised, unsupervised, and reinforcement learning—as presented in Figure 2. In supervised learning, algorithms infer a model from the training data that makes predictions about newly encountered data. Such algorithms are typically used for classification—prediction of a label from a finite set—or regression—predictions in an unrestricted format, for example, a continuous value. For example, a model may be trained from image data with the task of classifying whether an animal depicted in a new image is a cat. If a sufficiently large training dataset with a low level of noise is used, an accurate model can often be trained quickly. However, a model is generally static once trained and cannot be improved without retraining.

Semi-supervised algorithms are a form of supervised learning where feedback mechanisms are employed to automatically retrain models. For example, adversarial networks refine accuracy by augmenting the training data with new input by putting two supervised algorithms in competition. One of the algorithms creates new inputs that mimic training data, while the second predicts whether these are part of the training data or impostors. The first refines its ability to create convincing fakes, while the second tries to separate fakes from the originals. Semi-supervised approaches require a longer training time, but can achieve more optimal models, often with a smaller initial training set.

Unsupervised algorithms do not use previously-labelled data. Instead, approaches identify patterns in data based on the similarities and differences between items. They model the data indirectly, with little-to-no human input. Rather than making predictions, unsupervised techniques aid in understanding data by, for example, clustering related items, extracting interesting features, or detecting anomalies. As an example, a clustering algorithm could take a set of images and cluster them into groups based on the similarity of the images. Such an algorithm could not predict whether a specific image had a cat in it—as was done in supervised learning—but would likely place many of the cat-containing images in the same cluster.

Reinforcement learning algorithms select actions based on an estimation of their effectiveness towards achieving a measurable goal [5]. Reinforcement learning often does not require training data, instead learning through sequences of interactions with its environment. Reinforcement learning 'agents' use feedback on the effect of actions taken to improve their estimation of the actions most likely to maximize achievement of their goal (their 'policy'). Feedback is provided by a reward function—a numeric scoring function. For example, an agent may predict which animal is contained in an image. It would then get a score based on how close its guess was to being correct—for example, if the image contained a cat, then a guess of 'dog' would get a higher score than a guess of 'spider'. The agent can also adapt to a changing environment, as estimations are refined each time an action is taken. Such algorithms are often the basis of automated processes, such as autonomous driving, and are effective in situations where sequences of predictions are required.

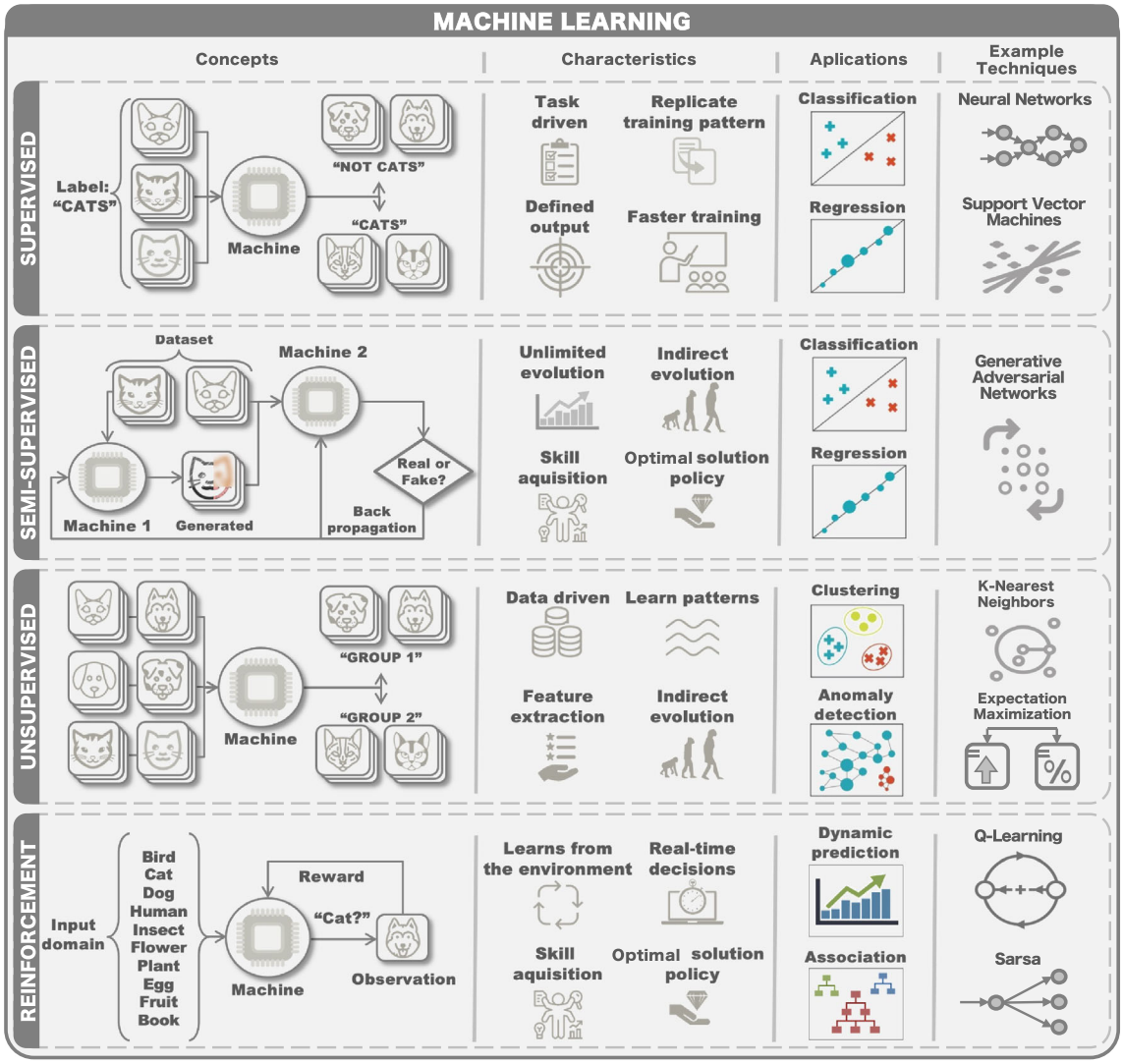


FIGURE 2 Types of ML and their concepts, characteristics, and applications.

Recent research often focuses on 'deep learning'. Deep approaches make complex and highly accurate inferences from massive datasets. Many DL approaches are based on complex many-layered neural networks—networks that attempts to mimic how the human brain works [7]. Such neural networks employ a cascade of nonlinear processing layers where one layer's output serves as the successive layer's input. Deep learning requires a computationally intense training process and larger datasets than traditional ML, but can learn highly accurate models, extract features and relationships from data automatically, and potentially apply models across applications. 'Deep' approaches exist for all four of the ML types discussed above.

2.3 | Common test generation techniques

Many techniques have been used to generate test input. In this subsection, we briefly introduce four common approaches: (a) random test generation, (b) search-based test generation, (c) symbolic execution, and (d) model-based test generation.

In random test generation, input is generated purely at random and applied to the system-under-test with the aim of triggering a failure. Random input generation is one of the most fundamental, simple, and easy-to-implement generation techniques [8]. Unfortunately, while random testing is often very efficient, most software has too large of an input space to exhaustively cover. Therefore, a weakness of random testing is that the generated input may only span a small, and uneven, portion of that input space. Therefore, many *adaptive* random testing techniques have been proposed. In

adaptive random testing, mechanisms are employed to partition the input space, and input is generated for each partition [9]. This ensures an even distribution across the input space.

Search-based test generation formulates input generation as an optimization problem [5]. Of that near-infinite set of inputs for an SUT, we want to identify—systematically and at a reasonable cost—those that meet our goals. Given scoring functions that measure closeness to the attainment of those goals—fitness functions—metaheuristic optimization algorithms can automate that search by selecting input and measuring their fitness. Metaheuristic algorithms are often inspired by natural phenomena. For example, genetic algorithms evolve a population of candidate solutions over many generations by promoting, mutating, and breeding fit solutions. Such techniques retain many of the benefits of random testing, including scalability, and are often better able to identify failure-inducing input [10], or input with other properties of interest [11].

Symbolic execution is a program analysis technique where symbolic input is used instead of concrete input to 'execute' the program [12]. The values of program variables are represented by symbolic expressions over these inputs. Then, at any point during a symbolic execution, the program's state can be represented by these symbolic values of program variables and a Boolean formula containing the collected constraints that must be satisfied for that path through the program to have been taken, also known as the path constraint. By identifying concrete input that satisfies a path constraint, we can ensure that particular paths through the program are covered by test cases. Constraint solvers can be used to identify such input automatically. Recent approaches often are based on dynamic symbolic execution (or concolic execution), where the symbolic execution is combined with concrete random execution to ease the difficulty of solving complex path constraints [8].

Finally, in model-based testing, lightweight models representing aspects of interest of an SUT are used to derive test cases [13]. Often, such models take the form of a state machine. The internal behaviour of the SUT is represented by its state. Transitions are triggered by applying input to the SUT. The model describes—at a chosen level of abstraction—the expected SUT behaviour over a sequence of input actions. Test cases can then be derived from this model by choosing relevant subsets of input sequences [8].

2.4 | Related work

Other secondary studies overlap with ours in scope. We briefly discuss these publications below. Our SLR is the first focused specifically on the application of ML to automated test generation, including both input and oracle generation, and no related study overlaps in full with our research questions. We have also examined a larger and more recent sample of publications.

Durelli et al. performed a systematic mapping study on the application of ML to software testing [3]. Their scope is broad, examining how ML has been applied to any aspect of the testing process. They mapped 48 publications to testing activities, study types, and ML algorithms employed. They observe that ML has been used to generate input and oracles. They note that supervised algorithms are used more often than other ML types and that Artificial Neural Networks are the most used algorithm. Jha and Popli also conducted a short review of literature applying ML to testing activities [14], and note that ML has been used for both input and oracle generation.

Ioannides and Eder conducted a survey on the use of AI techniques to generate test cases targeting code coverage—known as 'white box' test generation [15]. Their survey focuses on optimization techniques, such as genetic algorithms, but they note that ML has been used to generate test input.

Barr et al. performed a survey on test oracles [1]. They divide test oracles into four types, including those specified by humans, those derived automatically, those that reflect implicit properties of programs, and those that rely on a human-in-the-loop. Approaches based on ML fall into the 'derived' category, as they learn automatically from project artefacts to replace or augment human-written oracles. They discuss early approaches to using ML to derive oracles.

Balera et al. conducted a systematic mapping study on hyper-heuristics in search-based test generation [16]. Search-based test generation applies optimization algorithms to generate test input. A hyper-heuristic is a secondary optimization performed to tune the primary search strategy, for example, a hyper-heuristic could adapt test generation to the current SUT. A hyper-heuristic can apply ML, especially RL, but can also be guided by other algorithms. We also observe the use of ML-based hyper-heuristics.

3 | METHODOLOGY

Our aim is to understand how researchers have integrated ML into automated test generation, including generation of input and oracles. We have investigated publications related to this topic and seek to understand their methodology, results, and insights. To gain this understanding, we performed a systematic mapping study according to the guidelines of Petersen et al. [17].

We are interested in assessing the *effect* of integrating ML into the test generation process, understanding the *adoption* of these techniques—how and why they are being integrated, and which specific techniques are being applied, and identifying the potential *impact* and *risks* of this integration. Table 1 lists the research questions we are interested in answering and clarifies the purpose of asking such questions.

The first three questions are high-level questions that clarify how ML has enabled or enhanced test generation, why ML was applied, and which specific testing scenarios were targeted by the enhanced generation techniques. **RQ1** enables us to categorize publications in terms of specific testing practices. By 'testing practices', we refer either to the code or interface level that testing is aimed at (e.g., unit or GUI testing) or to specialized forms of testing (e.g., performance testing). To answer this question, we divide the sampled publications into categories based on the specific goals and targets of test generation. We did not start with pre-decided categories but analysed and thematically grouped publications. **RQ2** is motivational, covering the authors' primary objectives. We are interesting in how the authors intended to use ML—for example, to directly generate input, to enhance an existing test generation technique, or to identify weaknesses in a testing strategy.

In contrast, **RQ3–5** are technical questions. **RQ3** examines the broad category of ML technique (i.e., supervised, unsupervised, semi-supervised, or reinforcement learning), as well as its training and validation processes. **RQ4** examined which specific ML techniques (e.g., backpropagation neural networks) were used to perform the generation task. **RQ5** focuses on how the test generation approach is evaluated, including metrics and types of systems tested. This can include both the generation framework as a whole, or the specific ML aspect of the framework. Finally, the last research question covers the limitations of the proposed approaches and open research challenges (**RQ6**).

To answer these questions, we have performed the following tasks:

1. Formed a list of publications by querying publication databases (Section 3.1).
2. Filtered this list for relevance (Section 3.2).
3. Extracted and classified data from each study, guided by properties of interest (Section 3.3).
4. Identified trends in the extracted data to answer each research question (described along with results in Section 4).

3.1 | Initial study selection

To locate publications for consideration, a search was conducted using four databases: IEEE Xplore, ACM Digital Library, Science Direct, and Scopus. To narrow the results, we created a search string by combining terms of interest on test generation and machine learning. The search string used was

> *('test case generation' OR 'test generation' OR 'test oracle' OR 'test input') AND ('machine learning' OR 'reinforcement learning' OR 'deep learning' OR 'neural network')*

These keywords are not guaranteed to capture all publications on ML in test generation. However, they are intended to attain a relevant sample. Specifically, we combine terms related to test generation and terms related to machine learning, including common technologies. Our focus is not on any particular form of test generation. To

TABLE 1 List of research questions, along with motivation for answering the question.

ID	Research question	Objective
RQ1	Which testing practices have been supported by integrating ML into the generation process?	Highlights testing scenarios and systems types targeted for ML-enhanced test generation.
RQ2	What is the goal of using machine learning as part of automated test generation?	To understand the reasons for applying ML techniques to perform or enhance test generation.
RQ3	What types of ML have been used to perform or enhance automated test generation?	Identifies the type of ML applied, how it was integrated into the generation process, and how it was trained and validated.
RQ4	Which specific ML techniques were used to perform or enhance automated test generation?	Identify specific ML techniques used in the process, including type, learning method, and selection mechanisms.
RQ5	How is the test generation process evaluated?	Describe the evaluation of the ML-enhanced test generation process, highlighting common metrics and artefacts (programs or datasets) used.
RQ6	What are the limitations and open challenges in integrating ML into test generation?	Highlights the limitations of enhancing test generation with ML and future research directions.

obtain a representative sample, we have selected ML terms that we expect will capture a wide range of publications. These terms may omit some in-scope ML techniques, but attain a relevant sample while constraining the amount of manual inspection.

We limited our search to peer-reviewed publications in English. The search string was applied to the full text of articles. Our set of articles was gathered in March 2023, containing an initial total of 3227 articles. This is shown as the first step in Figure 3.

To evaluate the search string's effectiveness, we conducted a verification process. First, we randomly sampled ten entries from the final publication list. Then we looked in each article for ten citations that were in scope, resulting in 100 citations. We checked whether the search string also retrieved these citations, and all 100 were retrieved. Although this is a small sample, it indicates the robustness of the string.

3.2 | Selection filtering

We next applied a series of filtering steps to obtain a focused sample. Figure 3 presents the filtering process and the number of entries after applying each filter. The number in box 1 represents the initial number of articles. The numbers in the other boxes represent the number of entries removed in that particular step. The numbers between the steps show the total number of articles after applying the previous step.

To ensure that publications are relevant, we used keywords to filter the list. We first searched the title and abstract of each study for the keyword 'test' (e.g., 'testing'). We then searched the title and abstract of the remaining publications for either 'learning' or 'neural'—representing application of ML. We merged the filtered lists, and removed all duplicate entries. We then removed all secondary studies. This left 1192 publications.

We examined the remaining publications manually, removing all publications not in scope. Publications were included if they met the following conditions:

- The publication must be written in English.
- The publication must have appeared in a peer-reviewed venue. This includes journals, conferences, and workshops.
- The publication is a primary study (i.e., reporting original research).
- The research reported relates to test generation—including test inputs, oracles, or full test cases—**and** applies any machine learning technique as part of the generation process.

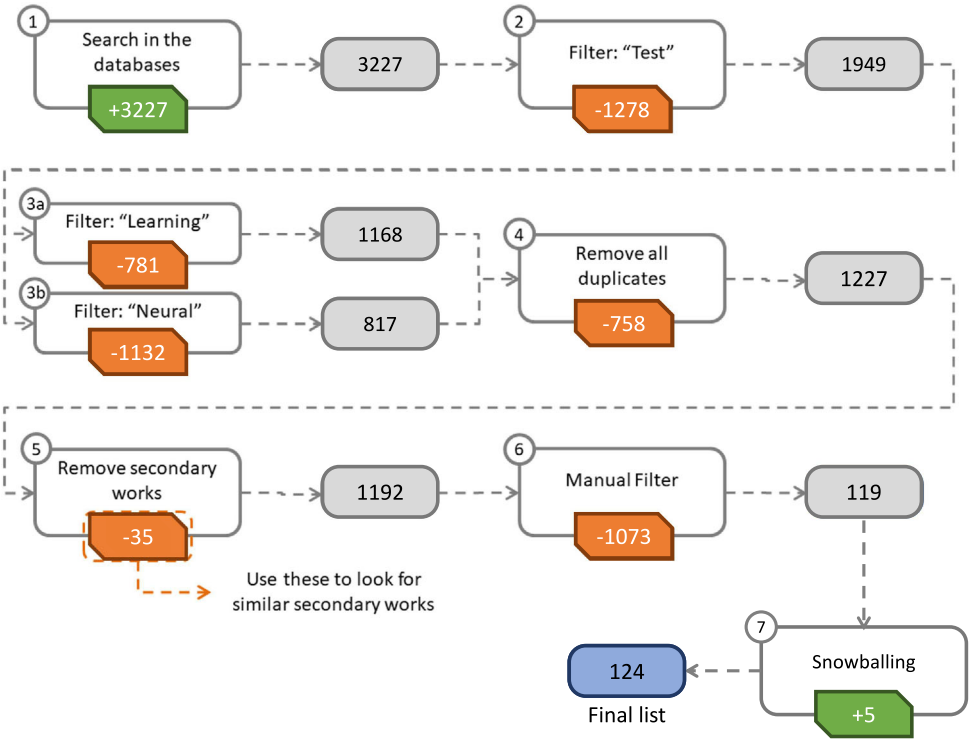


FIGURE 3 Steps taken to determine the final list of publications to analyse.

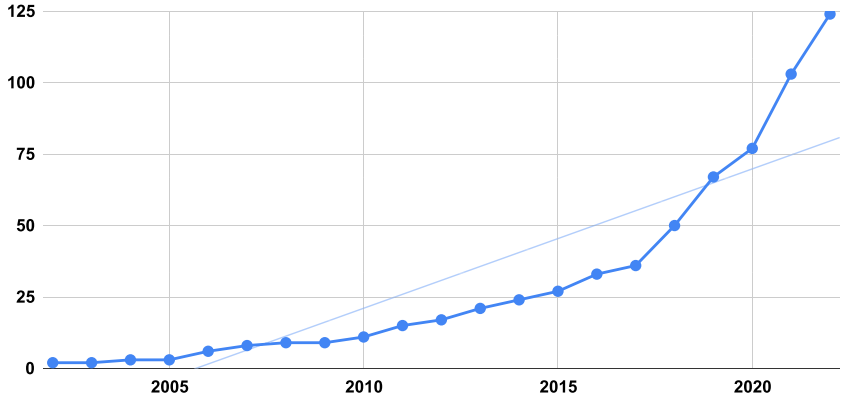


FIGURE 4 Growth of the use of ML in test generation since 2002

Articles were excluded under the following conditions:

- The publication was not written in English.
- The publication was not in a peer-reviewed venue (e.g., book chapters, letters, white or grey literature).
- The publication is a secondary study (e.g., systematic literature review or mapping study). However, such studies were considered for discussion in Section 2.4.
- The reported research does not relate to test generation, or the research is not discussed in the context of test generation.[3]
- The reported research does not apply ML as part of test generation (i.e., a non-ML technique is applied or ML is applied to an activity unrelated to test generation).
- The reported research relates to testing *of* ML-based systems rather than test generation.

This determination was made by first reading the title, abstract, and introduction. Then, if the publication seemed in scope, we proceeded to read the entire study. In a small number of cases, publications were deemed out-of-scope only after inspection of the full article. Both authors independently inspected publications during this step to prevent the accidental removal of relevant publications. In cases of disagreement, the authors discussed the study.

This process resulted in a sample of 119 articles. We then performed snowballing by inspecting the bibliography of each publication and adding any additional publications that met our inclusion criteria stated above. The snowballing process added five additional publications, resulting in a final sample of 124 publications.

The publications are listed in Section 4.2, associated with the specific testing practice addressed. Figure 4 shows the growth of interest in this topic since 2002 (only one study in this sample, from 1993, was published before this date). We can see modest, but growing, interest until 2010. The advancements in ML in the past decade have resulted in significantly more use of ML in test generation, especially starting in 2018. Over 70% of the publications in our sample were published in the past five years alone—with 38% in the past two years. This is an area of growing interest and maturity, and we expect the number of publications to increase significantly in the next few years.

3.3 | Data extraction and classification

To answer the questions in Table 1, we have extracted a set of key properties from each study, identified in Table 2. Each property listed in the table is briefly defined and is associated with the research questions. Several properties may collectively answer a RQ. For example, RQ2—covering the goals of using ML—can be answered using property P2. However, P1 provides context and the testing practice addressed may dictate how ML is applied.

Data extraction was performed primarily by the first author of this study. However, to ensure the accuracy of the extraction process, the second author performed a full independent extraction for a sample of ten randomly-chosen publications. We compared our findings, and found that we had near-total agreement on all properties. The second author then performed a lightweight verification of the findings of the first author for the remaining publications. A

[3]For example, ML could be applied as part of test suite reduction. Test suite reduction can be applied as part of a broader test generation framework, or it can be applied as a standalone testing technique. If the research was presented explicitly as part of test generation, we retained the publication. If the research was presented in a standalone context, then it was discarded.

TABLE 2 List of properties used to answer the research questions. For each property, we include a name, the research questions the property is associated with, and a short description.

ID	Property name	RQ	Description
P1	Testing practices addressed	RQ1, RQ2	The specific type of testing scenarios or application domain focused on by the approach. It helps to categorize the publications, enabling comparison between contributions.
P2	Proposed research	RQ2	A short description of the approach proposed or research performed.
P3	Hypotheses and results	RQ1, RQ3	Highlights the differences between expectations and conclusions of the proposed approach.
P4	ML integration	RQ3	Covers how ML techniques have been integrated into the test generation process. It is essential to understand what aspects of generation are handled or supported by ML.
P5	ML technique applied	RQ4	Name, type, and description of the ML technique used in the study.
P6	Reasons for using the specific ML technique	RQ4	The reasons stated by the authors for choosing this ML technique.
P7	ML training process	RQ4	How the approach was trained, including the specific data sets or arcts used to perform this training. This property helps us understand how each contribution could be replicated or extended.
P8	External tools or libraries used	RQ4	External tools or libraries used to implement the ML technique.
P9	ML objective and validation process	RQ4, RQ5	This attribute covers the objective of the ML technique (e.g., reward function or validation metric), and how it is validated, including data, artefacts, and metrics used (if any).
P10	Test generation evaluation process	RQ5	Covers how the ML-enhanced oracle generation process, as a whole, is evaluated (i.e., how successful are the generated input at triggering faults or meeting some other testing goal?). Allows understanding of the effects of ML on improving the testing process.
P11	Potential research threats	RQ6	Notes on the threats to validity that could impact each study.
P12	Strengths and limitations	RQ6	This property is used to understand the general strengths and limitations of enhancing a generation process with ML by collecting and synthesizing these aspects for both the ML techniques and entire test generation approaches.
P13	Future work	RQ6	Any future extensions proposed by the authors, with a particular focus on those that could overcome the identified limitations.

small number of corrections were discussed between the authors, but the data extraction was generally found to be accurate.

Systematic mapping studies generally address research questions by grouping publications into different *classifications*, then analysing trends in the publications in each group. We likewise group publications in the following ways:

- The testing practice addressed. We first divide the research into input and oracle generation, then a specific input granularity (e.g., unit or system-level input generation) or input/test type (e.g., performance testing) or specific oracle type (e.g., test verdicts, expected output).
- The type of ML applied (e.g., supervised or reinforcement learning).
- The specific ML technique applied (e.g., backpropagation neural network).
- The type of training data used, if applicable (e.g., previous system executions).
- The objective of applying ML. This includes both the type of prediction being made (e.g., classification or regression) and the purpose of the prediction (e.g., predicting the input that will cover a path in the code). For reinforcement learning, this includes the reward functions used.
- The evaluation metrics used to assess the proposed research. This includes both traditional test generation evaluation metrics (e.g., number of faults detected) and ML-related metrics (e.g., accuracy).
- The type of example systems used in the evaluation.

For the ML approach, we use the four primary categories of ML described in Section 2 to classify publications—supervised, semi-supervised, unsupervised, and reinforcement learning. For the other categories, we did not begin with pre-determined classifications. Rather, we performed thematic analysis of the articles to identify natural groupings in the publication sample. In all cases, our goal was to avoid oversimplification—we favoured a large number of specialized classes over a small number of over-arching classes. We describe classification more concretely in Section 4.

4 | RESULTS AND DISCUSSION

In this section, first, we identify the testing practices addressed by ML-enhanced test generation (RQ1, Section 4.1). We then note observations regarding research related to individual testing practices (Section 4.2). Finally, we present answers to RQ2–6 (Sections 4.3–4.7).

4.1 | RQ1: Testing practices addressed

The purpose of RQ1 is to give an overview of which testing practices have been targeted by the publications to help structure our examination of the sampled articles. Our categorization is shown in Figure 5. In this chart, we divide articles into layers, with each layer representing finer levels of granularity. The total number of publications in each category is reported below.

The specific formulation of a test case depends on the product domain and technologies utilized by the SUT [18]. However, broadly, a test case is defined by a set of input steps and test oracles [1], both of which can be the target of automated generation. Therefore, we decided that *input* and *oracles* constitute our first division.

A majority of articles focus on input generation (67% of the sample). Automated input generation has become a major research topic in software testing over the past 20 years [2], and many different forms of automated generation have been proposed, using approaches ranging from symbolic execution [12] to optimization [5]. Oracle generation has long been seen as a major challenge for test automation research [1,2]. However, ML is a realistic route to achieve automated oracle generation [6], and a significant number of publications have started to appear on this topic (33%).

Figure 6a shows the growth in both topics since 2002. Both show a similar trajectory until 2017, with a sharp increase in input generation after. New ML technologies, such as deep learning, and the growing maturity of open-source learning frameworks, such as PyTorch, Keras, and OpenAI Gym, have potentially contributed to this increase.

4.1.1 | Test input generation

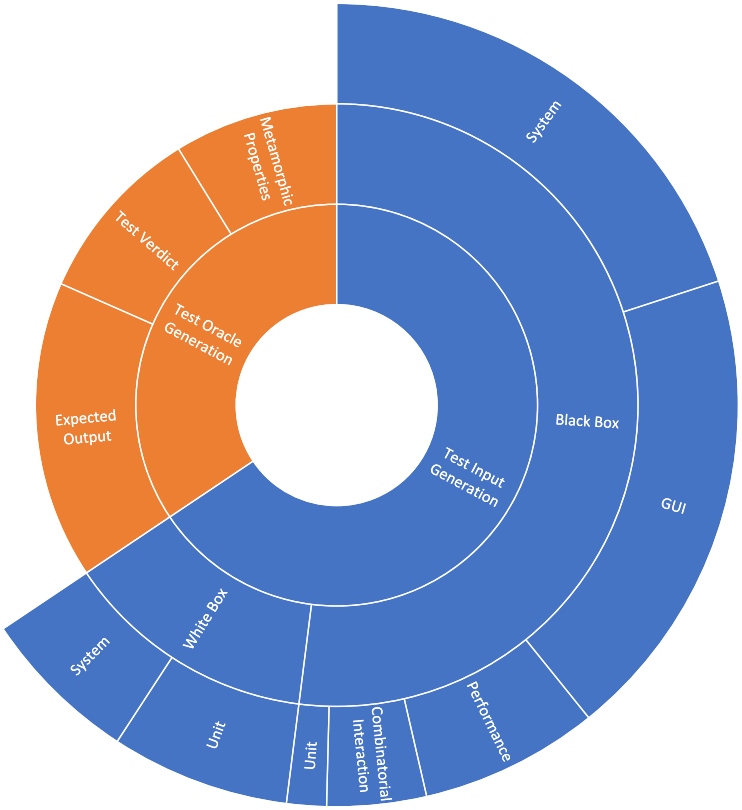
In the second layer of Figure 5, we further divided test input generation by the source of information used to create test input:

- Black Box Testing: Also known as functional testing [18], approaches use information about the program gleaned from documentation, requirements, and other project artefacts to create test inputs.
- White Box Testing: Also known as structural testing [18], approaches use the source code to select test inputs (e.g., generating input that covers a particular outcome for an `if`-statement). Approaches do not require domain knowledge.

Of the 82 publications addressing input generation, 65 propose Black Box and 17 propose White Box approaches. White Box approaches are traditionally common in input generation, as the 'coverage criteria'—checklists of goals [19]—that are the focus of White Box testing offer measurable test generation targets [5]. Such approaches benefit from the inclusion of ML [5]. However, ML may have great potential to enhance Black Box testing. Black Box approaches are based on external data about how the system should behave. ML can be used to automate analyses of that data—enabling new approaches to test generation—as shown by 80% of input generation publications proposing Black Box approaches.

In the third layer of Figure 5, we further subdivided approaches based on either the level of granularity that generated inputs were applied at or by the specialized form of input generated:

- **System Test Generation (33 publications):** A practice where tests target a subsystem or full system through a defined interface (e.g., API or CLI) and verify high-level functionality through that interface.
- **GUI Test Generation (24 publications):** A specialized form of system testing where tests target a GUI to identify incorrect functionality or usability/accessibility issues [20]. We also incorporate game testing (when conducted through a GUI) into this category [21].
- **Unit Test Generation (11 publications):** A practice where test cases target a single class and exercise its functionality in isolation from other classes.
- **Performance Test Generation (9 publications):** Tests are generated to assess whether the SUT meets non-functional requirements (e.g., speed, scalability, or resource usage requirements) [22].
- **Combinatorial Interaction Testing (5 publications):** A system-level practice that attempts to produce a small set of tests that cover important interactions between input variables [23].



Layer #1	
Test Input Generation	82
Test Oracle Generation	42

Layer #2	
Black Box	65
White Box	17
Expected Output	20
Test Verdict	12
Metamorphic Properties	10

Layer #3	
System Test Generation (Black Box)	25
GUI Test Generation	24
Performance Test Generation	9
Combinatorial Interaction Testing	5
Unit Test Generation (Black Box)	2
Unit Test Generation (White Box)	9
System Test Generation (White Box)	8

FIGURE 5 Testing practices addressed by test generation approaches incorporating ML.

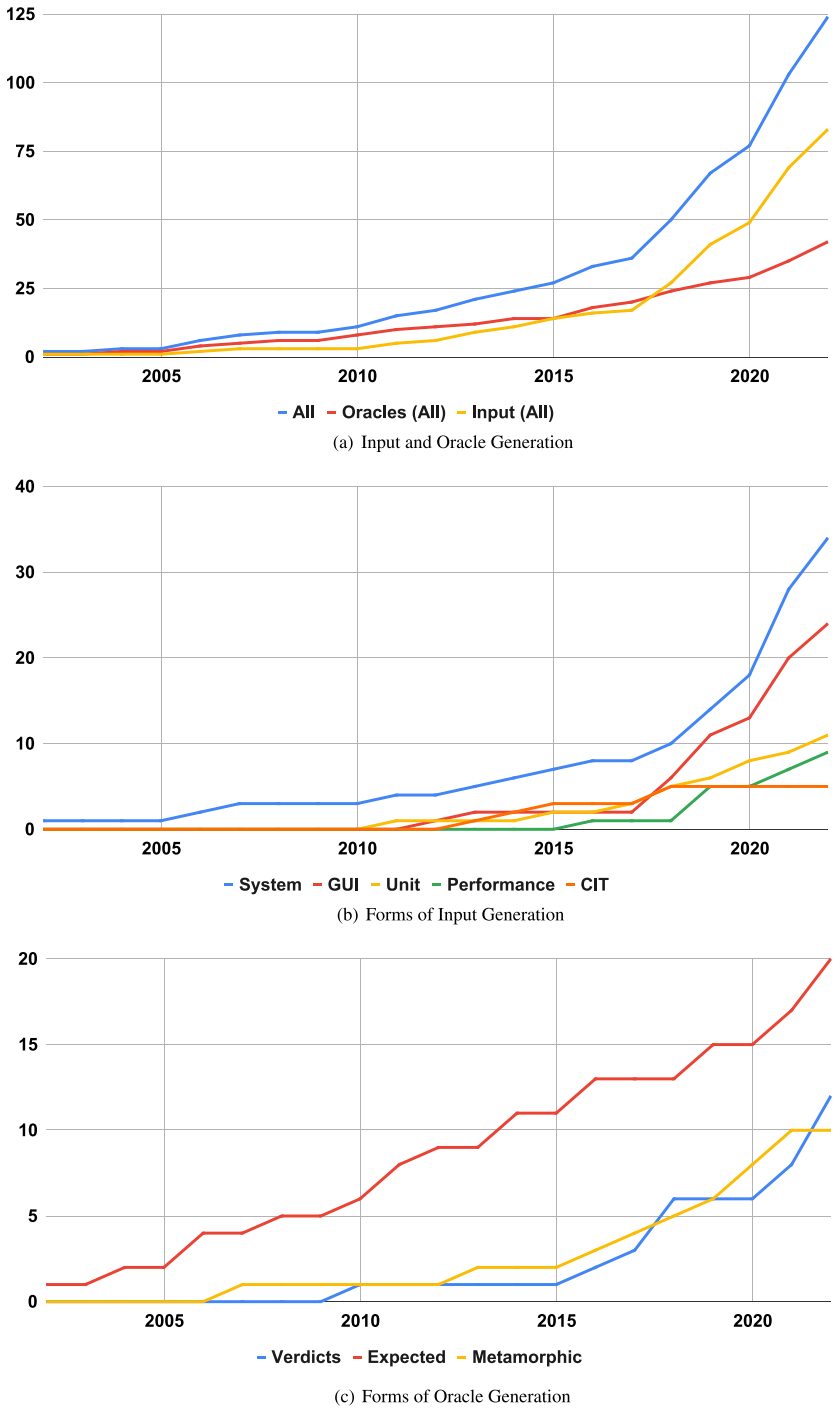
System-level testing is the most common category (41% of input generation), followed by GUI (29%), then unit testing (13%). GUI, performance (11%) and combinatorial interaction testing (CIT) (6%) represent specialized forms of system testing.

Figure 6b shows the growth in each area of input generation. We see a particularly strong growth in system and GUI testing since 2017. In addition to the emergence of open-source ML frameworks, we also hypothesize that this is partially driven by the emergence of mobile and web applications and autonomous vehicles. Mobile applications are tested primarily through a GUI, as are many web applications—leading to increased interest in GUI testing. Only two GUI test generation articles predate 2017, and of the post-2017 articles, 86% relate to testing of either mobile or web applications (see Table 5). Other web applications are tested through REST APIs, and are included in system testing. Autonomous vehicles also require new approaches, as they are tested in complex simulators [24]. Since 2017, web and autonomous vehicle testing constitute the two largest dedicated domains for system testing, with 15% and 19% of post-2017 publications, respectively (see Tables 3 and 4).

4.1.2 | Test oracle generation

The second layer under test oracle generation in Figure 5 divides approaches based on the *type* of test oracle produced:

- **Expected Output (20 publications):** The oracle predicts concrete output behaviour that should result from an input. Often, this will be abstracted (i.e., a *class* of output).



(a) Input and Oracle Generation

(b) Forms of Input Generation

(c) Forms of Oracle Generation

FIGURE 6 Grown in the use of ML in test generation since 2002.

- **Test Verdicts (12 publications):** The oracle predicts the final test verdict for a given input (i.e., a 'pass' or 'fail').
- **Metamorphic Relations and Other Properties of Program Behaviour (10 publications):** A metamorphic relation is a property relating input to expected output [54]—for example, $sin(x) = sin(\pi - x)$. Such properties, as well as other property types that specify the expected behaviour of a SUT, can be applied to many inputs. Violations of such properties identify potential faults.

ML supports decision processes. A ML technique makes a prediction, which can either be a decision or information that supports making a decision. Test oracles follow a similar model, consisting of *information* used to issue a verdict and a *procedure* to arrive at a verdict [55]. ML offers a natural means to replace either component. Test verdict oracles

replace the procedure, while expected output and property oracles support arriving at a verdict. Figure 6c shows steady growth for all types.

RQ1 (Testing Practices): ML supports the generation of both test input and oracles, with a greater focus on input generation (67% of the sample). Input generation research targets system testing, specialized types of system testing (GUI, performance, CIT), and unit testing. The majority of these are Black Box approaches, with White Box approaches primarily restricted to unit testing. There has been an increase in system and GUI input generation since 2017, potentially related to the emergence of web and mobile applications and autonomous driving, as well as to the availability of robust, open-source ML and deep learning frameworks. ML supports generation of test verdict, metamorphic (and other property-based), and expected output oracles.

4.2 | Examining specific practices

Before answering the remaining research questions, we examine concretely how ML has supported test generation.

4.2.1 | System test generation

A total of 33 publications target system testing. Table 3 outline Black Box approaches, while Table 4 outlines White Box approaches. Each table is sorted by ML approach, then by the first author's name. When discussing the objective, we indicate both type of prediction and the purpose of the prediction.

Input generation (supervised, semi-supervised): Supervised approaches generally train models that associate particular SUT input with targeted qualities. Multiple authors use supervised learning to infer a model from execution logs containing inputs and resulting output [34,35,42,56]. The model is used to predict input leading to output of interest. For example, Budnik et al. identify small changes in input that lead to large differences in output, indicating boundary areas where faults are likely to emerge [35]. Both Bergadano and Budnik et al. suggest comparing predictions with real output and using misclassifications to indicate the need to retrain [34,35].

Another concern is achieving code coverage. Majma et al. use supervised learning for both input and oracle generation [19]. A model associates inputs with paths through the source code, then generates new inputs that execute uncovered paths. Similarly, Feldmeier and Fraser generate test inputs for games by targeting code segments, then training neural networks to predict the player actions in particular game states that will cover the associated lines of code [50]. Utting et al. cluster log files—gathered from customer reports—then compare clusters to logs from executing existing test cases to identify weakly-tested areas of the SUT [44]. Supervised learning is used to fill in these gaps. These logs are formatted as vectors of actions, and the model predicts the next input in the sequence.

Others train models to predict which input will fail. Kirac et al. train a model to identify usage behaviours likely to lead to failures using past test cases [39]. Eidenbenz et al. randomly generate a set of inputs, execute them, label the execution based on whether they failed, and then cluster failing instances to enhance accuracy [36]. They train a model using several algorithms, then compare their ability to predict failing input. They propose an iterative process where more training data is added over time, and predictions are verified by developers.

Several authors generate input using models inferred from behavioural specifications (e.g., requirements). The generated input can then show that these specifications are met. Kikuma et al. create a dataset where requirements are tagged with output that should appear if the requirement is met. Their model associates input actions, conditions, and outputs in the requirements, then generates new tests with inputs, conditions, and expected output [38].

Ueda et al. transform specifications, written in natural language, into a structured abstract test recipe that can be concretely instantiated with different input [43]. Meinke et al. model use cases in a constraint language and generate input from the model inferred from the constraints [40]. In addition, both Deng et al. and Zhang et al. generate input for autonomous vehicles intended to violate properties written by human testers [32,33]. They present adversarial scenarios where multiple neural networks manipulate image data used as input to an autonomous driving system. Collectively, these models predict which input will violate properties by, for example, changing day to night or adding rain, and use feedback on their actions to retrain.

Finally, multiple authors generate complex inputs for particular system types. For example, Shrestha [13] train a model to generate valid Simulink models—a visual language for modelling and simulation—for testing tool-chains

TABLE 3 Publications under **System Test Generation (Black Box)** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[25]	2016	Reinforcement	Q-Learning	N/A	Reward (Plan Coverage)	Code Coverage, Assertion Coverage	Robotic Systems
[24]	2021	Reinforcement	Q-Learning	N/A	Reward (Criticality)	Faults Detected	Autonomous Vehicles
[26]	2013	Reinforcement	Delayed Q-Learning	N/A	Reward (Test Improvement)	% of Runs Where Requirements Met	Ship Logistics
[27]	2021	Reinforcement	Q-Learning	N/A	Reward (Code Coverage)	Code Coverage	Triangle Classification, Nesting Structure, Complex Conditions
[28]	2020	Reinforcement	Asynchronous Advantage Actor Critic	N/A	Reward (Transition Coverage)	Not Evaluated	OpenAPI APIs
[29]	2020	Reinforcement	Monte Carlo Control	N/A	Reward (Input Diversity)	Input Diversity, Code Coverage	XML, JavaScript Parsing
[30]	2022	Reinforcement	Deep Q-Network	N/A	Reward (Transition Coverage)	Efficiency, Sensitivity, Transition Cov.	State Machine Benchmark
[31]	2006	Reinforcement	Markov Decision Process	N/A	Reward (State Coverage)	State Coverage	Recycling Robot
[32]	2021	Semi-supervised	Generative Adversarial Network, Convolutional NN	Image Input	Regression (Speed)	Faults Detected	Autonomous Vehicles
[33]	2018	Semi-supervised	Generative Adversarial Network	Image Input	Regression (Steering Angle)	Input Validity, Faults Detected	Autonomous Vehicles
[34]	1993	Supervised	Not Specified	System Executions	Regression (Output)	Not Evaluated	N/A
[35]	2018	Supervised	Backpropagation NN	System Executions	Regression (Output)	Output Coverage	Train Controller
[36]	2021	Supervised	Gaussian Process, Decision Trees, AdaBoostedTree, Random Forest, Support Vector Machine, Artificial NN	System Executions	Regression (Output)	Accuracy	Power Grid Control
[37]	2019	Supervised	Long Short-Term Memory NN	Existing Inputs	Regression (Valid Input)	Accuracy, Code Coverage	FTP Programs
[38]	2019	Supervised	Conditional Random Fields	Test Descriptions	Regression (Requirement Associations)	Accuracy	Telecom Systems
[39]	2019	Supervised	Long Short-Term Memory NN	Existing Inputs	Regression (Failing Input)	Faults Detected, Efficiency	Smart TV
[40]	2021	Supervised	Parallel Distributed Processing	System Executions	Regression (Output)	Efficiency, Faults Detected, Model Size	Autonomous Vehicles

TABLE 3 (Continued)

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[41]	2021	Supervised	Multilayer Perceptron	System Executions	Classification (Input Validity)	Accuracy	REST APIs (GitHub, LanguageTool, Stripe, Yelp, YouTube)
[42]	2022	Supervised	Regression Tree, Feedforward NN	System Executions	Regression (Output)	Efficiency, Faults Detected	Numeric Functions
[13]	2020	Supervised	Long Short-Term Memory NN	Simulink models	Regression (Validity Rules)	Input Validity, Faults Detected	Simulink tools
[43]	2021	Supervised	Conditional Random Fields	Specifications	Regression (Requirement Associations)	Accuracy	Unspecified
[44]	2020	Supervised, Unsupervised	Decision Trees, Gradient Boosting, K-Nearest Neighbour, MeanShift	System Executions	Regression (Validity Rules), Clustering (Covered Input)	Num. Clusters, Accuracy, Event Coverage	Bus System, Supply Chain
[45]	2007	Supervised	Backpropagation NN	System Executions	Regression (Output)	Accuracy, Efficiency	Fault Tolerant System, Arc Length
[46]	2022	Supervised	Shallow NN	RNG Seeds, System Executions	Regression (Prob. Traffic Violation)	Adaptivity, Faults Detected, Sensitivity	Autonomous Vehicles
[47]	2019	Supervised	Support Vector Machine	Existing Inputs	Regression (Validity Rules)	Tests Generated, Tests Executed, Test Size, Faults Detected	Domain-Specific Compiler

Abbreviation: NN, neural network.

based on the language. For compilers, an input is a full program, resulting in a large space of inputs. Zhu et al. restrict the range of inputs to avoid wasted effort [47]. They focus on domain-specific compilers and generate input appropriate for those domains. They extract features from the code, such as number of loops or matrix operations, then train a model to predict whether a new test case belongs to that domain. Test cases not belonging are discarded. Protocols require textual input that conforms to a specified format. Often, determining conformance requires manual construction of a grammar. Gao et al. generate protocol test input without a pre-defined grammar [37]. Their model learns the probability distributions of every character of a message, enabling the generation of new valid text sequences.

Input generation (reinforcement learning): Araiza-Illan et al. [25], Huurman et al. [28], Shu et al. [30], and Veanes et al. [31] use reinforcement learning to generate input to cover states or transitions of a model. Araiza-Illan et al. generate input for robots [25]. The agent explores the robot's environment, using coverage of plan models as the reward function. Huurman et al. model APIs as stateful systems—where requests trigger transitions—and generate API calls intended to cover all transitions in the model [28]. Shu et al. [30] and Veanes et al. [31] use reinforcement learning to choose input actions for state-based system models. These tests can then be applied to the real system.

Baumann et al. use reinforcement learning to select input for autonomous driving that violates critical requirements [24]. The reward function encapsulates headway time, time-to-collision, and required longitudinal acceleration to avoid a collision.

Reddy et al. use reinforcement learning to generate valid complex inputs (e.g., structured documents) [29]. The reward function favours both unique and valid input. As uniqueness depends on previously-generated input, this is not a problem that can easily be solved with supervised learning.

Enhancing test generation: Rather than fully replacing existing test generation methods, ML can also be used to improve their efficiency or effectiveness. A common target for improvement are Genetic Algorithms. A Genetic

TABLE 4 Publications under **System Test Generation (White Box)** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[48]	2021	Reinforcement	ReLU Q-Learning	Constraints	Reward (Solving Cost)	Code Coverage, Queries Solved	GNU coreutils
[49]	2021	Reinforcement	Deep Q-Network	N/A	Reward (Code Coverage, Path Length)	Code Coverage	Sorting
[50]	2022	Supervised	Artificial NN	Game States	Regression (Input Action)	Code Coverage, Mutation Score	Scratch Games
[51]	2022	Supervised	Radial-Basis Function NN	Existing Inputs, Code Coverage	Regression (Code Coverge)	Code Coverage	Numeric Functions
[12]	2021	Supervised	Long Short-Term Memory NN, Tree-LSTM, K-Nearest Neighbour	Constraints	Regression (Solving Time)	Accuracy, Constraint Solving Time	GNU coreutils, Busybox utils, SMT-COMP
[19]	2014	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage	Binary Search, Sorting, Median, GCD, Triangle Class.
[52]	2011	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage	Triangle Classification
[53]	2022	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage, Efficiency	Numeric Functions

Abbreviation: NN, neural network.

Algorithm is a search-based method that generates test cases intended to maximize or minimize a fitness function—a domain-specific scoring function, like the reward function in reinforcement learning.

Buzdalov and Buzdalova use reinforcement learning to modify the fitness function, adding and tuning sub-objectives that assist in optimizing the core objective of the search [26]. Zhao and Lv replace the fitness function with a model that predicts which input will cover unseen output behaviours [45]. Liu et al., Mishra et al., and Shihao ey al. also replace the fitness function, training a model to predict which code will be covered by input [51–53]. These models would be used when there is no tool support to measure coverage, or in cases where measuring coverage would be expensive.

Esnaashari and Damia use reinforcement learning to manipulate tests within the population generated by the Genetic Algorithm by modifying their input [27]. Paduraru et al. similarly use reinforcement learning to improve the effectiveness of a random testing tool by taking generated input and modifying it to raise its coverage or execution path length [49]. Zhong et al. bias input selection in a fuzzing tool for autonomous driving simulators by learning which seeds for the random number generator are more likely to lead to traffic violations in the simulation [46]. Sharma et al. replace random input generation in property-based testing with a model inferred from system executions [42]. The model is submitted, along with properties of interest, to a SMT solver to find input potentially violating the properties.

Mirabella et al. train a model to predict input validity, allowing a generation framework to filter invalid input before applying it [41].

Luo et al. [12] and Chen et al. [48] both enhance constraint solving in symbolic execution. Normally, a fixed timeout is used. Luo et al. instead train a model using multiple methods to predict the time needed to solve a constraint [12]. Chen et al. use reinforcement learning to identify the optimal solving strategy for a constraint [48].

4.2.2 | GUI test generation

Table 5 details the 24 GUI testing publications. GUI test generation often focuses on a state-based interface model that formulates display changes as transitions taken following input. Fifteen publications generated input covering this model.

Almost all publications adopted reinforcement learning, as it can learn from feedback after applying an action to the GUI, and many GUIs require a sequence of actions to access certain elements. The main difference between publications lies in the reward function. Many base the reward on coverage of the states of the interface model (e.g., Yasin et al. [69]), while incorporating additional information to bias state selection. Additional factors include magnitude of the state change [60], usage specifications [63,64], unique code functions called [65], curiosity—favouring exploration of new elements [66,67,70]—coverage of interaction methods (e.g., click, drag) [61], validity of the resulting state [67], and avoidance of navigation loops [59].

TABLE 5 Publications under **GUI Test Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[57]	2018	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	State Coverage	F-Droid
[20]	2021	Reinforcement	Monte Carlo Tree Search, Sarsa	N/A	Reward (Test Goal Coverage)	Faults Detected	2D Games
[58]	2021	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	Qualitative	Resource Planning
[59]	2013	Reinforcement	Own Technique	N/A	Reward (State Coverage, Loop Interactions)	State Coverage	F-Droid
[60]	2021	Reinforcement	Deep Q-Network	N/A	Reward (State Change Magnitude)	Code Coverage, Faults Detected	F-Droid
[61]	2019	Reinforcement	Q-Learning	N/A	Reward (State Cov., Element Interaction)	State Coverage	F-Droid
[62]	2022	Reinforcement	Sarsa	N/A	Reward (Event Cov.)	Code Coverage	F-Droid
[63]	2020	Reinforcement	Double Q-Learning	N/A	Reward (State Cov., Specifications)	State Coverage	F-Droid
[63]	2021	Reinforcement	Double Q-Learning	N/A	Reward (Specifications)	Faults Detected	F-Droid
[64]	2018	Reinforcement	Q-Learning	N/A	Reward (State Cov., Specifications)	State Coverage	F-Droid
[65]	2012	Reinforcement	Q-Learning	N/A	Reward (State Cov., Calls)	State Coverage	Password Manager, PDF Reader, Task List, Budgeting
[66]	2020	Reinforcement	Q-Learning + Long Short-Term Memory	N/A	Reward (State Cov., Curiosity)	State Coverage	F-Droid, Other Android Apps
[67]	2022	Reinforcement	Q-Learning	N/A	Reward (Curiosity, Validity of Resulting State)	Code Coverage, Faults Detected, Input Diversity, State Coverage	Web Apps
[68]	2018	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	State Coverage	Android Apps
[69]	2021	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	Code Coverage, Faults Detected	F-Droid
[70]	2021	Reinforcement	Q-Learning	N/A	Reward (State Cov., Curiosity)	Code Coverage, Faults Detected, Scalability	Web Apps (Research, Real-World, Industrial)

(Continues)

TABLE 5 (Continued)

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[21]	2019	Reinforcement	Advantage Actor-Critic	N/A	Reward (Game-Specific)	Faults Detected, State Coverage, Code Coverage	Games
[71]	2019	Supervised	Feedforward NN	Generated Inputs	Regression (Output)	State Coverage	Login Web App
[72]	2022	Supervised	Residual NN, Transformer	UI Screenshots, Natural Language	Classification (UI Elements), Regression (Natural Language Test)	Accuracy, Flakiness, Input Validity	Android Apps
[73]	2022	Supervised	Residual NN, Transformer	UI Screenshots, Natural Language	Classification (UI Elements), Regression (Natural Language Test)	Accuracy, Efficiency, Flakiness, Input Validity	Web Apps
[7]	2019	Supervised	Deep NN	System Executions	Regression (Action Probability)	State Coverage	Android Apps
[74]	2018	Supervised	Recurrent NN	Existing Inputs	Regression (Test Flows)	State Coverage	Unspecified Web App
[75]	2019	Supervised	Random Forest	Web Pages	Classification (Page Elements)	Mutation Score	Task List, Job Recruiting Web Apps
[76]	2021	Supervised	UNet	Screenshots	Regression (Relevant Screen Areas)	Adaptivity, Code Coverage	Androtest, Web Apps

Abbreviation: NN, neural network.

Rather than state coverage, Koroglu and Sen base reward on finding violations of specifications [63]. Ariyurek et al. also apply reinforcement learning to select input for grid-based 2D games [20]. The game state is represented as a graph, and 'test goals' are synthesized from the graph. The reward emphasizes test goal coverage. Li et al. use supervised learning, training a model to mimic patterns from interaction logs [7]. Their model associates GUI elements with a probability of usage—using probabilities to bias action selection. Kamal et al. filter redundant test cases as part of enhancing a search-based test generation framework [71]. Their model associates input and output, then uses the predicted output to decide if tests are redundant.

Santiago et al. use supervised learning to generate sequences of interactions [74,75]. They trained using human-written interaction sequences spanning several web pages. Their second study extends the approach to interact with forms by extracting feedback messages from the forms [75]. The framework learns constraints for form input, and a constraint solver creates input that meets those constraints. A model is used to classify page components. This helps control how different component types are processed. Their approach requires a complex training phase and a large human-created dataset. However, models can be used for multiple websites, decreasing the training burden.

Khaliq et al. employ multiple forms of supervised learning to generate test cases for Android apps [72] and web apps [73] They use an object recognition model to detect UI elements, then use extracted data from the UI elements to prompt a transformer model for a natural language test description. They then use a parser to translate this description into an executable test case. Similarly, Yazdani et al. have trained a model to identify the UI elements relevant to a particular input action, and use the identified elements to focus random test generation for Android apps [76]. They also demonstrated that the model can be effectively transferred to web apps.

Zheng et al. use both search-based test generation and a deep reinforcement learning technique to generate input for games [21]. Test input is generated by the search algorithm. However, reinforcement learning is used to select policies that control the generation framework.

4.2.3 | Unit test generation

Because unit testing focuses on individual classes—making domain concerns less applicable—the majority of publications in Table 6 are 'White Box' approaches and are not tied to particular system types.

TABLE 6 Publications under **Unit Test Generation** with publication date, generation approach, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	Test gen. approach	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[9]	2017	Black	Supervised	Query Strategy Framework	System Executions	Regression (Output)	Mutation Score	Math Library, Time Library
[77]	2018	Black	Unsupervised	Backpropagation NN	Existing Inputs	Clustering (Input Similarity)	Not Evaluated	N/A
[78]	2020	White	Reinforcement	Upper Confidence Bound, Differential Semi-Gradient Sarsa	N/A	Reward (Input Diversity)	Input Diversity, Faults Detected	JSON Parser
[5]	2020	White	Reinforcement	Upper Confidence Bound, Differential Semi-Gradient Sarsa	N/A	Reward (Num. Exceptions)	Num. Exceptions, Faults Detected	Defects4J
[79]	2022	White	Reinforcement	Upper Confidence Bound, Differential Semi-Gradient Sarsa	N/A	Reward (Num. Exceptions, Input Diversity, Strong Mutation)	Num. Exceptions, Input Diversity, Mutation Score, Faults Detected	Defects4J
[80]	2011	White	Reinforcement	Not Specified	N/A	Reward (Code Coverage)	Code Coverage	Data Structures
[81]	2015	White	Reinforcement	Q-Learning	N/A	Reward (Code Coverage)	Code Coverage	Data Structures, Collection Library, Primitives Library, Java/XML Parsers
[4]	2018	White	Reinforcement	Double Deep Q-Network	N/A	Reward (Code Coverage)	Code Coverage, Efficiency	GCD, EXP, Remainder
[82]	2022	White	Supervised	Naive Bayes, Random Forest, SVM, J48	Existing Inputs	Classification (Code Coverage)	Accuracy, Code Coverage, Mutation Score, Efficiency	Numeric Functions, Wheel Brake, Rendering, Mine Control, Notification, XML Parser, Siemens Benchmark
[83]	2021	White	Supervised	Gradient Boosting	Code Metrics	Classification (Fault Prediction)	Accuracy, Faults Detected	Compression, Imaging Library, Math Library, NLP, String Library
[84]	2019	White	Supervised	Backpropagation NN	Existing Inputs	Regression (Code Coverage)	Not Evaluated	N/A

Abbreviation: NN, neural network.

Groce [80] and Kim et al. [4] use reinforcement learning to generate input, with code coverage as the reward. Groce applies reinforcement learning to generate input directly [80]. In contrast, Kim et al. use reinforcement learning to generate optimization-based input generation algorithms [4]. The agent manipulates heuristics controlling the search algorithms.

Walkinshaw and Fraser use a supervised approach to generate input for system parts that have only been weakly tested [9]. A model is trained to predict the output. The model will have more confidence in prediction accuracy for input similar to the training data. Input with low certainty is retained, as they are likely to test parts of the system ignored in the training data. These inputs can later be used to retrain the model, shifting focus to other parts of the system.

Many authors use ML to enhance existing test generation approaches—often based on Genetic Algorithms. Almulla and Gay use reinforcement learning to select which fitness functions will be optimized in service of a higher-level testing goal [5,78,79]. For example, the agent can learn which combinations of fitness functions best trigger exceptions [5], increase input diversity [78], or increase Strong Mutation Coverage [79]. He et al. use reinforcement learning to improve coverage of private and inherited methods by augmenting generated tests [81]. The agent can make two types of changes—it can replace a method call with one whose return type is a subclass of the original method's, and it can replace a call to a public method with a call to a method that calls a private method. The reward is focused on private method coverage. Chen et al. employ supervised learning to improve the effectiveness of random generation and concolic testing [82]. They generate classification models for particular branches in the code, and use predictions about whether a test will cover a branch to ease the constraint-solving process.

Hershkovich et al. predict whether a class is likely to be faulty [83]. This can improve generation efficiency by determining which classes to target. They train a model—using an ensemble of methods—using source code metrics, labelled on whether a class had faults. Ji et al. use supervised learning to replace a fitness evaluation in a Genetic Algorithm [84]. They focus on data-flow coverage, which is very expensive to calculate. The model replaces the need to actually measure coverage. Hooda et al. train a model to cluster test cases [77]. When new tests are generated, those too close to a cluster centroid are rejected, improving generation efficiency.

4.2.4 | Performance test generation

Performance test generation refers to the generation of test cases for the purpose of assessing whether the SUT meets non-functional requirements, such as speed, response time, scalability, or resource usage requirements [22]. Such tests are often generated to identify and eliminate performance bottlenecks. Table 7 details the performance testing publications. Performance can be measured, which offers feedback for subsequent rounds of generation. Thus, the majority of approaches are based on iterative processes, including reinforcement [22,85–87], rule [88], and adversarial learning [89].

Ahmad et al. generate input intended to expose performance bottlenecks, with reward based on maximized execution time [22].

They note room for improvement by integrating other performance indicators into the reward. Rather than generating explicit program input, Moghadam et al. apply reinforcement learning to control the execution environment [85,87,90]. They identify resource configurations (CPU, memory, disk) where timing requirements are violated, with reward based on response time deviation and resource usage. These environmental factors constitute 'implicit' input that can change the behaviour of the SUT.

Sedaghatbaf et al. generate input violating performance requirements using two competing neural networks [89]. The generator produces input, and the discriminator classifies whether input violates requirements. This feedback improves the generator. Chen et al. also employ adversarial learning to generate input for resource-constrained neural networks intended to expose performance bottlenecks for such networks [91].

Luo et al. use the RIPPER rule learner to identify input classes that trigger intensive computations [88]. When tests are executed, executions are clustered based on execution time. RIPPER learns and iteratively refines rules differentiating the clusters, which are then used to generate new input.

Schulz et al. generate workloads for load testing [92]. The model generates realistic load levels on a system at various times and scenarios. Past session logs are clustered, and a multivariate time series is applied to predict system load during a scenario. Finally, Koo et al. use reinforcement learning to improve symbolic execution during stress testing [86]. They identify input that triggers worst-case execution time, defined as inputs that trigger a long execution path. The agent controls the exploration policy used by symbolic execution so that long paths are favoured. The reward is based on path length and the feasibility of generating input for that path.

4.2.5 | Combinatorial interaction testing

Table 8 shows the publications that use ML as part of Combinatorial Interaction Testing (CIT). Mudarakola et al. [93,94] and Patil and Prakash [95] use neural networks to generate covering arrays—minimal sets of tests that cover all pairwise interactions between input variables. Patil and Prakash [95] predict the interactions covered by an input. They use this model to identify a covering array. Mudarakola et al [94] map each hidden layer of a neural network to a

TABLE 7 Publications under **Performance Test Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[22]	2019	Reinforcement	Duelling Deep Q-Network	N/A	Reward (Execution Time)	Identified Bottlenecks	Auction Website
[85]	2019	Reinforcement	Q-Learning	N/A	Reward (Response Time Deviation)	Not Evaluated	N/A
[86]	2019	Reinforcement	Q-Learning	N/A	Reward (Path Length, Feasibility)	Paths Explored, Efficiency	Biological Computation, Parser, Sorting, Data Structures
[87]	2019	Reinforcement	Q-Learning	N/A	Reward (Response Time Deviation)	Not Evaluated	N/A
[90]	2022	Reinforcement	Q-Learning + Fuzzy Logic	N/A	Reward (Response Time Deviation, Resource Usage)	Efficiency, Adaptivity	Resource-Sensitive Programs (e.g., compression)
[91]	2022	Semi-Supervised	Generative Adversarial Network	System Executions	Regression (Performance), Classification (Input Distribution)	Identified Bottlenecks, Efficiency, State Coverage, Model Sensitivity, Input Quality	Object Recognition
[89]	2021	Semi-Supervised	Conditional Generative Adversarial Network	System Executions	Regression (Perf. Requirements), Classification (Test Realism)	Identified Bottlenecks, Accuracy, Labelling and Training Effort	Auction Website
[88]	2016	Supervised	RIPPER	System Executions	Regression (Rule Learning)	Identified Bottlenecks	Insurance, Online Stores, Project Management
[92]	2021	Supervised	Multivariate Time Series	Session Logs	Regression (Load)	Accuracy	Student Information

Abbreviation: NN, neural network.

TABLE 8 Publications under **Combinatorial Interaction Testing** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metrics	Evaluated on
[23]	2015	Reinforcement	SOFTMAX	N/A	Reward (Input Combinations)	Covering Array Size, Efficiency	Misc. Synthetic, Real Systems
[93]	2018	Supervised	Artificial NN	Specifications	Other (Structure Input Space), Regression (Output)	Covering Array Size	Temperature Monitoring
[94]	2014	Supervised	Artificial NN	Pairwise Input Combinations	Other (Structure Input Space)	Covering Array Size	Web Apps
[95]	2018	Supervised	Artificial NN	Pairwise Input Combinations	Regression (Input Coverage)	Covering Array Size, Efficiency	Unspecified
[96]	2013	Unsupervised	Expectation-Maximization	System Executions	Clustering (Code Coverage)	Qualitative Analysis	Bubble Sort, Math Functions, HTTP Processing, Banking

Abbreviation: NN, neural network.

variable and each node to a value class. The values are connected by their connection to other variables. They do not use the network for prediction, but as a structuring mechanism to generate a covering array. Code coverage is used to prune redundant test cases. In a follow-up study [93], they manually construct a network using requirements, linking outputs to input values, with each input node mapping to an input variable, hidden layers linked to conditions from the

requirements, and output nodes linked to predicted SUT output. The network again provides structure—a covering array is generated based on paths through the network.

Jia et al. use reinforcement learning to tune the generation strategy of a search-based generation framework using Simulated Annealing [23]. The agent selects how Simulated Annealing mutates a covering array. The reward is based on the change in coverage of combinations after imposing a mutation. Their framework recognizes and exploits policies that improve coverage.

CIT assumes that input values are divided into classes. Division is generally done manually, but identifying divisions is non-trivial. Duy Nguyen and Tonella use clustering to identify value classes, based on executed code lines (and how many times lines were executed) [96].

4.2.6 | Test oracle generation

Tables 9–11 summarize the 42 oracle generation publications. Almost all approaches adopt supervised learning. These approaches train models, which stand in for traditional oracles, using previous system executions, screenshots, or metadata about source code features. The model predicts the correctness of output or properties of the expected output.

Test verdicts: The majority of studies employ neural networks to train models that directly predict whether a test should pass or fail [97–103]. Most are simple, traditional neural networks for simple programs. However, Ibrahimzada et al. and Tsimpourlas et al. have recently explored how deep learning can train models for complex programs [98,102,103]. Braga et al. also are able to generate models for a complex application using an ensemble technique [104].

Chen et al. train a model to identify rendering errors in video games by training on screenshots of previous faults [105]. Rafi et al. apply a similar process to identify object-placement errors in augmented reality apps [108]. They do not predict a concrete pass/fail verdict, as users may perceive object placement differently. Instead, when labelling data, they asked multiple humans to offer verdicts, then labelled examples with the percentage that responded with a pass verdict. The model, then, predicts the percentage of users that would see a placement as correct in a new screenshot.

Khosrowjerdi et al. combine supervised learning and model checking [107]. A model is learned from system executions that predicts output. Given the model and specifications, a model checker assesses whether each specification is met, yielding a verdict. For each violation, a test is generated that can be executed to confirm the fault. If the fault is not real, the test and its outcome can be used to retrain the model. In a follow-up study [106], they demonstrate their technique on systems-of-systems.

Expected output: The approaches generally train on system executions, and then predict the specific output expected for a new input. Output is often abstracted to representative values or limited to functions with enumerated values, rather than specific output. For example, a common application is 'triangle classification'—a classification of a triangle as scalene, isosceles, equilateral, or not-a-triangle. This function is often used as an initial demonstration for test generation algorithms because it has branching behaviour. Because it has limited outputs, it is also a common target for demonstrating the potential of oracle generation. Zhang et al. model a function that judges whether an integer is prime—a binary classification problem [127]. Many others also generate oracles for applications with a limited range of output [112,118–122]. However, some authors have generated oracles for functions with unconstrained—for example, integer—output [110,113,114,116,124,125].

The majority of approaches used some form of neural network [19,109,115–121,123–125,127]. Ding and Zhang [112] also used label propagation—a technique where labelled and unlabelled training data are used, and the algorithm propagates labels to similar, unlabelled data—to reduce the quantity of labelling to create the training data. Recently, Dinella et al. [111] and Yu et al. [126] demonstrated the use of language-generating transformer models for test oracle creation. Rather than inferring a model from system executions, a model is trained instead on source and test code, then given the code-under test and/or a partial unit test, the model directly produces assertions predicted to be appropriate for the prompt. Such models are trained on large datasets of code from many projects, and can potentially be applied generally.

Metamorphic and properties: Several publications build on the research of Kanewala and Bieman [131], whose approach (a) converts code into control-flow graphs, (b) selects code elements as features for a data set, and (c), trains a model that predicts whether a feature exhibits a particular metamorphic relation from a list. This requires training data where features are labelled with a classification based on whether or not they exhibit a particular relation. Kanewala et al. extended this work by adding a graph kernel [132]. Hardin and Kanawala adapted this approach for label propagation [54]. Zhang et al. extended the approach to a multi-label classification that can handle multiple metamorphic relations at once [136]. Finally, Nair et al. demonstrated how data augmentation can enlarge the training dataset using mutants as the source of additional training data [134].

Korkmaz and Yilmaz predict the conditions on screen transitions in a GUI [133]. Their model is trained using past system execution and potential guard conditions. Shu and Lee use supervised learning to assess security properties of

TABLE 9 Publications under **Test Verdicts Test Oracle Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metric	Evaluated on
[104]	2018	Supervised	Adaptive Boosting	System Executions	Classification (Verdict)	Mutation Score	Shopping Cart
[105]	2021	Supervised	Convolutional NN	Screenshots	Classification (Verdict)	Accuracy, Faults Detected	Games (Android, iOS)
[97]	2018	Supervised	Backpropagation NN	System Executions	Classification (Verdict)	Mutation Score	Embedded Software
[98]	2022	Supervised	Recurrent NN	Source/Test Code, System Executions	Classification (Verdict)	Accuracy, Efficiency, Faults Detected, Mutation Score	Defects4J
[99]	2022	Supervised	Artificial NN	System Executions	Classification (Verdict)	Correct Classifications	Not Specified
[106]	2018	Supervised	L*	System Executions	Classification (Verdict)	Faults Detected, Efficiency	Platoon Simulator
[107]	2017	Supervised	Not Specified	System Executions	Classification (Verdict)	Faults Detected	Automotive Applications
[100]	2016	Supervised	Multilayer Perceptron	System Executions	Classification (Verdict)	Accuracy	User Creation
[108]	2022	Supervised	Convolutional NN, Multilayer Perceptron	Screenshots	Regression(Deviation from Correctness)	Accuracy	Augmented Reality Apps
[101]	2010	Supervised	Backpropagation NN	System Executions	Classification (Verdict)	Mutation Score	Student Registration
[102]	2021	Supervised	Multilayer Perceptron, Long Short-Term Memory NN	System Executions	Classification (Verdict)	Accuracy, Training Data Size	Blockchain Module, Deep Learning Module, Encryption Library, Stream Editor
[103]	2022	Supervised	Multilayer Perceptron, Long Short-Term Memory NN	System Executions	Classification (Verdict)	Accuracy, Adaptivity, Training Data Size	Blockchain Module, Deep Learning Module, Encryption Library, Network Protocols, Stream Editor, String Library

Abbreviation: NN, neural network.

protocols [135]. A protocol is specified using a state machine, and message confidentiality is assessed on message reachability. A model is inferred, then assessed for violations. If a violation is found, input is produced to check against the implementation. If the violation is false, the test helps retrain the model.

Hiremath et al. predict metamorphic relations for ocean modelling [128,129]. The reinforcement learning approach poses relations, evaluates whether they hold, and attempts to minimize a cost function based on the validity of the set of proposed relations. Spieker and Gotlieb use reinforcement learning to *select* metamorphic relations from a superset of potentially-applicable relations [130]. Their approach evaluates whether selected relations can discover faults in an image classification algorithm.

4.3 | RQ2: Goals of applying ML

Table 12 lists the goals of authors in adopting ML, sorted into three broad categories. In the first two, ML is used directly to generate input or an oracle. As previously discussed, oracle generation uses ML to predict output, to properties of output, or a test verdict. Regarding input generation, the most common goal is to use ML to increase coverage

TABLE 10 Publications under **Expected Output Test Oracle Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metric	Evaluated on
[109]	2004	Supervised	Backpropagation NN	System Executions	Classification (Output)	Correct Classifications	Triangle Classification
[110]	2021	Supervised	Regression Tree, Support Vector Machine, Ensemble, RGP, Stepwise Regression	System Executions	Regression (Time)	Accuracy	Elevator
[111]	2022	Supervised	Transformer	Source/Test Code	Regression (Assertions)	Faults Detected, Accuracy	Defects4J
[112]	2016	Supervised	Support Vector Machine	System Executions	Classification (Output)	Mutation Score	Image Processing
[113]	2021	Supervised	Regression Tree, Support Vector Machine, Ensemble, TRGP, Stepwise Regression	System Executions	Regression (Time)	Accuracy	Elevator
[114]	2022	Supervised	Regression Tree, Support Vector Machine, Ensemble, RGP, Stepwise Regression	System Executions	Regression (Waiting, Execution Time)	Mutation Score, Accuracy	Elevator
[115]	2008	Supervised	Backpropagation NN	System Executions	Classification (Output)	Correct Classifications	Triangle Classification
[19]	2014	Supervised	Backpropagation NN	System Executions	Classification (Output)	Faults Detected	Static Analysis
[116]	2019	Supervised	Deep NN	System Executions	Regression (Output)	Mutation Score	Mathematical Functions
[117]	2011	Supervised	Radial-Basis Function NN	System Executions	Regression (Output)	Correct Classifications	Triangle Classification
[118]	2011	Supervised	Multilayer Perceptron	System Executions	Classification (Output)	Mutation Score	Insurance Application
[119]	2012	Supervised	Multilayer Perceptron	System Executions	Classification (Output)	Mutation Score	Insurance Application
[120]	2010	Supervised	Artificial NN	System Executions	Classification (Output)	Mutation Score, Accuracy, Precision, Correct Classifications	Student Registration
[121]	2016	Supervised	Backpropagation NN + Cascade	System Executions	Classification (Output)	Accuracy	Credit Analysis
[122]	2002	Supervised	Not Specified	System Executions	Classification (Output)	Mutation Score	Credit Analysis
[123]	2014	Supervised	Backpropagation NN, Decision Tree	System Executions	Classification (Output)	Mutation Score	Triangle Classification
[124]	2006	Supervised	Backpropagation NN	System Executions	Regression (Output)	Precision	Mathematical Functions
[125]	2006	Supervised	Multilayer Perceptron	System Executions	Regression (Output)	Mutation Score	Mathematical Functions
[126]	2022	Supervised	Transformer	Source/Test Code	Regression (Assertions)	Accuracy	Misc. Open-Source Projects
[127]	2019	Supervised	Probabilistic NN	System Executions	Classification (Output)	Correct Classifications	Prime, Triangle Class

Abbreviation: NN, neural network.

TABLE 11 Publications under **Metamorphic (And Other Properties) Test Oracle Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate.

Ref	Year	ML approach	Technique	Training data	ML objective	Evaluation metric	Evaluated on
[128]	2020	Reinforcement	Not Specified	N/A	Reward (Relations)	Not Evaluated	Ocean Modelling
[129]	2021	Reinforcement	Not Specified	N/A	Reward (Relations)	Not Evaluated	Ocean Modelling
[130]	2020	Reinforcement	Contextual Bandit	N/A	Reward (Faults Detected)	Faults Detected	Object Detection
[54]	2018	Supervised	Support Vector Machine	Code Features	Classification (Property)	Accuracy	Misc. Functions
[131]	2013	Supervised	Support Vector Machine, Decision Trees	Code Features	Classification (Property)	Mutation Score	Misc. Functions
[132]	2016	Supervised	Support Vector Machine	Code Features	Classification (Property)	Mutation Score	Misc. Functions
[133]	2021	Supervised	Decision Trees	System Executions	Regression (Conditions)	Accuracy	Android Apps
[134]	2019	Supervised	Support Vector Machine	Code Features	Classification (Property)	ROC	Matrix Calculation
[135]	2007	Supervised	L*	System Executions	Classification (Violation)	Training Data Size	Handshake Protocols
[136]	2017	Supervised	Radial-Basis Function NN	Code Features	Classification (Property)	Accuracy	Misc. Functions

Abbreviation: NN, neural network.

TABLE 12 ML goals and the number of publications pursuing each goal.

Type of goal	Goal	# Pubs.	Publications
Generate input	Maximize Coverage	32	[20,25,28,50,57–61,72,73,80] [4,7,19,62–66,93–95] [30,31,44,67–70,74,75]
	Expose Performance Bottlenecks	8	[22,85,87–92]
	Show Conformance to (or Violation of) Specifications	7	[24,32,33,38,40,43,63]
	Generate Complex Inputs	4	[13,29,37,47]
	Improve Input or Output Diversity	4	[9,34,35,96]
	Predict Failing Input	2	[36,39]
Generate oracle	Predict Output	20	[19,109–119] [120–127]
	Predict Test Verdict	12	[97–108]
	Predict Properties of Output	10	[54,128–136]
Enhance existing method	Improve Effectiveness	15	[5,23,26,27,42,49,76,78,79,81,82,86] [21,45,46]
	Improve Efficiency	10	[12,41,48,51–53,71,77,83,84]

of some criterion associated with effective testing. This includes coverage of code, states or transitions of models, or input interactions. Other uses of ML include generating input that exposes performance bottlenecks, demonstrates conformance to—or violation of—specifications, or increases input/output diversity. Others generate input for a complex data type or input likely to fail.

In the final category, ML tunes the performance or effectiveness of a generation framework—often search-based of Symbolic Execution-based approaches. To improve efficiency, ML clusters redundant tests, replaces expensive calculations with predictions, chooses generation targets, or checks input validity. To improve effectiveness, ML manipulates test cases (e.g., replaces method calls) or tunes the generation strategy (e.g., selects fitness functions, mutation heuristics, or timeouts).

RQ2 (Goal of ML): ML generates input (47%)—particularly to maximize some form of coverage—or oracles (33%)—particularly that predict an expected output. It is also used to improve efficiency or effectiveness of existing generation methods (20%).

4.4 | RQ3: Integration into test generation

RQ3 highlights where and how ML has been integrated into the testing process. This includes types of ML applied, training data, and how ML was used (regression, classification, reward functions).

Supervised techniques were the first applied to input and oracle generation, and remain the most common. Supervised techniques are—by far—the most common for oracle generation. They are also the most common for system and combinatorial interaction testing. The predictions made by models are either from pre-determined options (classification) or open (regression). Classification is often used in oracle generation, for example, to produce a verdict (pass/fail) or output from a limited range. Regression is common in input generation, where complex predictions must be made.

Both training time and quantity of training data need to be accounted for when considering a supervised technique. After being trained, a model will not learn from new interactions, unlike with reinforcement learning. A model must be retrained with new training data to improve its accuracy. Therefore, it is important that supervised methods be supplied with sufficient quantity and quality of training data. Supervised techniques generally learn from past system executions, labelled with a measurement of interest. If the label can be automatically recorded, then gathering sufficient data is often not a major concern. However, if the SUT is computationally inefficient or information is not easily collectible (e.g., a human must label data), it can be difficult to use supervised ML.

Adversarial learning may help overcome data challenges. This strategy forces models to compete, creating a feedback loop where performance is improved without the need for human input. Multiple publications adopted adversarial networks, generally in cases where input was associated with a numeric quality (performance, vehicle speed—for example, [32,33]). Neither case requires human labelling, so models can be automatically retrained. Other recent deep learning approaches—often trained on many systems—show promise in their ability to adapt to unseen systems (e.g., previous works [46,103]).

Reinforcement learning is the second most common type of ML. Reinforcement learning was even used more often than supervised in 2020, and almost as often in 2021. Reinforcement learning has been used in all input generation problems and is the most common technique for GUI, unit, and performance generation.

Reinforcement learning is appealing because it does not require pre-training and automatically improves accuracy through interactions. Reinforcement learning is most applicable when effectiveness can be judged using a numeric metric, that is, where a measurable assessment already exists. This includes performance measurements—for example, resource usage—or code coverage. Reinforcement learning is also effective when the SUT has branching or stateful behaviour—for example, in GUI testing, where a *sequence* of input may be required. Similarly, performance bottlenecks often emerge as the consequence of a sequence of actions, and code coverage may require multiple setup steps. Reinforcement learning is effective in such situations because it can learn from the outcome of taking an action. Therefore, it is effective at constructing sequences of input steps that ultimately achieve some goal of interest. Many supervised approaches are not equipped to learn from each individual action, and must attempt to predict the full sequence of steps at once.

Outside of individual tests, reinforcement learning is also effective at enhancing test generation algorithms. Genetic Algorithms, for example, evolve test suites over a series of subsequent generations. Reinforcement learning can tune aspects of this evolution, in some cases guided by feedback from the same fitness functions targeted by the optimization. If a test suite attains high fitness, reinforcement learning may be able to improve that score by manipulating the test cases of the algorithm parameters. Reinforcement learning can, of course, generate input effectively in a similar manner to an optimization algorithm. However, it also can often improve the algorithm such that it produces even better tests.

Authors of sampled publications applied unsupervised learning to cluster test cases to improve generation efficiency or to identify weakly tested areas of the SUT. While clustering has not been used often in the sampled publications, clustering is common in other testing practices (e.g., to identify tests to execute [3]). Therefore, it may have potential for use in filtering tasks during generation, especially to improve efficiency. Future work should further consider how clustering could be applied as part of test generation.

RQ3 (Integration of ML): The most common ML types are supervised (61%) and reinforcement learning (34%). Some publications also employ unsupervised (2%) or semi-supervised (3%) learning. (Semi-)Supervised learning is the most common ML for system testing, CIT, and all forms of oracle. Reinforcement learning is the most common technique for GUI, unit, and performance testing, and is used where testing goals often have measurable scores, a sequence of input is required, or existing generation tools can be tuned. Clustering was also used for filtering, for example, discarding similar test cases.

4.5 | RQ4: ML techniques applied

RQ4 examines specific ML techniques. Table 13 lists techniques employed, divided by ML type. Neural networks are the most common techniques in supervised learning. Support vector machines are also employed often, as are forms of decision trees.

In particular, backpropagation neural networks are used most (11%). Backpropagation neural networks are a classic technique where a network is composed of multiple layers [137]. In each layer, a weight value for each node is calculated. In such networks, information is fed *forward*—there are no cyclic connections to earlier layers. However, the

TABLE 13 ML techniques adopted—divided by ML type and family of ML techniques—ordered by number of publications where the technique is adopted.

Type	Family	Technique	# Pubs.
Supervised	Neural networks	Backpropagation NN	14
		Multi-Layer Perceptron	8
		Artificial NN	7
		Long Short-Term Memory NN	6
		Transformer	4
		Radial-Basis Function NN	3
		Convolutional NN, Deep NN, Feedforward NN, Recurrent NN, Residual NN	2
		Backpropagation NN + Cascade, Probabilistic NN, Shallow NN, UNet	1
	Trees	Decision Tree	5
		Random Forest	3
		Gradient Boosting, Regression Tree	2
		Ada-Boosted Tree, C4.5, J48, Tree-LSTM	1
	Others	Support Vector Machine	11
		L*, Conditional Random Fields, Ensemble, K-Nearest Neighbours, Regression Gaussian Process, Stepwise Regression	2
		Adaptive Boosting, Gaussian Process, Multivariate Time Series, Naive Bayes, Parallel Distributed Processing, Query Strategy Framework, RIPPER	1
Reinforcement	Q-learning	Q-Learning	16
		Deep Q-Network	3
		Double Q-Learning	2
		Delayed Q-Learning, Duelling Deep Q-Network, Double Deep Q-Network, Q-Learning + Fuzzy Logic, Q-Learning + Long Short-Term Memory, ReLU Q-Learning	1
	Others	Differential Semi-Gradient Sarsa, Upper Confidence Bound	3
		Sarsa	2
		Advantage Actor-Critic, Asynchronous Advantage Actor Critic, Contextual Bandit, Markov Decision Process, Monte Carlo Control, Monte Carlo Tree Search, SOFTMAX	1
Semi-supervised		Generative Adversarial Network	3
		Convolutional NN, Conditional Generative Adversarial Network	1
Unsupervised		Backpropagation NN, Expectation-Maximization, MeanShift	1

Abbreviation: NN, neural network.

backpropagation feature propagates error backward, allowing earlier nodes to adjust weights if necessary. This leads to less complexity and faster learning rates. In recent years, more complex neural networks have continued to implement backpropagation as one (of many) features.

Recently, neural networks utilizing Long Short-Term Memory have also become quite common. Unlike traditional feedforward neural networks, Long Short-Term Memory has feedback connections [138]. This creates loops in the network, allowing information to persist. This adaptation allows such networks to process not just single data points, but sequences where one data point depends on earlier points. Long Short-Term Memory networks and deep neural networks are likely to become more common in the next few years as more researchers adopt deep learning techniques.

The emergence of transformer models—complex neural networks that learn from, and generate, natural language [111]—is promising for both test and oracle generation. Transformers make use of a mechanism called 'self-attention' that uses backpropagation to infer the relationship between words in a phrase [139]. This mechanism enables automated context-extraction and summarization of text, which in term enables the model to produce complex textual output as well.

Reinforcement learning is dominated by forms of Q-Learning—Q-Learning and its variants are used in 22% of publications. Q-Learning is a prototypical form of off-policy reinforcement learning, meaning that it can choose either to take an action guided by the current 'best' policy—maximizing expected reward—or it can choose to take a random action to refine the policy [140]. Many other reinforcement learning techniques are also off-policy, and follow a similar process, with various differences (e.g., calculating reward or action decisions in a different manner).

Some authors have chosen specific techniques because they worked well in previous work (e.g., previous works [63,132]). Others saw certain techniques work on similar problems outside of test generation (e.g., Jia et al. [23]), or chose techniques thought to represent the state-of-the-art for a problem class (e.g., Kıraç et al. [39]). However, most authors do not justify their choice of technique, nor do they often compare alternatives.

In recent years, open-source ML frameworks have emerged that accelerate the pace and effectiveness of research by making robust algorithms available. The authors of 51 publications (41% of the sample) explicitly made use of existing frameworks. The most common ML frameworks used in the sampled publications include keras-rl (e.g., Kim et al. [4]), Matlab (e.g., Arrieta et al. [110]), OpenAI Gym (e.g., Huurman et al. [28]), PyTorch (e.g., Spieker et al. [130]), scikit-learn (e.g., Hardin et al. [54]), TensorFlow (e.g., Budnik et al. [35]), and WEKA (e.g., Luo et al. [88]). In the other 73 publications—especially older publications—authors either implemented ML algorithms or adapted unspecified implementations. The use of a framework constrains technique choice. However, all of these frameworks offer many techniques, and may allow researchers to compare results across techniques. This could lead to more informed and robust implementations.

RQ4 (ML Techniques): Neural networks, especially backpropagation neural networks, are the most common supervised techniques. Reinforcement learning is generally based on Q-Learning. Technique choice is often not explained, but may be inspired by insights from previous or related work, an algorithm having performed well on a similar problem, or algorithms available in open-source frameworks (e.g., OpenAI Gym or WEKA).

4.6 | RQ5: Evaluation of the test generation framework

RQ5 examines how authors have evaluated their work—in particular, how ML affects evaluation. The metrics adopted by the authors are listed in Table 14. We group similar metrics (e.g., coverage metrics, notions of fault detection, etc.). In most cases, these metrics are used to evaluate the quality of the input or oracle generation approach.

In most cases, the entire framework is evaluated. Almost all of these evaluations employ standard metrics for test generation. Some metrics are specific to a testing practice (e.g., covering array size) or aspect of generation (e.g., number of queries solved), while others are applied across testing practices (e.g., fault detection). Naturally—whether ML is incorporated or not—a generation framework must be evaluated on its effectiveness.

Many authors also evaluate the ML component separately. Supervised approaches were often evaluated using some notion of model accuracy—using various accuracy measurements, correct classification rate, and ROC. Approaches have also been evaluated on the quantity of required training data, whether a model can be applied to unknown systems, and the sensitivity of model predictions to small changes in the input or model parameters. In addition, one study used the size of the trained model to help explain the results of applying the technique, rather than using it to measure

TABLE 14 Evaluation metrics adopted (similar metrics are grouped), divided by ML approach, and ordered by number of publications using each metric. Metrics in bold are related to ML.

Type	Metric	# Pubs.
Supervised	**Prediction Accuracy (e.g., correct classifications, ROC)**	37
	Faults Detected (including mutants and performance issues)	33
	Efficiency (e.g., scalability, # tests generated/executed, time), Coverage Attained (e.g., code, state)	12
	Test Size (e.g., size of test cases, suite, or covering array)	4
	Adaptivity (whether a model can be transferred to a new system), Validity of Generated Inputs, **Quantity of Training Data Required**	3
	Flakiness of Generated Tests	2
	Input/Output Diversity, **Model Size**, **Sensitivity of Predictions**	1
Reinforcement	Coverage	25
	Faults Detected	13
	Efficiency	6
	Input/Output Diversity	4
	# Exceptions Discovered	2
	Adaptivity, Qualitative Analysis, # Queries Solved, # Requirements Met, **Sensitivity**, Test Size	1
Semi-supervised	Faults Detected	4
	Prediction Accuracy, Coverage, Efficiency, Quality of Generated Inputs, Validity of Generated Inputs, **Required Labelling and Training Effort**, **Sensitivity of Predictions**	1
Unsupervised	**# Clusters Produced**, Qualitative Analysis	1

solution quality. Semi-supervised approaches were also evaluated using accuracy, the required labelling/training effort, and sensitivity. Finally, one study employing an unsupervised approach used the number of clusters produced to analyse the results of applying their approach.

Reinforcement learning approaches were generally not evaluated using ML-specific metrics, except for a study that examined their adaptivity and sensitivity. This is reasonable, as reinforcement learning learns how to maximize a numeric function. The reward is based on the goals of the overall generation framework. Rather than evaluating using an absolute notion of accuracy, the success of reinforcement learning can be seen in improved reward measurements, attainment of a checklist of goals, or metrics such as fault detection.

RQ5 (Evaluation): The full generation framework is generally evaluated by traditional testing metrics (e.g., fault detection). However, the ML components are also evaluated—especially in supervised learning—using accuracy, adaptivity, quantity of training data needed, labelling/training effort, prediction sensitivity, and other ML metrics. Reinforcement learning is generally evaluated using testing metrics tied to the reward.

4.7 | RQ6: Limitations and open challenges

The sampled publications show great potential. However, we have observed multiple challenges that must be overcome to transition research into real-world use.

Volume, contents, and collection of training data: (Semi-)Supervised ML requires training data to create a model. There are multiple challenges related to the *required volume* of training data, the *required contents* of the training data, and *human effort* required to produce that training data.

Regardless of the testing practice addressed, the volume of required training data can be vast. This data is generally attained from labelled execution logs, which means that the SUT needs to be executed *many* times to gather the information needed to train the model. Approaches based on deep learning could produce highly accurate models but may require thousands of executions to gather required training data. Some approaches also must preprocess the collected data. While it may be possible to automatically gather training data, the time required to produce the dataset can still be high and must be considered.

This is particularly true for cases where a regression is performed rather than a classification—for example, an expected value oracle [110] or complex test input [13]. Producing a complex continuous value is more difficult than a simple classification, and requires significant training data—with a range of outcomes—to make accurate predictions.

In addition, the contents of the training data must be considered. If generating input, the training data must contain a wide range of input scenarios with diverse outcomes that reflect the specific problem of interest and its different branching possibilities. Consider code coverage prediction (e.g., previous works [19,84]). If one wishes to predict the input that will cover a particular element, then the training data must contain sufficient information content to describe how to cover that element. That requires a diverse training set.

Models based on output behaviour—for example, expected value oracles or models that predict input based on particular output values [34,35,56]—suffer from a related issue. The training data for expected value oracles must either come from passing test cases—that is, the output must be correct—or labels must be applied by humans. A small number of cases accidentally based on failing output may be acceptable if the algorithm is resilient to noise in the training data, but training on faulty code can result in an inaccurate model. This introduces a significant barrier to automating training by, for example, generating input and simply recording the output that results.

Similarly, models that make predictions based on failures—for example, test verdict oracles or models that produce input predicted to trigger a failure [39] or performance issue [88]—require training data that contains a large number of *failing test cases*. This implies that faults have already been discovered and, presumably, fixed before the model is trained. This introduces a paradox. There may be remaining failures to discover. However, the more training data that is needed, the less the need for—or impact of—the model.

In some cases, training data must be labelled (or even collected) by a human. Again, oracles suffer heavily from this problem. Test verdict oracles require training data where each entry is assigned a verdict. This requires either existing test oracles—reducing the need for a ML-based oracle—or human labelling of test results. Judging test results is time-consuming and can be erroneous as testers become fatigued [1], making it difficult to produce a significant volume of training data. Generation of metamorphic relation oracles requires overcoming a similar dilemma, where training data must be labelled based on whether a particular metamorphic relation holds. This requires labelling by a tester with significant knowledge of the source code.

For some problems, these issues can be avoided by employing reinforcement learning instead. Reinforcement learning will learn while interacting with the SUT. In cases where the effectiveness of ML can be measured automatically—for example, code coverage and performance bottlenecks—reinforcement learning is a viable solution. However, cases where ground truth is required—for example, oracles—are not as amenable to reinforcement learning. Reinforcement learning also requires many executions of the SUT, which can be an issue if the SUT is computationally expensive or otherwise difficult to execute and monitor, such as when specialized hardware is required for execution.

Otherwise, techniques are required that (1) can enhance training data, (2) can extrapolate from limited training data, and (3), can tolerate noise in the training data. Means of generating synthetic training data, like in the work of Nair et al. [134], demonstrate the potential for data augmentation to help overcome this limitation. Adversarial learning also offers a way to improve the accuracy of a model—reducing the need for a large training dataset. Again, however, such approaches are of limited use in cases where human involvement is required. In addition, deep learning approaches—such as transformers—can often be trained on data from many different projects, potentially yielding models that are also effective on projects not in their training set (e.g., Yu et al. [126]).

RQ6 (Challenges): Supervised learning is limited by the required quantity, quality, and contents of training data—especially when human effort is required. Oracles particularly suffer from these issues. Reinforcement learning and adversarial learning are viable alternatives when data collection and labelling can be automated.

Retraining and feedback: After training, models have a fixed error rate and do not learn from new mistakes made. If the training data is insufficient or inaccurate, the generated model will be inaccurate. The ability to improve the model based on additional feedback could help account for limitations in the initial training data.

There are two primary means to overcome this limitation—either retraining the model using an enriched training dataset or adopting a reinforcement learning approach that can adapt its expectations based on feedback. Both means carry challenges. Retraining requires (a) establishing a schedule for when to train the updated model, and (b), an active

effort on the part of human testers to enrich and curate the training dataset. Adversarial learning offers an automated means to retrain the model. However, there are still limitations on when it can be applied.

Enriching the dataset—as well as the use of reinforcement learning—requires some kind of feedback mechanism to judge the effectiveness of the predictions made. This can be difficult in some cases, such as test oracles, where human feedback may be required. Human feedback, even on a subset of the decisions made, reduces the cost savings of automation.

RQ6 (Challenges): Models should be retrained over time. How often retraining occurs depends, partially, on the cost to gather and label additional data or on the amount of human feedback required.

Complexity of studied systems: Regardless of ML type, many of the proposed approaches are evaluated on highly simplistic systems. 44% of the publications evaluate using toy examples, with only a few lines of code or possible function outcomes. While it is intuitive to *start* with simplistic examples to examine the viability of an ML approach, the real-world application requires accurate predictions for complex functions and systems with many branching code paths. If a function is simple, there is likely little need for a predictive model in the first place. Several recent studies feature thorough evaluations of complex systems (e.g., previous studies [48,70,79]), even on industrial systems (e.g., previous studies [23,41]). However, many studies evaluate on only a single example or a handful of examples, and many of those examples are still not very complex. It largely remains to be seen whether many proposed techniques can be used on real-world production code.

The generation of models for arbitrary systems with unconstrained output may be prohibitively difficult even for sophisticated ML techniques. This is particularly the case for expected value oracles. In such cases, some abstraction should be expected—either a simplification of the core logic of the system or a partition of inputs or outputs into symbolic values. One possibility to consider is a variable level of abstraction—for example, a training-time decision to cluster output predictions into an adjustable number of representative values (such as the centroids of clusters of outputs). Training could take place over different settings for this parameter, and the balance between accuracy and abstraction could be explored.

In any evaluation, a variety of systems should be considered. The complexity of the systems should vary. This enables the assessment of scalability of the proposed techniques. Researchers should examine how prediction accuracy, training data requirements (for supervised learning), and time to convergence on an optimal policy (for reinforcement learning) scale as the complexity of the system increases. This would enable a better understanding of the limitations and applicability of ML-based techniques in test generation for real-world systems.

RQ6 (Challenges): Scalability of ML techniques to real-world systems is not clear. When modelling complex functions, varying degrees of abstraction could be explored if techniques are unable to scale. In evaluations, a range of systems should be considered, and explicit analyses of scalability (e.g., accuracy, training, learning rate) should be performed.

Variety, complexity, and tuning of ML techniques: Authors rarely explain or justify their choice of ML algorithm—often stating that an algorithm worked well previously or that it is 'state-of-the-art', if any rationale is offered. It is even rarer that multiple algorithms are compared to determine which is best for a particular task. As the purpose of many research studies is to demonstrate the viability of an idea, the choice of algorithm is not always critically important. However, this choice still has implications, as it may give a false impression of the applicability of an approach and unnecessarily introduce a performance ceiling that could be overcome through the consideration of alternative techniques.

One reason for this limitation may be that testing researchers are generally ML *users*, not ML experts. They may lack the expertise to know which algorithms to apply. Collaboration with ML researchers may help overcome this challenge. The use of open-source ML frameworks can also ease this challenge by removing the need for researchers to

develop their own algorithms. Rather than needing to understand each algorithm, they could instead compare the performance of available alternatives. This comparison would also lead to a richer evaluation and discussion.

Many of the proposed approaches—especially earlier ones—are based on simple neural networks with few layers. These techniques have strict limitations in the complexity of the data they can model and have been replaced by more sophisticated techniques. Deep learning, which may utilize many hidden layers, may be essential in making accurate predictions for complex systems. Few approaches to date have utilized deep learning, but such approaches are starting to appear, and we would expect more to explore these techniques in the coming years. However, deep learning also introduces steep requirements on the training data that may limit its applicability.

Almost all of the proposed approaches utilize a single ML technique. An approach explored in many domains is an *ensemble* [36]. In such approaches, models are trained on the same data using a variety of techniques. Each model is asked for a prediction, and then the final prediction is based on the consensus of the ensemble. Ensembles are often able to reach stable, accurate conclusions in situations where a single model may be inaccurate. A small number of studies have applied ensembles [36,83,104,110,113], but such techniques are rare.

Many ML techniques have parameters that can be tuned (e.g., learning rate, number of hidden units, or activation function). Parameter tuning can significantly impact prediction accuracy and enable significant improvements in the results of even simple ML techniques. The sampled publications do not explore the impact of such tuning. This is an oversight that should be corrected in future work.

RQ6 (Challenges): Researchers rarely justify the choice of ML technique or compare alternatives. The use of open-source ML frameworks can ease comparison. Deep learning and ensemble techniques, as well as hyperparameter tuning, should also be explored more widely.

Lack of standard benchmarks: Research benchmarks have enabled sophisticated analyses and comparison of approaches for automated test generation. Such benchmarks usually contain a set of systems prepared for a particular type of evaluation. Bug benchmarks, in particular, contain real faults curated from a variety of systems, along with metadata on those faults. Such benchmarks ease comparison with past research, remove bias from system selection and demonstrate the effectiveness of techniques. Only a small subset of the sampled publications make use of existing research benchmarks. The most common, by far, is the F-Droid Android benchmark (e.g., previous studies [60,61]). Others made use of examples commonly used in research such as the Defects4J (e.g., previous studies [79,111]) or the RUBiS web app example (e.g., Sedaghatbaf et al. [89]). However, the majority of studies do not use benchmarks or open-source evaluation targets.

Some studies require their own particular evaluation. However, in cases where evaluation is over-simplistic, or where code or metadata is unavailable, this makes comparison and replication difficult. Benchmarks are typically tied to particular system types or testing practices. In cases where benchmarks exist—unit, web app, mobile app, and performance testing in particular—we would encourage researchers to use these benchmarks to enable comparison to past work or to allow researchers to make comparisons with their work.

In other cases, the creation of benchmarks specifically for ML-enhanced test generation research could advance the state-of-the-art in the field, spur new research advances, and enable replication and extension of proposed approaches. In particular, we recommend the creation of such a benchmark for oracle generation. Such a benchmark should contain a variety of code examples from multiple domains and of varying levels of complexity. Code examples should be paired with the metadata needed to support oracle generation. This would include sample test cases and human-created test oracles, at minimum. Such a benchmark could also include sample training data that could be augmented over time by researchers.

Lack of replication package or open code: A common dilemma is lack of access to research code and data. Often, a publication is not sufficient to allow replication or application in a new context. This applies to research in ML-enhanced test generation as well, as only 33% of the publications in our sample provided open-source code or replication packages.

Outside of this 33%, some publications made use of open-source ML frameworks. This is positive, in that the specific ML techniques are trustworthy and available. Potentially, experimental results could be replicated in such cases by applying the same techniques to the same settings. However, there still may not be enough information in the paper to enable replication, such as specific parameter settings. Further, these frameworks evolve over time, and the results may differ because the underlying ML technique has changed since the original study was published.

Researchers should include a replication package with their source code, execution scripts, and the versions of external dependencies used when the study was performed. This package should also include training data and the gathered experiment observations used by the authors in their analyses.

RQ6 (Challenges): Research is limited by the overuse of simplistic examples, the lack of common benchmarks, and the unavailability of code and data. Researchers should be encouraged to use available benchmarks, and provide replication packages and open code. New benchmarks could be created for ML challenges (e.g., oracle generation).

5 | THREATS TO VALIDITY

External and internal validity: Our conclusions are based on the publications sampled. It is possible that we may have omitted important publications. This can affect internal validity—the evidence we use to make conclusions—and external validity—the generalizability of our findings. Secondary studies can be valuable even if they do not capture all publications from a research field as long as their selection protocol (search string, inclusion/exclusion criteria, snowballing) ensures an adequate sample to infer similar findings to a complete set of relevant publications. We believe that our selection strategy was appropriate. We tested different search strings and performed a validation exercise to test the robustness of our string. We have used four databases, covering the majority of relevant venues, and performed additional snowballing. Our final set of publications includes 124 primary publications, which we believe is sufficient to make informed conclusions.

Conclusion validity: Subjective judgements are part of article selection, data extraction, and categorizing publications. To control for bias, protocols were discussed and agreed upon by both authors, and independent verification took place on—at least—a sample of all decisions made by either author.

Construct validity: We used a set of properties to guide data extraction. These properties may have been incomplete or misleading. However, we have tried to establish properties that were informed by our research questions. These properties were iteratively refined, and we believe they have allowed us to thoroughly answer the questions.

6 | CONCLUSIONS

Automated test generation is a well-studied research topic, but there are critical limitations to overcome. Recently, researchers have begun to use ML to enhance automated test generation. We have characterized emerging research on this topic through a systematic mapping study examining testing practices that have been addressed, the goals of using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how the full test generation process is evaluated, and open research challenges.

We observed that ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-)Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function.

The work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. Our findings can serve as a roadmap for both researchers and practitioners interested in the use of ML as part of test generation.

ACKNOWLEDGEMENTS

This research was supported by Vetenskapsrådet grant 2019-05275.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analysed in this study.

ORCID

Gregory Gay https://orcid.org/0000-0001-6794-9585

REFERENCES

1. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: A survey. IEEE Trans Softw Eng. 2014;41(5): 507–25.
2. Orso A, Rothermel G. Software testing: A research travelogue (2000–2014). In *Proceedings of the Future of Software Engineering, ACM, New York, NY, USA, FOSE*, 2014;117–32.
3. Durelli VH, Durelli RS, Borges SS, Endo AT, Eler MM, Dias DR, Guimaraes MP. Machine learning applied to software testing: A systematic mapping study. IEEE Trans Reliab. 2019;68(3):1189–212.
4. Kim J, Kwon M, Yoo S. Generating test input with deep reinforcement learning. In *International Workshop on Search-Based Software Testing, Association for Computing Machinery, New York, NY, USA, SBST, 51–58*, 2018.
5. Almulla H, Gay G. Learning how to search: Generating exception-triggering tests through adaptive fitness function selection. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020;63–73.
6. Fontes A, Gay G. Using machine learning to generate test oracles: A systematic literature review. In *International Workshop on Test Oracles, Association for Computing Machinery*: New York, NY, USA, TORACLE, 2021;1–0.
7. Li Y, Yang Z, Guo Y, Chen X. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In IEEE/ACM International Conference on Automated Software Engineering IEEE Press, ASE; 1070–3.
8. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, et al. An orchestrated survey of methodologies for automated software test case generation. J Syst Softw. 2013;86(8):1978–2001.
9. Walkinshaw N, Fraser G. Uncertainty-driven black-box test data generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017;253–63.
10. Salahirad A, Almulla H, Gay G. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. Softw Test Verification Reliab. 2019;29(4-5): e1701. https://doi.org/10.1002/stvr.1701,_e1701stvr.1701
11. Rojas JM, Campos J, Vivanti M, Fraser G, Arcuri A. 2015. Combining multiple coverage criteria in search-based unit test generation. In Search-Based Software Engineering, M. Barros, Y. Labiche (eds). Springer International Publishing; 93–108. https://doi.org/10.1007/978-3-319-22183-0_7
12. Luo S, Xu H, Bi Y, Wang X, Zhou Y. Boosting symbolic execution via constraint solving time prediction (experience paper). In *ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA, 336–347*, 2021.
13. Shrestha SL. Automatic generation of simulink models to find bugs in a cyber-physical system tool chain using deep learning. In *ACM/IEEE International Conference on Software Engineering: Companion Proceedings, Association for Computing Machinery, New York, NY, USA, ICSE, 110–112*, 2020.
14. Jha N, Popli R. Artificial intelligence for software testing-perspectives and practices. In *International Conference on Computational Intelligence and Communication Technologies (CCICT)*, 2021;377–82.
15. Ioannides C, Eder KI. Coverage-directed test generation automated by machine learning – a review. ACM Trans Des Autom Electron Syst. 2012; 17(1). https://doi.org/10.1145/2071356.2071363
16. Balera JM, de Santiago Júnior VA. A systematic mapping addressing hyper-heuristics within search-based software testing. Inf Softw Technol. 2019;114:176–89.
17. Petersen K, Vakkalanka S, Kuzniarz L. Guidelines for conducting systematic mapping studies in software engineering: An update. Inf Softw Technol. 2015;64:1–8.
18. Pezze M, Young M. Software Test and Analysis: Process, Principles, and Techniques. John Wiley and Sons, 2006.
19. Majma N, Babamir SM. Software test case generation test oracle design using neural network. In *Iranian Conference on Electrical Engineering (ICEE)*, 2014;1168–73.
20. Ariyurek S, Betin-Can A, Surer E. Automated video game testing using synthetic and humanlike agents. IEEE Trans Games. 2021;13(1):50–67. https://doi.org/10.1109/TG.2019.2947597
21. Zheng Y, Xie X, Su T, Ma L, Hao J, Meng Z, et al. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019;772–84.
22. Ahmad TD, Truscan, Porres I. Exploratory performance testing using reinforcement learning. In *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019;156–63.
23. Jia Y, Cohen MB, Harman M, Petke J. 2015. Learning combinatorial interaction test generation strategies using hyperheuristic search. In International Conference on Software Engineering IEEE Press, ICSE; 540–50.
24. Baumann D, Pfeffer R, Sax E. Automatic generation of critical test cases for the development of highly automated driving functions. In *IEEE Vehicular Technology Conference (VTC)*, 2021;1–5.
25. Araiza-Illan D, Pipe AG, Eder K. Intelligent agent-based stimulation for testing robotic software in human-robot interactions. In *Workshop on Model-Driven Robot Software Engineering, Association for Computing Machinery, New York, NY, USA, MORSE*, 2016;9–16.
26. Buzdalov M, Buzdalova A. Adaptive selection of helper-objectives for test case generation. In *2013 IEEE Congress on Evolutionary Computation*, 2013;2245–50.
27. Esnaashari M, Damia AH. Automation of software test data generation using genetic algorithm and reinforcement learning. Expert Syst Appl. 2021;183(115446). https://doi.org/10.1016/j.eswa.2021.115446
28. Huurman S, Bai X, Hirtz T. Generating api test data using deep reinforcement learning. In *IEEE/ACM International Conference on Software Engineering Workshops, Association for Computing Machinery, New York, NY, USA, ICSEW*, 2020;541–4.
29. Reddy S, Lemieux C, Padhye R, Sen K. Quickly generating diverse valid test inputs with reinforcement learning. In *ACM/IEEE International Conference on Software Engineering, Association for Computing Machinery*: New York, NY, USA, ICSE,2020;1410–21.
30. Shu T, Wu C, Ding Z. Boosting input data sequences generation for testing efsm-specified systems using deep reinforcement learning. Inform Softw Technol. 2023;155:107114. https://doi.org/10.1016/j.infsof.2022.107114. https://www.sciencedirect.com/science/article/pii/S0950584922002233

31. Veanes M, Roy P, Campbell C. 2006. Online testing with reinforcement learning. In Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification Springer-Verlag: Berlin, Heidelberg, FATES'06/RV'06; 240–53. https://doi.org/10.1007/11940197_16
32. Deng Y, Lou G, Zheng X, Zhang T, Kim M, Liu H, et al. Bmt: Behavior driven development-based metamorphic testing for autonomous driving models. In *International Workshop on Metamorphic Testing (MET)*, 2021;32–6.
33. Zhang M, Zhang Y, Zhang L, Liu C, Khurshid S. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018;132–42.
34. Bergadano F. Test case generation by means of learning techniques. SIGSOFT Softw Eng Notes. 1993;18(5):149–62. https://doi.org/10.1145/167049.167074
35. Budnik C, Gario M, Markov G, Wang Z. Guided test case generation through ai enabled output space exploration, 2018;53–6.
36. Eidenbenz R, Franke C, Sivanthi T, Schoenborn S. Boosting exploratory testing of industrial automation systems with ai, 2021;362–71.
37. Gao Z, Chang R, Ai C. The stacked seq2seq-attention model for protocol fuzzing. In *IEEE International Conference on Computer Science and Network Technology (ICCSNT), 126–130*, 2019.
38. Kikuma K, Yamada T, Sato K, Ueda K. Preparation method in automated test case generation using machine learning. In *International Symposium on Information and Communication Technology, Association for Computing Machinery, New York, NY, USA, SoICT, 393–398*, 2019.
39. Kıraç M, Aktemur B, Sözer H, Gebizli C. Automatically learning usage behavior and generating event sequences for black-box testing of reactive systems. Softw Qual J. 2019;27(2):861–83. https://doi.org/10.1007/s11219-018-9439-1
40. Meinke K, Khosrowjerdi H. Use case testing: A constrained active machine learning approach. Lect Notes Comput Sci. 2021; 12740 LNCS: 3–21. https://doi.org/10.1007/978-3-030-79379-1_1
41. Mirabella AG, Martin-Lopez A, Segura S, Valencia-Cabrera L, Ruiz-Cortés A. Deep learning-based prediction of test input validity for restful apis. In *International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, 2021;9–16.
42. Sharma A, Melnikov V, Hüllermeier E, Wehrheim H. Property-driven testing of black-box functions. Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering. In *Association for Computing Machinery, New York, NY, USA, FormaliSE '22*, 2022;113–23. https://doi.org/10.1145/3524482.3527657
43. Ueda K, Tsukada H. Accuracy improvement by training data selection in automatic test cases generation method. 2021;438–42.
44. Utting M, Dadeau F, Tamagnan F, Bouquet F. Identifying and generating missing tests using machine learning on execution traces. In *IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2020;83–90.
45. Zhao R, Lv S. Neural-network based test cases generation using genetic algorithm. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2007;97–100.
46. Zhong Z, Kaiser G, Ray B. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. IEEE Trans Softw Eng. 2022:1–5. https://doi.org/10.1109/TSE.2022.3195640
47. Zhu J, Wang L, Gu Y, Lin X. Learning to restrict test range for compiler test. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019;272–4.
48. Chen Z, Chen Z, Shuai Z, Zhang G, Pan W, Zhang Y, Wang J. Synthesize solving strategy for symbolic execution. 2021;348–60.
49. Paduraru C, Paduraru M, Stefanescu A. Riverfuzzrl - an open-source tool to experiment with reinforcement learning for fuzzing, 2021. 430–435.
50. Feldmeier P, Fraser G. Neuroevolution-based generation of tests and oracles for games. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery*: New York, NY, USA, ASE '22, 2023. https://doi.org/10.1145/3551349.3556939
51. Liu Z, Yang X, Zhang S, Liu Y, Zhao Y, Zheng W. Automatic generation of test cases based on genetic algorithm and rbf neural network. Mob Inf Syst. 2022:1489063. https://doi.org/10.1155/2022/1489063
52. Mishra K, Tiwari S, Misra A. Combining non revisiting genetic algorithm and neural network to generate test cases for white box testing. Adv Intell Soft Comput. 2011;124:373–80. https://doi.org/10.1007/978-3-642-25658-5_46
53. Pan S, Zhang H, Zuo X, Deng H. 2022. Method of generating program path test cases based on neural network. In International Conference on Optoelectronic Information and Computer Engineering (OICE 2022), Y. Yang (ed.). International Society for Optics and Photonics, SPIE; 123080D. https://doi.org/10.1117/12.2647709
54. Hardin B, Kanewala U. Using semi-supervised learning for predicting metamorphic relations. In *International Workshop on Metamorphic Testing, Association for Computing Machinery, New York, NY, USA, MET, 14–17*, 2018.
55. Gay G, Staats M, Whalen M, Heimdahl M. Automated oracle data selection support. Softw Eng IEEE Trans. 2015;41(11):1119–37. https://doi.org/10.1109/TSE.2015.2436920
56. Papadopoulos P, Walkinshaw N. Black-box test generation from inferred models. In *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, IEEE Press, RAISE, 19–24*, 2015.
57. Adamo D, Khan MK, Koppula S, Bryce R. 2018. Reinforcement learning for android gui testing. In ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation Association for Computing Machinery: New York, NY, USA, A-TEST; 2–8.
58. Brunetto M, Denaro G, Mariani L, Pezzé M. On introducing automatic test case generation in practice: A success story and lessons learned. J Syst Softw. 2021; 176(110933). https://doi.org/10.1016/j.jss.2021.110933
59. Choi W, Necula G, Sen K. Guided gui testing of android apps with minimal restart and approximate learning. ACM SIGPLAN Not. 2013; 48(10):623–39. https://doi.org/10.1145/2544173.2509552
60. Collins E, Neto A, Vincenzi A, Maldonado J. Deep reinforcement learning based android application gui testing. In *Brazilian Symposium on Software Engineering, Association for Computing Machinery*: New York, NY, USA, SBES, 2021;186–94.
61. Degott C, Borges JNP, Zeller A. Learning user interface element interactions. In *ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery*: New York, NY, USA, ISSTA, 2019;296–306.
62. Khan MK, Bryce R. 2022. Android gui test generation with sarsa. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)* 487–93.
63. Koroglu Y, Sen A. Functional test generation from ui test scenarios using reinforcement learning for android applications. Softw Testing Verification Reliab. 2021;31(3):e1752. https://doi.org/10.1002/stvr.1752
64. Koroglu Y, Muslu O, Mete Y, Ulker C, Tanriverdi T, Donmez Y. Qbe: Qlearning-based exploration of android applications. In *IEEE International Conference on Software Testing. Verification and Validation (ICST), 105–115*, 2018.

65. Mariani L, Riganelli O, Santoro M. Autoblacktest: Automatic black-box testing of interactive applications. In *IEEE International Conference on Software Testing, Verification and Validation, 81–90*, 2012.
66. Pan M, Huang A, Wang G, Zhang T, Li X. Reinforcement learning based curiosity-driven testing of android applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA*, 2020;153–64.
67. Sherin S, Muqeet A, Khan MU, Iqbal MZ. Qexplore: An exploration strategy for dynamic web applications using guided search. J Syst Softw. 2023;195:111512. https://doi.org/10.1016/j.jss.2022.111512. https://www.sciencedirect.com/science/article/pii/S0164121222001881
68. Vuong TAT, Takada S. A reinforcement learning based approach to automated testing of android applications. In *ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Association for Computing Machinery, New York, NY, USA, A-TEST*, 2018;31–7.
69. Yasin H, Hamid S, Yusof R. Droidbotx: Test case generation tool for android applications using q-learning. Symmetry. 2021;13(2):1–30. https://doi.org/10.3390/sym13020310
70. Zheng Y, Liu Y, Xie X, Liu Y, Ma L, Hao J, Liu Y. Automatic web testing using curiosity-driven reinforcement learning, 2021;423–35.
71. Kamal MM, Darwish SM, Elfatatry A. Enhancing the automation of gui testing. In *International Conference on Software and Information Engineering, Association for Computing Machinery*: New York, NY, USA, ICSIE, 2019;66–70.
72. Khaliq Z, Farooq SU, Khan DA. A deep learning-based automated framework for functional user interface testing. Inf Softw Technol. 2022; 150:106969. https://doi.org/10.1016/j.infsof.2022.106969. https://www.sciencedirect.com/science/article/pii/S0950584922001070
73. Khaliq Z, Khan DA, Farooq SU. Using deep learning for selenium web ui functional tests: A case-study with e-commerce applications. Eng Appl Artif Intell. 2023;117:105446. https://doi.org/10.1016/j.engappai.2022.105446. https://www.sciencedirect.com/science/article/pii/S0952197622004365
74. Santiago D, Clarke PJ, Alt P, King TM. Abstract flow learning for web application test generation. In *ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, Association for Computing Machinery, New York, NY, USA, A-TEST, 49–55*, 2018.
75. Santiago D, Phillips J, Alt P, Muras B, King T, Clarke P. Machine learning and constraint solving for automated form testing, 2019. volume 2019-October, 217–227.
76. Yazdani F, Daragh B, Malek S. Deep gui: Black-box gui input generation with deep learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021;905–16.
77. Hooda I, Chhillar R. Test case optimization and redundancy reduction using ga and neural networks. Int J Electr Comput Eng. 2018;8(6):5449–56. https://doi.org/10.11591/ijece.v8i6.pp5449-5456
78. Almulla H, Gay G. Generating diverse test suites for gson through adaptive fitness function selection. Lect Notes Comput Sci. 2020;12420 LNCS: 246–52. https://doi.org/10.1007/978-3-030-59762-7_18
79. Almulla H, Gay G. Learning how to search: generating effective test cases through adaptive fitness function selection. Empir Softw Eng. 2022; 27(2):38. https://doi.org/10.1007/s10664-021-10048-8
80. Groce A. Coverage rewarded: Test input generation via adaptation-based programming. In *IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, USA, ASE, 380–383*, 2011.
81. He W, Zhu Q. Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software. Chin J Electron. 2015;24(1):38–45. https://doi.org/10.1049/cje.2015.01.007
82. Chen B, Liu Y, Peng X, Wu Y, Qin S. Baton: symphony of random testing and concolic testing through machine learning and taint analysis. Sci China Inform Sci. 2022;66(3):132101. https://doi.org/10.1007/s11432-020-3403-2
83. Hershkovich E, Stern R, Abreu R, Elmishali A. Prioritized test generation guided by software fault prediction. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2021;218–25.
84. Ji S, Zhang P. Neural network based test case generation for data-flow oriented testing. In *IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019;35–6.
85. Helali Moghadam M, Saadatmand M, Borg M, Bohlin M, Lisper B. Machine learning to guide performance testing: An autonomous test framework. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019;164–7.
86. Koo J, Kulkarni M, Bagchi S. Pyse: Automatic worst-case test generation by reinforcement learning. In *IEEE Conference on Software Testing, Validation and Verification (ICST), 136–147*, 2019.
87. Moghadam MH. 2019. Machine learning-assisted performance testing. In Proceedings of the 2019 27th 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; 1187–9.
88. Luo Q, Poshyvanyk D, Nair A, Grechanik M. Forepost: A tool for detecting performance problems with feedback-driven learning software testing. In *International Conference on Software Engineering Companion, Association for Computing Machinery, New York, NY, USA, ICSE, 593–596*, 2016.
89. Sedaghatbaf A, Moghadam MH, Saadatmand M. Automated performance testing based on active deep learning. In *IEEE/ACM International Conference on Automation of Software Test (AST), 11–19*, 2021.
90. Moghadam MH, Saadatmand M, Borg M, Bohlin M, Lisper B. An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning. Softw Qual J. 2022;30(1):127–59. https://doi.org/10.1007/s11219-020-09532-z
91. Chen S, Haque M, Liu C, Yang W. Deepperform: An efficient approach for performance testing of resource-constrained neural networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery*: New York, NY, USA, ASE '22, 2023. https://doi.org/10.1145/3551349.3561158
92. Schulz H, Okanović D, van Hoorn A, Tu˙ma P. 2021. Context-tailored workload model generation for continuous representative load testing. In *ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, New York, NY, USA, ICPE, 21--32*.
93. Mudarakola L, Sastry J. A neural network based strategy (nnbs) for automated construction of test cases for testing an embedded system using combinatorial techniques. Int J Eng Technol (UAE). 2018;7(1.3):74–81.
94. Mudarakola L, Sastry J, Vudatha C. Generating test cases for testing web sites through neural networks and input pairs. Int J Appl Eng Res. 2014;9(22):11819–31.
95. Patil R, Prakash V. Neural network based approach for improving combinatorial coverage in combinatorial testing approach. J Theor Appl Inf Technol. 2018;96(20):6677–87.
96. Duy Nguyen C, Tonella P. Automated inference of classifications and dependencies for combinatorial testing, 2013;622–627.

97. Gholami F, Haghighi H, Asl MV, Valueian M, Mohamadyari S. A classifier-based test oracle for embedded software. In *Real-Time and Embedded Systems and Technologies (RTEST)*, 2018;104–11.
98. Ibrahimzada AR, Varli Y, Tekinoglu D, Jabbarvand R. Perfect is the enemy of test oracle. 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, 2022;70–81. https://doi.org/10.1145/3540250.3549086
99. Kamaraj K, Lanitha B, Karthic S, Prakash PNS, Mahaveerakannan R. A hybridized artificial neural network for automated software test oracle. Comput Syst Sci Eng. 2023;45(2):1837–50. https://doi.org/10.32604/csse.2023.029703. https://www.techscience.com/csse/v45n2/50392
100. Makondo W, Mapanga I, Kadebu P. Exploratory test oracle using multi-layer perceptron neural network. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI), 1166–1171*, 2016.
101. Shahamiri S, Wan Kadir W, Bin Ibrahim S. An automated oracle approach to test decision-making structures, vol. 5, 2010;30–4.
102. Tsimpourlas F, Rajan A, Allamanis M. Supervised learning over test executions as a test oracle. In *ACM Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA, SAC*, 2021;1521–31.
103. Tsimpourlas F, Rooijackers G, Rajan A, Allamanis M. Embedding and classifying test execution traces using neural networks. IET Softw. 2022;16(3):301–16. https://doi.org/10.1049/sfw2.12038
104. Braga R, Neto PS, Rabêlo R, Santiago J, Souza M. 2018. A machine learning approach to generate test oracles. Brazilian Symposium on Software Engineering. In Association for Computing Machinery: New York, NY, USA, SBES;142–51.
105. Chen K, Li Y, Chen Y, Fan C, Hu Z, Yang W. Glib: Towards automated test oracle for graphically-rich applications. In *ACM Joint Meeting of European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery*: New York, NY, USA, ESEC/FSE, 2021a;1093–104.
106. Khosrowjerdi H, Meinke K. Learning-based testing for autonomous systems using spatial and temporal requirements. In *International Workshop on Machine Learning and Software Engineering in Symbiosis, Association for Computing Machinery, New York, NY, USA, MASES, 6–15*, 2018.
107. Khosrowjerdi H, Meinke K, Rasmusson A. Learning-based testing for safety critical automotive applications. Lect Notes Comput Sci. 2017; 10437 LNCS: 197–211. https://doi.org/10.1007/978-3-319-64119-5_13
108. Rafi T, Zhang X, Wang X. Predart: Towards automatic oracle prediction of object placements in augmented reality testing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery*: New York, NY, USA, ASE '22, 2023. https://doi.org/10.1145/3551349.3561160
109. Aggarwal KK, Singh Y, Kaur A, Sangwan OP. A neural net based approach to test oracle. SIGSOFT Softw Eng Notes. 2004;29(3):1–6. https://doi.org/10.1145/986710.986725
110. Arrieta A, Ayerdi J, Illarramendi M, Agirre A, Sagardui G, Arratibel M. Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms. In *IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021;30–9.
111. Dinella E, Ryan G, Mytkowicz T, Lahiri SK. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery*: New York, NY, USA, ICSE '22, 2022;2130–41. https://doi.org/10.1145/3510003.3510141
112. Ding J, Zhang D. A machine learning approach for developing test oracles for testing scientific software, 2016. volume 2016-January, 390–395.
113. Gartziandia A, Arrieta A, Agirre A, Sagardui G, Arratibel M. Using regression learners to predict performance problems on software updates: A case study on elevators dispatching algorithms. In *ACM Symposium on Applied Computing, Association for Computing Machinery New York, NY, USA, SAC, 135–144*, 2021.
114. Gartziandia A, Arrieta A, Ayerdi J, Illarramendi M, Agirre A, Sagardui G, Arratibel M. Machine learning-based test oracles for performance testing of cyber-physical systems: An industrial case study on elevators dispatching algorithms. J Softw Evol Process. 2022;34(11):e2465. https://doi.org/10.1002/smr.2465
115. Jin H, Chen N, Gou Z, Wang S. Artificial neural network for automatic test oracles generation. Int Conf Comput Sci Softw Eng. 2008;2: 727–30.
116. Monsefi A, Zakeri B, Samsam S, Khashehchi M. Performing software test oracle based on deep neural network with fuzzy inference system. Commun Comput Inform Sci. 2019;891:406–17. https://doi.org/10.1007/978-3-030-33495-6_31
117. Sangwan OP, Bhatia PK, Singh Y. Radial basis function neural network based approach to test oracle. SIGSOFT Softw Eng Notes. 2011;36(5): 1–5. https://doi.org/10.1145/2020976.2020992
118. Shahamiri S, Kadir W, Ibrahim S, Hashim S. An automated framework for software test oracle. Inf Softw Technol. 2011;53(7):774–88. https://doi.org/10.1016/j.infsof.2011.02.006
119. Shahamiri S, Wan-Kadir W, Ibrahim S, Hashim S. Artificial neural networks as multi-networks automated test oracle. Autom Softw Eng. 2012; 19(3):303–34. https://doi.org/10.1007/s10515-011-0094-z
120. Shahamiri SR, Wan Kadir WM, Ibrahim S. 2010. A single-network ann-based oracle to verify logical software modules. In *2010 2nd International Conference on Software Technology and Engineering*, V2–NaN.
121. Singhal A, Bansal A, Kumar A. An approach to design test oracle for aspect oriented software systems using soft computing approach. Int J Syst Assur Eng Manag. 2016;7(1):1–5. https://doi.org/10.1007/s13198-015-0402-2
122. Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. Int J Intell Syst. 2002;17(1):45–62. https://doi.org/10.1002/int.1002
123. Vineeta AS, Bansal A. Generation of test oracles using neural network and decision tree model. In *Confluence The Next Generation Information Technology Summit (Confluence)*, 2014;313–18.
124. Ye M, Feng B, Zhu L, Lin Y. Automated test oracle based on neural networks. In *2006 5th IEEE International Conference on Cognitive Informatics*, vol. 1, 2006;517–22.
125. Ye M, Feng B, Zhu L, Lin Y. Neural networks based automated test oracle for software testing. Lect Notes Comput Sci. 2006b; 4234 LNCS - III: 498–507.
126. Yu H, Lou Y, Sun K, Ran D, Xie T, Hao D, et al. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery*: New York, NY, USA, ICSE '22, 2022;163–74. https://doi.org/10.1145/3510003.3510149
127. Zhang R, Wang Y-W, Zhang M-Z. Automatic test oracle based on probabilistic neural networks. Adv Intell Syst Comput. 2019;752:437–45. https://doi.org/10.1007/978-981-10-8944-2_50

128. Hiremath DJ, Hasselbring W, Rath W. Automated identification of metamorphic test scenarios for an ocean-modeling application. In *IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2020;62–3.
129. Hiremath DJ, Claus M, Hasselbring W, Rath W. Towards automated metamorphic test identification for ocean system models. In *IEEE/ACM International Workshop on Metamorphic Testing (MET)*, 2021;42–6.
130. Spieker H, Gotlieb A. Adaptive metamorphic testing with contextual bandits. J Syst Softw. 2020;165(110574). https://doi.org/10.1016/j.jss.2020.110574
131. Kanewala U, Bieman J. Using machine learning techniques to detect metamorphic relations for programs without test oracles. 1–10, 2013.
132. Kanewala U, Bieman J, Ben-Hur A. Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels. Softw Test Verification Reliab. 2016;26(3):245–69. https://doi.org/10.1002/stvr.1594
133. Korkmaz O, Yilmaz C. Sysmodis: A systematic model discovery approach, 2021. 67–76.
134. Nair A, Meinke K, Eldh S. Leveraging mutants for automatic prediction of metamorphic relations using machine learning. In *ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, Association for Computing Machinery, New York, NY, USA, MaLTeSQuE, 2019*, 2019;1–6.
135. Shu G, Lee D. Testing security properties of protocol implementations - a machine learning based approach. In *International Conference on Distributed Computing Systems (ICDCS)*, 2007;25.
136. Zhang P, Zhou X, Pelliccione P, Leung H. Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network. IEEE Access. 2017;5:21791–805. https://doi.org/10.1109/ACCESS.2017.2758790
137. Hecht-Nielsen R. 1992. Iii.3 - theory of the backpropagation neural network**based on "nonindent" by robert hecht-nielsen, which appeared in proceedings of the international joint conference on neural networks 1, 593–611, june 1989. © 1989 ieee. In Neural Networks for Perception, H. Wechsler (ed.). Academic Press; 65–93. https://www.sciencedirect.com/science/article/pii/B9780127412528500108
138. Graves A. Long Short-Term Memory. Springer Berlin Heidelberg: Berlin, Heidelberg, 2012;37–45. https://doi.org/10.1007/978-3-642-24797-2_4
139. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I. 2017. Attention is all you need. Advances in Neural Information Processing Systems. In Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
140. Sutton RS, Barto AG. Reinforcement Learning, Second Edition An Introduction, 2018. 550 pp.

How to cite this article: Fontes A, Gay G. The integration of machine learning into automated test generation: A systematic mapping study. Softw Test Verif Reliab. 2023;33(4):e1845. https://doi.org/10.1002/stvr.1845