software

MDPI

Article

MetaFFI-Multilingual Indirect Interoperability System

Tsvi Cherny-Shahar * and Amiram Yehudai *

Blavatnik School of Computer Science and AI, Tel Aviv University, Tel Aviv P.O. 69978, Israel
* Correspondence: tsvic@mail.tau.ac.il (T.C.-S.); amiramy@tau.ac.il (A.Y.)

Abstract

The development of software applications using multiple programming languages has increased in recent years, as it allows the selection of the most suitable language and runtime for each component of the system and the integration of third-party libraries. However, this practice involves complexity and error proneness, due to the absence of an adequate system for the interoperability of multiple programming languages. Developers are compelled to resort to workarounds, such as library reimplementation or language-specific wrappers, which are often dependent on C as the common denominator for interoperability. These challenges render the use of multiple programming languages a burdensome and demanding task that necessitates highly skilled developers for implementation, debugging, and maintenance, and raise doubts about the benefits of interoperability. To overcome these challenges, we propose MetaFFI, introducing a fully in-process, plugin-oriented, runtime-independent architecture based on a minimal C abstraction layer. It provides deep binding without relying on a shared object model, virtual machine bytecode, or manual glue code. This architecture is scalable (O(n) integration for n languages) and supports true polymorphic function and object invocation across languages. MetaFFI is based on leveraging FFI and embedding mechanisms, which minimize restrictions on language selection while still enabling full-duplex binding and deep integration. This is achieved by exploiting the less restrictive shallow binding mechanisms (e.g., Foreign Function Interface) to offer deep binding features (e.g., object creation, methods, fields). MetaFFI provides a runtime-independent framework to load and *xcall* (Cross-Call) foreign entities (e.g., getters, functions, objects). MetaFFI uses Common Data Types (CDTs) to pass parameters and return values, including objects and complex types, and even cross-language callbacks and dynamic calling conventions for optimization. The indirect interoperability approach of MetaFFI has the significant advantage of requiring only $2n$ mechanisms to support n languages, compared to direct interoperability approaches that need n^2 mechanisms. We developed and tested a proof of concept tool interoperating three languages (Go, Python, and Java), on Windows and Ubuntu. To evaluate the approach and the tool, we conducted a user study, with promising results. The MetaFFI framework is available as open source software, including its full source code and installers, to facilitate adoption and collaboration across academic and industrial communities.

Keywords: multilingual; cross-language; interoperability; foreign-function-interface; system

check for updates

Academic Editor: Haipeng Cai

Received: 8 June 2025
Revised: 8 August 2025
Accepted: 21 August 2025
Published: 26 August 2025

Citation: Cherny-Shahar, T.; Yehudai, A. MetaFFI-Multilingual Indirect Interoperability System. *Software* **2025**, *4*, 21. https://doi.org/10.3390/software4030021

Correction Statement: This article has been republished with a minor change. The change does not affect the scientific content of the article and further details are available within the backmatter of the website version of this article.

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Programming language interoperability is a popular and well-known practice [1–4]. In this work, a programming language refers to the language used to write the logic layer of a program, and a language is defined as a pair of syntax and runtime. Interoperability

between languages is a long-standing challenge that has become more acute in recent years, as applications require more functionality and performance, requiring more complex software and hardware [2–6]. Therefore, it is improbable that a single programming language or runtime can adequately serve an entire application, and in some cases, confining an entire application to one programming language is infeasible. Since the emergence of the first programming languages in the late 1950s, hundreds of new languages have been developed, some for general purposes and others for specific domains. Despite the availability of a wide variety of languages and runtimes for different tasks, it is difficult to use more than one programming language within a single application [2–4], as languages typically do not support interoperability between them or only with a limited number of languages.

Moreover, the reluctance to use multiple programming languages results in the reimplementation and rewriting of existing code from one language to another (e.g., porting), rather than reusing the existing code, even if the runtime of the original library offers better performance, maintainability, and popularity [7].

Although the benefits of reusing existing code are evident, reusing code between multiple languages in a single application is a complex and error-prone task in itself, as this kind of development obliges developers to devote considerable effort to cross-language techniques that are difficult to use and maintain and are prone to error and vulnerabilities [2,8–11].

Several solutions have been proposed to address this problem:

- Foreign Function Interface (FFI) [12] enables the invocation of cross-language functions, which we refer to as *shallow binding*, as it allows access to functions, but not to objects, fields, globals, etc.
- Virtual machines such as JVM [13] (Java Virtual Machine) and Microsoft CLR [14] (Common Language Runtime), facilitating direct access to foreign entities (functions, classes, fields, etc., in the guest language), which we refer to as *deep binding*. The *guest language* is the one being called, while the *host language* is the one initiating the call. This approach applies only to languages compiled to the virtual machine's *executable code* and running within the virtual machine runtime.
- Language-independent object models such as CORBA (Common Object Request Broker Architecture) [15] and COM (Common Object Model) [16], which provide access to objects in multiple languages by specifying a well-known structure of the interoperable objects. This approach is restrictive and might require from a programming language features it does not necessarily support.

Most existing solutions for language interoperability focus on binding a pair of languages (e.g., Python CTypes binding Python to C, JNI binding C and Java, SWIG [17] and more) or a small set of languages (e.g., VM languages that compile to the same VM bytecode). Many papers recognize C as a popular interoperable programming language, such as [1,3–6,9,18–21]. In [7], we analyze 173 interoperability tools showing that C is in fact the most interoperable PL. Furthermore, other solutions are challenging and cumbersome to use [1,4,10]. Chisnall [6] highlighted the significance and difficulties of language interoperability in 2013. As the number of languages grows, encompassing both general-purpose and domain-specific languages, the selection and ability to select the appropriate tool for each task become more crucial and more complicated.

To address the challenges mentioned above, we introduced *simple interoperability* [7]—an easy-to-use oriented interoperability model focusing on minimal developer effort, comprising four essential features required to facilitate user-friendly interoperability between arbitrary PLs. (In contrast, traditional "complex" interoperability models like CORBA/COM demand

extensive manual interface code and strict constraints.) This concept is derived from a large-scale empirical study [7] and is further explained in Section 4:

1. Host-Only Coding—Use guest entities without needing to know or modify the guest language.
2. No Manual IDL—The user is not required to write an Interface Definition Language (IDL) specification for interoperability.
3. Automatic Runtime Management—The user does not have to manage guest runtime initialization or other internals.
4. Mappable Common Data Types—The system provides a set of data types that can be used by all languages and mapped to each supported language's native types.

Based on *simple interoperability* specifications, we propose MetaFFI, a pluggable in-process indirect-interoperability system that supports simple interoperability as defined above while providing interoperability between arbitrary programming languages. The choice of these four features is explained and detailed in the empirical study on the multilingual development and interoperability of programming languages [7]. The empirical study shows the popularity of multi-PL among open source projects and interoperability tools. It identifies recurring challenges that hinder practical interoperability and, based on the large-scale data analysis, distills four recurring pain points. These lead to the definition of *simple interoperability* and the set of features presented above, which we elaborate on in Section 4.

The MetaFFI interoperability concept is derived from the exported entities' loading mechanisms of dynamic libraries provided by popular operating systems (e.g., `LoadLibrary` or `dlopen` and `GetProcAddress` or `dlsym`). In the same manner an application can load a binary, platform-specific, dynamic library and import its exported entities, MetaFFI provides a layer on which to load modules and their entities in multiple formats while providing a uniform API.

MetaFFI enables deep binding capabilities using only shallow binding mechanisms, facilitating the development of independent modules irrespective of the language of implementation. To use MetaFFI, the user performs the following steps:

- Loading the runtime using the `load_runtime` function.
- Loading the module using the `load_module` function, which is analogous to `LoadLibrary` in Windows or `dlopen` in Linux.
- Loading the required foreign entities using the `load_entity` function, which is analogous to `GetProcAddress` in Windows or `dlsym` in Linux.

MetaFFI adopts an indirect approach to link languages via a common C-interfaced module, rather than directly connecting them. This reduces the number of FFI and interoperability libraries needed to support n languages in a project from $O(n^2)$ to $O(n)$. In addition, MetaFFI's modular plugin design allows developers to extend the programming language support by implementing up to three C interfaces, which facilitate the interoperability between the newly added programming language and all existing supported languages. The complexity of adding a new language does not depend on the number of languages already supported. Plugins are integrated by implementing their C interface. The details of the plugin design are discussed in Section 6.

To evaluate the effectiveness of our framework design, we implemented a fully functional prototype. We carried out experiments on the system using Go, Python3, and OpenJDK Java, on Windows 11 and Ubuntu 22.04 platforms. The test suite covers both user-defined source code and third-party libraries.

Unlike existing interoperability systems such as CORBA [22], COM [16], JNI [23], and Truffle/GraalVM [5], MetaFFI introduces a fully in-process, runtime-independent ar-

chitecture based on a minimal C abstraction layer. It provides deep binding without relying on a shared object model, virtual machine bytecode, or manual glue code. This architecture is scalable (O(n) integration for n languages) and supports true polymorphic function and object invocation across languages. Our key innovations are summarized in Section 2.

To promote transparency, reproducibility, and broader adoption, we provide the complete source code and installers for MetaFFI under a permissive license. These resources are publicly available at the official MetaFFI repository: https://github.com/MetaFFI (accessed on 22 August 2025).

The rest of this paper is organized as follows: Section 2 clearly lists the system novelties and contributions. Section 3 discusses previous and related work with respect to our research. Section 4 describes the research and engineering methodology used in the design and development of MetaFFI. Section 5 presents MetaFFI with a usage example and explains the MetaFFI API. Section 6 details the MetaFFI system, including its design, mechanisms, components, common data types, and limitations. Section 7 presents the user study we conducted to evaluate MetaFFI and the proof of concept tool. Finally, Section 8 concludes the paper.

2. Scientific Contributions and Novelty

MetaFFI builds upon several well-known interoperability paradigms, including Foreign Function Interfaces (FFIs) [24], embedding APIs, virtual machine runtimes like JVM and CLR [14,25,26], and object models such as CORBA [22] and COM [16]. However, existing approaches suffer from major drawbacks: pairwise coupling (FFI), constrained language support (VMs), and complex schema enforcement (object models).

Key Contributions: This work presents a novel, in-process multilingual interoperability system with the following contributions:

- XCall Abstraction: We introduce *XCall*, a universal C-based function-pointer abstraction for cross-language calls. XCall generalizes the concept of a function vtable entry to an *ABI-agnostic, flat C function-pointer with an associated context*, enabling the invocation of foreign functions, methods, getters/setters, and callbacks with full polymorphism. Unlike COM [16] vtables, or a model that is tied to specific object models [22,27], XCall is a minimal interface that works uniformly across languages without imposing a shared object layout.
- Common Data Type System (CDTS): We design a *language-neutral, plugin-extensible-type system* to support rich data interchange across languages. CDTS defines a shared schema of core types (scalars, arrays, strings, callables, etc.), but plugins are free to support only a subset or extend the set locally to accommodate language-specific constructs. Crucially, CDTS provides opaque handles (a way to refer to a resource without knowing or seeing how it works inside), from language-neutral references to runtime-managed entities, which act as a universal fallback. This allows user-defined classes, closures, and other non-standard types to cross language boundaries by reference, without requiring structural extensions or schema coupling. Unlike traditional IDLs (e.g., CORBA or COM VARIANT [28]) that enforce closed, static schemas, CDTS allows full interoperability without lowest-common-denominator constraints, and ensures deep binding of complex or dynamic objects without manual marshaling [15,16]. It also supports opaque handles for plugin-specific types, e.g., Go's complex64 can be used across languages either as a native complex type (if both plugins support it), or via a handle if not. Table A1 summarizes the types supported by the Common Data Type System (CDTS), which are safely marshaled across language boundaries and verified in all directions during system testing (see Section 6.3).

- Plugin-Based Architecture (O(n) Scaling): MetaFFI employs a *plugin hub-and-spoke model* in which each language runtime is connected via a shared C hub (the Cross-Language Linker Runtime, XLLR) using a plugin. This design reduces integration complexity from $O(n^2)$ cross-language bridges to $O(n)$ (Section 6). *No pairwise adapters or bespoke bridges* are needed; adding a new language requires implementing up to three well-defined C interfaces in a plugin for (1) runtime, (2) automatic IDL generation, and (3) compiling based on IDL. The implementation is independent of how many languages are already supported. This approach contrasts with tools like SWIG [17] (which generate n^2 glue code for n languages) or object brokers like CORBA that require global IDL schemas for each pair.
- Dynamic Capability Negotiation: We introduce a runtime *capability* mechanism that allows the host and guest plugins to negotiate call semantics and optimizations on a per-call basis. When a foreign entity is loaded, the host advertises supported features (memory management strategy, exception handling, calling convention, etc.), and the guest specifies the minimum required (Section 6.9). This fine-grained negotiation configures the XCall's behavior (e.g., using a faster calling convention if both sides support it, or optimizing string/array handling). Such per-invocation optimization has no equivalent in fixed-schema systems (for instance, COM always uses a fixed HRESULT error code convention). *All calls default to safe settings* for correctness, but if both sides agree on an optimization (like specific ABI), MetaFFI will transparently apply it to improve performance.
- No Manual IDL—Automatic Interface Generation: MetaFFI completely *eliminates manual interface description*. Instead of requiring developers to hand-write IDL files or wrapper code, MetaFFI provides tooling (IDL plugins for each language) to automatically extract the necessary interface metadata from existing code. Each language's metadata (functions, classes, types) is captured in a language-specific IDL automatically, and MetaFFI uses it to bind languages together at runtime. This dramatically lowers the adoption barrier [29] compared to IDL-based systems (e.g., SWIG [17], COM [30]), in which IDL is written by hand. It also improves maintainability, as changes in foreign APIs are picked up by regenerating the IDL rather than rewriting glue code.
- True In-Process Execution: MetaFFI operates with an *in-process interoperable runtime*. All languages execute within the same OS process, each running its native runtime, linked through XLLR. This is in contrast to alternatives like code migration (sending code as a string to another language's interpreter for execution [19]) or multi-language compilation frameworks (e.g., Haxe [31]) or GraalVM's Truffle [5], which require converting code into a common format. MetaFFI does not force a common bytecode or external server; it preserves each language's engine and simply mediates function calls and data between them. This ensures *maximal performance and compatibility*, since calls are direct C-ABI calls and data remains in-process, avoiding RPCs (Remote Procedure Calls) [32] or serialization.

These contributions are summarized in Table 1, which contrasts the design choices of MetaFFI with those of prior interoperability systems.

Table 1. Core Novel Contributions of MetaFFI compared to traditional interoperability systems.

Core Novelty	Description	Why It Is Unique
XCall Abstraction	Universal C function pointer + context representing any callable entity	Not tied to objects or vtables; supports polymorphic calls across languages
CDTS	Runtime-extensible, language-neutral type system with rich semantics	More expressive than CORBA IDL, COM VARIANT, or flat C structs
Low-Language Constraints	Works with any language with C-embedding or C-FFI	Unlike COM/GObject/Truffle, no object model or VM constraint
Plugin-Based Indirect Model	Central C hub replaces n^2 pairwise bridges with n plugins	Achieves scalable $O(n)$ integration model
No Manual IDL	IDL auto-generated per language, invisible to the developer	Avoids hand-written IDL burden in CORBA/COM
Dynamic Capabilities	Per-call negotiation of memory, reference, and type behavior	Enables optimized dispatch logic vs. rigid schema (e.g., HRESULT)
Cross-Language Callbacks	Functions can be passed as first-class callables across PLs	Not available as raw callables in CORBA/COM (only via object proxies)

3. Interoperability Background and Related Work

In the following section, we begin by analyzing various interoperability approaches to assess their capabilities and limitations. Subsequently, we examine real-world systems and research-based implementations that utilize different interoperability mechanisms (refers to techniques that allow multiple programming languages to operate within the same operating system process, including mechanisms such as FFI and runtime embedding). Through this comparative analysis, we evaluate the advantages and disadvantages of these systems while critically discussing how MetaFFI addresses the challenges associated with existing interoperability solutions.

3.1. Interoperability Approaches

We discuss and classify interoperability systems into four approaches:

- Foreign Function Interface (FFI).
- Embedding.
- Common runtime.
- Object model.

FFI enables a host programming language to call functions of a foreign PL by invoking executable code across language boundaries. Typically, an FFI is implemented for a specific pair of languages (from one host to one guest), providing only a half-duplex (one-way) binding interaction. For example, Python's `ctypes` and Java's JNI allow calling C functions from Python and Java, respectively. This approach is widely used but inherently pairwise [23,33], leading to many redundant bindings as the number of languages grows [12].

Embedding involves integrating a foreign language's runtime within the host application via an API or SDK. For example, a C++ application can embed the Python interpreter using Python's C API [34], or embed Lua as a scripting engine [35]. Embedding imposes minimal constraints on language selection, since any language with an embedding API can be hosted. However, it introduces significant limitations in data type compatibility, binding depth, and unidirectional interoperability (half-duplex) similar to FFI. In practice, embedded languages still require the user to manually define foreign function signatures and data types (often using a dedicated IDL or wrapper API for the calls).

Common runtimes take the opposite approach, imposing strict constraints by requiring languages to compile to a shared intermediate bytecode and execute within a unified runtime environment. For example, the Java Virtual Machine (JVM [13]) and Microsoft's .NET Common Language Runtime (CLR [14]) are designed to host multiple languages in one runtime.

This approach enables deep binding integration and eliminates the need for users to manually define foreign entities and types, since all languages share the runtime's object model and type system. However, it also sharply limits the set of supported languages and the execution environment—only languages targeting that specific VM can interoperate.

Object models attempt to strike a balance between the flexibility of FFI and the integration of common runtimes by defining standardized interfaces for objects. Instead of sharing a whole runtime, this approach requires each language to implement a common object model interface and structure. Notable examples include CORBA [15,22] and COM [16], which mandate that objects adhere to a predefined layout (e.g., vtable and interface GUIDs) and that interfaces be described in an IDL. These models provide full-duplex binding (calls in both directions) and structured interoperability through interface contracts. However, they are restrictive: the programming language must support all features required by the model (e.g., pointer-based vtables, reference counting, exception mapping), and interoperability is limited to objects that conform to the standardized interface. Moreover, as with FFI or embedding, the developer must still define foreign entities in an IDL and is constrained to the object model's data types. The type system is thus limited to the IDL's types, which often do not cover all native types of a given language, reducing flexibility, unless dynamic typing or opaque types are supported.

Each approach presents trade-offs between flexibility, complexity, and compatibility, influencing their suitability for different integration scenarios. Table 2 provides a detailed comparison of the four main interoperability approaches, highlighting their respective constraints, supported language characteristics, and binding capabilities. These attributes help frame the design choices and limitations discussed throughout this work. Concrete examples of each approach will be provided in subsequent sections where real-world systems and use cases are examined.

3.2. MetaFFI Approach

MetaFFI takes a unique approach by leveraging FFI and embedding, as these impose the fewest constraints on programming language selection while enabling deep integration and the possibility of full-duplex binding. Full-duplex interoperability is not provided automatically; each direction must be explicitly supported by appropriate mechanisms. Calling from the host requires a C-FFI interface, while exposing entities in the guest requires an embedding mechanism. Rather than relying on a standardized object interface or a shared runtime environment, MetaFFI uses a C-based hub that facilitates communication and enables a less restrictive type system for passing complex types and objects. By combining the broad compatibility of FFI and embedding with the deep binding capabilities typically associated with object models and common runtimes, MetaFFI achieves a more balanced trade-off between language flexibility and interoperability depth.

A central enabler of this flexibility is the Common Data Type System (CDTS). The CDTS defines a shared schema of core types, such as scalars, strings, arrays, and callable references, but plugins are not required to support the entire set. Instead, each plugin may adopt only the subset relevant to its host language, or extend it with custom types if needed. For types that cannot be directly represented across runtimes, the CDTS provides opaque handles: references that preserve the identity of the original entity and are interpreted by the plugin of its originating language. This architecture supports interoperability of complex or dynamic objects, including user-defined classes and closures, without structural coupling or schema centralization. Thus, MetaFFI maintains strong type fidelity without requiring strict alignment between all participating languages.

Table 2. Programming language interoperability approaches.

Aspect	Foreign Function Interface (FFI)	Embedding (SDK/API)	Object Models	Common Runtime (e.g., VM)
Supported PL Set	No conceptual restrictions		The object model standards may not align with all PL paradigms	Limited to PLs running in the same runtime. PL pair must compile and execute in the same runtime.
Restrictions on Host Language	Host PL must be able to interact with binding mechanism PL		PL must implement or interact with standardized object model interfaces	PL must compile and run in the shared runtime
Restrictions on Guest Language	No conceptual restrictions		PL must support object model standards structure	PL must compile and run in the shared runtime
Binding Directionality	Half-duplex, host to guest		Full-duplex	
Limitations on Bindings	Data types must be recognizable by both PLs Complex data types, different memory management, and threading models can impact performance and limit bindings		The object model standards may not align with all PL paradigms	No conceptual restrictions
Restrictions on Host Module Calling Foreign PL	No conceptual restrictions			Module must be compliant with the runtime
Restrictions on Guest Module Called by Foreign PL	Must expose functions using conventions compatible with the FFI		Module must implement the interfaces as expected by the object model standard	Module must be compliant with the runtime
Restrictions on Entities	Direct memory access is not supported Language-specific constructs are not supported (e.g., closures, objects with complex lifecycles)		Limited to entities exposed through the standardized object model interfaces	No conceptual restrictions

For example, the Go programming language supports complex numbers (e.g., `complex64`) as first-class types. A MetaFFI plugin for Go may extend CDTS with a new `cdt_complex` type to represent these values directly. Meanwhile, other plugins, such as Java or Python, can either treat the value as an opaque handle or, optionally, implement their own `cdt_complex` support to interoperate natively. If both the Go and Python plugins recognize and support the same CDTS extension, a Python `complex` value can be passed to Go and converted to a `complex64`, and vice versa. This plugin-scoped extensibility preserves interoperability while avoiding global schema constraints or forcing all languages to support every type.

3.3. Comparison of Interoperability Systems

To systematically compare different interoperability systems, Table 3 presents an evaluation based on several key criteria. These criteria assess the capabilities, limitations, and requirements of each system. Due to space constraints, the table is split into two parts, with the second part continuing the comparison using the same criteria. Details about the various interoperability systems appear after the table.

3.4. Analysis of Interoperability Systems and Comparison to MetaFFI

LibFFI (https://sourceware.org/libffi/, accessed on 22 August 2025) is a runtime library that enables calling foreign functions using C-FFI (Foreign Function Interface mechanism for calling C functions from a host language), making it a fundamental tool for interoperability using the FFI approach. However, it has several limitations. It provides only half-duplex binding, meaning calls can only flow in one direction, and it lacks deep binding, limiting access to complex objects and structured data from the host language. Multi-language interoperability is also inefficient, requiring separate bindings for each language pair, leading to $O(n^2)$ complexity, assuming n languages that provide C-compliant functions that can work with LibFFI. MetaFFI overcomes these limitations using a central C-based hub to interconnect languages, reducing complexity to $O(n)$ instead of relying on direct bindings like LibFFI. Also, unlike LibFFI, which relies only on the FFI approach, which provides half-duplex binding, MetaFFI employs both FFI and Embedding approaches to provide full-duplex binding and callback support. Moreover, MetaFFI relies on the Common Data Type (CDTS) system, instead of the C type system, which provides deep binding integration. By eliminating the need for manual function signature definitions and enabling dynamic cross-language interaction, MetaFFI provides a more scalable and flexible interoperability solution.

SWIG is a compile-time solution that generates interface bindings from C/C++ to multiple languages [17]. It automates the generation of C/C++ code necessary for embedding and establishing half-duplex bindings, thereby eliminating the need for users to manually write code for each of the N required binding mechanisms. In this sense, SWIG serves as the counterpart to LibFFI, whereas LibFFI uses C as a guest, and SWIG uses C as a host. However, unlike LibFFI, SWIG is a code generator that relies on embedding mechanisms.

MetaFFI, by contrast, utilizes a C-based hub, allowing C to function as both a host and a guest. Furthermore, unlike SWIG, which operates at compile time, MetaFFI predominantly performs embedding at runtime, eliminating the need for compile-time code generation except in languages that require explicit export directives to expose functions to C, such as the `//export` annotation in Go or `__declspec(dllexport)` in C++. Additionally, while SWIG (due to its embedding approach) only supports half-duplex interoperability, MetaFFI integrates both approaches to enable full-duplex binding.

Table 3. Comparison of interoperability systems.

Criteria	MetaFFI <<ffi+embed>>	LibFFI <<ffi>>	COM <<Object Model>>	CORBA <<Object Model>>
Host PL Requirements	C-FFI (Plugin required)	C-FFI	Supports COM object standard interface	Supports CORBA object standard interface
Guest PL Requirements	C-embeddable runtime	C (or ability to create C-compliant function)	Ability to implement COM object standard interface and structure	Ability to implement CORBA object standard interface
Binding Directionality	Full duplex **without standard object**	Half duplex	Full duplex	Full duplex
Scaling (Full-duplex bindings or N PLs)	$O(n)$ **central C-based hub by employing FFI+Embedding**	Not applicable. Assuming N PLs have C-compliant functions—$O(n^2)$	$O(n)$ (common binary object model)	$O(n)$ (common object broker—ORB)
Callback Support (Passing cross-PL callable as argument)	Yes **by XCall (C-Pointer and CDTS)**	Limited to C-compliant functions	Limited to COM objects	Limited to CORBA objects
Deep Binding (Complex types, methods, etc.)	Yes **without object model or common runtime**	No	Partial Only COM objects	Partial Only CORBA
Platform-Independent	Yes	Yes	Windows Partial support on other platforms	Yes
Special requirements on guest code	No	No	Unique interface implementations (classes, GUIDs) and registration process components	Unique interface implementation
Runtime vs. Compile	Runtime (depending on PL features, some require compilation time step)	Runtime	Requires steps at both compile-time and runtime	Requires steps at both compile-time and runtime
Requires manual guest entity definition	No (Plugin required) Without plugin: definition using API	Yes definition using API	Yes	Yes
Error handling	Native language error mechanism	C errors returns to called	Returns error code as HRESULT	using CORBA_Environment C-struct

Table 3. *Cont.*

Criteria	GObject Introspection <<Object Model>>	Truffle/GraalVM <<Common Runtime>>	PolySPIN <<ffi+embed>>	SWIG <<embed>>
Host PL Requirements	Support GObject object standard interface	GraalVM PL implemented using Truffle framework	Any PL pair that has a PolySPINNER (matcher) from host to guest	C/C++
Guest PL Requirements	Ability to implements GObject object standard interface	GraalVM PL implemented using Truffle framework		C-Embeddable runtime
Binding Directionality	Full duplex	Full duplex	Half duplex	Half duplex
Scaling (Full duplex bindings or N PLs)	$O(n)$ (common GObject interface)	$O(n)$	$O(n^2)$	Not applicable
Callback support (passing cross-PL callable as argument)	Limited to GObjects	Yes	Yes	Yes
Deep Binding (Complex types, methods, etc.)	Partial Only GObject	Yes	Limited dependence on type matching	No
Platform-Independent	Yes	Yes (All platforms on which GraalVM is implemented)	Yes	Yes
Special requirements on guest code	Unique GObject introspection implementation is required	No	No	No
Runtime vs. Compile	Requires steps at both compile-time and runtime	Runtime	Compile-time	Compile-time
Requires manual guest entity definition	Yes Automatic only for C Others use external tools	No	No	Yes
Error handling	Sets GError string	JVM Exception	Not explicitly specified	Native language error mechanism

That said, MetaFFI does support compile-time code generation when programming language constraints necessitate it, ensuring flexibility when runtime embedding is insufficient.

PolySPIN [29] is an FFI-based approach that eliminates the need for IDLs by performing direct type matching between different programming languages. Instead of relying on a predefined type system, PolySPIN analyzes class definitions, identifies equivalent structures, and dynamically generates interoperation code. While this approach avoids the rigidity of IDLs, it introduces significant scalability challenges where each new language pair requires a dedicated matcher, leading to $O(n^2)$ complexity for N languages. Furthermore, as an FFI-based system, PolySPIN lacks a common intermediate representation, making deep binding and full-duplex communication difficult. In contrast, MetaFFI automatically generates its IDL, where the IDL generation is independent for each language, resulting in $O(n)$ generators instead of PolySPIN's pairwise matchers. Additionally, PolySPIN inherits the limitations of FFI-only systems, which MetaFFI overcomes by combining FFI and embedding, enabling full-duplex interoperability and deep binding across multiple languages.

COM [16], CORBA [15], and GObject introspection [27] employ object model-based interoperability, ensuring deep integration through standardized interfaces. While these models provide full-duplex directionality and structured interoperability, they require strict conformance to predefined object models, limiting flexibility. Additionally, implementing interoperability in these systems often requires extensive manual setup and language-specific wrappers. MetaFFI achieves deep binding integration without enforcing rigid object model constraints using its CDTS, allowing broader language compatibility while maintaining full-duplex directionality and deep binding.

TruffleVM [5] follows the common runtime approach by implementing a language in GraalVM using the Truffle framework, enabling seamless interoperability by executing multiple languages within a unified runtime. However, this approach requires that all participating languages be compiled to and executed within the shared runtime, limiting the selection of supported languages. In contrast, MetaFFI adopts the FFI and embedding approaches, preserving the native runtimes of different languages and minimizing restrictions on language support. While still achieving deep binding interoperability using CDTS, MetaFFI avoids the limitations associated with runtime-dependent solutions.

LLVM IR, Haxe, CoLoRS [36], and code migration [19] differ from the previously discussed interoperability systems in that they are not traditional interoperability solutions. Instead of enabling direct interaction between languages at runtime, these approaches focus on alternative methods for facilitating cross-language execution. This fundamental distinction is why they were not included in Table 3. While LLVM IR provides a common intermediate representation for multiple languages, it does not maintain the original runtime environments. Similarly, Haxe compiles a single high-level language into multiple target languages rather than facilitating interoperability between existing languages. CoLoRS, on the other hand, provides a shared memory model across multiple runtimes, allowing language-independent object sharing but requiring an external server process for memory management. Unlike conventional interoperability systems, CoLoRS does not directly facilitate function calls or method invocation across languages but instead offers memory-sharing abstraction. Code migration takes a different approach by sending code as a string to be evaluated in the guest language, enabling execution but not true interoperability. It is limited to languages with runtime evaluation and lacks direct function calls, restricting interaction to one-directional execution. The following sections discuss these systems and their differences from MetaFFI.

LLVM IR [37] provides cross-language interoperability by compiling different languages into a common intermediate representation (IR), allowing function calls between

them. However, this approach requires significant manual effort to maintain interoperability, as it effectively reimplements language runtimes instead of leveraging existing ones. Furthermore, LLVM IR-based solutions do not provide runtime-level interoperability. Instead, they enable static cross-compilation, which is a fundamentally different concept. In a way, this approach can be seen as a form of a common runtime, but with a strict and limited set of supported languages. Unlike TruffleVM, which provides dynamic runtime interoperability, LLVM IR requires manual linking and type matching of the generated IR, adding complexity. The advantage of this approach is that once LLVM IR compiles to binary, the performance overhead of interoperability is minimized. In contrast, MetaFFI, through employing FFI and embedding, maintains native execution environments while supporting a larger set of languages, enabling runtime interoperability without requiring manual compilation and linking efforts.

Haxe (https://haxe.org/, accessed on 22 August 2025) is a high-level programming language that compiles into multiple target languages, effectively serving as a multi-language compiler. However, this is not true interoperability as it translates code rather than allowing multiple languages to interoperate within the same execution environment. Haxe forces developers to work within its ecosystem and does not support arbitrary cross-language calls between existing runtimes. MetaFFI, in contrast, enables actual interoperability between independent runtimes, allowing languages to communicate without needing to be rewritten in a specific syntax.

A paper by Wegiel et al. [36] presented Co-Located Runtime Sharing (CoLoRS), a system that provides transparent multi-process shared memory across different programming languages and execution runs, utilizing a language-neutral object/class model. CoLoRS manages shared memory through a dedicated server process, enabling language-independent object access. While this approach facilitates efficient cross-runtime data sharing, it introduces additional complexity, including process coordination, memory cleanup, and failure handling. Furthermore, because CoLoRS operates in an out-of-process shared memory model, it requires explicit synchronization mechanisms to maintain consistency across multiple runtimes. In contrast, MetaFFI enables direct in-process function calls between languages, preserving their native runtime environments and avoiding the overhead and challenges associated with multi-process communication.

The papers discussed in this section can be leveraged by MetaFFI to support more languages, but not all interoperability approaches are compatible with MetaFFI. For instance, a paper by Chiba et al. [19] proposes a framework for executing a "code block" in a foreign language by enhancing the "string embedding" technique, which involves writing the foreign code as a string and running it using the `eval` function of the foreign language. This technique is only suitable for host and guest languages that offer `eval` functionality.

The code migration [19] approach enables interoperability by sending code as a string to be evaluated in the guest language at runtime. This method avoids predefined bindings but is limited to languages with runtime evaluation (e.g., Python, JavaScript) and does not support compiled languages like C or C++ without an interpreter. It lacks direct interaction with the foreign language entities, making interaction one-directional and requiring string manipulation for execution. In contrast, MetaFFI enables direct function calls across languages while preserving their native runtimes, supporting both interpreted and compiled languages and interacting with foreign language entities.

In this section, we examine various approaches to programming language interoperability, classifying them into four primary models: FFI, embedding, common runtimes, and object models. Each approach presents trade-offs between language flexibility, integration depth, and runtime constraints. A comparative analysis of interoperability systems highlighted key challenges, such as the limitations of half-duplex FFI, the rigidity of ob-

ject models, and the strict dependencies imposed by common runtimes. We explored how different systems, including LibFFI, SWIG, COM, TruffleVM, and PolySPIN, address interoperability, as well as their respective limitations. Additionally, we analyzed alternative cross-language execution strategies, such as LLVM IR, Haxe, CoLoRS, and code migration, which focus on compilation or shared memory rather than true runtime interoperability. Through this evaluation, we demonstrated how the MetaFFI approach of central C-hub, CDTS, and IDL automatic generation bridges the gap between flexibility and deep binding integration by combining FFI and embedding approaches, enabling full-duplex directionality, deep binding, and broad language support while preserving native execution environments.

4. Methodology

This section describes the research and engineering methodology used in the design and development of MetaFFI. While Section 7 presents the empirical evaluation of MetaFFI's usability, the methodology described here focuses on the process by which we identified the problem, derived the system requirements, designed the architecture, and implemented the framework.

To address the lack of simple and easy-to-use programming language interoperability systems, we began by reviewing the literature, as discussed in Section 3. In parallel, we conducted an empirical study on GitHub and Stack Overflow to better understand how interoperability is practiced in real-world software development [7]. This study included a survey of 173 interoperability tools. The findings indicated that most existing tools were complex, required manual integration effort, and lacked general-purpose support for interoperability between arbitrary programming languages. In particular, developers were often forced to write glue code in the foreign language, define bindings through manual interface descriptions (e.g., IDLs), manage runtime configurations themselves, and work around mismatches in type systems across languages. These recurring challenges, observed consistently across open source projects and interoperability tools, informed the formulation of *simple interoperability*, a minimal set of features required to enable usable and scalable interoperability systems. This definition, further detailed in Section 4, serves as a foundational design goal of MetaFFI.

The survey also revealed that the C programming language was, by far, the most commonly used language for interoperability, both as a guest (primarily through FFI mechanisms such as LibFFI and Python CTypes) and as a host (using embedding mechanisms such as JNI and the CPython API). This insight guided our decision to use C as a central hub for MetaFFI, where each programming language connects through a plugin. The design reduces the interoperability complexity from $O(n^2)$ to $O(n)$, assuming n languages. Any language that implements a plugin becomes interoperable with all other supported languages automatically. A plugin provides full-duplex support by embedding the guest language to expose its entities and using C-FFI to access the MetaFFI API for making calls to other languages. This architecture also supports half-duplex integrations where only one direction is needed or possible due to the lack of mechanisms. To enable this, MetaFFI plugins implement up to three C interfaces, covering type marshaling, runtime management, and execution semantics.

We then addressed the challenge of passing arguments and return values across different language runtimes. Languages vary in their support for variadic arguments and multiple return values. In addition, the differences in type systems and the semantics of those types complicate cross-language value passing. We reviewed both shared memory models (e.g., COM, GObject) and message-passing models (e.g., Protocol Buffers (https://developers.google.com/protocol-buffers (accessed on 22 August 2025)), Thrift ((https:

//thrift.apache.org/ (accessed on 22 August 2025))). Shared memory models fit the in-process nature of MetaFFI but were often designed with C as either host or guest. This motivated the design of a new general-purpose type system, CDTS, that does not assume any specific host or guest language. CDTS is expressive enough to model a wide range of data types and inter-language interactions, including edge cases such as ragged arrays or nested structures. It supports value passing as well as reference passing when copying is not feasible or safe.

Because CDTS is designed for the general case, it may introduce unnecessary overhead in scenarios where such generality is not needed. To mitigate this, MetaFFI includes a runtime optimization mechanism called *capabilities*. When a host plugin loads a foreign entity, it provides its supported capabilities to the guest plugin. The guest plugin responds with the minimum required features for that specific entity. This exchange is used to configure the CDTS behavior for that cross-language call. By default, all capabilities are enabled, ensuring correctness. Disabling unused features is a performance optimization only and does not affect functionality. Capabilities are discussed further in Section 6.9.

To validate the architecture and implementation of MetaFFI, we built and tested it on both Ubuntu and Windows, with full-duplex plugins for three distinct programming languages: Python (CPython) as an interpreted language, Java (OpenJDK/Hotspot) as a JIT-based VM language, and Go as a compiled, memory-managed language with limited support for C-FFI and embedding. We confirmed correctness using a comprehensive suite of unit and system tests, including tests against third-party libraries such as BeautifulSoup (https://www.crummy.com/software/BeautifulSoup/, accessed on 22 August 2025), Pandas (https://pandas.pydata.org/, accessed on 22 August 2025), NumPy (https://numpy.org/, accessed on 22 August 2025), GoMCache (https://github.com/eko/gocache, accessed on 22 August 2025), and log4j (https://logging.apache.org/log4j/, accessed on 22 August 2025). Performance comparisons were conducted to make sure the overhead is not prohibitive. Usability was evaluated through an empirical user study, described in Section 7, which demonstrated substantial improvements in developer experience using MetaFFI.

5. Usage Example

MetaFFI provides the functionalities mentioned in the Introduction (i.e., `load_runtime` and `load_entity`), in the Cross-Language Link Runtime (XLLR) (Section 6.2) through C-API. Each supported language also provides a MetaFFI API that wraps the XLLR-API and abstracts the usage of C-FFI from the end-user, making the usage simpler by not requiring any C-FFI knowledge from the MetaFFI user.

The `load_entity` function returns an XCall structure that contains a C function pointer to the foreign entity entry point, and a context structure containing call-related data. In the event that an entity is a data member or attribute, the returned C function acts as a getter or setter. The parameters and return values are passed using Common Data Types (CDTs) detailed in Section 6.6. MetaFFI APIs also wrap the returned XCall structure to simplify usage.

The manual loading of entities works well when the number of entities is small, but in the case of large libraries, writing the code to load each entity is a hefty task. To mitigate this problem, MetaFFI supports a host compiler (Section 6.10.2) that generates the runtime and foreign entity loading code and creates wrapper entities in the host language that wrap the foreign entities.

Listing 1 demonstrates how Python3.11 can use Log4J, a popular Java logging library that has many ports to other languages (e.g., Log4Net, Log4CXX, Log4Python and more). The code uses the Python3.11 MetaFFI API, which wraps calls to the XLLR C-API.

Listing 1. Log4J from Python3.11.

```
1  # load JVM
2  runtime = MetaFFIRuntime('openjdk')
3
4  # load log4j
5  log4j_api_module = runtime.load_module('log4j-api-2.21.1.jar;log4j-core-2.21.1.jar')
6
7  # load getLogger() method to get a new logger
8  getLogger = log4j_api_module.load(
9  'class=org.apache.logging.log4j.LogManager,callable=getLogger',
10 [metaffi_string8_type],
11 [new_metaffi_type_with_alias(metaffi_handle_type, 'org.apache.logging.log4j.Logger')])
12
13 # load error() method in logger
14 perror = log4j_api_module.load(
15 'class=org.apache.logging.log4j.Logger,callable=error,instance_required',
16 [new_metaffi_type_with_alias(MetaFFITypes.metaffi_handle_type),
17 new_metaffi_type_with_alias(MetaFFITypes.metaffi_string8_type)],
18 None)
19
20 # create logger with getLogger()
21 logger = getLogger('pylogger')
22 perror(logger, 'Logging error from python!')
```

- Line 2 loads the JVM runtime.
- Line 5 loads the modules required for Log4J.
- Line 8 loads the `LogManager.getLogger()` static method, which returns a log4j logger instance. `load` expects a *function path* (Section 6.4.2) and MetaFFI types (Section 6.7) used as parameters and return values.
- Lines 13–16 load the `Logger.error()` method, which prints an error using Log4J
- Line 19 creates a new logger named *pylogger*.
- Line 20 calls the `logger.error()` method.

Instead of manually loading foreign entities, the user can execute the following in the terminal: `metaffi -c -idl log4j-api-2.21.1.jar -h`. This invokes the MetaFFI host compiler, generating Python entities that wrap log4j entities, creating a smoother user experience without writing code that interacts with the MetaFFI API.

The underlying mechanism of MetaFFI makes foreign entities available as C functions; these function pointers can be passed to a foreign entity as arguments. Therefore, MetaFFI also provides pass callback functions, as shown in Listing 2.

The Python3.11 function `call_callback_binary_op`, defined in Listing 2, takes as an argument a function that operates on two `int` values and returns an `int` value. The Java static method `add` sums the two integers and returns the result. In the final code segment, Java employs the MetaFFI JVM API to pass the `add` method to the `call_callback_binary_op` function:

- Lines 2,3—loads Python3.11 runtime.
- Line 6—loads `runtime_test_target.py` module.
- line 9—loads `call_callback_add` function.
- Lines 14,15—loads MetaFFI JVM runtime to handle callbacks.
- Line 18—loads "add" method.
- Line 21—wraps Java "add" method and makes it MetaFFI-enabled.
- Line 24—calls Python3.11 `call_callback_binary_op`, passing the Java `add` method.

Listing 2. Java callback method passed and called from Python3.11 function.

```python
# runtime_test_target.py
def call_callback_binary_op(bi_op_func: Callable) -> int:
res = bi_op_func(1, 2) # call binary function
return res # return result
```

```java
public static int add(int x, int y)  // java add
{
return x+y;
}
```

```java
// load python3.11 runtime
var runtime = new MetaFFIRuntime("python311");
runtime.loadRuntimePlugin();

// load runtime_test_target.py
var module = runtime.loadModule("./python3/runtime_test_target.py");

// load the python "call_callback_binary_op" function
metaffi.Caller callCallback = module.load("callable=call_callback_add",
new MetaFFITypeInfo[]{ new MetaFFITypeInfo(MetaFFITypes.MetaFFICallable) },
null);

// must OpenJDK runtime to handle callbacks
var javaRuntime = new MetaFFIRuntime("openjdk");
javaRuntime.loadRuntimePlugin();

// use reflection to get the "add" method
Method m = TestClass.class.getDeclaredMethod("add", int.class, int.class);

// make the method callable from MetaFFI
metaffi.Caller callbackAdd = api.MetaFFIRuntime.makeMetaFFICallable(m);

// call python passing the "add" function has parameter
callCallback.call(callbackAdd);
```

MetaFFI embeds multiple runtimes in the same process, but in many cases, these runtimes are isolated and have no direct access to other runtimes. However, the C runtime is shared among all runtimes and serves as a bridge between them. The CDT structure and the communication between the runtimes uses mixed shared memory and a message-passing model inspired by the Microsoft Object Linking and Embedding (OLE) model [38] and the Remote Procedure Call (RPC) model proposed by Nelson [32]. If the runtimes are isolated, a message-passing approach is employed, whereas if the runtimes can share memory, a shared memory approach is utilized. Details of runtime management and the invocation of cross-language functions are presented in Sections 6.2 and 6.8, respectively.

6. MetaFFI System

MetaFFI is a flexible and adaptable system that aims to indirectly facilitate multilingual compatibility. It is structured to fulfill the "simple interoperability" criteria as outlined in Section 1, and presented in more detail in [7]. Through MetaFFI, external components can be integrated and utilized within the primary language's source code. The system is designed in an indirect manner, ensuring that the incorporation of new languages is not constrained by the existing language base. In addition, the inclusion of a new programming language automatically ensures compatibility with all previously integrated languages. Nevertheless, it is essential for the runtime-executed code to have the capability to call C functions for accessing other languages, as well as to be accessible from C to enable full-duplex cross-language invocation.

MetaFFI offers a C interface API for multilingual communication:

- `load_runtime_plugin` and `free_runtime_plugin`—These functions load or unload a runtime (such as JVM, CPython, etc.).
- `load_entity` and `free_function`—These functions load a foreign entity from a specified module and return it as a C/XCall (which is an extension of the C function; hence, the XCall function pointer is a valid C function pointer) function and context as `void**`. `free_function` releases the function and context.
- `make_callable`—This function wraps a callable in an XCall structure.

The language support implementer should provide an API in the supported language, which wraps MetaFFI's C-API, to enhance the user experience. Currently, the system offers an API for users in Go, JVM, and Python3.11.

As discussed in Section 1 and shown in code Listing 2, MetaFFI has four steps:

- Loading the runtime (Section 6.2).
- Loading the module.
- Loading the foreign entity (Section 6.4).
- Calling the foreign entity (Section 6.8).

Code Listing 2 also shows the usage of cross-language callback.

6.1. System Design

MetaFFI is composed of two architectural layers: the runtime architecture (mandatory) and the compiler architecture (used only in specific scenarios). These layers work together to enable runtime-independent, plugin-based interoperability between programming languages. Figures 1 and 2 illustrate the overall structure and workflows of each layer.

Runtime Architecture (mandatory). Figure 1 illustrates the core runtime execution flow, where a host language (L_1) calls into a guest language (L_2). This architecture is always required when using MetaFFI.

The call originates from L_1 through the L_1 API module, which is part of the L_1 plugin and provides user-facing functions like `load_runtime`, `load_entity`, and `xcall_caller`. These functions internally delegate to MetaFFI's C-based runtime interface.

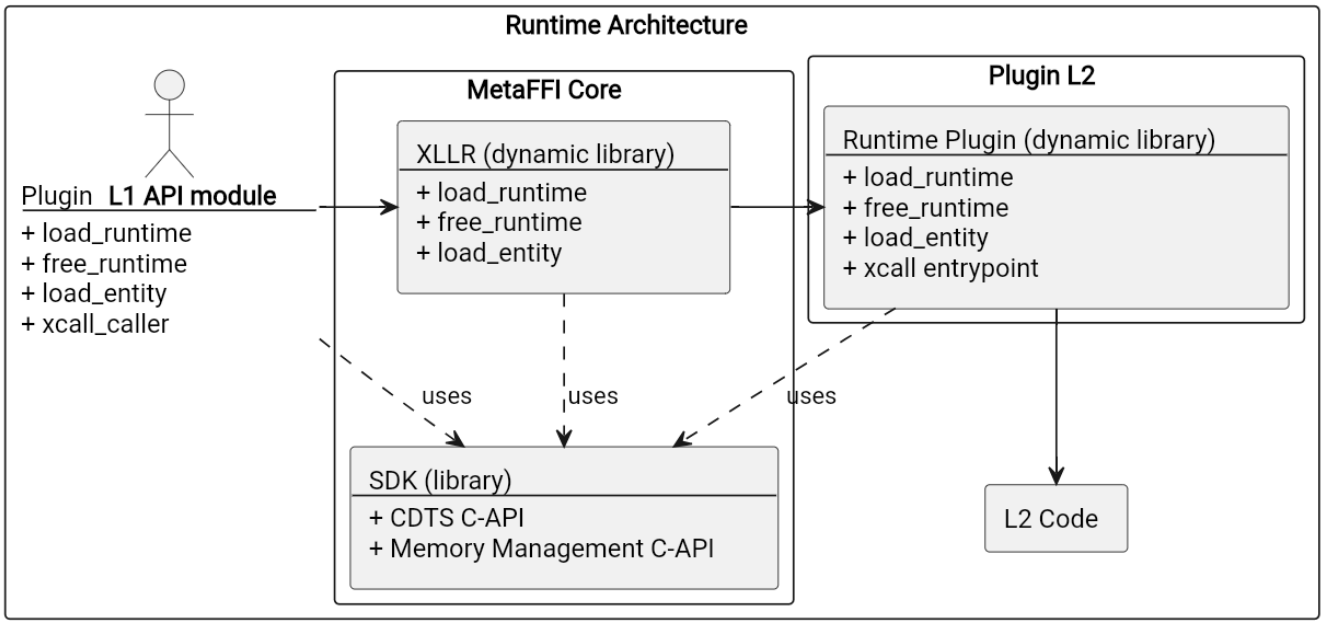


Figure 1. MetaFFI runtime architecture: L_1 (host) calling L_2 (guest).

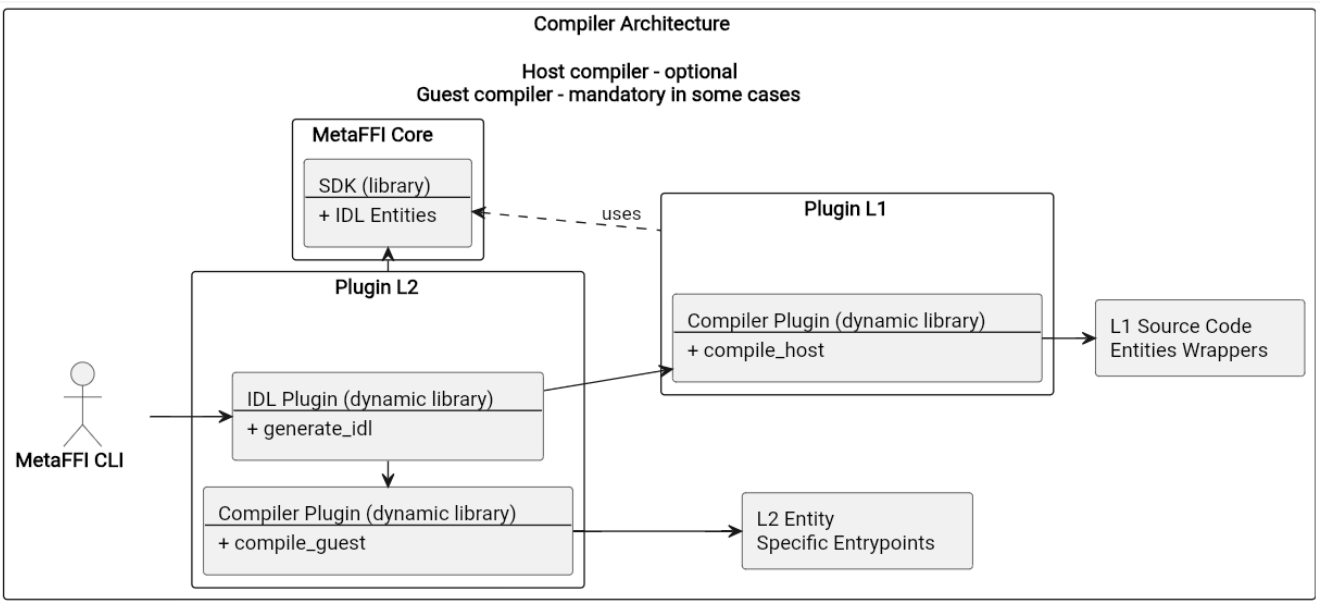


Figure 2. MetaFFI compiler architecture.

MetaFFI Core is responsible for coordinating the interaction. The XLLR dynamic library acts as a dispatcher that loads and manages plugins and delegates calls to them. It uses services from the shared SDK library, which provides APIs for memory management and MetaFFI's Common Data Type System (CDTS).

The actual logic for handling guest language behavior resides in the Runtime Plugin of L_2, which receives calls from XLLR and loads the relevant entities. It returns an `xcall` function pointer, a valid C function pointer representing the foreign entity, which can be invoked by the host through MetaFFI.

This design ensures complete decoupling: L_1 and L_2 only interact via standardized MetaFFI interfaces, and no language-specific knowledge is required in the host.

Compiler Architecture (conditional). Figure 2 presents the compiler layer of MetaFFI, which is not always required.

The L_2 compiler plugin is mandatory if the guest language's embedding mechanism requires explicit entry points in order to be invoked from C. For example, when using Go or Rust as L_2, functions must be exported to C via special keywords or wrappers (e.g., `export` in CGo). In such cases, the MetaFFI CLI calls the IDL plugin of L_2 to generate the intermediate representation (IDL Entities), which is then passed to the `compile_guest` function in the L_2 compiler plugin. This process outputs the necessary entry point code that makes guest language entities callable from C.

On the other hand, the L_1 compiler plugin is optional. It provides convenience by generating host-language wrappers (e.g., idiomatic Go or Java objects) that encapsulate calls to foreign entities. This is particularly useful when integrating with large foreign libraries or aiming for strongly typed development experiences in the host language. As with L_2, the MetaFFI CLI first calls the L_2 IDL plugin to generate the IDL, and then uses it as input to the `compile_host` function in the L_1 compiler plugin to produce wrapper code.

6.2. XLLR, Runtime Plugins, and Runtime Management

Cross-Language Link Runtime (XLLR) is the runtime component that handles the runtimes of the different languages and offers a C-API to load foreign entities (using MetaFFI plugins) that return XCall functions.

The XLLR C-API for runtime management, shown in Listing 3, consists of

Listing 3. C-API for runtime management.

```
void load_runtime_plugin(const char* runtime_plugin_name, char** err);
void free_runtime_plugin(const char* runtime_plugin, char** err);
```

The function `load_runtime_plugin` takes `runtime_plugin_name` as an argument, which is the *runtime plugin* to be loaded. XLLR loads the runtime plugin and forwards the call to the runtime plugin that provides the same API (excluding the `runtime_plugin_name` parameter).

The runtime plugins load (or free) their respective runtimes. For example, Python3.11's plugin load (or attach) the Python3.11 interpreter, while JVM's plugin creates (or attaches) the JVM. In case of an error, it returns to the caller via `err` as a heap-allocated null-terminated string.

The `load_runtime_plugin` and `free_runtime_plugin` functions implemented by the plugin must account for multiple invocations. XLLR ensures that concurrent calls to these functions are synchronized. If an error is returned, the caller is responsible for releasing `err`.

6.2.1. Memory Allocation and Deallocation

In a multilingual environment, it is common for different components to be built with various compilers. This is also true for C-FFI of different runtimes. While many programming languages can interoperate with C due to using the same Application Binary Interface (ABI), this does not imply that the toolsets (such as Visual C++, Clang, TDM, etc.) used to handle C are identical. Different toolsets may implement functions like `malloc` and `free` in different ways, which can lead to application crashes or undefined behavior if malloc and free from different toolsets are mixed. To address the variety of possible toolsets, any memory allocated and returned to the caller, such as error strings or heap return values, must be allocated using the `xllr_malloc` and `xllr_free` functions provided by XLLR. This ensures that the same toolset is used for both allocation and deallocation.

Memory Safety Guarantees

MetaFFI ensures memory safety by design: no language-owned entities ever pass directly through MetaFFI's runtime layer. Instead, all cross-language values are converted to CDTS representations, and any memory allocated by MetaFFI (e.g., for CDTS, error messages) is released by MetaFFI using the same allocator via `xllr_free`. Consequently, there is no risk of mismatched allocations across runtime boundaries.

The only case in which memory ownership must be externally managed is when a plugin returns a reference to a language-native entity. In such cases, the plugin ensures that the entity remains valid until a corresponding *releaser* function is invoked by the caller. This mechanism is equivalent to manual memory management patterns in systems like COM, where the user must invoke `Release()` on returned references. Failure to do so results in a leak, but this leak is strictly external to MetaFFI and analogous to those accepted in other interoperability systems such as CORBA and COM.

When the host programming language permits, this manual release can be automated by the plugin's API module. For example, in RAII-supporting (Resource Acquisition Is Initialization) languages like C++ and Rust, the foreign entity can be wrapped in a host-side object that invokes the *releaser* in its destructor, ensuring safe and automatic resource management.

We are currently exploring mechanisms to improve plugin-side ownership safety by optionally injecting runtime reference-counting or destructor hooks.

6.3. Correctness Justification

To evaluate correctness, we implemented comprehensive system tests across Python3, Java, and Go. These include type conversions, callbacks, lifecycle behavior, and third-party interoperability. The complete test matrix is included in Appendix B. All conversions listed in Table A1 have been validated in all directions (e.g., Python→Java, Java→Go), including multi-dimensional arrays and nested containers such as 3D byte arrays.

MetaFFI supports advanced callable features across languages, including named arguments, overloaded functions, inherited objects, and multiple return values.

6.4. Loading Foreign Entity

The XLLR C-API provides the following function, shown in Listing 4, to load a foreign entity and return its XCall structure:

Listing 4. C-API to load entities.

```
struct xcall{  void* pxcall_and_context[2]; };
struct xcall* load_entity(const char* runtime_plugin_name,
const char* module_path,
const char* function_path,
metaffi_type_info* params_types, int8_t params_count
metaffi_type_info* retval_types, int8_t retval_count,
char** err);
```

The call is forwarded to the runtime plugin (excluding the `runtime_plugin_name` argument).

The function expects the following arguments:

- `runtime_plugin_name`, which specifies the runtime plugin to use.
- `module_path` specifies the module the entity relies on (detailed in Section 6.4.1).
- `function_path` is a string declaring the location of the entity within the module (detailed in Section 6.4.2).
- `params_types` and `retvals_types` are arrays of `metaffi_type_info`, specifying the types the entity expects and returns (detailed in Section 6.7).
- `params_count` and `retval_count` are the sizes of the array params_types and retval_types.
- `err` is an `out` parameter (that is, the parameter is set by the called function) of a null-terminated string. If `err` $\neq$ `NULL`, then the function failed. The caller is required to free the error string using `xllr_free`.

6.4.1. Module Path

A module path refers to the module (or modules) managed by the specified runtime. The runtime plugin documentation should specify the required module path format. For instance, in JVM, the module path consists of a list of `.jar` files and/or directories containing the entity's dependencies. In Python3.11, the module path can be a directory containing the Python module, a path to a `py` file, or a package name.

6.4.2. Function Path

Windows `GetProcAddress` and Linux `dlsym` receive a string or ordinal to locate the exported function in the export table or the GOT (Global Offset Table), respectively. C++ also uses the export table using name mangling to "encode" additional information in the exported function name.

MetaFFI adopts the same basic idea, but an export table is not available in many programming language modules. Instead, MetaFFI uses the string to let the plugin know how to locate the entity using a simple list of attributes written in key–value pairs and tags encoded in a human-readable text separated by commas: $[Key_1 = Val_1, Tag_1 ... Tag_N, Key_N = Val_N]$. The runtime plugin documentation must document which keys and tags it requires. Although we strive to keep the keys uniform among the plugins, it is not mandatory.

For instance, the key `callable`, which is common to all existing plugins, denotes a name of a callable entity, such as a function, constructor, or a method. Another shared tag is `instance_required`, which signifies that the entity requires an instance and that the first parameter is that instance. The `global` key is used in Go to declare that a variable is in global scope, something that is not used in the JVM plugin. Likewise, the key `attribute` refers to an attribute in Python3.11 and is not used in the Go or JVM plugin.

Each runtime plugin specifies the key–value pairs and tags it expects when loading an entity. The function path is then passed to the runtime plugin, which uses this information to load the foreign entity and return an XCall struct, which contains a C function pointer to the foreign entity.

6.5. MetaFFI-Enabled Callable

The function `load_entity` takes the module path and the function path as input and returns a pointer to an XCall structure, which is used to invoke the entity. The function `make_callable` performs a similar task, but does not require the entity to be located in a module. Instead, it requires a callable entity within the runtime. `make_callable` in XLLR forwards the call to the plugin supporting the intended runtime, which knows how to handle a callable entity in its supported runtime, which in turn creates an XCall to call that entity. The function is defined in Listing 5.

Listing 5. C-API to make cross-language callable.

```
struct xcall* make_callable(const char* runtime_plugin_name,
void* pcallable_entity,
metaffi_type_info* params_types, int8_t params_count,
metaffi_type_info* retval_types, int8_t retval_count,
char** err);
```

The call is forwarded to the matching runtime plugin that exports the same signature (excluding `runtime_plugin_name`).

The parameter `pcallable` represents a callable entity in the target runtime. For instance, in Python3.11, it is `PyObject*`, which refers to a callable object. In JVM, it is a `jobject` that denotes a method.

6.6. Common Data Types

In the general case, MetaFFI assumes that all runtimes are isolated with a common shared memory, the XLLR, and its plugins. In order to construct a message containing all the arguments and return values, MetaFFI uses an array of Common Data Type (CDT) structures inspired by Microsoft's OLE (Object Linking and Embedding) VARIANT structure aimed at solving a similar goal of passing a message between processes on the same computer. CDT, as opposed to VARIANT, is designed specifically for the interoperability of different programming languages.

Unlike CORBA's IDL or COM's VARIANT, which rely on closed and statically defined type systems, MetaFFI's CDTS is designed to be flexible and extensible across languages while remaining backward-compatible. CDTS defines a core schema of common types, but individual plugins may implement support for only a subset of these types. Moreover,

plugins may locally extend CDTS with new types to suit language-specific needs, without requiring changes or coordination with other plugins. Crucially, CDTS provides an opaque handle mechanism that allows any language-specific entity, including user-defined objects, closures, or runtime values, to be passed by reference across language boundaries. This means even unsupported or plugin-specific types can interoperate via handles, with the receiving plugin responsible for interpreting the value. This design enables rich, language-aware interoperability without enforcing a lowest-common-denominator model or requiring structural changes to the CDTS itself.

Each CDT stores a MetaFFI data type. The different types are detailed in Section 6.7.

Code Listing 6 defines the structure `cdt`. The `cdt.type` field specifies the MetaFFI type of the CDT, which is an unsigned 64-bit integer. The `cdt.free_required` field indicates whether the data stored in the `cdt.cdt_val` field needs to be freed.

Listing 6. Common data type.

```
struct cdt
{
metaffi_type type; // The MetaFFI Type of the CDT---uint64
metaffi_bool free_required; // cdt_val's data need to be freed---uint8
union cdt_types cdt_val; // Union storing the relevant MetaFFI data type
};
```

The cdt.cdt_val field is of type`cdt_types`, which is a union presented in Listing 7. The `cdt_types` valid field is based on the value of `cdt.type`.

Listing 7. cdt_types union.

```
union cdt_types
{
metaffi_float32 float32_val;
metaffi_float64 float64_val;
metaffi_int8 int8_val;
metaffi_uint8 uint8_val;
metaffi_int16 int16_val;
metaffi_uint16 uint16_val;
metaffi_int32 int32_val;
metaffi_uint32 uint32_val;
metaffi_int64 int64_val;
metaffi_uint64 uint64_val;
metaffi_bool bool_val;
struct metaffi_char8 char8_val;
metaffi_string8 string8_val;
struct metaffi_char16 char16_val;
metaffi_string16 string16_val;
struct metaffi_char32 char32_val;
metaffi_string32 string32_val;
struct cdt_metaffi_handle* handle_val;
struct cdt_metaffi_callable* callable_val;
struct cdts* array_val;
};
```

6.7. MetaFFI Types

CDT supports 24 data types, detailed in Table 4.

Table 4. MetaFFI Types.

Numeric Types			
float32	float64		*bool*
int8	int16	int32	int64
uint8	uint16	uint32	uint64
String Types			
char8		string8	
char16		string16	
char32		string32	
Special Types			
handle		callable	
any			
Internal Types			
null		array	
size		type	

6.7.1. Numeric Types

The numeric types are passed by value. When there is no equivalent numeric data type in the target runtime, the plugin implementer has to choose a suitable conversion. For instance, Java does not support unsigned 64-bit integers, so the plugin can map them to signed int64 or use Java BigInteger. Alternatively, the plugin implementer can indicate that a certain type is not supported by returning an error, but this is not advisable.

Structure `cdt_T` represents a numeric type *T* and is defined in Listing 8.

Listing 8. cdt_T struct.

```
struct cdt_T
{
T val; // The value of type T
};
```

6.7.2. String Types

Like numeric types, string types are used to pass parameters and return values to the equivalent data type at the target runtime. While numeric types are always copied, string types might get copied, depending on the caller runtime support of C-String. In case the caller runtime can pass a null-terminated C pointer of the right encoding to the string parameter, there is no need to copy the string. Notice that if the parameter is considered an in–out or out-parameter, the caller plugin must take it into consideration if the string should be copied or not.

The structure `cdt_STRING_T` represents a string type *STRING_T** and is defined in Listing 9.

Listing 9. cdt_STRING_T struct.

```
struct cdt_STRING_T
{
STRING_T* val; // null terminated string
};
```

6.7.3. Handle Type

In MetaFFI, a handle is characterized as follows: If the handle is associated with the current runtime, it returns the original object; otherwise, it returns structure `cdt_metaffi_handle*`.

Structure `cdt_metaffi_handle` defined in Listing 10 represents a handle.

Listing 10. cdt_metaffi_handle struct.

```
typedef void* metaffi_handle;
struct cdt_metaffi_handle
{
metaffi_handle handle; // The void* representing the object
uint64_t runtime_id; // The runtime the handle originated from
void* release; // void((*)(cdt_metaffi_handle*)) C-function which releases the entity
};
```

A `handle` is a `void*` that refers to an entity from another language at runtime. The `void*` is not necessarily a valid pointer, but a representation of the object that its runtime can use to retrieve the original object. To determine the runtime associated with a `handle`, each runtime is assigned a distinct `runtime_id`, whose 64-bit integer is generated randomly and embedded in the runtime plugin. When a runtime registers an entity using the `cdt_metaffi_handle` struct, it assigns `handle` and `runtime_id` to the struct.

The release field points to a function that takes `cdt_metaffi_handle` and releases the object. It is important to note that a plugin developer should release the object after a function returns. Instead, the release function, which manages the object's lifetime, is defined by the user. Releasing the object does not mean that the object is deleted, as the object's original runtime determines when the object is actually being deleted.

The plugin implementer is responsible for mapping the `void*` to the entity at runtime. As an important note for implementers, in managed memory runtimes (e.g., Java, Python, Go), it is important to make sure that the managed entity is not deleted when used by an external runtime. The MetaFFI runtime plugin must prevent the entity from being deleted until the `release` function in the `cdt_metaffi_handle` is called.

Two simple examples of implementing a handle is in Python3.11 and JVM. In Python3.11, all objects are represented by pointers to a `PyObject` structure, so the handle value is simply a `PyObject*`. Python employs a reference counting scheme [39] to manage memory, so as to prevent the object from being garbage-collected; we need to increase the reference count using `Py_INCREF` when creating the handle, and decrease it using `Py_DECREF` when releasing the handle. In JVM languages, all objects are represented by pointers to a `_jobject` structure, so the handle value is simply a `_jobject*`. To prevent the object from being garbage-collected [40], we need to create a global reference using `NewGlobalRef` when creating the handle and delete it using `DeleteGlobalRef` when releasing the handle.

In Go, the situation is more complicated, as there is no C-compatible data type that can represent any Go object. In this case, we use a global object table, where the key is a random 64-bit unsigned integer, and the value is a Go object stored as an empty interface `interface`, which can hold any Go type. The handle value is then the key to the object table entry. Go uses mark and sweep garbage collection [41], but there is no direct method to mark an object. Therefore, as long as the object is in the table, the Go runtime will not collect it. The release function removes the entry from the table, thus freeing the reference to the object. Note that this technique requires using synchronization primitives for mutual exclusion, such as a reader/writer lock.

6.7.4. Any Type

The `any` type indicates that the CDT type is dynamic and determined in runtime. The CDT field `type` specifies the actual type.

6.7.5. Array Type

In programming languages, we can find several types of arrays:

- Classic array—fixed dimensions with the same length;
  example: `[1,2][3,4]`.
- Ragged array—fixed dimensions with different length;
  example: `[1][2,3][4,5,6]`.
- Mixed-dimension array—every element might be an array of different dimensions;
  example: `1,[2,3][[4,5][6,7]]`.

As MetaFFI tries to support all scenarios, arrays in CDT support mixed-dimension arrays (similar to lists in Python), which support all types of arrays. Every array in CDT has an array header of the CDTS structure, which can be held by CDT in the `cdt_types` union. The presented CDTS structure in Listing 11 holds an array of CDT. Therefore, every element can be either a data type or an array.

Listing 11. cdts struct.

```
struct cdts
{
struct cdt*     arr;
metaffi_size    length;
}
```

6.7.6. Callable Type

The `callable` type represents a C function that invokes MetaFFI XCall. The structure `cdt_metaffi_callable` is used to pass an XCall as an argument or return it as a result, as illustrated in Code Listing 2, where the Java method `add` is passed to the Python function `call_callback_binary_op` as a parameter and is called back from within Python.

The structure `cdt_metaffi_callable`, defined in Listing 12, contains the information required for a runtime plugin to create the proper CDT for a pointer to the C function that performs the XCall.

Listing 12. cdt_metaffi_callable struct.

```
struct cdt_metaffi_callable
{
struct xcall* val;

// the definition of the XCall
metaffi_type* parameters_types; // array of MetaFFI types
metaffi_int8 params_types_length; // length of parameters_types
metaffi_type* retval_types; // array of MetaFFI types
metaffi_int8 retval_types_length; // length of retval_types
};
```

The field `val` stores a pointer to the XCall structure that holds the C function pointer to the XCall and context. We further discuss the xcall structure in Section 6.8. The `parameter_types`, `retval_types`, and their respective lengths specify the signature of the XCall, similar to the types declared when loading a foreign entity using the `load_entity` function.

6.7.7. Null Type

The `null` type indicates `null` or `void` type, defining the foreign entity signature for the `load_entity` or `make_callable` functions. It is similar to passing a `NULL` pointer to indicate that there are no parameters or return values.

6.7.8. Size and Type Types

The `size` and `type` types are not CDT types in MetaFFI, but 64-bit unsigned integers that are used as `typedef` throughout the MetaFFI system. The `size` type is a data type for storing sizes, and the `type` type is a data type for storing MetaFFI types.

6.8. XCall Cross-Language Call

Functions `load_entity` and `make_callable` return an XCall structure, which is constructed of two elements: a C function pointer to the XCall entry point and a pointer to a runtime plugin-specific context structure. This simple two-element structure (a function pointer and context) is a universal abstraction for any callable entity across languages. Notably, XCall does not require the guest language to adopt any particular object layout or calling convention beyond a C-compatible interface. In other words, it generalizes COM's vtable entry (which is fixed to COM's object model) into a language-agnostic call mechanism. *Any function, method, or even closure* can thus be represented as an XCall, with the context pointer enabling callbacks and stateful closures to work uniformly.

The flow of cross-language function calls and callbacks using MetaFFI is illustrated in Figure 3.

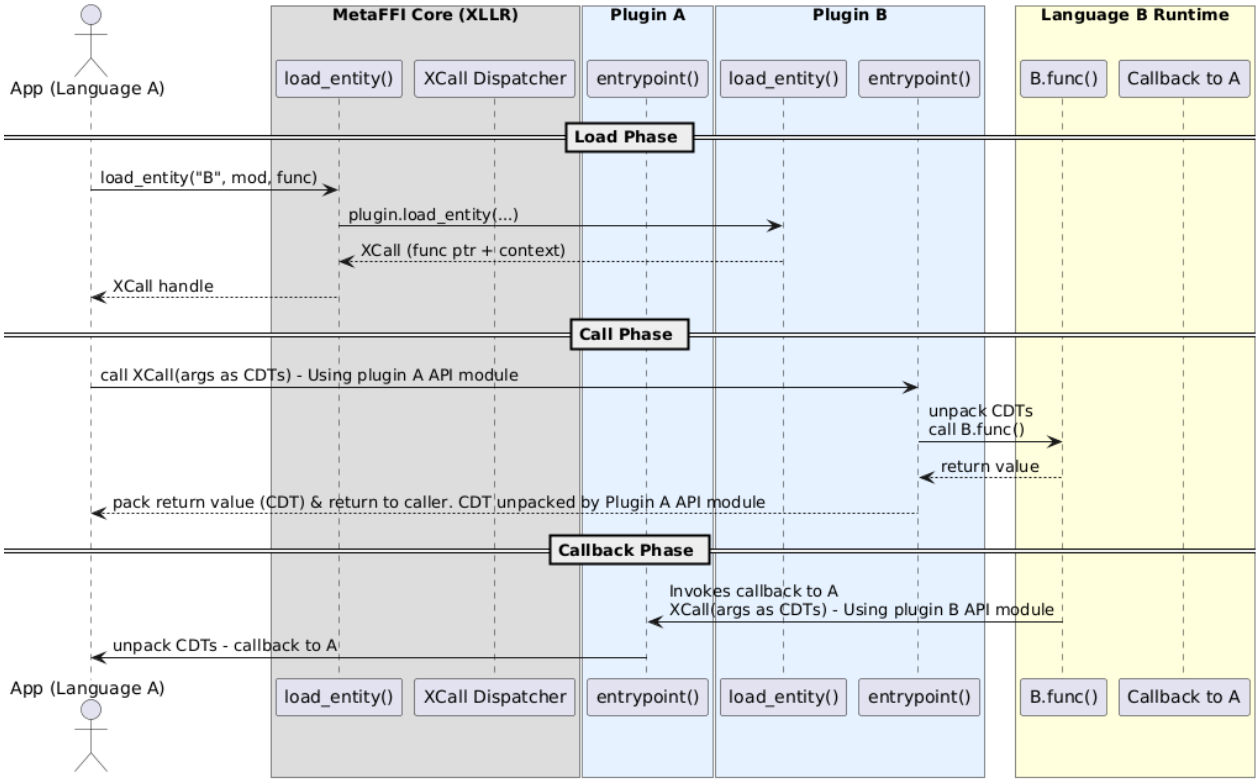


Figure 3. MetaFFI's full-duplex invocation flow. Language A calls a foreign function in Language B using XCall (Listing 13).

Listing 13. xcall struct.

```
struct xcall
{
void* pxcall_and_context[2];
};
```

The XCall entry point is a C function pointer shown in Listing 14.

Listing 14. XCall entry point.

```
void (*)(void* context, cdts* pcdts, char** out_err);
```

In case the foreign entity does not require parameters and has no return values entrypoint C function pointer type does not include the `cdts*` parameter, as shown in Listing 15.

Listing 15. XCall entry point to foreign function without parameters and without return values.

```
void (*)(void* context, char** out_err);
```

The first function pointer type is used for a foreign entity that has no parameters and no return values. In all other cases, the second type is used.

The parameter `context` is the second element returned by functions `load_entity` and `make_callable`. It stores the information that the runtime plugin needs to invoke and use the foreign entity. The functions are implemented in the runtime plugin and create the context. If the runtime plugin does not need any context, the context pointer is `null`.

In the entrypoint function, the parameter `pcdts` refers to two CDTS structures, where `pcdts[0]` contains the CDTS for the arguments and `pcdts[1]` contains the CDTS for the return values. If there are no arguments or results, the `len` field in the CDTS structure is set to 0.

The parameter `out_err` is an out parameter that stores a null-terminated error string, in the event of an error. The caller is responsible for initializing the parameter to `null`. As discussed in Section 6.2.1, the error string must be allocated using an XLLR memory allocation function. In case of an error, the caller must free the error using the XLLR free function. Therefore, the runtime plugin must allocate the error string on the heap. This can be improved in the future by adding another out Boolean parameter `is_free_err`.

6.8.1. CDTS Allocation

CDT is being reused in every MetaFFI XCall; therefore, its allocation and deallocation affect the performance of the system.

MetaFFI does not assume that the calling runtime can allocate CDT on the stack. Also, MetaFFI cannot provide a C function that allocates CDT on the stack because the CDT would be freed once the C function returns.

To overcome this, MetaFFI provides the CDT allocation and deallocation as defined in Listing 16.

Listing 16. Memory allocation functions.

```
cdts** alloc_cdts_buffer(metaffi_size params_count, metaffi_size ret_count);
void free_cdts_buffer(cdts** pcdts);
```

Instead of allocating CDTS and CDTs on the heap every call, MetaFFI preallocates 50 CDTS and 50 CDTs globally, per thread, using thread local storage [42] and an index for each type to specify the current offset of used CDTS and CDTs (Listing 17).

Listing 17. CDTS caching.

```
thread_local cdt cdt_cache[cdt_cache_size];
thread_local int cdt_current_index = 0;
thread_local cdts cdts_cache[cdts_cache_size];
thread_local int cdts_current_index = 0;
```

The function `alloc_cdts_buffer` reuses CDTS and CDTs from a global cache if the number of CDTs required for the call is within a predefined threshold; otherwise, it dynamically allocates them on the heap and updates the flag `allocated_on_stack` in the CDTS structure accordingly. The threshold is set at 50, which means that the function can handle up to 50 CDTSs and CDTs without allocating memory (i.e., `params_count+ret_count` $\leq$ 50). The function also updates `cdt_current_index` based on the number of parameters and return values, also adding two to `cdts_current_index`, since every call requires two CDTSs.

Based on the value of `allocated_on_stack`, the function `free_cdts_buffer` frees the CDTS and CDTs from the heap or decrements the fields `cdt_current_index` and `cdts_current_index`.

6.8.2. Cleanup

The `cdt_val` field of a CDT may contain data that is dynamically allocated to the heap by a runtime plugin. In such cases, the runtime plugin is required to set the `free_required` field of the CDT to TRUE (a nonzero value). This indicates that the caller needs to free the data after the call using the XLLR free function.

6.8.3. XCall and Callback Flow Diagram

Figure 3 illustrates the complete lifecycle of an `XCall` from a host application in Language A to a function defined in Language B, including the possibility of callbacks from B to A. This bidirectional flow demonstrates MetaFFI's support for full-duplex interoperability using a minimal, uniform C-based interface.

The process begins when the host loads a foreign entity in B by calling `load_entity`. XLLR delegates this request to the target language's plugin, which returns an `XCall` object, a C function pointer, and an opaque context pointer, encapsulating how to invoke the foreign entity.

When the host later calls this `XCall`, it provides arguments encoded in MetaFFI's Common Data Type System (CDTS). The plugin for Language B unpacks these CDTs into its native types and invokes the target function. If that function subsequently calls a callback originally defined in Language A, the invocation flows back through A's plugin, using the same `XCall` interface in reverse, ensuring continuity of context and consistent data marshaling.

This sequence demonstrates that MetaFFI does not rely on a shared runtime or object model to achieve rich interoperation. Instead, it decouples runtimes using CDTs and opaque handles, allowing each plugin to manage native type translation independently. This design enables not only function calls but also event handlers, lambdas, and asynchronous callbacks across language boundaries.

6.9. Dynamic Capabilities: (Has Not Yet Been Implemented in MetaFFI)

The MetaFFI runtime includes a planned optimization mechanism called *capabilities*. This mechanism allows two plugins, host and guest, to negotiate the most efficient invocation strategy at runtime based on their mutual abilities. When a foreign entity is loaded, the host advertises a bitmask of supported capabilities (e.g., zero-copy parameter passing, optimized calling conventions, pointer sharing, fast exception propagation), and the guest replies with its required capabilities. MetaFFI then selects the most efficient configuration that satisfies both parties. All calls default to a generic safe mode, but if an optimization is mutually supported (e.g., skipping deep copies for arrays), it is transparently applied. This per-entity, per-call negotiation is unique to MetaFFI. In contrast, systems like COM fix call semantics (e.g., HRESULT error codes) at the interface definition time and cannot adjust dynamically.

- Mechanism Design. MetaFFI exposes this negotiation via the `load_runtime` and `load_entity` functions. When `load_runtime` is called, a plugin returns a 64-bit bitmask representing the capabilities it supports. When `load_entity` or `make_callable` is invoked, the caller specifies the desired capabilities. If the requested capabilities are a subset of those returned earlier, MetaFFI configures the XCall accordingly. Otherwise, it falls back to the safe, general path.
- CDT Optimization via Capabilities. The capabilities mechanism can also influence how values are passed in the CDT union. Normally, CDT uses a general-purpose array of typed values, where each entry can represent a scalar, nested array, or object reference. However, in some statically typed interoperation cases, such as Go calling C++, it is possible to pass a raw pointer to a homogenous C array (e.g., `int32_t*`) using a dedicated CDT field, thereby avoiding boxing and allocation.
- Bypassing CDT Entirely. In extreme cases, the CDT layer may be bypassed altogether. For example, if calling from C++ to C and both plugins declare support for a shared low-level ABI (such as x86-64), MetaFFI can allow direct pointer invocation. The plugin advertises this capability via `load_runtime`, and the caller requests it in `load_entity`. The host then receives a direct function pointer compatible with its native calling convention, yielding maximal performance without compromising generality.
- General Case Required. Importantly, capabilities are optional performance optimizations. To ensure broad interoperability, each plugin must implement the generic case of CDT and XCall, which supports all other MetaFFI-compatible languages. Capability-based optimization is an enhancement layered on top of this foundation, not a replacement.

6.10. Compiler and Compiler Plugin

6.10.1. Guest Compiler

Many runtimes support the functionality of loading entities from their respective "executable-code" formats. For example, JVM can load entities from `.jar` or `.class` files, while CPython can load entities from `.py` or `.pyc` files. However, this functionality is not uniformly supported in all runtimes, and some runtimes, such as `Go`, require the creation of entry points to access foreign entities.

To address this challenge, MetaFFI provides a compiler that generates "executable code" for the target runtime with entry points to the foreign entities.

A *guest compiler* plugin is a dynamic library that implements the following C interface. In future versions of MetaFFI, a plugin will be loaded using MetaFFI; therefore, it could be implemented in any supported language. In case the loading fails, MetaFFI will revert to the C interface.

The `compile_to_guest` function, defined in Listing 18, is invoked by the MetaFFI command-line tool to generate a MetaFFI module for the target runtime with entry points for its entity. The function takes as input an IDL definition, which is a JSON (JavaScript Object Notation) formatted string that specifies the signatures of all foreign entities in the target code to be available via MetaFFI. The IDL and its automatic generation are explained in Section 6.11.

The function has the following parameters (strings are null-terminated):

- `const char* idl_def_json`: the IDL to compile.
- `const char* output_path`: the path where the module should be written.
- `const char* guest_options`: an optional key–value string in the format. key1=value1,...,keyN=valueN provides options for the plugin compiler. Options are passed using the MetaFFI command-line switch `--guest-options`.
- `char** out_err`: an output parameter that returns an error string in case of failure.

Listing 18. Compiler plugin entrypoint for guest generation.

```

// called once when the compiler plugin is loaded to perform any initialization tasks.
void init();

// generates executable for the target runtime based on given JSON IDL
void compile_to_guest(const char* idl_def_json,
const char* output_path,
const char* guest_options,
char** out_err);
```

For compiler plugin implementers, the error string returned by `out_err` must be allocated using the XLLR memory allocation function, as the MetaFFI tool will attempt to free it using the XLLR free function. Also, in case the runtime supports access control for entities (e.g., public, private), it is recommended to generate entry points only for public entities.

6.10.2. Host Compiler

The goal of the host compiler plugin is to automatically generate code that takes advantage of the language-specific MetaFFI API to load and wrap foreign entities. The generated code enables transparent use of the foreign entities, satisfying the *simple interoperability* [7] requirement (presented in Section 1), which states that the code accessing foreign entities should be the same as that accessing entities within the runtime.

The host compiler can be essential when using a large foreign library, as it frees the developer from writing code to load the foreign entities, which can be an exhausting task for large libraries.

A host compiler plugin, analogous to a guest compiler plugin, is a dynamic library that implements the following C functions. Similarly to a guest compiler, in the future, MetaFFI will first attempt to load host compiler as a MetaFFI module.

The `compile_from_host` function, defined in Listing 19, is invoked by the MetaFFI command-line tool to generate source code that uses the MetaFFI API to load and use foreign entities. The function takes as input an IDL definition that specifies the signatures of all entities in the target code. The IDL and its automatic generation are explained in Section 6.11.

Listing 19. Compiler plugin entrypoint for host generation.

```

// called once when the compiler plugin is loaded to perform any initialization tasks.
void init();

// generates code for the required runtime which uses MetaFFI API to load
// and call the foreign entities
void compile_from_host(const char* idl_def_json,
const char* output_path,
const char* host_options,
char** out_err);
```

The function has the following parameters:

- `const char* idl_def_json`: the IDL to compile
- `const char* output_path`: the path where the module should be written
- `const char* host_options`: similarly to guest options, an optional key–value string in the format key1 = value1, ..., keyN = valueN provides options for the plugin compiler. Options are passed using the MetaFFI command-line switch `-host-options`
- `char** out_err`: an output parameter that returns an error string in case of failure

Similarly to the guest compiler plugin, the error string returned by `out_err` must be allocated using the XLLR allocation function.

6.11. IDL Definition Model and IDL Plugin

The IDL, which defines the signature and structure of foreign entities, is a hierarchical structured JSON format string required by MetaFFI compiler plugins (Section 6.10). The IDL provides the information necessary for the guest compiler plugins to generate code that interacts with the foreign entities, and the host compiler to wrap the foreign entities. The IDL specifies the different entities and their properties. Detailed and technical information about the IDL can be found in Section 7 of [43].

As shown in Figure 4, which illustrates a simplified view of the definition model, the IDL consists of modules that correspond to the foreign modules. Each module contains four different entities:

- Globals, which are global variables and constants.
- Functions, which are global functions or global scripts. A global script can be represented as a function named `main`.
- External Resources, which are external modules and resources required by the guest compiler.
- Classes, which are complex types, such as `class` (as found in C++, Java, Python3.11, etc.), `struct` (as in C, Go, etc.), and `record` (in Emacs Lisp 26.1 [44]).
  - Fields, which are the attributes of the complex types.
  - Constructors, which used to create new instances of complex types, similar to constructors in C++ or Java. In cases where the runtime does not support constructors, the implementer may choose to represent certain functions as constructors, as long as they still return a new instance of the complex type.
  - Methods, which can also be functions in runtimes that do not support methods in case they accept an instance of the complex type as a parameter. The developer has the flexibility to define certain functions as methods, as long as they necessitate an instance of the complex type.

MetaFFI provides an implementation of the IDL entities and creates a JSON IDL.

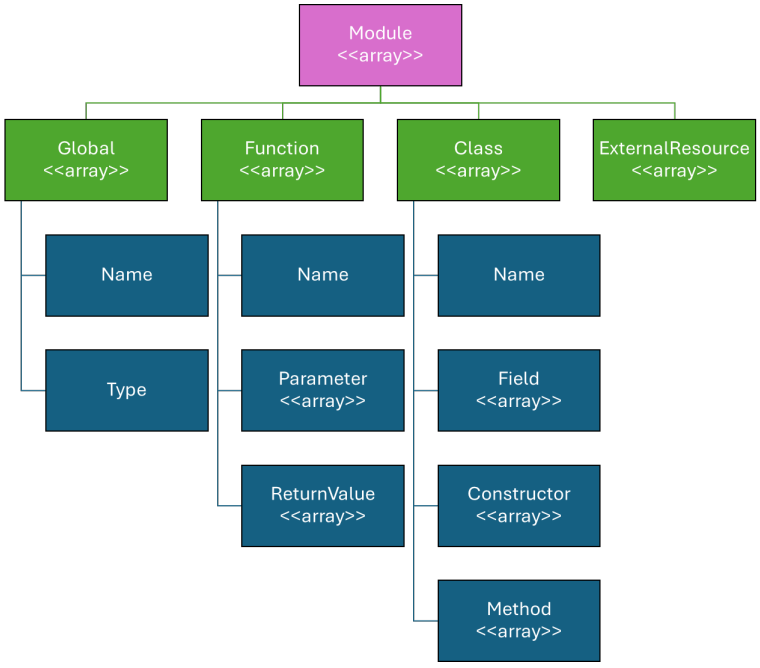


Figure 4. Simplified definition model.

IDL Plugin

Kaplan et al. [29] stated that writing IDL manually is a tedious and impractical approach with many drawbacks. As we agree with this statement, MetaFFI generates the IDL automatically and does not require any manual writing of the IDL. Instead, MetaFFI uses IDL plugins to extract IDL directly from source code or runtime execution code. Automated IDL generation also satisfies *simple interoperability* [7] as presented in Section 1.

The IDL plugin analyzes source code or runtime executable code and extracts the definitions of the foreign entities (excluding the actual logic of the foreign callable entities) and constructs the definition data using the entities of the definition model.

Like other MetaFFI plugins, the IDL plugin is a dynamic library that exports the following C functions. Similarly to the compiler plugins, the IDL plugin will attempt to load as a MetaFFI module in the future. The IDL compiler needs to implement `parse_idl`, defined in Listing 20.

Listing 20. IDL plugin entry point.

```
// called once when the idl plugin is loaded to perform any initialization tasks.
void init();

// Enum used in parse_idl to determine the type of input.
enum idl_input_type
{
source_code = 0,
path = 1
};

// IDL plugin must to implement this function
// return value is the JSON IDL
char* parse_idl(idl_input_type input_type,
const char* data, // input
char** out_err); // return error on failure, otherwise null
```

- `input_type` is the type of data passed in `data`.
- `data` contains a file or path of the source code or executable-code to parse.
- `out_err` contains an error string, in case of an error. The error string must be allocated by an XLLR allocation function.
- `return` is the generated IDL in JSON format. The IDL must be stored on the heap using the XLLR allocation function.

The IDL plugin processes either raw source code or a path to the codebase. Its purpose is to analyze the provided input and generate a JSON-based Intermediate Definition Language (IDL) describing the detected entities. This JSON IDL serves as the foundation for the host compiler plugin, which utilizes it to generate the necessary code for integrating with XLLR, enabling seamless interoperability.

6.12. MetaFFI-Supported Languages

MetaFFI focuses on programming languages, which are languages used to write the logic of the application, as defined in [7]. This differs from non-programming languages that are designed to store data, describe a web page, and so on (e.g., HTML [45], GNU Make [46], CSS [47], JSON [48], XML [49]).

MetaFFI does not limit itself to programming languages, as it also supports a broad definition of "entities", which can be found outside the programming language domain, for instance, SQL storage procedures. Even though SQL is typically not a programming language used to write the logic of an application, and is even executed outside the process that "calls" the SQL code, MetaFFI can be used to invoke storage procedures as if they were functions in the host language.

Table 5 lists several languages and the prerequisites they satisfy. A language that acts as a host can call C functions and interact with MetaFFI. For example, PowerShell can import a DLL using P/Invoke, a C-FFI mechanism supported by the .NET framework, which PowerShell relies on. Similarly, Bash can call C functions using the `gawk` command. These C-FFI mechanisms enable any programming language that can interface with C and load XLLR to utilize MetaFFI.

Table 5. Partial list of MetaFFI-supported languages.

Languages	Act as Host	Act as Guest
Go	✓	✓
Python3.11	✓	✓
OpenJDK	✓	✓
.Net Framework	✓	✓
C	✓	✓
C++	✓	✓
LLVM-IR	✓	✓
JavaScript NodeJS	✓	✓
Bash	✓	✓
Powershell	✓	✓
JavaScript V8 Engine	✓	✓
SQL		✓
In Browser JavaScript		
HTML		

A language that acts as a guest can be loaded and embedded. Notably, the MetaFFI runtime plugin is not restricted to embedding a language directly from C. It can also use an intermediate language to load the guest. As the table indicates, HTML is not supported because it is not a programming language and does not meet the criteria for programming language interoperability. Although in-browser JavaScript can interface with C through JavaScript-based C interpreters, it cannot load MetaFFI due to browser security restrictions.

Detailed and technical information about adding language support to MetaFFI can be found in Section 6 of [43].

7. User Study and Survey

We conducted a survey to evaluate the usability of MetaFFI. Usability is a critical aspect of interoperability frameworks because, while it is theoretically possible to bind multiple programming languages using a direct $O(N^2)$ approach, by manually implementing and maintaining all possible FFI and embedding mechanisms, the practical complexity is overwhelming. Interoperability frameworks exist to reduce this immense effort, making cross-language interaction more efficient and maintainable.

For instance, TruffleVM provides seamless interoperability between languages, but it achieves this by limiting the supported languages and runtime environments. This trade-off demonstrates the inherent challenge in interoperability design, balancing broad language support with ease of use. One of the primary goals of MetaFFI is to enable seamless cross-language function calls across a large number of programming languages while maintaining a high level of usability.

An interoperability system that is difficult to use can impose a steep learning curve, require extensive manual configuration, and ultimately fail to achieve its goal of reducing development effort. Therefore, usability evaluation is essential to ensure that MetaFFI effectively abstracts away the complexities of language interoperability while remaining

accessible to developers. By conducting this study, we assess whether MetaFFI's design successfully simplifies cross-language integration without sacrificing flexibility.

The survey involved 17 students participating in a distributed systems workshop. The students were divided into five groups, with up to four students per group. Each group received a Python function designed to crawl a web page and return its findings, and a Java object implementing a Chord distributed hash table. The primary programming language used in the workshop was Go, with which the students had no previous experience. The students could choose not to participate in the anonymous survey. In addition, the respondents were aware that their responses did not influence their grade.

Initially, the students were given a task that required interoperability to call the Python function from Go (using CPython) and to use the Chord object from Go (using JNI). To facilitate this, the students received a detailed step-by-step tutorial on how to perform the interoperability using CPython and JNI, with C serving as the intermediate language. They utilized CGo to access C from Go, and CPython and JNI to access the Python function and Java object. The Python code was provided as a .py file, and the Java code was provided as a directory containing .class files alongside the java source code.

In a subsequent part of the workshop, the students were asked to use the same Python function and Java object, but this time employing the MetaFFI system. They were provided with a tutorial offering step-by-step explanations for some of the entities, and were asked to complete the rest of the tasks independently. They were allowed to add more functions and methods in Java or Python as they saw fit.

The overall process of the experiment, including the steps taken by the students in both the traditional and MetaFFI-based approaches, is illustrated in Figure 5. This figure visually represents the transition from manual interoperability using CPython and JNI to the streamlined process enabled by MetaFFI.

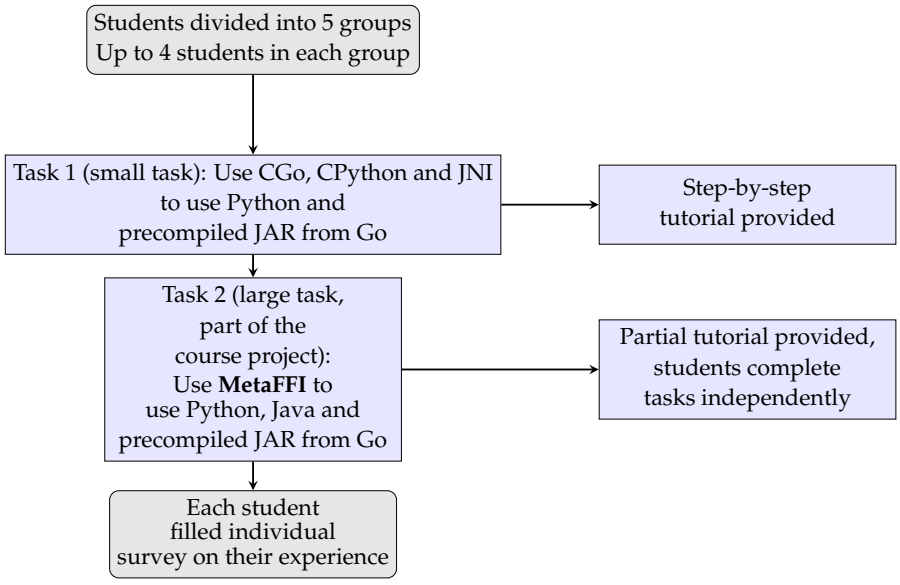


Figure 5. Steps of the experiment conducted in the workshop.

At the conclusion of the workshop, we surveyed the students about their experience using MetaFFI compared to the traditional methods without MetaFFI. Participants were instructed to rate their experience on a Likert scale, and optionally add free-text comments. The survey questionnaire is presented in Appendix A.

While the sample size (17 students) may seem limited, we emphasize that this was an exploratory study designed to assess early usability patterns in a controlled setting. Small-scale user studies are common in early-stage systems research [9], and the results

obtained were sufficient to validate the key usability goals and guide further development. Future studies will include a broader and more diverse developer population.

The survey aimed to capture the students' perspectives on the efficiency, ease of use, and overall experience with MetaFFI in contrast to the conventional approaches involving CPython and JNI.

The students used the MetaFFI system with plugins that support Go, Python3 (CPython), and Java (OpenJDK) in Ubuntu 20.04.

7.1. Results and Analysis

Effort Invested (in hours):

The data presented in Table 6 provides a detailed breakdown of the effort invested (in hours) by students in using MetaFFI for interoperability between Go and Java, and Go and Python. The results clearly indicate that the use of MetaFFI significantly reduces the time required for these tasks. On average, students spent 9.68 h on Go to Java interoperability without MetaFFI, compared to just 2.73 h with MetaFFI. Similarly, the average effort for Go to Python interoperability was 7.29 h without MetaFFI and only 2.26 h with MetaFFI.

Table 6. Effort Invested (in hours).

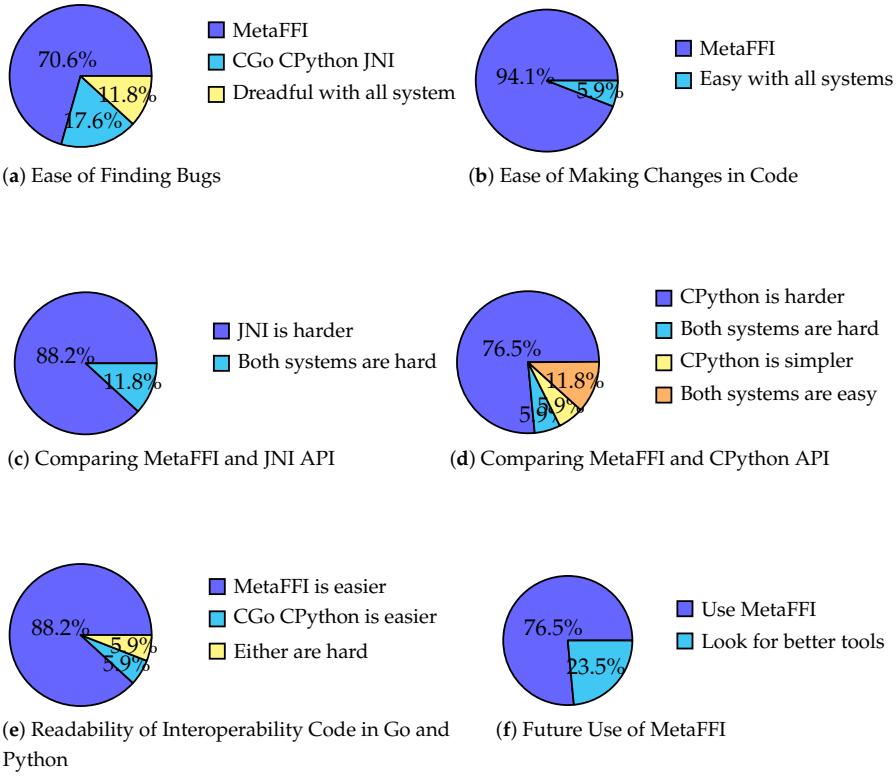
Student	Go to Java Without MetaFFI	Go to Java with MetaFFI	Go to Python Without MetaFFI	Go to Python with MetaFFI
1	1.5	0.5	1.5	0.5
2	10	0.5	4	0.5
3	5	1	5	1
4	15	7	15	7
5	3	0.5	2	0.1
6	1.5	0.3	1	0.4
7	19	0.1	12	0.1
8	3	0.5	3	0.5
9	20	10	15	10
10	8	2	8	2
11	24	2	12	0.5
12	24	2	11	0.5
13	3.5	4.5	3.5	4.5
14	2	1.5	1	0.83
15	7	2	6	2
16	8	2	16	1
17	10	10	8	7
Average	**9.68**	**2.73**	**7.29**	**2.26**
Median	**8**	**2**	**6**	**0.83**
Standard deviation	**7.85**	**3.23**	**5.27**	**2.98**

The median values further support these findings, with the median effort for Go to Java interoperability being 8 h without MetaFFI and 2 h with MetaFFI, and for Go to Python, it was 6 h without MetaFFI and 0.83 h with MetaFFI.

Furthermore, the standard deviation values reveal the variability in students' performance. For Go to Java without MetaFFI, the standard deviation was 7.85, indicating a high degree of variability in effort, likely due to differing levels of expertise or familiarity with the existing tools. With MetaFFI, this variability decreased significantly, with a standard deviation of 3.23, showing a more consistent effort among students. A similar trend was observed for Go to Python interoperability, where the standard deviation dropped from 5.27 without MetaFFI to 2.98 with MetaFFI. These results highlight not only the efficiency of MetaFFI in reducing the time required but also its ability to standardize the effort across users, making the process more predictable and user-friendly.

Survey Results:

The results reveal significant insights into the practical application and user experience of the MetaFFI interoperability system. The majority of participants found MetaFFI to be notably more efficient in development time and user-friendliness compared to traditional methods like CGo, CPython, and JNI. Specifically, 70.6% of respondents indicated that finding bugs was easier with MetaFFI, while only 17.6% preferred other methods, as shown in Figure 6a. When it came to making changes in the code, a compelling 94.1% found MetaFFI easier to use (Figure 6b). Furthermore, 88.2% of the participants reported that JNI is harder to use than MetaFFI, and 76.5% found CPython to be more challenging (Figures 6c and 6d, respectively). Additionally, 88.2% of respondents believed that the code readability improved with MetaFFI (Figure 6e). For future use, 76.5% of participants expressed a preference for MetaFFI, with only 23.5% inclined to look for better tools (Figure 6f). These findings underscore MetaFFI's potential to perform interoperability tasks, reduce development effort, and enhance overall code maintainability.



(**a**) Ease of Finding Bugs

(**b**) Ease of Making Changes in Code

(**c**) Comparing MetaFFI and JNI API

(**d**) Comparing MetaFFI and CPython API

(**e**) Readability of Interoperability Code in Go and Python

(**f**) Future Use of MetaFFI

Figure 6. Survey results.

Interoperability effort compared to overall workshop:

The survey also included questions regarding the effort, in percentage, that students estimated they spent on coding with and without MetaFFI during the workshop. The specific questions were the following: *If the whole workshop is 100%, how much effort, in percentage,*

do you estimate you have spent on coding with MetaFFI to interoperate Go with Python and Java? If the whole workshop is 100%, how much effort, in percentage, do you estimate you have spent on coding without MetaFFI (i.e., CGO/CPython/JNI) to interoperate Go with Python and Java?

The responses to the first question indicated that the average effort spent coding with MetaFFI was 14%, with a median of 10%. In contrast, the responses to the second question revealed that the average effort spent coding without MetaFFI was significantly higher, at 32%, with a median of 35%. These results suggest that students found coding with MetaFFI to be more efficient and less time-consuming than using traditional methods for interoperability.

The substantial reduction in effort required when using MetaFFI underscores its effectiveness in simplifying the process of integrating different programming languages. By reducing the time and effort needed for programming language interoperability, MetaFFI allows developers to focus more on other critical aspects of their projects. This efficiency gain highlights MetaFFI's potential to enhance productivity and streamline development workflows in multi-PL environments.

7.2. Discussion of Results

The survey conducted during the distributed systems workshop provides valuable insights into the practical application of MetaFFI for interoperability of the programming language. The results indicate that MetaFFI significantly reduces the effort required for interoperability tasks, improves code readability, and is generally preferred by students over traditional methods involving CGo, JNI, and CPython.

The qualitative feedback from students highlighted several key themes. The most common response was that MetaFFI significantly simplified interoperability compared to JNI and CPython. Many students noted that MetaFFI allowed them to focus on application logic rather than low-level interoperability challenges, which aligns with its design goal of abstracting language interoperability. One participant stated, "Working with MetaFFI was a breeze and made me only focus on issues/bugs in the implementation of the Java/Python code." Another noted that "MetaFFI was much easier to use, especially when it replaced JNI."

However, some students reported challenges in debugging their code, as they found it difficult to determine whether errors originated from the foreign code or from MetaFFI itself. One student stated, "When we had bugs, it was hard to identify what the bug actually is because we didn't understand how MetaFFI works." This suggests that while MetaFFI simplifies interoperability, additional documentation and improved error reporting mechanisms to better distinguish between interoperability-related issues and errors from the foreign code could enhance debugging visibility, as one of the teams did encounter a bug in MetaFFI.

Several participants emphasized the need for better documentation, particularly for onboarding and troubleshooting. One student suggested, "The only thing I think was missing for me is a short readme file or some documentation about the module (in GitHub for example). I think it would've made it more user-friendly and helped me find my mistakes."

Interestingly, some participants expressed their satisfaction with MetaFFI in a lighthearted manner, emphasizing the significant amount of time it saved them compared to traditional interoperability methods. While these comments were exaggerated for effect, they reinforce the perception that MetaFFI substantially reduces development time, allowing developers to focus more on application logic rather than low-level interoperability concerns.

Overall, the feedback suggests that MetaFFI is well-received in terms of usability and efficiency but could benefit from improved debugging tools and documentation.

7.3. Threats to Validity

This section discusses potential threats to the validity of the study and the steps taken to address them. We distinguish between external validity (generalizability) and internal validity (causal correctness).

External validity may be affected by the relatively small number of participants (17 students) and the fact that all of them were from the same academic course. However, despite the limited sample size, the results were consistent across participants. For example, the reported effort savings and the strong preference for MetaFFI were reflected both in the mean values and in the standard deviation. This indicates that even participants with weaker technical backgrounds were able to benefit from MetaFFI, as the survey was answered individually and not at the group level. Another external threat is the nature of the tasks, which were performed in an academic setting as part of a workshop. While this environment is not fully representative of industrial software projects, the tasks involved real challenges, such as using CPython, JNI, and integrating precompiled components. These are relevant to real-world interoperability work. A third external threat is the presence of instructional materials. Students received a full tutorial for the traditional methods (CGo, CPython, JNI) and only a partial tutorial for MetaFFI. This might have skewed the results in favor of the traditional approach, but the opposite trend was observed. This strengthens the validity of the usability findings in favor of MetaFFI.

Internal validity could be influenced by selection bias, maturation effects, and instrumentation differences. Students were not randomly assigned to groups but chose their own teams, which may have created an imbalance in experience across groups. However, the survey was answered individually, so group dynamics had no direct impact on the results. Moreover, the consistency of positive feedback, including from less experienced groups, indicates that MetaFFI was usable regardless of the participant's skill level. Maturation effects may also have influenced the outcome, since the MetaFFI tasks were performed after the traditional methods. However, the reduction in reported effort was substantial, even though the MetaFFI tasks required additional functionality. This makes it unlikely that learning alone accounts for the difference. Finally, instrumentation is a potential threat due to the tutorial disparity mentioned above. The traditional systems were fully documented, while MetaFFI had only partial guidance. This might have made the traditional approach easier to use, yet the results showed the opposite. Therefore, the instrumentation difference, if anything, worked against MetaFFI, making the results more robust.

To summarize, the study is subject to several standard threats to validity. However, the measures taken during the experiment, including individual response collection, use of realistic integration tasks, and consistent trends in both quantitative and qualitative results suggest that the conclusions are well supported. The limited documentation provided for MetaFFI and the increased task complexity in the MetaFFI phase imply that the observed usability benefits are not only valid but may be understated.

8. Conclusions

MetaFFI is the first mechanism to provide simple interoperability, as introduced in the Introduction and elaborated in Section 4 (see also [7] for the supporting empirical study). It achieves this by combining FFI and embedding interoperability approaches to support a wide range of programming languages while leveraging a central hub to reduce the number of required binding mechanisms from $O(n^2)$ to $O(n)$. Additionally, MetaFFI employs CDTS to enable deep binding integration and utilizes automatically generated IDLs to eliminate manual effort. This design allows MetaFFI to support both dynamic interoperability, by loading foreign modules at runtime, and compile-time interoperability, enabling seamless and transparent language integration.

Our user study, conducted through a distributed systems workshop, highlighted the significant advantages of MetaFFI over traditional interoperability methods such as CGo, JNI, and CPython. Participants reported a substantial reduction in development time and effort, along with improved code readability and maintainability. The majority found MetaFFI more efficient and user-friendly, reinforcing its potential to streamline development workflows and enhance productivity. However, feedback also emphasized the need for more comprehensive documentation and improved error reporting to better distinguish between issues arising from interoperability and those originating in the foreign language. While MetaFFI has proven to be a powerful tool for interoperability, addressing these aspects will further enhance its effectiveness and accessibility.

Future work will focus on six main areas. First, we plan to improve the documentation, incorporating insights and suggestions from the user study. Second is the performance optimization of implementation and comprehensive performance benchmarks. Third, we aim to refine the plugin framework to simplify the process of adding support for new programming languages. Fourth, we will analyze and extend the calling conventions supported by XCall in order to improve performance across language boundaries. Fifth, we intend to address the debugging challenges inherent in multilingual development environments. Sixth, we will continue expanding the range of supported languages. Finally, we plan to broaden and deepen usability studies to further enhance the system's ease of use.

The features of MetaFFI enable users to work with other languages within the familiar environment of the host language, without the need to switch contexts. The indirect method of achieving interoperability through the management of various runtime environments and CDTs, meeting the requirements of *simple interoperability* [7], demonstrates a notable enhancement in the quantity of necessary interoperability mechanisms and offers a straightforward and user-friendly approach for multilingual programming.

- Performance Considerations. The primary focus of this paper is to demonstrate the feasibility and correctness of MetaFFI's architecture and core abstractions. As this is an early-stage system design, we prioritized cross-language fidelity, semantic compatibility, and ease of use over runtime performance. Preliminary benchmarks were omitted intentionally, as optimization strategies and language-specific tuning are planned as future work. However, the architecture of MetaFFI is designed to minimize overhead by relying on native function pointers and in-process execution, making it well-suited for high-performance use cases in future iterations.
- Practical Implications. MetaFFI can serve as a foundational interoperability layer in software development organizations working across heterogeneous codebases, such as integrating Python third parties for LLM into Go microservices, expanding codebases outside the language barrier, or implement flashy GUI using C# or Node.JS in a C program. Developers can adopt MetaFFI incrementally, without rearchitecting their systems, while gaining seamless polymorphic invocation, callback support, and tooling simplification.

Author Contributions: Conceptualization, T.C.-S.; methodology, T.C.-S. and A.Y.; software, T.C.-S.; validation, T.C.-S. and A.Y.; investigation, T.C.-S. and A.Y.; data curation, T.C.-S.; survey design, T.C.-S. and A.Y.; writing, T.C.-S., A.Y.; supervision, A.Y.; funding acquisition, A.Y.; administration, A.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Blavatnik Family Foundation grant number 2953.

Data Availability Statement: The data and source code presented in this study are openly available in https://github.com/MetaFFI.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Survey Questionnaire

The students were asked the following questions:

1. How much effort overall (in hours) did you invest in calling from Go to Java without MetaFFI?
2. How much effort overall (in hours) did you invest in calling from Go to Java with MetaFFI?
3. How much effort overall (in hours) did you invest in calling from Go to Python without MetaFFI?
4. How much effort overall (in hours) did you invest in calling from Go to Python with MetaFFI?
5. To find your bugs in my code when I used the interoperability system, it was:
   - Easier with CGo/CPython or CGo/JNI.
   - Easier with MetaFFI.
   - Easy with all systems.
   - Dreadful with all system.
6. To make changes in the code to add/modify the functionality with the interoperability system it was:
   - Easier with MetaFFI.
   - Easier with CGo/CPython or CGo/JNI.
   - Easy with all systems.
   - Dreadful with all system.
7. Comparing MetaFFI and JNI API:
   - JNI is simpler to use than MetaFFI.
   - JNI is harder to use than MetaFFI.
   - Both systems are easy to use.
   - Both system are hard to use.
8. Comparing MetaFFI and CPython API:
   - CPython is simpler to use than MetaFFI.
   - CPython is harder to use than MetaFFI.
   - Both systems are easy to use.
   - Both system are hard to use.
9. My interoperability code in Go and Python is easier to read and understand with the interoperability system:
   - CGo/CPython is easier to read and understand.
   - MetaFFI the code is easier to read and understand.
   - Either of the codes are easy to understand.
   - Either of the codes are hard to understand.
10. In a future scenario, where you'll need to interoperate programming languages, would you:
    - Use MetaFFI.
    - I'd look for better interoperability tools, and if I can't find anything better, I'll use MetaFFI.
    - I would look for alternatives. I didn't like MetaFFI.
11. If the whole workshop is 100%, how much effort, in percentage, do you estimate you have spent on coding with MetaFFI to interoperate Go with Python and Java?

12. If the whole workshop is 100%, how much effort, in percentage, do you estimate you have spent on coding without MetaFFI (i.e., CGO/CPython/JNI) to interoperate Go with Python and Java?
13. Do you have any feedback for the MetaFFI system (Positive/Negative feedback, recommendations, etc.)?

Appendix B. System Test Coverage

To validate cross-language correctness, we implemented system tests across all supported host–guest combinations: Python↔Java, Python↔Go and Java↔Go. The tests covered the following:

- Data types: All CDTS-supported types, including primitives, arrays (up to 3D), dictionaries/maps, strings, and nullables.
- Function calls: Positional arguments, named arguments (Python), multiple return values, and method binding.
- Lifecycle: Entity allocation, explicit release, reuse, and lifetime scope checks.
- Callbacks: Verified between Java↔Go and Java↔Python. Python↔Go pending.
- Third-party libraries: We tested complex real-world packages:
  - From Python: BeautifulSoup, NumPy, Pandas.
  - From Java: Log4j, DateTimeFormat.
  - From Go: GoMemCache and other I/O libraries.

Table A1 presents the validated CDTS mappings, in all directions, using the system tests.

Table A1. Validated CDTS type mappings across languages.

CDTS Type	Python 3.x	Java	Go
Booleans	`bool`	`boolean`	`bool`
Signed Integers (8-64 bit)	`int`	`byte, short, int, long`	`int8-int64`
Unsigned Integers	N/A	`char`	`uint8-uint64`
Float32/Float64	`float`	`float, double`	`float32, float64`
Characters (UTF-8/16/32)	`str` (1-char)	`char`	`rune`
Strings (UTF-8/16/32)	`str`	`String`	`string`
Byte Arrays (1D-3D)	`bytes, list`	`byte[], byte[][][]`	`[][][]byte`
Dictionaries	`dict`	`Map<K,V>`	`map[K]V`
Objects (Handles)	Custom Classes	Reference Types	Struct Pointers/Interfaces
Callbacks	`callable`	Functional Interface	`func(...)`
Multiple Return Values	Tuple unpacking	Return Object/Tuple	Native support

The following list presents the validated callable features as guest languages from different programming languages:

- Named Parameters
- Keyword Arguments (kwargs)
- Overloaded Function Resolution
- Passing Inherited Object Instances
- Multiple Return Values
- Cross-Language Callbacks

References

1. Yang, H.; Nong, Y.; Wang, S.; Cai, H. Multi-language software development: Issues, challenges, and solutions. *IEEE Trans. Softw. Eng.* **2024**, *50*, 512–533. [CrossRef]
2. Mayer, P.; Kirsch, M.; Le, M.A. On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers. *J. Softw. Eng. Res. Dev.* **2017**, *5*, 1. [CrossRef]
3. Bissyandé, T.F.; Thung, F.; Lo, D.; Jiang, L.; Réveillère, L. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, Kyoto, Japan, 22–26 July 2013; pp. 303–312. [CrossRef]
4. Yang, H.; Lian, W.; Wang, S.; Cai, H. Demystifying issues, challenges, and solutions for multilingual software development. In Proceedings of the 2023 IEEE/ACM 45th international conference on software engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 1840–1852. [CrossRef]
5. Grimmer, M.; Schatz, R.; Seaton, C.; Würthinger, T.; Luján, M.; Mössenböck, H. Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.* **2018**, *40*, 8. [CrossRef]
6. Chisnall, D. The challenge of cross-language interoperability. *Queue* **2013**, *11*, 20–28. [CrossRef]
7. Cherny-Shahar, T.; Yehudai, A. Multi-lingual development & programming languages interoperability: An empirical study. *arXiv* **2024**, arXiv:2411.08388. [CrossRef]
8. Li, W.; Li, L.; Cai, H. On the vulnerability proneness of multilingual code. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 14–16 November 2022; Esec/fse 2022, pp. 847–859. [CrossRef]
9. Li, W.; Marino, A.; Yang, H.; Meng, N.; Li, L.; Cai, H. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 46. [CrossRef]
10. Abidi, M.; Rahman, M.S.; Openja, M.; Khomh, F. Are multi-language design smells fault-prone? An empirical study. *ACM Trans. Softw. Eng. Methodol.* **2021**, *30*, 56. [CrossRef]
11. Grichi, M.; Abidi, M.; Jaafar, F.; Eghan, E.E.; Adams, B. On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (JNI). *IEEE Trans. Reliab.* **2021**, *70*, 428–440. [CrossRef]
12. Rasmussen, C.E.; Sottile, M.J.; Shende, S.S.; Malony, A.D. Bridging the language gap in scientific computing: The Chasm approach. *Concurr. Comput. Pract. Exp.* **2006**, *18*, 151–162. [CrossRef]
13. Lindholm, T.; Yellin, F.; Bracha, G.; Buckley, A. The Java Virtual Machine Specification. 2013. Available online: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (accessed on 22 August 2025).
14. gewarren. Common Language Runtime (CLR) Overview—.NET Framework. 2019. Available online: https://learn.microsoft.com/en-us/dotnet/standard/clr (accessed on 22 August 2025).
15. Henning, M. The Rise and Fall of CORBA: There's a lot we can learn from CORBA's mistakes. *Queue* **2006**, *4*, 28–34. [CrossRef]
16. mcleanbyron. Component Object Model (COM)—Win32 Apps. 2018. Available online: https://learn.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal (accessed on 22 August 2025).
17. Beazley, D.M. SWIG: An easy to use tool for integrating scripting languages with C and C++. In Proceedings of the Tcl/Tk Workshop, Monterey, CA, USA, 10–13 July 1996; Volume 43.
18. Ospina, G.A.; Le Charlier, B. Towards precise descriptions for programming language interoperability: A general approach based on operational semantics. In *Proceedings of the Enterprise Interoperability II*; Gonçalves, R.J., Müller, J.P., Mertins, K., Zelm, M., Eds.; Elsevier: London, UK, 2007; pp. 581–586.
19. Chiba, S. Foreign language interfaces by code migration. In Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2019), New York, NY, USA, 21–22 October 2019; pp. 1–13. [CrossRef]
20. Malone, T. Interoperability in Programming Languages. *Sch. Horizons Univ. Minn. Morris Undergrad. J.* **2014**, *1*, 3. [CrossRef]
21. Reppy, J.; Song, C. Application-specific foreign-interface generation. In Proceedings of the 5th International Conference on Generative Programming and Component Engineering (Gpce '06), New York, NY, USA, 22– 26 October 2006; pp. 49–58. [CrossRef]
22. Group, O.M. About the Common Object Request Broker Architecture Specification Version 3.4 Beta. 2020. Available online: https://www.omg.org/spec/CORBA/3.4/Beta1/About-CORBA (accessed on 22 August 2025).
23. Oracle. JNI APIs and Developer Guides. 2020. Available online: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html (accessed on 22 August 2025).
24. Green, A. LibFFI. 2019. Available online: https://sourceware.org/libffi/ (accessed on 22 August 2025).
25. Lindholm, T.; Yellin, F.; Bracha, G.; Buckley, A. Chapter 6. The Java Virtual Machine Instruction Set. 2021. Available online: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html (accessed on 22 August 2025).
26. Common Language Infrastructure (CLI). 2012. Available online: https://ecma-international.org/publications-and-standards/standards/ecma-335/ (accessed on 22 August 2025).

27. Wilmet, S. The GLib/GTK+ Development Platform. 2019. Available online: https://www.gnome.org/ (accessed on 22 August 2025).
28. Microsoft. VARIANT (oaidl.h)—Win32 Apps | Microsoft Docs. 2018. Available online: https://learn.microsoft.com/en-us/windows/win32/api/oaidl/ns-oaidl-variant (accessed on 22 August 2025).
29. Kaplan, A.; Ridgway, J.; Wileden, J.C. Why IDLs are not ideal. In Proceedings of the 9th International Workshop on Software Specification and Design (IWSSD '98), Washington, DC, USA, 16–18 April 1998; p. 2.
30. Microsoft. The Interface Definition Language (IDL) File—Win32 Apps | Microsoft Docs. 2018. Available online: https://learn.microsoft.com/en-us/windows/win32/rpc/the-interface-definition-language-idl-file (accessed on 22 August 2025).
31. Foundation, H. Haxe—The Cross-Platform Toolkit. 2021. Available online: https://haxe.org/ (accessed on 22 August 2025).
32. Nelson, B.J. Remote Procedure Call. 1981. Available online: https://cs.nyu.edu/~apanda/classes/fa22/papers/nelson81remote.pdf (accessed on 22 August 2025).
33. Foundation, P.S. Ctypes—A Foreign Function Library for Python—Python 3.8.3 Documentation. 2020. Available online: https://docs.python.org/3/library/ctypes.html (accessed on 22 August 2025).
34. Foundation, P.S. Python/Cpython: The Python Programming Language. 2021. Available online: https://www.python.org/ (accessed on 22 August 2025).
35. Turcotte, A.; Arteca, E.; Richards, G. Reasoning about foreign function interfaces without modelling the foreign language (artifact). *Dagstuhl Artifacts Ser.* **2019**, *5*, 9:1–9:2. [CrossRef]
36. Wegiel, M.; Krintz, C. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. *SIGPLAN Not.* **2010**, *45*, 223–240. [CrossRef]
37. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis amp; transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, 20–24 March 2004; pp. 75–86. [CrossRef]
38. Cowell, J. Object linking and embedding (OLE). In *Essential Visual Basic 4.0 Fast: How to Develop Applications in Visual Basic*; Springer: London, UK, 1996; pp. 136–144. [CrossRef]
39. Foundation, P.S. Design of CPython's Garbage Collector—Python Developer's Guide. Available online: https://devguide.python.org/internals/garbage-collector/index.html (accessed on 22 August 2025).
40. Microsystems, S. Memory Management in the Java HotSpot Virtual Machine. Available online: https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf (accessed on 22 August 2025).
41. Gangemi, S. An Overview of Memory Management in Go. 2021. Available online: https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8 (accessed on 22 August 2025).
42. Sade, Y.; Sagiv, M.; Shaham, R. Optimizing C multithreaded memory management using thread-local storage. *Lect. Notes Comput. Sci.* **2005**, *3443*, 137–155. [CrossRef] [PubMed]
43. Cherny-Shahar, T.; Yehudai, A. MetaFFI—Multilingual indirect interoperability system. *arXiv* **2024**, arXiv:2408.14175.
44. Petersen, M. What's New in Emacs 26.1. Available online: https://www.masteringemacs.org/article/whats-new-in-emacs-26-1 (accessed on 22 August 2025).
45. Berners-Lee, T.; Connolly, D.W. Hypertext Markup Language—2.0. Number: 1866 Series: Request for Comments tex.howpublished: RFC 1866 tex.pagetotal: 77. 1995. Available online: https://www.rfc-editor.org/info/rfc1866 (accessed on 22 August 2025).
46. Stallman, R.; McGrath, R.; Smith, P.D. *GNU Make: A Program for Directing Recompliation; GNU Make Version 3.81*; Free Software Foundation: Boston, MA, USA, 2004.
47. Flanagan, H. Cascading Style Sheets (CSS) Requirements for RFCs. Number: 7993 Series: Request for Comments tex.howpublished: RFC 7993 tex.pagetotal: 14. 2016. Available online: https://www.rfc-editor.org/info/rfc7993 (accessed on 22 August 2025).
48. Bray, T. The JavaScript Object Notation (JSON) Data Interchange Format. Number: 8259 Series: Request for Comments tex.howpublished: RFC 8259 tex.pagetotal: 16. 2017. Available online: https://www.rfc-editor.org/info/rfc8259 (accessed on 22 August 2025).
49. Rose, D.M.T.; Hollenbeck, S.; Masinter, L.M. Guidelines for the Use of Extensible Markup Language (XML) Within IETF Protocols. Number: 3470 Series: Request for Comments tex.howpublished: RFC 3470 tex.pagetotal: 28. 2003. Available online: https://www.rfc-editor.org/info/rfc3470 (accessed on 22 August 2025).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.