**The Future of Software Development: The Art of Code GAN-eration**

Mansi Dhawan[a](#a),[1](#contri),[\*](#corr_auth), Tripti Aggrawal[a](#a)[,1](#contri)

a Independent Scholar; Delhi India

1 Equal contribution: co-authors listed in alphabetical order

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Abstract**

Code generation: automatic creation or completion of code, is still one of the most challenging problems which have lately garnered a lot of attention. There are abounding tools/methods available in the market that have tried to tackle this problem differently. Some key approaches include a transformer model, recurrent neural network, and abstract syntax tree. With the advent of the generative adversarial network (GAN), text generation has become even more popular. Many researchers are leveraging it to develop models and tools for automatic coding. Code generation using GANs has several applications, including code review, code synthesis, and program repair. In this paper, we have analyzed how GAN is being leveraged for code generation while also looking at the limitations and future of the same vis-à-vis other standard deep learning techniques in use currently. We found that while code generation using GANs has shown promising results, several challenges remain unresolved, including the generation of code that is both syntactically and semantically correct, as well as the need for large amounts of training data. However, with continued research in this area, code generation using GANs has the potential to revolutionize software development and accelerate the creation of new software applications.

**Keywords**: [Generative Adversarial Network, Code Generation, Deep Learning, Text Generation, NLP, Software Development]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Introduction**

Developing software or any other tool involving coding always requires much effort. A significant amount of research and development has focused on creating techniques that can automatically generate code, which is coined as "Code Generation". It has become increasingly popular in recent years because it accelerates the process, enhances productivity, and helps developers with standard and convoluted tasks. This technique automatically produces source code in statically and dynamically typed languages. Massive efforts have been made to solve this problem, with various approaches, including basic Unified Modeling Language (UML) models and state-of-the-art neural networks, attempted and evaluated. With the recent developments in generative techniques, the ongoing research in the field has taken a turn.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*Corresponding author Email Address: mansi.dhawan@outlook.in

Several code generation tools based on Neural networks have gained much attention, such as "DeepCoder" by Balog et al. ([2017](#deepcoder)), which used a Recurrent Neural Network (RNN) -based approach to automatic code generation. In addition, "CodeBERT" by Feng et al. [(2020)](#bert) used a transformer-based method for code completion and bug fixing.

These tools use a generative system coupled with various techniques, including graph-based representations, machine-learning models, etc. The choice of tool will depend on the specific needs and goals of the developer or development team. Moreover, the end goal is not just to produce code from a machine that hits the mark but to build a code that is efficient, secure, reliable, reconfigurable, re-usable, and shortens the software development cycle. Although GANs have shown promising results in generating code, the generated code may still require human intervention to ensure correctness, security, and efficiency.

The main objective of this paper is to:

* Give a walkthrough of existing methodologies used for automated generated programming.
* Delve into GAN and how it can be utilized for code generation.
* Discuss the risks, constraints, and future of Gan and automated programming.

The rest of the paper is organized as follows: Section 2 briefly explains related work to code generation using various techniques besides GAN. Section 3 explains these techniques and introduces Gan and how it is used in code generation. Section 4 compares GAN’s performance with other models while explaining its risk, limitations, and future. Finally, section 5 concludes the discussion.

**2. Literature review**

**2.1 UML**

Several researchers developed various methods/approaches using UML for code generation, starting with Stuermer et. Al. ([2007](#stuermer)) presented a review of the field of model-based code and provided a solution to improve the efficiency and reliability of the tools used to generate codes. The approach has the potential to significantly improve the efficiency and effectiveness of software development, particularly in safety-critical applications where reliability is paramount. Jakimi et al. ([2009](#jakimi)), in their paper, proposes a novel approach for code generation task using UML and state machines. The code generated by the model adheres to the object-oriented designed principles; a case study on an elevator control system has been done to test its effectiveness on complex and critical systems.

**2.2 RNNs and Abstract Syntax Trees (ASTs)**

RNNs, specifically LSTM and ASTs, have also been used by various authors to develop a model/framework, including Pengcheng Yin et al. ([2017](#penchen)) syntactic neural model combined abstract syntax trees and an RNN-based encoder-decoder model for a fundamental code generation problem. The model has been tested on various open-source datasets and generates code in Python. Similarly, Rabinovich et al. ([2017](#maxim)) proposed abstract syntax networks based on a bidirectional LSTM-based encoder-decoder architecture. The code generated as output is represented as ASTs instead of a particular programming language. The model is tested on various datasets, and the results are competitive with similar models. Finally, the authors concluded that the presented approach is suitable for semantic parsing tasks, and in the future, this study could be extended to general structure outputs. Pix2code, a GUI-to-code model, was developed by Tony Beltramelli ([2017](#pix)). The model is created by combining CNNs with LSTM networks. The model is trained to learn from the pixel values of the input image. The author said this model could generate code in different programming languages by training it on different datasets. Furthermore, the code is humanly readable, and the results outperformed various other approaches. R. Tiwnag et al. ([2019](#tiwang)) paper proposed a novel deep learning-based approach for code generation tasks. This model is based on LSTM and MLP-based architecture. The model is trained on a large dataset of codes processed as Abstract Syntax Trees rather than simple code texts. This has been tested on Python language-based tasks and has reported comparative results to the other existing models. The authors have claimed that their model has performed better than different neural network base approaches.

**2.3 Transformer**

Various developers have also utilized transformer architecture in their research. A paper by Parvez, Md Rizwan, et al. ([2021](#parvez)) proposed a framework called REDCODER which works on the retrieval-based technique suitable for both code summarization and code generation. This framework inputs code and textual description and extracts the most relevant summary/code from the retrieval database. Another significant development includes CodeX, a tool developed by OpenAI, and the research has been delineated in a paper by Chen et al. ([2021](#codex)). This paper demonstrates that CodeX outperforms various existing models in several aspects, including quality and diversity of code. The model has been fine-tuned on numerous sets of codes and can generate high-quality code snippets from natural language descriptions. In addition, the model could effectively capture the semantics and syntax of a programming language. The authors concluded that the tool has the potential to improve efficiency and capture complex problems. The paper by Li et al. ([2022](#alpha)) proposed an approach called AlphaCode which is based on GTrXL architecture. The model is trained on a vast corpus of codes written by humans. It has claimed that it can generate high-quality code even for challenges it has never seen before; thus, it can compete with humans in various coding challenges. Furthermore, the model has been tested in multiple coding tasks and has outperformed other existing techniques; Lastly, the authors described the risks and benefits of other existing tools.

**2.4 Others**

Then some researchers tried other approaches that played a significant role in code generation, including a paper by Sutskever et al. ([2014](#suts)) that introduced the sequence-to-sequence (seq2seq) based on encoder-decoder architecture. The model uses backpropagation and loss functions to measure actual and predicted output differences. The results demonstrate that this model performs better than conventional statistical approaches. Similarly, Ling, Wang, et al. ([2016](#lpn)) propose a method for code generation that is developed using sequence-to-sequence architecture and a latent predictor network. The model predicts the next token in a program where previous tokens are provided as input. Then Hayati et al. ([2018](#hayati)) presented a model combining generative and retrieval-based techniques. Authors claimed that this combination could help to overcome the issue that the generative model faces with syntax and semantics of code. The proposed model is trained using a retrieval technique, and output is generated using the generative technique. The model has been tested on code completion and function name generation tasks. And the results of this approach are better than other retrieval-based and generative models. Further, Soliman, A.S et. Al. ([2022](#solim)) used neural machine translation (NMT) to develop a pre-trained language model for code generation that generates Python code based on natural language description. The author claimed that this model outperformed other state-of-the-art models when trained on CoNaLa and Django datasets.

**2.5 Review**

Many researchers have also written review papers focusing on a specific aspect of the study. We have picked up a few for our research, including a paper by Liao et al. [(2010)](#rev_uml) proposing a case study of an automatic code generation tool. It used a combination of templates and model-driven development techniques. The authors claimed that the code was almost equivalent to the manual code with some errors. Nevertheless, they concluded that this field has the potential to improve though there are many challenges. Similarly, Lee et al. ([2021](#rev_lee)) reviewed research on code generation using the semantic parsing technique. The authors have reviewed various rule-based, statistical-based, and neural network-based models used along with semantic parsing for code generation tasks. The paper also highlights the limitations of approaches, mentions the critical aspects on which more research is required, and provides a roadmap for future studies and research. The most recent review paper was by Dehaerne et al. ([2022](#rev_deh)), which reviewed 58 studies published between 2010 and 2021 based on code generation. The authors provided an overview of the challenges and covered various techniques used for code generation. They analyzed certain aspects and found that deep learning approaches have given the most promising results. Lastly, the authors discussed research challenges and what is required to improve the models.

**3. Models and Methodologies**

*3.1 Generative AI*

Generative AI is a type of AI that enables the generation of novel content by using existing content or data rather than simply analyzing or acting upon it. It does so by leveraging various machine learning and deep learning algorithms while abstracting the underlying patterns in input data. It is mainly used in the image, audio, video, model, avatar, data, and text generation. It is still in its infancy. However, its impact grows with improvement, technological evolution, and increased research. Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs) are the most used generative models.

From a code generation perspective, Generative AI has revolutionized it, from developing tools that rely on old retrieval techniques to creating self-sufficient models that generate code by testing a range of neural networks and models. Although these conventional tools are helpful for programmers, they need to improve their ability to aid developers with complex tasks such as creating clear code or executing algorithms.

Code generation using generative AI involves training a machine learning model on a large corpus of source code or programming language syntax. Then, the model learns to generate new code that follows the syntax and structure of the training data.

One such example or approach is the language model, OpenAI's GPT-3, which is trained on a large corpus of code in multiple programming languages. When given a prompt, such as a few lines of code or a natural language description of a program, the model can generate new code that provides the solution to the problem statement given to it,

Another approach is using a specific generative AI type called a code generator. These models are specifically designed for generating code and often use techniques such as neural machine translation and other deep and machine learning algorithms to generate code based on a high-level description or a natural language utterance of a program or task.

In both cases, the generated code may require additional intervention or tweaking by a human programmer to ensure correctness, security, and efficiency. Nonetheless, generative AI can significantly accelerate the process of creating software by automating the initial stages of code generation.

*3.2 Code Generation Overview*

The definition of code generation has changed since it came into existence. Earlier, it meant the backend work performed by the compiler to convert a source code into machine-understandable language. Nowadays, it is the process of automatically generating source code or machine code from a higher-level specification or model. As a result, code generation can simplify and enhance the software development process. In code generation, a higher-level specification or model is created that describes the desired behavior or functionality of the software system. If implemented well, code generation can help reduce errors and improve the quality of the resulting code by improving code consistency and maintainability. This has been under research for many years, and multiple models, techniques, and systems have been implemented to automate this challenging coding task. Numerous technologies have been applied to solve this problem, from unsupervised learning to generative AI. In the below section, some of the successful approaches have been discussed.

*3.3 Existing Models*

*3.3.1. UML Model*

UML (Unified Modeling Language) is a visual modeling language used to represent software system design. UML diagrams provide a standardized way to visualize and describe the architecture, behavior, and structure using a set of standardized notation and symbols to represent various system elements, such as classes, methods, attributes, and relationships.

Though UML is not designed for code-generation tasks, it can be used as a base for generating code using specialized code-generation tools. One common approach is to use UML diagrams as input and generate code in a specific programming language, such as Python or Java. To use UML models for code generation, the UML diagrams must be designed accurately using class diagrams, sequence diagrams, and other types of diagrams that capture the structure and behavior of the software system. Tools created using this approach usually contain a set of templates or rules that map UML constructs to coding language constructs. The primary use case of using UML for code generation is reducing development time and improving the quality of the resulting code by providing a clear and consistent design specification. Non-developers primarily use this tool to code applications or websites without actual utilization of programming. However, to create a tool based on UML, it is essential to ensure that the UML diagrams accurately represent the desired behavior of the software system.

*3.3.2. RNN*

RNN (Recurrent Neural Network) is a neural network architecture that processes inputs in an orderly manner and has been widely used for text generation problems. RNNs are based on feedback architecture which takes the output of the previous unit as input to the current element. This feedback loop helps to maintain the memory of prior inputs and generates text output based on the whole sequence of information. RNNs can create code snippets or entire programs based on a given input in code generation. This architecture can be extended in numerous ways to improve efficiency and performance. Some of the most popular extension includes LSTM, which diminishes the vanishing gradient problem; another is the simple version of LSTM, i.e., GRU.

RNN can be used for code generation simply by providing a dataset of programs or small code snippets as input to the model. Sequential architecture helps the network predict the next element based on previous features during training. Once the model is, trained, natural language description can be fed as input to the web to generate code as output. Though there is an advantage of using RNN over other models, that is, it can generate semantically and syntactically correct codes, the code quality may need to be improved. Thus, even after the benefits of this architecture, human intervention is required to review and check the quality of the code generated.

*3.3.3. Transformer*

The transformer model was introduced in a 2017 paper by Vaswani et al. [[24](#transf)], designed to focus on tasks such as machine translation, language modeling, and text classification. This model is based on the self-attention mechanism; unlike traditional sequence models such as Recurrent Neural Networks (RNNs), the Transformer model processes input sequences in parallel to determine the relationship between positions. It can be used for code generation by training it on a legion of codes and then training it for a specific task, such as here for automatic code generation.

Specifically for the code generation task, the input to the model would typically be a sequence of tokens representing the natural language input so the model can generate desired functionality. The output would be the generated code as a sequence of tokens. Sequence–to–sequence architecture is the most common approach for code generation tasks specifically designed for natural language processing. In this architecture, the input that could be a textual description is first processed by an encoder network, which results in a fixed-length vector representation of the information. This vector is then used as the initial state of a decoder network, which generates a sequential code token as output. Finally, various pairs of these input-output sequences train the model using a supervised learning technique. Once it is introduced thoroughly, we can generate code based on textual input. Still, of course, the code quality depends on the accuracy of input text and training data.

*3.3.4. Tree-based approach*

Abstract Syntax Trees (ASTs) are a data structure commonly used to represent the structure of programs syntactically in compilers and interpreters. It is a hierarchical tree-based structure leveraged its use for code generation. This is used for generating code by utilizing each node in the tree as a particular language construct, such as a statement or expression. The procedure involves traversing the tree in a depth-first order and generating code for each node based on its type. This has gained popularity as its hierarchical structure helps to create syntactically correct and semantically meaningful code. Transformations are applied to nodes of trees to develop a new code. A node represents a code snippet that could be anything, for instance, a print statement; a function invokes, or a variable definition. All in all, AST is a powerful technique to generate semantically correct code.

*3.4 The Novel Generative Approach- GANs*

*3.4.1 Overview*

Generative Adversarial Networks (GANs) are a type of model that consists of a generator and a discriminator. The generator produces synthetic or fake data by utilizing unsupervised or supervised algorithms. The discriminator then competes with the generator to identify whether a given sample is real or fake. The two models compete in this process, called a zero-sum game framework. Essentially, the generative and discriminative models are pitted against each other. The generator aims to produce actual data that can fool the discriminator into thinking it is accurate. Initially introduced by Goodfellow et al. [(2014)](#gan), GANs are now called vanilla GANs. They employ the min-max function to produce an input dataset that is then utilized by the generator. Since the introduction of the vanilla model, several researchers have explored its nuances and drawbacks, leading to the development of alternative versions such as Cycle GAN and Style GAN. These newer models combine neural networks and other techniques to generate different forms of inputs and generators. They have proven helpful in various applications, such as creating high-quality images via deep fake technology and discovering drugs in healthcare. One of the highlighted use cases discussed here is code generation.

Before we dive into the nitty-gritty of GANs application in code generation, it is imperative to have a solid grasp of the underlying mathematics.



Fig. 1 Generative Adversarial Network

Binary Cross Entropy is a loss function used in the classification model. It optimizes the model and is calculated using the opposing average of corrected probabilities. In GAN, both the generator and the discriminator use this training model. The general loss function can be written as

 )

where y is real data and is synthetic data

The calculating loss function for Discriminator:

Both natural and artificial data are fed into the discriminator model

For real input data, y is 1 and =D(x), the output of the discriminator for the actual data, and the loss function is:

#  (1)

 For synthetic input data produced by the generator, y = 0 and =D(G(Z)), the output of the discriminator for the input generated by the generator loss function for the same is

 (2)

Where,

Z is the input noise vector given to the generator

G(z) is the output of the generator

D(G(z)) is the output of the discriminator with generated data as input

The main aim of the generator is to identify or ‘discriminate’ between the actual and synthetic data. This can be achieved by maximizing the combined loss function for both kinds of data.

Hence, the final combined discriminator loss function for a single data point is:

 (3)

Calculating loss function for Generator:

The main objective of the generator is to trick the discriminator by labelling synthetic data as real. In other words, the generator’s goal is to minimize the above loss function.

Hence, the loss function for a generator can be written as follows:



As we are only feeding synthetic data, G(z) , therefore, D(x) becomes zero. So, the final loss function becomes:



 The above loss function is for a single data point. However, for the entire dataset, the loss function can be written in terms of expectation:



Where,

Pdata(x) is the entire real dataset

 Pz(z) is the synthetic dataset

which is the same equation described in Goodfellow et al. [(2014).](#gan)

*3.4.2 Code Generation Using GANs*

Code generation builds upon the text generation technique, making it crucial to delve into and comprehend how text generation is executed using GANs. Understanding the underlying mechanisms of text generation through GANs can provide valuable insights into generating code, allowing for more effective and efficient code-generation techniques.

Text generation is a computational process of creating sequences of words or sentences that are coherent and meaningful, using various algorithms and models. The process involves analyzing and understanding the context, patterns, and structures of natural or programming language and then using this information to generate new grammatically correct, semantically correct, and meaningful text. One popular approach is using artificial neural networks, particularly recurrent neural networks (RNNs) and their variants, such as long short-term memory (LSTM) and recurrent gated units (GRUs) or using rule-based systems, etc. Text generation is an active area of research in natural language processing (NLP). It has many potential applications in several fields, including machine translation, virtual assistants, content creation, and creative writing. GAN is gradually gaining attention and popularity for the same and is believed to have the potential to revolutionize the field of natural language processing.

GANs have been highly influential in producing more lifelike images and have yet to be widely utilized for text sequences. This is due to the discrete nature of the text, making it impractical to backpropagate the gradient from the discriminator to the generator, as is standard in GAN training. Backpropagation is computing the rise of the loss function concerning the neural network’s weights, which allows the weights to be updated during training. However, with text data, the gradients cannot be computed through the discrete sampling operation to select the following word in the generated sequence.

 In other words,

* the generator network takes a random noise vector as input and outputs a sequence of discrete tokens (words). During training, the generator is trained to produce text similar to the actual text data, while the discriminator is trained to differentiate between actual and generated text.
* To optimize the generator, the loss function needs to be computed, and the gradients concerning the weights of the generator network need to be calculated. Backpropagation is the standard algorithm for computing these gradients in deep learning models, but it relies on the differentiability of the operations used in the network.
* In the case of text generation, the discrete sampling operation used to select the following word in the generated sequence is not differentiable. This is because the generator’s output is a probability distribution over the vocabulary, and the selected expression is sampled stochastically from this distribution. Therefore, the gradients cannot be computed through the sampling operation, making using backpropagation to train the generator impossible.

This means that backpropagation cannot be used directly to train the generator network in GANs for text generation, as it requires differentiable operations.

In recent times, there have been multiple proposals to overcome this problem. These proposals can be categorized into below methods:

* Reinforcement Learning: This approach uses reinforcement learning (RL) to train the generator network. RL is a type of machine learning that uses a reward signal to guide an agent toward a desired behavior. In the case of text generation, the reward signal can be based on feedback from the discriminator network. Specifically, the generator is trained to produce text that fools the discriminator. It is rewarded based on how difficult it is for the discriminator to distinguish between the generated and actual text. In addition, this reward signal can be used to update the generator's weights, allowing it to produce better text.
* Gumbel-Softmax: Another approach is to use the Gumbel-Softmax trick, which is a differentiable approximation of the discrete sampling operation. The Gumbel-Softmax scheme involves adding Gumbel noise to the logits (generator output before the softmax function) and then taking a softmax over the resulting values. This produces a continuous approximation of the discrete probability distribution over the vocabulary, which can be used to compute gradients through the sampling operation. The Gumbel-Softmax trick has been shown to work well for training GANs for text generation.
* Latent Variable Models: Another approach is to use latent variable models, such as variational autoencoders (VAEs) or autoregressive models, to represent the text as continuous vectors. These models use an encoder network to map the text to a constant latent space and a decoder network to map the latent vectors back to the text. Differentiable operations can be used during training by representing the text as continuous vectors, allowing backpropagation to train the generator. These models have been shown to work well for generating high-quality text.
* Adversarial Autoencoders (AAEs): AAEs are a variant of GANs that use an autoencoder to map the text to a continuous latent space and a discriminator network to distinguish between the actual and generated text. The generator is trained to produce text similar to the real text in the latent area. In contrast, the discriminator is trained to differentiate between actual and generated text in the latent space. AAEs have been shown to work well for developing high-quality text.

Several papers have addressed this problem, including:

* Rosa et al. [(2021)](#survey) proposed that Gan based text generation’s primary challenges are that GANs were not designed to work with discrete data, needing additional tricks to subdue such a problem. Such data do not provide differentiable outputs, which inhibits gradients from being appropriately calculated and updated. Therefore, most works attempt to override such a problem by employing modified training objectives, Reinforcement Learning, or continuous-based outputs, such as Soft-Argmax or Gumbel-Softmax distributions.
* Fedus et al. [(2018)](#mask_gan) introduced a text generation model trained on in-filling (MaskGAN) address the challenge of propagating the gradient from the discriminator to the generator in GANs for text sequences, and utilize Reinforcement Learning (RL) to train the generator.
* Yu et al. [(2016)](#seqgan) propose a novel SeqGAN modelling model. The generator is modelled as a stochastic policy using reinforcement learning (RL). It circumvents the generator differentiation problem by directly performing gradient policy updates. The RL reward signal comes from the discriminator judged on a complete sequence and is sent back to the intermediate state-action steps using Monte Carlo search. With testing on synthetic data and real-world task, it showed even more s improvements over strong baselines.
* Another approach worth mentioning is text generation using images. Wang et al. [(2022)](#mal_code_fam) proposed a novel method for classifying malicious code families using convolutional neural networks (CNNs) and (GANs). The proposed method uses code visualization techniques to convert the raw binary code into images, fed into the CNN and GAN models for classification. Using a softmax layer, the CNN model is trained to classify the images into different malware families. The GAN model is trained to generate images similar to authentic malware images

Text generation can be leveraged for code generation by training a GAN on a dataset of code snippets/ samples to complete source codes of existing software and application in various programming languages. The generator network takes a random input vector and generates a code snippet as output. The discriminator network takes this code snippet as the input and predicts whether the generator network developed it or if it is accurate. The generator is trained to produce code snippets that are realistic enough to trick the discriminator network. In contrast, the discriminator network is trained to distinguish between the real and generated code snippets. Through this adversarial training process, the generator network learns to generate code similar to the code in the real dataset. Once the GAN is trained, it can be used for programming. This can be useful for tasks such as code completion, code synthesis, software development, etc.

Overall, text generation can be leveraged for code generation in GANs by training a GAN on a dataset of code snippets and using the generator network to generate new code snippets similar to those in the training dataset.

Some recent developments in GANs for code generation include:

* Code synthesis: GANs can produce code snippets, algorithms, or entire programs. This could be useful in automating programming tasks and reducing time and effort, thereby increasing efficiency.
* Bug detection: GANs can be trained to identify bugs and vulnerabilities in code and generate patches to fix them. This could help improve software quality and make the application or code safer and more secure.
* Code optimization: GANs can optimize code by generating more efficient algorithms or data structures. This could help improve the performance of software and reduce its resource usage.
* Program understanding: GANs can be used to learn the structure and syntax of a plethora of programming languages, which could help in tasks such as code translation, code completion, code recommendation, etc

However, it is worth noting that these applications are still in their early stages and have yet to be widely adopted in the industry. Nevertheless, recent research has demonstrated the novel, state-of-the-art ways GAN can be used for programming. Below are some examples of such cases:

* Hu et al. ([2022)](#obfuscation) proposed a new method for detecting malicious code that has been obfuscated, i.e., intentionally obscured to bypass detection by antivirus software. GAN is first trained on a large dataset of benign code samples to learn the distribution and pattern of standard code. Then GAN is used to generate synthetic normal code samples. These codes are designed to mimic the characteristics of standard code and can be used to pre-process benign and malicious code samples. The authors then use a second GAN, the Anti-obfuscation GAN (AGAN), to detect whether a given code sample has been obfuscated. Finally, the AGAN is trained to differentiate between normalizers and obfuscated code samples. The authors show that the AGAN can detect various obfuscation techniques such as function reordering, instruction substitution, and register to rename.
* Xu et al. [(2021)](#sccg) proposed a novel approach for detecting code clones, which are code fragments that are identical or almost identical. The proposed SCCD-GAN model consists of a GAN and an SCCD model. The former is responsible for generating realistic code fragments, while the latter model is responsible for identifying code clones based on their semantic similarity.
* Zhu et al. ([2019)](#gancoder) proposed a novel semantic programming-based automatic programming method GANCoder. The author suggested that through the game confrontation training of GAN’s generator and discriminator, GANCoder can effectively learn data distribution characteristics and improve the quality of codes. The experimental results showed that it could achieve comparable accuracy with the state-of-the-art code generation model, and the stability is better. However, the method proposed in this paper can only realize the conversion between single-line natural language description and single-line code.

**4. Discussion**

*4.1 GAN vs. other approaches*

*4.1.1 Conceptual level*

Though generative adversarial networks are yet to be the best option for code generation problems, in some aspects, GANs stand out better than other available approaches in the market. Starting with one of the most fundamental approaches, i.e., UML might be able to generate a well-structured code, but only when UML diagrams are created accurately, which requires much effort. Moreover, even after this much hard work, UML cannot capture system nuances. On the other hand, GANs can produce codes based on a problem statement or natural language utterance. Similarly, in the ASTs approach, much effort is needed to build a real tree, which is optional in the case of GANs. Regarding RNN, though, it is a much better approach than UML, but the code generated by the model depends on the training data; thus, the code generated would be very similar to the trained corpus of codes. Transformers are the most highlighted technique used for code generation. Even the most famous tools, including the new virtual hero of the world GPT, have used this technique for development. GANs architecture is analogous to the transformer model, with a significant difference in output generation technique. Transformer uses a retrieval-based method to generate output; however, GANs use a generative process. This retrieval-based approach limits it to generate code based on training data, and even that data must be massively big to develop a good code. On the other hand, GANs generative nature benefits generating a more diverse code. This capacity to create various could help develop more nuanced and carefully designed software. Currently, many researchers are combining these different approaches to reap their benefits. One of them is proposed in [[8](#hayati)].

*4.1.2 Tools*

In the 2000s, UML was tested and utilized extensively to automate code development. However, these tools or approaches proposed required much effort. Some of the most common and still available market tools include Visual Paradigm (launched in 2002) and starUML (established in 2006). These tools contain templates that process the actual code when a particular template is utilized on the backend. These tools seem suitable for non-developers or fast development, but developing a diagram requires a lot of effort. RNN gained much limelight in the 2010s and hence have been utilized for code generation problem. Some of the most successful tools include DeepCoder[[1](#deepcoder)], which leverages LSTM sequential architecture to generate code tokens. The common thing about these tools process involves retrieval-based techniques. Based on the user's natural language description, they search for the relevant code in their large corpus of data and provide the most appropriate code. The code they generate is usually very well-structured, but since the results depend on the trained data, this could limit the extent to which code can be generated. TranX [[29](#tranx)] utilized AST and other techniques to generate code snippets. It supports a few languages but needs help with long and complex inputs. TreeGen [[22](#sun)] is another AST-based tool that utilizes AST architecture to generate code automatically. It also supports code synthesis and completion. But this could be done only when a user creates an accurate AST. That is, this tool does not support natural language description. Moving on to the transformer model, some widely known tools have been developed based on this architecture, including GPT-3, Github’s copilot, and Tabnine. These have been the most used tools, but they have some limitations too. While GPT-3 generated code might not be syntactically correct, tabnine code is limited to certain programming concepts; thus, it cannot provide suggestions for complex problems. Similarly, the copilot also has limitations because it is trained on public code, so there could be a chance of security vulnerabilities. Also, many developers have claimed it is very time-consuming as it takes time to generate code, and even most of the time, a human cannot understand the code generated. Some more famous tools are developed using transformers, including MarianCG [[20](#solim)] and CodeT5 [[26](#t5)], but like others, these have various limitations. Now, considering the tools based on GAN, there needs to be more development. Only a few tools are available in the market. One of them includes GANCoder [[28](#gancoder)]. This tool could generate complex code that could not be syntactically correct.

Here, we discussed some of the available significant tools and can conclude that each tool either has a major limitation or generates only a few snippets. Thus, still, more research is required to automate this task.

*4.2 Challenges with GAN*

Though GANs have shown excellent results and have gained much attention in the research field, there are still several challenges and limitations that need to be addressed to make them a suitable technique for code generation tasks:

* Usually, code generated by GANs faces significant issues with syntax and semantics. As a result, it can cause a new code, which is prone to errors and is not meaningful. Many developers/researchers have been working to resolve this issue; the most relevant strategy was Monte Carlo Tree Search.
* Despite its generative nature, it requires human intervention to check various factors, including bugs, errors, semantics, etc.
* Similarly to other existing models, GANs also struggle with advanced or complex coding concepts, and as of now, it is suitable only for simple tasks.
* Though generated code is not only based on the training data, it’s still required to be trained on a massive amount of high-quality to adapt the concepts effectively.
* There are no security measures in practice yet to protect the GANs-generated code; thus, it could be attacked and lead to specific incidents.
* There is no way to limit the model to avoid specific snippets or concepts; thus, it may violate intellectual property laws or generate unethical content.

The solution to these challenges or limitations could lead to the success of GANs in code generation.

*4.3 Future*

As the area of GANs is under continuous research, its future looks promising despite significant challenges and could lead to automatic development in the software world. However, improvement in certain aspects of care is required, such as:

* Better code quality: GANs generated code needs to be much more optimal and efficient and could function properly without any errors and bugs to make the software development process more reliable.
* A Better understanding of natural language: Input provided in the form of natural language needs much accuracy. Even after it, the model needed help understanding the requirement correctly.
* Improvement of generative architecture: The generative architecture available today could generate clear and high-resolution images that have not existed before, but this needs many more improvements for code generation.

Overall, sudden improvements could lead to a bright future for GANs for code generation.

*4.4 Can GANs be able to pass Turing Test?*

The Turing test measures the ability of a machine to be efficient so that it is indistinguishable from a human. For a code generation task to pass the Turing test, a machine-generated needs not to be better than a human, but it needs to be impressive enough to be consonant with a human-written code. An elephant amount of research is still required to develop an efficient code generation model. Even after that, an enormous amount of study will be needed to understand the intent behind the development. In the future, it could develop the whole software independently, but it is doubtful that it could pass the Turing test.

1. **Conclusion**

This paper discussed past and recent developments in code generation, its different models, methodologies, and tools, and how GAN could be the future of auto code generation. The discussion started with an overview of the topic in focus and the different existing approaches tested and utilized for automatically generating codes. The most fundamental process, UML, and other widely used techniques, including neural networks, language modeling, and tree-based approach, have been part of the section. Further, in the discussion, we delved into Generative AI and Gan’s architecture, advantages, limitations, and challenges and how it is better than other approaches. We concluded that the transformer had the most success with the code generation problem, but it also has certain limitations and can only partially replace human-written codes. On the other hand, with GAN, significant developments have happened on the text generation side, which could also be leveraged for code generation. Finally, we concluded by discussing the future of the GAN and developments in which aspects could support the process of an auto-code generation that could replace and aid programming.

**Funding** No funding was received to assist with the preparation of this manuscript.

**Declarations**

**Competing Interest**

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

**Acknowledgments**

This research received no specific grant from funding agencies in the public, commercial, or not-for-profit sectors**.** This research is solely the authors’ work, and their employer organization has nothing to do with the analysis performed. Moreover, data and content used for research have no relevance to the organization.

**References**

1. Balog, Matej, et al. "Deepcoder: Learning to write programs." *arXiv preprint arXiv:1611.01989* (2016). DOI: https://doi.org/10.48550/arXiv.1611.01989
2. Beltramelli, Tony. "pix2code: Generating code from a graphical user interface screenshot." *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2018. DOI: https://doi.org/10.48550/arXiv.1705.07962
3. Chen, Mark, et al. "Evaluating large language models trained on code." *arXiv preprint arXiv:2107.03374* (2021). DOI: https://doi.org/10.48550/arXiv.2107.03374
4. E. Dehaerne, B. Dey, S. Halder, S. De Gendt and W. Meert, "Code Generation Using Machine Learning: A Systematic Review," in IEEE Access, vol. 10, pp. 82434-82455, 2022. **DOI:**[10.1109/ACCESS.2022.3196347](https://doi.org/10.1109/ACCESS.2022.3196347)
5. Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." *arXiv preprint arXiv:2002.08155* (2020). DOI: https://doi.org/10.48550/arXiv.2002.08155
6. Gustavo H. de Rosa, João P. Papa,”A survey on text generation using generative adversarial networks”,Pattern Recognition,Volume 119,2021, 108098, <https://doi.org/10.1016/j.patcog.2021.108098>.
7. H. Liao, J. Jiang, and Y. Zhang, "A Study of Automatic Code Generation," *2010 International Conference on Computational and Information Sciences*, Chengdu, China, 2010, pp. 689-691, doi: 10.1109/ICCIS.2010.171.
8. Hayati, Shirley Anugrah, et al. "Retrieval-based neural code generation." *arXiv preprint arXiv:1808.10025* (2018). DOI: https://doi.org/10.48550/arXiv.1808.10025
9. Ian J. Goodfellow, Jean Pouget-Abadie , Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengi,” Generative Adversarial Nets”, arXiv Preprint, <https://doi.org/10.48550/arXiv.1406.2661>
10. Jakimi, Abdeslam & Koutbi, Mohammed. (2009). An Object-Oriented Approach to UML Scenarios Engineering and Code Generation. International Journal of Computer Theory and Engineering. 1. 35-41. DOI: 10.7763/IJCTE.2009.V1.6.
11. K. Xu and Y. Liu, "SCCD-GAN: An Enhanced Semantic Code Clone Detection Model Using GAN," *2021 IEEE 4th International Conference on Electronics and Communication Engineering* (ICECE), Xi'an, China, 2021, pp. 16-22, <https://doi.org/10.1109/ICECE54449.2021.9674552>
12. Lantao Yu, Weinan Zhang∗, Jun Wang , Yong Yu,” SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient”,ArXiv Preprint, <https://doi.org/10.48550/arXiv.1609.0547>
13. Lee, Celine, Justin Gottschlich, and Dan Roth. "Toward code generation: A survey and lessons from semantic parsing." *arXiv preprint arXiv:2105.03317* (2021). DOI: https://doi.org/10.48550/arXiv.2105.03317
14. Li, Yujia, et al. "Competition-level code generation with alphacode." *Science* 378.6624 (2022): 1092-1097.
15. Ling, Wang, et al. "Latent predictor networks for code generation." *arXiv preprint arXiv:1603.06744* (2016). DOI: https://doi.org/10.48550/arXiv.1603.06744
16. Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017.  In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics. DOI: 10.18653/v1/P17-1105
17. Parvez, Md Rizwan, et al. "Retrieval augmented code generation and summarization." *arXiv preprint arXiv:2108.11601* (2021). DOI: https://doi.org/10.48550/arXiv.2108.11601
18. Yin, Pengcheng, and Graham Neubig. "A syntactic neural model for general-purpose code generation." *arXiv preprint arXiv:1704.01696* (2017).https://doi.org/10.48550/arXiv.1704.01696
19. R. Tiwang, T. Oladunni and W. Xu, "A Deep Learning Model for Source Code Generation," 2019 SoutheastCon, Huntsville, AL, USA, 2019, pp. 1-7, doi: 10.1109/SoutheastCon42311.2019.9020360.
20. Soliman, A.S., Hadhoud, M.M. & Shaheen, S.I. MarianCG: a code generation transformer model inspired by machine translation. *J. Eng. Appl. Sci.* **69**, 104 (2022). <https://doi.org/10.1186/s44147-022-00159-4>
21. Stuermer, M. Conrad, H. Doerr and P. Pepper, "Systematic Testing of Model-Based Code Generators," in IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 622-634, Sept. 2007, doi: 10.1109/TSE.2007.70708.
22. Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., & Zhang, L. (2020). TreeGen: A Tree-Based Transformer Architecture for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*(05), 8984-8991. <https://doi.org/10.1609/aaai.v34i05.6430>
23. Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." *Advances in neural information processing systems* 27 (2014). DOI: https://doi.org/10.48550/arXiv.1409.3215
24. Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017). DOI: https://doi.org/10.48550/arXiv.1706.03762
25. W. Hu, J. Cheng, X. Chong, R. Zhang, B. Lin and A. Xia, "A GAN-Based Anti-obfuscation Detection Method for Malicious Code," *2022 3rd International Conference on Pattern Recognition and Machine Learning (PRML)*, Chengdu, China, 2022, pp. 484-488, <https://doi.org/10.1109/PRML56267.2022.9882255>
26. Wang, Yue, et al. "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation." *arXiv preprint arXiv:2109.00859* (2021). https://doi.org/10.48550/arXiv.2109.00859
27. William Fedus, Ian Goodfellow and Andrew M. Dai, ” MASKGAN: BETTER TEXT GENERATION VIA FILLING IN THE \_\_\_\_\_”,2018,arXiv Preprint, <https://doi.org/10.48550/arXiv.1801.07736>.
28. Yabing Zhu, Yanfeng Zhang, Huili Yang, and Fangjing Wang,”GANCoder: An Automatic Natural Language-to-Programming Language Translation Approach based on GAN”,2019, arXiv Preprint, [https://doi.org/10.48550/arXiv.1912.00609](%20https%3A//doi.org/10.48550/arXiv.1912.00609)
29. Yin, Pengcheng, and Graham Neubig. "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation." *arXiv preprint arXiv:1810.02720* (2018). DOI: https://doi.org/10.48550/arXiv.1810.02720
30. Ziyue Wang, Weizheng Wang, Yaoqi Yang, Zhaoyang Han, Dequan Xu, Chunhua Su, “CNN- and GAN-based classification of malicious code families: A code visualization approach”, International Journal of Intelligent SystemsVolume 37, Issue 12,2022, <https://doi.org/10.1002/int.23094>