Human Interaction and Emerging Technologies (IHIET 2023), Vol. 70, 2023, 149–162
https://doi.org/10.54941/ahfe1002939

Human Like Programming Using SPADE BDI Agents and the GPT-3-Based Transformer

Ratovondrahona Alain Josué, Rakotozanany Hanitriniaina Marielle, Mahatody Thomas, and Manantsoa Victor

Doctoral School Modeling-Computer Science, University of Fianarantsoa, Madagascar

ABSTRACT

Programming an application requires multiple people with skills and experience in that field. It will also take a lot of time with multiple steps before achieving the final result of an application. Today, developers are assisted by various tools, software, or applications based on Artificial Intelligence (AI) such as OpenAI's ChatGPT. These AI that automatically generates source code helps developers to develop applications much faster. However, although code generators are numerous and very helpful, we are not yet at the stage where we can generate a fully functional application, but just generate pieces of source code. And we don't know yet how to understand textual descriptions of Software Requirements to generate an application directly. Or where to find data to train an AI capable of generating a functional application from textual descriptions. Therefore, we created a new architecture composed of virtual intelligent agents called SPADE BDI to create virtual developers. The virtual intelligent agents were responsible for keyword extraction, Software Requirements synthesis, and source file creation. Then we used a transformer based on pre-trained GPT-3 for source code generation. This transformer is orchestrated by a virtual intelligent agent. To solve the problem of training data, we collected and created a new dataset called WSBL. The data came from several projects developed with the Laravel Framework over 4 years. The result allowed us to have a functional application directly from a textual description. Each intelligent virtual agent played a role like a developer by analyzing textual of Software Requirements and then generating source code. With a 15% reduction in time to develop an application compared to brute development. Our new architecture allows for processing textual descriptions (Software Requirements) step by step using intelligent virtual agents named SPADE BDI and source code generation is done by a transformer based on pre-trained GPT-3 to have a directly functional application.

Keywords: Software requirements, Source code generation, SPADE BDI, Transformer, WSBL

INTRODUCTION

The software industry is constantly evolving, and the demand for efficient and fast solutions is on the rise. The generation of source code from textual descriptions is a promising approach to meet these needs, allowing for the automated generation of code from specifications described in a clear and concise manner.

Specifically, our laboratory and research team GLORE focuses its efforts on helping developers in their tasks. For example, the work of (Dimbisoa, 2020) generates graphical interfaces using the MDA approach. The research of (Andrianjaka et al., 2019) and (Razafindramintsa et al., 2016)focused on generating code from UoD or Universes of Discourse using the Elaborate Lexicon Extended Language or ELEL and REstructuring Lexicon Extended Language or RELEL to extract useful keywords for code and database generation. For (Razafimahatratra et al., 2017), UML is used for emergent design to ensure application maintenance and quality. Finally, the work of (Tarehy et al., 2017) allows for the reuse of software components to optimize development time. All of this is done in order to meet the increasingly demanding requirements of software development, which includes saving time and delivering quality products.

However, there are also AI-based approaches proposed for code generation. With the advancement of deep learning, the problem of code generation can be treated as a textual translation problem. Source codes or programming languages are considered as a natural language that can be translated. Therefore, models that have been successful in textual translation, such as Seq2Seq (Weiss et al., 2017), have also been successful in code generation. And code generation from text is also the reverse of generating comments from code. Therefore, models (Hu et al., 2018b, 2018a; Kuang et al., 2022) have also been applicable to source code generation. With deep learning-based models, inputs are not just text but can also be in the form of images (Beltramelli, 2018) such as mockups. This shows the interest in works on code generation.

Despite the interests and results of research already conducted, AI is not always capable of directly generating an application (Le et al., 2020). The problem comes from the fact that it is first necessary to understand the software requirements in textual description form. Then, a lot of datasets are required for training (Yang et al., 2021). Finally, it is also necessary to consider the state of the art in source code generation (Le et al., 2020).

In this work, we collected data that we evaluated with existing deep learning approach. Compared to existing datasets, our particularity was on the form of software requirements that are the demands or needs of the non-developer but a real description by an end-user. And we also developed a new architecture to directly generate an application from user requirements or textual descriptions. We used intelligent virtual agents. The advantage of our work is that we do not simply rely on deep learning to generate code, but we also used a number of technologies to address the issues. We used SPADE BDI (Rafalimanana et al., 2018), which is interoperable with our source code generator.

We will detail this work with the following plan. The next section introduces Related Work. Then a section to explain the Methodology. Followed by Experiments conducted to confirm the hypotheses. Then, we will have a section for the Discussion of the results. Finally, the Conclusion section.

RELATED WORK

To develop or generate an application or source code, we have several existing methods that continue to be improved. Programming languages, DSL, MDA coupled with UML, n-gram models, probabilistic grammars, and solutions proposed by artificial intelligence, especially with deep learning.

Developers are much more comfortable with programming languages. They can read or modify the source codes they have written. And we can develop any software in various domains with a programming language of any level of complexity (Liu et al., 2020). However, programming languages are limited to the use of developers. The volume of work and complexity compared to the projects to be developed can be significant in terms of time.

And work on code generation for versatile programming languages has emerged. First, there is LPN or Latent Predictor Network, which can generate Python or Java code (Ling et al., 2016). Then, the SNM or Syntactic Neural Model (Yin and Neubig, 2018), which is generalized by the TRANX approach (Yin and Neubig, 2018) for various target programming languages, the latter using multiple deep learning decoders.

The DSL (Liang et al., 2013) is another approach for source code generation. It is useful for describing the results of generated source code. To reduce the complexity of code generation, researchers try to limit the complexity of the programming language. Hence, the interest in using a DSL. And sometimes the results are very accurate. Unfortunately, the DSL is specific to a given domain, and it is really difficult to apply it to other domains (Liu et al., 2020).

The MDA approach also promises good results for code generation. The works of (Andrianjaka et al., 2019) and (Razafindramintsa et al., 2016) provide transformation rules to convert requirements into source code. They used ReLEL to instantiate and capture the conceptual aspects of future software, and they used ATL to implement the MTM transformation phase and Acceleo for the M2T phase. The problem with this work is that the rules must be manually modified when the conditions have changed, or the constraints are no longer valid.

The n-gram model (Yang et al., 2022) and probabilistic grammars (Yang et al., 2022) are also exploitable approaches. However, they are ancestors to deep learning-based models today. Code generation can be treated as a translation problem or the inverse of comment generation from source code. Recent models that have evolved and contributed to this field include Seq2seq, Deepcoder, and the transformer models such as GPT-3.

The Seq2seq (Weiss et al., 2017) architecture is a deep recurrent neural network encoder-decoder that translates speech from one language into text in another. This solution has been adapted to generate code from text. Text and code are aligned to capture the correspondence and enable subsequent generation. The Seq2seq model also introduces the attention mechanism (Vaswani et al., 2017), which is used by transformers.

The Deepcoder (Balog et al., 2016) model proposed an encoder that ensures the comparison of an input-output example from a set M to a real latent

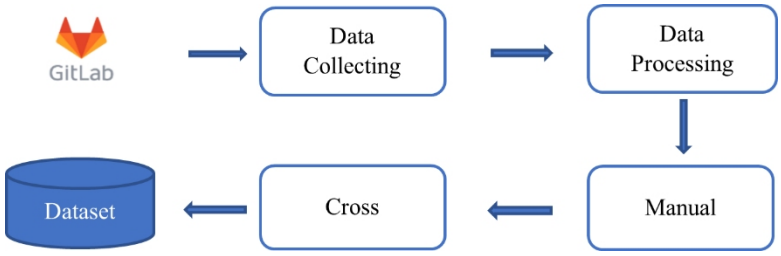

Figure 1: Creating the dataset.

vector. Its decoder also compares the predicted input-output of a set M to the real value. The attention mechanism is also used in this model.

Looking at the Seq2seq and Deepcoder models, transformers play a significant role in code generation. The transformer model avoids the recurrence that slows down processing and can directly encode a large sequence but not by character. Currently, the research conducted by OpenAI with the GPT-3 model (Brown et al., 2020), which is one of the transformer variants at the forefront of the state of the art in text and code generation, has exploded. The peculiarity of GPT-3 is that it has been trained with 175 billion parameters and it is an autoregressive language model.

The models described in this section are promising approaches for generating code. However, we have not yet had a model that will directly generate a functional application. Only raw development through a programming language can lead to an application. Deep learning models are effective in generating code snippets from keyword-driven queries by a developer. So in order to achieve our goal, in the next section we will detail our approach to directly generate an application.

METHODOLOGY

In this section, we will discuss data collection, data processing, applying state-of-the-art text-to-code generation techniques to our dataset, and the architecture of our model for automatic application generation.

Data Collecting and Processing

Figure 1 shows the process for developing the new dataset. First, we collect source code from the GitLab repository. Second, we remove duplicate data and associate it with textual descriptions (software requirements) from JIRA in CSV format. Third, we conduct a manual review of the source code. Finally, we apply cross-validation to exclude source code that contains bugs. The details of the creation process are presented in the following section.

Data Collecting

The source code comes from a GitLab repository. This repository contains projects from a company specialized in outsourcing IT development services. The projects are varied but coded in the same PHP programming language with Laravel 6 Framework and others that will not be treated but detailed

Table 1. Dataset's developpers.

IT professions	Numbers	Average Years of Experience
Senior Developer	7	8 years
Junior Developer	13	2 years
Senior QA Tester	4	6 years

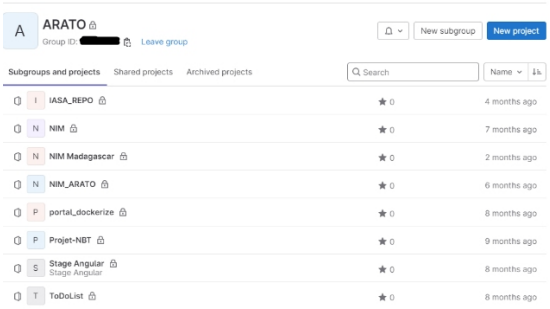

Figure 2: Sample of repository code.

in Table 2. To automate the pull request of source code from the GitLab repository in Figure 2, we set up a simple Python program on a VPS server to transfer the code automatically from the repository to our VPS server. Regarding the projects and codes, we provide details on the teams that developed the projects in Table 1. Project details are confidential, but information such as the number of software requirements and the length's code is detailed in Table 2, which results directly from the dataset processing. The technologies or languages are multiple, but we simply took the technologies with more data.

The reasons for using this dataset are:

- The requirements are actual requests from end-user not a developper.
- The software requirements are user stories, not just feature statements.
- Most existing datasets are primarily focused on Python and Java.
- Developers have already introduced unit tests to avoid non-functioning code and redundancies.

Data Processing

For the software requirements, we collected them from the backlogs in JIRA in CSV format, and then processed them with a Python program to remove unused columns during training. In Figure 3, we show an excerpt of the data from the JIRA backlog. Thus, we translated the software requirements into English using our method (Ratovondrahona et al., 2018) to facilitate comparisons and usage with the models that we will discuss in the next section.

We used the SimHash algorithm by (Manku et al., 2007) to compare redundant software requirements. The algorithm transforms texts into a fixed size hashed format. This method allows for the direct detection of redundant texts. For example, "list of data" is repeated several times but differs

	A	B	C	D	E	F	G
1	JIRA id	Title	User Story	Domaine	Priority	Planified in sprint	Location id
2	ARMZK - 1	authentification web page	##########	##########	##########	##########	1
3	ARMZK - 2	verification identifiant authentification	##########	##########	##########	##########	2
4	ARMZK - 3	deconnection	##########	##########	##########	##########	3
5	ARMZK - 4	affichage page avec de donnee en retour	##########	##########	##########	##########	4
6	ARMZK - 5	enregistrement des donnees pieces	##########	##########	##########	##########	5
7	ARMZK - 6	mise a jour piece	##########	##########	##########	##########	6
8	ARMZK - 7	enregistrement des donnees pneu	##########	##########	##########	##########	7
9	ARMZK - 8	mise a jour cheque	##########	##########	##########	##########	8
10	ARMZK - 9	suppression attachement	##########	##########	##########	##########	9
11	ARMZK - 10	liste des donnees autres articles	##########	##########	##########	##########	10
12	ARMZK - 11	liste des donnees approvisionnement	##########	##########	##########	##########	11
13	ARMZK - 12	liste des donnees pieces	##########	##########	##########	##########	12
14	ARMZK - 13	liste des donnees pneu	##########	##########	##########	##########	13
15	ARMZK - 14	selection d'une ligne de donnees piece	##########	##########	##########	##########	14
16	ARMZK - 15	enregistrement des donnees fournisseur	##########	##########	##########	##########	15

Figure 3: Backlog of software requirements.

Table 2. The dataset.

Languages	Average size of functionalities per file (tokens)	The average length of software requirements (tokens)
PHP	167/2038 Function/file	9.7
JavaScript	218/4894 Function/file	9.7
Python	54/1365 Function/file	9.7

for each implementation code. That's why we did manual checks after detecting redundancies to avoid deleting data intentionally. So, sr_1 and sr_2 will be duplicated if implementation codes have different functionalities.

We used the edit distance algorithm (Levenshtein, 1966) to compare two code fragments for the source code. If the distance is large, the code fragments are not similar, and if not, they are redundant. However, we manually checked the codes after automatic detection to avoid accidentally deleting data. Then, we used canonical machine learning to ensure an equal distribution of source code and software requirements (Krawczyk, 2016). We balanced our dataset with a 1:1 ratio (software requirement: code). Afterward, we grouped the codes that corresponded to a user story. Here, the ratio was 1:n (software requirement: n*code).

Manual Review

After creating the pairs (software requirement: code) or (software requirement: n * code), we manually rechecked the correspondence between the software requirement and the corresponding implementation codes. Then, we also verified the implementation codes if they work as provided. The next step shows how we automated the code testing.

Cross Validation by Software Testing

We used test cases, and the tests we formulated were done in two steps. First, we set up a template to specify the input parameters (data types and value ranges) and generate test cases automatically using fuzz testing. Finally, we run the tests automatically. We followed the same process in the work (Liu et al., 2020) using EvoSuite accompanied by a manually created template to explain the specific inputs and automatically generate test cases based on the template.

Dataset and Quality

To evaluate the quality of our dataset, we compared it with two datasets that are among the state of the art: CoNala (Yin et al., 2018) and ReCa (Liu et al., 2020). We observed that our software requirements are not only feature statements but also include user stories. Additionally, the codes are manually verified and validated to ensure functional codes. JavaScript and Python codes are not used in the dataset for confidentiality reasons.

For the evaluations, we have established three metrics: The first metric is the use of BLEU (Papineni et al., 2002). This metric was originally initiated for textual translation. For code generation, BLEU scores are calculated by comparing the generated code to a set of reference code. The score value ranges between 0 and 1. It can be considered that for a generated code c, if the similarity is high or identical compared to the reference codes, then the quality of the generated code is good. Otherwise, if the similarity is low, the quality is poor.

The second metric is the number of errors. Here, we did not include the number of warnings in the code since our goal is for the program to function properly.

The third metric is the percentage of test cases passed by the generated program. We automatically generate a test case for each generated code.

Code Generation

In order to generate code from a software requirement, we need to make the machine understand the request. We will detail this process in this section.

Understanding of Textual Description

When we want to generate code from text, the machine needs to understand what is being asked of it. Therefore, the machine must understand the context in the textual description and extract the essential words.

Word Embedding. We have already conducted previous work (Ratovondrahona et al., 2018) on context-based translation. Our objective was to capture the meaning of the text, not just the syntactic aspect. Word embedding works as follows. To capture the semantics between two words, we calculate the similarity score or cosine similarity between −1 and 1, where a higher score means more similarity and vice versa. Considering that the words are mapped into a vector space where each word has its own vector coordinate.

$$P(W_0|W_i) = \frac{e^{(V_{wi} \cdot V_{W_0}^T)}}{\sum_{j=1}^{v} e^{(V_{wi} \cdot V_{W_j}^T)}}$$

Then, to each word w it is assigned a vector representation v, and the probability that w_o is in the context of w_i is defined as the softmax of their vector product. In our project, we pre-trained directly with the word embedding of OpenAI (Neelakantan et al., 2022) to group keywords around a software requirement. For example, around the word 'truck' we have 'parts', 'tires', 'other items', and 'supply'. This will allow our architecture later to generate

requirements composed of these words around the main word. Then we pass the relay to our code generator.

Code Generation From Text

According to the RELATED WORK section, we will focus on Transformer-based models. First, we will see how the Transformer works in general.

Transformer. The Transformer model has optimized the Recurrent Neural Network (RNN) (Weiss et al., 2017), which was slow in processing due to encoding and processing word-by-word of a sentence and could also forget learned words if the sequence is long. There are three important points to consider: position encoding, attention, and self-attention. Position encoding refers to the idea of taking all the words in an input sequence of a sentence and adding to each word a number corresponding to its order. Attention is a mechanism that allows a text model to examine each word in the original sentence when making a decision on how to translate the words in the output sentence. Self-attention is done to build a model that understands the underlying meaning and language patterns. The sequence can be considered to capture this attention:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{\mathrm{dk}}}\right)V$$

The rows of Q are referred to as "queries," those of K "keys," and finally those of V "values." Here, $Q \in \mathbb{R}^{m*d_k}$, $K \in \mathbb{R}^{n*d_k}$ and $V \in \mathbb{R}^{n*d_v}$. Note that for the algebra to work out, the number of keys and values n must be equal, but the number of queries m can vary. Likewise, the dimensionality of the keys and queries must match, but that of the values can vary.

In our case, we will use GPT-3 (Brown et al., 2020) which is one of the Transformers with very good performance in terms of code generation.

General Architecture of the Code Generator

In this section, we will explain the general architecture of our solution for code generation from textual descriptione.

Figure 4 shows the overall architecture of our code generator. We have previously worked on a Multi-Agent System to create an intelligent webservice (Rafalimanana et al., 2018). However, we have reused the same foundations to orchestrate the code generation modules. The difference between the previous work and the present one is that we used Java, Netty, Grizzly, and Jason RS-WS in the former, while in this work, we used SPADE BDI, XMPP, and Python. This choice was made to facilitate the integration of the code written in Python for the word embedding and the transformer.

SPADE has the behavior of executing tasks in parallel (Thread) with the protocol provided by JABBER which allows transferring messages for each event. We used this channel to receive and send messages for each agent. The four agents have specific roles. The first is responsible for extracting words around the inputs. Then the second is responsible for searching for the software requirements that contain the words from the first agent. The third

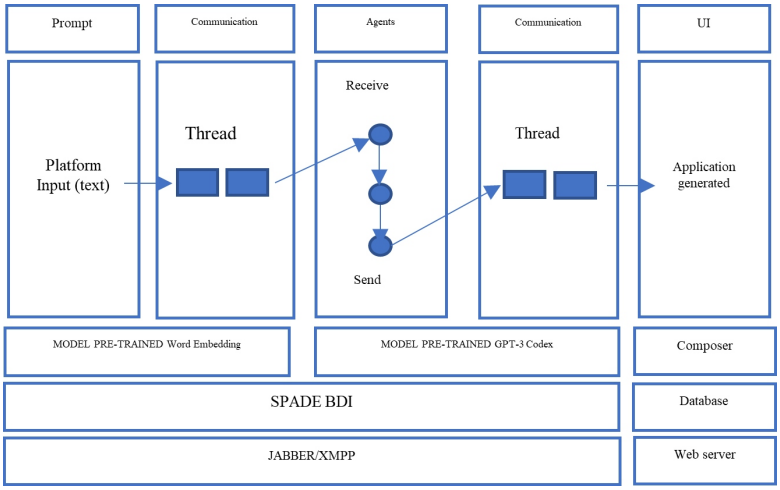

Figure 4: General architecture of WSBL.

is responsible for generating code for the software requirements selected by the second agent. And the fourth agent creates the root of the project and copies the code into each respective file and places it directly in the web server directory.

EXPERIMENTS, RESULTS AND DISCUSSION

In order to validate our hypothesis, we will detail in this section the experiments we conducted to validate our model, as mentioned in the RELATED WORK and METHODOLOGY sections. Compared to other deep learning-based code generators, transformers have shown the best performance in various fields, such as text and code generation, among others. However, code generators are still limited to generating a few functionalities, and not a fully usable application.

Validations Questions

The evaluation examines the following questions:

- **Q1** : "What can be learned from the word extraction? Are there any other approaches?
- **Q2** : In comparison to the state of the art, how does our code generator perform?
- **Q3** : Is the generated application usable?
- **Q4** : Does it take less time for development?

We chose Seq2Seq (Weiss et al., 2017) and TRANX (Yin and Neubig, 2018) because of their publicly available implementation, and for GPT-3 (Brown et al., 2020), we used the paid mode from OpenAI. However, the parameters for GPT-3 () have other criteria, so we separated the parameters for the evaluation of our model into two tables. For the Seq2seq model (Weiss et al., 2017) and TRANX (Yin and Neubig, 2018), we used the

Table 3. Parameters for evaluated approaches.

Models \ Parameters	Embedding Size	Hidden Size	Epoch	Batch Size	Decoder Dropout	Learning Rate
Seq2seq	200	N/A	120	20	0.4	0.01
TRANX	128	256	100	10	0.3	0.001

Table 4. Parameter for GPT-3 approach.

Models \ Parameters	Model	Max Tokens	Temperature	Batch Size	Epochs	Learning Rate
GPT-3	Davinci-Codex	1024	0.7	4	3	5e-5

Table 5. Parameter for GPT-2 word embedding.

Models \ Parameters	num layers	d model	num_ heads	batch size	epochs	learning rate	max_ length	d _ff
GPT-2	12	768	12	32	3	5e-5	128	3072

hyperparameters from (Liu et al., 2020). In Table 3, we have the details of the hyperparameters for the evaluation of our dataset. In Table 4, we have the hyperparameters for GPT-3 for the evaluation and training of our dataset.

Q1. This is an evaluation on capturing semantics from input sequences. And finally, Table 5 shows the hyperparameters for evaluation using the pre-trained GPT-2 word embedding. It aims to provide the essential words that group the functionalities to be generated. We opted directly for the case of word embedding with the recommended parameters from OpenAI during the training of our dataset. As a dataset, we simply took the software requirements and used "Embed" dataset to creates a single column in the original dataset containing vectors (lists) of N components.

Q2. This is about the performance of the approaches (Weiss et al., 2017), (Yin and Neubig, 2018), and (Brown et al., 2020) in relation to our new dataset. After the training phases, we carried out some tests on the generation of CRUD. Regarding the metrics we discussed in the METHODOLOGY section, we have a drop in BLEU score between 0.086 and 0.096 for the models (Weiss et al., 2017) and (Yin and Neubig, 2018). This indicates that this drop in score shows that the generated code is far from the reference code. On the other hand, the GPT-3 model has a better performance of 0.879. This is due to the attention mechanisms that the transformer brings to capture a large input set and the aligned codes at the same time. For both models (Weiss et al., 2017) and (Yin and Neubig, 2018), of course, our dataset is entirely new to them. Syntaxes and code styles can play a role in the results. The only problem with GPT-3 is with the generated codes for the "Modify" functionality. This still presents a bug in all the tests performed, as shown in Table 7. And Table 6 shows the evaluation on the new dataset.

Q3. We enable to say if the generated code is usable for a developer. Even in case of bugs, is the correction time-consuming or not. As already mentioned in Table 6, only the CRUD tested with GPT-3 works. The codes are directly

Table 6. Evaluation results on New dataset.

Approaches	BLEU on new Dataset	BLEU on Django	Syntactically Correct Programs	Executable Programs	Functionally Correct Programs
Seq2Seq	0.086	0.673	44.7%	6.0%	0%
TRANX	0.096	0.856	81.7%	9.0%	0%
GPT-3	0.879	0.952	86.8%	69.7%	54%
Average	0.353	0.827	71.06%	28.23%	18%

Table 7. Irrelevant tokens.

Languages	Average errors per generated code (tokens)
Seq2Seq	87.5%
TRANX	81.5%
GPT-3	18.9%
Average	62.63%

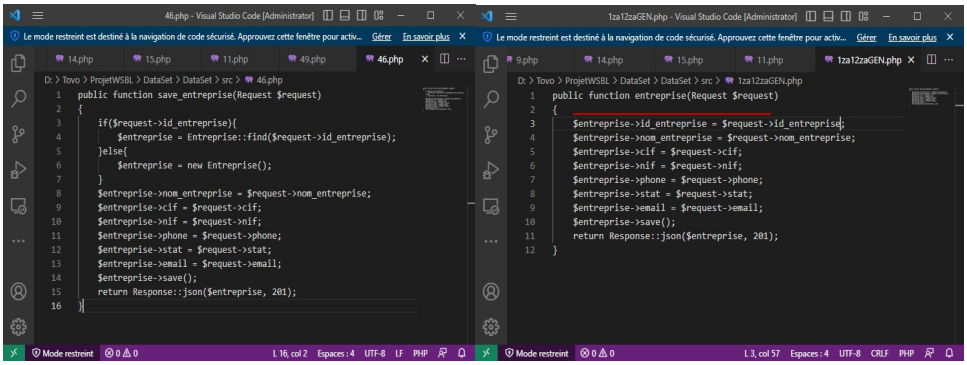

Figure 5: Comparison of a generated program with a reference.

usable on average 54%. Figure 5 shows a comparison between a reference code and one generated by GPT-3.

We can see that the function name is not really correct but not entirely wrong either. Then, there is a difference in the retrieval of the identifier. This is normal, since the majority of the additions in our dataset do not have this retrieval condition, but rather a direct addition of all attributes. We do not capture this notion of relationship between class by foreign keys.

Q4. This study shows the interest of our objective to directly generate an application. We have already shown the results for word embedding and the code generator based on pre-trained GPT-3, as shown in Figure 5. Tables 6 and 7 show the performances of our models. To illustrate our result in this section, we have set up a prompt for generating a simple program that creates an article with a number, title, content, and date. This is part of a blog.

Figure 6 shows that our agent responsible for creating the root of the project has completed its work. The immediate observation we made is that the

Figure 6: Structure of the generated project.

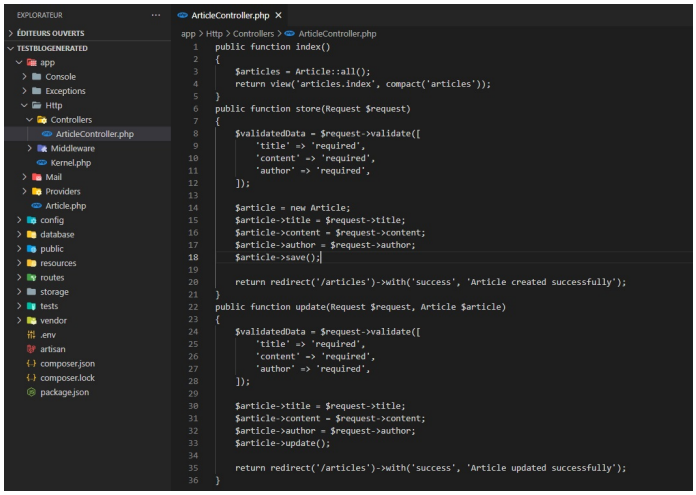

Figure 7: Generated code.

code for deletion is not given. Additionally, codes like controllers do not work because the necessary program headers need to be imported, such as models, class Facades, and class Requests. The configuration in the ".env" file is also not configured. The configuration of the routes file is not done either. The time it took us to code the entire application was 50 minutes. So, we asked two junior developers from the company to redo the same project. The first developer completed the task in less than an hour and 58 minutes, and the second one in 01 hour and 00 minutes.

According to this comparison between the generated program and the two developers, our result showed an approximate gain of 15% to 20% in development time.

CONCLUSION

Transformer models are very effective in generating code from text. By evaluating existing models compared to deep learning approaches with the new dataset, this evaluation showed good performance of the GPT-3 model with the Davinci model. The BLEU score shows high model performance at 0.879.

The generated code has only about 30% errors. We also used OpenAI's pretrained word embedding to extract words around the main word. From these models, we created an architecture for generating code from text. This architecture is orchestrated by SPADE BDI agents by extracting context words from the prompt and generating the corresponding code. Before generating the root of the project and placing each code in a file in the appropriate location. Our architecture saves time compared to development, approximately 15 to 20%. Despite this performance, the architecture is not always able to generate a complete functional application. The problem lies in the softwre requirements that contain both client demands and a functional problem statement made by a developer. The descriptions of the requirements are not explicitly stated in the narration. For the next steps of our research, we will delve into generating code from a mock-up because the attributes are complete and the features can be generated by modules. Also, an image is much more explicit than text.

REFERENCES

Andrianjaka, R.M., Luc, R.J., Mahatody, T., Ilie, S., Raft, R.N., 2019. Restructuring extended Lexical elaborate language, in: 2019 23rd International Conference on System Theory, Control and Computing (ICSTCC). IEEE, pp. 266–272.

Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D., 2016. Deepcoder: Learning to write programs. ArXiv Prepr. ArXiv161101989.

Beltramelli, T., 2018. pix2code: Generating code from a graphical user interface screenshot, in: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems. pp. 1–6.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., 2020. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 33, 1877–1901.

Dimbisoa, W.G., 2020. Génération automatique des IHM à partir de modèles conceptuels selon l'approche MDE (PhD Thesis). Université de Fianarantsoa (Madagascar).

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018a. Deep code comment generation, in: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, pp. 200–20010.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., Jin, Z., 2018b. Summarizing source code with transferred api knowledge.

Krawczyk, B., 2016. Learning from imbalanced data: open challenges and future directions. Prog. Artif. Intell. 5, 221–232.

Kuang, L., Zhou, C., Yang, X., 2022. Code comment generation based on graph neural network enhanced transformer model for code understanding in open-source software ecosystems. Autom. Softw. Eng. 29, 43. https://doi.org/10.1007/s10515-022-00341-1

Le, T.H., Chen, H., Babar, M.A., 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. ACM Comput. Surv. CSUR 53, 1–38.

Levenshtein, V.I., 1966. Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet Physics Doklady. Soviet Union, pp. 707–710.

Liang, P., Jordan, M.I., Klein, D., 2013. Learning dependency-based compositional semantics. Comput. Linguist. 39, 389–446.

Ling, W., Grefenstette, E., Hermann, K.M., Kočiskỳ, T., Senior, A., Wang, F., Blunsom, P., 2016. Latent predictor networks for code generation. ArXiv Prepr. ArXiv160306744.

Liu, H., Shen, M., Zhu, J., Niu, N., Li, G., Zhang, L., 2020. Deep Learning Based Program Generation from Requirements Text: Are We There Yet? IEEE Trans. Softw. Eng. 1–1. https://doi.org/10.1109/TSE.2020.3018481

Manku, G.S., Jain, A., Das Sarma, A., 2007. Detecting near-duplicates for web crawling, in: Proceedings of the 16th International Conference on World Wide Web. pp. 141–150.

Neelakantan, A., Xu, T., Puri, R., Radford, A., Han, J.M., Tworek, J., Yuan, Q., Tezak, N., Kim, J.W., Hallacy, C., 2022. Text and code embeddings by contrastive pre-training. ArXiv Prepr. ArXiv220110005.

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. pp. 311–318.

Rafalimanana, H.F., Razafindramintsa, J.L., Ratovondrahona, A.J., Mahatody, T., Manantsoa, V., 2018. Publish a Jason agent BDI capacity as web service REST and SOAP, in: International Conference on the Sciences of Electronics, Technologies of Information and Telecommunications. Springer, pp. 163–171.

Ratovondrahona, A.J., Razafindramintsa, J.L., Rafalimanana, H.F., Mahatody, T., Manantsoa, V., 2018. Word Embedding : Machine Translation according to the context. SETIT 18.

Razafimahatratra, H., Mahatody, T., Razafimandimby, J.P., Simionescu, S.M., 2017. Automatic detection of coupling type in the UML sequence diagram, in: 2017 21st International Conference on System Theory, Control and Computing (ICSTCC). IEEE, pp. 635–640.

Razafindramintsa, J.L., Mahatody, T., Razafimandimby, J.P., 2016. Elaborate Lexicon Extended Language with a lot of conceptual information. ArXiv Prepr. ArXiv160101517.

Tarehy, B.E., Mahatody, T., Razafindramintsa, J.L., Razafimandimby, J.P., 2017. Reuse environment based on elaborate lexicon extend language, in: 2017 18th International Carpathian Control Conference (ICCC). IEEE, pp. 310–315.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, \Lukasz, Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Weiss, R.J., Chorowski, J., Jaitly, N., Wu, Y., Chen, Z., 2017. Sequence-to-sequence models can directly translate foreign speech. ArXiv Prepr. ArXiv170308581.

Yang, Y., Xia, X., Lo, D., Grundy, J., 2022. A survey on deep learning for software engineering. ACM Comput. Surv. CSUR 54, 1–73.

Yang, Z., Keung, J., Yu, X., Gu, X., Wei, Z., Ma, X., Zhang, M., 2021. A multi-modal transformer-based code summarization approach for smart contracts, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, pp. 1–12.

Yin, P., Deng, B., Chen, E., Vasilescu, B., Neubig, G., 2018. Learning to mine aligned code and natural language pairs from stack overflow, in: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 476–486.

Yin, P., Neubig, G., 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. ArXiv Prepr. ArXiv181002720.