Programming Language interoperability in cross-platform software development

Anh Nguyen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 31.8.2022

Supervisor

Dr. Matti Siekkinen

Advisor

MSc. Aleksandr Ovchinnikov

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Anh Nguyen	
Title Programming Language interoperability in cross-platform software development	
Degree programme Master's Programme in Computer, Communication and Information Sciences	
Major Computer Science	**Code of major** SCI3042
Supervisor Dr. Matti Siekkinen	
Advisor MSc. Aleksandr Ovchinnikov	
Date 31.8.2022	**Number of pages** 65 **Language** English

Abstract

Recent years have witnessed the rising popularity of software that are constructed by combining various modules written in different programming languages. While the coexistence of multiple programming languages within the same codebase might bring certain benefits such as reusability and the ability to exploit the unique power of each language, this architecture certainly adds significant complexity to the development and maintenance process of such systems.

This thesis proposes an approach to alleviate the pain of language interoperability in those systems by automating the binding code generation process between different languages. The proposed method uses the metadata extracted from the Interface Description Language (IDL) to systematically generate the Application Programming Interface (API) in each involved language. As a result, the code written in one language can seamlessly interact with code developed in others. The experiment results showed that the developed code generator has improved the stability, scalability, and modularity of multi-language software systems.

Keywords Programming language interoperability, compiler, Interface Description Language, cross-platform SDK, API, C++, Java Native Interface, Objective-C++

Acknowledgements

The time that I spent pursuing my Master's degree at Aalto University was the most challenging yet transformative and rewarding years of my life. As all good things must come to an end, so do my studies at Aalto. This thesis marks the end of my student life but it also unfolds a new chapter full of adventures in my lifelong learning journey.

I would like to thank Mapbox for giving me an opportunity to work on amazing challenges that impact the daily life of millions of people. I am especially grateful for the guidance, assistance, and constructive feedback from my thesis advisor Aleksandr Ovchinnikov, my manager Bruno Abinader, Young Hahn, and all my colleagues at Mapbox. This thesis would not exist without you all. It would be an oversight if I forgot to thank Peng Liu for his open-sourced LATEX thesis template that I "borrowed" so that I could whip up a well-formatted thesis over the weekend with zero LATEX knowledge. Hopefully, there are no remaining parts of his thesis that were not replaced.

I would like to express my gratitude to my supervisor Dr. Matti Siekkinen from Aalto University for his valuable advice, feedback, and patience along the journey.

Thanks should also go to my friends who let me use their laundry detergent and toothpaste for the whole year without a clue, gave me free food occasionally, and continuously motivate me to push my boundaries. You know who you are.

Most importantly, I would like to thank my family for supporting me unconditionally during all these years.

Otaniemi, 31.8.2022

Anh Nguyen

Contents

List of Figures

List of Tables

Abbreviations

REST	Representational State Transfer
RPC	Remote Procedure Call
FFI	Foreign Function Interface
API	Application Programming Interface
ABI	Application Binary Interface
ML	Machine Learning
I/O	Input/Output
IR	Intermediate Representation
JNI	Java Native Interface
SDK	Software Development Kit
IDL	Interface Description Language
JVM	Java Virtual Machine
CPU	Central Processing Unit
AST	Abstract Syntax Tree

1 Introduction

Modern software has become increasingly complex during the past few decades. As a result, the use of a single programming language for every task has repeatedly shown its limitations. Alternatively, modern software is the composition of multiple components, each of which can be written in a different programming language.

There are several reasons behind the proliferation of the multi-language paradigm. One of the most important advantages of this architecture is that it allows software engineers to take advantage of the language that is best suited for each individual task. For instance, the majority of Machine Learning (ML) frameworks offer first-class Application Programming Interfaces (APIs) in Python programming language[1] even though they are often written in other languages [1]. In fact, Python is considered the language of choice within the data science community due to its simplicity, high-level abstractions, and the vast number of useful data-oriented libraries such as NumPy[2] and Matplotlib[3] in the Python ecosystem. The gentle learning curve and the dynamic nature of Python allow scientists to conduct experiments and iterate on their ideas swiftly. Nevertheless, the computational performance of Python has always been a known weakness of the language. To tackle the lack in performance of Python, ML frameworks such as Tensorflow[4] are often written in highly performant yet low-level and complex programming languages such as C++[5]. Within these frameworks, the Python API layer is merely a shim that internally invokes performance-critical code written in C/C++ [1]. As a result, combining both high-level and low-level programming languages often brings the best of both worlds to those multi-language systems.

Additionally, the higher number of languages involved within multi-language software means that the software may gain access to a broader set of existing libraries written in each involved language. As a result, engineers can reduce the development cost by reusing existing well-tested software modules without reinventing the wheel. Besides that, the coexistence of different languages within the system allows gradual migration of legacy software modules from one language to another, which often reduces the migration risks as the existing implementation does not have to be ported to the new language all at once [2].

1.1 Problem statement

As the number of languages involved in the system grows, so does the number of challenges in developing and maintaining such software systems. In fact, the task of combining multiple languages within the same software system has never been

[1]Python programming language. https://www.python.org. Accessed 04/2022.
[2]Numpy. https://numpy.org. Accessed 04/2022.
[3]Matplotlib. https://matplotlib.org. Accessed 04/2022.
[4]Tensorflow. https://www.tensorflow.org. Accessed 04/2022.
[5]C++ programming language. https://www.cplusplus.com/info/faq. Accessed 04/2022.

trivial. One such challenge is to design the interface between different languages in a reliable and scalable way. There exist multiple approaches to achieving programming language interoperability. However, the scope of this thesis is limited to only Foreign Function Interface (FFI) - a mechanism that allows interactions between languages by invoking functions across the boundaries of those languages [3]. While the FFI mechanism provides a way to interface with different languages, FFI glue code tends to be complex, verbose, error-prone, and repetitive on a large scale [3][4]. Thus, this type of code can quickly become problematic to write and maintain in substantial codebases. Consequently, software that uses FFI without extra care often becomes a source of defects and security problems.

This thesis aims to design and implement a code generator that automates the process of producing FFI glue code. The solution is expected to improve the productivity, stability, and scalability of multi-language systems. Hence, the developed tool and technical insights gained during the development should be helpful for the software community who are involved in the development and maintenance process of large-scale cross-language software systems.

It is important to note that the project is a collaborative effort between many engineers from different teams at Mapbox. Even though the thesis's author is the main maintainer of the project at the time of writing this thesis, the project was initiated earlier by other engineers. The contribution of the thesis' author to the project includes:

- Developing new generators from scratch for three languages
- Extending the functionality of the current generators.
- Improving the project's performance, stability, and maintainability by establishing test and benchmark infrastructure, optimizing, and fixing defects in the generated code.

1.2 Research questions

To evaluate the effectiveness of the solution, we seek to answer the following research questions throughout the thesis:

- **Does automatic bindings generator solve productivity problems within cross-language software development?** Does it improve the quality of the bindings code? Does it enhance the developer experience and productivity? Does it promote the modularity of cross-language software?
- **What are the limitations of automatic bindings code generation?** Is there any regression in terms of run-time performance and ergonomics of the generated API?

- **How extensible and scalable is the code generator solution?** How easy is it to support new features and new languages? What other types of code can it generate apart from language bindings?

1.3 Structure of the thesis

This thesis is divided into 8 chapters. Chapter 2 provides background information on the cross-language software development paradigm and language interoperability technologies. This chapter also discusses the problems of language bindings. Chapter 3 describes the methodology that was used in the project. Chapter 4 proposes solutions to alleviate those weaknesses. Chapter 5 reviews the end-to-end implementation of the project. Chapter 6 summarises the evaluation results of the project. Chapter 7 discusses challenges and future directions. Lastly, chapter 8 concludes the thesis.

2 Background

This chapter aims to provide a broad background of the problem that this thesis project attempts to solve. The first two sections offer a brief overview of programming language interoperability in a broad context. Subsequently, the remaining parts narrow the scope of the topic to only relevant industrial context while diving into more in-depth details.

2.1 Programming language interoperability

Programming language interoperability is the ability of multiple programming languages to collaborate within the same software system. This mechanism has played a crucial part in various systems as each language is designed to tackle specific problem domains but offers limited support in other areas [5]. JavaScript[6], for instance, is originally the scripting language that runs on web browsers to add user interactions to web pages. However, it has gained substantial popularity on the server side over the years thanks to the emergence of Node.js[7] - a Javascript runtime environment that executes JavaScript code outside web browser environments. JavaScript applications running on Node.js servers became increasingly viral since it allows engineers to use the same programming language on both client and server sides.

However, the Node.js application has its own limitations. JavaScript uses a single-threaded, event-driven, and non-blocking Input/Output (I/O) model as it is specifically designed to handle the Web User Interface (Web UI) [6]. This means that a Node.js server cannot handle concurrent requests while other tasks are blocking its event loop. As a result, JavaScript code running on Node.js servers is not suitable for handling synchronous CPU-intensive tasks due to the lack of proper multi-threading support.

Nevertheless, there are multiple approaches to tackle this problem, thanks to the interoperability between different programming languages. One solution is to delegate CPU-intensive tasks to a different server written in other performant, multithreaded languages such as Rust[8] programming language. With this approach, the Node.js server acts as a proxy service that receives the requests from the client-side and then forwards them to the newly created web service written in Rust. By avoiding direct execution of such blocking tasks, the Node.js application can handle multiple concurrent requests as it is merely responsible for asynchronous I/O operations, which do not block the event loop. Another alternative is to leverage Node C++ Addons[9] to interact with performant C++ modules within the same process that the Node.js server runs. This approach allows JavaScript code to offload the computation

[6]JavaScript programming language. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed 04/2022.

[7]Node.js run-time environment. https://nodejs.org/en/about. Accessed 04/2022.

[8]Rust programming language. https://www.rust-lang.org. Accessed 04/2022.

[9]Node C++ addons. https://nodejs.org/api/addons.html. Access 04/2022.

to C++ worker threads in the thread pool. Hence, the JavaScript main thread can be unblocked to handle other incoming requests. More details of these approaches are discussed throughout this chapter.

2.2 Categories

Various solutions for language interoperability have been introduced over the years. They can typically be categorized into two groups based on their communication style: inter-process and in-process. This thesis discusses several common approaches on a superficial level but dives deeper into more details where the concepts are relevant to the project.

2.2.1 Inter-process approaches

These methods typically leverage I/O streams to exchange data between components written in different languages. In these approaches, one component informs others what needs to be done by sending data messages. Usually, the sender and the receiver follow a set of predefined protocols that dictate the format of the input, what action should be performed, and the structure of the output. Those components could be running on the same machine or spanning across multiple machines in the network. This thesis briefly discusses two common approaches under this category: Representational State Transfer (REST) and Remote Procedure Call (RPC).

- **REST**

  REST stands for Representational State Transfer, the current de-facto architecture for delivering web services based on the traditional client-server model [7]. Essentially, there are two main concepts behind REST: resources and actions. REST APIs provide clients access to a set of resources that reside on the servers. Those resources are uniquely identified by Uniform Resource Identifiers (URI) - a string of characters that includes the location and name of each resource. To interact with a specific resource, the clients typically transfer messages over the Hypertext Transfer Protocol (HTTP) to the URI address of the resource while using the predefined HTTP verbs (`GET, POST, PUT, DELETE`) to select the kind of operation that must be performed on the chosen resource [7].

  For example, the resource representing Mapbox fonts is accessible at the URI `https://api.mapbox.com/fonts/v1/{username}`. Clients can get a list of font names by sending the `HTTP GET` request to the mentioned address. A new font can also be added by sending the `HTTP POST` request that includes the binary data of the font in the payload. Besides that, the font “Helvetica” can be removed from the list by sending the `HTTP DELETE` request to the address `https://api.mapbox.com/fonts/v1/{username}/Helvetica`.

  REST mechanism promotes the separation of concerns between the client and the server as the interaction is achieved solely by exchanging messages.

This allows the client and the server program to be implemented in different languages as long as they both adhere to a set of defined protocols.

- **RPC**

  Remote procedure call is the mechanism that enables computer programs to trigger the execution of procedures located in a different address space such as another process on the same machine or different machines within the network [8]. RPC applies the concept of transferring control and data, which occurs during the local procedure call, to the distributed network of processes. While REST is a resource-oriented mechanism, RPC focuses primarily on actions. When the caller process invokes a local procedure, it packs the parameters into a message and forwards it to the remote procedure over the network socket. Then the caller process waits for the result to be returned by the remote procedure once the execution has finished. RPC is typically designed so that the implementation details such as the location of the subroutine being invoked are transparent to the caller, which means that the caller program invokes a remote subroutine as if that subroutine is a natural part of that program. As this mechanism allows sharing the computation workload with other machines, it is commonly used to construct distributed systems [9].

  The RPC client (known as the caller) and the server (known as the callee) programs can be written in completely different programming languages. However, each language has different method calling conventions as well as different data structure representations of parameter types. To achieve language interoperability between the client code and server code, namely making the remote procedure call look and feel as if it is a local one, RPC introduces a third language known as Interface Definition Language (IDL) to describe the interface of the remote procedure in a language-agnostic way [9]. Indeed, the first step to working with RPC is to define the data structures of parameters and interface function definitions in the IDL. Once the procedure interface is described in the IDL, an IDL compiler can be used to parse the IDL and generate the stub code on both the client-side and server-side.

  The end-to-end RPC invocation workflow fundamentally consists of 9 steps as shown in Figure 1.

  The local procedure is first invoked on the client-side (1). Then its parameters are packed into a message (2) following a specific data format before being sent to the server (3). On the server side, the received messages are unpacked by the stubs code (4). Next, those unpacked parameters are passed as inputs to the procedure where the real computation occurs (5). After that, the returned result of the procedure is serialized into a response message (6) that is subsequently sent back to the client (7). On the client-side, the message is deserialized by the generated stub (8) and returned as a function result to the caller (9).

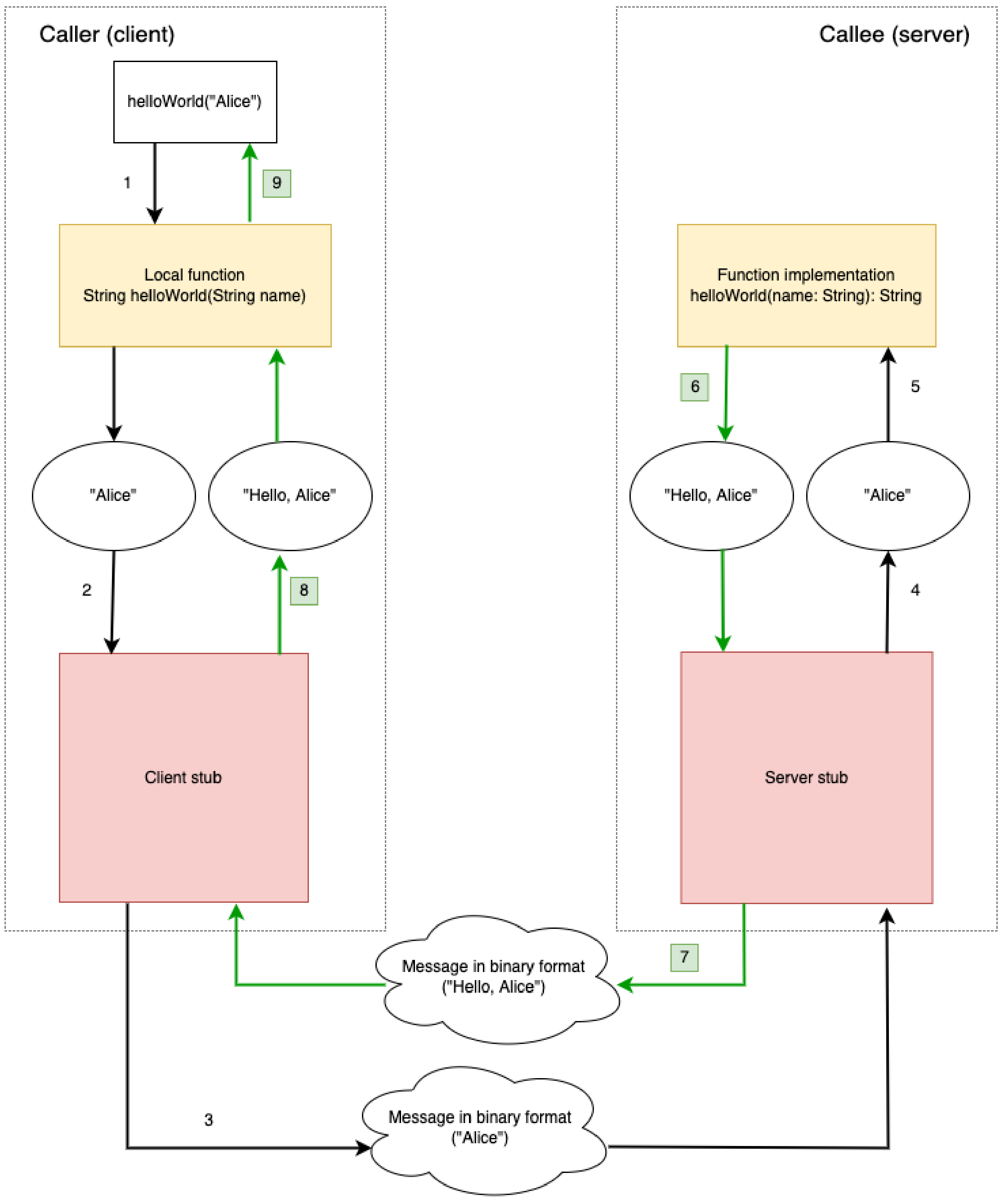


Figure 1: RPC end-to-end workflow.

One concrete example of the RPC mechanism is Google Remote Procedure Call [10] (gRPC) - a popular modern RPC framework that uses Protocol Buffers [11] as both its IDL and its data serialization mechanism. The first step to using gRPC is to describe the data structures and interface methods using the

[10]Google Remote Procedure Call framework. https://grpc.io/about. Accessed 04/2022
[11]Prococol Buffers. https://developers.google.com/protocol-buffers. Accessed 04/2022

Protocol Buffers IDL syntax (proto3) as shown in Listing 1.

```
syntax = "proto3";

package helloworld;

// The greeting service definition
service Greeter {
  // Sends a greeting
  rpc HelloWorld (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Listing 1: Interface definitions written in Protocol Buffers IDL

Next, the IDL file can be compiled with the Protocol Buffers compiler (protoc) to generate stub code on both the client and server languages. After that, the implementation of the remote method on the server-side must be provided as depicted in Listing 2.

```cpp
class GreeterServiceImpl final : public Greeter::Service {
  Status HelloWorld(
    ServerContext* context,
    const HelloRequest* request,
    HelloReply* reply) override {
    // user-provided implementation
    reply->set_message("Hello, " + request->name());
    return Status::OK;
  }
};
```

Listing 2: The implementation of the remote procedure in C++

Finally, the client code in Java[12] can invoke the remote procedure as if it is a local method call as depicted in Listing 3.

```java
HelloRequest request = HelloRequest.newBuilder().setName("↩
    Alice").build();
HelloReply response;
try {
  response = Greeter.helloWorld(request);
  logger.info(response.getMessage());
} catch (StatusRuntimeException e) {
  logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus())↩
    ;
}
```

Listing 3: The client code in Java that calls the remote procedure

2.2.2 In-process approaches

These solutions are typically used to provide interactions between different software modules running within the same OS process. They often rely on the low-level support of the compiler or language runtime environment to interact with another language. This thesis examines two common approaches: Common Intermediate Representation (IR) and Foreign Function Interface (FFI).

- **Common Intermediate Representation**

  Computer programs written in high-level languages are often compiled to machine code that the running hardware can understand. However, each processor architecture has its own type of assembly instruction set. Thus, to run directly on $\boldsymbol{K}$ different architectures, the program written in a source language must be compiled to $\boldsymbol{K}$ types of assembly language. This means that it requires

[12]Java programming language. https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html. Accessed 04/2022

H* * *K direct translations to compile ***H*** languages to target ***K*** architectures, as shown in Figure 2. The huge number of direct translations often indicates poor reusability and scalability.

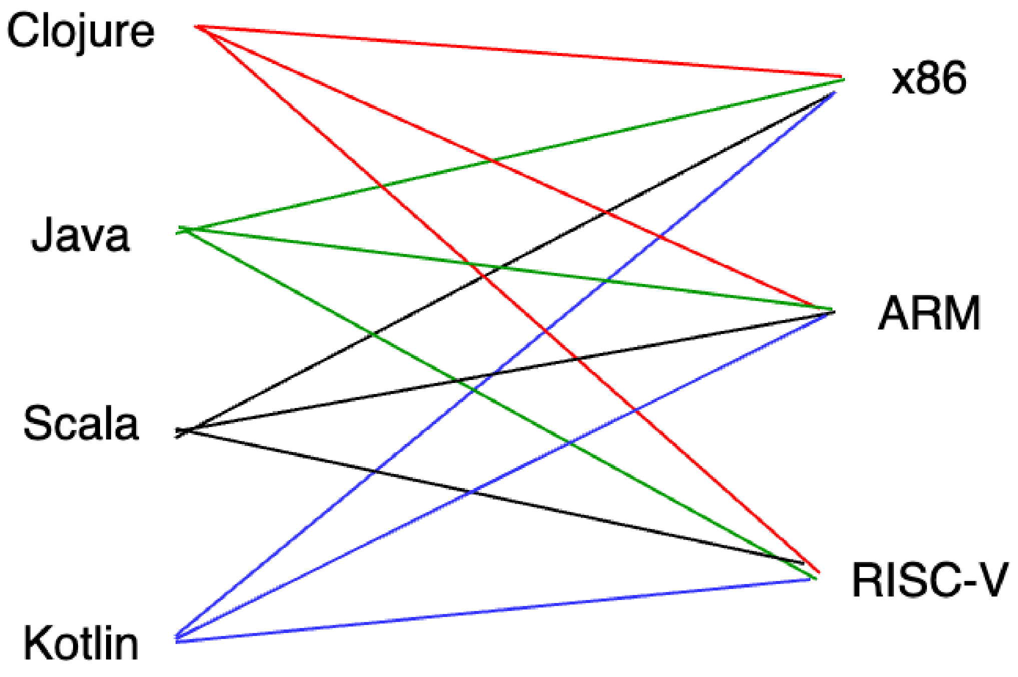


Figure 2: Direct language translation graph

To solve this problem, the majority of compilers introduce a third language as an Intermediate Representation (IR) [2]. This "middle" language is independent of the source language and the target language, and it must be designed in a way to abstract the assembly code of the target architecture while still having the capability to capture the necessary metadata of the source language [10]. By compiling ***H*** languages to the same IR language and subsequently compiling the IR language to ***K*** types of assembly language, we can significantly reduce the number of implementations from ***H* * *K*** down to ***H* + *K***, as depicted in Figure 3.

Software components written in different programming languages that are compiled to the same intermediate language can interact with each other as long as they follow the same Application Binary Interface(ABI) contract. This contract dictates several low-level aspects such as method calling conventions, data layout, and name mangling that the compilers of different languages must conform to in order to generate interoperable low-level code [11].

One notable example in this category is that languages running on Java Virtual Machine can interact easily with Java. For example, Kotlin[13] can interop

[13]Kotlin programming language. https://kotlinlang.org/. Accessed 04/2022.

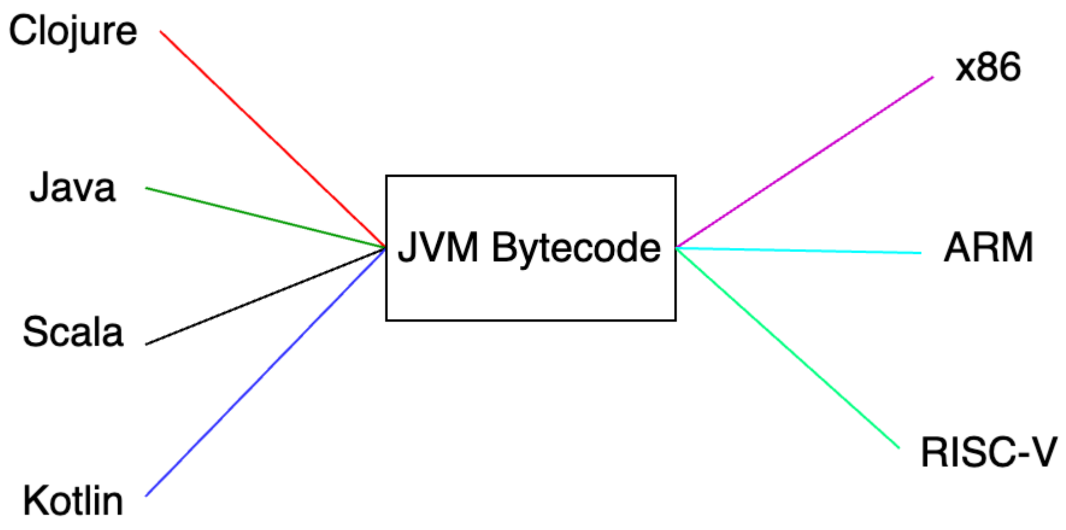


Figure 3: Language translation with shared IR on JVM

seamlessly with Java for two main reasons. Firstly, they are both compiled to Java bytecode - the shared IR of Java Virtual Machine (JVM). Secondly, the compiled Java bytecode of Kotlin and Java follows the same method calling conventions which specify the label names of the methods, how input parameters are structured and passed to the callee, and how to access the returned value of the callee method.

- **Foreign Function Interface**

  Foreign Function Interface (FFI) is a low-level programming language interface that allows code written in one programming language (known as the host language) to access objects and invoke functions that reside in a foreign language across the boundaries of the two languages [2]. Typically, FFI APIs, which are often exposed by the runtime environment of the host language, are used in a layer of bridging code to interact with the foreign language. This FFI code is often referred to as the glue code because it acts as the bridge connecting a pair of programming languages. The primary purpose of the FFI APIs is to allow high-level programming languages to interact with low-level C/C++ code [2]. Essentially, the runtime environment of the host language loads the dynamic native library into memory, and then it maps the set of the method declarations in the host language to the function pointers of the native library using the method `dlsym()`. When a function is called in the host language, the runtime converts the parameters to suitable data structures in the foreign language [12]. Next, it forwards the call to the corresponding native function, given that the runtime environment of the host language knows the ABI of the foreign language. Some of the notable FFI technologies include Java

Native Interface[14] (Java - C/C++ interop), Dart FFI[15] (Dart - C interop), and Node Native Addons (Node.js - C/C++ interop).

For example, Java Native Interface (JNI) is an FFI API that allows code written in Java to interact with native C/C++ code and vice versa. The first step to interact with C++ code via JNI is to load the native dynamic library `greeter.so` with `System.loadLibrary()` method as shown in Listing 4. Next, the native method must be declared using the modifier `native` to indicate that it is implemented on the native side. When the method `helloWorld()` is called on the Java side, the JVM searches for the symbol of the native method `Java_Greeter_helloWorld()` in the loaded library and forwards the call to the native implementation. In order to access data structures and methods from the Java side, C/C++ code can invoke JNI APIs[16] on the JNI parameters passed as input to the native function, as shown in Listing 5.

```java
public class Greeter {
  public native void helloWorld(String name);

  static {
      System.loadLibrary("greeter.so");
  }

  public static void main(String[] argv) {
      Greeter greeter = new Greeter();
      greeter.helloWorld("Foo");
  }
}
```

Listing 4: Java program that uses JNI

```cpp
JNIEXPORT void JNICALL Java_Greeter_helloWorld
  (JNIEnv *env, jobject thisObj, jstring name)
{
    std::cout << jStringToCString(env, name) << std::endl;
}
```

Listing 5: C/C++ implementation of the method helloWorld()

2.3 Industrial context

This section aims to provide the concrete industrial background of the thesis project. The first subsection introduces the industrial context where the multi-language scheme

[14] Java Native Interface. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html. Accessed 04/2022.

[15] Dart FFI. https://dart.dev/guides/libraries/c-interop. Accessed 04/2022

[16] JNI specifications. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html. Accessed 04/2022

is applied. The second subsection discusses the programming language choices and software architecture. Finally, the third subsection highlights the problems that are commonly encountered in the bridging code.

2.3.1 Cross-platform library

Cross-language software development is a broad topic. However, the scope of this thesis is limited to only cross-platform software library development. The case study was provided by the company Mapbox in 2022. Mapbox is a location technology provider that offers flexible and powerful tools for developers and designers to embed interactive maps, geocoding, turn-by-turn navigation, route optimization, and data visualization in their applications [13]. The solutions offered by Mapbox have been powering two million developers, with over 600 million users worldwide touching Mapbox maps every month. In particular, the case study focuses on the development and maintenance process of various Mapbox Software Development Kits (SDKs) available on platforms such as Android, iOS, and embedded Linux.

One approach to bringing SDK support to multiple platforms is to concurrently develop an SDK in multiple programming languages that are natively supported by each platform. For example, Java and Kotlin are first-class languages on Android[17] while Objective-C[18] and Swift[19] are the best-supported ones on iOS[20]. Other than that, C and C++ are often the first choices when it comes to embedded Linux. The main benefit of building software using the first-class language on a particular platform is the full support of the ecosystem. It means that developers can gain full access to the APIs, libraries, and documentation that the platform provides. Additionally, software can be specifically designed and fine-tuned to target the selected platform. As a result, the software's performance and user experience benefit vastly from the first-class support provided by the programming language. However, writing the same software in different languages is labor-intensive and often unscalable. Besides that, the parity of the software written in different languages tends to diminish over time as different languages and platforms tend to impose different practices and conventions. As a result, the development and maintenance costs would multiply when it comes to porting a set of different SDKs to various platforms.

On the other side of the spectrum is the approach in which software is written in a single programming language with the ability to run on all platforms. This approach prioritizes the reusability and cost-efficiency of the development process because a small development team can write the software once and run it everywhere. For example, programs written in languages such as C++ can be compiled and run performantly on a wide range of platforms. As a result, User Interface applications

[17]Android operating system. https://www.android.com/what-is-android. Accessed 04/2022.

[18]Objective-C programing language. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html. Accessed 04/2022.

[19]Swift programming language. https://developer.apple.com/swift. Accessed 04/2022.

[20]iOS operating system. https://www.apple.com/ios/ios-15. Accessed 04/2022.

can be built using only C++ with Qt[21] framework and run on both Android, iOS, and Linux. Nevertheless, cross-platform software tends to suffer from the limited support of the platform. This often leads to the disparity in the user experience of cross-platform applications compared to native applications. In addition, not every native API is available when using cross-platform frameworks, and it takes non-trivial efforts to expose those native APIs to cross-platform frameworks.

Mapbox cross-platform SDKs take a hybrid approach by combining the cross-platform C++ codebase with platform native code such as Java, Objective-C, and Node.js to get the best of both worlds. More in-depth technical details of the setup are discussed in the following sections.

2.3.2 Multi-language software architecture

Mapbox selected C++ as a primary language to share the business logic across platforms for several reasons. Firstly, C++ is one of the most performant cross-platform languages that excels at executing CPU-intensive tasks. In fact, C++ gives complete control over the memory allocation and management, which is the key that helps the map rendering engine achieve a high frame rate with minimal memory footprint. Efficient memory usage is especially critical when running on low-end embedded hardware with limited memory. Apart from the manual memory management scheme, C++ is a large and complex language that offers full access to low-level system APIs while developers can still express their ideas using high-level abstractions in C++ with minimal overhead. As a result, C++ gives developers fine-grained control over their program, which is essential to squeeze every last bit of performance from the hardware. Secondly, C/C++ is commonly used for low-level graphics programming, which is the core of Mapbox's map rendering engine. By using C/C++, the map rendering engine can easily access low-level graphic APIs offered by libraries such as OpenGL[22].

The low-level native C++ libraries at Mapbox are exposed to developers via layers of wrapper code written in native mobile languages such as Java/Kotlin on Android and Objective-C/Swift on iOS. These layers of bindings serve two main purposes. Firstly, they allow APIs written in C++ to be exposed in an idiomatic, transparent fashion to mobile developers who typically write their code in native mobile languages. Secondly, the interoperability between C++ and native mobile languages allows the cross-platform C++ code to leverage the existing libraries and platform-dependent APIs written in Java/Objective-C. For instance, both Mapbox C++ rendering engine `mapbox-gl-native`[23] and `Mapbox Android SDKs` use the same Java networking library named OkHttp[24] on the Android platform. By reusing the library `OkHttp` on both Java and C++ sides, Mapbox is able to reduce not only

[21]Qt framework. https://www.qt.io. Accessed 04/2022.
[22]OpenGL website. https://www.opengl.org. Accessed 04/2022.
[23]Mapbox-gl-native repository. https://github.com/mapbox/mapbox-gl-native. Accessed 04/2022
[24]OkHttp networking library. https://square.github.io/okhttp. Accessed 04/2022.

their engineering efforts but also the binary size of the final product because `OkHttp` has already been a dependency of Android SDKs.

Those wrapping layers contain a large portion of bindings that connect the mobile platform languages with the C++ codebase. To create an interface between C++ and Android/Kotlin, Mapbox SDKs leverage Java Native Interface (JNI), which was briefly introduced in the previous section. On iOS, language interoperability was achieved by a hybrid language named Objective-C++. Objective-C++ is the superset of both Objective-C and C++ as it allows mixing Objective-C and C++ code in the same source file. The overall architecture is depicted in Figure 4.

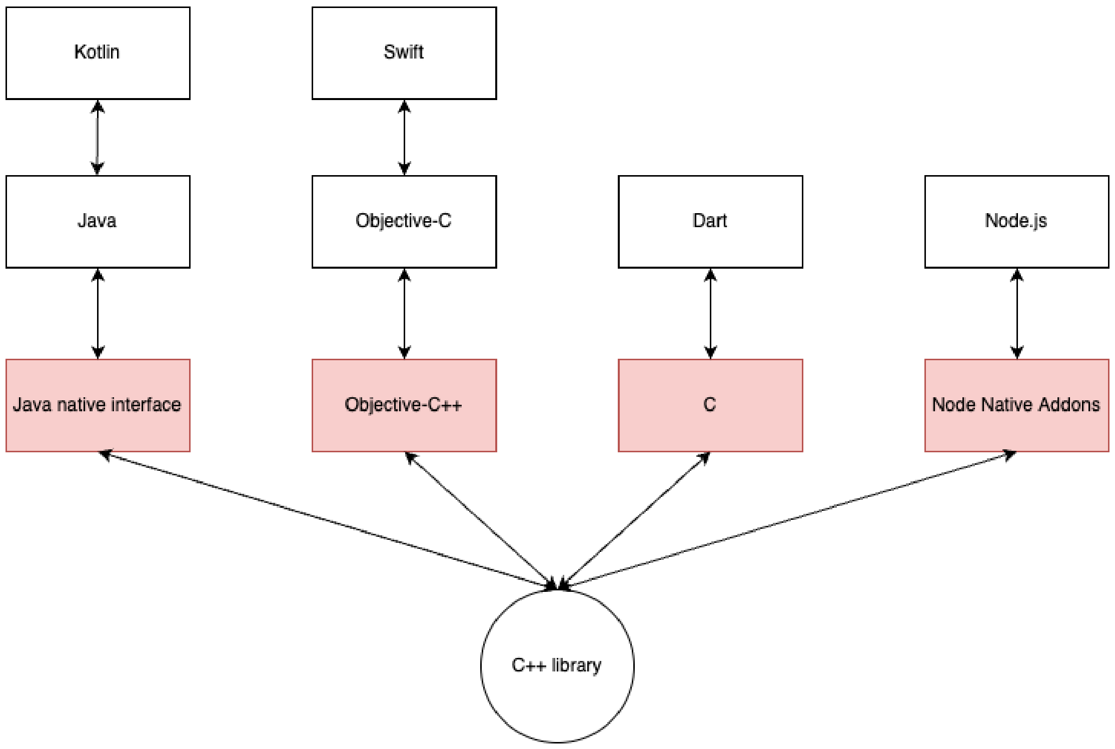


Figure 4: Cross-platform architecture

2.4 Known issues in FFI code

Despite the fact that Mapbox can share the same C++ codebase between different platforms, making native languages on these platforms interop seamlessly with C++ is not a trivial task. The two biggest challenges are the reliability and scalability of the FFI solutions.

Conceptually, the process of exposing an already existing C++ API to mobile platforms consists of 3 steps. Firstly, a stub class is declared in the platform languages. This class may contain several instance variables and methods so that it can be used transparently in the platform language as if there exists no foreign implementation.

Secondly, the glue code to convert data types between two languages must be provided in the body of those stub methods. Those data type conversions typically use FFI APIs such as Java Native Interface and Objective-C++ since these APIs offer access to foreign objects and methods. Finally, the converted parameters are passed to the foreign function where the actual implementation is located. If the foreign function produces a returned output, that value must be converted to the host language before being returned from the stub method. Essentially, each supported language requires its own marshaling implementation. This process is visualized in Figure 5.

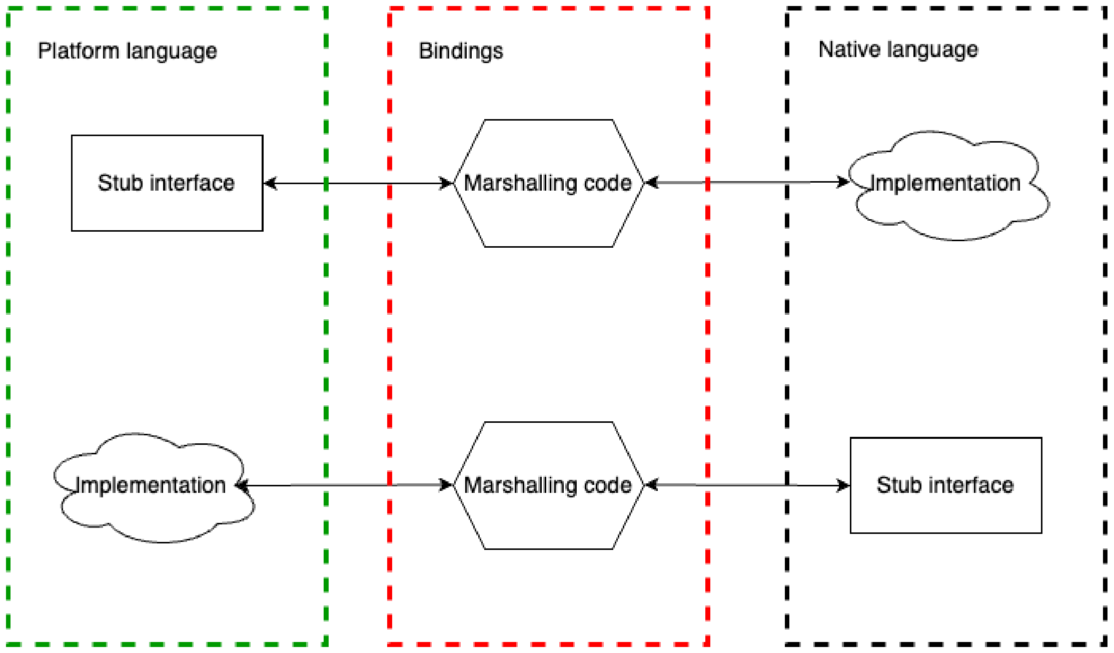


Figure 5: The process of exposing APIs in foreign language

This layer of bindings code can be particularly problematic due to several weaknesses of language interoperability:

2.4.1 Weak compile-time type safety

The majority of FFI technologies typically operate at run-time instead of compile time. As a result, the bindings between two languages can usually be considered as weakly typed code. Java Native Interface, for instance, is the FFI mechanism that allows two statically typed languages - Java and C/C++ to interact with each other. Unfortunately, there exist severe type-safety limitations in JNI code as C/C++ code interacts with Java objects and classes via the reflection mechanism. Reflection allows C++ code to dynamically create Java objects, call Java methods, and access object fields at run-time using their identifiers. While the reflection scheme provides a flexible way to interact with Java from C++, it can easily become error-prone due to the lack of static type analysis at compile time [14]. This means exceptions are only thrown at run-time when the reflected Java function calls are executed rather than at compile time. As a result, programmers often fail to detect their programming errors until their programs crash while being used by the end-users.

To understand how brittle the Java program using JNI can be, it is important to examine an example as depicted in Listing 6, and 7.

```java
public class Greeter {
  public static native void helloWorld();

  public static void invokeThisFunctionInC(long number) {
      System.out.println("long from C++: " + number);
  }

  static {
      System.loadLibrary("greeter.so");
  }

  public static void main(String[] argv) {
      Greeter.helloWorld();
  }
}
```

Listing 6: Java program using JNI

```cpp
JNIEXPORT void JNICALL Java_Greeter_helloWorld
  (JNIEnv *env, jclass klass)
{
    static jmethodID& javaMethod = env->GetStaticMethodID(
      klass,"invokeThisFunctionInC", "(J)V");
    if(javaMethod == nullptr) {
      std::cerr << "ERROR: method void invokeThisFunctionInC() ↩
   not found !" << std::endl;
      return;
    }
    env->CallStaticVoidMethod(cls2, javaMethod, 100L);
    if (env->ExceptionCheck())
    {
        env->ExceptionClear();
        std::cerr << "ERROR: Error is thrown when invoking the ↩
   Java method invokeThisFunctionInC() " << std::endl;
    }
}
```

Listing 7: C++ implementation that calls Java methods

If there is a typo when looking up the Java method `invokeThisFunctionInC()` using the name string `"invokeThisFunctionInC"` or a typo in the method's input/output signature string `"(J)V"`, the error is only detected when the body of the C/C++ function is executed. Similarly, errors can be detected only at run-time if the method invocation `env->CallStaticVoidMethod(cls2, javaMethod, 100L);` is provided with the wrong parameter value or the wrong number of parameters. Those scenarios can easily occur if the methods are modified without the synchronized updates in other parts of the codebase. As a result, the lack of compile-time checking for glue code can severely affect the quality of cross-platform software.

2.4.2 Complexity

Writing bridging code between 2 languages requires comprehensive knowledge of both languages. For example, programmers must deal with three types of bugs: bugs from the host language, bugs from the foreign language, and bugs from the bridging code when two languages interact. It is crucial to take into consideration the vast differences between languages to write bug-free, memory-efficient, and performant glue code. This thesis highlights several key differences between C++ and the platform languages used by Mapbox.

- **Memory management**

  Java, Dart[25], and JavaScript are all automatic garbage-collected languages. It means that their automatic memory management systems take responsibility for allocating memory for objects and deallocating them when those objects are no longer needed. Hence, programmers do not need to manage memory on their own. On the other hand, C/C++ typically does not have an automatic garbage collector, so developers need to handle memory management on their own. The disparities between automatic and manual memory management must be taken into consideration when writing code that connects a garbage-collected language and C/C++. Hence, programmers must determine which resource should be explicitly deallocated as not all objects are automatically garbage-collected when mixing both types of languages. For example, global references to Java objects, C pointers to Java String, or Java arrays in JNI must be manually freed to avoid memory leaks [15]. As a result, programmers must not assume that the garbage collector automatically manages all the resources.

- **Concurrency**

  The disparities in the concurrency models between different languages could also pose significant challenges when producing glue code. Many languages, such as C++ provide support for multithreading. It means that the running process can create multiple threads of execution that run the code concurrently while sharing the process's resources. Multithreading is widely used for performance reasons as this mechanism can allow running computation in parallel on multiprocessors. In contrast, some languages such as JavaScript use the single-threaded concurrency model, which means that the code section written in those languages can only be executed one statement at a time. Interfacing between a single-threaded language and a multithreaded language can be challenging in the situation where a worker thread in the multithreaded language invokes a callback written in the single-threaded language. One common use

[25]Dart programming language. https://dart.dev. Accessed 04/2022.

case is when a Node.js program invokes a long-running task that is executed on a C++ worker thread. During the execution of the task, the Node.js program needs to receive the progress status update in the form of callbacks. This requires the worker thread to invoke callbacks written in JavaScript, which is obviously forbidden because JavaScript code can be executed only on the main thread. As a result, a special implementation technique is needed to bridge the gap between 2 opposite concurrency models.

- **Exception handling**

  Another common problem when mixing two programming languages is the differences in the exception handling mechanism. Programmers often have to deal with two types of exceptions: one in the host language and another in the foreign language. Unfortunately, those two exception handling mechanisms can differ significantly, often requiring additional efforts to ensure exceptions are handled properly. For example, Java requires methods that can throw exceptions marked with the `throws` keyword, which provides the Java static analysis tools with information to force the caller to handle exceptions when compiling Java code. However, the Java compiler cannot statically check if the native C/C++ methods called via JNI throw any exception [15]. Hence, the caller in Java often misses handling native exceptions when invoking native methods, which might cause the application to crash unexpectedly. In addition, the different exception handling behaviors in JNI APIs can pose a huge challenge when creating the bridge between Java and C/C++. In fact, C/C++ code can throw Java exceptions using JNI exception APIs such as `env->ThrowNew()`. However, raising Java exceptions in C/C++ code via JNI does not automatically stop the execution of the C/C++ methods. This behavior can surprise both Java and C/C++ developers as exceptions in their languages typically stop the current execution flow [15]. Thus, programmers might not be aware that the remaining code of their C++ function is still executed even after the Java Exception is thrown in JNI. Hence, this behavior can easily create bugs and security problems if mishandled.

- **Lack of separation of concerns and poor scalability**

  Writing bridging code between two languages requires comprehensive knowledge of both languages. This means that C++ engineers often have to write code in Java/Objective-C while using JNI or Objective-C++, which may not be their expertise. Likewise, mobile developers tend to have limited knowledge of C++ as it is often overly complex and low-level compared to Java or Objective-C. Hence, it is often not clear who should be responsible for establishing the bindings between C++ and native mobile languages. One possible answer would be to hire experts who are competent in both languages and have good knowledge of language interoperability tools such as Java Native Interface, Objective-C++, Dart FFI, and Node Native addon. Unfortunately, the pool of engineers who would fit under this category is relatively small. Moreover, not every engineer in that group is comfortable with all the pairs of languages. For example, an engineer who is familiar with Objective-C++ might not feel comfortable dealing with Java Native Interface or Dart FFI.

There can be certain costs associated with developing and maintaining language bindings on a large scale. Whenever a new API is introduced in the C++ library, the engineering team has to write bindings to expose it to all the supported platforms as well. As the requirements of the software change constantly over time, the APIs of the C++ library along with their platform bindings need to be modified accordingly to reflect the changes. This can lead to huge development and maintenance costs as the number of supported platforms might grow over time. One example is that an organization is responsible for `X` numbers of C++ libraries, each of which has on average `Y` APIs and is exposed to `Z` platforms. This means that the organization needs to manage bindings in `X * Y * Z` places, which can easily become a huge source of bugs and security problems without extra attention.

3 Methodology

After having identified the challenges of the FFI mechanism in the previous chapter, this chapter aims to establish the requirements to be satisfied by the thesis project. The process can be divided into two main steps: collecting the requirements and analyzing those requirements to come up with usable solutions. Section 3.1 describes the requirements collection methodology. After that, section 3.2 covers the requirements analysis process. Finally, section 3.3 depicts the software design and execution.

3.1 Requirements collection

The first step to gathering requirements for this project is to identify stakeholders who might become potential customers of the product to be developed. In this case, the stakeholders are C++ engineers, platform SDK engineers, and engineering managers of all the teams in charge of cross-platform SDKs within Mapbox. The next step is to interview the representatives within each group to gain insight into the current system's status, their problems related to language bindings, and their vision of the desired system. Those interviews were conducted via online meetings and direct messages on an internal communication platform. Next, stakeholders' inputs were noted in a Software Design Document (SDD) for design consideration.

3.2 Requirements analysis

From the input provided by the stakeholders, we picked a handful set of common requirements to be met by the solution:

- Reduce not only developer efforts to expose APIs to other languages but also bandwidth to maintain language bindings across different platforms.
- Promote the separation of concerns: C++ engineers should be able to minimize their involvement in platform codebase and vice versa.
- Foster the reusability of language bindings solutions across various SDKs.
- Ensure consistency in the quality of language bindings across various SDKs.
- Enhance the visibility and manageability of cross-language APIs: It should be trivial to observe and control which platform a specific API is available on without browsing the source code. For example, the fine-grained visibility control is useful when an API is exposed in Java while intentionally hiding in Objective-C. Similarly, the fine-grained visibility control is useful when an API is exposed only in certain flavors of the SDKs.

3.3 Software design and execution

Once the requirements are identified and clarified, the next phase is to design the software solution that can address those requirements. This is an iterative process requiring the team to outline the design of the solution in the SDD and request feedback from stakeholders during the weekly SDK review forum. Next, the feedback was addressed by updating the design until the majority of the stakeholders reached a common consensus. After that, the team provided the project's roadmap, which consists of several milestones, their Level Of Effort (LOE), and their Expected Time of Arrival (ETA). Each of the milestones includes tasks to be completed within that milestone.

During the development process, we used scrum as an agile software methodology where user stories were broken down into smaller tasks executed on different bi-weekly sprints. The project used Git as version control and Github repository to host the code and collaborate with team members and stakeholders. Each task that requires code change was resolved by at least one pull request on Github. Pull requests were typically reviewed by experts in the related languages and platforms. Often, the committers and reviewers were also stakeholders from different teams who desired to influence the design decisions of the product at the early stages.

3.3.1 Simplified Wrapper and Interface Generator (SWIG)

One of the most popular tools in this category is Simplified Wrapper and Interface Generator (SWIG). SWIG is a battle-tested compiler capable of parsing C/C++ headers and generating language bindings to multiple host languages [17]. At a basic level, the key advantage of this approach is that it requires trivial efforts to expose a huge existing C++ library to other languages under time constraints. This is because raw C++ headers are the only requirement of SWIG to generate basic bindings out of the box. For advanced use cases, SWIG was designed with extensibility and flexibility in mind as it provides users with fine-grained control over the generated code. In particular, SWIG allows programmers to not only customize the output APIs but also provide custom data type conversion code via its typemap annotation mechanism [17]. Additionally, programmers can also extend SWIG by adding their custom language generators.

4 Technical solution

This chapter seeks to provide a solution that can potentially meet the requirements stated in the previous chapter. In particular, section 4.1 offers an overall view of the solutions. Section 4.2 provides hypothetical evidence that the proposed solution can satisfy most of the requirements. Section 4.3 evaluates the existing solutions.

4.1 Proposed solution

One observation is that most language bindings consist of a large amount of boilerplate code. Thus, the repetitive process of writing language bindings can mostly be automated. This thesis proposes a solution that uses Interface Description Language (IDL) to describe the common interface between different languages. The thesis project aims to develop a compiler that reads the interface description written in IDL syntax and generates all the necessary marshaling code in the chosen languages. The proposed solution is a hybrid approach that applies the concept of IDL, as described under the Remote Procedure Call category, to FFI code generation. Figure 6 shows the new architecture introduced by this solution.

Similar to the RPC workflow, the process of exposing APIs to different languages consists of three steps. Firstly, the APIs between languages are described in text files using a specific IDL syntax. IDL files are then compiled to generate all the stub APIs and necessary boilerplate bindings. Finally, the actual implementations are manually connected to the bindings by developers.

4.2 Hypothesis

Our hypothesis is that the new workflow might help improve cross-language SDK development in four key categories. Firstly, developers' productivity is expected to increase as the code generator exempts them from manually developing and maintaining language bindings. In fact, this approach should only require developers to describe the interfaces in the IDL files and then regenerate the bindings when their APIs are changed. Secondly, the stability of the bridging code could also be enhanced since human errors are substantially minimized thanks to automation. We firmly believe that code generation can compensate for the limitations of static code analysis tools in multi-language systems.

Thirdly, this approach offers a higher level of separation of concerns as experts in one language no longer need to work directly in codebases written in other languages. For example, Java developers no longer have to work directly in the C++ codebase and vice versa. Last but not least, scalability is the main advantage of this approach, thanks to the centralized and streamlined fashion of the code generator. In fact, the code generator can serve as a hub for handling language bindings in multiple libraries across the organization. This means bug fixes, improvements, and new features such as new languages added to the generator can easily cascade to all the dependent libraries. As a result, the quality of the bindings code can be assured at one single point instead of scattering all over multiple projects and teams within

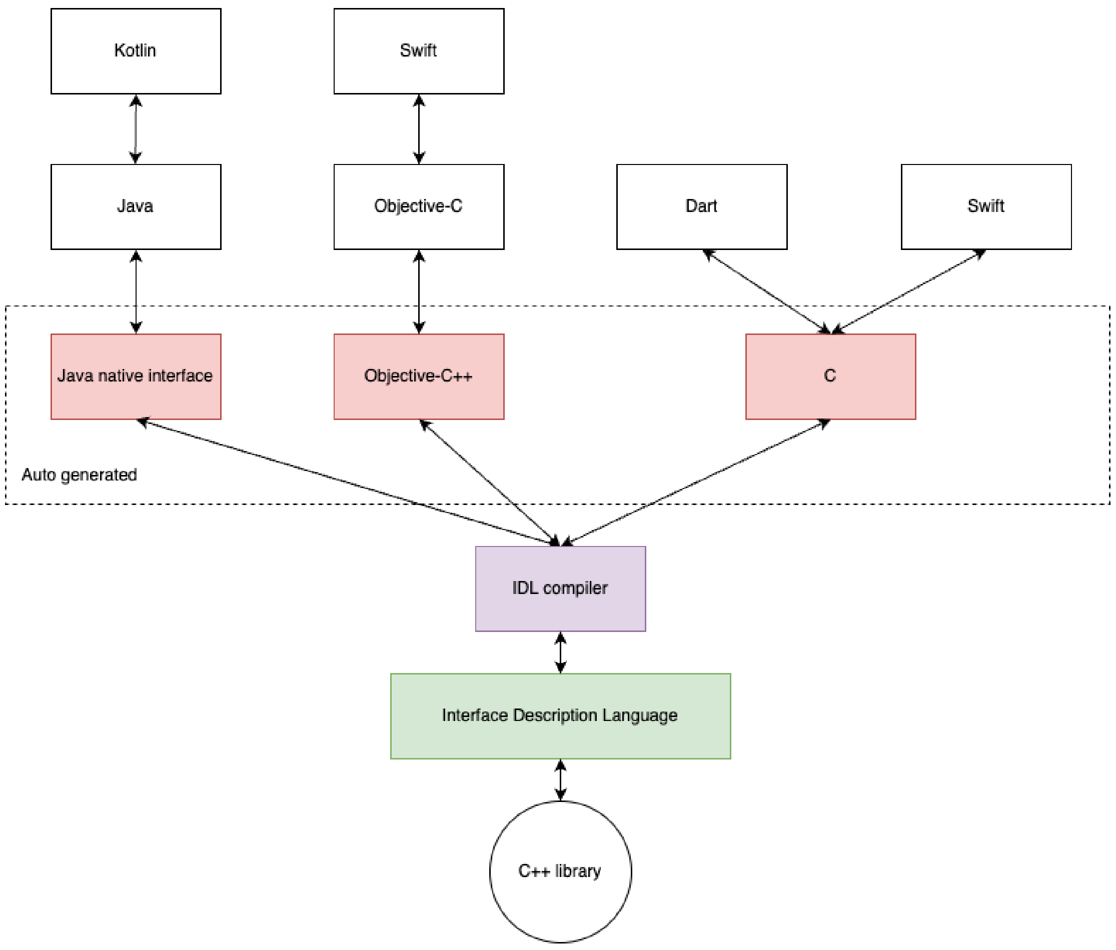


Figure 6: New architecture after the introduction of IDL

the organization. For example, an organization is responsible for `X` numbers of C++ libraries, each of which has on average `Y` numbers of APIs and is exposed to `Z` numbers of platforms. With the IDL approach, a team that is in charge of a single library only needs to manage `Y` APIs as opposed to `Y * Z` pieces of bindings. This means that the entire organization barely manages `X * Y` APIs instead of `X * Y * Z` pieces of bindings, as pointed out in section 2.2.3.

4.3 Existing solutions

Bindings code generator is certainly not a new topic. In fact, various solutions have been developed to solve language interoperability problems in the past few decades. Those solutions can be divided into two groups: single-pair and multi-pair generators. Single-pair generators cater towards a specific pair of programming languages such as Python-C++ interop with PYLCGDIC generator [16]. In contrast, the goal of multi-pair generators is to generate bindings for multiple pairs of languages. This section aims to briefly overview two prominent multi-pair binding generators (SWIG and Djinni) and highlight their key differences. Frankly, our solution does not aim to compete with those generators but rather solves Mapbox-specific problems in our multi-language software stack.

4.3.1 Simplified Wrapper and Interface Generator (SWIG)

One of the most popular tools in this category is Simplified Wrapper and Interface Generator (SWIG). SWIG is a battle-tested compiler capable of parsing C/C++ headers and generating language bindings to multiple host languages [17]. At a basic level, the key advantage of this approach is that it requires trivial efforts to expose a huge existing C++ library to other languages under time constraints. This is because raw C++ headers are the only requirement of SWIG to generate basic bindings out of the box. For advanced use cases, SWIG was designed with extensibility and flexibility in mind as it provides users with fine-grained control over the generated code. In particular, SWIG allows programmers to not only customize the output APIs but also provide custom data type conversion code via its typemap annotation mechanism [17]. Additionally, programmers can also extend SWIG by adding their custom language generators.

Despite the flexibility that SWIG provides, we believe this tool might not be suitable for our long-term use, mostly due to its heavy dependence on C++ headers and its lack of support for Objective-C language. First and foremost, C++ is a complex language with a history of over 40 years. Thus, it supports a wide range of semantics and idioms from both modern and legacy versions of the language. In practice, a considerable portion of C++ features can pose challenges when mapping directly to other languages [18]. One notable example is the ambiguous use of raw pointer as parameter type in function `void test(Bar* bar)`. This is because `bar` can be a pointer to either a single object of type `Bar` or an array of `Bar` objects, which requires different generated code. To resolve this ambiguity, programmers are often required to annotate the APIs to tweak the behavior of the generated code. Hence, the complexity of C++ syntax and semantics demands more responsibility from C++ programmers to ensure that the exposed C++ APIs are compatible with other languages.

Secondly, introducing additional custom marshaling code snippets and annotations to the already complicated C++ syntax not only reduces readability but also promotes tight coupling in C++ headers. Consequently, those drawbacks pose maintenance burdens for engineers to manage the exposed APIs, especially mobile engineers who are not necessarily familiar with C++. Thirdly, consuming raw C++ headers as input is not portable when targeting languages that are not C/C++. For example, we might want to replace C++ with Kotlin-native to share business logic between iOS and Android platforms in the future. In addition, the demands for glue code between native mobile languages and cross-platform languages such as JavaScript and Dart are increasing thanks to the growing popularity of cross-platform UI frameworks such as React-native and Flutter. Last but not least, SWIG does not provide official support for Objective-C bindings - one of the supported languages at Mapbox.

In short, SWIG shines when exposing an existing large-sized C++ library to other languages in one shot. However, SWIG might show shortcomings in maintenance and incremental development due to the dependence on C++ syntax.

4.3.2 Djinni

Djinni[26] is a code generator originally developed by Dropbox with the aim to share a common C++ codebase between mobile platforms. Conceptually, there exist ample similarities between Djinni and the generator proposed in this thesis: the exposed APIs are both described in separate IDL files and then parsed to generate binding code in each involved language. Indeed, the design philosophy of this approach is the opposite of the SWIG approach. In particular, SWIG gives ample flexibility to C++ programmers; thus, it puts more responsibility on C++ programmers to ensure the compatibility between the exposed C++ APIs and other languages. In contrast, Djinni dictates and restricts the design of the C++ API exposed to host languages because it generates the interface code and requires programmers to fill in the implementation.

In fact, we had considered leveraging Djinni as our potential solution to language interoperability instead of building our in-house tool. However, Djinni has no longer been actively maintained by Dropbox since 2019. Even though Djinni has become a community-driven project recently, it still has a long way to catch up with the growing demand for Mapbox. In particular, Djinni does not generate bindings for languages such as Swift, Dart, and Node.js, which we aim to support. In addition, Djinni does not provide support for many useful features such as asynchronous callback, class inheritance, and interface inheritance. Apart from those generic functionalities, we also desire to embed many of our own existing data structures like GeoJSON to the IDL and the support library.

As a result, we decided to build our in-house bindings generator to avoid the limitations of Djinni. We want complete control over the tool so that we can move quickly when solving Mapbox-specific problems. Indeed, the Djinni generator influences many ideas in our project. Table 1 summarises the differences between different tools.

Table 1: Comparison matrix between different tools

	Ease of adoption	Ease of maintenance	Support for Dart/Swift
SWIG	High	Low	No
Djinni	Medium	High	No
Our solution	Medium	High	Yes

[26]Djinni code generator. https://github.com/dropbox/djinni. Accessed 06/2022.

5 Implementation

This chapter describes the end-to-end implementation of the generator. In detail, the generator consists of 3 main components: front-end, back-end, and toolings, which are covered in chapters 5.1, 5.2, and 5.3 respectively. The front-end is the component where users interact with the software. It is responsible for defining the syntax and semantics of the IDL. In particular, the front-end part is in charge of parsing the IDL to construct a data structure known as Abstract Syntax Tree (AST). Once the AST is available, the front-end then validates the AST before passing it to the back-end side.

The back-end part plays the most crucial role in the generator. It traverses the AST and performs all the heavy-lifting bindings generation for multiple languages. The generated bindings typically include APIs in the host language, APIs in the foreign language, and marshaling code that converts data structures between languages. In the context of this problem, host languages refer to mobile platform languages while foreign language is C++. Finally, the toolings consist of several components such as the support library and build plugins to facilitate integration with the existing build systems of each language involved. Figure 7 depicts the high-level architecture of the compiler.

5.1 Front-end

This section provides implementation details on how the front-end component was constructed. First of all, subsection 5.1.1 begins by explaining the technical reasons behind the introduction of the third language. Next, subsection 5.1.2 walks through the key building blocks of the IDL. Then, subsection 5.1.3 presents the syntax of the IDL and parsing techniques. After that, subsection 5.1.4 provides information about the Abstract Syntax Tree (AST) data structure. Finally, subsection 5.1.5 explains how semantic analysis is performed on the AST.

5.1.1 Interface Description Language

There are two steps that the generator in this project performs: it parses the API specifications described in a certain format and then generates code according to those specifications. It means that the first step of the project is to pick a suitable data format for API specifications.

There exist various data formats for such purposes. For instance, JavaScript Object Notation (JSON) or Yet Another Markup Language (YAML) are commonly used for writing configuration. While JSON and YAML are language-agnostic, those data formats are often error-prone, verbose, and inexpressive for specifying this type of API. As a result, it requires a non-trivial amount of boilerplate to describe and verify the APIs. This poses challenges not only in writing the API specifications but also in maintaining those specifications on a large scale. Thus, we need a more

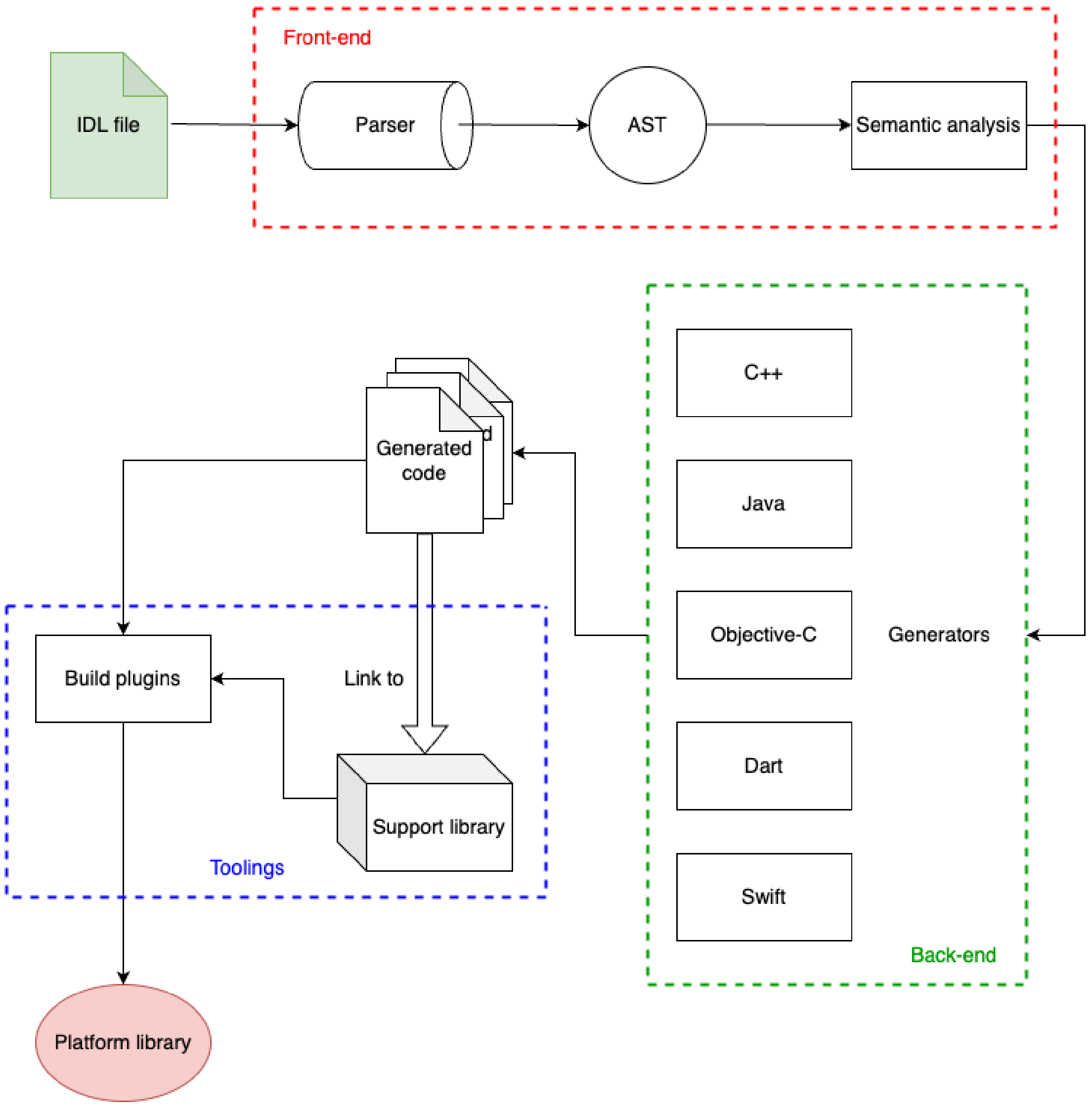


Figure 7: Architecture of the compiler

compact and expressive medium for API specifications.

Another alternative is to parse C++ header files to capture API specifications. Indeed, this is the approach of the SWIG generator. This approach might exempt programmers from describing the APIs in separate files since most metadata is already available or can be added directly within the corresponding C++ headers. While this solution allows C++ developers to expose their native APIs conveniently, it is not language-agnostic. Hence, it can be cumbersome for platform developers under certain situations since they are required to read and modify C++ code. Moreover, the dependency on C++ headers might hinder the portability of the solution since we might want to switch the target to a different language in the future. In addition, parsing C++ is non-trivial due to its syntax and semantics complexity.

Another solution is to introduce a new domain-specific language (DSL) that can be used for describing APIs. This language is supposed to be sufficiently compact, user-friendly, and independent from all the generated languages. Indeed, this IDL language is intentionally designed to be the lowest common denominator among all the generated languages. As a result, both C++ engineers and platform engineers should have little to no difficulty developing and maintaining API specifications. Thus, the IDL files can serve as a contract between all the parties, including C++ engineers, platform engineers, and managers. In particular, this solution can mitigate the weaknesses imposed by the two solutions discussed previously in this section. First, it can offer a higher level of conciseness and expressiveness that JSON and YAML lack when describing APIs. For example, describing C++ or Java methods in a simplified version of both languages feels more intuitive. It is because of the syntactical similarity between the IDL and those languages. Second, introducing a new language can also promote portability and reusability, which are lacking in the solution that parses C++ header files for API specifications. This solution is reusable as we no longer depend on a specific language such as C++. Indeed, the IDL can be used to generate bindings between various pairs of different languages, or it can be used for different purposes apart from bindings generation. The challenge of this approach is that we must introduce a new language, which requires additional engineering effort to develop the parser. In addition, C++ and platform engineers must also learn the new Interface Description Language. Table 2 depicts the summarised comparison matrix between 3 approaches under three criteria.

Table 2: Comparison matrix between different approaches

	Conciseness	Language Agnostic	Ease of implementation
JSON/YAML	Low	High	High
Parsing C++ headers	High	Low	Low
IDL	Medium	High	Medium

5.1.2 Key constructs

Before constructing the IDL, it is important to clearly define a minimal set of key features that are frequently exposed to platform languages and map them to IDL constructs. Those IDL building blocks can be categorized into functional constructs and data constructs.

The **functional category** is a set of constructs that can invoke user-defined code. This group includes class, interface, and callback. Types under this category are marshaled by reference across the language boundary.

- `Class` is the IDL construct used for exposing concrete classes whose implementation resides in C++. In practice, class is the most crucial building block in the IDL since most use cases are when the platform language invokes methods implemented in C++. An example of class `Foo` in IDL syntax is illustrated in Listing 8.

```
class Foo {
    constructor() // implementation in C++
    constructor(arg1: Type1)
    constructor(arg1: Type1, arg2: Type2)

    bar(): void // implementation in C++
    bar(arg1: Type1): Type2
    static staticMethod(arg1: Type1, arg2: Type2): Type1

    instanceVar1: Type1
    static var2: Type2
}
```

Listing 8: Example class in IDL

- `Interface` is the IDL construct used mainly for exposing an abstract platform class object to C++. This construct allows C++ to invoke methods implemented in platform languages via conformed interfaces. It is useful in situations where C++ code needs access to native platform APIs and libraries. An example of an interface `Bar` in IDL syntax is illustrated in Listing 9.

```
interface Bar {
    foo(): void
    bar(arg1: Type1, arg2: Type2): Type2
}
```

Listing 9: Example interface in IDL

- Callback is the IDL construct that allows C++ to invoke a lambda function created in platform languages and vice versa. It is primarily used for subscribing to asynchronous events whose emitters are located on the other side of the language boundary. Listing 10 shows an example of a callback named `Baz` in IDL syntax.

```
callback Baz(arg1: Type1, arg2: Type2)
```

Listing 10: Example callback in IDL

The **data constructs category** includes types primarily responsible for holding data. Those constructs are marshaled by copying value across language boundaries.

- `Primitive types` are the smallest building blocks in the IDL, including numeric types (`int8, int16, int32, int64, uint8, uint16, uint32, uint64`), `boolean, timestamp`, and `string`.

- `Enum construct` represents enumeration types.
- `Container types` are generic data structures that store a collection of a specific-typed element. This group includes `array<T>`, `map<K, V>`, `optional<T>`, and `result<S, F>`.
- `Record` is the IDL type that represents pure data objects. Typically, it contains a group of fields of various types. An example of a record named Person in IDL syntax is illustrated in Listing 11.

```
record Person {
    name: string
    age: uint8
    email: optional<string>
    favoriteColors: array<Color>
}
```

Listing 11: Example record in IDL

5.1.3 IDL parser

Once all the essential IDL constructs have been defined, the next phase is to parse the IDL constructs. This section does not aim to provide in-depth details about low-level parsing techniques but rather gives a cursory look into how parsing is performed in a large picture.

Similar to any language, Interface Description Language is constructed by arranging a set of keywords in a way that follows pre-defined grammar rules. In fact, parsing is a process that takes a linear stream of tokens and transforms them into a meaningful hierarchical data structure known as the Abstract Syntax Tree (AST). The first step of parsing is to define IDL keywords and grammar rules in Context-free grammar format. The over-simplified grammar rules of the IDL are shown in Listing 12.

```
IDL -> (CLASS | INTERFACE | RECORD | CALLBACK | ENUM)+ EOF

// top level

CLASS -> 'class' TYPE_NAME '{' CONSTRUCTOR* METHOD* ↩
    INSTANCE_VARIABLE* '}'

INTERFACE -> 'interface' TYPE_NAME '{' NON_STATIC_METHOD* '}'

RECORD -> 'record' TYPE_NAME '{' INSTANCE_VARIABLE* '}'

CALLBACK -> 'callback' TYPE_NAME '(' ARGUMENT? ')'

ENUM -> 'enum' TYPE_NAME '{' ENUM_VALUE+ '}'

```

```
// nested level

CONSTRUCTOR -> 'constructor' '(' ARGUMENT? ')'

METHOD -> 'static'? ID '(' ARGUMENT? ')' ':' RETURN_TYPE

NON_STATIC_METHOD -> ID '(' ARGUMENT? ')' ':' RETURN_TYPE

ARGUMENT -> ID ':' TYPE_ID | ARGUMENT ',' ARGUMENT

INSTANCE_VARIABLE -> ID ':' TYPE_ID

ENUM_VALUE -> ID

ID -> /[a-zA-Z][_a-zA-Z0-9]*/

RETURN_TYPE -> 'void' | TYPE_ID

TYPE_ID -> TYPE_NAME | NUMERIC | 'bool' | 'string' | 'timestamp' | CONTAINER

TYPE_NAME -> /[A-Z][_a-zA-Z0-9]*/

NUMERIC -> 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'float' | 'double'

CONTAINER -> 'optional' '<' TYPE_ID '>' | 'array' '<' TYPE_ID '>' | 'map' '<' TYPE_ID ',' TYPE_ID '>' | 'result' '<' TYPE_ID ',' TYPE_ID '>'
```

Listing 12: Context-free grammar of the IDL

The defined IDL grammar rules are then fed to a parser to parse IDL files and construct the AST for further processing. This project uses an open-sourced JavaScript parsing tool named Parsimmon to perform the heavy-lifting parsing work. Parsimmon is a top-down parser using the `Left-to-right, leftmost derivation(LL)` parsing algorithm that is suitable for the simple, unambiguous grammar of the IDL. The process of describing IDL grammars to Parsimmon parser is relatively straightforward. For example, the over-simplified grammar of the IDL Record in Parsimmon is shown in Listing 13.

```
const token = str => Parsimmon.string(str)
    .skip(Parsimmon.whitespace);

const recordParser = Parsimmon.createLanguage({
    Record: parser =>
        Parsimmon.seqObj(
            token('record'),
            ['name', parser.TypeID],
            token('{'),
            ['fields', parser.Field.many()],
            token('}')
        ),
    Field: parser =>
        Parsimmon.seqObj(
            ['name', Parser.ID],
            token(':'),
            ['type', Parser.TypeID]
        ),
    ID: parser => token(
        Parsimmon.regexp(/[a-zA-Z][_a-zA-Z0-9]*/)),
    TypeID: parser => token(
        Parsimmon.regexp(/[A-Z][_a-zA-Z0-9]*/))
});
```

Listing 13: Persimmon grammar format of IDL Record construct

5.1.4 Abstract Syntax Tree

The responsibility of the parser is to scan the input source code and convert them into a hierarchical representation known as Abstract Syntax Tree (AST). Essentially, AST is a tree data structure that can capture the semantics of a programming language. It also discards redundant details such as colons, semicolons, and parentheses that are only useful for parsing. For instance, the program in Listing 14 is parsed into the AST shown in Figure 8. One observation is that all the leaf nodes are colored in red, containing the smallest building blocks such as `ID` or `TYPE_ID`. On the other hand, non-leaf nodes are the ones that capture the higher-level construct such as `CLASS`, `METHOD`, and `FIELD`. In addition, leaf nodes containing redundant tokens are all discarded from the tree.

```
class Foo {
    constructor(arg1: Type1)

    bar(arg1: Type1): Type2

    variable: Type1
}
```

Listing 14: Example class `Foo` in the IDL

In fact, the front-end part hierarchizes a linear stream of tokens into a tree data structure while the back-end reverses the process by linearizing the AST into a string in a different language. Hence, the AST acts as a bridge or an intermediate representation between the front-end and back-end components of the compiler. This tree representation can be used for semantic analysis, early optimization, and code generation.

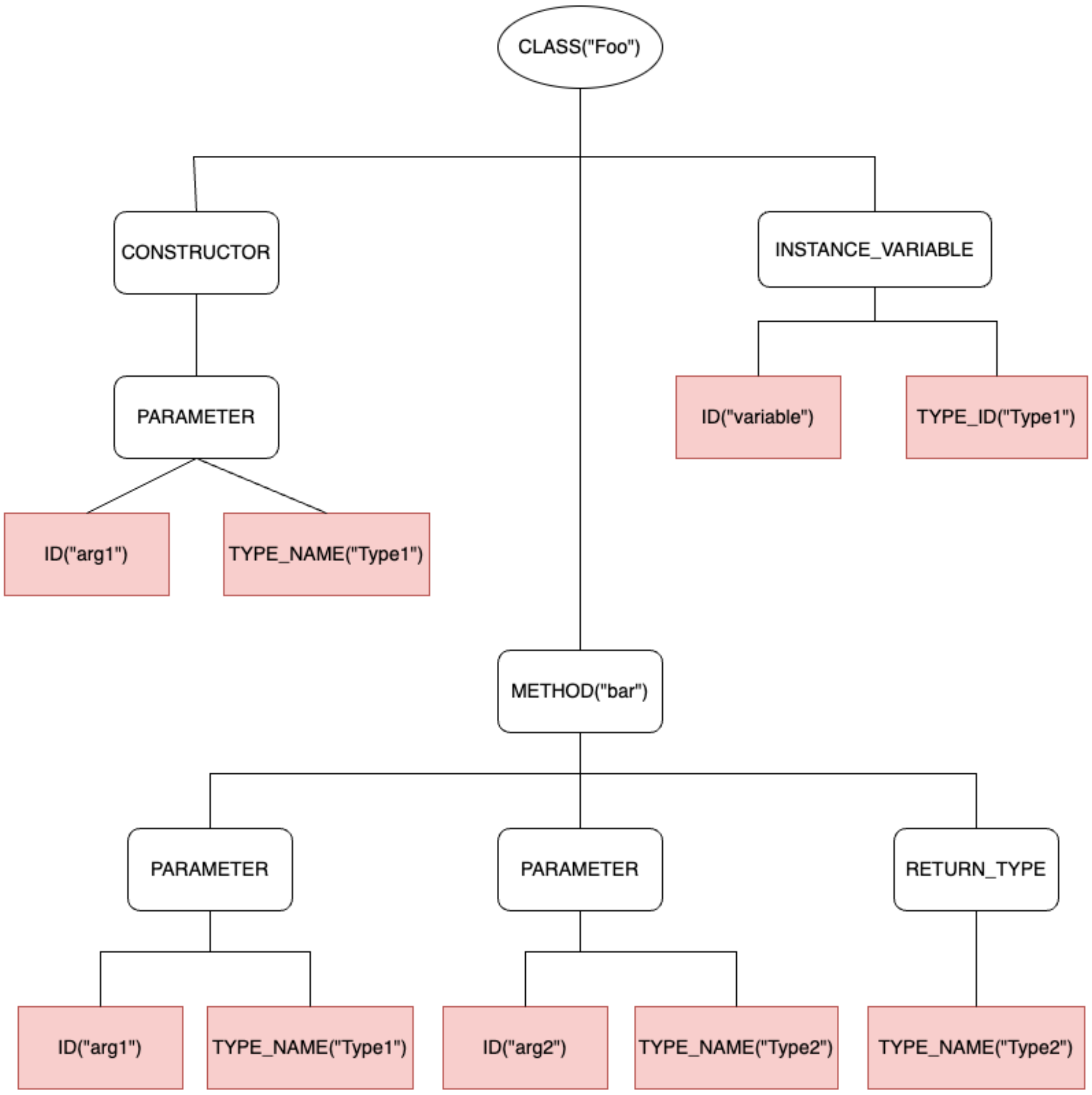


Figure 8: Abstract syntax tree of the class Foo in Listing 14

5.1.5 Semantic analysis

The parser can detect syntactical errors when the source code fails to follow language grammar rules. Nevertheless, other categories of errors exist that the parser usually fails to recognize. One such category of errors is semantic errors. For example, the interface illustrated in Listing 15 is syntactically correct even if the same namespace contains two top-level constructs with the same name `"Foo"`, and the type of the parameters `arg1` is undefined.

```
enum Foo { foo bar }

class Foo {
    constructor(arg1: MissingType): void
}
```

Listing 15: Semantically incorrect IDL interface

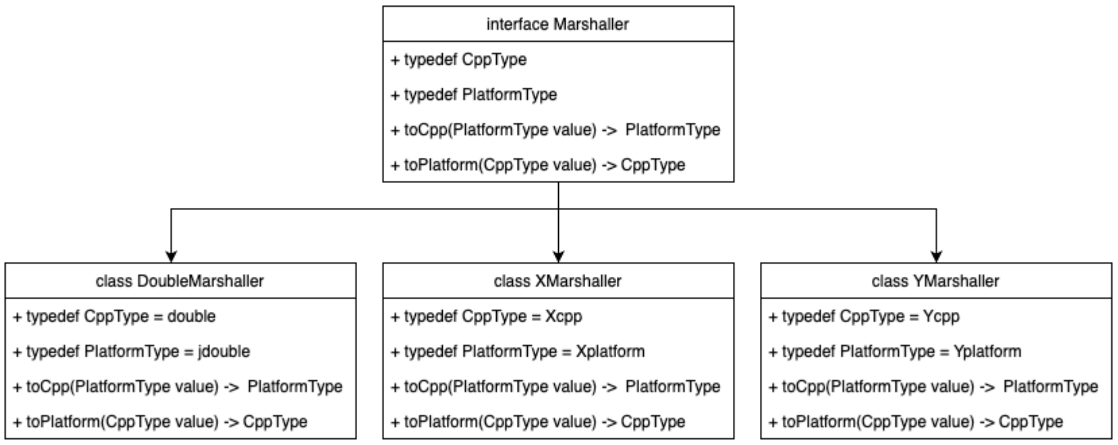
As a result, compilers often introduce another step to ensure the semantic correctness of the parsed source code. This semantic analysis step is typically performed on the AST because it contains all the necessary source code information. This step has two primary responsibilities for full-fledged programming language compilers: ensuring that variables are not used out of their scopes and type-checking the program. However, the IDL semantic rules in this project are reasonably trivial as we only check for the correctness of IDL declarations.

In this step, the tool first traverses the AST and uses a hash table to store the mapping between the label of a construct and its corresponding AST node. This data structure allows `O(1)` time-complexity when retrieving an AST node by its label, which is frequently used in type-checking and further code generation. This step ensures that all the types are defined and that the IDL definitions follow the semantical rules. In practice, the program is typically organized into a set of IDL files that can import definitions from each other. As a result, the generator recursively parses the specified IDL file and its imported dependencies to construct the symbol table for the semantic analysis process.

5.2 Back-end

While the user interface is a vital part of the tool, the heart of the generator inevitably lies on the back-end side. The back-end component is responsible for taking the previously constructed AST, traversing its nodes, and subsequently producing bindings code. This chapter presents the techniques for bridging different IDL constructs across language boundaries.

Generally, the generated code can be divided into three parts: APIs in platform languages, APIs in C++, and the marshaling code written in C++ or platform languages. The implementation details of the APIs may vary across different IDL constructs and languages. Nonetheless, the marshaling code of different IDL constructs all share a standard format. They are organized into a set of marshallers, each responsible for converting a separate IDL construct between two languages. Those marshallers of IDL constructs conform to the interface Marshaller and have the shape as shown in Figure 9:



interface Marshaller
+ typedef CppType
+ typedef PlatformType
+ toCpp(PlatformType value) -> PlatformType
+ toPlatform(CppType value) -> CppType
class DoubleMarshaller
+ typedef CppType = double
+ typedef PlatformType = jdouble
+ toCpp(PlatformType value) -> PlatformType
+ toPlatform(CppType value) -> CppType
class XMarshaller
+ typedef CppType = Xcpp
+ typedef PlatformType = Xplatform
+ toCpp(PlatformType value) -> PlatformType
+ toPlatform(CppType value) -> CppType
class YMarshaller
+ typedef CppType = Ycpp
+ typedef PlatformType = Yplatform
+ toCpp(PlatformType value) -> PlatformType
+ toPlatform(CppType value) -> CppType

Figure 9: Marshaller structure

As discussed previously, there are two groups of types in the IDL: default types and user-defined types. The marshallers of default types such as string, double, and date are provided out-of-the-box as part of the support library. In contrast, marshallers of custom types such as class, record, interface, and callback are all generated. Since all marshallers conform to the same interface named Marshaller, they can be composed arbitrarily to marshal complex, nested data structures such as `map<string, array<optional<record>>>`. In practice, the complexity of the marshaling code for reference types is substantially higher than that of value types such as record and enum. The following sections dive into the implementation details of a few notable IDL constructs.

5.2.1 Class

When a class is defined in the IDL, the generator generates three components: the APIs for that class in C++, the APIs in the platform language, and the marshaling code to convert between two class objects:

- **C++ class:**
  The C++ library exposed via this generated C++ interface can be consumed as a standalone product for embedded Linux environment. It uses Pointer to Implementation (PImpl) idiom in C++: the generator outputs a class that declares all the methods specified in the IDL. That class also holds a private pointer to a separate implementation class object in C++. When a method is invoked from the C++ interface, the generated code forwards the call to the implementation class, which programmers must provide manually.

This PImpl idiom hides the implementation details from users and allows the C++ implementation to change without triggering compilations for all of its dependencies.

- **Platform class:**
  The similar PImpl technique is also used for the platform class, which means that the platform class is a proxy that forwards method calls to its native peer. In particular, the object of the generated C++ class is instantiated in the constructor of the platform class. Next, an instance variable of the platform class object is used to store a memory address of the C++ class object. Then, the code that registers native handler functions for platform class methods is activated. After the setup has been completed, the platform object can forward the invocation to the native C++ code. This cross-language method call is implemented by extracting the native C++ object from the platform class object, marshaling the parameters to C++, calling the corresponding method on the extracted C++ object, and subsequently marshaling the result back to platform languages. It is important to wrap this cross-language method invocation inside a try-catch block to systematically catch and handle exceptions thrown by the C++ method. If an exception is thrown from C++, the generated code must translate it to platform exception and rethrow in the platform language to avoid leaking native exception. For example, C++ exceptions are translated to `Exception` in Java or `NSError` in Objective-C.

Regarding memory management, this PImpl design requires a special memory management technique since the C++ object must stay alive as long as the platform object is still alive. Otherwise, there exists a situation known as dangling pointer where methods are called on the proxy object after its native peer has already been destroyed. This means that the proxy class needs to hold a strong reference to the C++ object to keep it alive. When the proxy class object is about to be collected, it must trigger the C++ object deallocation to release the occupied memory. One tricky implementation detail is about C++ pointer ownership. A naive solution is to let the platform object directly hold a raw pointer to the native object. This means that the platform object takes ownership of the native object and must deallocate its native peer at the end of its life cycle. Nevertheless, C++ code sometimes also needs to take ownership of the C++ object. One example is when a singleton C++ object is exposed to platform code. The singleton pattern requires the object to be stored in a C++ static variable. If the platform object exclusively owns the singleton native object, it might as well prematurely deallocate the singleton object, which is typically supposed to live until the program terminates. Hence, C++ code should also be able to share the ownership of the C++ object. One solution to this problem is to use a shared pointer to manage memory for C++ objects and let the platform object exclusively hold a raw pointer to the heap-allocated shared pointer. Indeed, the platform object can safely deallocate the raw pointer to the shared pointer without affecting other parts of C++ code. Thus, this approach allows both C++ and platform objects to share ownership of the C++ object since it is only deallocated as soon as the last instance of the shared pointer is out of scope.

- **Marshaling code:**
  When a shared pointer to the C++ object is marshaled to platform languages, a copy of that shared pointer is allocated on the heap, and a new proxy class in platform languages is created to hold that pointer. However, there occur situations where the same C++ object is marshaled to the platform object more than once. Suppose a new platform proxy object and a new copy of that shared pointer are created whenever a C++ object is marshaled to platform languages. In that case, the binding code will have two problems: poor performance and loss of object identity. The first problem happens because new platform objects are created whenever C++ class objects are marshaled to the platform side, which is not only slow but also resource-consuming. The second problem exists because we always receive a new and different platform object when an original platform object is marshaled to C++ and back. The loss of object identity can be problematic in code that compares objects' identity, for example, when platform objects are used as keys in a hashmap. One solution to this object identity problem is to cache the reference to the platform proxy object in an instance variable of the C++ object to reuse the existing platform object. However, suppose the C++ object holds a strong reference to the platform proxy object. In that case, it will create a circular dependency situation because the platform object also holds a strong reference to the C++ object to keep the C++ object alive as long as the platform object stays alive. As a result, one will wait for the other to be deallocated, creating a strong circular reference, which prevents both objects from being collected. In order to avoid such a circular reference scheme that leads to the memory leak situation, the C++ object can only cache the weak reference to the platform object. Thanks to the nature of weak references, the circular reference scheme is broken because weak references do not interfere with the garbage collector. Thus, the platform object can now be deallocated before its native peer. If the native peer is still alive after its platform object is collected, the weak reference returns `null` when being marshaled back to the platform side. The marshaling code can create a new platform object to hold the native object in this situation. Figure 10 illustrates the memory ownership of the class object.

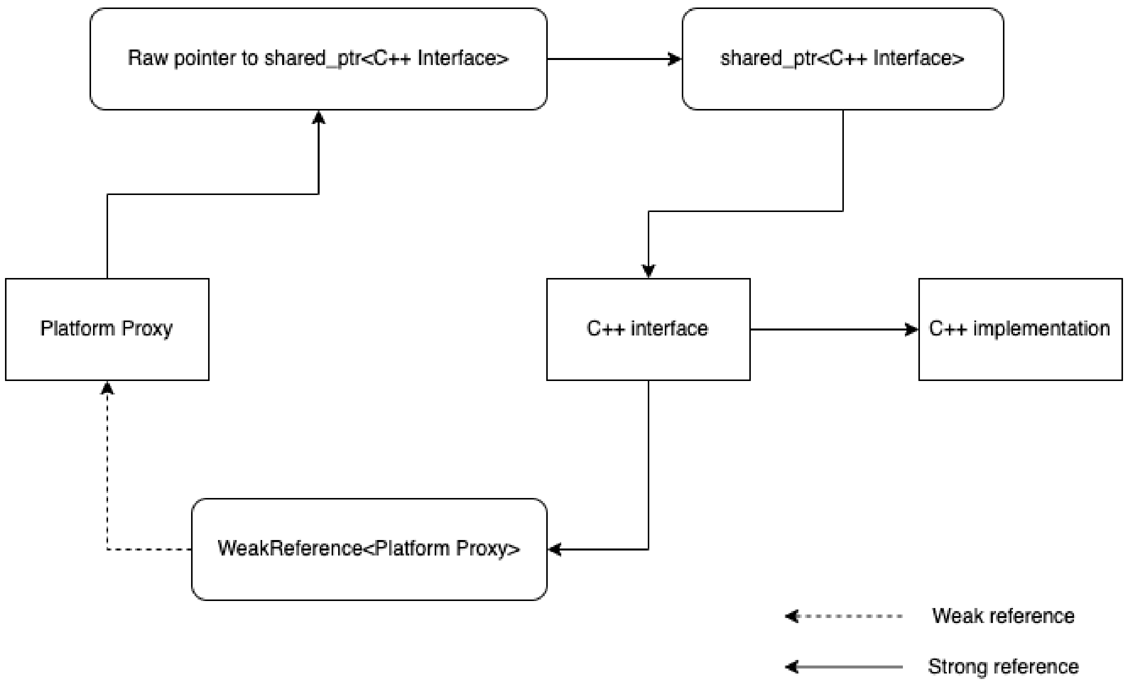

Raw pointer to shared_ptr<C++ Interface>
shared_ptr<C++ Interface>
Platform Proxy
C++ interface
C++ implementation
WeakReference<Platform Proxy>
Weak reference
Strong reference

Figure 10: Memory ownership of class object

Class objects are marshaled in 2 directions: from C++ to platform languages and from platform languages to C++, as shown in Figure 11. When a proxy class object is instantiated in platform languages, a native peer object is created as a result. The marshaling code ensures that the platform object always holds a strong reference to the C++ object while the native object holds a weak reference to the platform one. The process of marshaling a platform object to C++ is trivial as we only need to load the shared pointer to the C++ object from its address stored in the instance variable of the platform object. Nonetheless, the marshaling logic is slightly more complicated when marshaling C++ object to platform proxy object. First, we need to check if the C++ object has a reference to the alive platform object. If the platform object is alive, we simply return it to the platform code. Otherwise, we allocate memory for the shared pointer on the heap and create a new platform object to hold it.

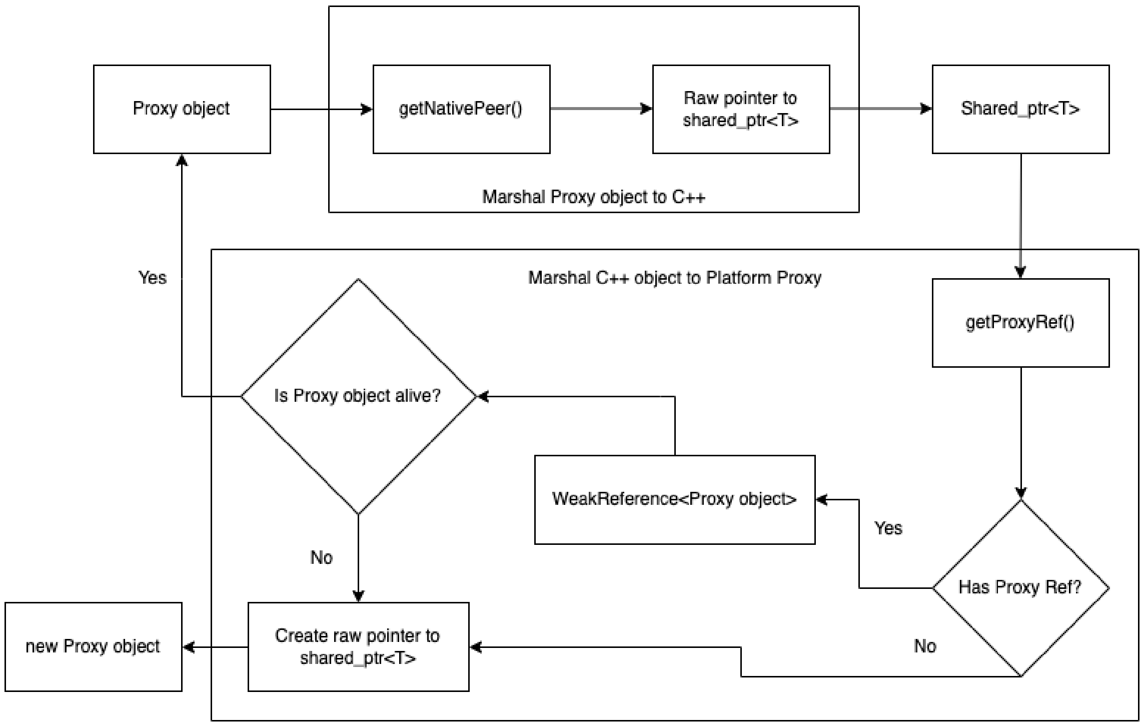

Proxy object
getNativePeer()
Raw pointer to shared_ptr<T>
Shared_ptr<T>
Marshal Proxy object to C++
Marshal C++ object to Platform Proxy
Yes
getProxyRef()
Is Proxy object alive?
WeakReference<Proxy object>
Yes
No
Has Proxy Ref?
new Proxy object
Create raw pointer to shared_ptr<T>
No

Figure 11: Marshalling flow chart for class construct

5.2.2 Interface

Interface construct is an abstract class that specifies a list of methods to be implemented by other concrete classes. Other parts of the code can interact with conforming classes via the interface without knowing the underlying class. An interface can be conformed by both platform classes (**platform interfaces**) and native classes (**native interfaces**). There are four situations where the interface concept is used:

- Platform code invokes methods of the conforming platform classes.
- C++ code invokes methods of the conforming C++ classes.
- Platform code invokes methods of the conforming C++ classes.
- C++ code invokes methods of the conforming platform classes.

The first two cases are trivial as those are the normal behaviors of interfaces in any language. However, the last 2 cases pose cross-language challenges to be solved in this subsection. In detail, native interfaces are useful when platform code needs to invoke methods on C++ objects via the interfaces that those objects implement. To achieve this goal, we generate an anonymous platform proxy object that conforms to the interface since platform code only interacts with the proxy class via the interface. Internally, that proxy class holds a strong reference to the native C++ object so that it can forward the call to C++ methods. In fact, it is undeniable that native interfaces work similarly to class constructs. Nevertheless, native interfaces differ from class constructs since they can not be instantiated by calling constructors directly from the platform code. This is because an interface is an abstract class implemented by an anonymous concrete proxy class. It is only created by the marshaling code when converting an existing C++ class object to the platform language.

Another type of interface that is more frequently used than native interfaces is platform interfaces. Platform interfaces allow C++ code to interact with platform objects via interface methods. In general, platform interfaces are the reverse version of native interfaces. C++ proxy objects hold a strong reference to platform interface objects, while platform interface objects hold a weak reference to C++ objects. When C++ code calls methods that are implemented in platform language, the proxy object extracts the reference to the platform object, marshals parameters, and then forwards the call to platform methods. Like native interfaces, C++ proxy objects are only constructed in the marshaling code when converting the existing platform objects from platform languages to C++.

The generator generates three main parts for interface construct:

- **Abstract classes/interfaces** that declare a set of abstract methods to be implemented by concrete classes in both languages.
- **Concrete anonymous proxy classes** that conform to the interface in each language. Each proxy class holds a strong reference to the implementation object on the other side of the language boundary. In contrast, each object of the implementation class must hold a weak reference to its proxy to preserve the identity when being marshaled bidirectionally. One challenge is that the user-provided concrete classes that implement interfaces are unaware of the weak references to the proxy object. The unawareness means that the weak reference to the proxy object cannot always be stored in an instance variable of the class. Hence, the trick is to use a hashmap to store user-defined class objects as keys and weak references to proxy objects as values. When the proxy object is under destruction, the associated entry is removed from the hashmap so that the implementation class objects can be eligible for garbage collection.
- **Marshaling code:**
  Like the marshaling process of class constructs, interfaces are marshaled in two directions. However, there are two distinct interface types, and they are marshaled differently. When marshaling a native interface from platform languages to C++, we simply extract the memory address of the heap-allocated native peer object from the platform object. In contrast, the native interface object is marshaled from C++ to platform languages by returning an alive platform peer object from the weak reference or by creating a new one if no alive platform peer object exists. Symmetrically, the platform interface object is converted from C++ to platform languages by extracting a strong platform

object from its C++ proxy. In the direction from platform languages to C++, we return an alive C++ proxy object from a weak pointer or create a new one if none exists. The marshaling flowchart of an interface is illustrated in Figure 12.

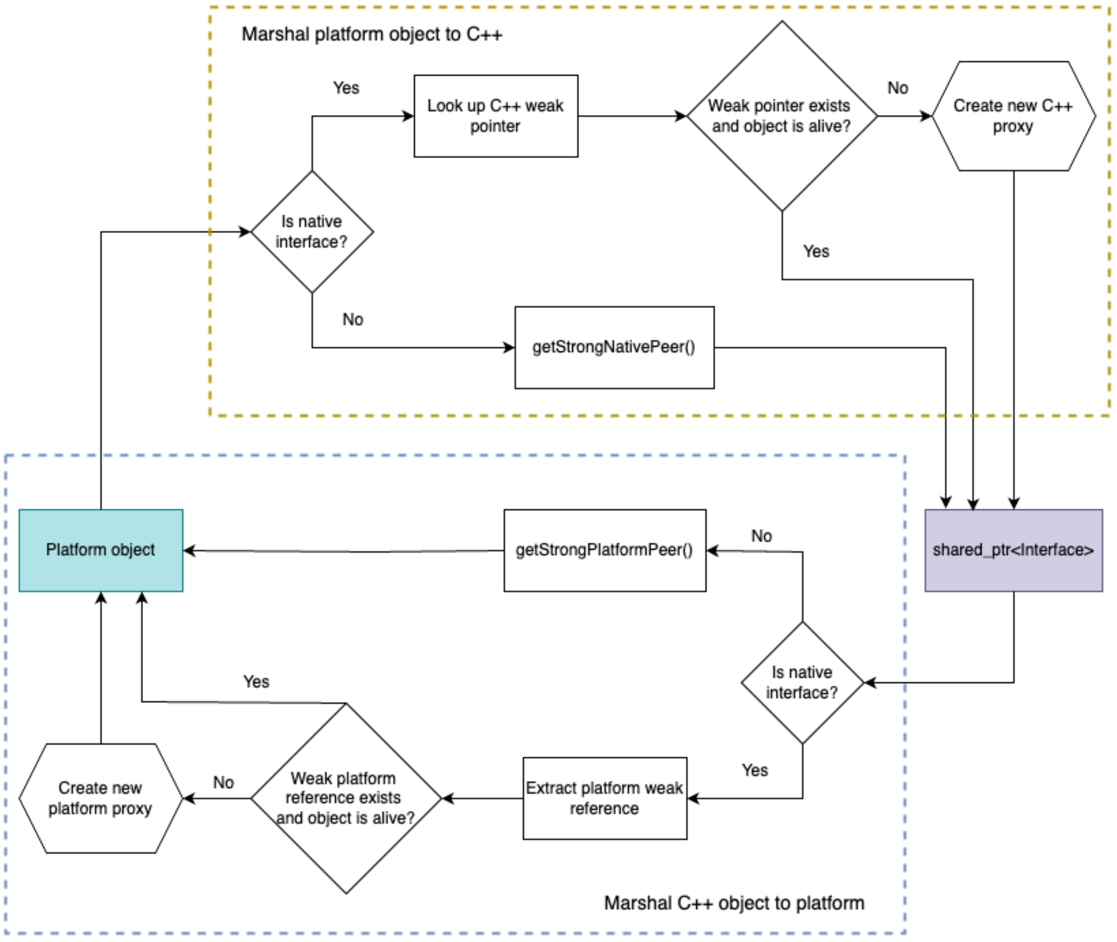

Marshal platform object to C++
Yes
Look up C++ weak pointer
Weak pointer exists and object is alive?
No
Create new C++ proxy
Is native interface?
Yes
No
getStrongNativePeer()
Platform object
getStrongPlatformPeer()
No
shared_ptr<Interface>
Is native interface?
Yes
Yes
Create new platform proxy
No
Weak platform reference exists and object is alive?
Extract platform weak reference
Marshal C++ object to platform

Figure 12: Marshalling flow chart for interface construct

Calling methods implemented in C++ from platform object is usually trivial. In contrast, extra attention is required when calling platform-implemented methods in platform languages due to inconsistencies between concurrency models of different languages. For example, Dart and JavaScript are single-threaded languages that rely on the asynchronous concurrency model powered by the event loop. It means that code written in those languages can only be executed on the main thread. Hence, calling methods implemented in JavaScript or Dart from C++ worker threads might crash the program. As a result, the native proxy object must always schedule tasks that invoke platform methods with the Node.js, and Dart event-loop and wait to execute those tasks. On an unrelated note, methods implemented in Java can only be called from C++ threads that have already been attached to the JVM. As a result, threads that are not created by the JVM need to be attached and detached from the

JVM to invoke Java methods. These specific implementation details are all considered in the generated code and are abstracted away from the end-users.

5.2.3 Record and enum

`Record` is a data object containing various fields, each of which can be of different types. We generate three pieces of code for each record: a class in platform language, marshaling code, and a struct in C++. The generated class/struct in each language simply includes a set of fields with different types in that language. The job of the marshaller is to convert each field of the record from one language to another and create a new object on the other side of the language boundary. Hence, records are marshaled by value rather than by reference like class, which means that we always receive a new record that looks identical to the original record if it is converted to the other language and back.

`Enum` is a data type containing a mutually exclusive set of predefined constants. We generate an enum class that consists of a list of enum values in both C++ and platform languages. When marshaling enum types between languages, the marshaling code converts the enum value to an integer, passes through the language boundary, and finally converts it back to the enum value in the other language.

5.2.4 Containers

Many container types such as `array, map, set, stack, queue`, and `optional` are objects holding a collection of other types. Those container types are parameterized types, which means that they are designed to work flexibly with different inner types to make the collection logic reusable. As a result, the marshaling code has to be generic as well. Indeed, the marshaler works similarly to the higher-order function `map()` on collection types of most modern languages. They accept the marshallers of their inner types as parameterized types, then iterate through the collection and use the marshallers of inner types to marshall each element. This higher-order marshaller design separates the concern between the collection marshaling code and its element marshaling code, which allows marshaling any arbitrary inner type as long as the marshaller of that inner type conforms to the common marshaller interface. This is because element marshallers are responsible for converting each collection element to another language. In contrast, the marshaller of container types marshalls the whole collection without knowing the low-level details on how to convert inner elements. For example, Listing 16 demonstrates how an array of strings is marshaled between Objective-C and Cpp.

```
// Parameterized ArrayMarshaller
template <class ElementMarshaller>
struct ArrayMarshaller {
  using CppType = std::vector<typename ElementMarshaller::CppType>;
  using ObjCType = NSArray * _Nonnull;

  static ObjCType toJava(const CppType& cppVector) {
    auto array = [NSMutableArray arrayWithCapacity:static_cast<↩
    NSUInteger>(cppVector.size())];
    for (const auto& element: cppVector) {
        [array addObject:ElementMarshaller::toObjc(element)];
    }
    return [array copy];
  }

  static CppType toCpp(ObjCType objcArray) {
    CppType result;
    result.reserve(objcArray.count);
    for (id element in objcArray) {
      result.push_back(ElementMarshaller::toCpp(element));
    }
    return result;
  }
};

struct StringMarshaller {
  using CppType = std::string;
  using ObjcType = NSString * _Nonnull;
  static ObjcType toObjc(const CppType& cppString) {
    return [NSString stringWithUTF8String:cppString.c_str()];
  }
  static std::string toCpp(ObjcType objCString) {
    return [objCString UTF8String];
  }
};

// Generated marshalling code
NSArray * objcArray = ArrayMarshaller<StringMarshaller>::toObjc↩
    ({"foo", "bar"});
```

Listing 16: Parameterized array marshaller

5.3 Quality assurance, documentation, and integration

A typical code generator project consists of two types of code: generator code and generated code. This means that quality assurance and documentation are required for both of those categories.

Quality assurance plays a crucial part in software development. Indeed, high-quality bidings code is especially critical in the Mapbox software stack as the generator serves as a central hub for producing bindings in various SDKs. This means that software defects in the generated code are likely to affect most SDKs. To ensure the high quality of the project, we employ not only unit tests to test the generator logic but also a collection of integration tests to assure that each feature behaves correctly in all supported languages. We use Jest as a testing framework for unit testing to test the parsing and generator logic. Apart from unit tests, the integration test of each feature not only ensures that the generated code works seamlessly on all supported platforms but also serves as an example and documentation on how that feature can be used. To make integration tests self-documenting to even non-code stakeholders, we chose Cucumber as our Behavior-Driven Development framework, which allows test steps to be specified in plain English. Apart from tests at the project level of the generator tool, the generated code in each Mapbox SDK is also extensively tested at the product level.

Once the behavioral quality of the generated code is ensured, the next step is to ensure that the generated code's style is consistent and follows the best practices. In particular, the generated code is checked and formatted using tools such as clang-tidy and clang-format.

One important aspect of the generated code is documentation. Similar to hand-written code where methods and classes require documentation, the generated code must be documented. In reality, documentation in multi-language software requires non-trivial efforts to create and maintain since a function's documentation in one language also needs to be populated to other languages. As a result, this requires a substantial amount of duplicated documentation, posing maintenance problems. To tackle this problem, our IDL allows programmers to add comments to each entity in the IDL and use those comment sections to generate correctly-formatted documentation for each supported language. For example, the generated documentation comments in Java follow `Javadoc` format while those in C++ follow the `Doxygen` format. This feature allows documentation to be populated from a single source of truth.

The code generator is distributed as a Node.js command-line program. This means that programmers can directly supply the IDL filename to the generator program to generate bindings for the constructs defined in that file. Using the generator this way usually requires programmers to maintain a separate script to generate bindings for a set of IDL files and add the generated source files to their build system. To facilitate this integration process, we provide build-script utilities in `CMake` and `Gradle` so that programmers only need to specify their IDL files in their build system. The build tool utilities take responsibility for generating bindings and including them in the built target at built time.

6 Evaluation

This chapter seeks to evaluate the solution proposed in the thesis. In particular, this thesis aims to create a tool that automates the process of producing language bindings within various Mapbox SDKs. Hence, the evaluation results were collected from stakeholders' feedback and statistics from Mapbox's Github repositories when four teams within Mapbox Maps, Navigation, and Search organizations had adopted the generated code in their code base. In particular, the feedback was collected via four main channels:

- Github issues submitted by internal customers in the repository of the generator project. In fact, the Github issues tab played an important part in the development process as it was the single source of truth where we documented all the actionable items and suggestions. Hence, all the feedback collected from the other three channels was eventually converted to respective Github issues for cross-team visibility.

- Comments in the Request For Comments (RFC) document collected during the Technical Discussion Forum meeting. Before the meeting, we described the problem the team was attempting to solve and provided questionnaires in a document. Then we asked for stakeholders' opinions written in the comment format of the document. During the online meeting, we discussed each point noted in the document and create a list of actionable items. In practice, Technical Discussion Forums are mainly designed for debating coarse-grained architectural decisions due to time constraints and broad audiences.

- Verbal feedback from engineers of other teams during code pairing sections. As opposed to Technical Discussion Forum meetings, pair programming sections offer useful discussions on fine-grained implementation details, which are still important yet insufficiently notable for Technical Discussion Forums.

- Self-reflection when generating code for new APIs and using them.

We attempt to evaluate the project in three categories guided by the research questions raised in the earlier chapter.

6.1 Productivity

The generator has been extensively leveraged in **4** core projects that involve nine teams within Mapbox. Specifically, API specifications have been written in **14002** lines of IDL code to generate **194366** lines of code that would otherwise be written and maintained manually by engineers across three Mapbox organizations. In fact, the amount of generated code accounts for approximately **50-70%** of the total amount of code in each repository, depending on the size of the repository. Figure 13 shows the breakdown percentage of each code category in production. As a result, automating the binding generation process drastically reduced programmers' bandwidth to write and maintain bindings by hand for various platforms.

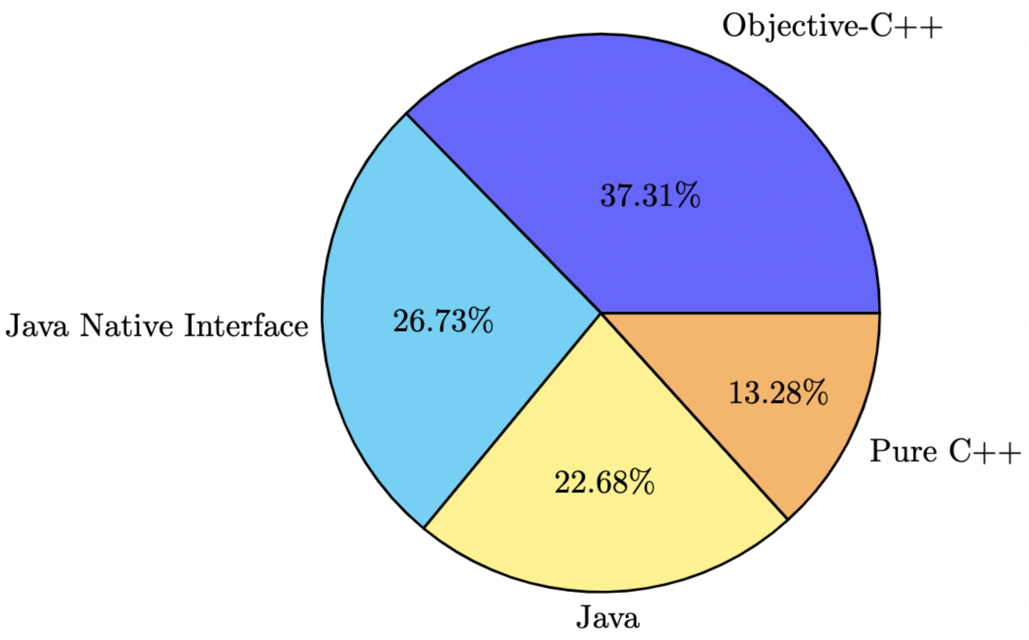


Figure 13: The breakdown percentage of each code category

One notable example is the Mapbox GL Native codebase which was a mono-repository before using the code generator. It contained three parts:

- C++ native library and third-party libraries.
- Platform bindings for Node.js, Java, Objective-C, and Qt.
- Platform adaptation code on top of the bindings for the platforms above, including examples, tests.

In this system, introducing a new API to both Android and iOS platforms typically requires one C++ engineer to expose a new native API and then two platform engineers to write bindings on their respective platforms. This workflow requires careful communication, coordination, and progress tracking to ensure the parity of the SDK on two distinct platforms as the process involves multiple parties. In reality, the process of exposing new APIs could take hours, days, or even months to finish due to delays in the coordination process, different priorities between different teams, or technical glitches. One notable example is the feature that adds aggregated cluster properties was introduced in C++ on August 16th, 2019. It was exposed on Android on August 20th, while the equivalent bindings were not added on iOS until October 11th. To make the matter worse, refactoring the already exposed APIs is a different challenge on its own. Every change in the C++ public API required adjusting the platform bindings for Node.js, Java, Objective-C, and Qt and the adaptation code, examples, and tests. C++ engineers were frequently required to update or refactor platform code outside their scope and expertise domain. Like-wise, platform engineers were required to understand how to develop and interact with a low-level platform adaptation code using Java Native Interface or solve Automatic Reference Counting(ARC) memory management issues in Objective-C++. The tightly coupled issues above adversely affected the overall project velocity, especially when introducing complicated concepts in the APIs.

After adopting the generated code, GL Native development is now divided into separate repositories: one for the C++ native library and individual repositories for the platform SDKs such as Maps Android SDK or Maps iOS SDK. C++ engineers are now responsible for maintaining an IDL file that describes their public API and generating the platform bindings code. Platform engineers develop their SDKs using the provided platform bindings as a base. All teams involved were no longer required to deal directly with issues associated with data marshaling and memory management issues between language domains, which were notoriously unpleasant to handle. As a result, the developer experience was significantly enhanced thanks to the new tool.

Thanks to the automation, the task of exposing APIs to different platforms became trivial since this step merely requires updating IDL files and regenerating code. Stakeholders are happy with the new workflow because we can now kill two birds with one stone: reducing human cost while iterating on our APIs more agilely. This is because APIs can now be exposed or modified within a matter of minutes instead of hours, days, or months as in the previous workflow. In addition, automating code generation reduced bandwidth for cross-team coordination since a C++ engineer can single-handedly generate bindings for both platforms with ease.

As the amount of manually-written bindings was minimized, so was the number of software defects related to glue code. For example, notorious Java Native Interface bugs such as JNI reference table overflow, which had been reported 16 times in the mapbox-gl-native Github repository since 2016, have yet to be seen after the transition. Apart from the JNI reference table overflow issue, JNI run-time exceptions caused by failing to lookup methods, fields, and classes because of typos in their signature strings were also eradicated in the generated code. In addition, pending exceptions thrown by FFI operations are now exhaustively handled in the generated code. More importantly, the code generator has become a hub for language bindings within Mapbox SDKs, which means that the quality of the bindings code is assured and improved at a centralized location instead of scattering all over multiple places. This is critical for future development when scaling Mapbox products and teams. According to a senior C++ engineer, we had never systematically audited the manual bindings to look for improvements in the previous workflow as the bindings code scatters in multiple places. The advantage of this centralized bindings generator is that all the teams using it will receive improvements in the bindings if one is added to the generator. As a result, the quality of the bindings code across different products can easily be monitored and controlled. On the other hand, the drawback of this approach is that a bug in the generator might affect multiple APIs and products.

Another advantage of adopting the generated bindings is the consistency and homogeneity of the exposed APIs. According to a senior Android engineer, the concept of generating code from specifications is appealing because the generated code is consistent and predictable. Same specifications always yield the same code when using the same generator version. He believes that the concept of IDL is powerful because a few simple IDL concepts can be combined arbitrarily to describe a full set of complex Mapbox products. Previously, the way the teams exposed APIs was fragmented across Mapbox products or even within the same product for various reasons. Using the code produced by the generator, we enforced strict rules, idioms, and practices on how Mapbox products should look, making different products look and feel the same from the client's point of view.

6.2 Scalability

One of the main advantages of the code generator is its high scalability and reusability. The solution shines in four situations: adding/modifying APIs to existing products, supporting new languages, exposing APIs for new products, and scaling engineering teams.

Firstly, exposing and modifying APIs have become a trivial task thanks to the generator. The generated bindings improve the scalability of our software stack on two different levels: bindings for a single platform and bindings for multiple platforms. Even for teams that expose their native library to only a single platform, using the generator to produce bindings can save considerable time and effort compared to writing them manually. As the number of APIs to expose grows, so does the number of benefits engineering teams get from generating bindings. However, the solution shines especially when the library is exposed to more than one language since a single IDL file can serve as the common source of truth for API specifications on both platforms. Hence, adding/modifying APIs requires the programmer to describe the API specifications in the shared IDL file and regenerate the bindings for all the desired languages with a single command. This means that the level of effort required from the users remains the same while the number of supported languages grows. Hence, adding support for a new language in multiple SDKs requires minimal effort once the generator for that language is added to the tool. Luckily, the front-end parts of the generator, such as the IDL parser and AST are reusable when adding new languages. As a result, adding support for new languages means adding a new language back-end that traverses the AST and generates bindings. From the business perspective, the generator allows Mapbox to reach new markets and platforms faster by adding new supported languages to the existing generators.

Lastly, automating bindings generation helped Mapbox scale the engineering teams as it brought a certain level of separation of concern. In fact, the pool of talents with proper knowledge of both C++ and platform languages is small. Hence, it is significantly easier to hire C++ and platform engineers who focus solely on doing one thing and doing it well. With the generated bindings, the team structure for each product can now be divided into three separate teams: native C++, iOS, and Android. Each team has become strongly specialized and can scale independently now as compared to the previous situation that required engineers to work with multiple languages.

In addition to bindings, we can also generate skeleton code for test cases. Feature telemetry code to track the usage of the APIs.

6.3 Limitations

Despite the benefits that the code generator provides, we also received multiple concerns related to the proposed solution:

- **Learning curve:**
  According to an engineering manager, the Interface Description Language adds an extra challenge for onboarding new engineers even though its syntax and constructs are relatively simple compared to other modern programming languages. In fact, the generator is for internal use only, which means that no engineers outside Mapbox have any experience with it. It also means no default integration support from popular IDEs and text editors. Hence, IDL integration with text editors and IDE requires additional work from the internal teams. As a result, the support for IDL syntax highlight and code formatting is limited, which is often considered inconvenient to some developers when editing or refactoring the IDL code. Besides the lack of integration support, another complaint is that error messages produced by the generator are not always informative, especially syntax error messages. Thus, a learning curve certainly exists for newcomers to use the tool effectively.

- **Ergonomics:**
  According to an iOS engineer, the IDL is not sufficiently expressive for exposing idiomatic APIs on the iOS platform in many cases. For example, `string` type is typically used to represent a `URL` in C++, while `NSURL` is a preferred type for URL on Apple platforms. As a result, URL is exposed as a `string` type in the IDL since there is no well-known equivalent of `NSURL` type in C++. The engineer strongly believes that the IDL should have been designed to be a union of the languages it generates rather than merely an intersection. However, this goal is tricky to achieve since many concepts and idioms in one language do not translate well to others. In fact, poor ergonomics is one of the main trade-offs of the generated APIs. To tackle this problem, platform engineers often have to manually wrap the non-idiomatic parts of generated APIs in another layer that feels intuitive to developers in several situations.

- **Workflow:**
  The workflow of the generator biasedly caters towards incremental API-driven software development. This means that the ideal workflow is that developers first specify the APIs to be exposed using the IDL, then generate the code

and provide the implementation class to develop the feature from the ground up. However, the workflow of the generator is sub-optimal when exposing a large existing C++ library to other languages in one pass. First, the clients must select the set of APIs to be exposed and write IDL code to expose those. Then they have to generate the APIs and bindings code. After that, they must write an adaptor class that connects the generated C++ class with the existing implementation. This adaptor can be as trivial as redirecting method calls, but sometimes it has to convert data structure between 2 distinct C++ data types, and each exposed method requires one such adapter piece of code. Even though writing adapter code manually between 2 C++ APIs does not require as much effort as writing language bindings, this step is repetitive and certainly requires some bandwidth, according to a C++ engineer. He believes this step is redundant and should be automated by an extension that can parse C++ APIs directly and generate everything without manual effort.

- **Performance:**
  Other weaknesses of the code generator are the inflexibility and the lack of awareness of the context where the exposed API is used. In most common situations, the benchmarking performance results of the generated code are on par with the manually written one. However, writing bindings by hand certainly provides programmers with a greater extent of flexibility. As a result, programmers can flexibly fine-tune and tweak the performance of the bindings at their own will. For example, `ArrayList<T>` is our generator's Java representation of the IDL type `array<T>`, which means that the generated Java APIs always use `ArrayList` regardless of the inner generic type. However, using Java's primitive array is substantially faster for marshaling when the inner type is primitive. In fact, marshalling a C++ `std::vector<int>` to a primitive Java array of integer(`int[]`) can be **26,3 times faster** than between a Java `ArrayList<Integer>` and a C++ `std::vector<int>` as shown in Table 3.

Table 3: Performance benchmark between ArrayList and Array in JNI

Test case	ArrayList<Integer> -std::vector<int>	int[]-std::vector<int>
Sending a vector of 1000 integers from C++ to Java	304468 ns	11604 ns

7 Discussion

As the generator enhanced the velocity, consistency, and scalability of our projects, it also showed weaknesses in certain areas, such as developer experience. As a result, there are several improvement areas and potential directions that could be explored as a future extension of the project.

From the front-end side, user interaction with the generator can be improved by adding syntax highlighting, code format, or auto-completion plugins for the IDL in common IDEs and editors such as VSCode and JetBrain's IDEs. Besides that, we can boost the expressiveness of the IDL by adding more features to the language, such as introducing annotations to tweak the ergonomics of the generated APIs. Another idea is to add an extension that parses C++ headers and generates IDL or bindings directly from the metadata of the APIs. This extension would simplify the adoption process of the generator, especially within legacy C++ projects.

From the back-end perspective, it makes sense to invest in a proper C language generator for two reasons. Firstly, most languages provide interoperability with C APIs, but only a few, such as Carbon and Objective-C++ interoperate with C++. Interestingly, invoking C++ from Swift via the C wrapper layer might yield better performance than going through Objective-C because we can avoid the dynamic message dispatch overhead in Objective-C. As a result, supporting C APIs opens the door for easier interoperability between C++ and various languages. Secondly, exposing C++ libraries via a thin layer of C APIs is a good practice to provide ABI stability for the native libraries since C ABIs mostly remain intact while C++ ABIs often change over the years.

Performance-wise, it is worth experimenting with marshaling large, nested data structures by serializing them in fast data format such as Protocol Buffer, sending the data across the language boundary, and deserializing on the other side. Regarding the generated code, we believe it is a perfect place to inject code that tracks metrics about API usage to improve the products. The generator can also target other languages, such as producing binding code between Dart and Java/Objective-C via the Flutter platform channel.

8 Conclusion

In summary, the rise of multi-language software systems often adds complexity to the software development process. One challenge of such systems is to make different languages interoperate seamlessly with each other. Specifically, Foreign Function Interface code, which allows one language to call functions written in another, can easily become problematic as the software grows.

The goal of this thesis, which is to automate the FFI code generation, was achieved since the generated glue code contributes significantly to the productivity, stability, and scalability of multi-language software. As every coin has two sides, the generated code certainly shows its weaknesses due to its inflexible nature, which might require human intervention in certain edge cases. Nonetheless, the generated code works out-of-the-box and is beneficial in most common situations, which justifies the decision to avoid writing glue code by hand.

References

[1] M. Grichi, E. E. Eghan, and B. Adams. On the impact of multilanguage development in machine learning frameworks. *In Proc. 36th IEEE International Conference on Software Maintenance and Evolution(ICSME)* (2020), pp. 546–547.

[2] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and M. Luján. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Trans. Program. Lang. Syst*, vol. 40, no. 2, art. 8 (May 2018), pp. 8:2-8:5.

[3] M. Furr, and J. S. Foster Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst*, vol. 30, no. 4, art. 18 (July 2008), pp. 18:1-18:5.

[4] A. Cleary, S. Kohn, S. G. Smith, B. Smolinski. Language Interoperability Mechanisms For High-Performance Scientific Applications. *In Proc. 1998 SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, vol. 99, SIAM (July 1999), pp. 1–10.

[5] T. Malone. Interoperability in programming languages *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal* 1981, vol. 1, iss. 2, art. 3, 2014, pp. 1-6.

[6] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A Comprehensive Study on Real World Concurrency Bugs in Node.js. *In Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, (Urbana-Champaign, IL, USA) (ASE 2017), pp. 520-522.

[7] A. Neumann, N. Laranjeiro, J. Bernardino. An Analysis of Public REST WebService APIs. *IEEE Trans. Serv. Comput*, vol. 14 (2021), pp. 957.

[8] P. Gomes-Soares. On remote procedure call. *In Proc. 1992 Conference of the Centre for Advanced Studies on Collaborative Research* (1992), pp. 218-226.

[9] S. Kiraly, S. Szekely. Analysing RPC and Testing the Performance of Solutions *Informatica*, vol. 42 (2018), pp. 555-558.

[10] C. Lattner, V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. *In Proc. International Symposium on Code Generation and Optimization* (2004), pp. 75-86.

[11] K. Atkinson, M. Flatt, G. Lindstrom. ABI Compatibility Through a Customizable Language. *Sigplan Notices - SIGPLAN. 46* (2010), pp. 147-156.

[12] H. Muhammad, R. Ierusalimschy. C APIs in Extension and Extensible Languages. *Journal of Universal Computer Science*, vol. 13, no. 6 (2007), pp. 839-853.

[13] Mapbox website. https://www.mapbox.com. Accessed 04/2022.

[14] D. Landman, A. Serebrenik, J. J. Vinju. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. *IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), pp. 507-518.

[15] M. Abidi, M. S. Rahman, M. Openja, F. Khomh. Are Multi-Language Design Smells Fault-Prone? An Empirical Study. *ACM Trans. Softw. Eng. Methodol*, vol. 30, no. 3, art. 29 (July 2021), pp. 5-9.

[16] J. Generowicz, P. Mato, W. Lavrijsen, M. Marino. Reflection-Based Python-C++ Bindings. *In Computing in High Energy Physics and Nuclear Physics 2004. Article, CERN (European Organization for Nuclear Research) and LBNL (Lawrence Berkeley National Laboratory)* (2004), pp. 1-4.

[17] D. Beazley. An Extensible Compiler for Creating Scriptable Scientific Software. *International Conference on Computational Science* (2002), pp. 824–833.

[18] T. Ravitch, S. Jackson, E. Aderhold, B. Liblit. Automatic generation of library bindings using static analysis. *In Proc. 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009). Association for Computing Machinery, New York, NY, USA*, pp. 352–362.