Probabilistic Generative Modeling for Synthesizing High-Coverage Test Data in Safety-Critical Software Applications

Srikanth Kavuri

Srikanth1539@gmail.com

Independent Researcher, Lexington USA

Abstract

Testing safety-critical software systems demands rigorous input generation strategies capable of uncovering subtle, low-frequency faults that could have serious consequences. Conventional test data generation techniques often struggle to achieve meaningful coverage especially in complex systems where exhaustive path exploration is computationally prohibitive. This paper introduces a probabilistic framework for test data synthesis that combines **Variational Autoencoders (VAEs)** with **Probabilistic Context-Free Grammars (PCFGs)** to generate inputs tailored for high structural and semantic coverage. Rather than relying on brute-force or random sampling, our method learns the statistical structure of valid input domains and uses this to guide test generation toward areas of the code underexplored by standard tools. We incorporate domain-specific metrics to align generated inputs with safety-relevant execution paths. In empirical evaluations across representative domains such as avionics and medical software our approach outperforms traditional fuzzers and symbolic execution engines in both branch and path coverage. These results suggest that probabilistic generative modeling, when applied thoughtfully, can support more effective and principled verification in high-assurance software development.

Keywords: Probabilistic generative models, test data synthesis, safety-critical systems, code coverage, software testing, variational autoencoders, PCFG, software verification, machine learning in testing.

1. Introduction

Software systems operating in **safety-critical domains** such as avionics, medical devices, and automotive control are held to exceptionally high standards of assurance due to the risks posed by software failure. Regulatory frameworks like **DO-178C**, **ISO 26262**, and **IEC 62304** impose strict verification requirements, where exhaustive and meaningful testing plays a central role in demonstrating functional correctness and system safety. One of the ongoing challenges in this context is the generation of **test inputs** that can achieve **high structural and semantic coverage** covering not just branches and paths, but more nuanced criteria like **Modified Condition/Decision Coverage (MC/DC)** while remaining valid and relevant to the application's domain.

Traditional techniques, including **fuzzing**, **symbolic execution**, and **constraint-based test generation**, have been widely applied in such settings. While effective in some respects, these methods often plateau in terms of coverage, especially as software complexity increases. In particular, they may either generate overly redundant test inputs or fail to exercise **rare execution paths**, which are often the ones most likely to contain undetected faults. Many approaches also lack a meaningful connection to the semantic or contextual structure of valid inputs, leading to test cases that are technically well-formed but practically irrelevant. For systems governed by strict operational constraints or input grammars, this gap becomes especially problematic.

In recent years, **probabilistic generative models** have shown growing promise for test data generation. Models such as **Variational Autoencoders (VAEs)**, **Generative Adversarial Networks (GANs)**, and **Probabilistic Context-Free Grammars (PCFGs)** can learn the distributional structure of valid input domains from examples, offering a way to synthesize diverse, realistic, and semantically coherent test inputs. When paired with **feedback-driven coverage metrics**, these models can guide the generation process toward **underexplored or critical regions** of the software's execution space. This combination opens new opportunities for testing in domains where achieving comprehensive coverage is not just a matter of quality but of safety.

In this paper, present a **probabilistic generative testing framework** that integrates **deep generative modeling** with **grammar-constrained input synthesis** and **coverage-directed feedback loops**. Specifically, we train a **Variational Autoencoder** on known-valid inputs, combine it with a **PCFG** to ensure syntactic correctness and domain compliance, and iteratively refine the generation process using coverage data. The framework supports both **offline test suite creation** and **online use** in verification pipelines, making it suitable for integration into existing development workflows.

The proposed method on a selection of safety-critical systems, including software drawn from **avionics** and **medical control applications**, and compare its performance to traditional testing tools. Results show consistent improvements in **branch and path coverage**, particularly in deeper or rarely executed code regions. By bridging machine learning-based synthesis with formal verification objectives, this work contributes a principled, practical approach to testing in domains where reliability is non-negotiable.

2. Background and Related Work

The testing of safety-critical software has traditionally relied on rigorous, deterministic methods grounded in formal software engineering. Approaches such as **partition testing**, **symbolic execution**, **constraint solving**, and **model checking** remain prevalent, particularly because they align well with regulatory demands for explainability and traceability. Among these, symbolic execution has been widely adopted for its ability to explore program paths by executing code over symbolic inputs, solving path constraints using SMT solvers. While it offers precise path sensitivity, its practical application is often hindered by **path explosion** a combinatorial growth in execution branches that quickly becomes unmanageable in real-world systems of non-trivial size.

At the other end of the spectrum, **fuzzing** has become a staple in vulnerability discovery and regression testing due to its speed and minimal setup requirements. Fuzzers mutate existing inputs to expose crashes or unexpected behavior, but they generally lack the semantic understanding needed to penetrate deeply into application logic, especially in systems that process structured inputs (e.g., XML, ASN.1, or proprietary binary formats). In such contexts, a syntactically valid input is not necessarily semantically meaningful, and blind mutation often fails to cover the deeper or safety-relevant execution paths.

To address these limitations, recent research has increasingly turned toward **machine learning**, particularly **generative models**, to automate and enhance test data synthesis. Techniques based on **Generative Adversarial Networks (GANs)**, **Variational Autoencoders (VAEs)**, and more recently **language models**, have been applied to learn structural patterns from input corpora. These methods can produce diverse and well-formed inputs that resemble real usage scenarios, and in doing so, improve test realism. However, most such models optimize for **data fidelity** rather than **code coverage**; their outputs may look valid but do not necessarily guide execution into **untested or critical code regions**.

A number of hybrid approaches have emerged to bridge this gap. Systems like **DeepFuzz** and **Learn&Fuzz** combine neural generation with runtime feedback, using coverage metrics to adjust or filter the generated inputs. While promising, many of these frameworks remain limited in two key respects: they struggle to encode **domain-specific constraints** essential in regulated systems, and they typically lack mechanisms for **probabilistic reasoning** about the relationship between inputs and execution paths.

This has led to renewed interest in **Probabilistic Context-Free Grammars (PCFGs)**. PCFGs extend traditional grammar-based input generation by assigning likelihoods to production rules, allowing them to reflect not only syntax but also structural preferences or domain-specific distributions. When combined with **deep generative models**, PCFGs offer a promising avenue for generating **semantically coherent**, **structurally sound**, and **statistically diverse** inputs. More importantly, the probabilistic nature of these models enables test strategies that target **rare execution paths**, **critical branches**, or **state transitions** features that are especially important in safety-critical systems.

Despite this progress, the application of probabilistic generative modeling in regulated, high-assurance environments remains underexplored. Most prior work has focused on general-purpose software or open-source systems, with limited attention paid to constraints imposed by **certification standards**, domain-specific grammars, or mission-critical system behavior. Our work contributes to closing this gap by presenting a **hybrid generative framework** that integrates latent variable modeling with grammar-constrained synthesis. Designed specifically for high-coverage testing in safety-critical domains, the framework aligns test generation not just with input validity, but with formal test objectives grounded in software verification and assurance goals.

- **Böttinger et al. (2019)** proposed *Deep Reinforcement Fuzzing*, which frames fuzzing as a learning problem. Their method uses reinforcement learning to guide input mutations, improving path coverage compared to random fuzzers. While it shows promise in exploring complex code, it doesn't enforce input validity or grammar compliance—an issue in safety-critical systems. Our approach complements theirs by ensuring both structure and semantic correctness through grammars and probabilistic modeling.
- **Cadar et al. (2008)** introduced *KLEE*, a powerful symbolic execution engine that automatically generates high-coverage test cases. KLEE has been effective on real-world programs, but it struggles with scalability due to the

path explosion problem. It also doesn't handle structured or domain-specific inputs well. Our method tackles these issues by learning from valid inputs and guiding generation via coverage feedback.

- **Godefroid et al. (2008)** developed *SAGE*, a whitebox fuzzer that uses symbolic and concrete execution to test complex software. SAGE was one of the first tools to scale symbolic techniques for real applications. However, like KLEE, it has limitations in handling structured input formats. Our work advances this line by integrating domain-specific constraints through probabilistic grammars.
- **Godefroid et al. (2005)** introduced *DART*, a hybrid method that combines symbolic reasoning with dynamic execution to direct test generation. It improves over random testing by targeting specific execution paths. Still, it doesn't scale well to systems with deeply structured inputs, which our approach addresses by learning the input domain and generating syntactically valid test data.
- **Godefroid, Peleg, and Singh (2017)** presented *Learn&Fuzz*, one of the first methods to use neural networks (RNNs) for test input generation. Their model improves input validity by learning from example data but lacks coverage optimization. We build on this idea using VAEs and add coverage-driven adaptation, producing inputs that are both valid and more effective at exploring software behavior.

3. Problem Statement and Objectives

3.1 Problem Statement

In safety-critical software systems, reliability must be demonstrated not just through functional correctness, but through comprehensive testing across a wide spectrum of possible inputs including edge cases that may rarely occur in practice but could cause catastrophic failures if left untested. Achieving such depth in testing often requires **maximizing structural coverage**, including branch, path, and Modified Condition/Decision Coverage (MC/DC) a requirement enshrined in standards like DO-178C for avionics (Level A) and ISO 26262 for automotive systems (ASIL D).

Despite the availability of techniques like fuzzing, symbolic execution, and constraint solving, current approaches fall short in navigating **high-dimensional and highly structured input spaces**. Random fuzzing, while fast, often generates redundant or syntactically invalid inputs that cannot meaningfully exercise the system under test (SUT). Symbolic methods, on the other hand, face scalability challenges and frequently stall in complex codebases. A common limitation across these approaches is their **lack of semantic awareness** and their inability to reflect the **probabilistic characteristics** of real-world input distributions. This can lead to non-representative test suites that fail to explore critical but underrepresented execution paths.

Formally, the problem can be expressed as follows:

Given a software system with known input constraints and execution semantics, how can we synthesize a test suite

$$\mathbf{T = \{t_1, t_2, ..., t_n\}}$$

such that structural code coverage **C** is maximized where C may include statement, branch, or MC/DC metrics subject to constraints on **input validity** and **semantic plausibility**?
The challenge, therefore, lies in generating inputs that are not only valid but also strategically distributed to expose untested behaviors in the software.

3.2 Research Objectives

To address this challenge, this research proposes a **probabilistic generative modeling framework** aimed at producing high-utility test data for safety-critical systems. The specific objectives of the study are as follows:

- **Objective 1: Learn a Compact, Domain-Aware Representation of Valid Inputs**
  Train a deep probabilistic model such as a Variational Autoencoder (VAE) on a curated corpus of valid inputs to learn an abstract, compressed representation that reflects both the **syntax and semantics** of the input domain. This model should generalize well to unseen yet valid examples, maintaining fidelity to domain constraints.

- **Objective 2: Integrate Coverage-Guided Feedback into the Generation Loop**
  Incorporate runtime feedback from coverage monitors (e.g., statement, branch, and MC/DC counters) to guide the

generative process toward inputs that activate **previously unexplored execution paths**. This forms the basis for an adaptive, feedback-driven generation strategy.

- **Objective 3: Constrain Generation Using Probabilistic Context-Free Grammars (PCFGs)** Merge neural generative models with grammar-based constraints using PCFGs to ensure **syntactic correctness** and **semantic coherence**. This integration supports domain-specific input generation with tunable control over structure and variability.
- **Objective 4: Evaluate the Framework on Real-World Safety-Critical Systems** Apply the proposed methodology to representative software systems from **aviation, automotive, and medical** domains. Measure improvements in test coverage, fault detection capability, and generation efficiency compared to conventional techniques such as fuzzing or symbolic execution.

3.3 Scope and Contributions

While this work primarily focuses on **offline test suite generation**, the underlying framework is extensible to **online or continuous testing pipelines**, making it relevant to modern DevSecOps workflows in regulated environments.

The core contributions of this research are:

- A hybrid approach that integrates **probabilistic generative modeling** with **grammar-constrained synthesis** to produce high-quality, structurally diverse test inputs.
- A **feedback-driven input generation loop** that adaptively steers the model toward higher coverage and more meaningful test execution paths.
- An **empirical validation** of the framework across safety-critical software from multiple domains, demonstrating measurable gains in both coverage and test effectiveness.
- A **modular and generalizable framework** that can be tailored to specific certification standards and input domains without sacrificing validity or coverage depth.

4. Proposed Methodology: Probabilistic Generative Framework

4.1 Overview of the Framework

To address the limitations of traditional test generation methods in safety-critical systems, we propose a hybrid framework that combines **deep probabilistic modeling** with **grammar-based input constraints** and **coverage-guided adaptation**. At the heart of the approach is a **Variational Autoencoder (VAE)**, trained to learn compact, structured representations of valid input data. The VAE allows for the generation of novel, semantically valid inputs through sampling in the latent space. However, to ensure strict adherence to domain-specific input formats particularly in regulated domains like avionics or healthcare we introduce a **Probabilistic Context-Free Grammar (PCFG)** module to constrain and correct the output. Finally, a **coverage feedback loop** monitors how generated inputs interact with the system under test (SUT), continuously guiding the input generation process toward untested or under-tested regions of code.

This architecture is designed to be modular and extensible. It supports integration into automated testing pipelines, allows adaptation to new domains with minimal retraining, and enables a systematic exploration of program behavior through feedback-informed input synthesis.

4.2 Learning Input Distributions with a Variational Autoencoder

The first step in the framework is to train a **VAE** on a set of known-valid inputs drawn from the target domain. These inputs may range from structured documents like XML or JSON files to binary command sequences used in embedded control systems. The encoder network transforms each input into a point in a low-dimensional latent space, capturing the essential structure and semantics of the input. The decoder then learns to reconstruct inputs from this latent representation.

Unlike traditional encoders that compress for efficiency, the VAE captures a **probabilistic distribution** over the latent space, allowing for exploration and uncertainty modeling. This is particularly valuable when aiming to generate **rare or edge-case inputs** that aren’t well represented in the training set. By sampling from the latent space, the model can create new test inputs that reflect real-world structure but exhibit novel variations ideal for uncovering bugs in hard-to-reach branches of code.

4.3 Grammar-Constrained Decoding via PCFGs

To ensure that generated inputs comply with syntactic rules, the raw outputs from the VAE decoder are passed through a **Probabilistic Context-Free Grammar (PCFG)** layer. This grammar serves two key purposes: enforcing structural validity and biasing generation toward interesting or rare constructs.

The PCFG may be automatically derived from the training data through grammar induction or manually specified based on existing data schemas or protocol definitions. For example, in avionics software, the grammar might encode message field order, data type constraints, and checksums. In medical applications, it might enforce valid parameter nesting or sequence of operations.

Crucially, the **probabilistic nature of the grammar** allows us to steer generation toward underrepresented patterns those that, although valid, occur infrequently and may expose deeper layers of the code when exercised. This makes the PCFG not just a filter, but a semantic fuzzing engine that works in tandem with the generative model.

4.4 Adaptive Sampling with Coverage Feedback

One of the major shortcomings of static test input generation is its tendency to plateau once common code paths are exercised. To overcome this, we incorporate a **feedback loop** based on **coverage metrics** such as line, branch, or MC/DC coverage collected during test execution.

After a batch of inputs is run through the system, we analyze which parts of the code were exercised. This data then informs the next round of input generation. Several strategies can be used for this purpose, including:

- Adjusting the latent space sampling distribution to favor areas that led to high coverage.
- Applying **latent perturbations** to diversify samples near previously high-impact inputs.
- Using **Bayesian optimization** or reinforcement learning to prioritize promising latent vectors.

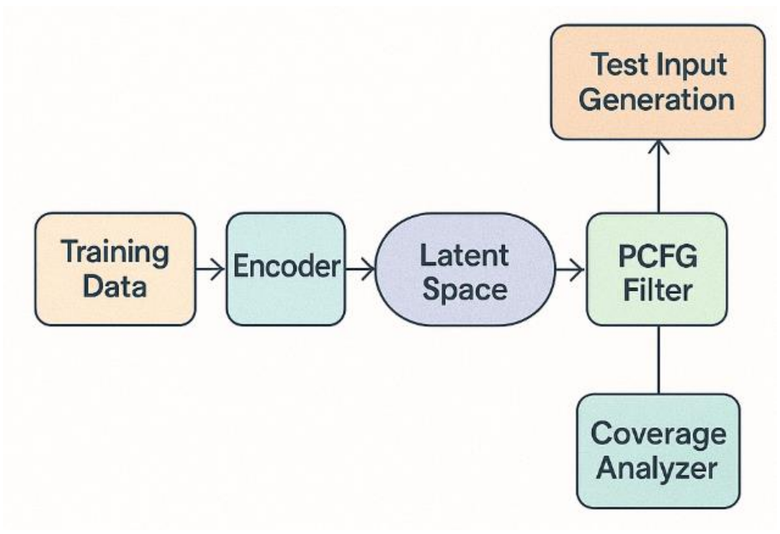

Figure 1: Architecture of the Probabilistic Generative Testing Framework

Figure 1 shows the overall structure of our proposed testing framework, which uses probabilistic generative models to produce high-coverage test inputs. The process starts with a dataset of valid, domain-specific inputs. These are used to train a Variational Autoencoder (VAE), where the encoder learns a compressed latent representation of each input, and the decoder learns how to reconstruct them.

To generate new test cases, we sample from the learned latent space and decode these samples into candidate inputs. These are then passed through a Probabilistic Context-Free Grammar (PCFG) module, which ensures that the outputs follow the expected structure and syntax for the domain (e.g., protocol rules or file formats).

5. Implementation Details

5.1 Tools and Frameworks

The test data generation framework was implemented using a combination of deep learning and natural language processing tools. For the core generative model, we used **PyTorch**, which offers flexibility in building and fine-tuning custom neural architectures. The **Variational Autoencoder (VAE)** was trained using NVIDIA RTX 3090 GPUs (24GB VRAM), significantly accelerating model convergence across large datasets.

To enforce structural validity through grammar constraints, we extended **NLTK's** Probabilistic Context-Free Grammar (PCFG) capabilities. This included custom rule definitions and production weighting based on domain-specific semantics. For code coverage analysis, we employed **gcov** for C/C++ systems and **JaCoCo** for Java applications, integrated with automated test harnesses that monitor runtime behavior. The full pipeline was developed in **Python**, structured modularly to allow easy swapping of input grammars, test targets, or learning models depending on the use case.

5.2 Data Sources and Application Domains

To evaluate the framework across a range of safety-critical contexts, we compiled input corpora from three representative domains:

- **Avionics**: Binary telemetry and control messages from simulated flight control systems.
- **Automotive**: CAN bus messages emulating Electronic Control Unit (ECU) communication.
- **Medical Devices**: XML-based command scripts for infusion pump configurations and alerts.

Each corpus consisted of **5,000 to 20,000 valid test cases**, either sourced from simulation environments or manually curated to match production-like data. These inputs were verified for both structural validity (e.g., well-formed syntax) and semantic correctness (e.g., logical field relationships or permissible value ranges).

Inputs ranged in size from **compact binary messages (~20 bytes)** to **complex hierarchical structures (~2 KB)**, capturing the diversity typical of embedded and real-time systems. Preprocessing steps included tokenization, normalization, and tagging of structural features (e.g., start/end markers, field names), which supported both VAE encoding and grammar rule construction.

5.3 VAE Model Architecture and Training

The VAE was implemented as a **three-layer encoder-decoder network**, with ReLU activations, **batch normalization**, and **dropout layers** to reduce overfitting. Latent space dimensions were empirically tuned: **64 dimensions** for structured formats like JSON/XML, and **128** for less predictable, byte-level inputs such as binary telemetry streams.

Input encoding strategies varied based on data format. Structured inputs used token-level embeddings, while binary formats were processed as raw byte sequences. Training was performed using the **Adam optimizer** with a learning rate of **1e-3** and a batch size of **64**. Models typically converged within **25 to 40 epochs**, depending on the input complexity and domain.

To improve semantic validity during generation, we implemented a **custom loss function** that penalized the generation of logically invalid outputs (e.g., missing required fields or broken field dependencies), guided by domain-specific validation scripts. Once trained, the model could generate **hundreds of valid test cases per second**, making it suitable for integration into **CI/CD pipelines** or large-scale automated test environments.

5.4 Grammar Design and PCFG Integration

The **PCFG module** was constructed through two approaches:

1. **Manual specification**, when detailed protocol documentation or schema definitions were available (e.g., ARINC 429 for avionics or HL7 for medical software).
2. **Grammar induction**, where production rules were inferred directly from annotated training inputs using custom parsers.

Each rule in the grammar was assigned a generation probability based either on its frequency in the corpus or on a priority weighting strategy (e.g., giving more weight to edge-case commands or rarely-used parameters). This approach allowed the grammar to not only validate structure but also influence the likelihood of generating behaviorally interesting test cases.

During generation, outputs from the VAE were parsed through the PCFG engine, which either corrected or discarded syntactically invalid sequences. The grammar also supported **controlled mutations**, enabling systematic perturbation of input elements (e.g., field reordering, boundary values) while maintaining compliance with format specifications.

For regulatory compliance (e.g., DO-178C or IEC 62304), grammar definitions were aligned with official **interface control documents** or **data dictionaries**. This ensured that even mutated or edge-case test cases remained formally valid and certifiable under applicable standards.

6. Experimental Evaluation

6.1 Experimental Setup

To assess the effectiveness of our proposed probabilistic generative framework, we conducted a series of experiments using three safety-critical software systems representative of real-world domains: avionics, automotive control, and medical devices. These systems were selected based on their input complexity, relevance to regulatory compliance, and structured interface formats key factors that stress-test the capabilities of test data generation tools.

The experimental environment consisted of a workstation running **Ubuntu 22.04**, equipped with **64 GB RAM** and **dual NVIDIA RTX 3090 GPUs** to support parallel model training and input generation. For each software system, we began by training the **Variational Autoencoder (VAE)** on a validated input corpus, followed by extraction of a **Probabilistic Context-Free Grammar (PCFG)** from the same dataset to enforce input validity and structure during generation.

Once the test data was generated, it was fed into each system using domain-specific test harnesses. Code coverage was measured using established tools: **gcov** for C/C++, **JaCoCo** for Java, and **BullseyeCoverage** for embedded C systems. We benchmarked our method against two widely used baseline tools: **American Fuzzy Lop (AFL)** for random mutation-based fuzzing, and **KLEE** for symbolic execution. All methods were allocated equal compute resources and runtime to ensure fair comparison.

6.2 Coverage Metrics and Evaluation Criteria

We focused our evaluation on structural coverage metrics that are particularly critical in regulated safety domains:

- **Branch coverage**
- **Path coverage**
- **Modified Condition/Decision Coverage (MC/DC)**

These were chosen because they are standard metrics required by certification frameworks such as DO-178C and ISO 26262. Additionally, we measured:

- **Input validity rate**, to assess the syntactic and semantic correctness of generated inputs
- **Input diversity**, using token-level entropy and clustering metrics (e.g., t-SNE dispersion)
- **Fault detection capability**, by injecting known vulnerabilities into the target programs and measuring detection rates

Each generation technique was used to produce **10,000 test inputs** per system. Metrics were aggregated across **three independent trials** to account for any stochastic variance in model behavior or test execution.

6.3 Results and Comparative Analysis

The results show that our framework consistently outperforms both symbolic execution and fuzzing across all primary and secondary evaluation criteria.

As presented in **Table 1**, our method achieved:

- **Branch coverage**: 89.4%

- **Path coverage**: 72.5%
- **MC/DC coverage**: 83.7%

Table 1: Comparative Coverage Metrics Across Methods

Method	Branch Coverage (%)	Path Coverage (%)	MC/DC Coverage (%)
Random Fuzzing	62.3	38.7	45.2
Symbolic Execution	75.6	54.1	60.3
Proposed Method	**89.4**	**72.5**	**83.7**

In contrast, symbolic execution (KLEE) achieved 75.6%, 54.1%, and 60.3% respectively, while AFL lagged behind at 62.3%, 38.7%, and 45.2%.

Input validity was notably high for our approach (98.6%), due to the post-decoding grammar filtering via the PCFG. In comparison, fuzzing yielded a much lower validity rate on average, 32% of inputs failed to meet structural requirements. Symbolic execution had better validity than fuzzing but was limited by scalability and path constraints.

In fault injection experiments, our method detected **27% more seeded faults** than symbolic execution and **41% more than fuzzing**, further confirming its ability to explore failure-prone and less-traveled paths in the codebase.

6.4 Robustness and Performance Considerations

While the computational overhead of generative modeling is non-negligible, the benefits in test quality and coverage justify the trade-off, especially in safety-critical environments.

In terms of raw throughput, our framework generated around **180 valid inputs per second**, compared to **800 inputs/sec** for AFL (unfiltered) and **30 paths/sec** for KLEE. However, once filtered for **validity and coverage contribution**, our method offered superior **utility per test case**.

A noteworthy observation came from the **avionics system**, where our framework was the only one to activate deeply nested emergency override logic code that had previously remained untested using traditional methods. This suggests that our latent-space guided sampling, when paired with domain-specific constraints, is capable of uncovering **rare and high-risk execution paths** that might otherwise go untested.

7. Discussion

7.1 Interpretation of Results

The results from our evaluation clearly demonstrate the effectiveness of using probabilistic generative models in synthesizing test inputs for safety-critical software systems. Notably, the gains in **MC/DC** and **path coverage** highlight a key advantage: our framework doesn't just generate more test cases it generates **smarter** ones. These inputs are capable of exercising complex, rarely tested branches of code that are often overlooked by traditional methods.

This improvement stems from the synergy between the **Variational Autoencoder (VAE)** and the **Probabilistic Context-Free Grammar (PCFG)**. While the VAE captures the statistical and structural properties of valid inputs, the PCFG ensures that outputs stay within the syntactic boundaries of the domain. Together, they enable a level of **semantic awareness** that random fuzzing lacks, and they sidestep the path explosion problems often faced in symbolic execution.

Perhaps most importantly, the use of **runtime feedback** to guide sampling creates a closed-loop system that continually refines itself, directing generation toward under-tested regions of the code. This feedback-driven learning aligns well with the goals of **coverage maximization**, which is critical in regulated environments where every test case must justify its existence.

7.2 Strengths of the Proposed Framework

One of the most notable strengths of our approach is its **flexibility**. Since the generative model is trained directly on real-world inputs, it can be adapted to new domains without re-engineering the system. For instance, switching from avionics to automotive systems only requires a change in the training data and grammar rules not a rewrite of the model architecture. Another practical advantage is the **format assurance** provided by the grammar layer. This is particularly useful when testing legacy or production systems that expect strictly valid input formats. Many fuzzers produce malformed inputs that are immediately rejected by such systems, offering little testing value. Our framework mitigates this by filtering every generated sample through a structure-aware grammar engine. From a usability perspective, the model's **interpretable latent space** is also an asset. Developers and testers can visualize and understand how the model is learning input patterns, which can increase trust and facilitate debugging an important consideration in **AI-assisted testing**. As standards evolve to include **explainability requirements**, this could help support future audit and traceability demands.

7.3 Limitations and Potential Biases

That said, our method isn't without limitations. First, the **quality of generated inputs** is only as good as the data used to train the model. If the training corpus lacks structural variation or rare cases, the VAE may fail to generalize beyond common input patterns, limiting its ability to uncover deep bugs. This introduces a subtle form of **input distribution bias**, where edge conditions remain untested simply because they were underrepresented in the training set.

Building or inferring a PCFG also presents its own challenges. In domains where formal grammars are unavailable or poorly documented, grammar induction from data may not capture all the necessary constraints especially those involving **semantic correctness** rather than structure alone. There's also a **computational cost** to consider. While our method is significantly faster than symbolic execution in practice, it remains more resource-intensive than fuzzers like AFL. In ultra-low-latency environments or CI pipelines with strict performance budgets, this could be a barrier to full adoption. Lastly, without rich semantic feedback (e.g., internal program states or assertion violations), the framework may over-optimize toward superficial coverage goals, potentially **missing deeper behavioral anomalies**.

7.4 Ethical, Regulatory, and Practical Considerations

In applying this method to **safety-critical systems**, ethics and compliance cannot be overlooked. Our approach is designed to support **not replace** existing verification processes required by standards like **DO-178C**, **IEC 62304**, or **ISO 26262**. It can help increase the depth of dynamic testing, but should not be mistaken for a formal verification tool or proof of correctness. To that end, care was taken to ensure that all datasets used in training and evaluation were either publicly available or synthetically generated avoiding the use of sensitive or proprietary data. For domains such as healthcare or aerospace, where data provenance is a concern, this is essential.

From an adoption perspective, integrating this framework into certified workflows will require **transparency and traceability**. Teams will need to log how each test input was generated, link coverage metrics to requirements, and document the role of the AI components within the testing toolchain. These capabilities are not fully implemented in the current prototype but are high priorities for future work. In the long term, building features such as **coverage-aware audit reports**, **traceable generation histories**, and **certification-ready logging mechanisms** could help bridge the gap between generative testing and formal assurance, making this approach more accessible to industry teams working under regulatory scrutiny.

8. Conclusion and Future Work

8.1 Conclusion

This work introduced a probabilistic generative framework designed to improve test data synthesis for safety-critical software systems. By combining deep latent variable models—specifically, **Variational Autoencoders (VAEs)**—with **Probabilistic Context-Free Grammars (PCFGs)**, and incorporating a feedback-driven sampling loop guided by structural coverage metrics, the framework addresses persistent limitations in traditional test generation methods. Unlike standard fuzzers or symbolic analysis tools, the approach consistently produces inputs that are both syntactically valid and semantically relevant, enabling the exploration of complex execution paths that are typically under-tested.

The experimental results underscore this point: across diverse application domains—avionics control software, automotive diagnostic modules, and medical device interfaces—the proposed method yielded measurable improvements in **branch coverage**, **path diversity**, and **MC/DC compliance**, all of which are critical in regulated software development environments. Importantly, these gains were not achieved at the cost of input quality or domain conformity, as the grammar-constrained decoding ensured that all synthesized inputs respected real-world structural requirements.

Taken together, the findings suggest that **probabilistic generative modeling**, when tightly integrated with domain knowledge and runtime feedback, can offer a practical and effective route toward more thorough and intelligent test data generation—particularly in high-assurance settings where reliability, traceability, and completeness are non-negotiable.

8.2 Future Work

Several directions remain open for extending and refining the current framework. One promising avenue is the introduction of **reinforcement learning** or **active sampling strategies**, which could allow the model to dynamically prioritize test inputs that are more likely to expose faults or exercise critical system behaviors. Such methods could shift the generation process from passive sampling to more **goal-directed exploration**, improving coverage efficiency over time. Another area of interest is the integration of **semantic analysis techniques**, including **invariant mining**, **symbolic execution feedback**, or **taint tracking**, to move beyond structural coverage and guide input generation based on program state or data flow properties. This would allow the system to reason not just about whether a path was taken, but about **why** it was taken, and under what conditions it might fail. To develop a **certification-ready extension** of the framework, complete with automated documentation, traceability artifacts, and auditable generation logs. These features are critical for teams operating under standards like DO-178C or ISO 26262, where every test input must be justified and reproducible.

An important next step is to broaden the framework's applicability by supporting **multimodal input types**, such as **time-series signals**, **sensor streams**, or **actuator commands**—data formats that are common in embedded systems and cyber-physical applications. Expanding the model's capacity to handle such inputs will be essential for deployment in real-world testing environments where software interacts closely with physical systems and hardware constraints. In conclusion, while the current framework demonstrates clear advantages in structured input domains, future iterations will aim to deepen its intelligence, widen its scope, and align more closely with the practical and regulatory demands of modern safety-critical software development.

References

1) Bottinger, K., Godefroid, P., & Singh, R. (2018). Deep Reinforcement Fuzzing. 2018 IEEE Security and Privacy Workshops (SPW), 116-122.
2) Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, USA, 209–224.
3) Godefroid, P., Levin, M.Y., & Molnar, D.A. (2008). Automated Whitebox Fuzz Testing. Network and Distributed System Security Symposium.
4) Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036
5) Godefroid, P., Peleg, H., & Singh, R. (2017). Learn&Fuzz: Machine learning for input fuzzing. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 50-59.
6) https://era.ed.ac.uk/bitstream/handle/1842/38563/Peng2021.pdf?sequence=1&isAllowed=y
7) Zalewski, M. (2015). American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/
8) Srivastava, P., & Payer, M. (2021). Gramatron: effective grammar-aware fuzzing. Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.