Received: 21 November 2022 | Revised: 2 March 2023 | Accepted: 16 April 2023

DOI: 10.1002/spe.3213

RESEARCH ARTICLE

WILEY

Large-scale characterization of Java streams

Eduardo Rosales | **Matteo Basso** | **Andrea Rosà** | **Walter Binder**

Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland

Correspondence
Eduardo Rosales, Faculty of Informatics, Università della Svizzera Italiana, 6900, Lugano, Switzerland.
Email: rosale@usi.ch

Funding information
The Swiss National Science Foundation, Grant/Award Number: 200020_188688

Abstract

Java streams are receiving the attention of developers targeting the Java virtual machine (JVM) as they ease the development of data-processing logic, while also favoring code extensibility and maintainability through a concise and declarative style based on functional programming. Recent studies aim to shedding light on how Java developers use streams. However, they consider only small sets of applications and mainly apply manual code inspection and static analysis techniques. As a result, the large-scale dynamic analysis of stream processing remains an open research question. In this article, we present the first large-scale empirical study on the use of streams in Java code exercised via unit tests. We present stream-analyzer, a novel dynamic program analysis (DPA) that collects runtime information and key metrics, which enable a fine-grained characterization of sequential and parallel stream processing. We use a fully automatic approach to massively apply our DPA for the analysis of open-source software projects hosted on GitHub. Our findings advance the understanding of the use of Java streams. Both the scale of our analysis and the profiling of dynamic information enable us to confirm with more confidence the outcome highlighted at a smaller scale by related work. Moreover, our study reports the popularity of many features of the Stream API and highlights multiple findings about runtime characteristics unique to streams, while also revealing inefficient stream processing and stream misuses. Finally, we present implications of our findings for developers of the Stream API, tool builders and researchers, and educators.

KEYWORDS

code repositories, dynamic program analysis, empirical studies, GitHub, Java streams, Java virtual machine

1 | INTRODUCTION

Since Java 8, the *Stream API*[1] stands out at supporting data processing on the Java virtual machine (JVM), leveraging an intuitive and declarative style based on functional programming.[2] The Stream API offers two key abstractions. First, the *stream*, a view of a sequence of elements made available by a data source. Second, the stream *pipeline*, a sequence

Abbreviations: DPA, dynamic program analysis; JVM, Java virtual machine.

of *operations* that are applied to the elements in the stream upon execution. The elements in a stream can come from multiple data sources, including collections, arrays, generators, and files.[3] The operations in the pipeline allow multiple actions, typically the manipulation of collections to perform *MapReduce*[4] data transformations (e.g., `map`, `filter`, `reduce`).

Streams are getting the attention of developers targeting the JVM, mainly because they are versatile and can ease writing clear and concise data-processing logic.[5-7] Indeed, streams can be used to improve software design by leveraging the extensibility and maintainability favored by functional-programming styles.[8] Another key feature of streams is that they can be executed in parallel just by calling a single operation. As a result, streams can take advantage of modern multicores, potentially achieving performance gains without the need for writing parallel code.

To the best of our knowledge, only three recent studies have analyzed the use of streams. Tanaka et al.[9] mine 100 software repositories to study the use of functional idioms in Java applications. The study focuses on *lambda expressions*,[10] streams, and the `java.util.Optional` class.[11] Khatchadourian et al.[6] examine 34 Java projects to study the use of streams in Java. The work of Khatchadourian et al. represents the first attempt to specifically discover use cases of the Stream API. Finally, Nostas et al.[7] study the use of streams in 610 projects. The work is a partial replication of the study of Khatchadourian et al. considering a larger number of projects and validating the previously obtained results. The aforementioned work considers only a small number of projects and apply mostly manual code inspection and static analysis techniques, letting the dynamic analysis of a representative number of software projects an open research pathway.

This article aims at filling this gap by conducting a large-scale empirical study on the use of streams. Our work has multiple goals. We aim at giving feedback to the community developing the Java class library. In particular, we report the most used features of the Stream API, an information that could be considered when prioritizing future extensions to the API. Moreover, we aim at providing suggestions that may help improve the API according to our observations on how developers are using streams. Another goal is to provide information that can help tool builders and researchers identify potential gaps and understand the kind of support and functionality required to guide Java developers in making better decisions while using streams. Finally, our study aims at helping educators who train Java developers identify little exploited features of the Stream API that can be emphasized in learning processes, such that practitioners are better aware of code patterns enabling more efficient data processing in Java.

Our work faces several challenges. Our analysis targets stream code exercised by unit tests, which is available in a large set of open-source software projects. Differently from related studies,[6,7,9] we use a fully automated approach avoiding manual intervention and specifically target runtime metrics that enable a finer-grained characterization of stream processing. This imposes challenges in carefully selecting a set of dynamic information and metrics suitable to be collected in the wild and in developing an *instrumentation*[*] accurately tracking the key features of the Stream API. Moreover, the information collected should enable data analyses revealing common practices in the use and misuse of streams by Java developers.

Our work makes the following contributions. We present, Stream-Analyzer, a new *dynamic program analysis* (DPA) targeting a relevant selection of metrics that allow characterizing sequential and parallel stream processing on the JVM. Stream-Analyzer builds upon DiSL,[13-15] a framework for the JVM to perform dynamic analysis via bytecode instrumentation. DiSL has been successfully used to build many DPAs[16-19] targeting the Java applications. Moreover, the use of DiSL enables the complete instrumentation of the Java class library, which is crucial for the accurate detection of all streams used in a Java application. Stream-Analyzer is also designed to run on top of *NAB*,[20] a distributed infrastructure allowing the execution of custom analyses on code exercised via unit tests and available in large code-hosting facilities. We use a fully automatic approach that avoids manual intervention and which eases a safe, containerized, and distributed deployment of the analysis. In comparison to related work addressing the analysis and optimization of streams,[21-27] we introduce a profiler able to detect all forms of stream creation and execution available in the Stream API. Moreover, our study is the first to identify and profile multiple runtime metrics that are both specific to streams and suitable to be massively collected in the wild.

We use Stream-Analyzer to conduct the first large-scale empirical study on the use of the Stream API. Our work considers software projects publicly available in GitHub,[28] which were last updated between January 1, 2020 and April 30, 2022. We use a fully automatic approach to dynamically analyze the stream code exercised via unit tests available in these projects. Compared to related work studying the use of streams by Java developers,[6,7,9] our study analyzes the largest

[*]Program *instrumentation* is the process of inserting additional code into a target application with the purpose of collecting metrics at runtime.[12]

number of streams specifically used in application code. Our findings confirm the observations that related work made at a smaller scale. Moreover, our work reports for the first time the popularity of many features of the Stream API, analyzes several carefully selected metrics describing the runtime behavior of streams, and reveals inefficient stream processing and stream misuses currently present in publicly available software projects. Unlike related work[6] reporting fixed stream misuses by manually checking the history of git commits, we detect inefficient stream processing and stream misuses that are currently present in several open-source software projects.

This work significantly extends our previous work on characterizing Java streams at a large scale.[29] We improved our methodology enabling large-scale characterization of stream processing (Section 4). Our new methodology significantly extends the scope of our study, that is, we consider new runtime metrics, we consider a much larger time frame, use the latest long-term support Java version, and safely increase the analysis timeout to target long-running code. A key difference w.r.t. our previous study is that our new methodology targets streams used in application code, discarding streams used in test harnesses or streams internally used by the Java class library. We also revised the description of our approach to dynamically analyze streams (Section 3). We detail the instrumentation logic required to handle some peculiarities in the implementation of the Stream API and describe the extensions done to both Stream-Analyzer and DiSL in the aim of profiling new and more complex stream-related metrics for this study.

Another difference w.r.t. our previous study, is that we significantly expanded the outcome of our study (Section 5) with many findings that are presented for the first time in this article. Moreover, we report problematic stream code patterns and stream misuses affecting multiple open-source software projects, including some popular ones. Finally, we add a dedicated section (Section 6) for the discussion of implications of our findings from the perspective of three audiences: developers of the Stream API, tool builders and researchers, and educators.

This article is structured as follows. Section 2 provides background information on Java streams, introducing the terminology used in the article. Section 3 describes the entities targeted by Stream-Analyzer and outlines its implementation details. Section 4 explains our approach to conduct the large-scale characterization of streams. Section 5 presents the results of our study. Section 6 discusses implications of our work. Section 7 explains threats to validity to our study. Section 8 discusses work related to our study. Finally, Section 9 presents our conclusions.

2 | BACKGROUND

This section introduces key concepts and the terminology used in our study.

Stream and pipeline. A *stream* is a view of a sequence of data elements supporting either sequential or parallel operations that are structured in stages within an associated *pipeline*. The elements in a stream can come from multiple data sources including collections, arrays, files, and strings.[3]

Figure 1 shows a simple Java code example of a stream. In line 2, a stream is generated from `txnList`, that is, a list of objects of class `Transaction`. The pipeline contains four operations: `parallel` which parallelizes the execution of the pipeline (line 3), `filter` that discards invalid transactions (line 4), `map` which extracts the identifiers from the remaining transactions (line 5), and `collect` that here accumulates the identifiers of valid transactions into a set (line 6). Hence, the stream in the example is used to collect the set of IDs of valid transactions from `txnList`.

Source method. *Stream creation* takes place upon the call to a *source method*, that is, the method from the Java class library used to create the stream. In the example, `stream` (line 2) is the source method, which is defined in the interface `java.util.Collection` and here generates a stream from the data source `txnList`.

Source collection type. Many stream data sources are *collections*, that is, they implement the interface `Collection`[†]. We call *source collection type* the specific collection type from which a stream is created, if any. In the example, since `txnList` is a list, the source collection type is a concrete runtime implementation of `java.util.List` (e.g., `java.util.ArrayList`).

Stream type. The Stream API supports four types of streams. The interface `java.util.stream.Stream` models a stream of objects, while the interfaces `IntStream`, `LongStream`, and `DoubleStream` (all these interfaces belong to the `java.util.stream` package) model streams of primitive types. In the example, we use an object-based stream type.

Characteristics of the data source. A stream has associated a *spliterator*, that is, the parallel analogue of an iterator; it describes a (possibly infinite) collection of elements, with support for sequentially advancing, bulk traversal, and splitting off some portion of an input data for parallel computation.[1,30] Among others, the characteristics of the spliterator

[†]The fully qualified name of a class/interface appears upon first occurrence in the text; thereafter, we report only the class/interface name.

```
public Set<Long> getValidIDs(List<Transaction> txnList){
  return txnList.stream()
     .parallel()
     .filter(t -> t.getStatus() == Transaction.VALID)
     .map(Transaction::getID)
     .collect(Collectors.toSet());
}
```

```
class Transaction{
  ...
 int getID(){...}
 int getStatus(){...}
  ...
}
```

FIGURE 1 A simple Java code stream example.

define whether the data source has an *encounter order* (i.e., the data source makes its elements available in a defined order), is *sorted* (i.e., the elements in the data source have a sort order), is *concurrent* (i.e., the data source is designed to handle concurrent modification), is *distinct* (i.e., the data source does not allow duplicates), is *immutable* (i.e., the elements in the data source cannot be modified), is *sized* (i.e., the number of elements in the data source can be computed) or is *non null* (i.e., the data source does not allow null values). In the example, the characteristics of the data source upon stream creation depend on the concrete implementation of `List` used at runtime to create the stream. For instance, an `ArrayList` by default has an encounter order (the elements are encountered in index order), is sized, and is neither sorted, nor concurrent, nor immutable, nor distinct, nor non null.

Data-source size. The spliterator may also report the *data-source size*, which is the total number of elements in the data source upon stream creation, if known. With *empty stream*, we denote a stream whose data-source size is exactly zero. On the other hand, an *infinite stream* is a stream with a data source capable of generating an infinite number of elements. For instance, method `Stream.iterate` returns an infinite sequential stream whose elements are produced by the iterative application of a given function.[31] In the example, the data-source size would correspond to the total number of elements in `txnList` at the time the stream is created from it.

Operations. Upon stream creation, a pipeline is generated and it may have associated operations. Operations are divided into *intermediate* (i.e., operations that produce other streams that can be further processed) and *terminal* (i.e., operations triggering the execution of the stream).

Intermediate operations. Intermediate operations are *lazy*, that is, they do not perform any processing until a terminal operation is invoked.[1] There are *stateless* and *stateful* intermediate operations. In stateless operations, each element can be processed independently from operations on other elements, while in stateful operations, the current state may depend on the state of previously seen elements.[1] An example of a stateful operation is `limit`, which truncates elements such that the size of the resulting stream is no longer than a given length.[31] In Figure 1, `parallel`, `filter`, and `map` are all stateless intermediate operations.

Terminal operations. *Stream execution* takes place only if a stream has a terminal operation. The terminal operation triggers the *traversal* of the pipeline either to return a result (e.g., an array) or to produce side effects (e.g., printing all elements to the standard output). A stream can have at most one terminal operation, which can be executed only once.[1] In the example, `collect` is the terminal operation that triggers stream execution. Terminal operations can be *non-deterministic*. As an example, `forEach`, which performs a given action for each element of the stream, has a non-deterministic behavior since the operation does not guarantee to respect the encounter order, if any.[1]

Both intermediate and terminal operations can be *short-circuiting*, that is, when operating on an infinite input, the operation may produce a finite stream as a result.[1] Methods `limit` and `anyMatch` (i.e., a terminal operation that returns whether at least one of the elements in the stream matches a provided criteria) are examples of intermediate and terminal, short-circuiting operations, respectively.

Execution mode. A stream has an *execution mode* defining whether the stream is to be executed sequentially or in parallel. When a parallel stream is executed, typically pipeline processing is performed by fork/join[‡] tasks, all of which execute in a fork/join pool.[33] The execution mode of a stream is first set upon stream creation. In the example, method `Collection.stream` creates a sequential stream. Alternatively, calling method `Collection.parallelStream` would create a parallel stream. The execution mode can be switched by calling the `sequential` or `parallel` intermediate operations. In the example, the execution mode is parallel.

Pipeline length. A pipeline has a *length*, that is, the total number of operations in the pipeline. In the example, the length is four, as the operations `parallel`, `filter`, `map`, and `collect` compose the pipeline.

[‡]Fork/join parallelism recursively splits (*fork*) work into tasks that are executed in parallel, waiting for them to complete, and then typically merging (*join*) the results produced by the forked tasks.[32]

Collector. In line 6, `collect` performs a *reduction* using a *collector*, that is, an implementation of the interface `java.util.stream.Collector` that enables mutable reduction operations, such as accumulating elements into collections or summarizing elements according to various user-defined criteria.[34] In the example, the collector is the object returned by `Collectors.toSet()`, a method that accumulates input elements into a set.

Stream result type. After execution, the stream produces an output or a side effect. We call *stream result type* the type of the output produced after the stream is executed (typically a collection, an array, or a scalar). In the example, the stream result type is the concrete implementation of `java.util.Set` used at runtime by the collector, that is, `java.util.HashSet`.

Stream result size. After execution, the total number of elements returned by the stream, if any. In the example, the stream result size would correspond to the number of elements in the returned set.

Creation and execution methods. With *creation method* and *execution method*, we denote the topmost method in the call stack (discarding methods in the Java class library) where stream creation and execution take place, respectively. In the example, method `getValidIDs` is both the creation and execution method. On the other hand, a stream can be first created and later executed in a different class or method, such that the creation and execution method can differ.

3 | STREAM-ANALYZER

In this section, we present Stream-Analyzer, our tool for profiling stream processing on the JVM. We first detail the events and entities targeted by our tool and describe how the metrics are collected (Section 3.1). Then, we outline implementation details of Stream-Analyzer (Section 3.2).

3.1 | Metric collection

Stream-Analyzer is a novel DPA specifically designed to characterize streams at a large scale. To do so, our tool accurately detects each stream used in a Java application. Moreover, Stream-Analyzer targets a careful selection of runtime metrics suitable to be massively collected in the wild. This subsection explains the data collected by our tool.

3.1.1 | Stream creation

Upon stream creation, a pipeline associated to the new stream is created. We consider as a stream every instance of the interface `java.util.BaseStream`, the top-level interface of the Stream API.[35] We consider as a stream pipeline every subtype of the abstract class `java.util.stream.AbstractPipeline`, the abstract class in the Stream API that represents a stream and the operations in its pipeline (if any).

In the Stream API, the call to a source method triggers pipeline creation, instantiating a new object that represents the first stage of the pipeline, i.e., the *head*. This object is created via the constructors of the subtypes of `AbstractPipeline`, that is, `ReferencePipeline$Head`, `IntPipeline$Head`, `LongPipeline$Head`, or `DoublePipeline$Head`. These classes belong to the `java.util.stream` package and are the core implementations of the `Stream` interface and the primitive stream types (i.e., `IntStream`, `LongStream`, and `DoubleStream`, respectively). Stream-Analyzer uses the reference to the newly-created object representing the head of the pipeline (which is associated only to a single stream) to produce a unique ID identifying the stream. This identifier, which we call *stream ID*, is stored in class `AbstractPipeline`, so that it can be queried by the instrumentation to associate operations to a specific stream (as explained later in this section).

In the Stream API, the creation of a stream takes place in two ways. In *low-level stream creation*,[1] the user calls one of the low-level methods supporting stream creation in class `java.util.stream.StreamSupport`, all of which receive as an argument some form of a spliterator. In the second way to create streams, which we refer to as *high-level stream creation*, the user calls to one of the widely available methods in the Java class library that allow creating streams from collections, arrays, and generators of pseudo-random numbers, among others (e.g., `Collection.stream`). Such methods are wrappers to the methods available in class `StreamSupport` for creating streams, therefore high-level stream creation always ends up using low-level stream creation. Stream-Analyzer detects the two ways of creating streams to collect both the source and the creation method. When high-level stream creation is used, our instrumentation properly

recognizes calls to other source methods within the Java class library, which are discarded to track the method explicitly invoked to create the stream. In addition, the stream data source may implement the interface `Collection`, which is instrumented by Stream-Analyzer to collect the source collection type, if any.

Stream-Analyzer also intercepts stream creation to query the characteristics of the data source from which the stream is created, as reported by the associated spliterator. Each characteristic corresponds to the value returned by method `Spliterator.hasCharacteristics`, which returns a boolean indicating whether the spliterator has the given characteristic. In addition, Stream-Analyzer collects the data-source size as reported by method `Spliterator.getExactSizeIfKnown`, which returns the number of elements in the data source, provided it can be computed or −1 otherwise[§]. The data-source size is a key metric to categorize streams according to the number of input elements upon stream creation. Lastly, Stream-Analyzer captures the execution mode of the stream upon creation. Overall, the data collected upon stream creation enables describing how a stream is generated, characterizing in detail the data source.

3.1.2 | Intermediate operation creation

Once a pipeline is created, it may have operations associated. Stream-Analyzer intercepts calls to all intermediate operations defined in the interfaces `BaseStream`, `Stream`, `IntStream`, `LongStream`, and `DoubleStream`. In the Stream API, the invocation to an intermediate operation typically involves instantiating a new object representing a new stage of the pipeline. Stream-Analyzer uses the reference to this new stage to produce an ID which uniquely identifies the new operation. The profiler also captures a reference to the previous stage of the pipeline, which can be either the head or another operation. Collecting the ID of the previous stage of the pipeline allows preserving information on the structure of the pipeline as originally created by the user.

Some intermediate operations only switch a state for the execution of the pipeline, without creating a new stage, for example, `sequential` and `parallel`, which only switch the stream's execution mode. Also, some intermediate operations may trigger the creation of a new stage depending on the current state of the pipeline. For instance, the intermediate operation `unordered`, which makes an ordered stream unordered, creates a new stage in the pipeline only if it is not already unordered. Our instrumentation recognizes the case when an intermediate operation does not create a new stage, generating a unique ID for the operation, such that the full pipeline structure is captured. Lastly, we query the stream ID stored in class `AbstractPipeline` (via a reference to the head of the pipeline, available to the instrumentation upon the call to an operation) to attribute each operation to the corresponding pipeline.

Due to the design of the Stream API, some intermediate operations may internally call other operations. Our instrumentation detects these cases so that an operation can be categorized as internally executed by the Stream API. Stream-Analyzer also collects the name of the method used to invoke an intermediate operation, which is used to derive type information (i.e., whether the operation is stateful or stateless, and whether it is short-circuiting). Overall, the data profiled allows preserving the pipeline structure, tracking all the data transformations performed.

Lastly, our instrumentation treats specially the call to a `filter` operation, which filters elements in the pipeline according to a given criteria in the form of a *predicate*.[1] Stream-Analyzer computes the effectiveness of the predicate of a `filter` operation, retrieving the number of elements that match the predicate. To this end, the instrumentation replaces the given predicate by a piece of code computing its effectiveness (without altering any data element). After profiling predicate matches, the original predicate is applied to the elements in the pipeline. Since intermediate operations are lazy, Stream-Analyzer can track the effectiveness of the predicate of a `filter` operation only upon stream execution.

3.1.3 | Stream execution

Stream execution requires the invocation of a terminal operation that triggers the traversal of the pipeline. Stream-Analyzer tracks calls to all methods triggering stream execution as defined in the interfaces `BaseStream`, `Stream`, `IntStream`, `LongStream`, and `DoubleStream`. Upon terminal operation invocation, Stream-Analyzer assigns it a unique ID and attributes the operation to the corresponding pipeline thanks to the stream ID.

[§]The value returned by `Spliterator.getExactSizeIfKnown` is the number of elements that would be encountered upon pipeline traversal (`Long.MAX_VALUE` if the stream is infinite) or −1 if the data source is not sized or its size is too expensive to be computed.[30]

Stream-Analyzer collects the name of the method used to call the terminal operation. As an implementation detail of the Stream API, some methods described in the documentation of the API as terminal operations, are wrappers to terminal operations. For instance, `max` and `min`, which respectively return the maximum and minimum elements in a stream, are wrappers to the `reduce` operation, which performs a reduction on the elements in the stream using an associative accumulation function. When a wrapper is called to execute a stream, our instrumentation exclusively collects its name to track the name of method originally called by the user. The name of the terminal operation is used to derive type information, that is, whether the terminal operation is non-deterministic or short-circuiting.

When the terminal operation performs a reduction via `collect`, Stream-Analyzer captures the name of the collector used. For each terminal operation, Stream-Analyzer tracks the returned result, if any. The instrumentation determines the runtime type of the stream result when available. In addition, when the stream result is an object which size is queryable, Stream-Analyzer obtains the stream result size. For instance, the size of collections is obtained via the method `Collection.size`, implementations of the interface `CharSequence` (e.g., `java.lang.String`, `java.lang.StringBuffer`, `java.lang.StringBuilder`) provide their character count via method `length`, while arrays also implement a `length` method to query their size. We also take advantage of some terminal operations that allow us to obtain the stream result size. For instance, `count` returns a long representing the total number of elements in the stream after execution. Similarly, the terminal operation `summaryStatistics` (which is available for streams of primitive types and gives a summary data about the elements of the stream), returns an object which method `getCount` provides the same information retrieved by `count`. Also, a stream executed via `spliterator` returns a spliterator for the elements of the stream, which allows obtaining the stream result size via method `Spliterator.getExactSizeIfKnown`. The stream result size is a key metric that allows categorizing executed streams according to the number of output elements returned.

Finally, the length of the pipeline is computed along with the execution mode of the stream. The length is a key metric quantifying the set of data transformations done through a pipeline. Overall, the collected data is used to analyze how a pipeline is traversed, describing the kind of data processing performed, and the output produced by the stream.

3.2 | Implementation

This section presents implementation details of Stream-Analyzer, which high-level architecture is shown in Figure 2.

Stream-Analyzer is built on top of *DiSL*,[13] a framework for the JVM to perform dynamic analysis via bytecode instrumentation. DiSL eases developers the expression of instrumentation logic that leverages the conciseness of *aspect-oriented programming*[36] styles. Moreover, the DiSL weaver guarantees *complete bytecode coverage*, that is, DiSL instruments every Java method with a bytecode representation, enabling the complete instrumentation of the Java class library, which is notoriously hard to instrument.[37,38] Complete bytecode coverage is crucial as the Stream API is fully implemented within the Java class library, making possible for Stream-Analyzer to accurately profile all streams used by a Java application. We emphasize that our profiling technique is independent from the mechanism used to insert the instrumentation code. As a result, our technique can be implemented via any instrumentation framework that can cover the relevant events or by using manual instrumentation.

During application execution, a native Java Virtual Machine Tool Interface (JVMTI)[39] agent attached to the JVM executing the application intercepts classloading, sending loaded classes to the *DiSL server*, that is, a Java process deployed on a separated JVM. The DiSL server determines the methods to be targeted according to the instrumentation logic and inserts the instrumentation code needed to collect the target metrics. Then, the instrumented classes are returned to the JVM executing the application where they are linked,[40] allowing Stream-Analyzer to characterize stream processing.

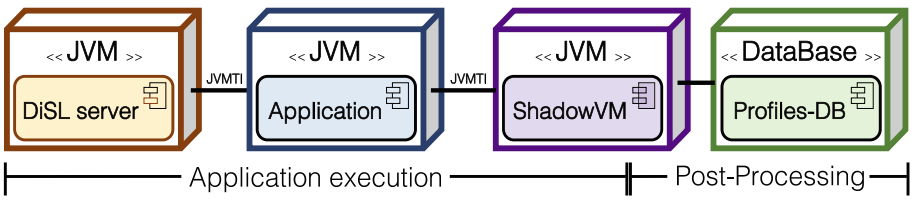

FIGURE 2 High-level architecture of Stream-Analyzer (UML deployment diagram).

To further isolate the analysis from the execution of the application, Stream-Analyzer uses *Shadow VM*,[41] a deployment setting of DiSL which allows running analyses in a separate JVM process. This feature helps mitigate the perturbations incurred by the inserted instrumentation code while also prevents known issues inherent to non-isolated approaches.[38] Upon collection, the information and metrics are sent to the Shadow VM, which contains most of the logic and data structures supporting the analysis of stream processing. This is possible thanks to a second JVMTI agent attached to the observed JVM, which dispatches the profiled data to the Shadow VM.

Finally, upon application completion, Stream-Analyzer performs an offline post-processing of the collected data. During this phase, a stream profile is produced for each stream used by the application, which includes all the information collected upon its creation and execution (as explained in Section 3.1). Such profiles are stored in a database that can be queried later to produce aggregated data and statistics of all analyzed software projects. We describe additional statistics stored in the database for each analyzed project in Section 4.2 and show examples of analyses on the data collected in Section 5. We remark that we extended DiSL for the generation and retrieval of the unique IDs assigned to the head and operations (which allow attributing an operation to the corresponding stream) as well as to support the bytecode-level transformations required to profile the effectiveness of the predicate in filter operations, as described in Section 3.1.2.

4 | CHARACTERIZING STREAMS

In this section, we describe our approach to characterize streams at a large scale, which steps are summarized in Figure 3. First, we select software projects from a large-scale, open source code base. Then, we run Stream-Analyzer on each project to characterize any stream processing. Finally, we analyze the profiles generated by Stream-Analyzer to produce aggregated statistics that evaluate many aspects on the use of streams in application code. The rest of the section further details the three steps of our methodology.

4.1 | Project selection

To increase the representatives of our study, we target a large set of open source, publicly available software projects. We select projects available in GitHub, which at time of writing is the largest code-hosting platform with more than 83 millions developers, 4 million organizations, and more than 200 million projects.[28] We first crawl GitHub to search for candidate software projects. To this end, we rely on NAB[¶], a distributed infrastructure for automatically executing custom analyses on code exercised via unit tests available in software projects hosted in large code repositories.[20] We target Java projects last updated (at the time of analysis) between January 1, 2020 and April 30, 2022. We consider the latest version (commit) of the projects in that period. We remark that our choice is motivated by the need for completing our study in a reasonable time with the available resources. We also extended NAB to filter projects according to the *number of stars*[#], which is a very common criteria for researchers to rank software projects in GitHub according to their popularity.[5,43-45] We select software projects ranked with at least three stars. This is a key difference w.r.t. our previous study,[29] which analyzed a software project, regardless of its popularity. Considering projects with at least three stars enables us to better discard too small or toy projects. Furthermore, the number of stars and the chosen time frame helps us avoid old, obsolete, or inactive projects that could bias the study. The first step of our methodology produces as an outcome a list of selected projects, which is the input of the second step as explained below.

4.2 | Project analysis

The analysis of a selected project starts by cloning its sources. Given the large-scale nature of our study, for each cloned project it is required to automatically search, build, and execute the available source code. To this end, we leverage software tests. Specifically, we target tests executable via *JUnit*,[46] a well-established and very popular framework for the JVM to implement software testing. As JUnit tests can be built and run from a build system, we target all projects using *maven*,[47]

[¶]NAB's recursive name stands for "*NAB is an Analysis Box*".

[#]Similarly to many popular social networks where users can *like* or *upvote* a content, the GitHub Web interface provides a button that a user can click to increase the number of stars of a software project. The GitHub API[42] allows the query of this number (i.e., the *stargazerCount*).

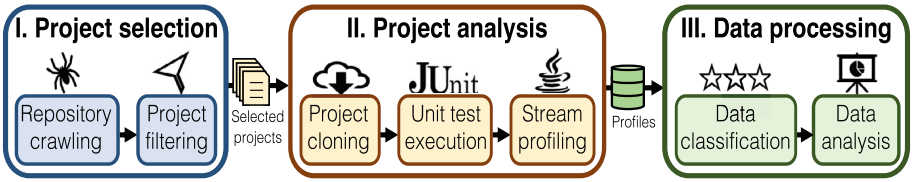

FIGURE 3 Overview of our approach to characterize stream processing.

a popular build system which we use to download the project dependencies (as configured originally by the project developers), compile the sources, and automatically run the available unit tests. We use Stream-Analyzer to profile all streams used during the execution of JUnit tests available in a software project, if any. Prior work[20,29,48] succeed to analyze multiple runtime characteristics of workloads exercised by unit tests, providing relevant statistics and revealing code patterns. Following this approach, we leverage JUnit tests to automatically exercise code available in publicly available software projects, enabling the otherwise extremely tedious task of locating candidate executable workloads for the massive application of dynamic analysis. Since our analysis focuses on streams used in application code, we discard any other stream (e.g., streams used by JVM-related processes or used in tests classes) during the data processing (see Section 4.3).

A key challenge in conducting our study is the need for automatically and massively analyzing the runtime behavior of software projects that were not designed or tested for this purpose, and hence may fail if dynamic analyses are applied to them. At a very early stage, the cloning of the project can fail (e.g., due to its deletion from GitHub or because its visibility is not public anymore). In our approach, we retry cloning the project up to three times, after which we discard it from our analysis. Another challenge we face is the need for dealing with failures both when attempting to build a project and when trying the execution of unit tests. Build failures can be caused by dependencies that cannot be resolved successfully by the build system (e.g., undeclared dependencies, deprecated/removed libraries), missing prerequisites or specific settings that the owners of the project assume as properly configured in the execution environment where the project is built (e.g., pre-installed software, credentials, datasets, files, environment variables, or paths), or problematic sources that were committed in a state where they fail to compile. Similarly, the execution of unit tests can fail due to multiple reasons, including missing prerequisites and wrong settings. In our approach, we retry up to three times the execution of a project for which maven fails to execute testing code, after which we discard the project.

When the testing code can be executed, there is also the need for dealing with non-terminating code. To deal with this issue, we set an *analysis timeout*, that is, a maximum execution time for the analysis to complete. While some projects may timeout while waiting for an input (e.g., a password, a passphrase, or an explicit selection required to a user via the standard input), other projects may execute buggy or even malicious code. In the latter case, it is crucial to ensure that the available computing resources are not compromised during the execution of potentially harmful code. To this end, we use containerization to isolate the underlying execution environment and operating system, to prevent potential issues triggered while dynamically analyzing unverified software projects. Moreover, the use of containers also eases parallelizing analysis execution by leveraging multicores, along with simplifying the deployment of the analysis on multiple machines. We extended NAB to leverage the resource quotas configurable in containerization platforms such as Docker.[49] Thanks to resource quotas, we prevent an analysis from excessively consuming the available resources, enabling the analysis of long-running unit tests. We set the analysis timeout to be three hours, after which the analysis of a project is halted. The data collected before reaching the analysis timeout is considered in our study.

Once the analysis of a project ends, analysis-related statistics are persisted in our database (the same database used to store the stream profiles) for later query. This project-level information includes the date when the project was last updated, the number of stars, the hash of the commit analyzed, and the total number of unit tests analyzed.

4.3 | Data processing

As explained in Section 3.2, Stream-Analyzer stores the generated profiles into a database where they can be later queried. The data processing starts by classifying the streams detected according to their origin. The goal of this classification is to discard all streams that were not used in application code in a three-step filtering. First, we filter all streams used outside the dynamic extent of a unit test. We exclude such streams from our analysis as streams belonging to this group are used in the set-up phase of unit tests (e.g., via methods tagged with the `@Setup` annotation) or during the building of the

project (e.g., streams used by the build system or employed by JVM-related processes). Second, we exclude all streams used directly by a *test class*, that is, any class with a method annotated with a JUnit tag (e.g., `@Test`) or extending a JUnit test class (e.g., `junit.framework.TestCase`) or a subtype of a test class. Finally, we discard streams which execution method indicates that the stream was executed internally by the Java class library. We consider the remaining streams as used in application code. Our analysis focuses exclusively on these streams since their study can more effectively reveal how Java developers are using the Stream API. This is an important difference w.r.t. our previous study,[29] which reported the use of streams regardless of their origin, including the analysis of streams used by test-harness classes, the build system, JVM-related processes, and streams executed internally by the Java class library.

Our data analysis is supported in Python scripting that queries the stream profiles in the database to produce statistics on the popularity of features of the Stream API and allows the detection of singularities in the use of streams, including code patterns and misuses, as shown in the next section.

5 | RESULTS

In this section, we present the results of our study. We first present general analysis- and project-related statistics (Section 5.1). Then, we present an analysis on the popularity of features of the Stream API and findings on how developers are using streams (Section 5.2). Finally, we present problematic stream processing detected in the projects analyzed (Section 5.3).

5.1 | Analysis- and project-related statistics

Table 1 shows the number of projects considered in our study, including statistics on how many of them rely on streams processing and the number of streams detected. We initially analyzed a total of 43,496 Java projects, which distribution per year can be seen in column *Total projects*. This extensive selection allows diversity in aspects such as size, domain, and popularity. We note that a significant number of projects were last updated during the current year. As we are targeting projects with stars, many of the analyzed projects are expected to be updated recently. From the initial set of projects, we discarded the ones that failed to clone, failed to build, lacked JUnit tests, failed to execute unit tests, or used no stream processing, resulting in 4063 projects (column *#Projects with streams*). In these software projects, we detected a total of 23,254,335 streams (column *#Streams*), from which 2,653,467 are used as part of application code (column *#Streams in app code*) during the execution of 5818 JUnit tests. We present in Table 2 the ten software projects using the highest number of streams. Each row reports the name of the owner of the project followed by the name of the project as in the GitHub url (e.g., https://github.com/agarciadom/xeger) and the number of lines of code (LOC) in the project (considering only Java code). We describe some of those projects more in detail while discussing our findings later in this section.

5.2 | Characterization

Here, we present our results regarding the degree of adoption of the main features of the Stream API by Java developers along with the analysis of the dynamic metrics collected to show how streams are used. We show our results in the form of findings and, where applicable, we compare them with the corresponding outcome highlighted by related work. Table 3 summarizes all our findings. A checkmark (✓) in the table indicates that related work found consistent results at a smaller scale.

TABLE 1 Number of projects and streams detected per year.

Year	Total projects	%	#Projects with streams	%	#Streams	%	#Streams in app code	%
2020	10,438	24.00%	897	22.08%	3,357,050	14.44%	804,561	30.32%
2021	14,331	32.95%	1365	33.60%	5,191,929	22.33%	1,364,608	51.43%
2022	18,727	43.05%	1801	44.33%	14,705,356	63.24%	484,298	18.25%
Total	43,496		4063		23,254,335		2,653,467	

TABLE 2 Projects using the highest number of streams.

Project	LOC	Streams	%
agarciadom/xeger	348	911,000	34.33
andrewkkchan/client-ledger-core-db	3585	780,449	29.41
dmeoli/WS4J	3061	190,106	7.16
ikhoury/rstreamer	2094	167,396	6.31
awslabs/amazon-sqs-java-temporary-queues-client	6748	142,156	5.36
vangj/jbayes	7337	87,476	3.30
korlucene/argo-nori-analyzer	6931	67,919	2.56
yu-iskw/kuromoji-for-bigquery	529	60,455	2.28
finmath/finmath-forward-initial-margin	13,186	30,675	1.16
amaembo/streamex	6769	27,080	1.02

Source methods. We find a total of 60 different source methods used for stream creation and we summarize the ten most popular ones in Table 4. Our results show that streams are created mainly from collections (94.25%, considering methods `List.stream`: 85.08%, `List.parallelStream`: 5.43%, `SortedSet.stream`: 2.18%, `Set.stream`: 0.88%, and `Collection.stream`: 0.68%), in particular, from lists and sets. Other popular source methods include `Arrays.stream`: 1.72% (returns a sequential stream with the specified array as its data source[50]), `IntStream.range`: 1.71% (returns a sequential `IntStream` of consecutive integers in a given range[51]), `Stream.concat`: 0.73% (returns a new stream containing all the elements of two streams passed as input[31]), and `Stream.generate`: 0.23% (returns an infinite sequential stream where each element is generated by the `java.util.function.Supplier`[52] passed as argument). We also find that less than 1% of the detected streams generated from low-level stream creation, that is, `StreamSupport.stream` (0.59%), `StreamSupport.intStream` (0.11%), `StreamSupport.longStream` ($7 \cdot 10^{-5}$%), and `StreamSupport.doubleStream` ($7 \cdot 10^{-5}$%). We are not aware of any other study characterizing the source method from which streams generate.

F1: *Streams are mainly created from collections.*

Source collection types. We detect a total of 95 different source collection types, among which we summarize the ten most popular ones in Table 5. Our findings show that developers generate streams mainly from lists and sets. We are not aware of related work reporting the specific collection types from which streams are created.

F2: *Streams generated from collections are mostly created from lists and sets.*

Stream types. We find that streams of objects are the most popular ones. 97.05% of the streams detected are of type `Stream`, 2.26% of type `IntStream`, 0.64% of type `DoubleStream`, and only 0.04% are of type `LongStream`. Our findings show that developers prefer streams whose elements are references to objects. As far as we are aware, previous studies do not report the popularity of the stream types used in the projects analyzed.

F3: *Object-based stream types are far more popular than primitive-based stream types.*

Characteristics of the data source. We characterize the data source from which the stream is generated[‖]. We find that most of the streams are created from a data source having an encounter order (92.45%). This is expected as most

[‖]We note that our instrumentation is able to capture the spliterator associated to the stream upon its creation. Such characteristics may be changed upon stream execution.

TABLE 3 Summary of our findings.

	Finding	Tanaka et al.[9]	Khatchadourian et al.[6]	Nostas et al.[7]	This study
F1	Streams are mainly created from collections	✓	✓	✓	✓*
F2	Streams generated from collections are mostly created from lists and sets				✓
F3	Object-based stream types are far more popular than primitive-based stream types				✓
F4	Stream data sources often allow null and duplicated values, have ordering constraints, are mutable, sized, and do not support concurrent modification		✓†		✓
F5	Stream data sources typically have few elements				✓
F6	Mapping and filtering are by far the most popular intermediate operations	✓	✓	✓	✓
F7	Stateful and short-circuiting intermediate operations are rarely used		✓		✓
F8	50.55% of the predicates of filter operations match no element of the pipeline				✓
F9	`collect`, `findFirst`, `forEach`, and `toArray` are the most common terminal operations	✓	✓	✓	✓
F10	Deterministic terminal operations are more popular than non-deterministic ones		✓		✓
F11	Parallel streams are not popular		✓	✓	✓
F12	Stream pipelines are typically composed of few operations				✓
F13	Collectors returning lists, sets, strings, and maps are the most used		✓	✓	✓
F14	Concurrent collectors are not popular		✓		✓
F15	Lists and Optionals are by far the most popular types collecting the stream result				✓
F16	Among the streams returning an `Optional`, empty optionals or null optionals are rarely used				✓
F17	The output produced by the stream often contains few elements				✓
F18	The most used sequential pipeline structures follow basic map-reduce-like patterns (`filter-collect`, `filter-findFirst`, `map-collect`)			✓	✓
F19	Parallel streams mostly perform iterative-style processing using `forEach` patterns			✓	✓
F20	Less than 15% of the streams detected are executed by third-party libraries				✓
F21	Streams are mostly created and executed in the same method				✓

*Khatchadourian et al. report only ordering of the data source.
†Unlike related work,[6,7,9] we consider all high- and low-level stream creation methods.

TABLE 4 Most popular source methods.

Source method	Occurrences	%
`java.util.List.stream`	2,257,693	85.08
`java.util.List.parallelStream`	144,128	5.43
`java.util.SortedSet.stream`	57,774	2.18
`java.util.Arrays.stream`	45,636	1.72
`java.util.stream.IntStream.range`	45,270	1.71
`java.util.Set.stream`	23,466	0.88
`java.util.stream.Stream.concat`	19,305	0.73
`java.util.Collection.stream`	18,107	0.68
`java.util.stream.StreamSupport.stream`	15,710	0.59
`java.util.stream.Stream.generate`	6083	0.23

TABLE 5 Most popular source collection types.

Source collection type	Occurrences	%
`java.util.Arrays$ArrayList`	913,521	36.47
`java.util.Collections$SingletonList`	782,578	31.24
`java.util.ArrayList`	288,161	11.50
`java.util.Collections$EmptyList`	161,536	6.45
`software.amazon.awssdk.core.util.DefaultSdkAutoConstructList`	142,058	5.67
`java.util.LinkedList`	85,650	3.42
`jre.com.google.common.collect.Sets$FilteredSortedSet`	43,296	1.73
`java.util.Collections$UnmodifiableRandomAccessList`	30,641	1.22
`jre.com.google.common.collect.AbstractMapBasedMultimap$WrappedNavigableSet`	15,710	0.63
`java.util.HashMap$EntrySet`	12,335	0.49

streams generate from lists and arrays (data structures that typically preserve an index order). We find that most of the data sources allow duplicates (65.40%) and null values (68.49%), which can also be explained by the fact that most of the streams are generated from lists. Lastly, we find that most of the streams are created from data sources not having a sort order (95.80%), which are mutable (66.10%), sized (97.98%), and do not support concurrent modification (99.96%). Khatchadourian et al.[6] report that streams in the analyzed projects are largely ordered. We confirm this finding by detecting *ordering constraints* (i.e., whether the stream data source has an encounter and/or a sort order to be preserved) as reported at runtime by the spliterator upon stream creation.

F4: *Stream data sources often allow null and duplicated values, have ordering constraints, are mutable, sized,and do not support concurrent modification.*

Our findings indicate that most of the streams detected may not be parallelized straightforwardly. As explained in the documentation of the Stream API, operations in the pipeline typically run in parallel more efficiently in the absence of ordering constraints.[1] To our knowledge, no other study analyzes the characteristics reported at runtime by the spliterator associated to the stream.

Data-source size. Table 6 reports the distribution of the data-source size. We were able to query the data-source size of 97.98% of all streams detected. For the remaining ones (reported as *N/A* in the table), the exact size was reported as unknown. An example of this case are streams created from files (e.g., via the source methods `java.nio.file.Files.list`, `java.util.zip.ZipFile.stream`, `java.util.jar. JarFile.stream`),

TABLE 6 Distribution of the stream data-source size.

Size range	Occurrences	%
N/A	53,705	2.02
0	508,664	19.16
$[10^0, 10^1)$	2,041,446	76.93
$[10^1, 10^2)$	37,738	1.42
$[10^2, 10^3)$	10,204	0.38
$[10^3, 10^4)$	1234	0.05
$[10^4, 10^5)$	340	0.01
$[10^5, 10^6)$	8	$3 \cdot 10^{-3}$
$[10^6, 10^7)$	9	$3 \cdot 10^{-3}$
$[10^7, 10^8)$	9	$3 \cdot 10^{-3}$
$[10^8, 10^9)$	2	$7 \cdot 10^{-4}$
Infinite	108	$4 \cdot 10^{-2}$

which data source is not sized. We find that most of the data sources used to generate streams contain a number of elements between 10^0 and 10^1. In this range, the most popular data-source sizes are 1 (35.09%), 2 (36.40%), and 3 (2.39%). We also find that 19.16% of all the data sources analyzed contain zero elements upon creation. We detected 108 infinite streams (explaining the last category in the table). When not considering infinite streams, the maximum data-source size is 100,000,000 and the mean is 124. Overall, we find that streams often generate from data sources containing few elements. Since our study only focuses on the size of input data used by unit tests, we remark that these sizes can be smaller than the ones used in production scenarios, which may lead to different conclusions regarding the data-source size, as we discuss in Section 6. We are not aware of related work reporting the data-source size. In Section 5.3.1, we analyze projects exposing abundant empty streams.

F5: *Stream data sources typically have few elements.*

Intermediate operations. We detect the occurrence of 2,666,245 intermediate operations in total. We show the ten most popular ones in Table 7. Our analysis shows that intermediate operations used for filtering (73.92%) and mapping (20.12%, considering operations `map`: 14.61%, `mapToInt`: 3.47%, `mapToDouble`: 1.63%, `mapToLong`: 0.27%, and `mapToObj`: 0.15%) are the most popular ones. This finding is consistent with the results by Tanaka et al.,[9] Khatchadourian et al.,[6] and Nostas et al.[7] (obtained at a smaller scale), indicating that streams are mainly used to perform MapReduce style processing.

While we find that 47,264 of the streams detected use the `sorted` operation, we do not find any occurrence of `unordered`. This confirms that most of the analyzed streams have ordering constraints due to the nature of the stream data source (as previously reported) or due to pipeline transformations. We analyze inefficient ordered streams later in this section (see Section 5.3.3).

F6: *Mapping and filtering are by far the most popular intermediate operations.*

We find that only 0.24% of the intermediate operations detected are short-circuiting. Lastly, Khatchadourian et al. report that stateful operations are rarely used. We confirm this observation at a much larger scale, finding that stateless operations are by far more popular (95.69%). The exclusive use of stateless operations is recommended as according to the documentation of the Stream API, pipelines containing only stateless intermediate operations can

TABLE 7 Most popular intermediate operations.

Intermediate operation	Occurrences	%
`filter`	1,970,944	73.92
`map`	389,505	14.61
`mapToInt`	92,407	3.47
`sorted`	47,264	1.77
`mapToDouble`	43,415	1.63
`boxed`	41,300	1.55
`flatMap`	31,939	1.20
`mapToLong`	7309	0.27
`limit`	6514	0.24
`mapToObj`	3882	0.15

TABLE 8 Distribution of the number of elements matching a predicate in filter operations.

Range	Occurrences	%
0	996,224	50.55
$[10^0, 10^1)$	965,942	49.01
$[10^1, 10^2)$	6569	0.33
$[10^2, 10^3)$	2150	0.11
$[10^3, 10^4)$	57	$3 \cdot 10^{-3}$
$[10^6, 10^7)$	2	$1 \cdot 10^{-4}$

be processed in a single pass, whether sequential or parallel, with minimal data buffering, potentially improving performance.[1]

F7: *Stateful and short-circuiting intermediate operations are rarely used.*

As previously reported, we detected 1,970,944 occurrences of `filter`. We find that all filter operations were executed. The total number of elements evaluated by the predicates of the filters detected was 26,722,820. Among them, 11,936,568 elements matched the predicate of the filter operation. Table 8 reports the distribution of the elements matching the predicate of a filter operation. We find that most of the predicates matched a number of elements between 10^0 and 10^1. In this range, the most popular number of elements matched by a predicate are 1 (47.46%), 3 (1.17%), and 2 (0.22%). The maximum number of elements matched by a predicate is 5,101,466 and the mean is 12.24. We also find that 996,224 (50.55%) predicates match no element of the pipeline. In Section 5.3.2, we discuss our findings on a project with an abundant number of predicates evaluating only a single element that is never matched.

F8: *50.55% of the predicates of filter operations match no element of the pipeline.*

Terminal operations. We detect the occurrence of 2,653,451 terminal operations in total and report the ten most popular ones in Table 9. Our results show that the most used terminal operations are `collect` (56.89%) and `findFirst` (30.49%). The terminal operation `findFirst` returns an `Optional` describing the first element of the stream, or an *empty optional* (i.e., an `Optional` which has no value present, as reported by method `Optional.isPresent`[11]) if the

TABLE 9 Most popular terminal operations.

Terminal operation	Occurrences	%
`collect`	1,509,663	56.89
`findFirst`	808,925	30.49
`forEach`	172,638	6.51
`toArray`	146,910	5.54
`sum`	8871	0.33
`count`	2048	0.08
`reduce`	1201	0.05
`max`	952	0.04
`findAny`	786	0.03
`min`	532	0.02

stream is empty.[31] There is also an important presence (6.51%) of streams executed via `forEach`. While `collect` is key for map-reduce-like data processing, the use of `forEach` indicates that developers rely on streams also to process data iteratively and nondeterministically. This finding confirms the results by Tanaka et al., Khatchadourian et al., and Nostas et al. at a larger scale. We also find the presence of `toArray` (5.54%), which returns an array containing the elements of the stream, and of multiple forms of data aggregations and reductions.

F9: *`collect`, `findFirst`, `forEach`, and `toArray` are the most common terminal operations.*

We find that deterministic terminal operations are the most used (93.46%). In this context, our results are consistent with the ones reported by Khatchadourian et al. We also find that `forEachOrdered` (the deterministic and ordered counterpart of `forEach`) is less popular (0.002%) than `forEach` (6.51%), and that `findFirst` is far more popular (30.49%) than its non-deterministic counterpart `findAny` (0.03%), which returns an `Optional` describing any element of the stream, or an empty optional if the stream is empty.

F10: *Deterministic terminal operations are more popular than non-deterministic ones.*

Execution modes. Among a total of 2,653,467 streams detected, 2,508,651 were created as sequential streams and 144,816 as parallel ones. We find that switches in the execution mode are rare as only 36 sequential streams were switched to parallel execution (we found no parallel stream switched to be sequential). We also find that 16 streams created as sequential ones, use an unneeded `sequential` operation (which has no effect on an already sequential stream).

We find that 2,653,451 streams were executed whereas only 16 were not (i.e., streams that are created but lack a terminal operation). Only three stream executions ended throwing an exception (i.e., `java.lang.RuntimeException` due to missing data to read, `java.lang.IllegalArgumentException` due an incorrect HTTP response, and `java.lang.NoClassDefFoundError` due a missing class at runtime).

Among the executed streams, we find that 94.54% were executed sequentially. This finding is consistent with the results obtained on a smaller scale by Khatchadourian et al. (34 projects and 1038 streams analyzed, of which 13 were parallel) and Nostas et al. (610 projects analyzed, among which the authors report that in 113 there was at least a single parallel stream creation).

F11: *Parallel streams are not popular.*

TABLE 10 Distribution of the pipeline length.

Pipeline length	Occurrences	%
2	2,319,571	87.42
1	190,286	7.17
3	104,158	3.93
4	19,617	0.74
5	19,580	0.74
6	230	$8 \cdot 10^{-2}$
7	19	$7 \cdot 10^{-3}$
0	5	$1 \cdot 10^{-3}$
8	1	$3 \cdot 10^{-5}$

Considering the low usage of stateful operations in the analyzed stream pipelines (F7)**, our finding may reveal potential missed speedups that could be obtained by parallelizing stream processing. Nonetheless, as pointed out by Lea et al.,[53] when deciding whether to parallelize a stream, it is crucial to estimate if the sequential execution already exceeds a minimum threshold, which—as Lea et al. propose—could be measured in terms of execution time or an estimation of the number of elements processed. The idea is finding whether, despite the presence of parallelization overheads, the parallel execution of a stream can result in performance gains. According to our findings, stream data sources typically contain few elements (F5), therefore only a small selection of projects processing large amount of data may truly benefit from stream parallelization.

Pipeline length. Table 10 reports the distribution of the pipeline length. Our results show that few operations are used in the projects analyzed, with an average length of only 2.05. Most of the streams are composed by only one (7.17%) or two (87.42%) operations. We also find that five streams are created but no operation is called in their pipelines (meaning that they were not executed). Overall, the results show that pipeline composition involves mostly few operations. To our knowledge, no previous study analyzes the pipeline length.

F12: *Stream pipelines are typically composed of few operations.*

Collectors. As shown in Table 9, we detected a total of 1,509,663 invocations to `collect`, used to perform mutable reductions. Table 11 shows the ten most popular collectors used during a mutable reduction. Note that we identify the collector with the method used to obtain it (from class `Collector`). We denote as *Custom* the category aggregating the occurrences of custom collectors (created by the developer and therefore not available by default in the Java class library). Our outcome is consistent with the results reported by Khatchadourian et al. and Nostas et al. in finding that reductions via `collect` mostly produce lists and sets.

F13: *Collectors returning lists, sets, strings, and maps are the most used.*

Like Nostas et al. we find that many streams return a concatenation of the input characters via `Collectors.joining` or collect the reduction output into a map (e.g., via `Collectors.toMap`). We also confirm the observation done by both Khatchadourian et al. and Nostas et al. that *concurrent reductions*†† are rarely used. In particular, we do not find any use of a concurrent collector (e.g., `Collectors.groupingByConcurrent`). This finding shows

**The labels in the format F# point to the corresponding finding as presented in this section.

††In the absence of ordering constraints to process a pipeline, a reduction can use a concurrently modifiable collection, eliminating the need for combining intermediate reduction results. This is formally known as *concurrent reduction* and can potentially provide a boost to the parallel execution performance.[1]

TABLE 11 Most popular collectors.

Collector	Occurrences	%
`Collectors.toList`	1,420,063	94.06
`Collectors.joining`	52,133	3.45
`Collectors.toSet`	19,079	1.26
`Collectors.collectingAndThen`	8635	0.57
`Collectors.toMap`	3416	0.23
`Collectors.groupingBy`	2763	0.18
Custom	2235	0.15
`Collectors.toCollection`	1247	0.08
`Collectors.toUnmodifiableMap`	5	$3 \cdot 10^{-3}$
`Collectors.toUnmodifiableList`	4	$3 \cdot 10^{-3}$

TABLE 12 Most popular stream result types.

Stream result type	Occurrences	%
`java.util.ArrayList`	1,420,518	53.53
`java.util.Optional`	810,283	30.54
`void`	172,752	6.51
`int[]`	89,725	3.38
`java.lang.String`	52,125	1.96
`double[]`	30,324	1.14
`java.lang.Object[]`	24,385	0.91
`java.util.HashSet`	19,149	0.72
`long`	8438	0.32
`java.util.Collections$EmptySet`	7877	0.30

that despite the Stream API offers a variety of collectors, streams seem mostly used to perform simple non-concurrent reductions whose results are mainly stored in collections and strings.

F14: *Concurrent collectors are not popular.*

Stream result types. We detected a total of 114 different stream result types. Table 12 summarizes the 10 most popular stream result types found. We find that `ArrayList` is the most popular data structure used to store the result of a stream (53.53%), followed by `Optional` (30.54%), while 6.51% of the executed streams do not return any result (e.g., streams performing the `forEach` and `forEachOrdered` terminal operations, which are `void` methods). This is expected, given the popularity of reductions collecting the results in lists as well as of `forEach`-like terminal operations.

F15: *Lists and Optionals are by far the most popular types collecting the stream result.*

Among the streams returning a boolean, that is, executed via the terminal operations `allMatch` (0.011%) and `anyMatch` (0.004%) (no occurrence of `noneMatch` was detected), we find that 65.79% of the occurrences of `allMatch` returned `false`, while all occurrences of `anyMatch` returned `true`. Among the streams returning an `Optional`, we

find that 4290 streams return a *null optional* (i.e., an `Optional` containing a null value), while only 27 streams return an empty optional. We are not aware of any related work characterizing stream result types.

F16: *Among the streams returning an* `Optional`*, empty optionals or null optionals are rarely used.*

Stream result size. Table 13 shows the distribution of the stream result size. We were able to query the stream result size of 62.45% of all streams executed. For the remaining ones (reported as *N/A* in the table), this size could not be obtained. This is the case for streams executed via a `void` terminal operation or streams returning either a scalar (except streams executed via `count`), a not sized spliterator, or more in general, an object not reporting its size.

We find that most of the outputs returned by streams contain a number of elements between 10^0 and 10^1. In this range, the most popular sizes are 1 (38.11%), 3 (2.18%), 4 (1.52%), and 2 (1.36%). Also, many streams return zero elements (15,20%), among which we find that their outputs are almost all of type `ArrayList` (373,377), followed by `HashSet` (13,987), `Collection$EmptySet` (7,877), and `int[]` (5,119). In Section 5.3.1, we analyze some projects using abundant streams with a result size equal to zero. We find three infinite streams, as reported by the returned spliterator (explaining the last category in the table). When not considering infinite streams, the maximum data-source size is 10,000,000 and the mean is 37. Therefore, we find that among the streams returning an output, it often contains few elements. To our knowledge, no related work reports statistics on the stream result size.

F17: *The output produced by the stream often contains few elements.*

Pipeline structure. We find a total of 310 different pipeline structures among all the streams detected, which ten most used are shown in Table 14. The most common pipeline structures involve filtering and then either mutable reductions via `collect` (42.13%) or searches via `findFirst` (30.45%), followed by `map-collect` (9.12%), `forEach` (6.21%), and `mapToInt-toArray` patterns (3.38%). Our results are consistent to some extent with the work of Nostas et al. as they find that sequential pipeline structures involving `filter`, `map`, `collect`, and `findFirst` are the most popular ones. Complementary, we find the presence of patterns involving `toArray`, along with pipelines relying on the intermediate operations `mapToInt`, `boxed`, `mapToDouble`, `distinct`, `sorted`, and `flatMap`.

Among the pipeline structures detected, only 19 correspond to parallel stream processing (144,851 occurrences in total). We find that most of them (99.91%) are single-operation pipelines executing the `forEach` operation (144,721). Similarly to Nostas et al., we find that the abundant presence of pipelines using only the `forEach` operation indicates

TABLE 13 Distribution of the stream result size.

Size range	Occurrences	%
N/A	996,374	37.55
0	407,543	15.36
$[10^0, 10^1)$	1,192,568	44.94
$[10^1, 10^2)$	40,680	1.53
$[10^2, 10^3)$	14,681	0.55
$[10^3, 10^4)$	1253	0.05
$[10^4, 10^5)$	349	0.01
$[10^5, 10^6)$	9	$3 \cdot 10^{-3}$
$[10^6, 10^7)$	4	$3 \cdot 10^{-3}$
$[10^7, 10^8)$	3	$1 \cdot 10^{-3}$
Infinite	3	$1 \cdot 10^{-3}$

TABLE 14 Most popular pipeline structures.

Pipeline structure	Occurrences	%
`filter, collect`	1,118,026	42.13
`filter, findFirst`	808,091	30.45
`map, collect`	244,541	9.22
`forEach`	164,716	6.21
`mapToInt, toArray`	89,628	3.38
`map, map, collect`	43,365	1.63
`mapToDouble, boxed, collect`	29,869	1.13
`collect`	23,165	0.87
`map, distinct, sorted, toArray`	16,972	0.64
`flatMap, filter, collect`	14,439	0.54

TABLE 15 Most popular third-party libraries (executing streams) used by the analyzed projects.

Library	#Streams	#Projects using the library
`Mockito`	167,596	7
`Apache Lucene`	89,863	6
`Apache Beam`	60,455	1
`Hibernate ORM`	18,760	8
`Eclipse Glassfish`	6190	14
`ImgLib2`	1766	5
`Apache Solr`	1327	1
`JSON Schema Validator`	1138	2
`neo4j`	125	2
`Eclipse Yasson`	41	2

that parallel streams are mostly used by developers to perform iterative, non-deterministic processing. In Section 5.3.3, we analyze problematic streams relying on `forEach`.

F18: *The most used sequential pipeline structures follow basic map-reduce-like patterns (`filter-collect`, `filter-findFirst`, `map-collect`).*

F19: *Parallel streams mostly perform iterative-style processing using `forEach` patterns.*

Creation and execution methods. We analyze the creation methods of the streams detected to classify them according to their origin. We find that some software projects use streams executed by third-party libraries. In Table 15, we show the ten most used libraries according to the number of streams executed by them, along with the number of projects using such libraries. We find that less than 15% of the streams detected are executed by external libraries. In Section 5.3.4, we analyze inefficient stream code executed by a popular open-source library. Finally, we also find that 97.76% of the streams detected were created and executed in the same method.

F20: *Less than 15% of the streams detected are executed by third-party libraries.*

F21: *Streams are mostly created and executed in the same method.*

5.3 | Stream-related issues

In this subsection, we present inefficient stream processing in the projects analyzed, including some misuses of the Stream API. Unless explicitly specified otherwise (see Section 5.3.4), the following results target streams used in application code.

5.3.1 | Empty streams

As previously reported, we find a total of 508,664 empty streams. Although creating an empty stream is supported by the Stream API (i.e., via the source methods `Stream.empty` and `Stream.ofNullable`), we find that only 249 empty streams were created using `Stream.empty` and we do not find any occurrence of `Stream.ofNullable`. The remaining empty streams (508,415) are mainly created from collections (97.93%), in particular from lists (95.62%) and sets (1.76%). Despite these streams are widely spread within 198 software projects, we find that most of them are used by only two projects as follows.

First, we find 161,165 empty streams in `dmeoli/WS4J`[54] (*WordNet*[‡‡] Similarity for Java), a library providing algorithms to measure the semantic similarity/relatedness between words in WordNet. All the empty streams used in this project are executed sequentially and have a pipeline in the form `map-collect`, using `Collectors.toList`. Moreover, we find that the stream result size of all of these streams is always zero[§§]. Therefore, the `map` operation and the reduction are applied to no input element and the streams always produce an empty list.

Second, we find 142,058 empty streams in `awslabs/amazon-sqs-javatemporary-queues-client`,[56] an *Amazon Simple Queue Service*[¶¶] Java client that supports creating lightweight queues, for use in common messaging patterns such as *Request/Response*.[58] All the empty streams used in this project are executed in parallel and have a single-operation pipeline containing the `forEach` terminal operation. As a result, for the detected stream executions the side effect triggered via `forEach` is never reached in the absence of an input element.

In both cases, the abundant use of empty streams in application code may lead to overheads due the unneeded allocation of resources to process no data (e.g., object allocations/de-allocations and extra *virtual method calls*[##]). Provided that the usage of empty streams that we detected via unit tests matches a realistic scenario commonly taking place in application code, these empty streams should be considered candidates for removal (e.g., avoiding using a stream when the data source is known to be empty).

5.3.2 | Single-element streams

As previously shown, we find that 931,119 streams have a data-source size equal to one. We examine these streams and find that most of them are used by a single project. In particular, 780,449 single-element streams are used in `andrewkkchan/client-ledger-core-db`,[59] an event sourcing finance system. All of these streams have a pipeline in the form `filter-findFirst`. We find that the predicate used in the `filter` operation never matches the single

[‡‡] WordNet is a lexical database of semantic relations between words in more than 200 languages[55]

[§§] Since the data-source size is captured upon stream creation, it is possible that a stream is created from an empty data source which is filled before the stream is later executed. In such a case, the stream result size of the previously empty stream would differ from zero (which is not the case for the analyzed empty streams).

[¶¶] Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables to decouple and scale microservices, distributed systems, and serverless applications.[57]

[##] Since the Java bytecode does not support function types, lambda expressions are treated as interfaces with a single abstract method, which can lead to many virtual method calls. These virtual calls could prevent optimizations performed by the Just-In-Time (JIT) compiler.[26]

element evaluated, such that it is never filtered. In consequence, upon the execution of `findFirst`, all the streams return always the unique and unmodified input value made available by the data source.

If the aforementioned scenario (i.e., abundant single-element streams producing no transformation through the pipeline) reflects real application-code usage, then those single-element streams should be considered for refactoring or removal to help mitigate potential stream-related overheads (e.g., replacing stream processing by a conditional recognizing a single-element data source and performing a direct action accordingly).

5.3.3 | Misuses of sorted-forEach-like patterns

As previously reported, we find that 47,264 streams rely on the `sorted` intermediate operation. Among them, we find streams that are nevertheless executed via the `forEach` terminal operation. As stated in the documentation of the Stream API, the behavior of `forEach` is explicitly non-deterministic.[1] On the one hand, when executed in parallel, `forEach` does not guarantee to respect ordering constraints in the stream, as this would sacrifice the benefit of a parallel execution. For any given element, the action performed by `forEach` may be performed at whatever time and in whatever available thread.[1] On the other hand, as pointed out by Goetz et al.,[60] `forEach` has the freedom to not preserve the encounter order even for sequential streams.

We find that `ysc/short-text-search`[61] (a customizable precise short text search service) uses 11 streams with pipelines in the form `sorted-limit-forEach`. These streams are created from method `Set.parallelStream`, therefore the streams are parallel and their data source has no encounter neither sort order. Upon execution, the ordering introduced by `sorted` in the aforementioned streams is ignored and the action delegated to the parallel `forEach` cannot assume that ordering constraints are met. Moreover, if ordering is not required, the developers should consider removing the statefulness introduced by the `sorted` operation, which may help speedup parallel stream processing.[1]

We also find 11 software projects containing sequential streams that use the `sorted` operation and are executed via `forEach`. For instance, we find that `GoogleCloudPlatform/kafka-pubsub-emulator`[62] (an implementation of *Google Cloud Pub/Sub*‖‖ backed by Apache Kafka[64]) and `cryptomkt/cryptomkt-java`[65] (the *CryptoMarket**** Java SDK), resort to streams with a pipeline implementing `sorted-forEach`-like patterns.

Overall, given the non-deterministic nature of `forEach` stream processing, the aforementioned streams can be considered as misuses of the Stream API. The code should be changed either to use `forEachOrdered` (which specification ensures the encountered order regardless of sequential or parallel stream processing) or to remove the `sorted` operation if no order is actually required.

5.3.4 | Inefficient streams in library code

We find five projects using streams executed by `Selenium`,[67] a popular open-source software project encapsulating tools and libraries to enable Web browser automation. All of these streams (40) share three characteristics. Their source collection type is `java.util.TreeMap$EntrySet`, meaning that the data source is distinct and it preserves an encounter and sort order. They are executed in parallel and use the `distinct` operation. In particular, they implement pipelines in the form `filter-distinct-filter-collect` (32 occurrences) and `filter-filter-distinct-collect` (eight occurrences) across five classes filtering the capabilities††† of the browsers supported by the Selenium.

In the absence of an intermediate operation potentially introducing duplicates (e.g., a mapping), the use of `distinct` is unneeded in these streams, as the data source does not allow duplicates. Moreover, as stated in the documentation of the Stream API, pipelines containing stateful operations (such as `distinct`) may require multiple passes on the data or may need to buffer significant data, resulting in limited parallel performance.[1] Therefore, removing the `distinct` operation from the aforementioned streams may help improve parallel stream performance.

On the other hand, as pointed out by Lea et al.,[53] it is crucial to determine whether parallel stream execution is worthwhile despite parallelization overheads. To this end, a technique they suggest is the use of a threshold defining whether

‖‖ Google Cloud Pub/Sub is a messaging middleware providing many-to-many, asynchronous messaging between services[63]

*** CryptoMarket is an online platform to buy cryptocurrencies.[66]

††† In Selenium, it is possible to declare the set of capabilities that should be enabled in a browser to perform the testing of Web applications.[68]

the pipeline is to be executed in parallel. For instance, the data-source size can be considered a factor in the estimation of the computations to be performed by a stream. In such a case, if a data-source size is greater than a given threshold, then the stream generated from it can be switched to be executed in parallel in the search for speedups. Otherwise, a sequential execution may be preferred to avoid parallelization overheads (e.g., due to the execution and scheduling of parallel tasks in the fork/join pool). We find that the average data-source size of the streams detected is only 4.12. The developers of the project may consider to evaluate whether parallel execution is worthwhile (e.g., via a threshold considering the data-source size) to better leverage parallel stream processing.

Finally, we note that for 32 of the aforementioned streams, the stream result type is `com.google.common.collect.RegularImmutableMap`.[69] Such streams may also benefit from a pipeline refactoring to include `unordered`, since `RegularImmutableMap` is a data structure that does not guarantee to preserve ordering constraints. As explained in the documentation of the Stream API, for parallel streams, relaxing ordering constraints often enables more efficient parallel execution.[1]

Overall, we report the popularity of many features of the Stream API, including the analysis of an extensive selection of runtime metrics unique to streams. Moreover, we find inefficient streams affecting application code in open-source software projects, including supporting technologies behind two popular cloud platforms[56,62] and an operative cryptocurrency marketplace.[65] Finally, through the analysis of application code in software repositories, we located inefficient stream processing on a popular open-source library (24K+ stars) that is widely used for the automation of Web testing.[67] In the next section, we further discuss some implications of our findings.

6 | DISCUSSION

In this section, we discuss the implications of our work, which we group according to the target audience.

Developers of the Stream API. Our findings can provide feedback to the developers of the Java class library. In particular, we report the most used features of the Stream API, an information that could be considered when prioritizing future extensions and optimizations to the API. Complementary to prior work proposing alternatives or extensions to the Stream API to improve data processing in Java,[70-73] we suggest improving the Stream API by leveraging the knowledge that it has about both the pipeline structure and the characteristics of the data source upon stream execution.

Concretely, our findings on empty streams (see F5 and Section 5.3.1), suggest that it can be worthwhile for the implementation of the API to consider the data-source size upon stream execution. Provided that this size is equal to zero, in some cases (e.g., data reductions requiring multiple input elements) pipeline traversal could be fully avoided, with the benefit of mitigating overheads due to unneeded resource allocation. Moreover, we also find inefficient stream processing relying on the `sorted` and `distinct` intermediate operations (see Sections 5.3.3 and 5.3.4). In the current implementation of the Stream API, such operations are applied to the elements in the pipeline regardless on whether the data source is already sorted or distinct. When no operation in the pipeline can affect such conditions (e.g., mappings), the Stream API could query such data-source characteristics to avoid the unneeded stateful processing that `sorted` and `distinct` require, potentially enabling performance gains, particularly for parallel streams.[1]

Tool builders and researchers. Our findings on inefficient stream processing (see Section 5.3) points out the need for tools that help developers better code Java streams. We note that the automation of code assistance guiding Java developers to optimize stream processing in the form of approaches such as *optimization coaching*[74] or *automatic software repair*,[75-77] remains an open research path. We also provide a list of publicly available open-source software projects that heavily use stream processing (see Table 2), which could be considered as potential workload candidates for benchmarking by researchers and tool builders. Indeed, prior work[20,48] has shown promising results on using JUnit tests as workloads for benchmarking. Suitable workload candidates would use stream code from modern open-source applications, complementing benchmark suites made either from stream code adaptations of traditional MapReduce algorithms[78] or by converting workloads designed for relational databases to stream-based code.[79]

Educators. We find that Java developers seem to rarely make use of complex stream processing that truly benefits from the richness, versatility, and fluency of the Stream API. Indeed, our findings show that developers use mostly object-based sequential streams (F3, F11), often for the manipulation of collections (mainly lists and sets) (F1, F2), containing few elements (F5), relying on pipelines composed of few operations (F12) (mostly stateless operations enabling basic map-reduce-like patterns) (F6, F7, F18, F19), and typically performing simple data reductions (F14) whose output is also mainly stored in collections (F9, F13, F15) containing few elements (F17). Educators training Java developers

could see this as a motivation to better emphasize the learning of currently little exploited features of the Stream API, such that practitioners can do a well-supported decision making when implementing stream processing. For instance, the unpopularity of parallel streams, concurrent collectors, and the removal of ordering constraints (F4) (e.g., via `unordered`), reveals potential missed optimization opportunities, as leveraging such features can make an important difference on the goal of speeding up data processing in Java. Also, educators could further emphasize teaching best practices[3,10,53] that steer developers towards avoiding stream misuses, including the ones presented in this study (see Section 5.3.3).

Moreover, we find problematic stream processing that may have an impact on application performance (see Section 5.3). Academic material (e.g., books and tutorials) could more extensively discuss about stream-related performance issues (e.g., abstraction[25,80,81] and parallelization overheads[26,53] due to the use of streams), such that developers better leverage the Stream API and are particularly well-informed while deciding on whether involving streams in the implementation of performance-critical functionalities.

7 | THREATS TO VALIDITY

In this section, we discuss threats to validity to our work.

Like any large-scale empirical study, the validity of our findings depends on the set of analyzed projects, which may not be representative of the general use of Java streams. A study targeting proprietary codebases may lead to different conclusions. Nevertheless, the analyzed projects are diverse in aspects such as popularity, size and domain. Moreover, our findings show that some of the analyzed streams are used in consolidated open-source software projects.

The time frame selected for our study limits the number of projects that were analyzed. However, complementary to the filtering of projects based on their popularity, the chosen time frame enables us to analyze current practices in the use of streams, while still considering a large number of software projects.

Our study targets only GitHub. Nonetheless, it is currently the largest source-code-hosting facility.[28] Furthermore, GitHub provides an advanced search API,[42] enabling the crawling and filtering required for our project selection. Also, our study only considers projects that can be built via maven. However, it has been found that more than 76% of developers targeting the JVM use maven to build their applications,[82] a tendency that has been growing during the time frame chosen for our study.[82,83] As part of our future work, we aim at extending our analysis by considering projects hosted in large-scale repositories other than GitHub along with additional build systems.

Finally, an important limitation of our study is that it only targets source code exercised by unit tests. While we specifically target streams used in application code, the patterns analyzed via unit tests may not be fully representative of real usage scenarios of an application in production. In particular, our study is limited by the fact that unit tests may use a smaller input data than the one used in production scenarios, which may result in different conclusions (e.g., regarding the distribution of stream data-source sizes). However, we note that previous work[20,29,48] has shown that massively applying dynamic analyses on workloads exercised by unit tests can provide useful information, highlight coding practices, and derive statistics. We remark that we leverage unit tests because we aim at automatically running Stream-Analyzer on a multitude of software projects. Since JUnit tests can be automatically executed by the build system via simple commands (e.g., `mvn test`), they make large-scale dynamic analysis possible.

8 | RELATED WORK

In this section, we review work related to our approach. First, we compare our work to other studies focused on the use of streams (Section 8.1). Next, we review work addressing the optimization of streams (Section 8.2). Then, we discuss studies analyzing unit tests in open-source Java projects (Section 8.3). Finally, we review studies targeting functional programming in Java (Section 8.4).

8.1 | Empirical studies on the use of Java streams

To the best of our knowledge, there are only three studies examining the use of streams by Java developers, all of which mainly rely on static analysis and manual code inspection.

Tanaka et al.[9] mine 100 software projects to study the use of lambdas, streams, and the `Optional` class. They report that developers using such idioms mainly aimed at improving performance and producing short, clear, and readable code. Regarding streams, they find that the most popular operations are `map`, `filter`, and `collect`. Khatchadourian et al.[6] examine 34 projects to specifically study the use of streams in Java. The authors report several findings, including that parallel streams are not popular, that pipelines often have ordering constraints, and that streams are mostly used to iterate over collections and to perform data reductions. Finally, Nostas et al.[7] study the use of streams in 610 projects. Their work is a partial replication of the study of Khatchadourian et al. by considering a larger number of projects. The authors mainly confirm the results obtained by Khatchadourian et al., while also revealing commonly used sequential and parallel pipeline structures.

In comparison to the aforementioned related work, our study is conducted at a much larger scale thanks to the use of a fully automated approach that avoids manual intervention. Furthermore, we characterize runtime information that related work overlooks, because such studies are mainly based on static analysis techniques. Both the scale of our analysis and the profiling of key dynamic information enable us to confirm with more confidence the findings of related work regarding the usage of operations (F6, F7, F9, F10), execution modes (F11), collectors (F13, F14), and pipeline structures (F18, F19). Moreover, thanks to the novel perspective enabled by dynamic analysis, our work is the first to report both the popularity of many features of the Stream API and how developers use streams considering (both high- and low-level) source methods (F1), source collection types (F2), stream types (F3), the effectiveness of the predicate of filter operations (F8), pipeline lengths (F12), stream result types (F15, F16), stream result sizes (F17), and the analysis of creation and execution methods (F20, F21). As far as we are aware, we are also the first ones to analyze several characteristics of stream data sources as collected at runtime, including their size (F5), mutability, and support for concurrent modification, among others (F4). Finally, complementary to the study of Khatchadourian et al., which reports stream misuses by manually checking the history of git commits in 22 projects, we detect unsolved inefficient stream processing and stream misuses (see Section 5.3) present at the time of writing in several publicly available software projects.

8.2 | Analysis of Java streams

Some authors have addressed the analysis and optimization of streams. While helping developers spot stream-related drawbacks, the tools discussed above are not designed for large-scale dynamic analysis, as Stream-Analyzer does.

Ishizaki et al.[21] modify the IBM J9 JVM[84] and the Testarossa compiler[85] to translate streams into optimized GPU code. Hayashi et al.[22] extend this work, proposing a supervised machine-learning approach producing heuristics that the runtime can use to select between CPU or GPU to speed up parallel stream processing. However, both approaches target only parallel `IntStream` pipelines executed via `forEach`. Moreover, they require a proprietary compiler and a specific JVM implementation.

Khatchadourian. et al.[23,24] introduce an Eclipse plugin helping developers better code streams. The plugin evaluates whether it would be safe and potentially advantageous to restructure a stream pipeline to improve performance. To this end, the plugin infers how the execution of a stream would take place by analyzing it mainly via static analysis techniques.[86,87] As highlighted by the authors, their approach is unable to assess the optimization of a stream considering input size/overhead trade-offs.[24] Indeed, the plugin relies on inferences mainly done via static analysis that may not reflect the runtime behavior of the analyzed streams.

Other authors have focused on removing streams at a bytecode level, as stream-related overheads can sometimes be avoided by using semantically-equivalent imperative code. Møller et al.[25] present StreamLiner, a proof-of-concept bytecode-to-bytecode tool to transform sequential streams into more efficient imperative code. Due to limitations inherent to the static analysis techniques used, StreamLiner is not able to analyze streams whose creation and execution take place in different methods. Another important limitation of StreamLiner is that it cannot analyze parallel streams. Basso et al.[26] exploit high-level static analysis and bytecode-to-bytecode transformations to remove the abstraction overhead of parallel streams. In particular, their approach transforms parallel streams into corresponding imperative code that executes specialized fork-join tasks. Similarly to StreamLiner, their approach cannot optimize streams that are created and executed in different methods. However, this limitation may not greatly affect the applicability of both tools since, as shown in F21, we find that most of the detected streams are created and executed in the same method.

Finally, in our prior work,[27] we present StreamProf, a cycle-accurate stream profiler which is used to optimize sequential and parallel stream-based workloads from the Renaissance benchmark suite,[78] enabling significant speedups.

Differently from StreamProf, Stream-Analyzer targets multiple stream-related runtime metrics that can be collected with no significant perturbation, making them suitable to be profiled in the wild.

8.3 | Studies targeting unit tests in Java

Some authors have explored unit tests as a source of workloads to create benchmark suites. Zheng et al.[48] explore the feasibility of using unit tests available on open-source projects as workloads for custom benchmarks. They find more than 500 Java and Scala projects containing unit tests that can be considered as good candidate workloads for benchmarking purposes. Villazón et al.[20] conduct a large-scale study to analyze units tests available in open-source projects written in JavaScript (109,286), Java (25,918), and Scala (4076). They study the adoption of new constructs and find common bad coding practices in JavaScript. The authors also locate Java and Scala workloads which may be suitable to build task-parallel benchmarks. None of the aforementioned studies aim to dynamically analyze unit tests to characterize Java streams, which is the focus of our work.

Several empirical studies have analyzed unit tests from open-source projects for various purposes. Ma'ayan[88] conducts an analysis of the quality of unit tests in 128 Java projects and finds a significant difference between best practices in unit testing and actual practices. In particular, the authors report that 61% of the analyzed tests contain multiple assertions, and nearly 78% of the `assertTrue` and `assertFalse` assertions lack an error message. Petrić et al.[89] analyze the effectiveness of unit tests on 7 Java projects (out of a pool of 5508 candidates). They find that only about 30% of defective methods are covered by unit tests. Hilton et al.[90] analyze test coverage in 47 projects, 29 of which are written in Java. The authors find that relying only on *statement coverage* (computed as the ratio of statements executed by tests to the total number of statements) is inadequate in accurately assessing code testability and capturing the nuances of code evolution in large projects. In contrast to our work, these prior studies shed light on the characteristics of unit tests and prevailing practices. As part of our future work, we plan to explore ways to analyze aspects such as code coverage of unit tests in the analyzed projects. This will help us better assess the amount of application code that can be reached via the considered unit tests.

8.4 | Studies on functional programming in Java

Some work focused on functional programming in Java. Differently from our scope, they mostly investigate the use of lambdas.

Tsantalis et al.[91] study the use of lambdas to refactor duplicated code to benefit from *behavior parameterization*.[10] They find that lambdas are highly effective to avoid duplicated code. The study of Tsantalis et al. focuses solely on lambdas, disregarding streams. Mazinanian et al.[5] mine 241 software repositories containing over 100,000 lambdas, and survey 97 Java developers to understand how they are using lambdas. They find that developers are increasingly using lambdas to replace anonymous classes and for behavior parameterization. However, the study of Mazinanian et al. does not focus on streams. Nielebock et al.[92] study the use of lambdas in 2923 projects implemented in C#, C++, and Java. They locate several lambdas in both application and testing code. Similarly to our finding showing that parallel streams are not popular (F11), the authors find that developers tend to avoid using lambdas within concurrent code. Also the study of Nielebock et al. does not consider streams. Finally, Mehlhorn et al.[8] conduct a randomized controlled trial (RCT) to evaluate the understandability of collection operations performed on 20 participants, with response time and correctness as the dependent variables. The authors report that declarative code based on lambdas and streams to manipulate collections had a large, positive effect compared to the use of traditional loops significantly reducing the number of errors.

9 | CONCLUSIONS

Our work bridges the gap between the need for understanding how Java developers are using the Stream API and the lack of studies considering both a representative number of software projects and runtime metrics specific to streams.

9.1 | Summary of contributions

In this article, we present Stream-Analyzer, a novel DPA for collecting dynamic information that enable a fine-grained characterization of modern sequential and parallel stream processing on the JVM. To the best of our knowledge, Stream-Analyzer is the first tool enabling the large-scale characterization of stream processing. To this end, our tool allows the accurate detection of all streams used by an application running on the JVM. Moreover, Stream-Analyzer targets a relevant set of runtime metrics, whose analysis allows characterizing the behavior of stream processing on a wide and diverse selection of open-source Java workloads.

With Stream-Analyzer, we conduct the first large-scale empirical study on the use of Java streams. Thanks to our fully automated approach, we massively apply Stream-Analyzer to stream code exercised via unit tests available in software projects hosted on GitHub. We target 43,496 open-source software projects, which were last updated (at the time of analysis) between January 1, 2020 and April 30, 2022. We find 4063 projects using a total of 2,653,467 streams. We confirm at a large scale some of the findings of related work.[6,7,9] Moreover, our work reports new insights about the adoption of many features of the Stream API by Java developers and reveals inefficient stream-related practices currently affecting publicly available software projects.

9.2 | Future work

The work presented in this article opens up new research pathways. We plan to address the periodical inspection of code modifications in software hosting platforms to automatically report to the developers stream-related issues and potentially missed optimization opportunities, in the form of approaches such as automatic software repair.[75-77] Moreover, a continuous monitoring of stream processing in the commits performed across multiple software projects would enable the in-depth study of historical trends and the evolution of common stream code patterns and anti-patterns. Finally, we plan to release Stream-Analyzer as open-source software to facilitate the replicability of our results.

AUTHOR CONTRIBUTIONS

Eduardo Rosales designed and conducted the experiments. Eduardo Rosales and Matteo Basso designed the profiling methodology and implemented it. All authors performed the analysis of the results, contributed substantially to the writing of the manuscript, and performed multiple reviews of the article.

FUNDING INFORMATION

The research presented in this article was supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688).

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interests.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Large-scale Characterization of Java Streams at https://doi.org/10.5281/zenodo.7681472.

ACKNOWLEDGMENT

Open access funding provided by Universita della Svizzera italiana.

ORCID

Eduardo Rosales https://orcid.org/0000-0002-6404-3128

REFERENCES

1. Oracle. Package java.util.stream. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/Stream.html

2. Bird R, Wadler P. *An Introduction to Functional Programming*. 1st ed. Prentice Hall International (UK) Ltd.; 1988.
3. Bloch J. *Effective Java (2nd Edition) (The Java Series)*. 2nd ed. Prentice Hall PTR; 2008.
4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun. ACM*. 2008;51(1):107-113. doi:10.1145/1327452.1327492
5. Mazinanian D, Ketkar A, Tsantalis N, Dig D. Understanding the use of lambda expressions in Java. *Proc. ACM Program. Lang.* 2017;1(OOPSLA):1-31. doi:10.1145/3133909
6. Khatchadourian R, Tang Y, Bagherzadeh M, Ray B. An Empirical Study on the Use and Misuse of Java 8 Streams. Paper presented at: FASE. Springer. 2020:97–118. doi:10.1007/978-3-030-45234-6_5
7. Nostas J, Alcocer JPS, Costa DE, Bergel A. How Do Developers Use the Java Stream API? Paper presented at: ICCSA. Springer. 2021:323-335. doi:10.1007/978-3-030-87007-2_23
8. Mehlhorn N, Hanenberg S. Imperative versus Declarative Collection Processing: An RCT on the Understandability of Traditional Loops versus the Stream API in Java. Paper presented at: ICSE. ACM. 2022:1157–1168. doi:10.1145/3510003.3519016
9. Tanaka H, Matsumoto S, Kusumoto S. A study on the current status of functional idioms in Java. *IEICE Trans Inf Syst*. 2019;E102.D(12):2414-2422. doi:10.1587/transinf.2019MPP0002
10. Urma RG, Fusco M, Mycroft A. *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming*. 1st ed. Manning Publications Co; 2014.
11. Oracle. Class Optional<T>. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Optional.html
12. Gosain A, Sharma G. A Survey of Dynamic Program Analysis Techniques and Tools. In: FICTA. Springer. 2015:113–122. doi:10.1007/978-3-319-11933-5_13
13. Marek L, Villazón A, Zheng Y, Ansaloni D, Binder W, Qi Z. DiSL: A Domain-Specific Language for Bytecode Instrumentation. Paper presented at: AOSD. ACM. 2012:239–250. doi:10.1145/2162049.2162077
14. Rosà A, Rosales E, Binder W. Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM. Paper presented at: GPCE 2017. ACM. 2017:104–116. doi:10.1145/3136040.3136061
15. Rosà A, Binder W. Optimizing type-specific instrumentation on the JVM with reflective supertype information. *J Vis Lang Comput*. 2018;49:29-45. doi:10.1016/j.jvlc.2018.10.007
16. Rosà A, Rosales E, Binder W. Analyzing and Optimizing Task Granularity on the JVM. Paper presented at: CGO. ACM. 2018:27–37. doi:10.1145/3168828
17. Rosà A, Rosales E, Binder W. Analysis and optimization of task granularity on the Java virtual machine. *ACM Trans. Program. Lang. Syst.* 2019;41(3):1-47. doi:10.1145/3338497
18. Rosales E, Rosà A, Binder W. FJProf: Profiling Fork/Join Applications on the Java Virtual Machine. Paper presented at: ACM. 2020:128–135. doi:10.1145/3388831.3388851
19. Basso M, Rosales E, Schiavio F, Rosà A, Binder W. Accurate Fork-Join Profiling on the Java Virtual Machine. Paper presented at: EuroPar. Springer. 2022:35-50. doi:10.1007/978-3-031-12597-3_3
20. Villazón A, Sun H, Rosà A, et al. Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild. Paper presented at: ECOOP. ACM. 2019:20:1–20:27. doi:10.4230/LIPIcs.ECOOP.2019.20
21. Ishizaki K, Hayashi A, Koblents G, Sarkar V. Compiling and Optimizing Java 8 Programs for GPU Execution. Paper presented at: PACT. IEEE. 2015:419-431. doi:10.1109/PACT.2015.46
22. Hayashi A, Ishizaki K, Koblents G, Sarkar V. Machine-Learning-Based Performance Heuristics for Runtime CPU/GPU Selection. Paper presented at: PPPJ. ACM. 2015:27–36. doi:10.1145/2807426.2807429
23. Khatchadourian R, Tang Y, Bagherzadeh M, Ahmed S. A Tool for Optimizing Java 8 Stream Software via Automated Refactoring. Paper presented at: SCAM. IEEE. 2018:34-39. doi:10.1109/SCAM.2018.00011
24. Khatchadourian R, Tang Y, Bagherzadeh M, Ahmed S. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. Paper presented at: ICSE. IEEE. 2019:619–630. doi:10.1109/ICSE.2019.00072
25. Møller A, Veileborg OH. Eliminating abstraction overhead of java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 2020;4(OOPSLA):1-29. doi:10.1145/3428236
26. Basso M, Schiavio F, Rosà A, Binder W. Optimizing Parallel Java Streams. Paper presented at: ICECCS. IEEE. 2022:23-32. doi:10.1109/ICECCS54210.2022.00012
27. Rosales E, Basso M, Rosà A, Binder W. Profiling and optimizing java streams. *Art Sci Eng Program*. 2023;7(3):1-39. doi:10.22152/programming-journal.org/2023/7/10
28. GitHub. About GitHub. 2022. https://github.com/about
29. Rosales E, Rosà A, Basso M, et al. Characterizing Java Streams in the Wild. Paper presented at: ICECCS. IEEE. 2022:143-152. doi:10.1109/ICECCS54210.2022.00025
30. Oracle. Interface Spliterator<T>. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Spliterator.html
31. Oracle. Interface Stream<V>. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/Stream.html
32. Lea D. A Java Fork/Join Framework. In: JAVA. ACM. 2000:36–43. doi:10.1145/337449.337465
33. Oracle. Class ForkJoinPool. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ForkJoinPool.html
34. Oracle. Class Collectors. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/Collectors.html
35. Oracle. Interface BaseStream<T,S extends BaseStream<T,S≫. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/BaseStream.html
36. Kiczales G, Hilsdale E. Aspect-oriented programming. *SIGSOFT Softw. Eng Notes*. 2001;26(5):313. doi:10.1145/503271.503260

37. Binder W, Hulaas J, Moret P. Advanced Java Bytecode Instrumentation. Paper presented at: PPPJ. ACM. 2007:135–144. doi:10.1145/1294325.1294344
38. Kell S, Ansaloni D, Binder W, Marek L. The JVM is Not Observable Enough (and What to Do about It). Paper presented at: VMIL. ACM. 2012:33–38. doi:10.1145/2414740.2414747
39. Oracle. Java Virtual Machine Tool Interface (JVM TI). 2022. https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti
40. Oracle. The Java Virtual Machine Specification–Chapter 5. Loading, Linking, and Initializing. 2022. https://docs.oracle.com/javase/specs/jvms/se19/html/jvms-5.html
41. Marek L, Kell S, Zheng Y, et al. ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform. Paper presented at: ACM. 2013:105–114. doi:10.1145/2637365.2517219
42. GitHub. REST API. 2022. https://docs.github.com/en/rest
43. Borges H, Hora A, Valente MT. Predicting the Popularity of GitHub Repositories. Paper presented at: PROMISE. ACM. 2016. doi:10.1145/2972958.2972966
44. Al-Rubaye A, Sukthankar G. Scoring Popularity in GitHub. Paper presented at: CSCI. 2020:217-223. doi:10.48550/ARXIV.2011.04865
45. Costa D, Andrzejak A, Seboek J, Lo D. Empirical Study of Usage and Performance of Java Collections. Paper presented at: ICPE. ACM. 2017:389–400. doi:10.1145/3030207.3030221
46. The JUnit Team. JUnit. 2022. https://junit.org
47. The Apache Software Foundation. Apache Maven Project. 2022. https://maven.apache.org
48. Zheng Y, Rosà A, Salucci L, et al. AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses. Paper presented at: SANER. IEEE. 2016:639-643. doi:10.1109/SANER.2016.70
49. Docker. Docker. 2022. https://www.docker.com
50. Oracle. Class Arrays. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Arrays.html
51. Oracle. Interface IntStream<T>. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/IntStream.html
52. Oracle. Interface Supplier<T>. 2022. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/Supplier.html
53. Lea D, Goetz B, Sandoz P, Shipilev A, Kabutz H, Bowbee J. When to Use Parallel Streams. 2014. http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html
54. Shima H. WordNet Similarity for Java. 2022. https://github.com/dmeoli/WS4J
55. The Trustees of Princeton University. WordNet. 2022. https://wordnet.princeton.edu
56. Siri J. Amazon SQS Java Temporary Queue Client. 2022. https://github.com/awslabs/amazon-sqs-java-temporary-queues-client
57. Amazon Web Services. Amazon Simple Queue Service. 2022. https://aws.amazon.com/sqs
58. Hohpe G, Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc.; 2003.
59. Chan A. Client Ledger Core DB. 2022. https://github.com/andrewkkchan/client-ledger-core-db
60. Goetz B, Bosboom J. Does Stream.forEach respect the encounter order of sequential streams? 2015. https://stackoverflow.com/a/34253279
61. Shangchuan Y. Customized precise short text search service. 2022. https://github.com/ysc/short-text-search
62. Prodonovich J. Pub/Sub Emulator for Kafka. 2022. https://github.com/GoogleCloudPlatform/kafka-pubsub-emulator
63. Google. Google Cloud Pub/Sub. 2022. https://cloud.google.com/pubsub
64. The Apache Software Foundation. Apache Kafka. 2022. https://kafka.apache.org
65. Bustamante P. CryptoMarket-Java. 2022. https://github.com/cryptomkt/cryptomkt-java
66. CryptoMarket. CryptoMarket. 2022. https://www.cryptomkt.com/en
67. Software Freedom Conservancy. Selenium. 2022. https://www.selenium.dev
68. Software Freedom Conservancy. WebDriver Capabilities. 2022. https://www.selenium.dev/documentation/webdriver/capabilities
69. Guava. Class ImmutableMap<K,V>. 2022. https://guava.dev/releases/snapshot-jre/api/docs/com/google/common/collect/ImmutableMap.html
70. Mei H, Gray I, Wellings A. Integrating Java 8 Streams with The Real-Time Specification for Java. Paper presented at: JTRES. ACM. 2015:1-10. doi:10.1145/2822304.2822314
71. Biboudis A, Palladinos N, Fourtounis G, Smaragdakis Y. Streams a la carte: Extensible Pipelines with Object Algebras. Paper presented at: ECOOP. LIPIcs. 2015:591–613. doi:10.4230/LIPIcs.ECOOP.2015.591
72. Kiselyov O, Biboudis A, Palladinos N, Smaragdakis Y. Stream Fusion, to Completeness. Paper presented at: POPL. ACM. 2017:285–299. doi:10.1145/3009837.3009880
73. Ribeiro F, Ja S, Pardo A. Java Stream Fusion: Adapting FP Mechanisms for an OO Setting. Paper presented at: Brazilian Symposium on Programming Languages. ACM. 2019:30-37. doi:10.1145/3355378.3355386
74. St-Amour V, Tobin-Hochstadt S, Felleisen M. Optimization Coaching: Optimizers Learn to Communicate with Programmers. Paper presented at: ACM. 2012:163–178. doi:10.1145/2398857.2384629
75. Monperrus M. Automatic software repair: a bibliography. *ACM Comput. Surv.* 2018;51(1):1-24. doi:10.1145/3105906
76. Monperrus M, Urli S, Durieux T, Martinez M, Baudry B, Seinturier L. Repairnator patches programs automatically. *Ubiquity*. 2019;2019:1-12. doi:10.1145/3349589
77. Weimer W, Nguyen T, Le Goues C, Forrest S. Automatically Finding Patches Using Genetic Programming. Paper presented at: ICSE. IEEE. 2009:364–374. doi:10.1109/ICSE.2009.5070536
78. Prokopec A, Rosà A, Leopoldseder D, et al. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. Paper presented at: ACM. 2019:31–47. doi:10.1145/3314221.3314637

79. Schiavio F, Rosà A, Binder W. SQL to Stream with S2S: An Automatic Benchmark Generator for the Java Stream API. Paper presented at: GPCE. ACM. 2022:179–186. doi:10.1145/3564719.3568699
80. Biboudis A, Palladinos N, Smaragdakis Y. Clash of the Lambdas. 2014. doi:10.48550/arXiv.1406.6631
81. Kiselyov O, Biboudis A, Palladinos N, Smaragdakis Y. Stream fusion, to completeness. *SIGPLAN Not.* 2017;52(1):285-299. doi:10.1145/3093333.3009880
82. Snyk. JVM Ecosystem Report 2021. 2022. https://snyk.io/jvm-ecosystem-report-2021
83. Snyk. JVM Ecosystem Report 2020. 2022. https://snyk.io/blog/jvm-ecosystem-report-2020
84. Renouf C. *The IBM J9 Java Virtual Machine for Java 6*. Apress; 2009:15-34.
85. Grcevski N, Kielstra A, Stoodley K, Stoodley M, Sundaresan V. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. Paper presented at: VM. USENIX Association. 2004:12. https://dl.acm.org/doi/10.5555/1267242.1267254
86. Fink SJ, Yahav E, Dor N, Ramalingam G, Geay E. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 2008;17(2):1-34. doi:10.1145/1348250.1348255
87. WALA Team. WALA. 2019. http://wala.sourceforge.net
88. Ma'ayan DD. The Quality of Junit Tests: An Empirical Study Report. Paper presented at: SQUADE. ACM. 2018:33–36. doi:10.1145/3194095.3194102
89. Petrić J, Hall T, Bowes D. How Effectively Is Defective Code Actually Tested? An Analysis of JUnit Tests in Seven Open Source Systems. Paper presented at: PROMISE. ACM. 2018:42-51. doi:10.1145/3273934.3273939
90. Hilton M, Bell J, Marinov D. A Large-Scale Study of Test Coverage Evolution. Paper presented at: ASE. ACM. 2018:53–63. doi:10.1145/3238147.3238183
91. Tsantalis N, Mazinanian D, Rostami S. Clone Refactoring with Lambda Expressions. Paper presented at: ICSE. IEEE. 2017:60–70. doi:10.1109/ICSE.2017.14
92. Nielebock S, Heumüller R, Ortmeier F. Programmers do not favor lambda expressions for concurrent object-oriented code. *Empir Softw. Eng.* 2019;24(1):103-138. doi:10.1007/s10664-018-9622-9

How to cite this article: Rosales E, Basso M, Rosà A, Binder W. Large-scale characterization of Java streams. *Softw: Pract Exper*. 2023;53(9):1763-1792. doi: 10.1002/spe.3213