Type-Safe Transactional Monadic IO in Java 17

Leonid Meshcheriakov

Innopolis University, Innopolis, Tatarstan Republic, Russia

Abstract. Most popular Java approaches rely on imperative constructs, exception handling with try-catch blocks, and transaction frameworks. Many of them are thread-bound, which create challenges for asynchronous workflows and system composability. Inspired by functional languages like Haskell and Scala, we adapt the IO monad abstraction to pure Java using modern language features including sealed interfaces and records and add support for transaction management in this abstraction. Our implementation provides a declarative, type-safe mechanism for sequencing computations with side effects through monadic composition, structured exception flow control, and transaction management powered by lifecycle hooks. The library also supports inner transactions, the Saga pattern, and asynchronous computations. The final implementation show that functional concepts such as the IO monad can be seamlessly integrated into Java 17 without custom runtime or syntax, making code more composable with preserved type safety.

Keywords: Java · Side Effects · Functional Programming · IO Monad · Transactions.

1 Introduction

In modern software engineering, reliable management of side effects and transactions is a fundamental requirement for building scalable and maintainable systems. Java, while widely used in enterprise systems, relies primarily on imperative constructs and framework-level mechanisms for handling side effects and transactions, such as exceptions and annotations. Moreover, widely used Java frameworks for transactions, such as Spring Transaction Management or Java Transaction API, are thread-bound, meaning that a transaction initiated in one thread cannot be safely continued, committed or rolled back in another. This limitation makes it hard to design asynchronous or parallel transactional workflows.

In contrast, functional programming languages such as Haskell and Scala hava addressed this problem through the abstraction of the IO monad, which provides a declarative and type-safe mechanism for sequencing computations with possible side effects, and the *Handle* pattern, where resource and context management lies outside the computation. These approaches allow developers to model side effects explicitly, improving reasoning about program behavior, composability, and testability.

Despite the proven advantages of monadic abstractions in functional programming, there is no native or standardized mechanism in Java for encapsulating and composing side effects in a type-safe and purely functional manner. Although some researchers have attempted to introduce algebraic effects and handlers in Java [1,2], these solutions typically rely on non-standard language extensions, complex syntax, or advanced functional abstractions, which increase the barrier to entry and limit their usability in practical applications.

This research explores whether the idea of the IO monad, as used in functional languages, can be effectively adapted to Java 17 to create a type-safe, composable, and transactional model for managing side effects.

The goal of the research is to design and implement a library for Java 17 that enables the representation of computations as declarative IO actions with explicit and composable error handling and support for asynchronous execution and transaction management. Moreover, the library should be user-friendly for the average Java developer; in other words, it should have the lowest possible barrier to entry.

The library is implemented using modern features from Java 17, such as sealed interfaces and records, which help model data in a type-safe and concise way. The design follows the principles of IO monads from Haskell and Scala but extends them with built-in transactional support.

To evaluate the library, unit tests will be used to check compliance with monad laws and correctness of transactional workflows. Additionally, a test program will be implemented to validate the functionality, correctness and performance of the library.

2 Literature Review

2.1 The Limitations of Exception and try-catch Mechanisms

Exception handling using try-catch blocks has been a key component of error management in object-oriented programming languages such as Java. However, many studies show that, although this mechanism aims to improve reliability and maintainability, in practice it often leads to hidden complexity and problems with modular design. The work by Miller and Tripathi [3] was one of the first to analyse this problem, in which the authors state that traditional exception handling conflicts with object-oriented principles such as abstraction, inheritance, and encapsulation. Exceptions are propagated implicitly, and they can be handled far from the point where they were thrown. This makes control flow less transparent and introduces dependencies between modules that are not visible in the program structure. As a result, it becomes harder to reason about system behavior or to modify and reuse components safely. [3] concludes that exception handling, although designed to improve fault tolerance, can in fact reduce robustness if it is not designed properly.

Empirical evidence supports these theoretical claims, for example, the authors of [4] found that in large software product lines exceptions flow frequently goes beyond the scope of individual features and modules in ways that were not originally intended. Their study discovered that many exceptions are thrown by variable components, but caught in core parts of the system or sometimes not caught at all. This leads to unplanned exception propagation and unpredictable program behavior. Handlers are often generic, located far from the original source of the error, and therefore unable to recover properly. The authors conclude that try-catch mechanisms, when used in large modular systems, do not ensure the composability or predictability required for reliable software design. Moreover, in a large empirical study of Eclipse and Apache Tomcat projects, the authors analyzed 220 exception handling bugs and surveyed 154 professional developers [5]. They found that exception handling is often ignored in practice: only 27% of organizations reported having exception handling policies and standards, and 70% had no specific tests for exception handling code. Common problematic practices were overly generic error handlers or empty `catch` blocks, which made systems more fragile and harder to debug.

2.2 Monads and Side Effects

Studies [6,7] show that functional concepts, especially monads, can be effectively integrated into object-oriented and imperative languages. The authors of [6] focus on representing effectful computations with type-level monadic structures that make side effects explicit. Their library translates imperative constructs into typed bind chains, ensuring that each effect of `Option`, `State`, `Writer`, and `List` is safely composed. In [7] the Funk library was implemented in C#, introducing types such as `Maybe`, `OneOf`, and `Exc` to express optionality, branching, and exceptions as explicit monadic values. Both approaches prove that effect-safe functional abstractions can coexist with object-oriented design. This integration can improve code composability, clarity, and reliability and reduce boilerplate and failures by forcing developers to handle side effects properly.

Simon Peyton Jones [8] explains how Haskell deals with real-world side effects such as IO, concurrency, exception handling, and foreign language calls, while keeping the language pure and functional. The author shows that monads make it possible to describe side effects as typed values, so that programs stay predictable and composable. The IO monad is used to control execution order and isolate side effects inside a safe type structure. The paper also shows how monads make concurrent and exception-handling code easier to reason about.

In [9], the author presents the theoretical idea that exception handling can be seen as a special case of monadic binding, not as separate control-flow construct. He shows that the usual `catch` operation works in the same way as the monadic `bind` when it is used with an error monad. This means that exceptions can be handled as first-class values inside a typed computational context.

3 Design

To be more user-friendly for the average Java developer, the Java standard API and `Mono` from Project Reactor [10]—one of the widely used Java libraries among those that adapt functional programming concepts—were considered while designing API.

3.1 Constraints of Java Type System

Java does not support higher-kinded types, but they can be introduced in type-safe way [12]. However, this approach makes the library more complex and harder to understand, which increases the barrier to entry. Moreover, this approach leads to boilerplate code and a loss of type information: if an operation is defined generically over all monads, its result will be only generic kind without type-specific methods, such as error recovery from the IO. In this work, the focus is limited to the implementation of the IO monad, because a separate library for a general Monad abstraction introduces more drawbacks than benefits.

3.2 Exceptions as IO Error

According to Java language standards [13], developers should throw `Exception` and its subclasses only to indicate that a problem has occurred. Therefore, we can consider an exception as an error state of our monad, that can be recovered from. If an exception is thrown inside a transformation function, it will be also treated as recoverable error state of the IO monad, ensuring that exception do not propagate outside.

3.3 Support of Asynchronous Computation

Modern applications often require asynchronous computations to handle I/O operations, external API calls, and concurrent tasks efficiently without blocking threads. To support such workflows, the library should rely on `CompletableFuture`—the standard Java API for asynchronous computations—so that the developer can seamlessly integrate existing libraries. This can be achieved by representing the result of each computation step as a `CompletableFuture`, to which subsequent operations are then composed. Additionally, to give developers control over thread management, it is important to provide the ability to explicitly specify which `Executor` should be used to run asynchronous computations when executing an `IO`.

3.4 Debugging Support

Since `IO` computations are lazy and asynchronous, standard Java stack traces become uninformative—they capture only the point where the computation was executed, such as a thread pool worker, rather than where the `IO` was actually constructed. To solve this problem, the library should capture stack traces at each `IO` step, so that the developer can trace the entire composition chain and identify which specific operation failed and how it was built.

3.5 Lack of Syntactic Sugar

Haskell has `do`-blocks, just like Scala `for`-comprehension, which make code more concise and less overloaded, when working with monads. Basically, these constructs are syntactic sugar over the `bind` operator. This kind of functionality is important for writing concise code. Since the library uses the standard Java without syntax changes, this problem can be solved by creating static methods.

Also, while designing the library, boilerplate code was observed when specifying the `IO` type with its parameters—context, error and result. Not only did the type itself become overly long and impractical to use, but it also needed to be repeated in multiple locations. Although this problem can be avoided by using the `var` keyword instead of a type, so it will be inferred, but it can be applied only to variables.

This problem can be solved in Haskell and Scala through type aliasing mechanism, which is not available in Java. However, it can be implemented in at least two ways. First, the developer defines a type alias via an interface, for example: `MyIO<R> extends IO<Nothing, Exception, R>`, and uses it in the code. Second, either the code is transformed at compile time by substituting `IO` instead of these user-defined interfaces, or the library generates classes implementing these interfaces at runtime. The first approach provides better runtime performance, but its implementation is significantly more complex than the second approach. This complexity arises from the requirement for advanced static analysis and code transformation. Moreover, it is not clear whether it will provide a significant performance improvement compared to the second solution. Hence, the second approach was chosen.

3.6 Transactions

Transaction handling should be designed so that transaction handlers, which configure environment, are defined in the same place as the code that requires it, rather than elsewhere. Therefore, the end user does not need to know, which implementation of service is used, to handle transactions correctly, which aligns with the object-oriented principle of inheritance. This approach enables easy combination of different modules with service implementations, reduces complexity and simplifies testing. Additionally, there should be support for isolated transaction for workflows that require it.

This behavior can be achieved by allowing the attachment of transactional hooks to an `IO` instance. The attached hooks should be executed before computation of that particular `IO` starts and at the end of its computation, providing information about whether it completed successfully. To properly isolate one transaction from another, their hooks must be handled separately from each other.

Unfortunately, some systems lack transaction mechanisms, for example, API calls, or cannot rely on them due to various reasons, such as the absence of atomicity, consistency, isolation or durability (ACID) guarantees. For such scenarios, there should be support for the Saga pattern [11]—a transaction pattern that maintains data consistency through a sequence of local transactions. In this pattern, the developer can define a compensating action for each step that will be executed on failure to undo the effects of previously completed steps. This approach ensures consistency even when rollback mechanisms are not available.

3.7 Stack Overflow Problem

The most popular IO implementations in Scala—ZIO [14] and Cats Effect[15]—use a trampolining technique, where computations are executed step by step in a loop instead of recursive function calls, preventing stack overflow. Stack safety of asynchronous computations is ensured by storing the continuation and resuming it later via an event loop rather than through recursive calls.

In practice, Java programs are typically written in an object-oriented and imperative style, where stack overflow rarely occurs and usually happens only when recursion is used incorrectly. Moreover, we need to work with `CompletableFuture` that is not stack-safe, so that the developer can integrate libraries based on it. Due to these factors, this implementation was not designed to be stack-safe.

4 Implementation

We present results of our implementation[1] of the IO monad. This implementation demonstrates the feasibility of combining functional programming concepts with transaction management in a type-safe manner using an object-oriented language. While the core monadic operations and basic transactional management are in place, a number of advanced features are still under development.

4.1 Core Design

The `IO<C, E extends Exception, R>` interface defines a computation that is parameterized by a context type `C`, an exception type `E`, and a result type `R`.

This interface defines three core abstract methods that must be implemented: the `prepare` method simply returns an instance of `IORunnable` that represents the computation, the `getHooks` method returns the list of transactional hooks attached to this instance, and the `getInitializationTrace` method returns the stack trace of this instance initialization, needed for easing debugging. To run the computations, one of the execute methods, such as `tryExecute` or `tryExecuteAsync`, should be used.

```java
public interface IO<C, E extends Exception, R> {
    // ...
    @NotNull
    IORunnable<C, R> prepare(@NotNull C context);

```

[1] The implementation and documentation are available at https://gitlab.com/worldm/functional/io.

```
    @NotNull
    RecList<IOHook<? super C>> getHooks();

    @NotNull
    RecList<IOInitializationTrace> getInitializationTrace();

    @NotNull
    default Try<Exception, R> tryExecute(
            @NotNull C Context
    ) {
        // ...
    }

    @NotNull
    default CompletableFuture<Try<Exception, R>>
        tryExecuteAsync(
            @NotNull C Context
    ) {
        // ...
    }
    // ...
}
```

Besides these methods, the `IO` interface provides a comprehensive set of default methods for monadic composition, exception flow control, the type aliasing mechanism, transaction management, and utility operations.

Also, the library provides several standard implementations such as `IO.success(R result)` and `IO.error(E error)` for pure values, `IO.of(Supplier<R> supplier)` for simple effectful operations, `IO.fromCompletionStage(...)`, and so on.

```
IO<Nothing, Exception, String> io1 = IO.success("abc");
// Java allows type inference via "var" keyword.
// Type of io2 is IO<Object, RuntimeException, Long>
var io2 = IO.of(
        () -> System.currentTimeMillis()
);
```

4.2 Monadic Composition

The implementation satisfies monad laws through the `map` and `flatMap` methods. The `map` method simply transforms results, while the `flatMap` method creates a sequence of `IO` computations. Method signatures ensure compatibility with other context types and allow flexible result type transformations. Moreover, it is impossible to compose `IO` instances with different context types directly, it is required to explicitly change the context through the `mapContext` method. If any of these methods produce an exception inside an user-defined transformation function, it will be caught, and the `IO` computation will be in a recoverable error state.

```java
IO<Nothing, Exception, String> io1 = IO.success("abc");
IO<Nothing, Exception, Integer> io2 = io1.map(
        s -> s.length()
);
// Type inference works even on transformations.
// Type of io3 is IO<Nothing, Exception, Integer>
var io3 = io2.flatMap(
        i -> IO.success(i + 1)
);
IO<Connection, Exception, Integer> io4 = io3.mapContext(
        c -> Nothing.INSTANCE
);
```

4.3 Exception Flow Control

Structured exception flow control is provided through three methods: `recover` converts exceptions into successful results, `flatRecover` allows exception handlers to return an `IO` computation for complex recovery strategies, and `mapError` transforms exception types for adapting to different error contexts. The type system tracks exception types through composition chains, providing compile-time safety.

```java
IO<Nothing, SQLException, String> io1 = IO.error(
        new SQLException("abc")
);
IO<Nothing, SQLException, String> io2 = io1.recover(
        e -> e.getMessage()
);
IO<Nothing, SQLException, String> io3 = io1.flatRecover(
        e -> new Random().nextBoolean()
                ? IO.success(e.getMessage())
                : IO.error(e)
);
IO<Nothing, Exception, String> io4 = io1.mapError(
        e -> (Exception) e
);
```

4.4 Asynchronous Computations

The implementation is built on top of `CompletableFuture` to support asynchronous computations, allowing the wrapping of existing asynchronous code based on the standard Java API as `IO` computations, thus gaining the benefits of this library, such as transaction management. It is worth noting that there is a simple optimization: if an `IO` is executed asynchronously, the entire computation will be performed in the same thread that called the execution until it reaches the first asynchronous computation. Therefore, if an `IO` has no asynchronous operations, the entire computation runs synchronously in the same thread.

```java
IO<ApiContext, Exception, String> io = IO.fromCompletionStage
    (context -> {
    String password = context.getApiPassword();

    CompletableFuture<String> future = apiService.getToken(
        password);

    return future;
});
```

Additionally, the developer can specify the `Executor` that should be used to run asynchronous computations by providing an execution context that implements the `ExecutionContext` interface.

```java
class MyContext implements ExecutionContext {
    private final Executor executor;

    public MyContext(@NotNull Executor executor) {
        this.executor = executor
    }

    @Override
    @Nullable
    public Executor getExecutor() {
        return executor;
    }

    @NotNull
    public ExecutionContext withExecutor(
            @Nullable Executor executor
    ) {
        return new MyContext(executor);
    }
}
```

```java
IO<Object, Exception, String> io = IO.success("abc")
    .flatMap(s -> IO.fromCompletionStage(...))
    .map(o -> o.toString());

Executor executor = Executors.newSingleThreadExecutor();

// Asynchronous computation will use the executor,
// provided to the context
Try<Exception, String> result = io.tryExecute(
        new MyContext(executor)
);
```

4.5 Type Aliasing

The implementation contains type aliasing mechanism that allows the use of domain-specific type definitions via interfaces. The `withTypeAlias` method adapts

IO to the user-defined interface type without losing type-safety, allowing the developer to write concise code. Additionally, the developer can override methods to change the type of the return value to the corresponding type alias. All type constraints work at compile time; hence, the reliability of the type aliasing mechanism is significantly increased.

```java
interface MyIO<R> extends IO<Nothing, Exception, R> {
    @Override
    default <NR> MyIO<NR> map(
            CheckedFunction<
                    ? extends SQLException,
                    ? super R,
                    ? extends NR
            > function
    ) {
        return IO.super.map(function)
                .withTypeAlias(new TypeRef<>() {
                });
    }
}
```

```java
IO<Nothing, Exception, String> io1 = IO.success("abc");
MyIO<String> io2 = io1.withTypeAlias(new TypeRef<>() {});
MyIO<Integer> io3 = io2.map(s -> s.length());
IO<Nothing, Exception, Integer> io4 = io3;
```

4.6 Transaction Management

Transaction lifecycle hooks (onStart, onEnd, onException) attach to IO instances by using the addHook method and are preserved through monadic composition. It is guaranteed that a hook will be considered only once if it correctly defines a key—a combination of objects that uniquely identifies this hook. Moreover, the library ensures correct handling of hooks after IO instance transformations. The current implementation supports single-phase transactions with proper resource management and commit-rollback logic. Additionally, the library supports nested transactions by introducing the isolate method that runs a new transaction independently of the current one.

```java
class SqlHook implements IOHook<Connection> {
    @Override
    public void onStart(@NotNull Connection connection) {
        connection.setAutoCommit(false);
    }

    @Override
    public void onEnd(@NotNull Connection connection) {
        connection.commit();
        connection.close();
    }

    @Override
    public void onException(@NotNull Connection connection) {
        connection.rollback();
        connection.close();
    }

    @NotNull
    public IOHookKey getKey() {
        // Hook does not have any arguments or dependencies,
        // so the key is just the class
        return new IOHookKey(getClass());
    }
}
```

```
IO<Connection, SQLException, Unit> io1 = IO.of(connection ->
   {
    Statement statement = connection.prepareStatement(
        "INSERT INTO users(name, age) VALUES (?, ?)"
    );
    statement.setString(1, "Ivan");
    statement.setInt(2, 10);
    statement.executeUpdate();
    return Unit.INSTANCE;
}).addHook(new SqlHook());

IO<Nothing, Exception, Unit> io2 = IO.success(Unit.INSTANCE)
    .flatMap(unit ->
            io1.isolate(() ->
                    DriverManager.getConnection("...")
            )
    );

Try<Exception, Unit> result = io2.tryExecute(
        Nothing.INSTANCE
);
```

4.7 Saga Pattern

The library provides the Saga pattern for systems, where resources lack proper transactional support. The developer can use the `compensate` method to associate compensating actions with an `IO` instance, so that when a failure occurs, they execute automatically in reverse order, ensuring consistency across the system.

```
Map<Integer, String> storage = new ConcurrentHashMap<>();
AtomicInteger lastId = new AtomicInteger();

IO<Nothing, Exception, String> io = IO.defer(() -> {
    int id = lastId.incrementAndGet();
    return IO.of(() -> {
        String data = "data";
        storage.put(id, data);
        return data;
    }).compensate(IO.of(() -> {
        storage.remove(id);
    }));
});
```

4.8 Debugging Support

In the library, each `IO` instance captures the stack trace of its initialization, maintaining a recursive list of the full transformation history. When an `IO` computation raises an exception, these initialization stack traces are automatically filtered and attached as suppressed exceptions, allowing developers to trace the failure. Moreover, developers can control what should be filtered out by providing an execution context that implements the `InitializationTraceContext` interface.

```
class MyContext implements InitializationTraceContext {
    @Override
    @NotNull
    public IOTraceIgnoreTrie getStackIgnoreTrie() {
        return new IOTraceIgnoreTrie()
                .addStartsWith("java.util.concurrent.");
    }

    @Override
    public int getInitializationTraceDepth() {
        return 20;
    }
}
```

```
IO<Object, Exception, String> io = IO.success("abc")
        .map(s -> s.repeat(2))
        .flatMap(s -> IO.error(new Exception));

// Contains an suppressed exception with information
// about the initialization trace of the IO
Try<Exception, String> result = io.tryExecute(
        new MyContext()
);
```

4.9 Composition Utilities

The library provides several utilities to simplify composition and transformation of `IO` instances, improving code clarity and maintaining type safety.

For-Comprehension The IODo class provides Scala-like for-comprehension syntax to simplify the composition of multiple IO instances. It allows developers to express sequential operations in a declarative style, reducing syntactic overhead while maintaining type safety and monadic semantics.

```java
IO<Nothing, Exception, String> io = IO.success("a");

IO<Nothing, Exception, List<String>> result = IODo.of(
        io,
        s -> IO.success(1),
        (s, i) -> IO.success(Collections.nCopies(i, s))
).yield((s, i, l) -> l);
```

Result Aggregation The IOZip class provides the ability to execute multiple IO computations in sequence and zip their results together into a single value in a type-safe manner. It provides a more concise alternative to IODo when the goal is simply to collect and combine multiple independent results.

```java
IO<Nothing, Exception, String> io1 = IO.success("a"),
IO<Nothing, Exception, Integer> io2 = IO.success(1),
IO<Nothing, Exception, Long> io3 = IO.success(2L),

IO<Nothing, Exception, List<String>> result = IOZip.of(
        io1,
        io2,
        io3
).yield((s, i, l) -> new MyEntity(s, i, l));
```

Monadic Transformations The IOOps class provides static methods for several monadic operations. It includes the Optional monad transformer for concise handling of nullable values, as well as the sequence method that converts a collection of IO instances into a single IO instance that contains a collection of the results.

```java
List<IO<Nothing, Exception, String>> list = List.of(
        // ...
);

IO<Nothing, Exception, List<String>> io1 = IOOps.sequence(
        list
);

IO<Nothing, Exception, Optional<String>> io2 = IO.of(
        Optional.of("a")
);

IO<Nothing, Exception, Optional<String>> io3 =
        IOOps.flatMapOptional(
                io2,
                string -> IO.success(string + "b")
        );
```

4.10 Extension Libraries

To demonstrate the practical applicability and extensibility of the `IO` library, two extension libraries were created: a Java Database Connectivity (JDBC) integration layer—JDBC is the standard Java API for working with databases—and a remote procedure call framework supporting distributed transactions.

io-jdbc This library[2] allows developers to easily wrap actions with database based on JDBC to the `IO` instances with transaction support:

```
record User(int id, @NotNull String name, int age) {

}

String name = "foo";

SqlIO<SqlContext, List<User>> io1 = JdbcQuery.select(
                "SELECT id, age FROM users WHERE name = ?"
        )
        .argument(name)
        .build(new ListOutcome<>(
                (resultSet) ->
                        new User(
                                resultSet.getInt("id"),
                                name,
                                resultSet.getInt("age")
                        )
                )
        );

IO<SqlContext, SQLException, List<User>> io2 = io1;
```

io-rpc This library[3] allows the developer to delegate execution of an `IO` computation to a remote server in a single distributed transaction via any backend, that the user can implement. The library contains an implementation for HTTP and AMQP backends. Distributed transactional behavior is achieved through the following steps:

[2] The implementation and documentation are available at https://gitlab.com/worldm/functional/io-jdbc.

[3] The implementation and documentation are available at https://gitlab.com/worldm/functional/io-rpc.

1. The client sends the first RPC request for this server in transaction A.
2. The server stores the already existing context if the first RPC request in this transaction was sent by this server; otherwise, it creates and stores a new one.
3. The server calls the method with arguments, provided by the client, runs the received `IO` and sends the result to the client.
4. The server asynchronously waits for another RPC request or a request that indicates whether the transaction should be rolled back or committed.
5. If waiting exceeds the timeout, the transaction will be rolled back.

Moreover, multiple parallel distributed transactions on one server instance are supported.

Unfortunately, there is a case that can lead the system to an inconsistent state. If the timeout is exceeded at one of the servers and it then receives a commit request, the request will fail. However, it will not affect commit requests to other servers, because in the current implementation the `IO` library ignores exceptions from transactional hooks. To handle such situations correctly, a two-phase commit protocol must be implemented in the future.

```java
// API module, shared between the server and its clients
public interface HashService {
    @NotNull
    IO<Nothing, Exception, String> hash(
            @NotNull String s
    );
}
```

```java
// Server code
// -- Configuration
ServerRpcIO<Nothing> serverRpcIO = new ServerRpcIO<>(
        new ServerId("my-service"), // Server ID
        () -> Nothing.INSTANCE, // Context factory
        Duration.ofSeconds(5) // Transaction timeout
);

serverRpcIO.addCallHandler(
        HashService.class,
        new Sha256HashService() // Implementation
);

// -- Usage
String rpcJsonFromClient = "...";

RpcClientMessage<?> request = serverRpcIO
        .getObjectMapper()
        .readValue(
                rpcJsonFromClient,
                RpcClientMessage.class
        );
```

```

CompletableFuture<RpcResponseMessage<RpcServerMessage>>
        response = serverRpcIO.handle(request);

// Wait for response, serialize it and send
// to the client
```

```
// Client side
// -- Configuration
RpcBackend rpcBackend = ...;

ClientRpcIO<Nothing> clientRpcIO = new ClientRpcIO(
        rpcBackend,
        Nothing.class // Context class
);

HashService remoteHashServer = clientRpcIO.implement(
        HashService.class
);

// -- Usage
String hashedData = IO.success("data")
        .flatMap(p -> remoteHashServer.hash(p))
        .execute(Nothing.INSTANCE);
```

4.11 Constraints

The current implementation has several constraints that must be considered when using the library.

As discussed in Section 3.7, the implementation is not stack-safe. This design focuses on practical needs: stack overflow rarely happens in typical Java code, and compatibility with the existing standard Java API for asynchronous computations—which itself is not stack-safe—is required, since there is a huge code base built on top of it.

All methods enforce non-null semantics: null values are forbidden in all arguments and results in transformation functions even for the `map` method. This is achieved Developers must use `Optional` types when representing absent values.

Exception handling currently captures only `Exception` and its subclasses, meaning other `Throwable` types such as `Error` are not caught and will propagate unchecked. While this aligns with standard Java practices, where `Error` indicates state where it is impossible to recover, some third-party libraries inappropriately use `Error` subclasses for recoverable failures. To properly support badly designed libraries, we plan to refactor the library to handle all `Throwable` types while maintaining appropriate semantics for unrecoverable errors.

The transaction hook system has a critical limitation in error handling: exceptions thrown by `onEnd` and `onException` hooks are only logged and not handled properly. This creates a problem with consistency when several hooks manage different resources. For example, if a file system and an SQL connection hooks have been initialized, and one fails during commit, the other proceeds independently, potentially creating an inconsistent state. To address this issue, there must be support for two-phase commits to ensure the atomicity principle, which is our primary focus.

To change or define the behavior of the effects, the code should rely on the execution context provided by the developer. However, due to the inheritance principle, developers should declare the most general context possible, without exposing implementation details. For example, if we have a `UserRepository` service with in-memory and database implementations, the context type of the interface cannot be `SqlContext`. Otherwise, the code becomes non-extensible: for instance, it is impossible to create an implementation for MongoDB without using type-unsafe context casts. Therefore, the developer should use parameterized types for the context in all abstractions and implementations to ensure fully type-safe operations. It is worth noting that in Java it is possible to obtain a union type only by using parameterized types. This approach is demonstrated in the code example below.

```java
// Service and context definitions
interface UserRepo<C> {
    @NotNull
    IO<C, Exception, User> getUser(int id);
}

interface SqlContext {
    // ...
}

class PostgresUserRepo implements UserRepo<SqlContext> {
    // ...
}

class MemoryUserRepo implements UserRepo<Object> {
    // ...
}

interface PermissionService<C> {
    @NotNull
    IO<C, Exception, Boolean> isAdmin(int id);
}

interface PermissionContext {
    // ...
}

class UserPermissionService<
        // Union type
        C extends URC & PermissionContext,
        // UserRepo context type
        URC
> implements PermissionService<C> {

    private final UserRepo<URC> repo;

    public UserPermissionService(
            @NotNull UserRepo<URC> repo
    ) {
        this.repo = repo;
    }

    // ...
}
```

```
// Configuration
interface CombinedContext extends
    SqlContext,
    PermissionContext {
}

UserRepo<SqlContext> repository =
        new PostgresUserRepo();

PermissionService<CombinedContext> service =
        new UserPermissionService<>(repository);
```

Unfortunately, there is one problem with contexts: the user can use a final class as a context type definition, which limits the ability to handle effects, since these classes cannot be overridden or extended with interface implementations. Addressing this limitation is a priority for future work.

4.12 Conclusion

In this work, the library for the functional IO monad with transaction support was implemented in Java 17, while preserving all the benefits of this approach, including type-safety, side effects handling, and robust workflow. The current implementation provides a solid foundation, but further work remains to be done to achieve more robust workflow.

References

1. Brachthäuser, J., Schuster, P. & Ostermann, K. Effect handlers for the masses. *Proc. ACM Program. Lang.*. **2** (2018,10), https://doi.org/10.1145/3276481
2. T. Mahler. Jiffy, https://github.com/thma/jiffy, Accessed: 2026-01-11
3. Miller, R. & Tripathi, A. Issues with exception handling in object-oriented systems. *ECOOP'97—Object-Oriented Programming*. pp. 85-103 (1997), https://doi.org/10.1186/2195-1721-1-3

4. Melo, H., Coelho, R., Kulesza, U. & Sena, D. In-depth characterization of exception flows in software product lines: an empirical study. *Journal Of Software Engineering Research And Development*. **1**, 3 (2013,10), https://doi.org/10.1186/2195-1721-1-3
5. Ebert, F., Castor, F. & Serebrenik, A. An exploratory study on exception handling bugs in Java programs. *Journal Of Systems And Software*. **106** pp. 82-101 (2015), https://www.sciencedirect.com/science/article/pii/S0164121215000862
6. Anderlind, J. & Åsberg, M. Monadic Programming in Imperative Languages. (2023), https://hdl.handle.net/20.500.12380/306361
7. Ćerim, H. Extending C# with a Library of Functional Programming Concepts. (Univerzita Karlova, Matematicko-fyzikální fakulta,2020), https://hdl.handle.net/20.500.11956/121344
8. Peyton Jones, S. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. (2002), https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/mark.pdf
9. Malakhovski, J. Exceptionally Monadic Error Handling. (2019), https://arxiv.org/abs/1810.13430
10. VMWare. Project Reactor, https://projectreactor.io/, Accessed: 2026-01-11
11. Garcia-Molina, H. & Salem, K. Sagas. *ACM Sigmod Record*. **16**, 249-259 (1987), https://doi.org/10.1145/38714.38742
12. Higher-Kinded-J Maintainers. Higher-Kinded-J, https://higher-kinded-j.github.io, Accessed: 2026-01-11
13. Oracle Corporation. Throwing Exceptions, https://dev.java/learn/exceptions/throwing/, Accessed: 2026-01-11
14. ZIO Maintainers. ZIO Documentation, https://zio.dev/reference, Accessed: 2026-01-11
15. Typelevel. Cats Effect Documentation, https://typelevel.org/cats-effect/docs/getting-started, Accessed: 2026-01-11