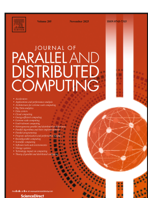
Journal of Parallel and Distributed Computing 213 (2026) 105263

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



Cross-platform integration and extension of replicated data libraries in distributed systems

Provakar Mondal *, Eli Tilevich

Software Innovations Lab, Virginia Tech, 220 Gilbert Place, Blacksburg, 24061, Virginia, USA

ARTICLE INFO

Keywords:
Distributed software architecture
Replicated data systems
Cross-platform systems
Extensible software libraries
Software quality and maintainability

ABSTRACT

Modern distributed systems replicate data across multiple execution sites to achieve high availability, low latency, and resilience against a single point of failure. These systems span heterogeneous platforms, ranging from native binaries to managed runtimes, thereby complicating the integration and extension of replicated data libraries (RDLs) across diverse execution environments. Contemporary RDLs typically presume a single dominant platform or expose limited interoperability through foreign function interfaces (FFIs), making cross-platform integration cumbersome. Moreover, extending an RDL with new capabilities often requires deep integration with a specific runtime or toolchain, hindering its reuse across diverse platforms. To address these challenges, we present BabelRDL, a software architecture that simplifies cross-platform interoperability and plug-in extensibility for replicated data management. BabelRDL establishes a common data format (CDF) that enables seamless interoperability among components running in compiled, interpreted, and managed execution environments. Its extensible plug-in model allows developers to introduce new functionality within a single language while preserving compatibility across others, reducing integration overhead and architectural fragmentation. We compare BabelRDL with FFI-based RDL integration, measuring performance and software quality characteristics. BabelRDL delivers up to 4.6× lower latency and 3.2× lower memory usage than FFI-based approaches, while improving throughput and balancing integration effort and long-term maintainability. Although requiring more upfront integration effort, BabelRDL yields simpler, more maintainable integration logic. As modern distributed systems increasingly depend on seamless interaction between diverse platforms, our work provides new insights into building maintainable, high-quality software architectures for cross-platform replicated data systems.

1. Introduction

To improve data accessibility, reduce access latency, and prevent a single point of failure, distributed systems often replicate data across multiple execution sites (*replicas*) [1]. As replicas modify their local state, updates must be synchronized across replicas to maintain consistency, which is achieved via a consistency model [2]. Modern distributed systems commonly deploy replicated data libraries (RDLs) to manage all interactions with and synchronization of data. RDL serves as a foundational software component for data replication, providing API methods for accessing and updating replicated data and for synchronizing updates behind the scenes [3].

Evolving business needs and heterogeneous infrastructure often lead to distributed components running across heterogeneous execution environments [4]. While the need to replicate data in modern distributed applications has led to the creation of various RDLs, most of which are tied to a specific execution platform or runtime [5]. If written in the same language, the application code and an RDL can interact straightforwardly. When application code at different replicas needs to be executed on diverse platforms, developers integrate an RDL via foreign function interfaces (FFIs) [6,7]. Some RDLs provide special-purpose FFIs in the form of bindings for interoperability [8] across particular runtime pairs (e.g., Rust↔JavaScript [9]). While FFIs can enable the integration, they often come with convoluted build procedures, dependency management, and debugging challenges [10,11]. FFI-based integration becomes even more unwieldy when RDLs need to offer additional functionalities, such as persistence and error handling [12,13].

To address these challenges, this paper presents BabelRDL[1], a cross-platform software architecture designed to simplify interoperability and coordination in replicated data systems. BabelRDL enables replicas run-

[1] Our system title is inspired by the biblical story of the Tower of Babel, which explains why people of the world speak different languages [14].

* Corresponding author.
E-mail address: provakar@cs.vt.edu (P. Mondal).

https://doi.org/10.1016/j.jpdc.2026.105263
Received 21 November 2025; Received in revised form 27 January 2026; Accepted 15 March 2026
Available online 18 March 2026
0743-7315/© 2026 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

ning on compiled, interpreted, and managed execution environments to coordinate updates by exchanging synchronization messages in a common data format (CDF). Furthermore, BABELRDL provides a plug-in architecture that supports systematic extension with new features. Developers can implement an RDL plug-in on any platform, and the resulting feature will be seamlessly integrated with replicas across heterogeneous execution environments. To that end, BABELRDL takes declarative metadata as input and generates the required cross-platform coordination functionality. In this way, BABELRDL supports scalable replicated data management in heterogeneous systems while offering a structured approach for extending RDLs with new functionality.

By describing BABELRDL's design, implementation, and evaluation, this paper makes the following contributions:

1. A CDF-based software architecture for RDL design to enable efficient integration and extension across heterogeneous replicated environments.
2. A reference implementation—BABELRDL that enables seamless interaction across diverse execution platforms in compiled, interpreted, and managed environments, with plug-in support for additional feature extensions.
3. An evaluation comparing BABELRDL with FFI-based RDL integration alternatives, demonstrating advantages in latency, memory consumption, and throughput, while highlighting trade-offs between integration effort and software quality.

The rest of this paper is structured as follows: Section 2 describes the background and motivates the problem; Section 3 details BABELRDL's system design and workflow; Section 4 describes the implementation; Section 5 reports the evaluation results; Section 6 discusses the insights derived from creating and evaluating BABELRDL; Section 7 presents the related state of the art; and finally Section 8 concludes the paper with future work plans.

2. Background and motivation

This section outlines the technical background, including general concepts and specific technologies. Then, we present a software development scenario that motivates the need for an RDL capable of supporting integration and extensibility in heterogeneous distributed systems.

2.1. Background

Frameworks for heterogeneous distributed systems. Several architectural approaches have been introduced to support the development of distributed systems across heterogeneous platforms. In Common Object Request Broker Architecture (CORBA), the Interface Definition Language (IDL) is used to define interfaces implemented in multiple languages [15]. CORBA's inter-language Remote Procedure Call (RPC) marshals arguments/return types, and maps error reporting to language-specific constructs. Java-based MapReduce provides mechanisms to execute tasks across different platforms, e.g., Python and C++ runtimes [16]. The open-source MapReduce implementation, Hadoop, supports additional languages by providing platform-specific data serialization and I/O [17]. Spark extends cross-platform execution support to Java, Scala, Python, and R by relying on the Java Virtual Machine (JVM) services [18]. In the RESTful architecture, heterogeneous peers communicate via a limited set of verbs, such as HTTP GET, PUT, POST, etc. [19].

While these frameworks facilitate general-purpose cross-platform communication or large-scale data processing, they are not sufficient for the unique requirements of RDLs for several reasons:

- **Lack of Specialized RDL Logic:** General communication frameworks like CORBA and REST provide basic primitives but cannot be used out of the box due to their lack of built-in mechanisms for state synchronization, causality tracking, and ensuring eventual convergence across heterogeneous replicas [20–22]. Adding these specialized functionalities within the conventions of the aforesaid frameworks would impose an implementation burden exceeding our design objectives.
- **Runtime Dependencies:** Frameworks such as Spark and Hadoop often rely on a single dominant execution environment, typically the JVM, which hinders their use in cross-platform settings that can combine compiled, managed, and interpreted runtimes without a central dependency [23].
- **Insufficient Extensibility:** Existing solutions provide limited support for extending data-level functionality-such as persistence or error handling-uniformly across multiple programming languages. Adding such capabilities typically requires deep, platform-specific coupling or complex FFI-based integration, which complicates the software architecture and increases resource overhead [24].

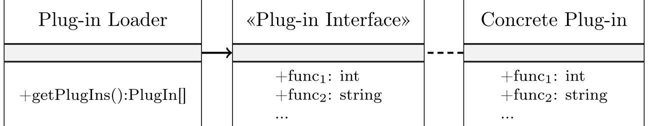


Fig. 1. A typical plug-in structure.

Replicated data libraries (RDLs). Having introduced RDLs earlier, we next describe their core architectural characteristics. A typical RDL consists of three major components: (1) `state`, (2) `interface`, and (3) `update propagation` [25]. The `state`, the internal data structure, represents the data type; RDL data types range from basic data structures, including counter, set, and map, to more complex ones, such as a JSON document [26]. The `interface` provides an API, a set of operations for clients to access and update the RDL state. The `update propagation` component ensures convergence by propagating local updates across replicas. Examples of RDL implementations include Explicitly Consistent Replicated Objects (ECROs)[27], Mergeable Replicated Data Types (MRDTs) [28], and Conflict-Free Replicated Data Types (CRDTs)[29].

Plug-in extensibility. In many software systems, developers often need to extend functionality without altering core components [30,31]. A common approach to accomplish this objective is the plug-in design pattern [32]. The known advantages of this pattern include extensibility, flexibility, and isolation of core application features from custom logic. A high-level structure of the plug-in design pattern appears in Fig. 1. It depicts this pattern's three key components: (1) Plug-in Loader, (2) Plug-in Interface, and (3) Concrete Plug-in [33]. Plug-in Loader is responsible for locating a plug-in interface and ensuring that the corresponding plug-in modules are loaded and ready for invocation; the diagram depicts these functionalities via the `getPlugIns()` function. The Plug-in Interface exposes the method interface for invoking a plug-in's functions. Concrete Plug-in represents the actual plug-in module provided as an external feature to be integrated with the main application. Plug-ins have been widely used as a systematic mechanism for enhancing core functionality with additional features in domains such as web services, content management systems, and integrated development environments [34].

2.2. Motivation

Next, we present a software development scenario that motivates the need for an RDL that facilitates cross-platform integration in heterogeneous distributed systems.

Fig. 2 depicts a distributed system that manages ambient data, such as air temperature and humidity. The system comprises four replica nodes: a `Collection Replica`, hosting an IoT device that collects ambient data; a `Persistence Replica`, hosting a database engine; a

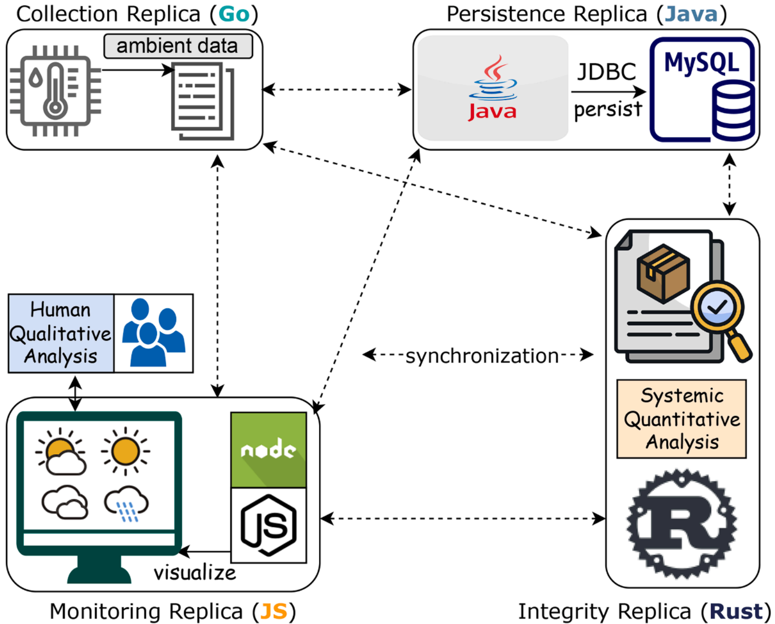


Fig. 2. Ambient data mesh: collection, persistence, monitoring, and auditing.

`Monitoring Replica` that graphically depicts the collected data for human oversight; and an `Integrity Replica` for automated system auditing. Each replica is implemented in a language best suited for its functionality: the `Collection Replica` uses Go to capture sensor data resource-efficiently; the database engine of `Persistence Replica` uses a popular JDBC API [35] for easy database interfacing; the `Monitoring Replica` uses the powerful Node.js graphical frameworks; and the `Integrity Replica` utilizes Rust to provide memory-safe, high-performance quantitative analysis.

Each replica platform and runtime environment is selected based on its respective strengths [36]. The data, replicated across four replicas in this mesh architecture, can be managed using an RDL. As ambient data is collected, the updates propagate to the persistence, monitoring, and integrity replicas. This architecture supports diverse sources of data correction: while a human analyst at the `Monitoring Replica` performs qualitative analysis to correct visual anomalies, the `Integrity Replica` simultaneously performs quantitative analysis to detect and correct systemic errors or sensor drift. Updates from any of these sources are automatically synchronized across the remaining replicas to maintain a consistent state.

This example calls for support of four heterogeneous platforms spanning compiled, managed, and interpreted runtimes. Existing RDLs cannot be applied directly in this scenario. A monolingual RDL would be inapplicable; an RDL with a binding to a designated language would be inapplicable too, unless the example uses that particular language as well. Hence, to integrate an existing RDL, developers must rely on FFI techniques, which raise questions about whether using FFI to interoperate with RDL would substantially complicate the resulting software architecture. Would the resource constraints of the `Collection Replica` or the `Integrity Replica` preclude meeting the memory consumption or access requirements of the FFI libraries?

In addition, it would be impossible to create an RDL that provides *all* functionalities required in different application scenarios. For instance, consider error handling. The `Collection Replica` could collect faulty sensor data (e.g., incorrect temperature values due to hardware malfunction); the `Persistence Replica` could incorrectly persist the data without proper validation; the Node.js frontend of `Monitoring Replica` could display inaccurate visualizations, while the auditing logic of `Integrity Replica` could fail to flag corrupted entries due to localized memory faults. If an RDL lacks error-handling support, application programmers must implement this functionality in the application space. Providing effective error handling for a distributed system is challenging [37]. Doing so in a heterogeneous cross-platform environment would be even harder due to the heterogeneity of languages, frameworks, and data exchange formats. If an RDL does provide error handling, using it across heterogeneous replica platforms would face challenges similar to those encountered when integrating the library itself. Hence, the issue of extensibility is closely linked to the integration strategies for RDLs in cross-platform distributed systems.

This example motivates the need for a multilingual RDL that not only supports efficient cross-platform coordination but also allows developers to extend its functionality in a single language and propagate those capabilities across replicas. BabelRDL addresses both needs: it uses a common data format (CDF) to enable interoperable synchronization across compiled, interpreted, and managed execution environments, and it provides a plug-in extensibility mechanism that allows functionality—such as error handling—to be implemented once and reused across replicas regardless of their implementation language. While this example illustrates a specific use case, it is representative of numerous systems that employ replicated data to fulfill their business requirements. Specifically, this example combines compiled, interpreted, and managed execution environments to leverage their respective strengths to meet specific requirements.

3. System design and overview

In this section, we describe BabelRDL's system assumptions, design, and the workflow between its components.

3.1. System assumptions

BabelRDL assumes a replicated data system consisting of a set of uniquely identified replicas that communicate via asynchronous message passing. Replicas may experience crash failures and are assumed to recover by restoring their state from durable storage. At least one replica is assumed to remain available to preserve system liveness. The

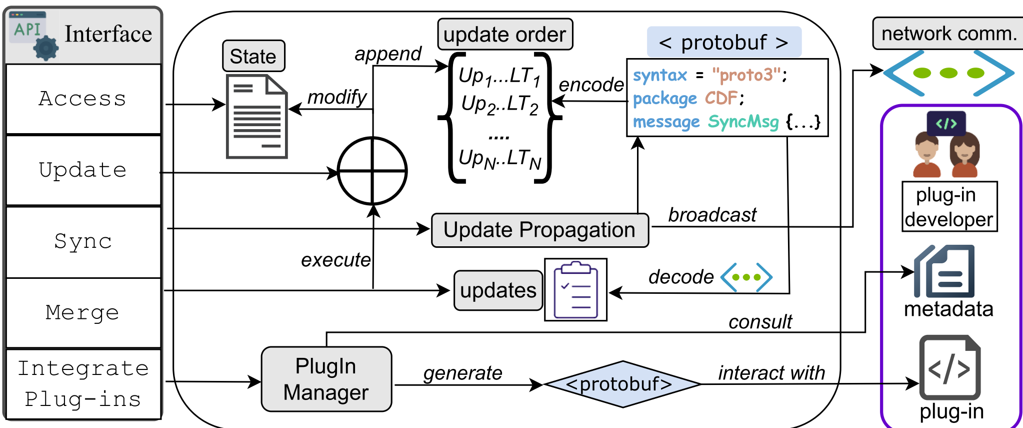


Fig. 3. BabelRDL's system components and their interactions.

system model assumes reliable communication channels that eventually deliver messages without loss, duplication, or corruption, and that temporary network partitions eventually heal. These assumptions do not model raw network behavior; rather, they reflect guarantees commonly provided by existing transport and middleware layers on which RDLs are typically built. We adopt this abstraction boundary to focus on BabelRDL's architectural contribution—namely, cross-platform interoperability and extensible integration of replicated data libraries—rather than on fault-tolerance mechanisms [38] that are orthogonal and well-studied. Relaxing these assumptions would require changes to the underlying communication substrate but would not fundamentally alter the design principles or extensibility model introduced by BabelRDL.

3.2. System components

Fig. 3 depicts BabelRDL's system components and their interactions. BabelRDL's Interface component exposes five API functions to application developers: (i) `Access`, (ii) `Update`, (iii) `Sync`, (iv) `Merge`, and (v) `Integrate Plug-ins`. The functions from (i) to (iv) are commonly found across various RDLs; in contrast, function (v) is unique to BabelRDL and enables its plug-in extensibility.

Guided by classic RDL design principles, the core API functions deliver the following functionalities. `Access` retrieves the local replica's state. `Update` modifies the local replica's state in a fashion required by the data structure (e.g., increment/decrement for Counter, or add/remove for Set, etc.). When `Update` modifies the local state, the replica's *update order*—a replica's causal record of update operations—is modified accordingly. The *update order* is represented with Lamport timestamps (LT) [39]. `Update` can be invoked by both the application code and the RDL implementation itself to process the merge requests from other replicas. How LT-based update ordering ensures strong eventual consistency is proved in Appendix A.

`Sync` propagates a replica's local state to the remaining replicas. As BabelRDL's *raison d'être* is to support cross-platform coordination among heterogeneous replicas, its Update Propagation component provides enhanced services. It can accept a propagation request in any supported source language and deliver it in the target language. To that end, BabelRDL represents the update order in a language-independent fashion. The reference implementation uses the CDF provided by Google Protocol Buffers (protobuf) [40], one of the most widely used implementations of this concept supported by a major corporation. In particular, protobuf provides cross-platform and language-neutral data serialization [41]. Furthermore, protobuf's compact binary message format reduces the amount of data transmitted, potentially improving network latency and throughput, especially in high-performance or resource-constrained environments [42]. With the help of protobuf, BabelRDL encodes the update order and then broadcasts the encoded message over the network, marked as *network comm.* in the diagram. `Merge` gets triggered when a replica receives synchronization requests from other replicas. To execute the merge, BabelRDL first decodes the incoming message using protobuf, thereby extracting a list of updates. Then BabelRDL executes the updates on the list one by one by invoking the `Update` function.

Due to the great diversity of application domains and types of applications that use replicated data, it would be impossible to determine a complete set of features that should be provided by an RDL, particularly if it is cross-platform. To that end, `Integrate Plug-ins` provides a systematic way to extend BabelRDL with additional features by following the plug-in architectural pattern. BabelRDL's programming model for developing third-party plug-ins includes two affordances: (1) receiving the information (e.g., parameters, state changes, etc.) of the invoked core functions; (2) requesting the invocation of any core function. Notice that these affordances embody bi-directional interaction between the BabelRDL's core API and plug-ins.

To configure which plug-in to execute, developers can parameterize the `Integrate Plug-ins` function with metadata that specifies how the plug-in interacts with RDL core functions. Plug-ins are distinguished by unique system-wide IDs, which are also passed as parameters. `Integrate Plug-ins` uses the PlugIn Manager component, which, after consulting metadata, generates the protobuf code required for interaction between the plug-in and RDL core functions. By relying on CDF, BabelRDL ensures cross-platform interaction not only for core RDL functions but also for its plug-in extensibility. By supporting plug-ins written in a single execution platform chosen by the developer, BabelRDL enables plug-in extensibility in a heterogeneous environment. BabelRDL integrates these single-language plug-ins with core RDL functions, irrespective of which execution environments the core functions use. The curious reader may wonder whether the plug-in-introduced functionalities may violate system consistency. Appendix B explains formally why it is not the case.

4. Implementation

Next, we describe BabelRDL's reference implementation. This description includes BabelRDL's data structures and their implementation in different languages. We conclude the section by describing BabelRDL's plug-in extensibility.

4.1. BabelRDL's data structures

BabelRDL currently supports three RDL data structures: Counter, Set, and Map. A counter is a replicated integer and can be used in distributed systems to maintain a consistent count across replicas. For example, a replicated counter can be used for counting the number of likes across replicated social media sites, so each replica increments or decrements its local counter state, with the counters on each site eventually converging [43]. A Set data type maintains a collection of unique elements across replicas. For example, a set can represent an online gift registry, where the same item can be added from multiple replicas, but only one copy ends up in the set [44]. A Map data type maintains a collection of key-value pairs across replicas. This data type is more expressive, as each of its keys can be associated with another data type (such as Counter or Set) as its value. For example, a replicated map can maintain a shopping cart in an e-commerce application [45]. Each product identifier (key) can be associated with a Counter RDL (value) that tracks the quantity.

4.2. Multilingual implementation & interoperability

We aim to demonstrate BabelRDL's versatility within a manageable implementation effort. Hence, our approach strategically selects representative languages from the main execution models: compiled, interpreted, and managed. Specifically, BabelRDL currently supports Rust and Go (compiled), JavaScript (interpreted), and Java (managed). By demonstrating support for these diverse languages, we argue that, with additional engineering effort, BabelRDL can be extended to other languages that follow these execution models. In terms of required engineering effort, implementing the three RDL data structures required approximately 965 lines of Rust code, 1382 lines of Go code, 788 lines of JavaScript code, and 1056 lines of Java code. These uncommented lines of code (ULOC) measurements reflect the effort required to implement the common RDL core functionalities of `Access`, `Update`, `Sync`, and `Merge`.

These are popular languages, so it is not surprising that they have been used to implement several existing RDLs. However, most of these libraries are designed for a single language, requiring additional effort to support cross-platform replication. Instead, these libraries have become a source of implementation insights that BabelRDL draws on to provide the building blocks for replicated data systems. What distinguishes our approach is its seamless cross-platform coordination of replicas via CDF. We use the CDF of Protobuf to define the *update order*, which is propagated to the remaining replicas to ensure convergence. The protobuf libraries for each language[2] transform and propagate the transformed data across different replicas. As a specific example of how BabelRDL uses protobuf, consider the definition of the synchronization message format that enables interoperability across languages for a replicated Set data structure.

Listing 1 Set's Synchronization Message protobuf

```
1 syntax = "proto3";
2 package replicatedSet;
3 message SyncMsg {
4     int32 replicaId = 1;
5     repeated string updates = 2;
6 }
```

Listing 2 Rust's Encoding and Decoding with protobuf

```
1 // Encode
2 let msg = replicated_set::SyncMsg {
3     replica_id: self.id,
4     updates: self.updates.clone(),
5 };
6 // Serialize to bytes
7 let encoded_msg: Vec<u8> = prost::Message::encode_to_vec(&msg);
8 // Decode
9 let msg = replicated_set::SyncMsg::decode(&encoded_msg[..]).
10   expect("Failed to decode SyncMsg");
11 let requester_replica_id: i32 = msg.replica_id;
12 let requested_updates: Vec<String> = msg.updates;
```

Listing 3 Go's Encoding and Decoding with protobuf

```
1 // Encode
2 msg := &replicatedSet.SyncMsg {
3     replicaId:  int32(Set.Id()),
4     Updates:    []string {},
5 }
6 msg.Updates = append(msg.Updates, Set.updates...)
7 // Decode
8 var msg replicatedSet.SyncMsg
9 proto.Unmarshal(requestSyncMsg, &msg)
10 requesterReplicaId := int(msg.GetreplicaId())
11 requestedUpdates := msg.GetUpdates()
```

Listing 4 JavaScript's Encoding and Decoding with protobuf

```
1 // Encode
2 let msg = new SyncMsg()
3 msg.setreplicaId(this.getId())
4 msg.setUpdatesList(this.updates)
5 // Decode
6 let receivedMessage = SyncMsg.deserializeBinary(requestSyncMsg)
7 let requesterReplicaId = receivedMessage.getreplicaId()
8 let requestedUpdates := receivedMessage.getUpdatesList()
```

Listing 5 Java's Encoding and Decoding with protobuf

```
1 // Encode
2 SyncMsg msg = SyncMsg.newBuilder().setreplicaId(this.id)
  ↪   .addAllUpdates(this.updates).build();
3 // Decode
4 SyncMsg msg = SyncMsg.parseFrom(requestSyncMsg);
5 int requesterReplicaId = msg.getreplicaId();
6 List<String> requestedUpdates = new ArrayList<>();
7 for (int i = 0; i < msg.getUpdatesCount(); i++) {
8     requestedUpdates.add(msg.getUpdates(i));
9 }
```

The code listing 1 demonstrates how the declarative programming model of protobuf allows a concise definition of messages that can be shared across different languages. Listings 2, 3, 4, and 5 show how the defined message maps to language-specific structures in Rust, Go, JavaScript, and Java, respectively.

4.3. Plug-in extensibility

BabelRDL's `Integrate Plug-ins` API function performs two tasks: (1) consulting the provided metadata to determine the information that

[2]
- tokio-rs/prost [46] for Rust
- google.golang.org/protobuf [47] for Go
- npm protocol-buffers [48] for JavaScript
- protobuf-java-3.21.12.jar [49] for Java

needs to be shared between the provided plug-in and BABELRDL's PlugIn Manager component, and (2) generating the corresponding protobuf code that reflects the information. Algorithm 1 demonstrates the key steps that BABELRDL follows to integrate a plug-in. To accomplish task (1), parameterized with a unique plug-in ID and corresponding metadata, this function checks for the presence of a JSON metadata file (lines 3–4). For example, to integrate the plug-in whose ID is idx, developers can invoke `IntegratePlugIns(idx, metaData_idx.json)`. Based on the metadata, BABELRDL then generates and compiles the protobuf code required for the interaction, thus accomplishing task (2) (lines 9–14). If the compilation fails, BABELRDL reports an error and abandons the attempt to integrate that plug-in (lines 16–17). If compiled successfully, BABELRDL then deploys the plug-in to communicate with the core RDL functionality via a socket, an implementation choice we have made. The language-specific parts of `Integrate Plug-ins` were implemented in approximately 134, 175, 120, and 158 ULOC of Rust, Go, JavaScript, and Java, respectively.

Algorithm 1 Integrate plug-ins function.

```
1: procedure INTEGRATE_PLUGINS(pluginIDs, metadata)
2:     for all pId ∈ pluginIDs do
           Task 1: Consult Metadata
3:         if metadata file exists for pId then
4:             protoSyntax ← consult(metadata)          ▷ Info to share with PlugIn Manager
5:         else
6:             reportError()                             ▷ Missing metadata, skip plugin
7:             continue
8:         end if
           Task 2: Generate and Deploy
9:         protobufCode ← generate(protoSyntax)
10:        if compile(protobufCode) then
11:            pluginCode ← locate(pId)
12:            address ← GetPluginAddress(metadata)
13:            deploy(pluginCode, address)               ▷ Deploy pluginCode at address
14:            interact(PlugInManager, pluginCode, address)   ▷ Interact with plugin
15:        else
16:            reportError()                             ▷ Compilation failed
17:            continue                                  ▷ proceed to next plug-in
18:        end if
19:    end for
20: end procedure
```

We adopt a socket-based communication mechanism between BABELRDL and integrated plug-ins. Sockets provide a language- and runtime-agnostic communication channel that supports independent processes and multiple execution environments, making them a practical choice for cross-platform extensibility. This design follows established precedents in software architecture, where socket-based APIs are used to integrate loosely coupled components across languages and runtimes in domains such as robotics, gaming engines, and operating system services [50–52].

5. Evaluation

The following research questions drive the evaluation of BABELRDL:

1. **RQ1: Performance Characteristics:** How does BABELRDL compare to FFI-based integration alternatives in terms of their respective performance characteristics?
2. **RQ2: Integration Software Quality:** How does BABELRDL compare to FFI-based integration alternatives in terms of their impact on software quality?
3. **RQ3: Extension Software Quality:** In terms of extensibility, how does BABELRDL compare with ad-hoc approaches with respect to their impact on software quality?

We begin by describing the two cross-language interoperability strategies used in our evaluation, the experimental setup, and the evaluation subjects. Then, we present the results of our performance and software quality experiments, which address the research questions above.

5.1. Strategies for cross-language interoperability

Fig. 4 depicts the two strategies for integrating RDLs in multilingual environments, widely utilized in modern distributed systems [53,54]. In

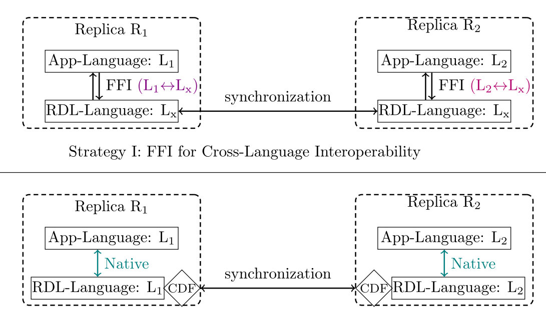


Fig. 4. Cross-language integration strategies.

both strategies, we assume that the system comprises two replicas: R_1 and R_2. The application logic in R_1 is written in language L_1, while in R_2 it is written in L_2. In Strategy I (top), replicated data management is provided by a monolingual RDL, written in language L_x. In this strategy, the application interacts with the library via FFI, whose type is determined by the specific pair of interacting languages. In this case, two different FFIs are required: $L_1 \rightleftarrows L_x$ for replica R_1, and $L_2 \rightleftarrows L_x$ for replica R_2. Please note that the direction of language interaction can further differentiate the type of FFI libraries required [55]. In other words, FFI for $L_1 \rightarrow L_x$ may not be applicable for $L_x \rightarrow L_1$.

In Strategy II (bottom), the RDL is multilingual, meaning that the library code is written in L_1 for replica R_1 and in L_2 for replica R_2. That is, the RDL's language is the same as the replica's application logic. Hence, the application code interacts natively with the library through regular local function calls. However, when exchanging messages across replicas, the message format must be mapped to a common data format, as indicated by the CDF in the figure. For interoperability, BABELRDL follows Strategy II using protobuf as CDF.

5.2. Experimental setup

For our experiments, we implemented the system used as our motivating example described in Section 2.2: a distributed system that collects, persists, visualizes, and audits ambient weather data. We deployed our implementation on an eight-replica cluster with the following configuration. Two `Collection Replicas` are hosted on separate 32-bit Raspberry Pi 3 devices running Raspbian 9. Each collection replica is equipped with a DHT11 temperature-humidity sensor to collect ambient environmental data. To emulate a resource-constrained deployment environment, we used Linux control groups (cgroups) [56] to limit CPU resources on these devices. Two `Persistence Replicas` are deployed as separate processes on a 64-bit Ubuntu 20.04 desktop machine with 32 GB RAM and an Intel Core i7 processor. These replicas are responsible for persisting the most recent sensor readings in a MySQL database using JDBC-based access. Two `Monitoring Replicas` are deployed as separate processes on a 64-bit Windows 11 laptop with 8 GB RAM and an Intel Core i5 processor. These replicas run Node.js-based applications that provide visualization and monitoring facilities, enabling users to inspect, analyze, and manually correct collected ambient data. Two `Integrity Replicas` are hosted as separate processes on a 64-bit Ubuntu 20.04 laptop with 8 GB RAM and an Intel Core i5 processor. These replicas support continuous data integrity checking by performing concurrent quantitative analyses to identify inconsistencies, systemic errors, or sensor drift in the collected data and to assist in corrective actions. To emulate a geographically distributed deployment, each replica is assigned a distinct network endpoint using WireGuard [57], enabling replicas to communicate as if deployed across a wide-area network. The system uses three replicated data types across replicas: Counter to track the total number of collected data points; Set to ensure the uniqueness of data points in the system; Map to store key-value pairs mapping col-

lection timestamps to sensor readings. In addition, the integrity replicas employ Map data types to maintain and analyze derived metadata required for integrity verification and anomaly detection.

5.3. Evaluation subjects

To compare BabelRDL with FFI-based RDL integration (Strategy I), we experimented with four open-source third-party RDLs. Three of these libraries are monolingual, each written in one of the languages used in our motivating example. Go-CRDT [58] is Go's built-in RDL package, available from version 1.19. Legion [59] provides a JavaScript implementation of various replicated data types along with a WebRTC-based networking layer. T-CRDT [60] is a Java RDL that allows programmers to choose either predefined concurrency-conflict policies or define their own. From each RDL, we evaluated Counter, Set, and Map.

To enable these monolingual libraries to work with our multilingual application, we provided FFIs for each library's API. For example, Go-CRDT works directly with `Collection Replica`; however, to use it with `Persistence Replica` and `Monitoring Replica`, interactions are routed via Go↔Java and Go↔JavaScript FFIs, respectively. Similarly, for Legion, interactions from Go and Java replicas are routed via JavaScript↔Go and JavaScript↔Java FFIs, and for T-CRDT, interactions from Go and JavaScript replicas are routed via Java↔Go and Java↔JavaScript FFIs. With the inclusion of `Integrity Replica`, additional FFI paths were required. To invoke Go-CRDT from Rust, we compiled the Go library as a C-compatible shared library and accessed it via `extern "C"` bindings. Legion was accessed by embedding a JavaScript runtime within Rust and invoking the JavaScript-based RDL through the runtime's foreign-function interface. For T-CRDT, Rust interacted with the Java library through the Java Native Interface (JNI), linking against the JVM and invoking Java methods via C Application Binary Interface (ABI) bindings. Hence, although these RDLs are monolingual, FFI-based integration enables our multilingual replicas to access the required RDL APIs, at the cost of increased integration complexity and fragile cross-language dependencies.

In contrast to these monolingual libraries, we used Automerge [61] as a fourth evaluation subject. Automerge adopts a Rust-centric architecture in which the core RDL logic is implemented exclusively in Rust, while Go [62], Java [63], and JavaScript [64] access this core through language-specific bindings. In our evaluation, each replica uses the Automerge API provided for its host language, while all RDL operations are executed by the shared Rust core. This design eliminates the need for application-level FFI construction across replicas, while still allowing heterogeneous replicas to interact through a common Rust-based RDL implementation.

5.4. RQ1: performance characteristics

Summary: BabelRDL exhibits lower latency and memory consumption, and higher throughput than FFI-based integration alternatives.

To evaluate the performance characteristics of BabelRDL and compare them with those of the other four evaluation subjects, we conducted several benchmarks with workloads that differed in the number of RDL operations. The benchmarked operations concerned the fundamental RDL functionalities of accessing, updating, merging, and propagating the replicated states. We excluded the operation that collects ambient data in the IoT device, as this functionality is not part of the classic RDL API. Figs. 5a, 5b, and 5c depict the average latency of Counter, Set, and Map, respectively, for all five RDL systems. Across all experiments, the latency increased proportionally to the workload. However, BabelRDL exhibited the lowest latency among the four evaluation subjects. This efficiency is due to BabelRDL's capability to interact with the application logic in the same language, whereas the other RDLs rely on FFI-based interactions. Among the other four evaluation subjects, Legion showed the worst performance due to the heavy overhead of JavaScript's runtime and interaction mechanisms, which involve WebAssembly as the FFI enabler. The need to compile code at runtime incurs high performance overhead, unlike the interactions between Go and Java, which require no runtime code generation [65]. Automerge's performance falls between that of BabelRDL and the FFI-based monolingual RDLs. This is because, although all RDL operations are executed by the shared Rust core, each replica accesses the core through language-specific bindings. These bindings introduce modest overhead compared to BabelRDL's fully native execution, but remain more efficient than the multiple cross-language FFIs required by Go-CRDT, Legion, and T-CRDT. On average, BabelRDL showed the shortest latency, compared to Automerge, Go-CRDT, Legion, and T-CRDT by ≈ 1.34×, ≈ 2.97×, ≈ 4.63×, and ≈ 4.08×, respectively.

Figs. 6a, 6b, and 6c depict the average of the peak memory usage by Counter, Set, and Map, respectively. For each of the three data types, memory consumption increases slightly with the number of operations in the workload. However, for Set and Map, memory consumption increases more than for Counter, as the size of these two data types has a greater impact on memory usage. BabelRDL exhibited the lowest memory consumption due to the efficiency of the application code that invoked RDL functions in the same language. The memory consumption of Automerge is also intermediate: its Rust core maintains a single in-memory state per data type, avoiding the duplication of FFI-induced buffers or runtime contexts. However, the language-specific bindings incur slight additional memory usage compared to BabelRDL's native integrations. On average, BabelRDL outperformed Automerge, Go-CRDT, Legion, and T-CRDT by ≈ 1.29×, ≈ 1.76×, ≈ 3.18×, and ≈ 2.76×, respectively.

Figs. 7a, 7b, and 7c demonstrate the throughput distribution for Counter, Set, and Map, respectively. BabelRDL's median throughput is the highest, indicating its superior performance as compared to the other evaluation subjects under the same workload. BabelRDL's interquartile range (IQR) is always the narrowest due to the diminished variability across workloads. Automerge achieves higher throughput than the monolingual FFI-based RDLs but lower than BabelRDL, reflecting the trade-off between central Rust execution and per-replica native execution. Its narrower IQR compared to FFI-based RDLs also indicates more predictable performance due to the uniformity of the Rust core across replicas. In contrast, the high IQR among the other subjects indicates high variability in their throughput due to their reliance on dissimilar FFI techniques across languages.

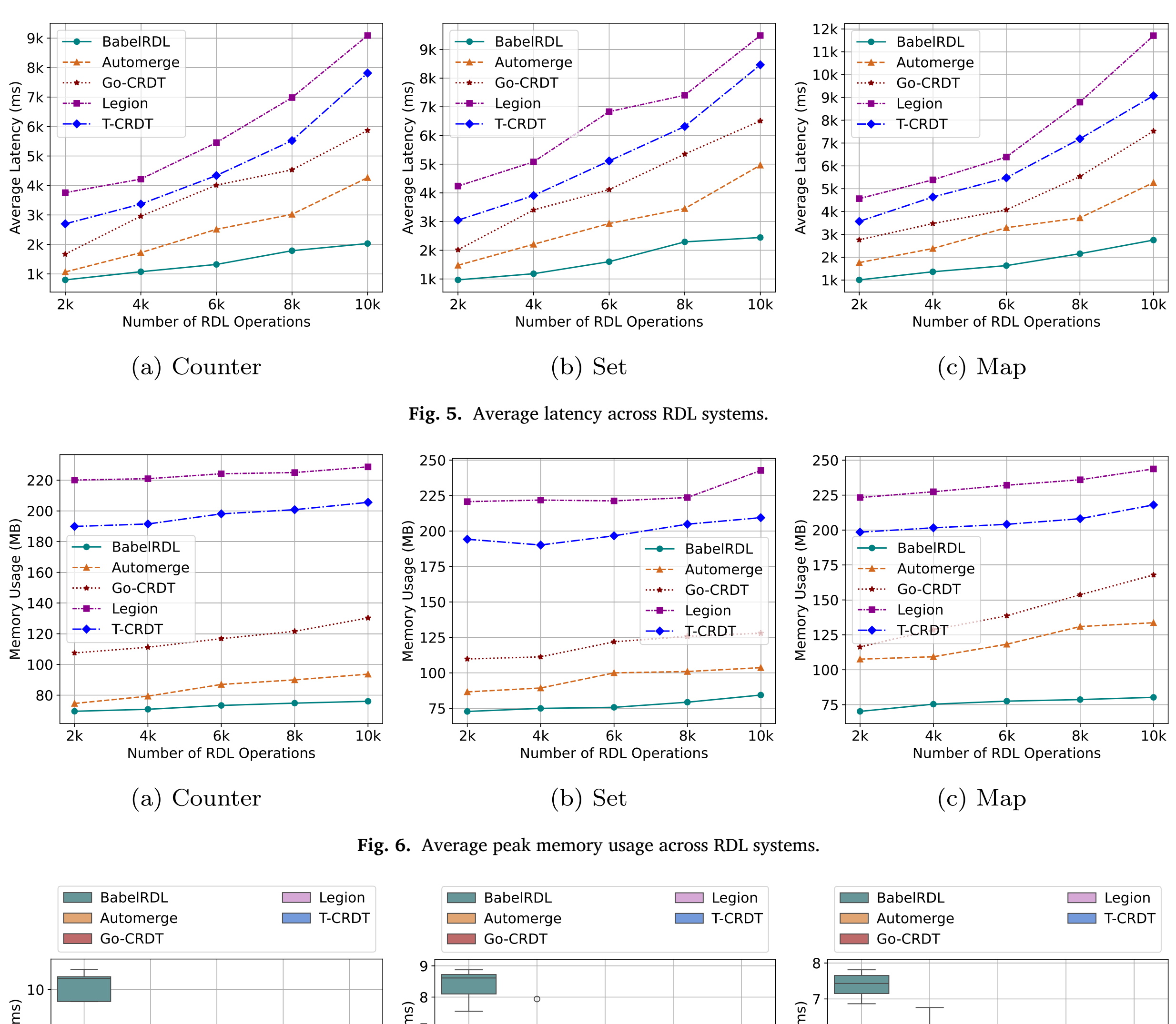
Table 1
Uncommented lines of code (ULOC) for RDL integration.

RDL	Rust	Go	JavaScript	Java
BabelRDL	965	1382	788	1056
Automerge	–	–	–	–
Go-CRDT	417	–	358	506
Legion	653	812	–	465
T-CRDT	482	656	373	–

5.5. RQ2: Integration software quality

Summary: BabelRDL demonstrates a trade-off between implementation effort and integration code quality, as evidenced by higher integration effort but lower Cyclomatic complexity and Halstead effort.

To assess the programming burden of integrating an RDL into a multilingual application, we measured the uncommented lines of code (ULOC) required for integration. ULOC has been widely used as an estimate of implementation effort [66]. A higher ULOC indicates a greater one-time programming effort required to integrate an RDL into application code. To evaluate the structural complexity and maintainability of



(a) Counter (b) Set (c) Map

Fig. 5. Average latency across RDL systems.

(a) Counter (b) Set (c) Map

Fig. 6. Average peak memory usage across RDL systems.

(a) Counter (b) Set (c) Map

Fig. 7. Average throughput across RDL systems.

Table 2
Cyclomatic complexity of RDL integration code.

RDL	Rust	Go	JavaScript	Java
BabelRDL	39	48	33	42
Automerge	39	117	69	118
Go-CRDT	71	48	55	102
Legion	83	94	33	51
T-CRDT	57	78	59	42

Table 3
Halstead effort of RDL integration code.

RDL	Rust	Go	JavaScript	Java
BabelRDL	6921.5	5784.9	9450.2	7979.97
Automerge	6921.5	49064.87	19045.67	63427.54
Go-CRDT	13679.54	5784.9	18915.78	45281.67
Legion	34605.4	71891.0	9450.2	93057.5
T-CRDT	27852.67	69758.01	80271.35	7479.97

the integration logic, we additionally measured Cyclomatic complexity and Halstead effort, two widely adopted software quality metrics [67]. Cyclomatic complexity quantifies the number of linearly independent execution paths in the source code by counting decision points such as conditionals and loops [68]. Higher values indicate more complex control flow and reduced maintainability. Halstead effort measures the cognitive complexity of source code based on the number of operators and operands [69]. For both metrics, lower values correspond to simpler, more maintainable code [70].

Tables 1, 2, and 3 report the required ULOC, Cyclomatic complexity, and Halstead effort, respectively, for RDL integration. Our measurements focus exclusively on the integration functionality, i.e., the appli-

cation logic that invokes the RDL API. ULOC reflects the amount of additional code required during integration, whereas Cyclomatic complexity and Halstead effort characterize the complexity and maintainability of that code.

For Automerge, the ULOC required for integration is zero, as all language-specific bindings are provided by the library itself. Nonetheless, Cyclomatic complexity and Halstead effort values are non-zero because they capture the structural and cognitive complexity of the pre-existing language-specific API code that developers invoke to integrate Automerge with their application. These values serve as a reference point for comparing the inherent complexity of integration interfaces across all RDLs, without imposing any additional implementation effort on developers.

Unlike FFI-based approaches, BabelRDL reimplements the core RDL API in each supported system language. As a result, BabelRDL incurs a higher ULOC due to the need for explicit, language-native implementations. This requires a higher one-time implementation effort than approaches that rely on cross-language bindings. However, this additional implementation effort yields substantial improvements in code quality. Across all evaluated languages, BabelRDL consistently exhibits lower Cyclomatic complexity and Halstead effort than FFI-based alternatives. These results indicate that the integration logic in BabelRDL is structurally simpler, easier to understand, and less cognitively demanding to maintain.

By avoiding FFIs and exposing fully native APIs, BabelRDL eliminates cross-language control flow and error-handling logic, while preserving language-specific type safety, memory management, and runtime semantics. This design reduces integration complexity and avoids subtle bugs commonly introduced by mismatched ownership, lifetimes, and exception models across language boundaries [71]. Consequently, while BabelRDL requires more integration code, that code is architecturally cleaner and better suited for long-term evolution and maintenance.

For each supported language, BabelRDL follows the same logical implementation as the corresponding native RDL. Therefore, the Cyclomatic complexity and Halstead effort of BabelRDL match those of the native implementations. For example, in the case of T-CRDT, the Java integration exhibits Cyclomatic complexity and Halstead effort values of 42 and 7479.97, respectively, identical to those of the native Java implementation.

5.6. RQ3: Extensibility software quality

Summary: BabelRDL demonstrates superior software quality as evidenced by the measures of lower Lack of Cohesion in Methods and Coupling Factor when extending RDLs with additional features.

To evaluate BabelRDL's extensibility, we conducted an experiment in which we provided three non-trivial features. The provided features are (i) logging, (ii) undo, and (iii) rollback. Logging, a common middleware feature, provides a detailed and chronological record of all updates executed across distributed replicas [72]. Frequently used as an audit trail, this record enables developers to trace error sources, monitor system behavior, and ensure compliance with applied policies. As such, logging is essential for non-trivial debugging, reproducing bugs, and handling faults. Both undo and rollback are examples of error-handling strategies, essential for ensuring the reliability of distributed systems. Both strategies aim to handle errors by reverting the replicated state to a previous point. Despite their shared objective, these strategies operate in distinctly different ways. While rollback simply brings the replicated state to a given prior persisted state [73], undo counteracts the effects of erroneous updates, thus offering a more fine-grained form of state restoration [74].

To prepare the evaluation subjects to compare with BabelRDL, we integrated the aforementioned features in two modes: (1) using BabelRDL's plug-in architecture and (2) in an ad-hoc fashion for the other three evaluation subjects. While BabelRDL can integrate any extra feature implemented in a single language via its plug-in architecture, the other subjects require each feature to be implemented directly in the same language as the RDL core. Hence, we provided all three features in Go for Go-CRDT, JavaScript for Legion, and Java for T-CRDT. In terms of implementation effort, whether provided as a plug-in or ad hoc, each feature required approximately the same amount of source code, with ULOC numbers ranging from about ≈ 200 for logging to more than ≈ 350 for undo.

Table 4
Software quality for plug-in extensibility.

Features	Subjects	LCOM	CF
Logging	BabelRDL	**9**	**0.46**
	Automerge	12	0.49
	Go-CRDT	16	0.78
	Legion	13	0.71
	T-CRDT	15	0.86
Undo	BabelRDL	**12**	**0.65**
	Automerge	17	0.71
	Go-CRDT	16	0.87
	Legion	21	0.94
	T-CRDT	18	0.99
Rollback	BabelRDL	**13**	**0.69**
	Automerge	18	0.78
	Go-CRDT	20	0.77
	Legion	17	0.86
	T-CRDT	22	0.91

We aim to assess the suitability of a plug-in architecture for introducing additional features, so we selected software metrics commonly used to measure modularity, a key indicator of strong separation of concerns [75]. Specifically, we collected Lack of Cohesion in Methods (LCOM) and Coupling Factor (CF). As higher modularity streamlines comprehensibility and maintainability, lower LCOM indicates better modularity [76]. With fewer inter-dependencies across modules, lower CF indicates enhanced maintainability, higher scalability, and streamlined testing [77]. Table 4 shows these metrics for each evaluated subject. Both LCOM and CF values are lower for BabelRDL, highlighting its superior software quality, attributed to its plug-in architecture. This quality is achieved by encapsulating additional functionalities within modular plug-ins, each designed with a clear single-entry access point for external clients. Although Automerge provides structured APIs, feature extensions still need to be woven into existing language-specific interfaces, resulting in higher cohesion and coupling than BabelRDL's plug-in-based extensions.

5.7. Comparative summary of evaluation results

To consolidate the findings from RQ1–RQ3, Table 5 summarizes how each evaluated approach supports the key architectural and software quality claims examined in this work. The table contrasts properties supported by empirical measurements with those grounded in architectural design and qualitative analysis, clarifying the strength and scope of evidence for each claim. Overall, this synthesis highlights the trade-offs among existing approaches and situates BabelRDL within the broader design space of cross-platform RDLs.

6. Discussion

In this section, we discuss our evaluation results and their implications for RDL integration and extension.

6.1. FFI implementation insights

To better understand why using FFI increases the programming burden and code complexity, we next shed light on the specific techniques

Table 5
Summary of key evaluation dimensions across RDL systems.

Aspect	BabelRDL	Automerge	Monolingual RDLs
Cross-platform Integration	Native language APIs; cross-replica interoperability via a CDF	Rust-centric core exposed through language-specific bindings	No built-in cross-language interoperability; restricted to single runtimes
Performance	Lower latency and memory usage across heterogeneous platforms	Competitive native performance; overhead via bindings in polyglot settings	Native performance in monolingual setups; high FFI overhead for polyglot use
Integration Complexity	Higher upfront engineering effort; simpler control flow and maintenance	Low upfront engineering effort; high complexity at FFI boundaries	Low initial code volume; complex FFI-based integration and maintenance
FFI Dependency	No cross-replica FFI; message-level interoperability	Heavy reliance on FFI bindings to the shared Rust core	Mandatory reliance on FFI for any cross-language scenarios
Extensibility Mechanism	Language-local plug-ins with cross-platform reuse	API-level extensions tightly coupled to library bindings	Ad-hoc, language-specific feature integration and porting
Extensibility Software Quality	Lower coupling; improved modularity and architectural clarity	Moderate modularity; constrained by core-binding synchronization	Tightly coupled feature additions; difficult to port across languages
Suitable Deployment	Heterogeneous, multi-language replicated data systems	Rust-centric applications requiring polyglot support	Mnolingual systems with optional FFI integration

we used to enable different languages to interact via FFI. As it turns out, multilingual functionalities are far from trivial to implement and maintain in the modern development ecosystem. For Go→Java, we used `Java Native Interface (JNI)`, which allows Go code to invoke Java methods through native C functions. However, to invoke the JNI C functions, the Go code had to use the special-purpose `cgo` library [78], which follows domain-specific usage conventions. For Java→Go, the Go functions had to be wrapped in C-calling-convention functions so they could be exported and accessible via the `Java Native Access (JNA)` API. For Go→JavaScript, we used the `go/goja` [79] package, which provides an ECMAScript 5.1 JavaScript runtime within Go, through which Go code can invoke JavaScript functions. For JavaScript→Go, we used `syscall/js` [80] library that makes it possible to compile Go functions to the WebAssembly format, which can then be invoked from JavaScript. For Java↔JavaScript, we used `Nashorn Engine` [81], which embeds a JavaScript engine within Java, through which Java code can invoke JavaScript functions and vice versa.

When combining different languages via FFIs, we encountered significant variability across techniques and frameworks. We were also struck by how fragile some of these technologies are. Many of them require a specific language version or a set of features, with several crippling limitations. For example, the Nashorn engine used for Java↔JavaScript interoperation provides no support for modern ES6 features such as Sets, Maps, and Arrays. One might argue with the FFI choices we have made, but our conclusions are likely to stand with possible alternatives. Overall, the evaluated FFI technologies have proven to be error-prone and fragile, and require a steep learning curve.

Automerge represents a contrasting integration strategy among our evaluation subjects, and examining its interoperability model helps contextualize the FFI-related trade-offs observed above. It adopts a Rust-centric architecture in which the core RDL logic is implemented exclusively in Rust and exposed to other languages through ecosystem-specific interoperability mechanisms. Go bindings rely on `cgo` to interface with a C Application Binary Interface (ABI) wrapper over the Rust core [82], enabling Go applications to invoke Automerge APIs as foreign functions. Java bindings use the Java Native Interface (JNI) to call Rust-compiled shared libraries, enabling Java applications to manipulate Automerge documents via native method invocations. JavaScript bindings compile the Rust core to WebAssembly using `wasm-bindgen` [83], enabling JavaScript applications in both browser and Node.js environments to invoke Automerge functionality without embedding a JavaScript runtime in Rust. While this design avoids reimplementing CRDT logic across languages and demonstrates a range of FFI and cross-runtime integration techniques, the resulting interoperability remains asymmetric: in all cases, control flow originates in the host language, which invokes the Rust library as a dependency, and the Rust core does not invoke application-level logic in the host language. Consequently, Automerge enables cross-language library reuse rather than bidirectional or peer-level multi-language execution.

6.2. *Threats to validity*

Our evaluation results are subject to several threats to validity, which we categorize into internal, external, and construct validity threats. Internal validity concerns whether the observed effects can be attributed to the evaluated designs rather than unintended confounding factors, while external validity concerns the generalizability of the findings to other systems and settings [84]. Construct validity concerns whether the chosen metrics accurately reflect the properties being studied [85].

Internal validity. Our results are derived from a single, end-to-end example application, and some design decisions may have been influenced by its structure. To mitigate this threat, we evaluated four third-party RDLs using their publicly available APIs and applied the same workloads, data types, and deployment settings across all subjects. In addition, all measurements were collected using identical hardware and network configurations, ensuring fair and consistent comparisons. Another internal threat arises from our choice of FFI technologies, as the FFI ecosystem offers multiple alternative libraries and frameworks. We mitigated this risk by selecting widely adopted, well-documented FFI mechanisms for each language pair, favoring standard, commonly used solutions over experimental ones.

External validity. Our evaluation is based on a representative multilingual replicated application that exercises core RDL functionality, including update, merge, and state propagation across heterogeneous replicas. While real-world industrial systems of this type exist, they are rarely open-sourced due to proprietary constraints, making them inaccessible for controlled experimentation. Conversely, publicly available open-source replicated systems typically lack sufficient complexity in their use of multiple programming languages to serve as realistic experimental subjects for multilingual RDL integration. As a result, we designed an example application that captures the essential characteristics of such systems while remaining amenable to systematic evaluation. Furthermore, the evaluated RDLs span diverse design points-including monolingual libraries, FFI-based integration, and a Rust-centric cross-language architecture-broadening the scope of our conclusions. Nevertheless, our results may not directly generalize to systems with fundamentally different replication models or deployment environments.

Construct validity. We evaluated performance using latency, throughput, and peak memory usage, which are standard indicators for assessing replicated data systems. To assess integration and extensibility quality, we used ULOC, Cyclomatic complexity, Halstead effort, LCOM, and CF-widely adopted software engineering metrics for estimating implementation effort, control-flow complexity, cognitive load, and modularity. While these metrics may not capture all aspects of developer experience, they provide quantitative and reproducible proxies for programming burden, maintainability, and extensibility. Finally, our reference implementation of BabelRDL relies on protobuf for cross-replica communication, and thus inherits its serialization and runtime characteristics. As a result, some measurements may be influenced by language-specific protobuf optimizations, though this impact is consistent across all evaluated subjects using the same communication layer.

6.3. Implications

Various factors influence the choice of an implementation language. Sometimes, domain constraints require the use of a particular language or language type. For example, the resource constraints of edge computing may necessitate selecting a language that minimizes resource utilization [86]. Some languages require extensive runtime environments, such as virtual machines or interpreters, which may be impractical to install on resource-limited devices, such as those in IoT setups [87]. Hence, modern distributed applications mix languages and, as such, rely on various FFI technologies and tools for cross-language interoperability [88,89]. Additionally, various business scenarios require maintaining data replicas [90,91]. These applications thus would benefit from an RDL that supports multilingual integration.

Our evaluation has demonstrated the advantages of BabelRDL over alternatives that use a monolingual RDL with FFI bindings to each language. Having compared their performance and software quality, we derived actionable insights to inform the design and integration of RDLs for multilingual replicated data systems.

7. Related work

Our work draws inspiration from multiple research areas, including cross-platform interoperability and system extensibility, which we discuss in turn.

7.1. Cross-platform interoperability

As distributed systems have become increasingly complex and require heterogeneous architectures, it would be unrealistic to expect a single execution environment to be the best choice for the entire system. Hence, mixing different languages is a critical requirement for modern distributed systems. The design challenge is not only to enable seamless interoperation across platforms but also to avoid complex inter-platform interfaces, rigid coding conventions, and poor performance [92]. TruffleVM, a virtual machine-based runtime, executes and combines multiple programming languages, with the languages interoperating via FFI [24]. Another VM-based approach aims to provide a simple, low-overhead FFI for Python software [93]. FIG generates FFI wrappers from annotations to enable interoperability with C/C++ code [94]. Jeannie enables C/Java interoperability via JNI by mixing the constituent code in a single file [95]. Code-interoperability mode treats JVM and Java as a unified programming platform for prototyping heterogeneous code that can execute in parallel on various hardware and from different languages [96]. More recently, cross-platform interoperability has also been explored in the context of replicated data systems and CRDT frameworks. Similar to Automerge, Yjs is a Rust-centric CRDT framework that provides language bindings to environments such as WASM, Ruby, Python, Swift, and Java [97]. However, these frameworks primarily rely on language bindings around a central implementation rather than offering a general architecture for interoperable replicas implemented natively in different languages. In contrast to prior approaches that target specific platform pairs or runtime environments, BabelRDL introduces an architecture that enables cross-platform interoperability across replicas without depending on rigid FFI conventions or incurring their associated performance overhead.

7.2. System extensibility

System extensibility is a design quality that enables systems to adapt and grow without requiring significant modifications to the core architecture. If properly designed, extensibility promotes modularity, enabling developers to build and deploy extensions independently and thereby reducing development time and maintenance costs. Hence, a substantial body of prior research focuses on systematic feature extensibility. One recent approach discusses the challenges of extending knowledge-based system development platforms and describes the obstacles to integrating external functionality via a service-oriented approach [98]. Another work focuses on optimizing the software architecture by efficiently allocating tasks to nodes, thus enabling real-time distributed systems to offer extensibility [99]. Keris, a backward-compatible extension of Java, enables the creation and linking of modules to provide a type-safe, non-invasive extension of Java applications [100]. Several prior works apply plug-in architectures to systematically add features in domains such as collaborative work [101], debugging and testing [102,103], and adaptive web service composition [104]. Although we derive valuable design insights from these prior works, their focus is not on replicated data systems operating across heterogeneous platforms. In contrast, our approach applies plug-in extensibility directly to the architectural design of platform-agnostic RDLs, enabling modular features to operate consistently across diverse deployment environments.

8. Conclusion

We have presented BabelRDL, a software architecture that enables seamless integration and extension of RDLs across diverse platforms and runtime environments in replicated data systems. BabelRDL supports replicas deployed on heterogeneous platforms, including different operating systems and hardware architectures, allowing them to interoperate seamlessly via a CDF. Beyond integration, another benefit of BabelRDL's system design is its systematic extensibility. Specifically, developers can add new features as plug-ins in a single implementation, albeit integrated with all deployment platforms. To our knowledge, BabelRDL is the first architecture for RDL design that features both seamless cross-platform integration and systematic plug-in extensibility, as evidenced by our empirical evaluation.

As future work, we plan to further explore the versatility of BabelRDL's design by supporting additional platforms and runtime environments. We also intend to investigate the generality of BabelRDL's extensibility by developing new plug-ins that maintain compatibility across diverse platforms. Another avenue for investigation is to analyze how the choice of a common data format affects cross-platform replica coordination and overall system performance.

Given that modern software ecosystems combine multiple platforms and runtime environments to meet diverse functional and performance requirements, platform-agnostic architectures are now essential to distributed system design. As data replication becomes increasingly pervasive, the insights and architectural principles introduced in this work provide a foundation for developing maintainable, high-quality replicated data systems.

CRediT authorship contribution statement

Provakar Mondal: Writing – review & editing, Writing – original draft, Conceptualization; **Eli Tilevich:** Writing – review & editing, Writ-

ing – original draft, Supervision, Software, Project administration, Investigation, Conceptualization.

Data availability

We make BabelRDL's source code and its evaluation artifacts available at https://doi.org/10.5281/zenodo.18371279.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Provakar Mondal reports was provided by Virginia Polytechnic Institute and State University. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Lamport timestamp-based eventual convergence

In this section, we demonstrate that by combining a CDF with a deterministic total-ordering, the design of BabelRDL preserves Strong Eventual Consistency (SEC).

A.1. System model and definitions

Let $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ be a set of n replicas. Each replica r_i maintains a local state S_i and an update order $\mathcal{L}_i$ (a log of operations).

Definition A.1 (Update Tuple). An update u is defined as a tuple $u = \langle op, arg, id_r, ts\rangle$, where op denotes the operation type (e.g., increment, add), arg represents the operation arguments, $id_r \in \mathbb{N}$ is the unique identifier of the originating replica, and $ts \in \mathbb{N}$ is the Lamport Timestamp (LT) [39] assigned at the moment of update generation.

Definition A.2 (Total Ordering $\Longrightarrow$). For any two updates $u_a, u_b \in \mathcal{L}$, we establish a strict total order $u_a \Longrightarrow u_b$ based on the lexicographical comparison of their timestamps and replica identifiers. Formally, $u_a \Longrightarrow u_b$ iff $(u_a.ts < u_b.ts) \vee (u_a.ts = u_b.ts \wedge u_a.id_r < u_b.id_r)$.

A.2. Convergence theorem

Theorem A.1 (Eventual Convergence). *Given a set of updates $\mathcal{U}$ received by all replicas in $\mathcal{R}$, for any two replicas $r_i, r_j \in \mathcal{R}$, the final states are identical ($S_i = S_j$) regardless of the network arrival order of updates.*

Proof. The proof relies on the properties of determinism, semantic invariance, and total order replay.

Determinism of State Transition: Let $f : S \times u \to S'$ be the state transition function. In BabelRDL, for a given RDL type, f is a deterministic function. Since the core logic is implemented natively in each language ($L_{Go}, L_{Java}, L_{JS}, L_{Rust}$), we ensure f behaves identically across platforms for the same input u.

Preservation of Semantics via CDF: Because all updates u are serialized via Protocol Buffers, the fields $\langle op, arg, id_r, ts\rangle$ are invariant under cross-platform transmission. Formally, for any two languages L_x and L_y: $\forall u \in \mathcal{U} : \text{decode}_{L_y}(\text{encode}_{L_x}(u)) = u$.

As a specific example, this semantic preservation prevents "semantic drift" when a Rust replica processes an update originated in Java.

Existence of a Global Total Order: By Definition A.2, the Lamport clock mechanism ensures that for any finite set of updates $\mathcal{U}$, there exists a unique, globally consistent permutation $P = (u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(|\mathcal{U}|)})$ such that $u_{\pi(k)} \Longrightarrow u_{\pi(k+1)}$ for all $1 \le k < |\mathcal{U}|$.

Log-Based Replay: Each replica r_i maintains its update order $\mathcal{L}_i$ sorted by $\Longrightarrow$. Upon receiving a remote update u_{rem}, the replica updates its log $\mathcal{L}_{new} = \text{sort}(\mathcal{L}_{old} \cup \{u_{rem}\}, \Longrightarrow)$. The state S is computed as the sequential application of operations in the sorted log: $S = f(f(\ldots f(S_0, u_{\pi(1)}), \ldots), u_{\pi(|\mathcal{U}|)})$.

Since the initial state S_0 is constant, the transition function f is deterministic, and the total order $\Longrightarrow$ is unique and universally applied, it follows that $S_i = S_j$ for any two replicas that have processed the same set of updates $\mathcal{U}$. □

Appendix B. Consistency preservation with plug-in extensibility

To address the correctness of BabelRDL's extensibility model, we formally demonstrate that integrating functionalities, such as Logging, Undo, and Rollback as plug-ins, does not compromise Strong Eventual Consistency (SEC).

B.1. Taxonomy of plug-in interactions

Let S denote the replicated state and $\mathcal{L}$ denote the set of all updates (the causal log). We define two primary interaction models:

Definition B.1 (Observer Model $\mathcal{P}_{obs}$). An observer plug-in (e.g., Logging) is a functional mapping $\mathcal{P}_{obs} : S \to \Omega$, where Ω is an external output space. This model is strictly side-effect-free with respect to the RDL state.

Definition B.2 (Compensatory Model $\mathcal{P}_{comp}$). A compensatory plug-in (e.g., Undo, Rollback) is a generator of corrective updates. Given an event trigger ϵ, the plug-in executes a function $g : (\mathcal{L}, \epsilon) \to u_c$, where u_c is a new update designed to offset or reset state transitions.

B.2. Theorem: Non-interference of extensibility

Theorem B.1. *The integration of plug-ins $\mathcal{P}_{obs}$ and $\mathcal{P}_{comp}$ into a BabelRDL replica does not violate the Eventual Convergence property.*

Proof. We analyze the impact of each interaction model on the replica invariants:

Memory and Logical Isolation: BabelRDL utilizes a socket-based IPC mechanism for plug-in communication, enforcing a process-level boundary. For $\mathcal{P}_{obs}$, this ensures the transition function $f(S, u)$ is executed in isolation. Because the plug-in possesses no write-access to the RDL's internal memory, S remains invariant under observation.

Forward-Only Log Progression: For corrective operations in $\mathcal{P}_{comp}$, the system preserves the integrity of the causal history by avoiding destructive edits to $\mathcal{L}$, which would otherwise invalidate the Lamport Timestamp sequence. Specifically, an *Undo* is realized as a compensating update u_{undo} that negates a prior effect, while a *Rollback* is implemented as a state-reset update u_{roll} that instructs the deterministic state machine to return to a prior state S_{ts}. In both cases, $\mathcal{L}$ remains an append-only sequence, preserving the foundational ordering invariants.

Uniform Propagation and Ordering: Every compensatory update u_c is treated as a native RDL operation. It is assigned a unique Lamport Timestamp $ts_c > \max(\mathcal{L})$ and propagated via the CDF to all replicas. The total order $\Longrightarrow$ (Definition A.2) ensures u_c is applied at the identical logical point across all heterogeneous replicas ($L_{Go}, L_{Java}, L_{JS}, L_{Rust}$). Because the augmented update set $\mathcal{U} \cup \{u_c\}$ is processed by a deterministic transition function f over a globally unique permutation, all replicas necessarily reach an identical final state S_{final}. □

References

[1] M. Eischer, T. Distler, Resilient cloud-based replication with low latency, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 14–28.
[2] A.D. Fekete, K. Ramamritham, Consistency models for replicated data, in: Replication: Theory and Practice, Springer, 2010, pp. 1–17.
[3] S. Burckhardt, A. Gotsman, H. Yang, M. Zawirski, Replicated data types: specification, verification, optimality, ACM Sigplan Not. 49 (1) (2014) 271–284.
[4] K. Kontogiannis, P. Linos, K. Wong, Comprehension and maintenance of large-scale multi-language software applications, in: 2006 22nd IEEE International Conference on Software Maintenance, IEEE, 2006, pp. 497–500.
[5] K. De Porre, F. Myter, C. Scholliers, E.G. Boix, CScript: a distributed programming language for building mixed-consistency applications, J Parallel Distrib. Comput. 144 (2020) 109–123.

[6] R. Bisiani, A. Forin, Architectural support for multilanguage parallel programming on heterogeneous systems, ACM SIGPLAN Not. 22 (10) (1987) 21–30.
[7] W. Li, A. Marino, H. Yang, N. Meng, L. Li, H. Cai, How are multilingual systems constructed: characterizing language use and selection in open-source multilingual software, ACM Trans. Software Eng. Method. 33 (3) (2024) 1–46.
[8] A. Métaireau, A comparison of JS CRDTs, 2024. https://blog.notmyidea.org/a-comparison-of-javascript-crdts.html.
[9] Z. Chen, Introduction to Loro's Rich Text CRDT, 2023. https://loro.dev/blog/crdt-richtext.
[10] A. Turcotte, E. Arteca, G. Richards, Reasoning about foreign function interfaces without modelling the foreign language, in: 33rd European Conference on Object-Oriented Programming (ECOOP 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
[11] D.M. Lyons, S.B. Zahra, T.M. Marshall, Towards Lakosian multilingual software design principles, (2019) arXiv:1906.08351.
[12] N. Saquib, C. Krintz, R. Wolski, Log-based CRDT for edge applications, in: 2022 Thinspace IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2022, pp. 126–137.
[13] M. Kleppmann, Making CRDTs byzantine fault tolerant, in: Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data, 2022, pp. 8–15.
[14] The Editors of Encyclopaedia Britannica, Tower of Babel, 2022. https://www.britannica.com/topic/Tower-of-Babel.
[15] S. Vinoski, CORBA: integrating diverse applications within distributed heterogeneous environments, IEEE Commun. Mag. 35 (2) (1997) 46–55.
[16] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[17] D. Borthakur, The hadoop distributed file system: architecture and design, Hadoop Proj. Website 11 (2007) (2007) 21.
[18] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10), 2010.
[19] L. Richardson, S. Ruby, RESTful Web Services, " O'Reilly Media, Inc.", 2008.
[20] I. Jacobi, A. Radul, A RESTful messaging system for asynchronous distributed processing, in: Proceedings of the First International Workshop on RESTful Design, 2010, pp. 46–53.
[21] W. Tansey, E. Tilevich, Efficient automated marshaling of C + + data structures for MPI applications, in: 2008 Thinspace IEEE International Symposium on Parallel and Distributed Processing, IEEE, 2008, pp. 1–12.
[22] J. Li, Monitoring and characterization of component-based systems with global causality capture, in: 23rd International Conference on Distributed Computing Systems, 2003. Proceedings., IEEE, 2003, pp. 422–431.
[23] C. Piñeiro, R. Martínez-Castaño, J.C. Pichel, Ignis: an efficient and scalable multi-language big data framework, Future Gener. Comput. Syst. 105 (2020) 705–716.
[24] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, H. Mössenböck, Cross-language interoperability in a multi-language runtime, ACM Trans. Program. Lang. Syst. (TOPLAS) 40 (2) (2018) 1–43.
[25] Y. Mao, Z. Liu, H.-A. Jacobsen, Reversible conflict-free replicated data types, in: Proceedings of the 23rd ACM/IFIP International Middleware Conference, 2022, pp. 295–307.
[26] M. Kleppmann, A.R. Beresford, A conflict-free replicated JSON datatype, IEEE Trans. Parallel Distrib. Syst. 28 (10) (2017) 2733–2746.
[27] K. De Porre, C. Ferreira, N. Preguiça, E. Gonzalez Boix, ECROs: building global scale systems from sequential code, Proc. ACM Program. Lang. 5 (OOPSLA) (2021) 1–30.
[28] G. Kaki, S. Priya, K.C. Sivaramakrishnan, S. Jagannathan, mergeable replicated data types, Proc. ACM Program. Lang. 3 (OOPSLA) (2019) 1–29.
[29] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS 2011, Springer LNCS volume 6976, 2011, pp. 386–400. https://doi.org/10.1007/978-3-642-24550-3_29
[30] R. Wolfinger, Plug-in architecture and design guidelines for customizable enterprise applications, in: Companion to the 23Rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, 2008, pp. 893–894.
[31] D. Notkin, W.G. Griswold, Extension and software development, in: ICSE, 88, 1988, pp. 274–283.
[32] D. Birsan, On plug-ins and extensible architectures: extensible application architectures such as eclipse offer many advantages, but one must be careful to avoid "plug-in hell.", Queue 3 (2) (2005) 40–46.
[33] J. Mayer, I. Melzer, F. Schweiggert, Lightweight plug-in-based application development, in: Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002 Erfurt, Germany, October 7–10, 2002 Revised Papers 4, Springer, 2003, pp. 87–102.
[34] N. Cosentino, Plugin architecture design pattern - a beginner's guide to modularity, 2023. https://www.devleader.ca/2023/09/07/plugin-architecture-design-pattern-a-beginners-guide-to-modularity/.
[35] M. Fisher, J. Ellis, J. Ellis, J. Bruce, JDBC API Tutorial and Reference, Addison-Wesley Professional, 2003.
[36] A. Stefik, S. Hanenberg, The programming language wars: questions and responsibilities for the programming language community, in: Proceedings of the 2014 Thinspace ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, 2014, pp. 283–299.
[37] J. Armstrong, Making reliable distributed systems in the presence of software errors, Ph.D. thesis, 2003.
[38] Y.-W. Kwon, E. Tilevich, T. Apiwattanapong, DR-OSGi: hardening distributed components with network volatility resiliency, in: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2009, pp. 373–392.
[39] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (1978) 558–565.
[40] D. Castro, Protocol Buffers, 2001. https://protobuf.dev/.
[41] L.I. Hatledal, A. Styve, G. Hovland, H. Zhang, A language and platform independent co-simulation framework based on the functional mock-up interface, IEEE Access 7 (2019) 109328–109339.
[42] W. Schwitzer, V. Popa, Using protocol buffers for resource-constrained distributed embedded systems, Technische Universitaet Muenchen, Tech. Rep. (2011).
[43] P.S. Almeida, C. Baquero, Scalable eventually consistent counters over unreliable networks, Distrib. Comput. 32 (1) (2019) 69–89.
[44] S. Laddad, C. Power, M. Milano, A. Cheung, N. Crooks, J.M. Hellerstein, Keep CALM and CRDT On, (2022) arXiv:2210.12605.
[45] V.B. de Sousa, Key-CRDT stores, Master's thesis, Universidade NOVA de Lisboa (Portugal), 2012.
[46] D.B. .T. Contributors, Protocol Buffers implementation for the Rust Language: PROST!, 2022. https://github.com/tokio-rs/prost.
[47] D. Symonds, J. Tsai, Go support for protocol buffers, 2020. https://github.com/golang/protobuf.
[48] M. Buus, Protocol Buffers - npm, 2014. https://www.npmjs.com/package/protocol-buffers.
[49] G.P.B. Developers, Protocol Buffers Java 3.21.12, 2022. https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java/3.21.12.
[50] E. Babaians, M. Tamiz, Y. Sarfi, A. Mogoei, E. Mehrabi, Ros2unity3d; high-performance plugin to interface ros with unity3d engine, in: 2018 9th Conference on Artificial Intelligence and Robotics and 2nd Asia-Pacific International Symposium, IEEE, 2018, pp. 59–64.
[51] A.B. Beck, N.A. Andersen, J.C. Andersen, O. Ravn, Mobotware–a plug-in based framework for mobile robots, IFAC Proc. Vol. 43 (16) (2010) 127–132.
[52] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, R. West, A quality-of-service enhanced socket API in GNU/linux, in: Proceedings of the 4th Real-Time Linux Workshop, Boston, Massachusetts, 2002, p. 31.
[53] A. Raja, S. Kaladiya, S. Bhatia, X. Peng, Uber's Next-Gen push platform on gRPC, 2022. https://www.uber.com/blog/ubers-next-gen-push-platform-on-grpc/.
[54] A. Unnikrishnan, What coding language is used in Salesforce?, 2024.
[55] M. Hu, Q. Zhao, Y. Zhang, Y. Xiong, Cross-language call graph construction supporting different host languages, in: 2023 Thinspace IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2023, pp. 155–166.
[56] M. Yuan, Cgroups - ArchWiki, 2010. https://wiki.archlinux.org/title/Cgroups.
[57] J.A. Donenfeld, WireGuard: next generation kernel network tunnel, in: NDSS, 2017, pp. 1–12.
[58] M. Ladany, Go-CRDT-Package, 2024. https://github.com/mattladany/crdt.
[59] A. van der Linde, T. Vale, J. Leitão, Legion, 2017. https://github.com/albertlinde/Legion.
[60] A. Rijo, T-CRDT Library, 2018. https://github.com/AndreRijo/T-CRDTs.
[61] Automerge Contributors, Automerge: welcome to automerge documentation, 2025. https://automerge.org/docs/hello/.
[62] C. Irwin, O. Kovalov, Automerge-Go, 2022. https://github.com/automerge/automerge-go.
[63] A. Good, J. Arhart, Automerge for Java, 2023. https://github.com/automerge/automerge-java.
[64] O. Henry, A. Jeffery, Automerge, 2021. https://github.com/automerge/automerge.
[65] P. Sestoft, Runtime code generation with JVM and CLR, Unpublished. Available at http://www.dina.dk/sestoft/publications.HTML (2002).
[66] E. Morozoff, Using a line-of-code metric to understand software rework, IEEE Softw. 27 (1) (2009) 72–77.
[67] M. Zhang, N. Baddoo, Performance comparison of software complexity metrics in an open source project, in: European Conference on Software Process Improvement, Springer, 2007, pp. 160–174.
[68] G.K. Gill, C.F. Kemerer, Cyclomatic complexity density and software maintenance productivity, IEEE Trans. Softw. Eng. 17 (12) (1991) 1284–1288.
[69] T. Hariprasad, G. Vidhyagaran, K. Seenu, C. Thirumalai, Software complexity analysis using Halstead metrics, in: 2017 International Conference on Trends in Electronics and Informatics (ICEI), IEEE, 2017, pp. 1109–1113.
[70] M. Alfadel, A. Kobilica, J. Hassine, Evaluation of halstead and cyclomatic complexity metrics in measuring defect density, in: 2017 9th IEEE-GCC Conference and Exhibition (GCCCE), IEEE, 2017, pp. 1–9.
[71] Z. Li, J. Wang, M. Sun, J.C.S. Lui, Detecting cross-language memory management issues in Rust, in: European Symposium on Research in Computer Security, Springer, 2022, pp. 680–700.
[72] S. Guo, R. Dhamankar, L. Stewart, Distributedlog: a high performance replicated log service, in: 2017 Thinspace IEEE 33rd International Conference on Data Engineering (ICDE), IEEE, 2017, pp. 1183–1194.
[73] M. Vassor, J.-B. Stefani, Checkpoint/rollback vs causally-consistent reversibility, in: International Conference on Reversible Computation, Springer, 2018, pp. 286–303.
[74] W. Yu, V. Elvinger, C.-L. Ignat, A generic undo support for state-based CRDTs, in: OPODIS 2019-Proceedings of 23rd International Conference on Principles of Distributed Systems, 2019.
[75] S. Sarkar, A.C. Kak, G.M. Rama, Metrics for measuring the quality of modularization of large-scale object-oriented software, IEEE Trans. Softw. Eng. 34 (5) (2008) 700–720.

[76] M. Hitz, B. Montazeri, Measuring Coupling and Cohesion in Object-Oriented Systems, na, 1995.
[77] F.B. Abreu, R. Esteves, M. Goulão, The design of eiffel programs: quantitative evaluation using the MOOD metrics, in: Proceedings of TOOLS'96, Citeseer, 1996.
[78] P. Ferlet, Understand how to use C libraries in Go with cgo, 2023. https://dev.to/metal3d/understand-how-to-use-c-libraries-in-go-with-cgo-3dbn.
[79] D. Panov, Goja: ECMAScript 5.1(+) implementation in Go, 2016. https://github.com/dop251/goja.
[80] T.G. Authors, Go package syscall/js, 2018. https://pkg.go.dev/syscall/js.
[81] K. Sharan, Java APIs for Nashorn, in: Scripting in Java: Integrating with Groovy and JavaScript, Springer, 2014, pp. 319–341.
[82] G. Zheng, L. Gao, L. Huang, J. Guan, Application binary interface (ABI), in: Ethereum Smart Contract Development in Solidity, Springer, 2020, pp. 139–158.
[83] H. Chatham, D. Carney, Spxg, D. Crawford, M. Akari, The 'wasm-bindgen' guide, 2018. https://wasm-bindgen.github.io/wasm-bindgen/.
[84] J. Siegmund, N. Siegmund, S. Apel, Views on internal and external validity in empirical software engineering, in: 2015 Thinspace IEEE/ACM 37th IEEE International Conference on Software Engineering, 1, IEEE, 2015, pp. 9–19.
[85] P. Ralph, E. Tempero, Construct validity in software engineering research and software metrics, in: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, 2018, pp. 13–23.
[86] M. Maheswaran, R. Wenger, R. Olaniyan, S. Memon, O. Fadahunsi, R. Echomgbe, A language for programming edge clouds for next generation iot applications, (2019) arXiv:1906.09962.
[87] Z. Li, X. Peng, L. Chao, Z. Xu, EveryLite: a lightweight scripting language for micro tasks in IoT systems, in: 2018 Thinspace IEEE/ACM Symposium on Edge Computing (SEC), IEEE, 2018, pp. 381–386.
[88] H. Yang, Y. Nong, S. Wang, H. Cai, Multi-language software development: issues, challenges, and solutions, IEEE Trans. Softw. Eng. 50 (3) (2024) 512–533.
[89] M. Grichi, Towards Understanding Modern Multi-Language Software Systems, Ecole Polytechnique, Montreal (Canada), 2020.
[90] S. Goel, R. Buyya, Data replication strategies in wide-area distributed systems, in: Enterprise Service Computing: From Concept to Deployment, IGI Global, 2007, pp. 211–241.
[91] E. Cecchet, G. Candea, A. Ailamaki, Middleware-based database replication: the gaps between theory and practice, in: Proceedings of the 2008 Thinspace ACM SIGMOD International Conference on Management of Data, 2008, pp. 739–752.
[92] D. Chisnall, The challenge of cross-language interoperability, Commun. ACM 56 (12) (2013) 50–56.
[93] S. Kell, C. Irwin, Virtual machines should be invisible, in: Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11, 2011, pp. 289–296.
[94] J. Reppy, C. Song, Application-specific foreign-interface generation, in: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, 2006, pp. 49–58.
[95] M. Hirzel, R. Grimm, Jeannie: granting java native interface developers their wishes, ACM Sigplan Not. 42 (10) (2007) 19–38.
[96] A. Stratikopoulos, F. Blanaru, J. Fumero, M. Xekalaki, O. Papadakis, C. Kotselidis, Cross-Language interoperability of heterogeneous code, in: Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming, 2023, pp. 17–21.
[97] P. Nicolaescu, K. Jahns, M. Derntl, R. Klamma, Yjs: a framework for near real-time p2p shared editing on arbitrary data types, in: International Conference on Web Engineering, Springer, 2015, pp. 675–678.
[98] A.I. Pavlov, A.B. Stolbov, A.A. Lempert, Towards extensibility features of knowledge-based systems development platform, in: 4th Scientific-Practical Workshop Information Technologies: Algorithms, Models, Systems, Bychkov, IV, Karastoyanov, D., Eds, 2021, pp. 87–94.
[99] Q. Zhu, Y. Yang, M. Natale, E. Scholte, A. Sangiovanni-Vincentelli, Optimizing the software architecture for extensibility in hard real-time distributed systems, IEEE Trans. Ind. Inf. 6 (4) (2010) 621–636.
[100] M. Zenger, Evolving software with extensible modules, in: Proceedings of the International Workshop on Unanticipated Software Evolution, 2002.
[101] J. Grundy, J. Hosking, Engineering plug-in software components to support collaborative work, Softw. Pract. Exp. 32 (10) (2002) 983–1013.
[102] J. Campos, A. Riboira, A. Perez, R. Abreu, Gzoltar: an eclipse plug-in for testing and debugging, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 378–381.
[103] M. Greiler, H.-G. Gross, A. Van Deursen, Understanding plug-in test suites from an extensibility perspective, in: 2010 17th Working Conference on Reverse Engineering, IEEE, 2010, pp. 67–76.
[104] A. Charfi, T. Dinkelaker, M. Mezini, A plug-in architecture for self-adaptive web service compositions, in: 2009 Thinspace IEEE International Conference on Web Services, IEEE, 2009, pp. 35–42.

Provakar Mondal is a PhD candidate in Computer Science at Virginia Tech, advised by Dr. Eli Tilevich. His research lies at the intersection of distributed systems, software engineering, and system support for machine learning. He builds practical systems that enhance the reliability, interoperability, and performance of replicated data systems and domain-specific AI models.

Eli Tilevich is a Professor in the Dept. of Computer Science at Virginia Tech, where he leads the Software Innovations lab. Tilevich's research interests lie on the Systems end of Software Engineering, with a particular emphasis on distributed systems, mobile/IoT applications, middleware, automated program transformation, energy efficiency, privacy & security, and CS education. He has published over 120 refereed research papers on these subjects. Tilevich has earned a B.A.summa cum laude in Computer Science/Math from Pace University, an M.S. in Information Systems from NYU, and a Ph.D. in Computer Science from Georgia Tech.