Research Article

Trans-Compiler-Based Conversion from Cross-Platform Applications to Native Applications

Amira T. Mahmoud[1,*], Moataz-Bellah Radwan[1], Abdelrahman Mohamed Soliman[1], Ahmed H. Yousef[2], Hala H. Zayed[2] and Walaa Medhat[1]

[1]Nile University, Giza, Egypt
AmTarek@nu.edu.eg; Mo.Radwan@nu.edu.eg; Abd.Soliman@nu.edu.eg; Wmedhat@nu.edu.eg
[2]Egypt University of Informatics, Cairo, Egypt
Ahmed.yousef@eui.edu.eg; Hala.Zayed@eui.edu.eg
*Correspondence: AmTarek@nu.edu.eg

Received: 6th December 2023; Accepted: 23rd September 2024; Published: 1st October 2024

Abstract: Cross-platform mobile application development is emerging widely in the mobile applications industry. Cross-platform Frameworks (CPFs) like React Native, Flutter, and Xamarin are used by many developing companies. The technology these frameworks use faces performance and resource use efficiency limitations compared to native applications. The native applications are written in the native languages of the platforms. Trans-complier-based conversion between native languages of different platforms of mobile applications has been addressed in recent research. However, the problem statement needed to be mathematically represented. The solution depended on hard coding and needed more generalization. In addition, it might not be a practical solution for companies that are using and already have built applications using CPFs. Therefore, in this paper, we present an enhanced-trans-compiler-based converter to convert applications made by CPFs to native applications. We implemented the architecture to convert React Native and Xamarin applications. The React Native to Native tool converted thirteen applications to native Android and iOS applications, with accuracies ranging from 40% for large applications to 100% for simple applications. The maximum conversion time was seven minutes for converting 40% of an 8K LOC application. In addition, since Large Language Models (LLMs) are the trendiest technology in our era, we compared our proposed solution output with LLMs. We concluded its superiority compared to the status of LLMs. Performance evaluation is also done to compare the React Native applications against native applications generated by the trans-compiler tool. The assessment showed that the native applications perform better than React Native regarding runtime memory consumption, storage, and speed. The Xamarin to Native tool was also tested to show the genericness of the architecture and how it can be extended to convert from any CPF to Native applications.

Keywords: ***Code generation; Cross-platform; Mobile applications; Innovation; Resource use efficiency; Trans-compilation***

1. Introduction

Mobile application technology is widely used in our lives nowadays. Therefore, it is essential to provide mobile application developers and end-users with optimal solutions for easy development and smooth usage of mobile applications. Native development costs companies a development team for each programming language under each platform, Java or Kotlin for Android and Swift for iOS. In addition, it

Amira T. Mahmoud, Moataz-bellah Radwan, Abdelrahman Mohamed Soliman, Ahmed H. Youssef, Hala H. Zayed and Walaa Medhat, "Trans-compiler-based Conversion from Cross-platform Applications to Native Applications", *Annals of Emerging Technologies in Computing (AETiC)*, Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 1-29, Vol. 8, No. 4, 1st October 2024, Published by International Association for Educators and Researchers (IAER), DOI: 10.33166/AETiC.2024.04.001, Available: http://aetic.theiaer.org/archive/v8/v8n4/p1.html.

requires a long development time. Many cross-platform frameworks (CPFs) have recently evolved due to the growing need to develop mobile applications that can work on multiple platforms. According to the industry survey, the tools most used in the industry [1] are Flutter, React Native, and Xamarin. These tools adopted the development-once-run concept everywhere, reducing the development cost and time of mobile applications. However, these tools have limitations and drawbacks compared to traditional native development.

Different experiments were done in [2–10] to compare applications done by several CPFs and native development. The experiments showed that native mobile applications have better resource use efficiencies like memory usage, CPU usage, and power consumption. Native applications also perform better in terms of speed and user experience. These results are consistent with the outcomes of our performance comparison in this paper.

Mobile applications development challenges between native, hybrid, and web approaches were addressed in an empirical study [11]. Results showed that native development faces the challenges of code reuse and change management more than hybrid and web development. Conversely, hybrid and web approaches face more challenges regarding user experience, compatibility, and lack of expertise than the native approach.

Another approach towards solving the problem different from the known cross-platform tools is the native-to-native mobile app source code conversion [12–19]. This approach is still under research. It seeks to develop an application natively and generate a native application that runs on another platform. These solutions converted parts of the application's source code successfully. However, they lack the generalization concept as they depend on hard coding. For automated solutions like [17] and [18]The proposed solutions need a massive dataset of applications written in Java and Swift to be reliable. This kind of data is not available. However, as technology emerged, large language models (LLMs) appeared. LLMs like GPT and Bard are trained on massive amounts of data, including code sources available on the web. These LLMs are strong candidates for mobile application code translation, although they have not yet been widely tested for this task. The experiment done by Mahmoud *et al.* [20] tested ChatGPT to convert the code of web services in mobile applications from Swift to Java. The experiment was limited to web service functions only. They concluded that ChatGPT is a good candidate for this task. However, sound prompt engineering needs to be developed to specify the efficient target library of the target language to avoid generating target code with higher overhead or worse performance.

From the above paragraphs, we can infer that the CPFs provide a satisfactory solution for mobile application developers to develop the application once and run it on multiple platforms. In addition, they provide various libraries and continuous evolution to support the different and evolving functionalities of mobile applications. However, these CPFs still need to be questioned regarding the performance of mobile applications. Therefore, the main objective of this research is to address the performance limitations of cross-platform mobile applications like battery consumption, device memory, and application runtime. We address these limitations by proposing an enhanced trans-compiler-based conversion approach, emphasizing better performance, and fostering innovation for developing native applications from cross-platform frameworks like React Native. In addition, the study also aims to test LLM's capability to do the applications translation task by comparing the introduced tool results to GPT3 and BARD results for the same translation tasks.

The converter implemented in this work converts from React Native applications to Android and iOS. In addition, we have implemented the same architecture to convert from Xamarin applications but on a smaller scale to prove that the converter approach can be implemented for other CPFs. This second implementation used the same code generators we implemented for the React Native tool. We chose React Native to fully implement the converter since it is the second most used platform [1]. It has some limitations regarding memory consumption, which is not valid for native applications.

The contribution of this paper includes 1) introducing a converter that converts cross-platform developed applications to native applications to gain the advantages of the native functionalities with easy development and code reuse of cross-platform tools, 2) enhancing the compiler-based approach for mobile applications code conversion to make it more generic through implementing abstract code generators. 3) Handling the challenges of converting from scripting, weakly typed languages like JavaScript (for React Native applications) to Java and Swift, which are strongly typed languages. 4) Assessing the capability of

LLMs like GPT 3 and BARD to translate mobile applications and comparing their results to the trans-compiler-based conversion. 5) Evaluating the performance of React Native applications against native mobile applications generated by our converter. 6) Implementing the same architecture for Xamarin as a different CPF to prove the genericness of the presented architecture.

The paper is organized as follows: Section 2 presents the related work. Section 3 presents a background on LLMs. Section 4 explains the methodology of converting cross-platform development to native development and enhancing the code conversion approach for the compiler-based solution towards a more generalized tool. Section 5 details the implementation of conversion from React Native applications to native applications. Then, section 6 presents the evaluation methods of the React Native converter tool, and the methodology used for performance evaluation. It also presents the test setup, and each application is explained in the data set. Section 7 illustrates the implementation of the same architecture to convert applications from Xamarin to Native. Section 8 presents the results and discusses the outcomes that were attained. Then, section 9 presents threats to the validity of the presented approach. Finally, Section 10 presents the conclusion and future work.

2. Related Work

This section presents the categorization of previous solutions for mobile application development. Previous survey papers [10, 21–27] categorized the cross-platform solutions from the followed approach perspective. Those categories mainly are 1) web approach, 2) interpreted approach, 3) hybrid approach, and 4) cross-compilation approach. However, this paper focuses on the conversion from cross-platform languages to native languages. Therefore, in this section, previous research is categorized from the perspective of fundamental differences in the tools and programming languages used, inspired but not similar to the taxonomy done in [28], into four categories: 1) cross-platform development without native languages, 2) native-to-native development, 3) Native code generation through image processing, and 4) Converting from cross-platform development to native development.

2.1. Cross-platform Development Without Native Languages

Many cross-platform development tools nowadays use web, hybrid, interpreted, or cross-compiled approaches. These tools are all characterized by using languages other than Java and Swift for development. For example, the Ionic/Cordova framework uses a web-based approach, which depends on smartphone web browsers. Developers use HTML, CSS, and JavaScript to develop the applications. Applications developed by this approach run on the platform's WebView; thus, they lack the native feel of mobile applications.

On the other hand, React Native, introduced by Facebook in 2015, uses the hybrid approach; developers use JavaScript to develop the applications. The idea is that both Android and iOS platforms use JavaScript engines, which browsers use. So, both platforms can understand and run JavaScript through virtual machines. React Native made use of this fact and built a bridge over these virtual machines to develop applications in JavaScript one time and run them on both platforms. The advantage of this bridge is that it can build platform-specific interface objects for each platform, which gives the applications a native look and feel for users. However, developers who use React Native sometimes need to write some native code for each platform. This limitation requires developers to adapt to the frequent evolution of native components on each platform. Consequently, in the long run, projects created using React Native may be challenging to maintain. This is because their code will contain more native implementations than the shared JavaScript implementation.

A few years ago, Google introduced the Flutter tool, which follows an interpreted approach. Developers implement the applications using the Dart programming language. The applications are AOT (Ahead of Time) and compiled into the native architecture. Flutter uses a rendering engine to create and interact with the application's User Interface (UI). It is considered one of the promising tools nowadays. However, developers need to learn Dart programming language, which is unpopular outside Google. In addition, applications developed by Flutter require more memory for both application storage and runtime memory consumption [1].

Another tool that follows the interpreted approach is Xamarin. Microsoft introduced Xamarin using C# to implement mobile applications. The logic part of the applications is then compiled for each platform, but the UI part is created separately for each platform with its SDK (XML for Android and storyboard for iOS). Recently, the founders added "Xamarin.Forms" to allow developers to create the UI using the cross-platform tool. However, C# is not a native language; developers must learn it well. In addition, they must know how to use the "Xamarin.Forms" and all the needed plugins. Moreover, Xamarin is considered an interpreted solution. Therefore, it requires high memory usage, such as Flutter. A summary of cross-platform development tools that don't use native languages is presented in Table 1, along with the development language, the advantages, and the disadvantages of each tool.

Table 1. Cross-platform development tools that do not use native languages

Tool	Approach	Development language	Advantage	Disadvantage
Ionic/Cordova	Web-based approach	HTML, CSS, and JavaScript	-quick testing -large community	-WebView that lacks the native feel.
React Native	Hybrid approach	JavaScript	-Native feel -large developing community	-performance overhead due to virtual machines. -difficult to maintain due to native components
Flutter	Interpreted approach	Dart	-High performance regarding application speed -Rapidly increasing developing community	-High memory consumption regarding storage and runtime
Xamarin	Interpreted approach	C#	-Native UI provided -Professional support by Microsoft	-High memory consumption regarding storage and runtime

2.2. Native to Native Development Tools

Another approach to solving the problem is native-to-native conversion. This approach favors letting developers implement the applications natively in either Java for Android development or Swift for iOS development, and a corresponding native code will be generated for the other platform. In [13] and [15], a methodology that converts parts of Java code to Swift code and vice versa, was presented. Conversion is done by passing Java code as a web search query over popular programming resources. After that, the engine returns the equivalent Swift code block. The method is successful with small code block conversion, and it needs further fine-tuning from the developers to reach accurate codes.

Many tools that were based on the trans-compiler approach are used to convert from Java to Swift, Including the j2Swift tool and the TCAIOSC tool [16][20][29]. Both tools depend on a parser generator called ANTLR[1] that takes the grammar of the input language and generates parsing functions for it. These functions are overridden to generate the target output code. The disadvantage of this solution is the lack of generalization concept and depending on hardcoding and static development. Vendramini *et al.* also use the compiler-based approach. [30] to convert from Swift to Kotlin. The idea was introduced as a proof of concept, and only prototype applications were tested. Results showed that compiler-based application conversion is a promising way to reach a cross-platform framework.

iSpecSnapshot tool was introduced in [31]. The tool converts iOS UI to Android UI using a model-driven reverse engineering approach. It extracts design information from the iOS applications and generates functional specification documentation and the Android UI skeleton.

Another native-to-native approach was introduced in [17] The authors proposed an inference engine that inputs Java and corresponding Swift source codes. The engine aligns corresponding lines of code through string similarity and braces, marking code blocks. After alignment, the engine generates inference rules like mapping between Java and Swift depending on code alignment and syntax trees of both source codes. To run the converter, the Java code is taken as input, and according to the rules already generated, Swift code is built and generated as output. The limitation of this tool is the need for a vast data set that includes pairs of code blocks in Java and Swift to be able to generate inference rules that cover any new input code to be translated—a similar limitation faced by the tool introduced by Hassan *et al.* [18]. A neural

[1] https://www.antlr.org/

machine-based approach converts from Java to Swift and vice versa. Although this solution appears to be easily generalized, like [17], massive datasets for mobile applications written in different languages are needed to train the neural network and generate the correct code for mobile applications. Native-to-native UI components mapping was introduced in [33]. The mapping engine is done through reverse engineering of applications that are developed for Android and iOS. The tool separates the UI into modules in both versions, identifies the modules contributing to the same visual and functional effect, and automatically mines the mapping relations. However, like [17] and [18], this approach needs a massive dataset of applications that are developed natively for Android and iOS to get a reliable mapping engine.

WasmAndroid tool that converts native Android to Assembly code was introduced in [34]. The tool requires developers to compile applications' source code to web assembly to make native applications work cross-platform. However, this solution makes applications work as web-based applications only with web-view.

The following table (Table 2) summarizes related work concerned with the native development of mobile applications.

Table 2. Mobile application development approaches with native languages

Reference	Approach	Advantage	Disadvantage
[13], [15]	web search query over popular programming resources	-using the native language of each platform increases the performance	-successful with small code block conversion -it needs further fine-tuning from the developers to reach accurate codes
[16], [29]	Trans-compiler based native-to-native code translation	-using the native language of each platform increases the performance -depending on parsers and grammar rules increases the robustness of code translators	-The tools are incomplete, and translated code needs manual editing -limited by the hard-coded translation rules and lacks generalization
[31]	Model-driven reverse engineering approach	-generalized approach -using the native language of each platform increases the performance	-depending on the UI to translate, the application might miss functional properties
[17]	inference engine	-generalized approach -using the native language of each platform increases the performance	-needs a vast dataset to increase the accuracy of translation
[33]	mapping engine through reverse engineering	-generalized approach -using the native language of each platform increases the performance	-needs a vast dataset to increase the accuracy of translation
[18]	Neural machine-based translation	- generalized approach -using the native language of each platform increases the performance	-needs a vast dataset to increase the accuracy of translation
[34]	web-based approach	-Android applications are native and thus have high performance	-web view applications of iOS lack the native feel
[35]	Image processing and deep learning	-generalized approach -no implementation is needed	- only Android applications are generated
[36]	Image processing and CNN	-generalized approach -no implementation is needed	-can only deal with simple, non-complex applications
[37]	RNN	-generalized approach -no implementation is needed	-only generates Android applications that consist of one screen
[38]	Reverse engineering	-Evaluate user satisfaction and compare between Ionic and native Android	-not generic. It was just a case study -only UI was evaluated. No other performance aspects were measured.

2.3. Native applications code generation using image processing

Some research groups reached an automated general way to generate the application code in native languages. In [35], an automated code-generating tool is proposed that inputs the applications' UI pages- The tool uses image processing and deep learning classification techniques to generate the UI code of the input applications in the native language of each platform. This solution is a good start. However, it generates only the UI code and does not generate any backend code. On the other side, the ImagingDev tool [36] also takes the application's UI pages as input and generates the UI and backend code for Android, iOS, and Windows phones. The tool uses image processing and CNN (Convolutional Neural Networks) to identify the elements and components in the UI and generate the code. ImagingDev is also considered a good start toward automation. However, it can only deal with simple, non-complex applications where all

features are recognized from UI. Another tool called Doodle2APP [37] converts freehand UI sketches to native Android code using RNN (Recurrent Neural Network). The tool generates only Android applications that consist of one screen[39].

2.4. Cross-platform to Native Development

A step towards converting cross-platform developed applications to native applications was taken in [38], reverse engineering was used to migrate an open-source movie application from Ionic to native Android. Furthermore, a user study was conducted to test the user experience differences when using an Ionic-developed app and a native app. Results showed that users preferred the native version of the application. The study doesn't present a tool for general conversion; they just tested the migration of one application to measure the user experience and interaction with the Ionic application Versus the Native application. The evaluation depended only on user satisfaction measurement and didn't mention any performance criteria. In our assessment, we focus on evaluating the performance of the generated application versus the application developed by the CPF.

3. Background on LLMs for code translation

In recent years, the utilization of Language Models (LMs) has gained significant attraction across various domains, with one notable application being code-to-code translation. Huge language models (LLMs) have revolutionized the field of natural language processing and found applications beyond conventional tasks such as text generation and sentiment analysis. In the context of programming languages, LLMs have shown promise in facilitating the conversion of code from one programming language to another, enabling cross-platform development, and enhancing code reusability.

The emergence of LLMs in the code translation landscape can be attributed to advancements in deep learning techniques, particularly transformer-based architectures. Transformers, introduced by Vaswani *et al.* in the paper "Attention is All You Need"[40] have become a cornerstone in natural language processing tasks. With their attention mechanisms, transformers have demonstrated the ability to capture intricate patterns and dependencies in natural and programming languages.

Recent research on code translation or code generation was limited to translating on the function level and included languages like C++, Python, Java, and C#[41–46]. No work has been done to translate CPF languages like JavaScript or Flutter, although LLMs can perform such tasks. However, no work has been done specifically to translate mobile application codes or complete applications to other applications. Therefore, we have focused on the more extensive and general LLMs like ChatGPT and BARD. OpenAI's GPT-3 (Generative Pre-trained Transformer 3) stands out as a prominent model capable of understanding and generating human-like text across various applications, including code. GPT-3, with its 175 billion parameters and a training data size of 570 GB[47], exhibits a remarkable capacity to comprehend the context and generate coherent responses, making it a potential candidate for code translation tasks. In the realm of LLMs, Google introduced BARD based on their earlier breakthrough PALM (Pathway Language Model). BARD is trained on a massive dataset of text and code. It utilizes the PALM architecture, which employs a novel "data parallelism" technique to train large language models on massive datasets efficiently. In this paper, we will compare our trans-compiler-based code converter to the conversion resulting from ChatGPT and BARD.

4. Proposed Architecture

The proposed architecture shown in Figure 1 is a trans-compiler-based approach where applications are converted from the high-level programming language of CPF, which will be referred to as source programming language (Ls), to another target destination language (Lt) of the native application. If the input application is composed of a sequence of tokens (t_1, t_2, t_3,…..t_n) that form code statements (S_1, S_2, S_3,…S_m), these code statements follow the Grammar of the input language GLs that consist of a set of Rules (R_1, R_2, R_3,…R_x). The output of our Trans-compiler is a sequence of tokens (t_{t1}, t_{t2}, t_{t3},…t_{ty}) that form code statements (S_{t1}, S_{t2}, S_{t3},…S_{tz}) in the target language (Lt). Each output code statement must follow two constraints: 1) It must follow the target language grammar GLt to be syntactically correct, 2) it must maximize the BLEU score (bilingual evaluation understudy) [48] The BLEU score is based on comparing

each output code statement to the golden truth or the reference statement in the reference code, which performs the same functionalities as the input code.

The first step towards conversion is building a parser for the source language Ls. The parser's role is to discover the structure of the input code. The grammar file of the input language GLs is passed to ANTLR (Another Tool for Language Recognition) parsing tool to generate parser and visitor classes for the input language. Since our software tool implementation is in Python, ANTLR is configured to create a parser and visitor in Python language.

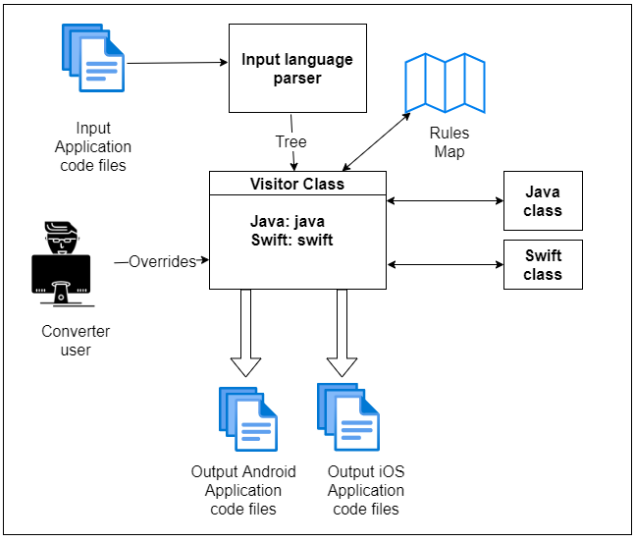


Figure 1. Converter Architecture and Workflow

The visitor class, generated by ANTLR, is a class that acts as a tree walker. This class contains x visit methods, one method for each rule R in the GLs. Those methods are overridden in a local class visitor in our converter code to make specific actions: 1) Collect the needed parameters to generate the target codes in Java (L_{t1}) and Swift (L_{t2}). 2) Put these parameters in a dictionary to be easily accessed in the code generation step. 3) Call the JSON map with the needed rule to be translated and send the parameters' dictionaries. shows the workflow of our converter.

When the converter receives an input file, the top-level rule corresponding to the parse tree's root node is called. The parse tree of array declaration in JavaScript is shown in Figure 2. The root function corresponding to the root node is called the "visitProgram" function in the JavaScript Grammar, which is the GLs in our case. Then, the other overridden visit methods are called in a nested manner according to the input code hierarchy.

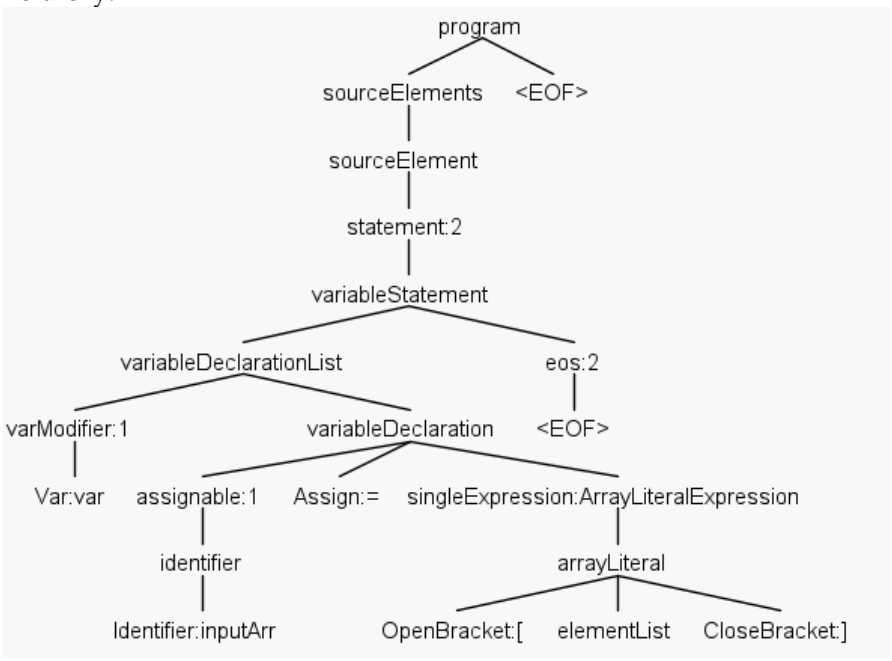


Figure 2. Parse Tree generated by ANTLR for array declaration in JavaScript

For example, the input JavaScript code in Figure 3 has an array declaration and a for loop that appends numbers to that array. The parse tree for the array declaration statement is shown in Figure 2. The visitProgram method is called from main, then the visitStatementElements, visitStataementElement, then visitStatament, and so on according to the hierarchy of the parse tree shown in Figure 2. These hierarchical calls are continued until the functions corresponding to the nodes that have the essential parameters for mapping are visited. For the example of the array declaration, we need the array name and elements instantiated in the array, if any. This information can be retrieved from the identifier and array literal nodes. We need the iterator, its instantiation, the increment, and the stopping condition for the for loop. Therefore, when the corresponding functions to the nodes with the needed information are called, we get the information using the get text built-in method in the visitor class by ANTLR. This information is saved in a dictionary with keys representing the needed information. For the for loop, the dictionary has keys like var_data_type, identifier, start condition, end condition, and post-increment. Each key has the value corresponding to it retrieved from the text of the input code.

```
var inputArr = []
for(i = 0;i<100;i++)
    {
    inputArr.push(i)
    }
```

Figure 3. Input JavaScript code snippet example

Class JAVA and class SWIFT are created inside the visitor class to build up the Android and iOS application code files. Those classes have functions that act as a code generator for each rule R in the grammars of the target languages L_{t1} and L_{t2}. The code generators are generic and can convert from any language to Java and Swift. A map function is defined in the visitor class, and it receives the input rule name and the parameters needed to build up the Java and Swift statements. For the for loop in the code snippet example Figure 3, the rule name "forstatement" is passed to the JSON map that maps it to java_for_statement and swift_for_statament. Both functions are then called, and the for_loop dictionary is passed as a parameter to each function. Figure 4 shows the Java and Swift functions that act as code generators for the for loop.

```
def java_for_loop_statement(self,statements):
  try:
    java_statement = "for(" + statements["var_data_type"] + " " +
    statements["start_condition"] + "; " + statements["end_condition"] +
    "; " + statements["post_increment"] + ")"
    return java_statement
  except Exception as e:
    print(e)
```

```
def swift_for_loop_statement(self,statements):
  try:
    java_statement = "for " + statements["identifier"] + " in "
    + statements["start_condition"] + "..." + statements["end_condition"]
    return java_statement
  except Exception as e:
    print(e)
```

Figure 4. Java and Swift Code generator for the for-loop statement

Each function builds up the Java or Swift statement and returns it to be written in the target application files. The former trans-compiler-based solutions write Java and Swift codes inside the visitor class. This means that for any source language Ls, the code generation would be done inside the visitor class and written every time from the beginning. The separate Java and Swift classes created in our tool will avoid the repeated work of writing the code generation part every time for different Ls. That's why the code

generator is considered more generic than the previous tools and can be used to convert from any source language to native mobile application languages, Java for Android and Swift for iOS, as long as GLs is available. Figure 5 compares the old and the new architectures of code generation.

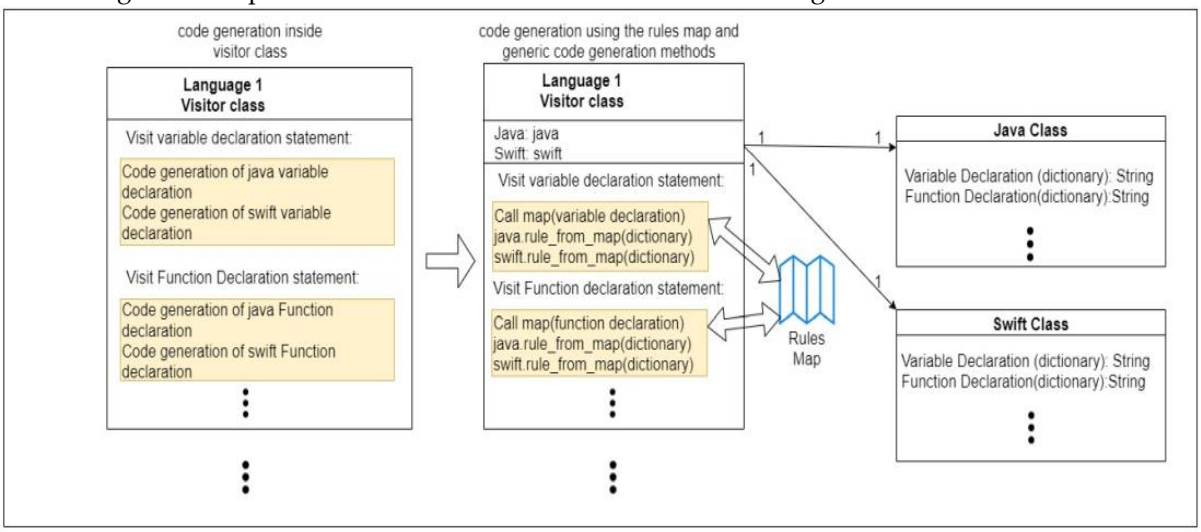


Figure 5. Architecture comparison between code generation inside visitor class and code generation using a map with generic code generators

Lines Of Code (LOC) is one of the famous software product metrics. Somerville [49] stated in his famous book on software engineering that "Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be." The LOC metric compared the development effort and code complexity between the old and the new architectures. Suppose we defined the conversion problem as having a set of source languages LS that we need to convert to the native languages, Java Lt1 and Swift Lt2. The conversion using a compiler-based solution has complexity Xs*(C+Ft1+Ft2) measured by LOC, where Xs is the number of Rules of GLs or visitor class methods that will be overridden. C is the LOC responsible for collecting the needed data for the code generation step. Ft is the LOC required for code generation inside each statement for the target language. Consider Lp the set of source languages (L1, L2, L3,…..Lp). The complexity measured in LOC for converting from source languages (Lp) to two target languages, Lt1 and Lt2, is shown in equation 1. Xsi is the number of grammar rules of the source language LSi; using the generic code-generating functions will eliminate the terms Ft1 and Ft2 from equation 1.

$$Complexity\ of\ code\ conversion\ from\ set\ of\ languages\ P = \sum_{i=1}^{i=p} X_{Si} * (\mathrm{C} + \mathrm{F}_{t1} + \mathrm{F}_{t2}) \qquad (1)$$

For example, the LOC of our converter, when implemented to convert from React Native to Android and iOS, is 3021 LOC, and the generic part that is responsible for code generation and will be reused in conversion from any other platform is 1025 LOC. This means that 33.9% of LOC are reduced and will not be repeated in future conversion from a different cross-platform tool.

5. Implementing the architecture for conversion from React Native to Native

To illustrate the tool architecture, we implement the conversion from React Native to native applications. React Native cross-platform is selected since it is one of the most used tools nowadays, according to the study made in [1] [6] . React Native is a JavaScript library for building user interfaces. The mobile applications written by the React Native tool are written in JavaScript language, or its newer edition called TypeScript, and use the React Native library.

React Native applications run three main threads for any app: a native thread for Java or Swift, a JavaScript thread that runs JavaScript (JS) and UI components, and a bridge thread that links between the first two threads. Application development is relatively easy using React Native due to its simplified UI and the presence of a large developer community. However, it has drawbacks and performance issues, like the application size. Whenever the application communicates with third-party sources, uses multiple screens, or many libraries, the application size becomes more extensive, especially for the Android version. Developers must compress the application files like images, graphics, and JSON files. Another problem is

the difficulty of application maintenance. The maintenance process is not smooth due to the JS thread, native thread, and the distributed components among the threads.

For the mentioned drawbacks above, we implemented our tool to convert React Native applications to native ones with higher performance and easier maintenance. As mentioned in the general description of the converter, ANTLR builds a parser for the input language and JS for the React Native conversion. The converter reads the React Native application and converts each file. The converter generates two applications: one for Android, which is composed of Java and XML files for UI, and another for IOS, which is composed of Swift files. The following subsections will show some code conversion challenges and how each was addressed.

5.1. Backend code conversion

One of the most challenging problems in converting from React Native to native Android and iOS is converting from a loosely typed language like JS to a strongly typed language like Java and Swift. Since loosely typed languages have some ambiguity in data types, variable declaration, function declaration…etc. The following subsections represent the discovered ambiguity problems, and their solutions or possible solutions implemented in our converter.

5.1.1. Variable declaration

Variable datatype is not written in JS, while in Java and Swift, it must be specified. JS has only three variable types: numeric, string, and object literal. In contrast, Java and Swift have many other data types like double, float, char, string …. etc. Three solutions were introduced to handle the variables of the datatype problem. Each solution is used in a particular case: 1) If a variable assignment is with the declaration, the variable type can be extracted according to the assigned value. For example, in the statement" var x=1," the datatype can be extracted to be int, and for (var name=" john"), the data type is extracted as a string. 2) Another way is to use the "var" keyword. The "var" keyword is available in Swift and Java version ten or later. In this case, the variable declaration can go without explicitly declaring the datatype. However, the "var" keyword must be accompanied by an assignment to that variable since the compiler decides the variable's datatype from that assigned value. This way is proper when the assignment is an expression or a function that returns a value. 3) The third way is the hardest. It is used when the variable declaration is written in JS without any assignment to the variable. Object declaration is used in Java, and the keyword "Any" is used in Swift. Both can be used as a datatype to declare a variable with an unknown type.

5.1.2. Function signature

Function signature in JS has only function name and parameters, leaving many ambiguous elements, including function access modifiers, return type, and parameters' datatypes. These elements are needed for correctly writing Java and Swift code.

For the function signature ambiguity, we navigate the function body to check the return statement inside the function. This check either gets the function with no return and generates Java and Swift functions with void, or it uses the "Any" keyword to indicate that the function's return is unknown. The "Any" keyword can be used in Java 10 and later versions, Swift 3, and later versions.

The "Any" keyword can be used for the function parameters datatypes. However, to assign the parameter by a value inside the function, it must be declared first inside the function. This caused inaccuracies when using the "Any" keyword. Another powerful solution for this problem is using the Object data structure in Java. Declaring the function parameters as objects allows the function to assign any value to the parameters inside it; moreover, declaring the function's return type as an object also allows the return of any data type. This solution is valid only for Java for Android.

5.1.3. Arrow Functions

Arrow functions are used heavily in JS. Arrow functions are used to write shorter function syntax. However, these functions can have no name and no parameters. They can be written in arrays or loops like lambda expressions in Java. In this case, they are run automatically. Sometimes, they are declared inside a variable and have a name. In this case, they run only when called like normal functions. Arrow functions have several syntaxes and uses. The code in Figure 6 shows four different syntaxes for arrow functions in JS.

```
const temp = ()=>{ statements }
const temp ()=> FunctionCall;
const temp => console.log("hello world");
<Button title = "count" onPress={()=>
{openCamera()}}/>
```

Figure 6. Arrow function different syntaxes

The problem is that arrow functions do not have an equivalent in Java and Swift languages. We cannot convert it to a lambda expression or a standard function in all cases. As a solution for this problem, three cases are identified for the arrow function. For each case, a suitable solution is implemented: 1) Arrow function with a name: if the arrow function is declared within a variable and has a name, it is mapped to a standard function declaration and runs whenever it is called. 2) Arrow function with no name: if the arrow function has no name and is written explicitly, then this means that it runs automatically. Therefore, it is mapped to a standard function declaration followed directly by the function call. 3) Arrow function with no name inside on press listener: arrow functions are commonly used inside on press functions of buttons or pressable text. In this case, the arrow function's body is selected to be converted inside the body of the "onclick listener" of the button in Java and the action function in Swift.

5.2. UI code conversion

UI of the React applications is written as embedded HTML tags within the JS code, in addition to a stylesheet function written in JS that specifies the style of each UI element by id. Both HTML code and style are in the same JS file. However, for Android, the backend part is written in Java files, and the style part is written in XML files. The same applies to iOS; it has parts written in Swift's backend, and other parts need storyboard files. Converting the UI components of the mobile applications is divided into two parts.

5.2.1. UI backend code conversion

Two JSON files are used for UI backend code conversion. One has a mapping between HTML UI elements' names and Java UI elements' names. The other file has the mapping for the Swift UI elements' names. In the visitor class, the Parse tree visits the HTML tag. The tag is sent to the JSON files to be mapped to the UI element in the target language. A dictionary is made to collect all needed parameters in a key-value pair for passing the UI element parameters. The dictionary is then sent with the mapped tag to form the backend code of the UI element in the target languages. Table 3 shows the name mapping of some UI elements between HTML, Java, and Swift.

Table 3. UI Elements Mapping

JavaScript	Java	Swift
Text	TextView	Text
TextInput	EditText	TextField
Image	ImageView	Image
Switch	Switch	Switch
Button	Button	Button
Activity Indicator	Progressbar	Circle

The mapped UI components are the most used ones. Other UI mappings could be done, but we focus on some essential UI elements to show the ability of UI conversion, including UI backend and UI style, besides the limitations that can appear in UI conversion. More UI mapping will be done in our future work to improve the tool's overall performance. One of the challenges encountered in converting the UI back end is naming UI objects. In React Native, UI components might not have a name. For example, when multiple buttons are created on the same screen, there is no identifier for each button. This doesn't cause a problem for iOS applications, while each UI component must have an ID in Android. This ID is essential and is used to create the style in an XML file. As a solution to this problem, the button title was taken as its name in Android.

5.2.2. UI style conversion

The second part of converting UI is to convert the style of the UI element. Some of the style attributes of UI elements can be written in the target backend code (Java for Android and Swift for iOS), or they can be written in separate XML and storyboard files. The UI style is mapped to XML for Android. However, it

is almost impossible to map the HTML and JS style to the storyboard since the storyboard depends on positioning and placing the items one after the other. Conversely, HTML, like XML, depends on naming each style. For React Native applications, whenever a UI object is created, it takes one of the named styles written separately in a style function. Another reason for not generating the storyboard file is that XCode automatically generates it, and no developers write storyboards manually.

As a solution for the iOS UI elements style, the converter generates the style that can be written inside the Swift file. However, this solution is challenging since the stylesheet is written separately after declaring the tags in the JS file. Therefore, we created an extracting function to extract the style of UI elements from the stylesheet function. Then, the extracted style attributes are sent as a dictionary with the Swift UI element to the code-generating function.

5.3. Mobile Applications-Specific Operations Conversion

Making phone calls, sending SMS or email, opening a camera, or opening Google Maps are essential functionalities in many mobile applications. Therefore, supporting the conversion of these operations is important. The standard statements mapping cannot be used for these operations since each operation is done by multiple lines either in the source or target platforms. Functionalities mapping is divided into two parts as follows.

5.3.1. Simple functionalities

Simple functionalities are the ones done in a few sequential lines of code. A recognizer function first recognizes the operation and then sends it to the JSON map to be mapped to the target platform operation. After the operation is identified and mapped, the code generators of both target languages are called, the necessary parameters are sent, and the operation code is generated. Examples of simple functionalities are making phone calls, sending SMS, and opening Google Maps from an application.

5.3.2. Complex functionalities

React Native has some powerful built-in functionalities that are unavailable on Android and iOS. For example, the RN camera library has a built-in function made by Google that can recognize barcodes and return them in a list. This function is written in a single line in React native applications. However, doing the same task in Android or iOS requires multi-line code with many functions and setups. To solve this problem, the ready-made function in React Native is mapped to a separate file in the Android and iOS applications. Then, this separate class is imported, and an object is made inside the code to perform the same task.

5.4. Library-to-Library Mapping

UI elements and special functionalities include using platform-specific libraries and classes. Therefore, it is important to map importing the necessary libraries and classes for these elements to work properly. Mapping the imported libraries and classes in React Native to native applications is divided into three parts:

5.4.1. Direct Mapping of Built-in Libraries

Many classes in React Native, especially the UI elements-specific classes, have correspondence in Android and iOS. The proposed converter can map more than twenty classes from React Native to native Android and iOS classes. The mapping is done through two JSON files, one for Android and one for iOS. Mapping these classes proves that mapping the usage of these classes and their methods inside the code is applicable.

5.4.2. Mapping classes made by application developer

Sometimes, application code files may need to import methods from each other since they are classes made by the application developer. This import is written in the form “import method_name from class_name.” The syntax of import from class is the same as that of Java and Swift. However, not all "import from" is an internal class; sometimes, it is a React Native-specific class. To map this kind of import, a lookup table is created that saves all code file names at the beginning of the conversion. Whenever an import-from statement is triggered, the lookup table is checked. If the class name after import is found in the look-up table, then the import statement is mapped as it is. Other than that, the class name should be looked up in

the JSON files used in the previous case. If it is not found in JSON files, the converter does not support it, and the import statement is not translated.

5.4.3. Built-in Functionality

The previous sub-section mentioned that some complex functionalities are mapped to a separate class in Android and iOS applications. These classes need to be imported into the code for use. Whenever a complex function is called in React, it is converted to a class object in Java and Swift. However, the import statement must be written at the top of the code. To solve this problem, the import statements are written in a separate file during the conversion. After the file is completely converted, the import file is merged with the code file into one single code file.

5.5. Tool user manual

The React Native to native converter tool is implemented in Python programming language on Google Colab. The tool's intended users are mobile application developers. The tool provides a simple UI that allows the developer to provide the input React Native application, convert the files, and take the output iOS and Android applications files easily. The tool User Interface is shown in Figure 7.

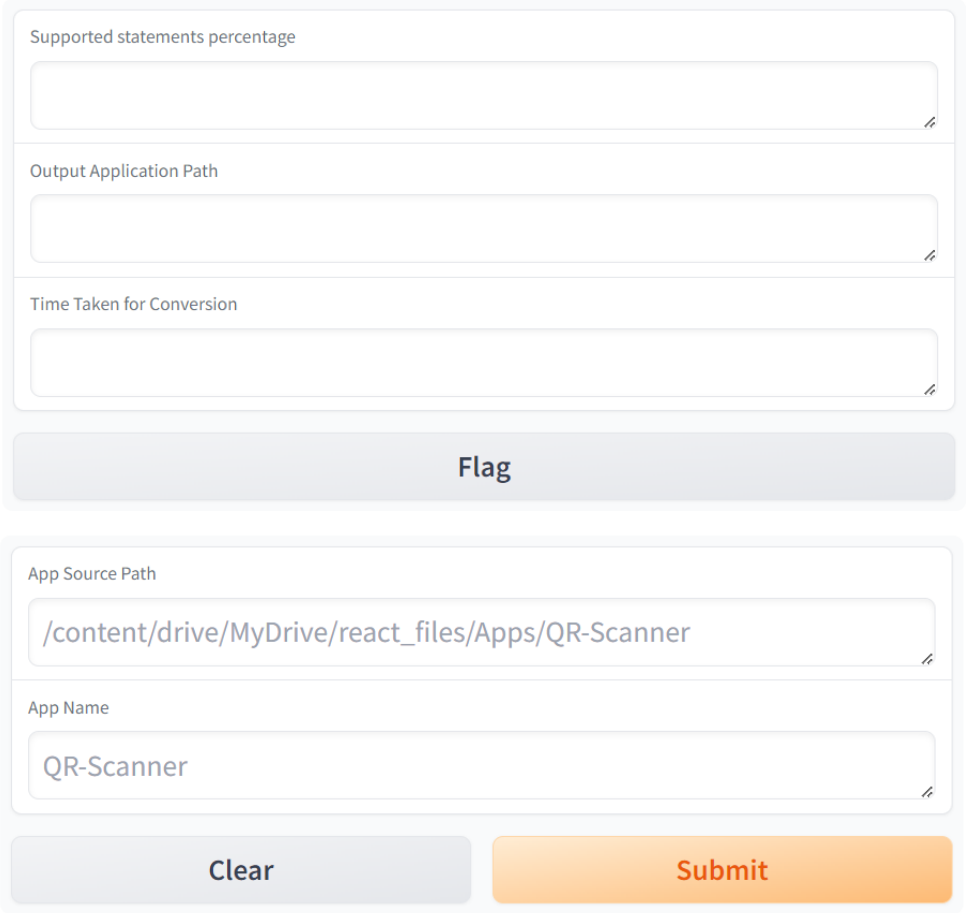


Figure 7. Code converter tool User Interface

The user should mount his drive to be connected to the Colab notebook. The user should also upload the react files found on the tool's GitHub repository2, to his connected drive. Then, the user should upload the React Native application folder to the apps folder inside the React files uploaded to his drive. Afterward, the user provides the React Native application path and application name in the text box, as shown in Figure 7. Then, the user can press submit to start the conversion process.

After the conversion finishes, the tool will create an output repository on the user's drive, where a folder with the application name will also be created. This folder has two separate folders: Java, which has all the Android application files, XML and Java files, and Swift, which has all the iOS application files. The

[2] https://github.com/amiratarek3007/React-to-Native

developer can see through the UI the percentage of supported statements by the tool calculated as the number of code statements converted with no errors over the total number of statements of the input code. This percentage is calculated for each JavaScript file and shown to the user after conversion. More details about the statement's percentage evaluation methodology are in subsection 6.1. The user can also flag the output to store these data in a CSV file. The CSV file acts as a log file that stores data for each conversion process. The log file stores the path of the input application, the path of the output application, the percentage of converted statements, the time taken for the conversion process measured in seconds, and a time stamp.

The tool also allows the developer's users to edit the JSON files responsible for the functions, libraries, and UI elements mappings from React Native to Android and iOS. For example, suppose the user wishes to add a new UI component mapping from React to Android. In that case, he can choose the JSON file Java UI map from the drop-down menu shown in Figure 8. Then, the user can add a new key-value mapping directly through the UI, updating the conversion process dynamically. For example, the Image UI component should be mapped to ImageView in Java. After the user adds this entry, the new mapping is saved in the Java UI map JSON file and will be effective for later translations.

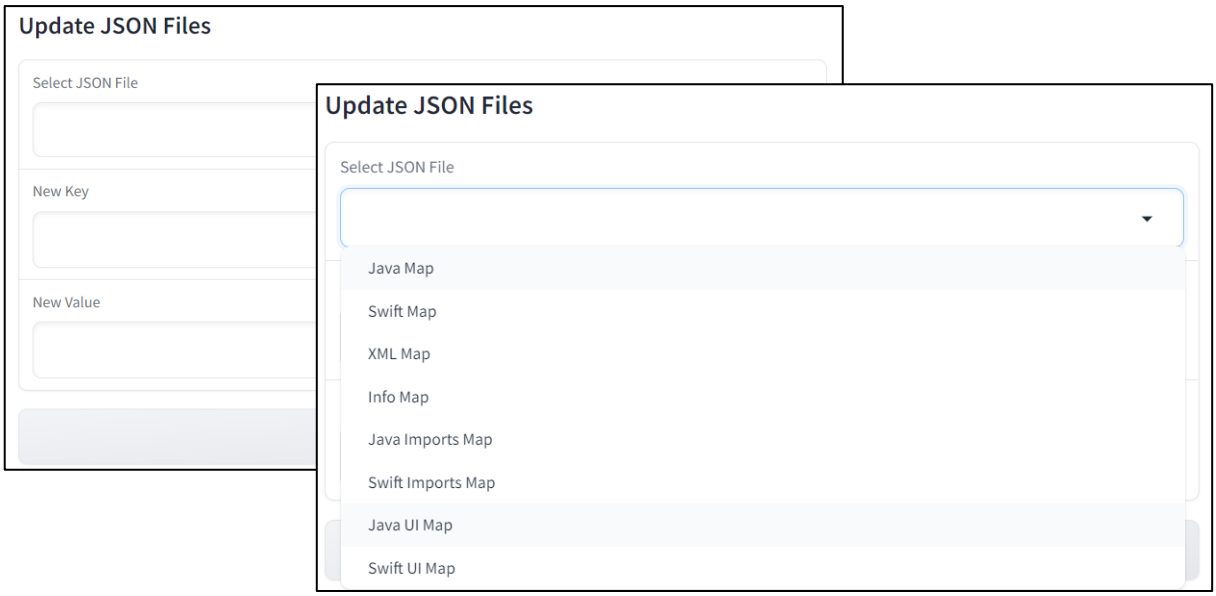


Figure 8. Adding a new mapping entry using the JSON file update interface.

6. Evaluation Methodology

The Evaluation has two main aspects. The first is evaluating the completeness and correctness of the generated native code. The second is assessing the performance of the generated native applications versus the React native applications.

6.1. Code correctness evaluation

We have two phases for the assessment of code correctness. Phase one involved relatively simple applications that consisted of tens to hundreds of LOC. Evaluation is done by calculating BLEU scores for the generated applications. BLEU scores are also compared to scores of LLMs' translations to evaluate and benchmark our tool for small application conversion. The second phase involved assessing the ability of the converter to translate real applications of thousands of codes and tens of files. In this phase, compiling on target, IDE is used to compute the correctness of the translation of the actual application. The following subsections will explain BLEU and accuracy by compiling it on IDE. Afterward, we will explain the two phases of evaluation and the test applications used in each.

The first method is BLEU (bilingual evaluation understudy) [48]. BLEU is a tool created to measure the quality of machine-translated natural language. The idea behind BLEU is that it compares the machine-translated piece of text to the reference translation that humans do. BLEU is widely used to evaluate LLMs' ability for code translation [42–46]. Therefore, BLEU was a suitable tool to measure the quality and accuracy

of the output code from our converter. The methodology behind BLEU is that it compares the machine-generated test with the reference text sentence by sentence regardless of the positioning of the words in the same sentence. It counts the number of words that match. The larger the count of the matches, the better the translation quality.

In our case, we made multiple reference translations: one for evaluating our tool, one for assessing ChatGPT output, and the third one for evaluating Bard output. We compare the three BLEU scores to one another for each target language L_t. This comparison not only aims to assess our tool but also to evaluate the LLMs' capability to translate mobile applications. This comparison allows future scientific research to integrate trans-compiler-based conversion with LLMs. This integration can achieve a more correct and accurate code translation.

The evaluation against ChatGPT and Bard is done only for the small applications scope since passing large application files to these models is not available, the number of tokens passed is limited, and passing a massive application with many files is not currently available with the free open version of these LLMs. In addition, to compare the translation of real applications with the BLEU score, we must create a complete reference application, which is very costly and time-consuming for large applications. Figure 9 shows the flow for evaluation using the BLEU score for the small applications.

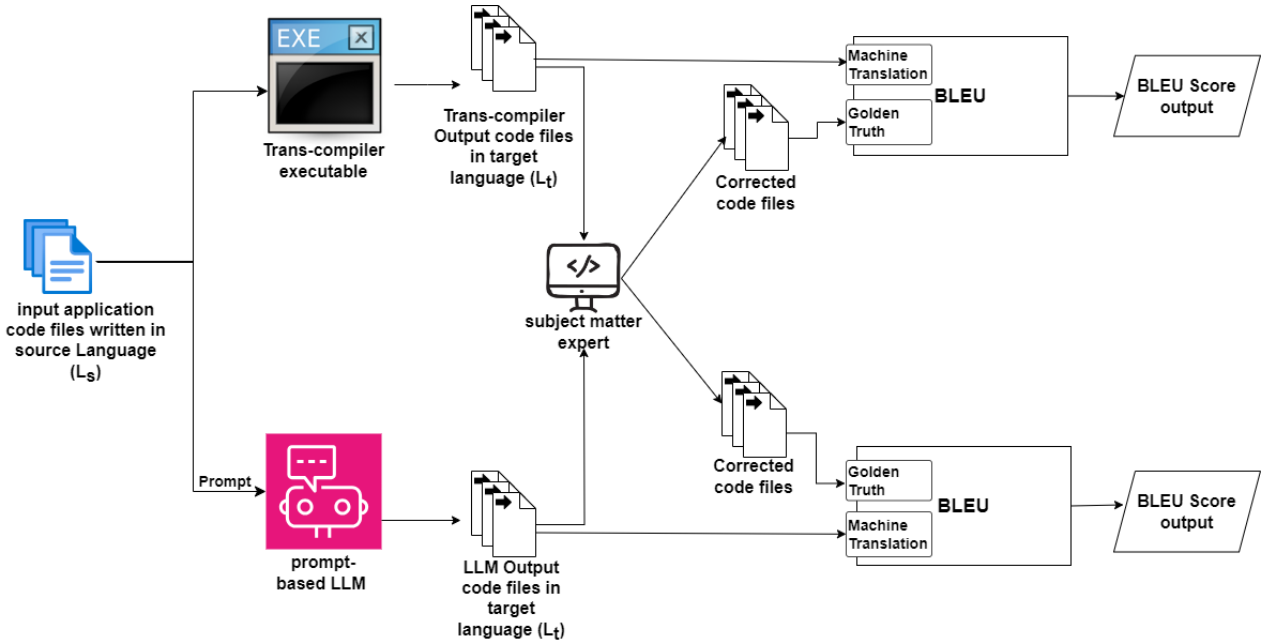


Figure 9. Evaluating Trans-complier and LLMs output translation using BLEU

Once we have output translation for the input application, our subject matter experts correct the output code generated by the trans-compiler, ChatGPT, and BARD. Each application has two target languages: Java for Android and Swift for iOS. Therefore, the experts use the IDEs of both output languages to correct the output code, Xcode for iOS and Android Studio for Android. After correction, they ensure the application runs on a mobile device or an emulator with the target platform and performs the same functionalities as the input application. Afterward, the corrected application code is considered the reference or the ground truth to which the output of the trans-compiler or the LLM is compared. We pass the ground truth and the machine-translated output to the BLEU evaluator, once for every converter, once for the trans-compiler, another for ChatGPT, and the third for BARD translation. This process is done for each application for both output versions, Java and Swift, for the three machine translators.

The second method evaluates the application syntax by placing each generated application on the target IDE. Android Studio is used for Android applications, and XCode is used for iOS applications. This way of evaluation doesn't imply having a reference translation. Therefore, we used this evaluation technique for real applications conversion evaluation. This phase aims to test our converter's ability to deal with tangible applications that involve more complex UI and database-related statements. As mentioned in the above section, we couldn't compare with LLMs' translation in this phase, as they suffer from limited input tokens.

Accuracy is determined by calculating the number of correct lines over the total number of lines in the code. The following subsections explain in detail the five real applications used in testing.

$$Compiling\ Accuracy = \frac{Number\ of\ correct\ LOC\ of\ the\ output\ application}{Total\ number\ of\ LOC\ of\ the\ output\ application} \quad (2)$$

The third method is automated inside the converter itself. It evaluates the converter's ability to read the input code statement without parsing errors and find the corresponding code generation function. A code statement means a main code structure, like a function declaration statement, a variable declaration, a condition, a loop…etc. The evaluation depends on the try-except blocks placed inside the translator code. Whenever no error is caught, this code statement is supported by the tool and mapped to a code generation function that writes the code to the output file directly. This way of evaluating does not imply having a reference translation. Therefore, this evaluation technique is helpful for real application conversion evaluation. The accuracy calculated by equation 3 depends on the translator's behavior with the input code statements, and the accuracy is calculated as supported conversions over the total statements of the input code. Therefore, the calculated percentage is independent of the output LOC in Android and iOS output applications. This evaluation acts as an indicator of the percentage of statements covered by the tool for the input code. The tool's UI shows the percentage statements covered to the user after the conversion process is completed. The percentage of the covered code statements is shown for each input file, as well as the average total percentage.

$$Covered\ statements\ percentage = \frac{Number\ of\ supported\ code\ statements\ of\ the\ input\ application}{Total\ number\ of\ code\ statements\ of\ the\ input\ application} \quad (3)$$

6.1.1. Phase one: Simple applications

This section explains each application used to evaluate our tool and what conversion functionality each application tests. Four of the simple applications are from GitHub. The QR scanner application is an app that opens the camera, reads QR codes, and generates the link for the QR code to be read. This application tests the converter's ability to map a function in React Native to a class in Android. The age calculator application takes the user's birth year, calculates the age, and prints it on the screen. It tests converting different data types like date, string, and int. The currency converter application receives the amount of money from the user as text input, and then the user presses a button to convert from USD currency to EGP currency. It tests the ability to convert taking input from user and dealing with it in a mathematical equation. The calculator application is a simple calculator that has buttons from 0 to 9. It takes an input equation from the user as an operand followed by an operation, then another operand. The application tests the ability to convert a complete application that depends on dealing with user input. In addition, it takes user input and passes it to a series of nested if conditions. Finally, it does a mathematical equation with the user input and prints the output on the screen.

The other four applications are custom-made to include the most used simple functionalities in any application. The test cases are complete React Native applications; each application covers some features. The Map application has an image and two buttons as examples of UI component mapping. One button allows the user to open Google Maps, and the other opens the camera. It tests the ability of the converter to convert phone-specific functionality, such as accessing the camera. It also tests the external navigation to another application, Google Maps. In addition, the open camera code was created in the target iOS application as a separate class and called inside the main class.

The Communicate application allows the user to enter a number in a Text input as an example of a different UI component that deals with user input. It makes a phone call to the number the user enters as an example of phone-specific functionality. It also allows the user to open WIFI settings from a button. This shows the ability to convert another phone-specific functionality and navigate from the application to the device's internal settings.

Bubble Sort is an application where the user selects the array size using multiple buttons. Upon user selection, an array of the selected size is created, filled with random numbers, and sorted. This application aims to test the converter's ability to translate back-end code statements like loops and conditions and convert arrays with its built-in functions.

The screens application was made mainly to test the navigation between multiple screens inside the application. The first screen is a welcome page with an image and text input to take the username and print hello and the name entered by the user. It tests dealing with changeable text views. Then, a button navigates

to the second screen. The second screen is a counter. The count keeps increasing with each button press by the user. It also has an activity indicator UI component in the shape of a moving circle. Finally, another button is placed to navigate to the third screen. The third screen acts as a social media navigator. The screen has three buttons, and each navigates to a social media application, which is called external navigation. A button opens Facebook, another opens Instagram, and the last opens Twitter.

6.1.2. Phase Two: Real applications

Five different applications were used to test the converter's accuracy for converting real applications. We focused our selection on applications ten times larger than the simple applications in the first phase of the section. Moreover, since we support only JS conversion in the current phase, we selected applications that used JS and not typescript in the backend code.

The first application is a water tracker application. The water tracker application includes features like setting a daily target through glasses based on the user's BMI. Users can log in to their consumption, and the levels should be updated. It also has a time-based reminder with a message. The Application consists of 15 JS files with a total of 825 LOC.

The second application is the NFTs marketplace application. NFTs are non-fungible tokens that represent transferrable rights to digital assets, such as art, in-game items, collectibles, or music [50].The application users can place bids on NFTs and see current bids. It has a total of 980 LOC in 14 JS files. The application users can create their own profiles and upload photos of their products.

The third application is Authentication using React Native. It involves login and Signup, which is the base of any application. It is a pervasive and fundamental flow in many applications. For example, each application needs this flow in finance, E-Commerce, social media, etc. The application has five different screens: sign-in, register, home, settings, and logout screens. The application includes navigation between screens, custom sidebars, and input components. It uses a web API connection for authentication. It comprises 10 JS files with a total size of 1049 LOC.

The fourth application is a Travel application. The app recommends popular tourist destinations to users based on specified categories. The application consists of 33 JS files, a total of 2177 LOC. The application has a welcome screen where users can log in or create a new profile. To create a profile, the user can upload his photo or open a camera to take the picture. Since users can create profiles, authentication, and database connections are implemented by Firebase. After a user creates their profile, they can insert the location they are interested in, and the application suggests places with pictures of each place.

The last and largest application is an e-commerce application with 95 JS files and 8690 LOC. The application is considered the largest and most advanced tested application to be converted by the tool. It offers the complete experience of any e-commerce application, including a user profile, real-time updates, a shopping cart, placing orders, payment methods, resetting passwords, and updating order information.

6.2. Performance evaluation

Two applications were used to evaluate the performance of native and React Native applications. One application involves intensive processing equation shown in equation 4. This equation is used in many applications, for example, games, applications with augmented reality, and applications for image treatment, among others [51].

$$s = \sum_{j=1}^{5} \sum_{k=1}^{100000} (\log_2(k) + \frac{3k}{2j} + \sqrt{k} + k^{j-1}) \quad (4)$$

This equation was used previously in previous evaluations done in [51] and [52]. The second application is a video playback application that opens a video using its link inside the application. Video playback and media playing in general is used as a benchmark in previous evaluations like [51] and [53]. It is also common in social media applications, which are the most used nowadays.

The test is done by implementing both applications with the React Native framework and then passing the application to our converter to generate the native code for Android. Since the generated code is not 100% complete, we had to complete it manually to ensure it is correct and can run on Android devices. Neither application was included in the test applications mentioned in sub-sections 6.1.1 or 6.1.2 since converting these applications is to measure performance rather than the converter's ability. Therefore, they are simple applications and do not involve many UI components or code conversion challenges. One device

was used for the Android platform: Oppo Reno 5 Model CPH2159 with a Qualcomm Snapdragon 720 G Octa-core and 8.00GB RAM. The Android version during the test period is Android 12 with operating system ColorOS version V12.1. iOS platform was not included in the performance evaluation test as it is much harder on iOS devices to run a test application without going through the publishing procedures, which are hectic ones. In addition, we depended on the internal device measurements to capture the performance, and using the simulator for that purpose would not be effective. Therefore, the Android platform was only included in the performance evaluation.

Four performance aspects were compared to evaluate the React Native applications vs the native applications: 1) Storage, 2) Runtime memory consumption, 3) Speed, and 4) Battery consumption. Regarding the storage measurement, each application's APK file size is compared for React native and native Android implementations. Runtime speed was calculated inside the application itself; we used a function that gets time at the start of the application, calculates the time again at the end of the app, and prints the difference.

Average runtime is calculated for 30 runs for the application. Thirty runs were used in several previous works, like[54], [51]and[52] and they were proved to be a good number for an accurate mean value. After running the application 30 times, memory usage is checked from the internal measurements of the device and power consumption in the form of the percentage of battery used by the application. For runtime memory and power consumption, different measurement tools were used in the previous work done for performance evaluation. The use of external tools for measurement is aimed at reducing the effect of the different test devices used and unifying the measurement approach to get accurate measures. However, performance evaluation is not the primary context of our work. The evaluation is just done to compare the cross-platform framework against the native development without interest in the individual efficiency of each approach in general. Therefore, we decided to use one device for Android, and the internal system measurements provided by the Android device were used to make the comparison. The thirty runs are done while no other application is open on the device and with a full battery charge to reduce external effects.

7. Implementing the Architecture for conversion from Xamarin to Native

To prove the genericness concept of the presented architecture, we implemented the architecture for converting from applications made by Xamarin frameworks to Native applications. The presented architecture provides general code generators for Java and Swift that could be reused for conversion from any cross-platform framework. However, the part of the visitor class responsible for parsing the input code must be rewritten to parse each source language. Therefore, to convert from the Xamarin framework that uses C# language, the visitor class was implemented to parse the input applications and pass the needed code elements to the Swift and Java code generators. The Xamarin converter implementation was done on a smaller scale compared to the React Native to the native tool as it was done to prove the genericness concept of the presented architecture.

Xamarin is a mobile cross-platform framework founded in May 2011 by Miguel de Lcaza and Nat Friedman. Microsoft acquired Xamarin in February 2016. Developers can use Xamarin to develop Android, iOS, and Windows apps. Xamarin uses C# as codebase and Microsoft .NET as Common Language Infrastructure. C# is one of the most widely spread languages, and it has well-illustrated documents within a vast community. However, Xamarin has worse memory, runtime, and CPU usage compared to the native versions of the same application[55, 56] which could be more efficient in terms of performance.

In the following subsections, we will present the primary conversion covered by the Xamarin to Native tool and the main challenges of conversion from C# to Java and Swift.

7.1. Backend code conversion

The conversion challenges for backend code structures like declarations, control flow statements, and loops are less for generating Java than Swift. This is due to the close nature of language structure between C# and Java. Table 4 shows examples of backend code statements for the three different languages.

Table 4. Backend code statements in C#, Java, and Swift

C#	Java	Swift
int myNum = 15;	int myNum =15;	var myNum:Int =15;
string[] arr = {"a", "b", "c", "d"};	String[] arr ={"a", "b", "c", "d"};	Var arr:[String] ={"a", "b", "c", "d"}

for (int i=0;i<10;i++){
Console.writeLine(i);
}
	for(int i=0;i<10;i++){
	System.out.println(i);
	}
		for i in 0…10{
		print(i)
		}

while(i<10){
Console.writeLine(i);
i++;
}
	while(i<10){
	System.out.println(i);
	i++;
	}
		while i<10{
		print(i)
		i++
		}

If(i==10){
Console.writeLine("number is 10");
}
	If(i==10){
	System.out.println("number is 10");
	}
		if i==10{
		print("number is 10")
		}

7.2. Mapping of Built-in Libraries

Mapping built-in libraries also depended on JSON files, which act as a database of code mappings. We could use the automated library mapping tool by Ahmed *et al.* [57] to automatically generate the JSON files instead of manually writing them. This approach couldn't be used in the React Native tool since the JavaScript language is loosely typed, and the library mapping tool [57] depended on the function parameters and their datatypes in computing the similarity of the functions' signatures to be mapped. Seven JSON files were extracted from the tool in [57] for the libraries: Math, String, File, Array, HashMap, Array List, and Calendar. The mapping was done from C# to Java and from C# to Swift.

7.3. Xamarin tool evaluation methodology

To evaluate the Xamarin to Native converter, we tested the tool on one complete application, a calculator application. The application involves back-end code statements like loops, conditions, declarations, and built-in functions of Math and String libraries. The BLEU score method explained in section 6.1 evaluates the generated code.

8. Results and Discussion

8.1. React Native to Native Evaluation Results

After passing the applications to the converter, the generated applications are evaluated using two approaches. One approach is to calculate the BLEU score for the generated applications. The other approach is compiling the output code, which includes Android applications on Android Studio and iOS applications on XCode. Since using BLEU implies having a reference code written in the native languages, we couldn't use BLEU to evaluate big applications like E-commerce, the authentication app, the travel app, the water tracker, and the NFT marketplace app explained previously. Compiling an IDE and calculating the percentage of correct LOC was used to evaluate the big applications. In addition, the percentage of covered statements of the input code was also measured for the big applications as a primary indicator for the tool users before they compile the output on the target IDE. Therefore, the conversion accuracy is divided into two separate parts:

8.1.1. Small Applications Conversion Results

The conversion accuracy measured by BLEU for the applications generated by our tool vs. ChatGPT and BARD LLMs are shown in Table 5.

The XML files generated by our tool for all eight test cases were 100% accurate and didn't need any modification to work on Android Studio. From the results, we can notice that applications like QR scanner, Map, communicate, and multiple screens apps have higher accuracy than bubble sort, age calculator, currency converter, and calculator.

The limitations of our presented tool and the reasons why these applications got lower accuracy can be summarized as follows:

- These applications deal with different data types and need many data type casting to work appropriately in Java and Swift. Our converter cannot determine datatype casting. For the converter

to know the data type it needs to cast, it will need to know the value that will be assigned to these variables. This is something almost impossible as many of these values are given by the user during runtime and can never be known before runtime.

- In the Age calculator application, the date library was used, and it is not supported by our converter yet. We supported the well-known built-in functions like functions of arrays and strings. In future work, more built-in functions will be mapped.
- Sending function parameters as "Any" sometimes produces an error, and the datatype must be written significantly. As mentioned in the challenges section, we encountered the ambiguous data type problem of parameters sent to functions. The parameters sent as "Any," especially in Swift, gave an error when they were assigned inside the function. There is no known way till now to fix this problem.
- Putting the text of Buttons as an identifier in Android produced errors. Buttons in React Native don't have an identifier or name. However, every UI component must be given a name in Android. To solve this issue, we used the text of the button as its identifier in Android. This produced errors in the calculator application as the test of the buttons had numbers, and numbers could not be identified in Java. Identifiers must begin with a letter. We are considering developing a function that produces random names for components like this to solve this issue. This function will be implemented and tested in future work.
- For Android applications, the application name must be written in the first line of the Main Activity file to set the application ID, and this information is not provided in the JS code files. Our converter also cannot give any random name as this name is important and will be used by the developer working on the application. Therefore, we left it for the application developer to manually decide and write the package declaration.

Table 5. BLEU scores for generated applications from trans-compiler, ChatGPT, and BARD

Converter	Trans-compiler (BLEU score)		Chat-GPT (BLEU score)		BARD (BLEU score)	
Platform/Application	Android	iOS	Android	iOS	Android	iOS
QR scanner	100	93.97	6.43	4.52	6.93	0.84
Map Main	99.26	100	89.84	19.06	97.62	3.98
Communicate	98.86	86.87	86.42	94.94	100	92.27
Bubble sort	93.59	93.59	45.32	77.68	100	99.11
Age calculator	83.78	81.76	100	95.11	100	69.99
Currency converter	80.68	79.58	100	94.59	98.92	97.78
Calculator	70.75	71.10	79.75	83.44	56.14	3.57
Welcome.js	98.18	94.68	100	96.82	100	79.03
Counter-App.js	94.93	96.49	100	51.13	100	74.94
Social media.js	99.03	100	100	100	100	98.0
Average score	91.90	89.80	80.77	71.72	85.96	61.95

Regarding the analysis and comparison to ChatGPT and BARD results, the trans-compiler demonstrated consistent performance across various applications, particularly excelling in mobile applications-specific scenarios like the QR scanner, map navigation, and communication application. It consistently provided high BLEU scores for both Android and iOS conversions. This indicates that the trans-compiler effectively mapped React Native code to native languages, showcasing adaptability to different functionalities and UI components. The average BLEU score, around 91.91 for Android and 89.80 for iOS, reflects its overall proficiency. However, applications involving user input and using these inputs in calculations, like the calculator and currency converter, had lower BLEU scores. As mentioned before, this returns to the need for many data type casting and data type differences between the source and target language.

GPT, specifically ChatGPT, demonstrated notable performance, especially in more straightforward applications like the calculator, age calculator, and currency converter. On the other side, for the bubble sort algorithm, it didn't translate the algorithm itself, but it called the built-in sorting function of the array in the Android translation. This was tricky since it appeared to be better and more intelligent; however, when we passed the same input but with an inverse sort algorithm, it used the same function, which got a different output. For the applications that involved screen navigation, like the welcome app, it outperformed Android outputs. However, it failed to perform the navigation code in iOS. It struggled with mobile application-specific scenarios, which is evident in the lower BLEU scores for applications such as map navigation and QR scanner. The average BLEU scores for Android (80.77) and iOS (71.72) indicate an excellent overall performance, but the model's limitations in handling intricate functionalities are apparent.

BARD exhibited strong performance in specific applications, particularly communication, bubble sort, and currency converter. However, it faced challenges in handling mobile application-specific functionalities, as seen in lower BLEU scores for the QR scanner for both iOS and Android, as well as map and navigation for iOS outputs. The most apparent limitation of BARD was that it couldn't handle long inputs like the calculator application; it couldn't generate a complete output for the application. In addition, iOS outputs are very inconsistent. The average BLEU scores for Android (85.96) and iOS (61.95) highlights a mixed performance.

The trans-compiler outperformed both GPT and BARD in handling complex functionalities and scenarios. ChatGPT demonstrated adaptability in more straightforward applications, while BARD showed specificity in specific scenarios. The trans-compiler has maintained consistency across various applications, while GPT and BARD exhibited mixed performance. The trans-compiler consistently achieved higher average BLEU scores, indicating overall effectiveness on the tested applications.

In conclusion, the choice of converter depends on the complexity and specific requirements of the application. The trans-compiler stands out for its versatility, while GPT and BARD may be more suitable for more straightforward or specific-use cases.

8.1.2. Real Applications Conversion Results

Five more complicated applications, explained in section 6, were used to test the conversion accuracy. The generated applications for both iOS and Android platforms were imported to Xcode and Android Studio, respectively. The accuracy of conversion was measured by applying the LOC percentage methodology and the percentage of statements covered, explained in subsection 6.1. The LOC percentage shows the correct LOC for the output application. Therefore, we have a percentage for Android and another for iOS for each application. The statement percentage indicates the number of statements covered from the input React Native application. Therefore, it is represented as one percentage for each application. The LOC of the input application, the statement percentage covered by the tool, the correctly generated LOC percentage, and the conversion time in minutes are shown in Table 6.

Table 6. Conversion Accuracy Results for the real applications

Application	Input Application LOC	Percentage of input app statements covered by the tool	Android studio test (Java)	XCode test (Swift)	Time of conversion process (minutes)
Authentication app	1049	63.85%	60.0%	62.01%	1.80
E commerce app	8690	57.90%	41.57%	62.55%	7.61
Water Tracker app	825	49.25%	48.15	50.39%	0.55
NFT Marketplace app	980	46.20%	42.63%	45.05%	1.36
Travel experience app	2177	39.56%	38.62%	40.10%	3.43

The authentication with the API application had the most remarkable accuracy compared to the other four apps. The inaccuracy in converting these apps, in addition to the points mentioned above for the simple apps part, are:

- The tool does not yet support Web API connection statements used in the authentication app. APIs are common and used in many applications. However, it is constantly changing; finding a way to support different APIs is challenging and needs to be collected as a database that developers can edit.
- Many UI components failed to be converted since they are not usual or commonly used components, but they are custom-made inside the application itself, like the water tracker and NFT market

applications. Converting custom-made UI is almost impossible since the application developer creates it himself. These kinds of code statements will need developer interference. In addition, the current tool doesn't support many UI components, as we focused on the backend statements more than the UI components.

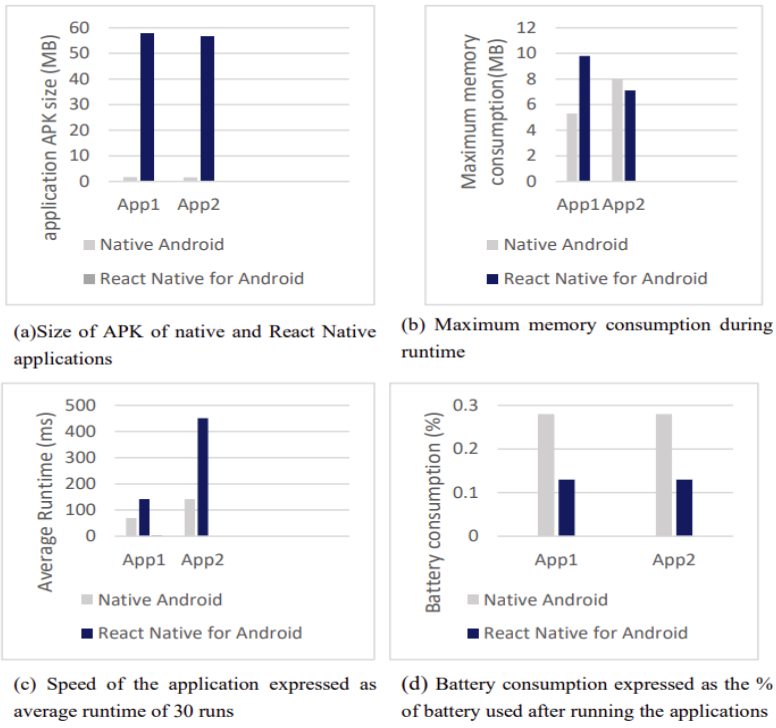
- The tool does not yet support database and implementation statements. The E-commerce and travel applications had database-related statements, as mentioned in the description of the tested applications. Database support in our tool can be done in the future. Still, it needs much research and experience to convert these types of statements and different database frameworks like Firebase, SQLite, Realm, etc.
- Some backend statements, like importing libraries with no equivalence in the target languages, were not converted, which affected the accuracy of the conversion.

The conversion times were measured for the five applications of varying sizes, as shown in Table 6. The time taken for conversion shows promising results, especially when compared to the anticipated time required for manual conversion of the same applications. The most extensive application in this study, with 8690 LOC, took 7.61 minutes for conversion. Given the sheer size and complexity of an e-commerce application, this conversion time, even for 41% of the Android application and 62% of the iOS application, is considered tiny. A manual conversion for the same portion of an application of this magnitude would typically take weeks of work by two skilled developers, one for each platform. The time savings achieved by the tool are notable and provide strong evidence of its practical utility in accelerating the development process, particularly for applications with large codebases or complex functionality.

Unlike the small applications, the conversion of real applications was not compared to GPT and BARD. Since these real applications are composed of tens of files and cannot be passed easily to the LLMs we are using, we limited the comparison to the small applications only.

8.1.3. Performance Evaluation Results

Performance is one of many metrics used to evaluate a mobile application, and many other metrics can evaluate a mobile app, as mentioned in [58]. However, in this paper, we focus on evaluating the performance of mobile applications developed by a cross-platform framework like React Native vs native applications generated from our tool. As mentioned previously in the test setup section, we have used two benchmarks for evaluating the performance. The result of assessing average runtime memory, application APK size, average runtime, and battery consumption is shown in Figure 10, where App 1 is the data processing application and App 2 is the video playback application.



(a)Size of APK of native and React Native applications

(b) Maximum memory consumption during runtime

(c) Speed of the application expressed as average runtime of 30 runs

(d) Battery consumption expressed as the % of battery used after running the applications

Figure 10. Performance evaluation test results to compare React Native

The maximum consumption of memory during runtime for the data processing app on the React Native platform is double the consumption for the native Android app. For the video playback app, the React native application had less memory consumption, with a very small difference (0.9 MB). This difference happened since the internal code of the video player library in the input React application might not be implemented in the same way in the output Android application. The video player library is a built-in library, and we converted it to the built-in library in Java for Android, and both libraries might have different implementations. The application size represented by the size of the APK of each application is very different. The React Native app is almost 50 times larger than the native app. The size of an application is a crucial aspect for smartphone users. As reported in some surveys[3,4], people may delete apps due to the lack of storage on their phones.

Regarding the speed of the application, it is also noticed that the native app is almost four times faster than the React Native one for the video playback application and two times faster for the data processing application. This is due to the overhead caused by JS threads and native components in the React native application and the communication between these threads. The speed of an application is crucial, especially for gaming applications that are very popular among smartphone users.

The React native applications had slightly better battery usage performance. The native applications' battery usage didn't exceed 0.28%, while the React Native apps were in the range of 0.13%. The difference might not be very significant for a mobile app user.

8.2. Xamarin to Native Evaluation Results

To evaluate our conversion output for the calculator application, we tested the output compared to the expected output for Android and iOS. The evaluation is done by passing the actual output and the expected code to BLEU, as illustrated earlier in section 6.1. The BLEU score achieved is 88.25 for Android and 83.09 for iOS. The calculator application test shows that over three-quarters of the application was converted successfully. The Java for Android score beats Swift for iOS, as C# and Java have many similarities in structure and UI components. The unconverted ratio is mainly due to the difference in datatype mappings and function signatures in function declarations. For the library mapping, especially for the Swift language, there were unmapped built-in functions between libraries like the Math, String, and Array libraries. Finally, the UI components not covered by the converter tool at this stage also caused a lower BLEU score.

9. Threats to validity

The introduced approach target is to convert mobile applications from one source code to the native source code of mobile applications. The conversion is based on a trans compilation process that parses the input code and edits the tree walkers to generate the corresponding native code. The first threat to our tool validity is that for a trans-compiler-based converter to be able to convert any input code, it needs to test all possible inputs, trace the parse tree, and override to write the corresponding target code. Our trial to alleviate this problem is that we tested as many statements as possible and searched for the many and most common ways a single statement can be written to be supported by the converter. However, the converter will fail to convert any new statement that was not supported. Our plan to minimize this threat is to open the tool to receive inputs from mobile application developers for statements that are not supported and try to support them if possible. In addition, they can also add corresponding native code to the generator classes responsible for generating the target native code. Another solution for this limitation will be to integrate our tool with LLM solutions and use this integrated module only for statements not supported by the trans-compiler.

The second challenge is the non-availability of direct corresponding code in the target language. We have solved this partially, as mentioned in the complex functionality mapping. For example, the QR scanning and camera opening didn't have corresponding libraries in the target native languages, and our solution was that we created specific classes for these cases written in the target native languages that can be easily called in the native application. This solution can be expanded to cover more libraries and

[3] https://themanifest.com/app-development/blog/mobile-app-usage-statistics

[4] https://gadgets360.com/mobiles/news/62-percent-of-indians-run-out-of-smartphone-space-every-3-months-sandisk-1829349

functionalities that don't have corresponding direct translation. However, it might not be easy to adopt this solution in the case of customizable functionalities that might be created by the developer while building an application. For example, adopting this solution with customized UI components is difficult.

Another threat would be the ability of the converter to deal with the fact that the application code is constantly changing. For example, if the input code "A" to our converter corresponds to output "B", it would need some modifications "B¯" to be complete. Our concern is that when modification happens in the source application, and it becomes "A+dA," the corresponding code would be "B+dB"; however, it should be "B¯+dB." Porting the changes continuously would not be a practical solution. A more practical solution would be to separate the generated code from the customizable one. Currently, this threat has not been solved using our approach. However, we plan to work on this concern in our future work.

10. Conclusion and future work

This article introduces the architecture of a tool that converts cross-platform mobile application code to native code. The architecture introduced is more generic and less dependent on hardcoding than previous compiler-based solutions. The generalization is done in the part specified with Java and Swift code generation. Code generation introduced in the architecture can be converted from native to native and from any cross-platform framework to native. The architecture is illustrated by implementing the conversion from React Native and Xamarin applications to native Android and iOS applications. Our tool aims to provide mobile application developers with an accessible development environment. It gives a "develop once, run everywhere" privilege without sacrificing the high-performance advantages of native mobile applications. It also provides easier application maintenance since the native code of the application will be available.

For the React Native tool, the conversion evaluation was done by converting eight simple and five real applications. For real applications, the average LOC accuracy for our converter is 45.8% and 44.6% for Android and iOS applications, respectively. The conversion accuracy is computed for the other eight applications by computing the BLEU score and comparing it to LLM-based translation. The average BLEU scores of the eight simple applications are 91.90 and 89.80 for Android and iOS, respectively. By comparing the small applications' BLEU scores to the translations generated by GPT3 and BARD, we found that the trans-compiler gave the highest average scores for the eight applications. Although GPT and BARD had higher scores in some applications, neither maintained a consistent performance. Both succeeded in translating simple backend code but failed to translate mobile application-specific functionalities. Finally, a performance evaluation test was done to evaluate the generated native applications against React Native applications. The test assessed runtime memory consumption, application size, average runtime, and battery consumption. The native applications were better than the React Native applications in all aspects except battery consumption. However, the battery consumption in the maximum case did not exceed a 0.1% difference.

For the Xamarin to Native tool, the conversion was tested on a complete application involving multiple code structures and library mappings. The test done is less than that of the React Native tool since the Xamarin tool is less mature. It was done to prove the genericness concept of the presented architecture. However, the results show that more than three-quarters of the complete Xamarin application was successfully converted to an Android and iOS application.

Results show that this approach can be the basis for a good tool for mobile application developers. However, there are some limitations to reaching a rigid, powerful tool that can be used in the actual industry of mobile applications. Future work to enhance and improve the conversion tool and address its limitations will include:

- Increasing the features, functions, and libraries supported by the React Native to native converter and supporting the conversion of UI components.
- Supporting the conversion of frameworks used by React Native, like redux, to manage application updates and scaling.
- Separate the generated code from the customized code to adapt to modifications done to the input code.

- Making the tool smarter by learning from real mobile application developers' suggested translations. Suggested translations will be added as new rules in the code conversion source code. Currently, the tool allows the developers to add a new mapping to the JSON files. Expanding this feature to add to the source code will enhance the tool's performance and ability to translate more libraries and mobile application-specific functionalities.
- Classifying the current gaps in our approach to the ones solvable by trans-compiler-based solutions and gaps addressed by LLM-based solutions. Then, we will integrate our trans-compiler with an LLM-based translator to complete the non-solvable gaps by trans-compilation.
- Enhancing the Xamarin to native tool to support more mobile application-specific functionalities and implementing more framework converters like Flutter to native.

References

[1] Amira. T. Mahmoud, Ahmed. A. Muhammad, Ahmed H. Yousef, Hala. H. Zayed, Walaa. Medhat *et al*., "Industrial Practitioner Perspective of Mobile Applications Programming Languages and Systems", *International Journal of Advanced Computer Science and Applications (IJACSA)*, Print ISSN: 2158-107X, Online ISSN: 2156-5570, Vol. 14, No. 5, July 2023, pp. 257-285, Published by The Science and Information (SAI) Organization, DOI: 10.14569/issn.2156-5570, Available: https://thesai.org/Publications/ViewPaper?Volume=14&Issue=5&Code=IJACSA&SerialNo=29.

[2] Piotr Nawrocki, Krzysztof Wrona, Mateusz Marczak and Bartlomiej Sniezynski, "A Comparison of Native and Cross-Platform Frameworks for Mobile Applications", *Computer*, Print ISSN: 0018-9162, Online ISSN: 1558-0814, Vol. 54, No. 3, 15th March 2021, pp. 18-27, Published by IEEE, DOI: 10.1109/MC.2020.2983893, Available: https://ieeexplore.ieee.org/document/9378923.

[3] Stefan Huber, Lukas Demetz and Michael Felderer, "Analysing the performance of mobile cross-platform development approaches using ui interaction scenarios", *Communications in Computer and Information Science*, Print ISSN: 1865-0929, Online ISSN: 1865-0937, 2020, pp. 40–57, Published by Springer, DOI: 10.1007/978-3-030-52991-8_3, Available: https://link.springer.com/chapter/10.1007/978-3-030-52991-8_3.

[4] Kewal Shah, Harsh Sinha and Payal Mishra, "Analysis of Cross-Platform Mobile App Development Tools", in *Proceedings of the 2019 IEEE 5th International Conference for Convergence in Technology*, 29-31 March 2019, Pune, India, Print ISSN: 9781538680766, Online ISSN: 1538680769, DOI: 10.1109/I2CT45611.2019.9033872, pp. 29–31, Published by IEEE, Available: https://ieeexplore.ieee.org/document/9033872.

[5] Thomas Dorfer, Lukas Demetz and Stefan Huber, "Impact of mobile cross-platform development on CPU, memory and battery of mobile devices when using common mobile app features", *Procedia Computer Science*, Online ISSN: 1877-0509, Vol. 175, January 2020, pp. 189–196, Published by Elsevier, DOI: 10.1016/j.procs.2020.07.029, Available: https://www.sciencedirect.com/science/article/pii/S1877050920317099.

[6] Mehmet Işitan and Murat Koklu, "Comparison and Evaluation of Cross Platform Mobile Application Development Tools", *International Journal of Applied Mathematics, Electronics and Computers*, Print ISSN: 2147-8228, Online ISSN: 2147-8228, Vol. 8, No. 4, December 2020, pp. 273–281, Published by İsmail SARITAŞ, DOI: 10.18100/ijamec.832673, Available: https://dergipark.org.tr/en/pub/ijamec/issue/57538/832673.

[7] Lucas Pugliese Barros, Flávio Medeiros, Eduardo Cardoso Moraes and Anderson Feitosa Júnior, "Analyzing the Performance of Apps Developed by using Cross-Platform and Native Technologies", in *Proceedings of the 32nd International Conference on Software Engineering and Knowledge Engineering (SEKE 2020)*, 9-11 July 2020, Pittsburgh, USA, Print ISSN: 2325-9000, Online ISSN: 2325-9086, pp. 186-191, DOI: 10.18293/SEKE2020-122, Published by KSI Research Inc., Available: https://ksiresearch.org/seke/seke20paper/paper122.pdf.

[8] Biørn-Hansen, Andreas, Tor-Morten Grønli and Gheorghita Ghinea, "Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance", *Sensors*, ISSN: 1424-8220, Vol. 19, No. 9, 5th May 2019, Article ID: 2081, Published by MDPI, DOI: 10.3390/s19092081, Available: https://www.mdpi.com/1424-8220/19/9/2081.

[9] Andreas Biørn-Hansen, Christoph Rieger, Tor M. Grønli, Tim A. Majchrzak and Gheorghita Ghinea, "An empirical investigation of performance overhead in cross-platform mobile development frameworks", *Empirical Software Engineering*, Print ISSN: 1382-3256, Online ISSN: 1573-7616, Vol. 25, No. 4, July 2020, pp. 2997–3040, Published by Springer, DOI: 10.1007/s10664-020-09827-6, Available: https://link.springer.com/article/10.1007/s10664-020-09827-6.

[10] Ayoub Korchi, Mohamed Karim Khachouch, Younes Lakhrissi and Anis Moumen, "Classification of existing mobile cross-platform approaches", in *Proceedings of the 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, 12-13 June 2020, Istanbul, Turkey, Print ISSN: 1466-6642, Online ISSN: 1741-8070, pp. 1-5, Published by Inderscience Publishers, DOI: 10.1109/ICECCE49384.2020.9179222, Available: https://dl.acm.org/doi/10.1504/ijict.2024.135305.

[11] Arshad Ahmad, Kan Li, Chong Feng, Syed M. Asim, Abdallah Yousif *et al.*, "An Empirical Study of Investigating Mobile Applications Development Challenges", *IEEE Access*, ISSN: 2169-3536, Vol. 6, March 2018, pp. 17711–17728, DOI: 10.1109/ACCESS.2018.2818724, Available: https://ieeexplore.ieee.org/document/9179222.

[12] Rameez Barakat, Moataz-Bellah A. Radwan, Walaa M. Medhat and Ahmed H. Yousef, "Trans-Compiler-Based Database Code Conversion Model for Native Platforms and Languages", in *Lecture Notes in Computer Science (LNCS)*, Vol. 13761, Print ISBN: 978-3-031-21594-0, Online ISBN: 978-3-031-21595-7, Series Print ISSN: 0302-9743, Series Online ISSN: 1611-3349, 19th November 2022, pp. 162–175, Published by Springer, DOI: 10.1007/978-3-031-21595-7_12, Available: https://dl.acm.org/doi/abs/10.1007/978-3-031-21595-7_12.

[13] Antuan Byalik, Sanchit Chadha and Eli Tilevich, "Native-2-Native: Automated cross-platform code synthesis from web-based programming resources", in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*, 26-27 October 2015, Pittsburgh, USA, ISBN: 978-1-4503-3687-1, pp. 99–108, DOI: 10.1145/2814204.2814210, Available: https://dl.acm.org/doi/10.1145/2814204.2814210.

[14] Shaymaa Sayed El-Kaliouby, Sahar Selim and Ahmed H. Yousef, "Native Mobile Applications UI Code Conversion", in *Proceedings of the 2021 16th International Conference on Computer Engineering and Systems (ICCES)*, 15-16 December 2021, Cairo, Egypt, Print ISBN: 978-1-6654-0868-4, Online ISBN: 978-1-6654-0867-7, pp. 1-5, Published by IEEE, DOI: 10.1109/ICCES54031.2021.9686093. Available: https://ieeexplore.ieee.org/document/9686093.

[15] Sanchit Chadha, Antuan Byalik, Eli Tilevich and Alla Rozovskaya, "Facilitating the development of cross-platform software via automated code synthesis from web-based programming resources", *Computer Languages, Systems & Structures*, ISSN: 1477-8424, Vol. 48, June 2017, pp. 3–19, Published by Elsevier Ltd, DOI: 10.1016/j.cl.2016.08.005. Available: https://www.sciencedirect.com/science/article/abs/pii/S1477842415300634.

[16] Ahmad A. Muhammad, Amira T. Mahmoud, Shaymaa S. Elkalyouby, Rameez B. Hamza *et al.*, "Trans-Compiler based Mobile Applications code converter: swift to java", in *Proceedings of the 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, IEEE, Cairo, Egypt, 24-26 October 2020, pp. 247–252, Published by IEEE, DOI: 10.1109/NILES50944.2020.9257928, Available: https://ieeexplore.ieee.org/document/9257928.

[17] Kijin An, Na Meng and Eli Tilevich, "Automatic inference of Java-to-swift translation rules for porting mobile application", in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, Gothenburg, Sweden, 27-28 May 2018, Online ISBN:978-1-4503-5712-8, Print ISBN: 978-1-5386-6170-3, pp. 180–190, Published by ACM Press, DOI: 10.1145/3197231.3197240, Available: https://dl.acm.org/doi/10.1145/3197231.3197240.

[18] Mohammed H. Hassan, Omar A. Mahmoud, Omar I. Mohammed, Ammar Y. Baraka, Amira T. Mahmoud *et al.*, "Neural Machine Based Mobile Applications Code Translation", *in Proceedings of the 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES 2020)*, Giza, Egypt, 24-26 October 2020, Print ISBN: 978-1-5386-6170-3, Online ISBN: 978-1-4503-5712-8, pp. 302–307, Published by IEE, DOI: 10.1109/NILES50944.2020.9257935, Available: https://ieeexplore.ieee.org/document/9257935.

[19] David I. Salama, Rameez B. Hamza, Martina I. Kamel, Ahmad A. Muhammad and Ahmed H. Yousef, "TCAIOSC: Trans-Compiler Based Android to iOS Converter", *Advances in Intelligent Systems and Computing*, Vol. 1058, October 2019, pp. 842–851, DOI: 10.1007/978-3-030-31129-2_77. Available: https://link.springer.com/chapter/10.1007/978-3-030-31129-2_77.

[20] Amira T. Mahmoud, Ahmad A. Muhammad, Ahmed H. Yousef, Walaa Medhat, Hala H. Zayed *et al.*, "Compiler-based Web Services code conversion model for different languages of mobile application", in *Proceedings of the 2023 Intelligent Methods, Systems, and Applications (IMSA)*, 15-16 July 2023, Giza, Egypt, Print ISBN: 979-8-3503-3557-6, Online ISBN: 979-8-3503-3556-9, pp. 464-469, Published by IEEE, DOI: 10.1109/IMSA58542.2023.10217471. Available: https://ieeexplore.ieee.org/document/10217471.

[21] Wafaa S. El-Kassas, Bassem A. Abdullah, Ahmed H. Yousef and Ayman M. Wahba, "Taxonomy of Cross-Platform Mobile Applications Development Approaches", *Ain Shams Engineering Journal*, Vol. 8, No. 2, June 2017, pp. 163–190, Published by Ain Shams University, DOI: 10.1016/J.ASEJ.2015.08.004, Available: https://www.sciencedirect.com/science/article/pii/S2090447915001276.

[22] André Ribeiro and Alberto R. da Silva, "Survey on cross-platforms and languages for mobile apps", in *Proceedings of the 2012 8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*, Lisbon, Portugal, 3-6 September 2012, Print ISBN:978-1-4673-2345-1, pp. 255–260, Published by IEEE, DOI: 10.1109/QUATIC.2012.56, Available: https://ieeexplore.ieee.org/document/6511821.

[23] Andreas Biørn-Hansen, Tor-Morten Grønli and Gheorghita Ghinea, "A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development", *ACM Computing Surveys (CSUR)*, Vol. 51, No. 5, ISSN: 0360-0300, EISSN: 1557-7341, pp. 1–34, November 2018, DOI: 10.1145/3241739, Available: https://dl.acm.org/doi/10.1145/3241739.

[24] Tatiana F. Bernardes and Mario Y. Miyake, "Cross-platform Mobile Development Approaches: A Systematic Review", *IEEE Latin America Transactions*, Vol. 14, No. 4, 2016, ISSN 1548-0992, pp. 1892–1898, DOI: 10.1109/TLA.2016.7483531, Available: https://ieeexplore.ieee.org/document/7483531.

[25] Mounaim Latif, Younes Lakhrissi, El Habib Nfaoui and Najia Es-Sbai, "Cross platform approach for mobile application development: A survey", in *Proceedings of the 2016 International Conference on Information Technology for*

Organizations Development (IT4OD), Fez, Morocco, 30 March 2016 - 1 April 2016, ISBN:978-1-4673-7689-1, pp. 1–5, Published by IEEE, DOI: 10.1109/IT4OD.2016.7479278, Available: https://ieeexplore.ieee.org/document/7479278.

[26] Mounaim Latif, Younes Lakhrissi, El Habib Nfaoui and Najia Es-Sbai, "Review of mobile cross platform and research orientations", in *Proceedings of the 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS),* Fez, Morocco, 19-20 April 2017, Print ISBN: 978-1-5090-6682-7, Online ISBN: 978-1-5090-6681-0, pp. 1–4, Published by IEEE, DOI: 10.1109/WITS.2017.7934674, Available: https://ieeexplore.ieee.org/document/7934674.

[27] Shaymaa Sayed El-Kaliouby, Ahmed H. Yousef and Sahar Selim, "Mobile Application Code Generation Approaches: A Survey", in Communications in Computer and Information Science, Switzerland: Springer Nature, Vol. 1751, 10th January 2023, Online ISBN: 978-3-031-23119-3, Print ISBN: 978-3-031-23118-6, ch. 10, pp. 136–148, DOI: 10.1007/978-3-031-23119-3_10, Available: https://link.springer.com/chapter/10.1007/978-3-031-23119-3_10.

[28] Robin Nunkesser, "Beyond web/native/hybrid: A new taxonomy for mobile app development", in *Proceedings of the International Conference on Software Engineering*, 27 May 2018 - 3 June 2018, Gothenburg, Sweden, Print ISBN: 978-1-5386-6170-3, Online ISBN: 978-1-4503-5712-8, pp. 214–218, Published by IEEE, DOI: 10.1145/3197231.3197260, Available: https://ieeexplore.ieee.org/document/8543456.

[29] Rameez B. Hamza, David I. Salama, Martina I. Kamel and Ahmed H. Yousef, "TCAIOSC: Application Code Conversion", in *Proceedings of the 2019 Novel Intelligent and Leading Emerging Sciences Conference (NILES),* 28-30 October 2019, Giza, Egypt, Print ISBN: 978-1-7281-3174-0, Online ISBN: 978-1-7281-3173-3, pp. 230–234, Published by IEEE, DOI: 10.1109/NILES.2019.8909207, Available: https://ieeexplore.ieee.org/document/8909207.

[30] Vinícius J. Vendramini, Alfredo Goldman and Grégory Mounié, "Improving mobile app development using transpilers with maintainable outputs", in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 21-23 October 2020, Natal, Brazil, ISBN:9781450387538, pp. 354–363, Published by Association for Computing Machinery (ACM), DOI: 10.1145/3422392.3422426, Available: https://dl.acm.org/doi/10.1145/3422392.3422426.

[31] Khalid Lamhaddab, Mohamed Lachgar and Khalid Elbaamrani, "Porting mobile apps from iOS to android: A practical experience", *Mobile Information Systems,* Vol. 2019, No. 1, 3rd September 2019, Online ISSN: 1875-905X, Print ISSN: 1574-017X, Article ID: 4324871, Published by Wiley, DOI: 10.1155/2019/4324871, Available: https://onlinelibrary.wiley.com/doi/full/10.1155/2019/4324871.

[32] Ruihua Ji, Junyu Pei, Wenhua Yang, Juan Zhai, Minxue Pan *et al.*, "Extracting mapping relations for mobile user interface transformation", in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 28-29 October 2019, Fukuoka Japan, ISBN: 978-1-4503-7701-0, pp. 1–10, Published by Association for Computing Machinery (ACM), DOI: 10.1145/3361242.3361250, Available: https://dl.acm.org/doi/10.1145/3361242.3361250.

[33] Elliott Wen, Gerald Weber and Suranga Nanayakkara, "WasmAndroid: a cross-platform runtime for native programming languages on Android (WIP paper)", in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21)*, 22nd June 2021, virtual, Canada, pp. 80–84, Published by Association for Computing Machinery (ACM), DOI: 10.1145/3461648.3463849, Available: https://dl.acm.org/doi/10.1145/3461648.3463849.

[34] Sen Chen, Lingling Fan, Ting Su, Lei Ma, Yang Liu *et al.*, "Automated Cross-Platform GUI Code Generation for Mobile Apps", in *Proceedings of the 2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile*, 24th February 2019, Hangzhou, China, Print ISBN:978-1-7281-1812-3, Online ISBN: 978-1-7281-1811-6, pp. 13–16, Published by IEEE, DOI: 10.1109/AI4Mobile.2019.8672718, Available: https://ieeexplore.ieee.org/document/8672718.

[35] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li *et al.*, "StoryDroid: Automated Generation of Storyboard for Android Apps", in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 23-31 May 2019, Montréal, Canada, Online ISSN: 1558-1225, Print ISSN: 0270-5257, pp. 596-607, Published by IEEE, DOI: 10.1109/ICSE.2019.00070, Available: https://ieeexplore.ieee.org/document/8812043.

[36] Viviana Y. Rosales-Morales, Laura N. Sánchez-Morales, Giner Alor-Hernández, Jorge L. Garcia-Alcaraz, José L. Sánchez-Cervantes *et al.*, "ImagIngDev: A New Approach for Developing Automatic Cross-Platform Mobile Applications Using Image Processing Techniques", *Computer Journal*, Vol. 61, No. 1, April 2019, Online ISSN: 1460-2067, Print ISSN: 0010-4620, Published by Oxford University Press, DOI: 10.1093/comjnl/bxz029, Available: https://ieeexplore.ieee.org/document/9266976.

[37] Mohian, Soumik and Christoph Csallner, "Doodle2App: Native app code by freehand UI sketching", in *Proceedings of the 2020 IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 5-11 October 2020, Seoul, South Korea, Print ISBN: 978-1-7281-9842-2, Online ISBN: 978-1-4503-7959-5, pp. 81–84, Published by IEEE, DOI: 10.1145/3387905.3388607, Available: https://ieeexplore.ieee.org/document/10187631.

[38] Maria Caulo, Rita Francese, Giuseppe Scanniello and Antonio Spera, "Does the Migration of Cross-Platform Apps Towards the Android Platform Matter? An Approach and a User Study", in *Lecture Notes in Computer Science*, Vol. 11915, 18th November 2019, Print ISBN: 978-3-030-35332-2, Online ISBN: 978-3-030-35333-9, Print ISSN: 0302-9743, Online ISSN: 1611-3349, pp. 120–136, Published by Springer, DOI: 10.1007/978-3-030-35333-9_9, Available: https://link.springer.com/chapter/10.1007/978-3-030-35333-9_9.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones *et al.*, "Attention is all you need", in *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017),* 02-04 December 2017, California, USA, ISBN: 9781510860964, pp. 6000-6010, Published by Neural Information Processing Systems Foundation, Inc. (NeurIPS), DOI: 10.5555/3295222.3295349, Available: https://dl.acm.org/doi/10.5555/3295222.3295349.

[40] Priyan Vaithilingam, Tianyi Zhang and Elena L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models", in *Proceedings of the Conference on Human Factors in Computing Systems,* 29 April - 5 May 2022, New Orleans, LA, USA, ISBN: 978-1-4503-9156-6/22/04, Article 332, pp. 1–7, Published by Association for Computing Machinery (ACM), DOI: 10.1145/3491101.3519665, Available: https://dl.acm.org/doi/10.1145/3491101.3519665.

[41] Satya P. Tiwari, Shivam Prasad and Thushara Mg, "Machine Learning for Translating Pseudocode to Python: A Comprehensive Review", in *Proceedings of the 2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS),* 17-19 May 2023, Madurai, India**,** Online ISBN: 979-8-3503-9725-3, Print ISBN: 979-8-3503-9726-0, pp. 274-280, Published by IEEE, DOI: 10.1109/ICICCS56967.2023.10142254, Available: https://ieeexplore.ieee.org/document/10142254.

[42] Fang Liu, Jia Li and Li Zhang, "Syntax and Domain Aware Model for Unsupervised Program Translation", in *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 14 – 20 May, Melbourne, Victoria, Australia, Electronic ISBN: 978-1-6654-5701-9, Print ISBN: 978-1-6654-5702-6, pp. 755-767, Published by IEEE, DOI: 10.1109/icse48619.2023.00072, Available: https://ieeexplore.ieee.org/document/10172589.

[43] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu and Baishakhi Ray, "NatGen: generative pre-training by 'naturalizing' source code", in *Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022),* 14-18 November 2022, Singapore, Singapore, ISBN: 978-1-4503-9413-0, pp. 18–30, Published by Association for Computing Machinery(ACM), DOI: 10.1145/3540250.3549162, Available: https://dl.acm.org/doi/proceedings/10.1145/3540250.

[44] Junaed Y. Khan and Gias Uddin, "Automatic Code Documentation Generation Using GPT-3", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering,* 10 – 14 October 2022, Rochester MI, USA, ISBN: 978-1-4503-9475-8, Article 174, pp. 1–6, Published by Association for Computing Machinery(ACM), DOI: 10.1145/3551349.3559548, Available: https://dl.acm.org/doi/abs/10.1145/3551349.3559548.

[45] Hao Bai, "A Practical Three-phase Approach to Fully Automated Programming Using System Decomposition and Coding Copilots", in *Proceedings of the 2022 5th International Conference on Machine Learning and Machine Intelligence,* 23 - 25 September 2022, Hangzhou, China, ISBN: 978-1-4503-9755-1, pp. 183–189, Published by Association for Computing Machinery(ACM), DOI: 10.1145/3568199.3568228, Available: https://dl.acm.org/doi/abs/10.1145/3568199.3568228.

[46] Meilin Shi, Kitty Currier, Zilong Liu, Krzysztof Janowicz, Nina Wiedemann *et al.*, "Thinking Geographically about AI Sustainability", *AGILE: GIScience Series,* Vol. 4, 2023, ISSN: 2700-8150, Published by Copernicus GmbH, DOI: 10.5194/agile-giss-4-42-2023, Available: https://agile-giss.copernicus.org/articles/4/42/2023/.

[47] Ömer Aydın, "Google Bard Generated Literature Review: Metaverse", *Journal of AI*, Vol. 7, No. 1, 6th June 2023, e-ISSN: 3023-4018, pp. 1-14, Published by İzmir Academy Association, DOI: 10.21541/apjess.1293702, Available: https://agile-giss.copernicus.org/articles/4/42/2023/.

[48] Kishore Papineni, Salim Roukos, Todd Ward and Wei-Jing Zhu, "BLEU: a method for automatic evaluation of machine translation", in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 7 - 12 July 2002, Philadelphia, Pennsylvania, USA, pp. 311–318, Published by Association for Computing Machinery (ACM), DOI: 10.3115/1073083.1073135, Available: https://dl.acm.org/doi/10.3115/1073083.1073135.

[49] Ian Sommerville, *Software Engineering*, 9th Edition, London, UK: Pearson Education Inc., 2010, Available: https://dl.acm.org/doi/10.5555/1841764.

[50] Lennart Ante, "The non-fungible token (NFT) market and its relationship with Bitcoin and Ethereum", *FinTech,* Vol. 1, No. 3, 29th June 2022, ISSN: 2674-1032, pp. 216–224, Published by MDPI, DOI: 10.2139/SSRN.3861106, Available: https://www.mdpi.com/2674-1032/1/3/17.

[51] Leonardo Corbalan, Juan Fernandez, Alfonso Cuitiño, Lisandro Delia, Germán Cásere *et al.*, "Development frameworks for mobile devices: A comparative study about energy consumption", *Proceedings of the International Conference on Software Engineering,* 27-28 May 2018, Gothenburg, Sweden, Electronic ISBN: 978-1-4503-5712-8, Print ISBN: 978-1-5386-6170-3, pp. 191–201, Published by IEEE, DOI: 10.1145/3197231.3197242, Available: https://ieeexplore.ieee.org/document/8543454.

[52] Lisandro Delía, Nicolás Galdamez, Leonardo Corbalan, Patricia Pesado and Pablo Thomas, "Approaches to mobile application development: Comparative performance analysis", *Proceedings of Computing Conference 2017,* 18-20 July 2017, London, UK, Electronic ISBN:978-1-5090-5443-5, Print ISBN: 978-1-5090-5444-2, pp. 652–659, Published by: IEEE, DOI: 10.1109/SAI.2017.8252165, Available: https://ieeexplore.ieee.org/abstract/document/8252165.

[53] Peixin Que, Xiao Guo and Maokun Zhu, "A Comprehensive Comparison between Hybrid and Native App Paradigms", *Proceedings of the 2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, 23-25 December 2016, Tehri, India, Electronic ISBN: 978-1-5090-1144-5 , Print ISBN: 978-1-5090-1145-2, pp. 611–614, Published by IEEE, DOI: 10.1109/CICN.2016.125, Available: https://ieeexplore.ieee.org/document/8082717.

[54] Luis Cruz and Rui Abreu, "Performance-Based Guidelines for Energy Efficient Mobile Applications", *Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 22-23 May 2017, Buenos Aires, Argentina, Electronic ISBN: 978-1-5386-2669-6, Print ISBN: 978-1-5386-2670-2, pp. 46–57, Published by IEEE, DOI: 10.1109/MOBILESOFT.2017.19, Available: https://ieeexplore.ieee.org/document/7972717.

[55] KOvács, Márk and Zsolt Csaba Johanyák, "Comparative Analysis of Native and Cross-Platform iOS Application Development", *Papers on Technical Science*, Vol. 15, 2021, Online ISSN: 2601-5773, Print ISSN: 2610-5773, pp. 61–64, Published by Cluj-Napoca: Erdélyi Múzeum-Egyesület, DOI: 10.33895/mtk-2021.15.12, Available: https://real.mtak.hu/146682/1/10.33894_mtk-2021.15.12.pdf.

[56] Xiaoping Jia, Aline Ebone and Yongshan Tan, "A performance evaluation of cross-platform mobile application development approaches", in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 27 May 2018 – 3 June 2018, Gothenburg, Sweden, Electronic ISBN: 978-1-4503-5712-8, ISBN: 978-1-5386-6170-3, pp. 92-93, Published by IEEE, DOI: 10.1145/3197231.3197252, Available: https://ieeexplore.ieee.org/document/8543442.

[57] Ahmed A. Muhammad, Abdelrahman M. Soliman, Sahar Selim and Ahmed H. Yousef, "Generic Library Mapping Approach for Trans-Compilation", *Proceedings of the 2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, 26-27 May 2021, Cairo, Egypt, Electronic ISBN:978-1-6654-1243-8, Print ISBN: 978-1-6654-2953-5, pp. 62–68, Published by IEEE, DOI: 10.1109/MIUCC52538.2021.9447641, Available: https://ieeexplore.ieee.org/document/9447641.

[58] Christoph Rieger and Tim A. Majchrzak, "Towards the definitive evaluation framework for cross-platform app development approaches", *Journal of Systems and Software,* Vol. 153, July 2019, Print ISSN: 0164-1212, Online ISSN: 1873-1228, pp. 175–199, Published by Elsevier, DOI: 10.1016/J.JSS.2019.04.001, Available: https://www.sciencedirect.com/science/article/pii/S0164121219300743.