Telemetry of Legacy Web Applications: An Industrial Case Study

Anas Shatnawi, Bachar Rima, Zakarea Alshara, Gabriel Darbord, Abdelhak-Djamel Seriai, Christophe Bortolaso

Telemetry of Legacy Web Applications: An Industrial Case Study

Anas Shatnawi, Bachar Rima, Zakarea Alshara, Gabriel Darbord, Abdelhak-Djamel Seriai and Christophe Bortolaso

***Abstract*—Berger-Levrault, like many companies, has legacy web applications that still bring great values, and cannot be easily replaced. To maintain these applications, it needs data about user navigation, backend actions and client-server data exchange. Berger-Levrault has relied on a traditional logging approach that partially collects these data, requires modifying the application code and heavily impacts its performance. To address the limitations of this logging approach, we propose to replace it by a modern software telemetry approach. Existing telemetry approaches do not meet our needs, they should be extended based on our objectives, technological constraints and industrial regulations. In this paper, we report our experience in instrumenting real, large-scale, industrial legacy web applications based on a telemetry approach. Our goal is to automatically instrument legacy web applications to collect data fulfilling our industrial needs. We extend the automatic instrumentation capabilities of OpenTelemetry agents to instrument our applications without modifying their code. We define a telemetry architecture to integrate telemetry components with legacy web applications. Also, we empirically evaluate the performance overhead produced by our agents. The results show that there is no significant overhead when using OpenTelemetry agents. However, this overhead is sensitive to the size of data being serialized when instrumenting client-server data exchange. Moreover, we discuss lessons learned about the technical challenges we faced during the industrialization of our approach.**

***Index Terms*—Software telemetry, instrumentation, legacy software, industrial experience, GWT, Spring**

I. Introduction

LEGACY applications refer to applications that were built using outdated technologies, but still bring great value to their clients [1], [2]. Many companies possess critical legacy web applications that cannot be easily modernized or replaced with new ones [3]–[5]. As they rely on old technologies that are not supported anymore, we do not have advanced tools to help maintain them. Berger-Levrault [6], an international software publisher, is one of these companies that still has legacy web applications developed based on millions of lines of code using old or deprecated technologies such as Google Web Toolkit (GWT) [7] and the old Spring framework.

To help maintain these applications, our internal stakeholders (e.g., product managers, product owners, architects) need data about the frontend user navigation, the backend actions, the client-server data exchange, and the identity of the end-user initiating them. The frontend user navigation is needed to measure the user experience [8]. The backend actions are required to trace the flow of requests (e.g., REST, RPC, RMI) and their corresponding method invocations within these applications. This allows us to build a dependency call graph of how different components interact with each other, which can be used for various engineering tasks such as program understanding, change impact analysis, and debugging [9]. The client-server data exchange supports the creation of a corpus of input/output data of backend services, which is invaluable for test case generation [10]. Assuming that the current application behavior is correct and stable, this data serves as a reference point, capturing the expected behavior under the existing codebase. As our applications evolve, this data is a valuable resource for non-regression testing, ensuring that changes do not inadvertently disrupt existing functionality. End-user identities are crucial for identifying users who initiate specific actions within our applications. In the event of a problem, this helps maintenance teams to quickly pinpoint the affected user, facilitating support and issue resolution. It also makes it possible to attribute responsibility for those actions, which helps with auditing, compliance, and security purposes.

Berger-Levrault has relied on a traditional software logging approach (e.g., Log4J [11]) to partially collect some of this data at runtime. This traditional approach requires instrumentation code snippets to be injected into the application source code. When executed, these snippets generate execution logs that are stored in local files and then collected for later offline analysis. However, this logging approach suffers from four main limitations compared to the requirements of our stakeholders. First, it requires modifying the application source code. Second, it heavily impacts application performance due to the overhead of executing a large number of instrumentation code snippets. Third, it does not allow us to perform real-

Anas Shatnawi, Bachar Rima and Christophe Bortolaso are with the Direction of Research and Innovation at Berger-Levrault, France. Their email: anas.shatnawi@bergerlevrault.com, bachar.rima@berger-levrault.com, and christophe.bortolaso@berger-levrault.com.

Zakarea Alshara is with the Department of Software Engineering at Jordan University of Science and Technology, Jordan. Email: zmalshara@just.edu.jo

Gabriel Darbord is with Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France. Email: gabriel.darbord@inria.fr

Abdelhak-Djamel Seriai is with the Department of Software Engineering at University of Montpellier, France. Email: seriai@lirmm.fr

Manuscript submitted October 20, 2023.

time instrumentation because the logs are stored in local files on different servers. Fourth, it does not provide us with all data needed by our stakeholders (e.g., client-server data exchange).

To address these limitations, we propose to replace this traditional logging approach by a modern software telemetry approach. Telemetry provides a real-time instrumentation that allows practitioners to know how an application is currently behaving [12]. It allows us to collect and analyze all the data we need at runtime. Telemetry data is structured and includes various traces, metrics and logs related to system performance, resource utilization, user interactions, and other operational aspects. Telemetry allows us to aggregate data from multiple services and applications into a telemetry and observability backend (e.g., Elastic Observability [13]), where it can be processed, stored, indexed and visualized in dashboards, while also being subject to real-time alerts. This is important to provide holistic views and high-level overviews of the performance and behavior of multiple services and applications simultaneously. As such, telemetry is primarily used for proactive monitoring, enabling teams to detect and respond to real-time anomalous behavior as it occurs.

Together with our stakeholders, we define a set of telemetry objectives based on their needs. Our Functional Objectives (FOs) include collecting data about (FO1) the frontend user navigation, (FO2) the backend actions, (FO3) the client-server data exchange and (FO4) the identity of the end-user initiating them. Our Non-Functional Objectives (NFOs) include (NFO1) avoiding source code modifications and (NFO2) minimizing the performance overhead of the telemetry system.

However, when defining a telemetry approach for our legacy web applications based on these telemetry objectives, the following questions arise: What tools should be used to implement a telemetry platform properly? How do we extend them appropriately to serve our industrial needs? How to avoid modifying the application source code? How do we instrument a legacy frontend GWT code (i.e., Java source code compiled into executable JavaScript)? How do we store and manage the generated telemetry data? What is the incurred performance overhead of the telemetry system during the instrumented application's execution? What are the main challenges we might face when using telemetry in the industry?

Existing approaches do not meet our needs [14]–[16]. They do not collect the needed data of our objectives. They collect user navigation partially (only click events [14]), or only RPC calls [15] [16] (does not contribute to test automation). None of them trace the client-server data exchange based on the input/output of RPC services or user identity. Existing technologies should be extended based on our objectives, technological constraints and industrial regulations. There are no sufficient and systematic recommendations on how to extend telemetry approaches for industrial GWT-Spring legacy applications. We only identify technical documentations, tutorials and blogs that partially and indirectly explain artifacts related to our needs, e.g., a technical tutorial on how to extend the OpenTelemetry agent in general, but not specifically for our objectives and our application technologies.

In this paper, we report our experience in instrumenting real, large-scale, industrial legacy web applications based on a telemetry approach. Our goal is to automatically instrument our GWT-Spring applications (as instances of our legacy applications) to generate data allowing us to fulfill our industrial telemetry objectives. Automatic instrumentation refers to the process of automatically generating telemetry data from our applications without injecting manual and explicit instrumentation code in the application source code. It is therefore an efficient alternative to manual instrumentation, which proves to be time-consuming and error-prone. Among existing telemetry instrumentation agents, we select OpenTelemetry agents because they are open-source and can be extended to serve our telemetry objectives. Hence, we extend and adapt their automatic instrumentation capabilities to instrument our GWT-Spring applications without modifying the application source code (NFO1). On one hand, the JavaScript agent is adapted to trace the frontend user navigation (FO1) and the corresponding end-user identity (FO4). On the other hand, the Java agent is extended to trace the backend actions (FO2) and the client-server data exchange (FO3), including their corresponding end-user identities (FO4). Additionally, we define a telemetry architecture to show how to integrate these agents with our GWT-Spring applications and other telemetry components.

Furthermore, we empirically evaluate the performance overhead of our telemetry agents by comparing the response time of the GWT-Spring application with and without running these agents. The results show that there is no significant performance overhead when using OpenTelemetry agents (NFO2). However, we find that this overhead is sensitive to the size of the serialized data when instrumenting client-server data exchanges. Therefore, we should be careful when using it with large objects.

During the industrialization of our telemetry approach, we tackle technical challenges related to data serialization (large objects, lazy loading of properties and nested Data Transfer Objects), Cross-Origin Resource Sharing and manual instrumentation of GWT frontend code using JavaScript Native Interface.

The main contributions of this paper include:

- We extend the automatic instrumentation capabilities of OpenTelemetry agents to serve our functional and non-functional industrial objectives for instrumenting legacy GWT-Spring applications.
- We conduct an empirical evaluation to measure the performance overhead produced by these agents.
- We test some existing telemetry technologies to demonstrate their usability for our context.
- We report telemetry experiences based on real, large-scale, industrial applications. Moreover, we report some guidelines for practitioners on the implementation of a telemetry solution in industrial settings.

This paper is organized as follows. In Section II, we discuss telemetry concepts and our proposed telemetry architecture. Next, Section III presents details about the experimental setup of our telemetry system for our industrial applications. Then, we measure the application performance overhead resulting from the use of our telemetry agents in Section IV. Next, we discuss the lessons learned in Section V. Related works are analyzed in Section VI. Finally, we conclude the paper and draw our future directions in Section VII.

II. Telemetry: Concepts & Architecture

Telemetry is the on-site collection of measurements (*or other data*) at remote points, using agents and collectors, and their automatic transmission to receiving equipment for monitoring and analysis [17]. In the remainder of this section, we discuss telemetry data and the architecture of our proposed telemetry system.

A. Telemetry Data

Telemetry data can be harnessed to make an application observable by increasing the visibility of its internal behavior to the outside world. Due to its crucial value, many well-known companies such as Microsoft, Google, Facebook, Twitter, Cisco, etc. are keen on capturing telemetry data, which can be continuously tracked in real-time, often via dashboards and alerts, to provide insights about the instrumented applications.

Telemetry data covers a wide range of data types that can be used to capture information about the system at different levels of abstraction and in different formats, depending on the desired telemetry objectives. In the context of our telemetry objectives, we are interested in three foundational telemetry data types: traces, metrics and logs. These telemetry data types cover the primary categories of data typically used in monitoring and observability. However, in some specialized contexts or specific industries. additional telemetry data types that are tailored to unique requirements may be encountered such as snapshots, profiles, flow data and others.

1) Traces: A *trace* captures everything that happens between an initial request (e.g., an HTTP request) and its returned response. Figure 1 shows an example of a trace generated for an HTTP request. Essentially, it records time and metadata about operations that take place throughout the lifecycle of a request. Traces establish causal chains of events to determine relationships between different entities in the system.

To get a micro view of what happens within traces, each trace is composed of a set of *spans*. In our example in Figure 1, the trace consists of six spans that describe what happens as a result of an HTTP request. A *span* represents a discrete unit of work that is tracked within a trace, such as a remote procedure call or a database query. A trace is therefore a series of causally related and potentially nested spans that cover an end-to-end request within a system. For example, if a system receives

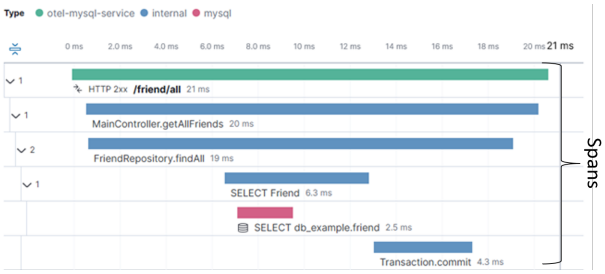

Fig. 1. An example of a trace composed of six spans

a request to retrieve a user account from a database, the trace might include a span for the initial request, a span for the database query to retrieve the account, and a span for the response to the original request. Each span would have a unique identifier and could include data such as the duration of the operation, any associated metadata thereof, and any errors or issues that may have occurred. Nested spans provide a hierarchical view of the work being performed, allowing us to see the relationships between different operations performed to fulfill the traced request.

2) Metrics: Metrics are measurements collected at regular intervals. Typically, a metric has a timestamp, a name, one or more numeric values, and a count of how many events it represents. In this context, an event is a discrete action that happens at any given time. Furthermore, adding metadata to events makes them much more powerful. Examples of metrics include error rate, response time, CPU mean usage, etc. Metrics can harness the power of mathematical modeling and prediction to derive knowledge about the behavior of a system over time intervals in the present and future.

3) Logs: A log entry consists of a textual timestamped description of an event. It also comprises the dynamic execution context and the criticality (*e.g., trace, debug, info, warn, error, fatal*) of the recorded event. Its content can be structured (*recommended*) or unstructured. Log entries are collectively referred to as logs. To generate such logs, we can resort to one or more logging libraries with distinct features and that can be used with different programming languages. Although logs are a standalone data source, they can also be associated with spans. For example, OpenTelemetry considers as logs all data that does not fall under the category of traces or metrics.

B. Telemetry Architecture

To better understand its structure, the first step in building a telemetry platform is to define its architecture. Figure 2 shows the telemetry architecture in relation to the target legacy web applications and how they interact with each other. The architecture consists of nine components that are deployed in four different locations. These are *Web Browser, Frontend App, Backend App, Database, Frontend Agent, Backend Agent, Telemetry Collector, Telemetry Storage and Analytics, and Visualization Tool.*

Frontend Agent and *Backend Agent* are software components that collect telemetry data consisting of traces,

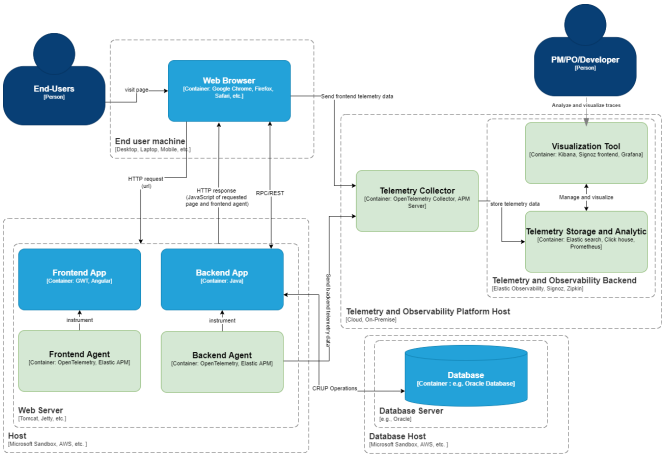

Fig. 2. The telemetry architecture

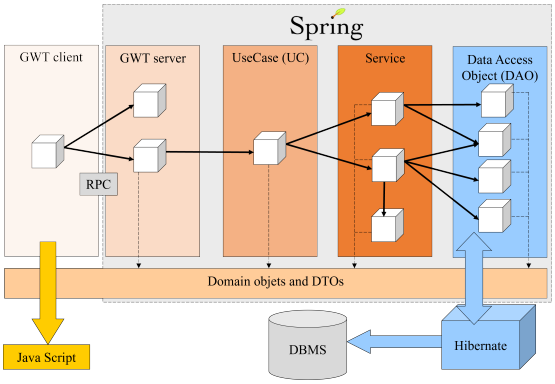

Fig. 3. The architecture of our GWT-Spring applications

metrics and/or logs about the performance, usage, and behavior of our frontend and backend applications.

End users use *Web Browser* to access Web pages via their URLs. When a page is requested, *Web Browser* sends an HTTP request to *Web Server* that's hosting the web application. Then, *Web Server* returns a corresponding HTTP response to *Web Browser*. This response encapsulates the executable JavaScript code of the requested page, which is sent along with *Frontend Agent*, both of which are executed by *Web Browser*. This allows *Frontend Agent* to send telemetry data about events that happened at the frontend (e.g., user navigation) to *Telemetry and Observability Platform* via its *Telemetry Collector*.

The frontend JavaScript code executed on *Web Browser* relies on RPC or REST to communicate with *Backend Application* to retrieve data, perform actions, or invoke server-side functionalities. For example, RPC allows the client-side code to remotely call server-side Java methods based on a set of Java interfaces and shared Data Transfer Object (DTO) classes. When an RPC service is requested, *Backend Application* executes the Java methods associated with that RPC service. Meanwhile, *Backend Agent*, running on the server-side, sends telemetry data about events that happened on the backend (e.g., RPC calls, SQL queries) to *Telemetry Collector*.

Telemetry Collector serves as an intermediary between the agents of instrumented applications and *Telemetry and Observability Backend*. Its primary function is to collect telemetry data from various agents and store it in *Telemetry Storage and Analytics* of our *Telemetry and Observability Backend*. Within this *Telemetry and Observability Backend*, *Visualization Tool* acts as the frontend, allowing stakeholders such as Product Managers (PMs), Product Owners (POs) and Developers to explore telemetry data through features such as dashboards and alerts.

III. Telemetry at Berger-Levrault

In this section, we discuss the experimental environment of our proposed telemetry approach for legacy web applications at Berger-Levrault. First, we present the target case study GWT-Spring applications. Then, we discuss the selection and extension of telemetry agents for the frontend and backend applications. Finally, we report on our experience in selecting a telemetry collector as well as a telemetry and observability backend.

A. Industrial Case Study of GWT-Spring Applications

Our experiment is based on three real, large-scale, industrial GWT-Spring applications that are currently used by many clients for their human resources and financial management systems. Due to internal company policy, we cannot disclose their names or semantics.

It is important to understand the architecture of these applications in order to instrument them. As shown in Figure 3, the architecture of our applications is based on five layers, as follows. *GWT client* refers to the frontend application, which is responsible for presenting the graphical user interface and handling user interactions. The client-side components and graphical user interface widgets are written in Java code, which is then compiled into executable JavaScript to run in the client web browser. *GWT server* connects the GWT client layer with the other business logic layers. Communication with the GWT client is realized through RPC calls, where data is serialized and exchanged. *UseCase (UC)* manages and orchestrates the business logic and workflow of user requests. This layer is responsible for defining and executing appropriate services or business logic components that correspond to the user requests. *Service* houses the core business logic of the application, where essential business operations are performed. A service is responsible for performing specific business tasks. *Data Access Object (DAO)* is the data access layer that acts as a bridge between the service layer and the database. It handles data persistence and retrieval from the database using Data Access Objects (DAOs). This layer communicates with Hibernate to store/retrieve the data into/from the actual database.

Table I shows the size of our case study applications. For each application, we provide the number of lines of code, Java classes, Java methods and Java attributes that make up its GWT frontend and Spring backend applications,

TABLE I
COMPLEXITY METRICS ABOUT OUR CASE STUDY APPLICATIONS

Applications		# LOC	# Classes	# Methods	# Attributes
App 1	Frontend	31325	408	2737	1713
	Backend	31511	418	3536	1208
App 2	Frontend	848330	6926	68833	43974
	Backend	861814	8597	111514	44427
App 3	Frontend	793363	6175	69044	39794
	Backend	894029	7909	88200	40594

respectively. These values provide a comprehensive view of the size and complexity of the applications.

In terms of technology, the applications are based on Open Java Development Kit (OpenJDK) version 1.8u212. The frontend and the backend applications are developed using the GWT and Spring frameworks, respectively. The database is based on Oracle 12c. Hibernate is used for server-side data persistence and management.

B. Telemetry Agents

As mentioned earlier, our case study applications rely on the GWT and Spring frameworks for their frontend and backend, respectively. The frontend source code is written in Java, compiled into JavaScript, and executed in the client browser. Therefore, instrumenting the GWT frontend with a Java agent is not straightforward, as Java agents are primarily focused on instrumenting applications running natively in the Java Virtual Machine (JVM) [18]. Thus, we rely on a JavaScript agent to instrument the frontend applications. In contrast, since the backend applications are based on Java source code compiled into native bytecode, we use a Java agent to instrument them.

Among the existing telemetry instrumentation agents, such as those provided by Elastic APM and Spring Cloud Sleuth, we select agents provided by OpenTelemetry. Our choice is motivated by the following criteria:

Serving our telemetry objectives: OpenTelemetry supports the instrumentation of our legacy web applications, i.e., GWT and Spring, through its JavaScript and Java agents, respectively. These agents provide extensible *automatic* instrumentation capabilities to collect telemetry data, i.e., traces, that serve our objectives. This allows them to be integrated into our web applications without modifying their source code. This is achieved, for example, by adding end-user identity data, and by specifying which data to retain and discard during collection. The latter technique allows us to control the performance overhead.

Compatibility with other tools: OpenTelemetry agents can be seamlessly integrated to export telemetry data to any Telemetry and Observability Platform, regardless of its implementation details, through an OpenTelemetry Collector. This is crucial to grant the user the flexibility of working with any existing telemetry and observability platform, e.g. Elastic Observability, SigNoz, Jaeger, Zipkin, Prometheus.

Support and accessibility: OpenTelemetry agents are *open-source*, which allows us to understand their implementation and extend their capabilities. They are also documented and supported by a large open-source community. For instance, the OpenTelemetry GitHub account consists of 68 repositories covering about 18 main projects, collectively supported by a team of 193 contributors.

Credibility: OpenTelemetry agents are widely used in the industry [19] such as IBM [20], Ericsson [21], Splunk [22], and DoorDash Engineering [23].

In the following, we discuss the selected OpenTelemetry JavaScript and Java agents and how they are extended to collect traces fulfilling our telemetry objectives.

1) Frontend Instrumentation Agent:

a) Objectives: The goal of the frontend instrumentation is to automatically collect traces related to user navigation (FO1) and end-user identity (FO4).

b) Methodology: The frontend instrumentation is realized at the *GWT Client* layer of the architecture of our GWT applications, illustrated in Figure 3. Hence, we adapt the OpenTelemetry JavaScript agent to automatically instrument the compiled JavaScript code of our GWT frontend applications.

For FO1, the idea is to trace the user navigation behavior of our frontend GWT applications based on user interactions with their graphical user interface elements (e.g., clicking on a button, filling out a form, etc.). Specifically, this is achieved by relying on the automatic instrumentation APIs[1] provided by the OpenTelemetry community. Accordingly, based on a list of desired events (*e.g., click, submit, select, scroll, toggle, etc.*) [24], we could generate and collect traces related to user navigation. In addition, to achieve FO2, we extend the instrumentation capabilities of our agent to automatically add the end-user identity to each generated trace.

All of our collected frontend traces conform to a schema consisting of mandatory and optional properties [25]. These properties are either *trace-level*, describing the global context of the request, or *span-level*, describing the context of an operation captured by a span within a trace. In addition, sometimes a trace can consist uniquely of a single span. For example, clicking on a clickable element in our GWT application automatically generates a corresponding trace consisting of a single span.

This span is described by span-level properties, including the span ID, the trace ID of the encompassing trace, the span name (which identifies the "click" event), the span timestamp, the URL where the event was triggered, the target HTML element receiving the event (i.e., the element), and others.

c) Implementation of Automatic Instrumentation: The automatic instrumentation of the JavaScript agent is based on intercepting calls from target functions, to capture relevant telemetry data, such as the start and end times of an operation, any relevant attributes, and error information if an error occurs. This relies on several mechanisms such as wrapping interfaces, subscribing to system-specific callbacks, or translating system-specific telemetry into the OpenTelemetry model under the hood [18].

[1]https://www.npmjs.com/package/@opentelemetry/auto-instrumentations-web

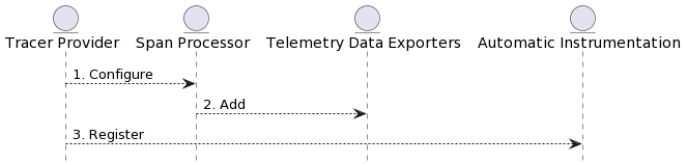

Fig. 4. The sequence diagram of the process of implementing automatic instrumentation for our frontend GWT applications

The actual definition of the agent's instrumentation code is carried out according to the following three-step process: (1) configuring a tracer provider, (2) adding a data telemetry exporter, and (3) registering the tracer provider for automatic instrumentation [25]. Figure 4 shows the sequence diagram of these three steps and the involved instrumentation API elements. Configuring a tracer provider starts with creating an instance of `TracerProvider` and attaching a `Resource` object to it with the name and the instance of the GWT application's service, as shown in Listing 1.

```
... name = "<service-name>";
id = "<service-instance-ID>";
... tracerProvider = new WebTracerProvider({
    resource: new Resource({
        [Resource.SERVICE_NAME]: name,
        [Resource.SERVICE_INSTANCE_ID]: id,})});
```

Listing 1. Configuring the OpenTelemetry tracer provider

Next, any built-in or custom processing of the traces' spans is further specified. Built-in span processors include a simple span processor and a batch span processor [26]. The simple span processor passes spans to the configured span exporter as they are completed, while a batch span processor passes them in batches after a specified delay. To customize span processing, the API can be extended such that any attribute(s) can be added/removed from the generated spans, depending on the desired objectives. For instance, in Listing 2 we extend the `SpanProcessor` interface [26] to define a custom span processor that encapsulates the recording of user session data, in accordance with our objectives.

```
export class SessionIdSpanProcessor implements
     SpanProcessor {
    onStart(span: Span, parentContext: Context): void {
        span.setAttribute('app.session.id',
        SessionGateway().getSession().sessionId); this.
        _nextProcessor.onStart(span, parentContext);
        }...
```

Listing 2. Defining a custom span processor to add user session data to the generated traces

Afterwards, telemetry data exporters are created and added to the tracer provider's configuration to indicate where the generated data will be sent [26]. We create an exporter to send the generated telemetry data to an OpenTelemetry Collector, as shown in Listing 3.

```
... const collectorExporter = new OTLPTraceExporter({
    url: '<collector_url>/v1/traces'
});
tracerProvider.addSpanProcessor(new
     SessionIdSpanProcessor(new SimpleSpanProcessor(
     collectorExporter)));
tracerProvider.register();...
```

Listing 3. Defining and registering OpenTelemetry exporters with the created tracer provider

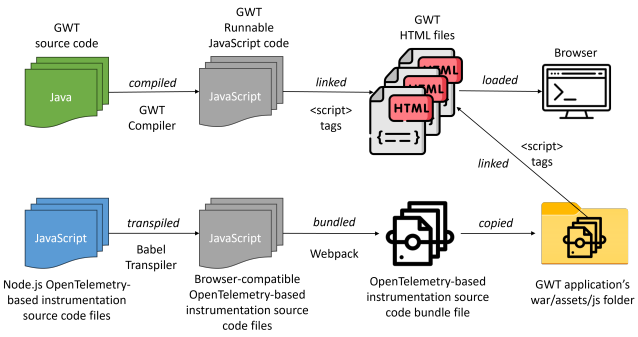

Fig. 5. The workflow for building and running our frontend agent with our GWT applications

Finally, as previously mentioned, automatic instrumentation is configured and registered with the tracer provider using the modules provided by the OpenTelemetry community's *contrib* packages for automatic instrumentation.

d) Run the Agent with Our Applications: Building and running the frontend agent with our GWT applications follows the process illustrated in Figure 5. The process starts with the definition of the instrumentation files, as described earlier in Figure 4. Then, since the OpenTelemetry JavaScript agent's API is defined in Node.js [27], which is natively incompatible with the browser, the next step consists of transpiling the defined instrumentation Node.js code into browser-compatible JavaScript. This is realized accordingly using Babel [28]. Afterwards, we bundle all the transpiled JavaScript instrumentation files and their dependencies into a single JavaScript bundle produced by webpack [29].

Subsequently, the generated bundle is copied into the GWT application's `war/` folder, from which it gets linked into its main HTML template through a `<script>` tag, then loaded and executed. As a result, the scope of the agent spans the entire application and generates telemetry data throughout its execution.

2) Backend Instrumentation Agent:

a) Objectives: The goal of the backend instrumentation is to collect traces related to actions performed at the backend (FO2), the client-server data exchange (FO3), as well as end-user identity (FO4).

b) Methodology: The backend instrumentation is concerned with the *GWT server*, *UseCase*, *Service* and *DAO* layers of our application architecture in Figure 3. For FO2, the idea consists of understanding the inner workings of the actions performed on the different layers. This is done by tracing the HTTP requests received by the GWT server (e.g., through RPC calls) and the corresponding use-cases, services, and DAOs executed on their respective layers. For FO3, we instrument the methods of the Java classes at the *UseCase* layer, based on their originating RPC calls, in order to obtain their arguments and return values. For FO4, we rely on the Spring security API to recover the end-user identity.

To achieve these objectives without modifying the application source code, we extend the automatic instru-

mentation capabilities of the OpenTelemetry Java agent. In particular, we propose two extensions: the *Program Understanding* and *Data Exchange* extensions. *Program Understanding* generates traces about the inner workings of the backend. It creates an initial span for each HTTP request received by the *GWT Server* layer. To capture the flow of this request and its method invocations throughout the application, this span is nested with a set of children spans corresponding to other sub-actions performed at the *UseCase*, *Service*, and *DAO* layers as a result of the initial HTTP request. In other words, each child span contains information about the Java methods executed at the corresponding layer. The *Data Exchange* extension generates traces about the RPC calls mapped to the *UseCase* layer. The data is captured by serializing the Java objects used as method arguments and return objects and attaching them to the traces as JSON.

In addition, traces generated by both extensions conform to a structured schema similar to that of the traces generated by the frontend agent. This schema includes default mandatory properties introduced by OpenTelemetry to capture span-level and trace-level names, IDs, statuses, etc. Moreover, the schema provides trace-level properties that record useful information about the execution environment, such as the host name, the JVM version, and the application process ID. Finally, both extensions make the agent add span-level properties that capture the identifiers of the executing method and its class, as well as the identity of the user who sent the request.

c) Implementation of Automatic Instrumentation: The idea is to extend the automatic instrumentation capabilities of the agent by informing it to augment the collected traces with additional properties that capture the data related to each functional objective. The OpenTelemetry community defines a mechanism[2] based on the Service Provider Interface (SPI) design pattern that enables the dynamic discovery and integration of extensions into the agent. This allows us to specify the instrumentation logic (e.g., capturing method arguments) to be automatically injected by the agent into the desired parts (e.g., the UseCase layer) of the application.

To inform the agent that we want to inject instrumentation logic into some parts of the application, we extend the `InstrumentationModule` abstract class, which corresponds to a service provider interface, and register the subclass as a service provider. By overriding the methods of the SPI, we can define the name of the extension and associate it with one or more `TypeInstrumentation` implementations. Then, we implement the `TypeInstrumentation` interface. The implementation matches the methods that we want to instrument using element matchers provided by the framework (e.g., match types in the *UseCase* layer using package naming conventions: `nameContains("usecases.")`, then match all public methods). It also defines the instrumentation logic, such as adding attributes to the current span (e.g., `span.setAttribute("uc.method", methodName)`), which is executed along with the instrumented methods.

d) Run the Agent with Our Applications: We package the implementation of each extension as an executable JAR file by using the Gradle script provided by OpenTelemetry. To start automatically instrumenting our applications, we integrate the OpenTelemetry agent and our extensions with the Java Virtual Machine (JVM). This is done by passing the paths of their JAR files as arguments to the JVM options `-javaagent` and `-Dotel.javaagent.extensions` respectively. We can further configure the agent by providing additional JVM options as needed. These options include specifying the service name (`-Dotel.resource.attributes=serviceName`), specifying the target collector to which traces are sent (`-Dotel.exporter.otlp.endpoint`), and other relevant options to achieve our goals.

C. Telemetry Collector

As mentioned earlier, telemetry collectors are used to collect telemetry data generated by the telemetry agents in the instrumented applications, and to export this data to one or more telemetry and observability backends.

In our context, we experimented with two popular telemetry collectors, namely OpenTelemetry Collector and Elastic APM Collector. We experiment with OpenTelemetry Collector mainly because of OpenTelemetry's widespread adoption and reputation in the telemetry and observability industry [30]. Moreover, we experiment with Elastic APM Collector because we chose Elastic Observability as our telemetry and observability backend, which includes it as part of its stack [13].

OpenTelemetry Collector [31] is an example of an independent telemetry collector. It provides flexible, customizable, and vendor-agnostic telemetry data collection, processing, and export capabilities. This allows one to ingest different types of telemetry data, including traces, metrics, and logs, from various data sources, in different formats, and using different data exchange protocols (e.g., http/protobuf, grpc). In addition, OpenTelemetry Collector provides compatibility with several telemetry and observability backends, such as Elastic Observability [13], SigNoz [32], Zipkin [33].

In contrast, Elastic APM Collector [34] is an example of an integrated telemetry collector, as it is designed to work seamlessly with the Elastic ecosystem, where it integrates directly with its Elasticsearch [35] component to store and analyze telemetry data. This data is captured and transmitted by native Elastic APM agents embedded in the code of the instrumented application.

Based on our experiments, we decided to adopt OpenTelemetry Collector as it collects telemetry data regardless of its size and origin. Its policy matches the size of our traces, which can contain data longer than 1024 characters. This is in contrast to Elastic APM Collector,

[2]https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/docs/contributing/writing-instrumentation-module.md

TABLE II
COMPARISON OF TELEMETRY AND OBSERVABILITY BACKENDS: ELASTIC OBSERVABILITY, SIGNOZ, AND ZIPKIN

Feature	Elastic Observability	SigNoz	Zipkin
Availability	Open-source (*main features*)	Open-source (*main features*)	Open-source
OpenTelemetry compatibility	Yes	Yes	Yes
Leveraged telemetry data	Logs, metrics, traces	Logs, metrics, traces	Traces
Storage	Elasticsearch	ClickHouse	Cassandra, Elasticsearch or MySQL
Processing	• Ingests, aggregates, indexes, and processes telemetry data in real-time, ensuring data freshness • Supports parsing, transforming, and enriching data for deeper analysis	• Ingests, aggregates, indexes, and processes telemetry data in real-time, ensuring data freshness • Supports parsing, transforming and enriching data for deeper analysis	• Ingests, aggregates, indexes, and processes traces only in real-time, ensuring data freshness
Analysis	• Utilizes machine learning and AI models for advanced analysis, including anomaly detection and root cause analysis • Provides deep insights through application performance, infrastructure, real user, and synthetic monitoring • Allows performance optimization through universal profiling and in-depth analytics	• Monitors low-level metrics like request rates, error rates, and request duration for performance analysis • Provides deep insights through application performance, infrastructure, real user, and synthetic monitoring	• Offers detailed insights into service request flows and latency within distributed systems
Visualization	• Utilizes Kibana • Provides a wide range of built-in and customizable visualization options, including charts, maps, graphs, dashboards, and service meshes	• Utilizes SigNoz's frontend • Provides a wide range of built-in and customizable visualization options, including charts, maps, graphs, dashboards, and service meshes	• Utilizes Zipkin UI • Offers basic visualization features for analyzing traces

which is limited to 1024 characters. Furthermore, because of its support for multiple telemetry and observability backends, OpenTelemetry Collector allows us to switch to any telemetry and observability backend without affecting the other telemetry components (i.e., we only need to change the URL of the telemetry export target in the collector configuration). Meanwhile, Elastic APM Collector is designed to work exclusively with the Elastic ecosystem.

To facilitate the integration of many services/applications with our telemetry system, we have deployed an OpenTelemetry Collector instance on Amazon Web Services (AWS) [36] where each generated trace is exported. The frontend and backend agents use the OpenTelemetry Protocol (OTLP) [12], a data formatting protocol, to structure and serialize the exported telemetry data, and HTTP to transport it. Upon receipt, we configured the collector to forward the collected telemetry data to our *Telemetry and Observability Backend*, also hosted on AWS.

D. Telemetry and Observability Backend

Telemetry and observability backends are used to store, analyze, and visualize the application execution traces collected by our frontend and backend agents. We test three main telemetry and observability backends, namely Elastic Observability [13], SigNoz [32], and Zipkin [33]. Table II compares these tools based on their availability, OpenTelemetry compatibility, leveraged telemetry data, storage components, supported processing, analysis and visualization capabilities.

We selected these tools because they share a foundation of open-source availability and compatibility with OpenTelemetry Collector for seamless integration with telemetry data sources. In addition, they have a large community support [37] [38]. They also collect logs, metrics, and traces while ensuring real-time processing for data freshness, except Zipkin which is limited to traces only.

Nonetheless, the differences between them are noteworthy. Elastic Observability offers prominent analytic capabilities that leverage machine learning and AI models to provide advanced insights across various domains, including application performance, infrastructure monitoring, real user and synthetic monitoring, and universal profiling for performance optimization. In contrast, SigNoz, while supporting similar data types, focuses primarily on low-level metric analysis, while also covering application performance, infrastructure, real user and synthetic monitoring. Meanwhile, Zipkin is mainly tailored for tracing, providing deep insight into service request flows and latency in distributed systems.

In terms of storage, Elastic Observability relies on Elasticsearch, SigNoz relies on ClickHouse, and Zipkin offers flexibility with support for Cassandra, Elasticsearch, or MySQL. In the realm of visualization, Elastic Observability uses Kibana, which offers a wide range of advanced options, including charts, graphs, maps, and service mesh visualization. SigNoz's frontend application similarly offers robust visualization features, while Zipkin's visualization capabilities are rudimentary.

Ultimately, our decision to use Elastic Observability was driven by the maturity of the Elastic ecosystem and our developers' familiarity with it. Consequently, we established a free and open-source Elastic Observability instance on AWS.

IV. Performance Overhead Evaluation

A. Overview

Our objective is to measure the performance overhead incurred by using our telemetry agents to instrument the frontend and backend of our legacy application. We measure this performance overhead by answering four research questions defined in Section IV-D. The methodology for answering these research questions is based on running the application based on three execution use case scenarios defined in Section IV-C and evaluating the application performance based on five metrics related to the response time demonstrated in Section IV-B. The obtained results are discussed in Section IV-E and the threats to their validity in Section IV-F.

B. Evaluation Metrics

To measure the performance overhead of our telemetry agents, we rely on the response time and the performance overhead metrics as follows.

1) Response Time Metrics: The *Response Time* (RT) metric is the time taken by the web server to respond to a request from the web browser. We rely on the RT because it has an inverse correlation with the user experience and the overall performance of web applications. For example, a faster RT results in a more seamless and satisfying user experience. We measure the RT in milliseconds (ms). To better understand the RT of multiple requests made at different execution scenarios, we rely on:

1) The **minimum** RT, which refers to the fastest HTTP request.
2) The $\mathbf{50}^{th}$, $\mathbf{75}^{th}$, $\mathbf{95}^{th}$, **and** $\mathbf{99}^{th}$ **PCTs** that compare the X^{th} RT in relation to others, e.g., if $\mathbf{50}^{th}$ percentile is 3 ms, it means that 50% of responses are <= 3 ms.
3) The **maximum** RT, referring to the slowest HTTP request.
4) The **mean**, referring to the average RT.
5) The **standard deviation**, referring to how much the RTs differ from the mean.

2) Performance Overhead Metric: To measure the additional performance overhead introduced by our telemetry agents, we rely on the *Percentage Difference* metric [39]. It measures the differences in the RTs of a given agent compared to the ground-truth, as follows.

$$Overhead = (\frac{RTAgent - GroundtruthRT}{GroundtruthRT}) * 100 \quad (1)$$

Where *RTAgent* and *GroundtruthRT* refer to the RT values obtained when running the application with a given agent and a ground-truth RT value respectively. Overhead values are constrainted within the range [*-1, 1*], where *0* signifies the absence of any overhead caused by the agent, positive values indicate the extent of performance overhead caused by that agent, and negative values indicate an improvement in application performance upon running the agent with the application.

C. Execution Scenarios

To better generalize the results, we generate different workloads based on three execution scenarios as follows.

a) ***Scenario 1 - Minimal Application Load****:* This scenario consists of logging into the application and loading its home page. We consider this scenario because it refers to the minimal use of the application when the user simply connects to it. This allows us to evaluate the performance overhead of the telemetry agents under minimal application load. This execution scenario creates 156 HTTP requests including 36 RPC calls, among other things, such as resource loading.

b) ***Scenario 2 - Page Navigation****:* This scenario consists of logging into the application, loading its home page, and navigating to another web page from another module. We chose this scenario to show the difference in application performance of a page navigation compared to the first scenario of minimal application load. Seeing how it creates a total of 189 HTTP requests including 48 RPC calls, this scenario generates an extra 33 HTTP requests, including 12 RPC calls, compared to Scenario 1. These additional HTTP requests and RPC calls give us an indication about application performance upon executing a page-to-page navigation action.

c) ***Scenario 3 - Updating Client Address****:* This scenario begins by logging into the application, loading the home page and navigating to another page to update a client address. We chose this scenario to represent a simple real-world application use case as defined by our stakeholders. It also includes several user navigation actions (such as clicks and text inputs) and sending HTTP requests to the backend application. In total, it generates 236 HTTP requests and 70 RPC calls. The additional load generated by the extra HTTP requests of this scenario, compared to Scenarios 1 and 2, allows us to evaluate application performance for the use case of updating a client address.

D. Research Questions and Evaluation Methodology

To better generalize the results and simulate a real environment, we run each execution scenario based on multiple loads by *changing the number of virtual users*. We automate these runs based on *Gatling* [40], an automated performance testing tool. We run the performance test three times, with 1 user, 100 users, and 250 users at once to simulate the minimum (1), average (100), and maximum (250) load situations, respectively. This allows us to determine the relationship between performance overhead and application load size. Selecting 250 users as the maximum load is recommended by our application's operations team because each deployment instance can only handle a maximum of 250 users. A new instance will be created before reaching this number of users, based on the automatic elasticity feature of our deployment platform.

To get more realistic results about RT metrics, we run the performance test 10 times for each execution scenario for each different load size and measure their averages.

TABLE III
THE RESPONSE TIME GROUND-TRUTH REFERENCES FOR EVALUATING THE PERFORMANCE OVERHEAD OF TELEMETRY AGENTS

Execution Scenario	Load Size	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Execution Scenario 1	**1 User**	23.3	26	29.3	66.7	150.9	400.3	34.1	35.1
	100 User	22.8	46.5	79.5	213.7	940	7708.8	110.9	465.4
	250 Users	22.8	50.7	81.6	211.5	7902	22296.4	263.5	1792.4
Execution Scenario 2	**1 User**	23.9	26.3	30	61.2	151.9	386.2	33.5	31.3
	100 User	22	40.3	66.4	164.2	1173.6	10237.2	108.6	548.5
	250 Users	22	45.4	67.7	144.3	1666.7	35381.7	265.9	2210
Execution Scenario 3	**1 User**	23.7	26.8	31	70.1	147.5	373.5	34.6	29.4
	100 User	22	35.6	59	183.7	1184.6	15350	123.4	706.2
	250 Users	22	37.6	58.5	144.8	15153.9	47895.9	378.3	3152.3
Average of averages of all scenarios		22.72	37.24	55.88	140.02	3163.45	15558.88	150.31	996.73
Standard deviation of averages		0.77	9.35	20.86	60.77	5105.44	16824.82	123.84	1124.26

We evaluate the performance overhead by answering four Research Questions (RQs) as follows.

1) ***RQ1: What is the ground-truth reference for evaluating the performance of agents?:*** This RQ aims to identify a ground-truth reference that allows us to evaluate the performance overhead of our telemetry agents. To obtain this ground-truth reference, we run the application without any telemetry agent to collect RT metrics (c.f., Section IV-B1) about application performance in a regular environment. We consider these collected metrics as a ground-truth reference that allows us to measure the additional performance overhead caused by our telemetry agents.

2) ***RQ2: What is the performance overhead of the frontend agent?:*** The goal of this RQ is to measure the performance overhead caused by the frontend telemetry agent. As such, we run the application with only the frontend agent enabled to collect RT metrics about the performance of the application and the frontend agent. These metrics are compared to the ground-truth metrics obtained in RQ1 to measure the additional overhead caused by the frontend agent, based on Equation 1.

3) ***RQ3: What is the performance overhead of the program understanding backend agent?:*** The goal of this RQ is to evaluate the performance overhead incurred by our program understanding backend agent. As such, we run the application with only this backend agent enabled to collect performance metrics related to both the application and the backend agent. We compare these collected RT metrics to the ground-truth metrics to gauge the additional overhead introduced by the program understanding backend agent. Similar to RQ2, we rely on the *Percentage Difference* metric to measure the additional overhead, based on Equation 1.

4) ***RQ4: What is the performance overhead of the data exchange backend agent?:*** This RQ aims to measure the performance overhead caused by the data exchange backend agent. We run the application with only this agent enabled to collect RT metrics. These metrics are then compared to the ground-truth metrics to assess the additional performance overhead resulting from this agent, based on Equation 1.

E. Results

1) ***RQ1: What is the ground-truth reference for evaluating the performance of agents?:*** Table III shows the RT ground-truth references for evaluating the performance overhead of telemetry agents. For each execution scenario, we present the average RT of its 10 runs for the different load sizes in terms of the minimum, 50^{th}, 75^{th}, 95^{th}, and 99^{th} PCTs, maximum, mean and Standard Deviation (SD).

The results show that fast HTTP requests are not affected by changing the load size of the three execution scenarios. The average of minimum RTs is 22.72ms with a very small SD of 0.77 for 1, 100 and 250 users respectively.

We also notice that changing the execution scenario does not affect the RTs of all HTTP requests when we have *only one user*. This is highlighted by investigating the close values of all RT metrics for the minimal single-user load size (e.g., 34.1, 33.5, 34.6 ms for their means, 26, 26.3 and 26.8 ms for their 50^{th} PCT, etc.). Despite involving an additional 33 and 80 HTTP requests, respectively, compared to the first execution scenario, the second and third scenarios' extra requests result in negligible overhead when the system is subjected to a single-user load.

Nonetheless, the results show that RTs are generally affected by changing scenarios and load sizes. When we have larger load sizes, i.e. 100 and 250 users, the RT values for the second and third execution scenarios increase accordingly. It is demonstrated by the increase in their RT mean values. For instance, the mean RT is increased from 108.6 ms to 378.3 ms as we move from the second to the third execution scenario and from 100 to 250 users3. This is consistent since we have a positive correlation between the size/number of requests and the RTs.

The additional HTTP requests in the second and third execution scenarios are distributed in the first three quarters of the RT metric values. Meanwhile, the long HTTP requests, in all three scenarios, have shifted to the last fourth quarter. We analyze the maximum RT values, along with their 50^{th}, 75^{th}, 95^{th}, and 99^{th} percentiles (PCTs)cs and make the following observations: (1) the values of the 50^{th} and 75^{th} PCTs have decreased, (2) the values of the 95^{th} PCT are very similar, and (3) both the maximum RT values and the 99^{th} PCT values have significantly

increased for the second and third execution scenarios when there are 100 and 250 users.

In summary, we find that RT is affected by changing the number of HTTP requests. We change this number in our experiment by increasing the number of users and changing the execution scenarios. In our applications, RT is more sensitive to the number of users than execution scenarios. Thus, we rely on changing the number of users as the main axis to compare the performance of telemetry agents with our ground truth in RQ2, RQ3 and RQ4.

> The RT is affected by changes in the number of HTTP requests. It is more sensitive to the number of users than execution scenarios.

*2) **RQ2: What is the performance overhead of the frontend agent?:*** Figure 6 illustrates the outcomes of our application's performance evaluation when executed with the frontend agent enabled, compared to our ground-truth RT values. In particular, we report the minimum, 50^{th}, 75^{th}, 95^{th}, and 99^{th} PCTs, and maximum values, along with their means and standard deviations (SD). The results are organized into three distinct charts (a, b and c), each corresponding to a different load size: 1, 100, and 250 users, respectively. To facilitate the comparison of RT metrics with their corresponding ground-truth references across the different execution scenarios, we employ pairs of color-coded bar plots (orange, green and blue for the execution scenarios) differentiated by fill patterns (dotted and solid patterns for ground-truth and agent metrics, respectively).

Based on our telemetry architecture, the frontend agent is shipped with the frontend application code to the client web browser (c.f., Figure 2). This means that client machines bear the instrumentation overhead of the frontend agent, while our server remains unaffected when the application is executed with the agent enabled. This observation is supported by the consistent performance behavior of the application, regardless of whether the agent is enabled, for the same number of users and execution scenarios. When we compare the results of each RT metric with its corresponding ground-truth value, we find that the application performance exhibits some variability. In some cases, enabling the frontend agent leads to performance improvements. This variability can be attributed to the fact that the application performance may naturally vary slightly for the same load size (i.e., exact HTTP requests) across different runs, even without any agent. This is clearly evident in the slight fluctuations of the same RT metric across 10 runs in our ground-truth experiments.

The results of the *Overhead* metrics (c.f., Equation 1) show that we have negligible overhead, averaging 0.13%, 0.51% and 0.24% for 1, 100 and 250 users, respectively.

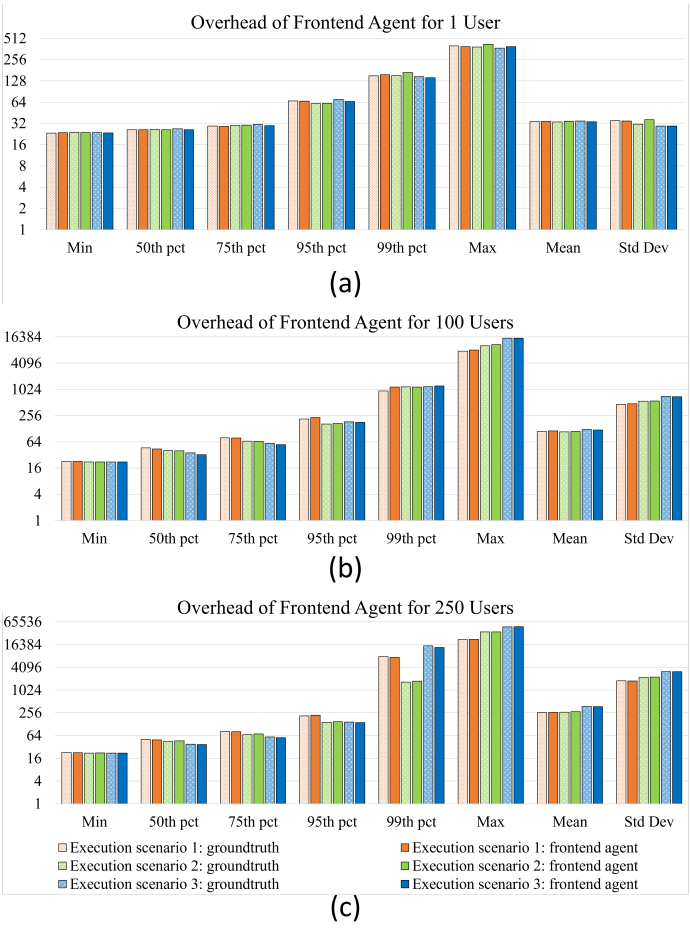

Fig. 6. Performance of frontend agent compared to our ground truth

> In summary, we can consider that, regardless of the number of users, no overhead is produced by the frontend agent on our server, because it runs on the client browsers.

*3) **RQ3: What is the performance overhead of the program understanding backend agent?:*** The performance evaluation results of the program understanding backend agent are presented in Figure 7 in three distinct charts (a, b and c), each corresponding to a different load size: 1, 100, and 250 users, respectively. We report the minimum, 50^{th}, 75^{th}, 95^{th}, and 99^{th} PCTs, and maximum values, along with their means and standard deviations (SD). Similar to RQ1, we facilitate the comparison of the RT metrics with their respective ground-truth references by depicting them as pairs of bar plots for each execution scenario. These bar plots are organized by color to distinguish between different execution scenarios and employ fill patterns to differentiate the ground-truth reference values from the obtained RT values.

In Figure 8, we present the performance overhead results, calculated based on the percentage difference. On average, our agent incurs an overhead of 2.14% (with means of mean RTs of 1.98%, 1.86% and 2.56% for 1, 100,

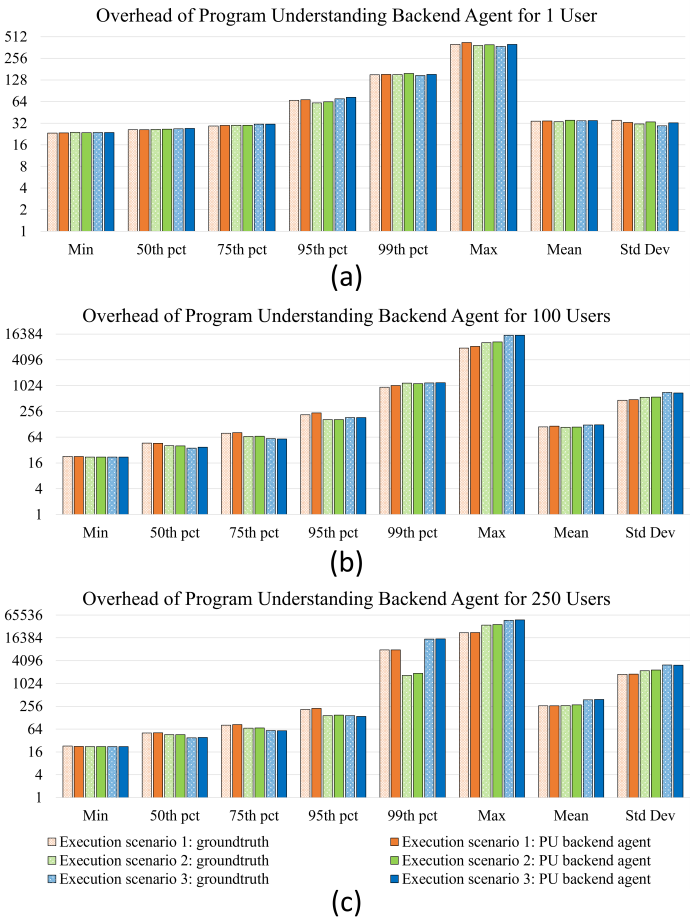

Fig. 7. Performance of program understanding backend agent compared to our groundtruth

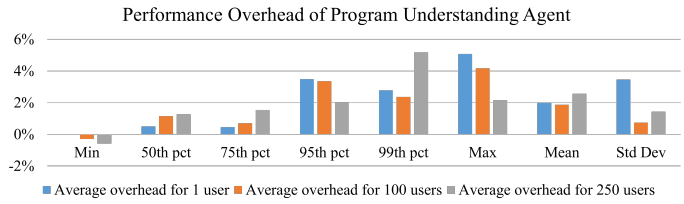

Fig. 8. Overhead of program understanding backend agent

and 250 users, respectively).

In approximately 75% of HTTP requests, we observe negligible overhead across various execution scenarios and numbers of users. Specifically, this negligible overhead amounts to 0.97% and 0.89% based on the average of the 50^{th} PCT and 75^{th} PCT, respectively. Consequently, the application performance is largely unchanged when running the application with this agent compared to our ground truth. This indicates that these HTTP requests involve executing minor operations (e.g., small number of method invocations, database operations) on the backend server. As a result, the agent's cost in transmitting its traces has a relatively minimal impact on the performance of our GWT application.

We also identify some outliers related to fast and slow requests. For instance, the minimum RT box plots show that the fastest HTTP requests had smaller RT values when we ran the application with our agents. This is because these requests are related to resource fetches that are not tracked by our backend agent. Some of these fast requests are RPC calls that involve minimal operations, such as retrieving a single value (e.g., date) calculated by a simple SQL statement.

Conversely, for the slowest requests, our agent produces an average overhead of 3.8% (with means of 5.07%, 4.17% and 2.15% for the maximum RTs with 1, 100, and 250 users, respectively). We observe a similar behavior for HTTP requests located in the fourth quarter of our RT values, as indicated by the 95^{th} PCT and 99^{th} PCT, where the means show overheads of 2.95% and 3.44%, respectively. We note that these slow requests involve executing numerous operations (e.g., complex SQL statements on a large dataset) on the server to compute their responses, thus necessitating more resources for the agent to transmit its traces.

> In conclusion, we observe a small overhead produced by the program understanding backend agent. This overhead has a positive correlation with the number of operations that need to be performed on the server.

*4) **RQ4: What is the performance overhead of the data exchange backend agent?:*** The performance evaluation results of our data exchange backend agent are shown in Figure 9. The results are illustrated in three separate charts (a, b and c), each corresponding to a different load size: 1, 100 and 250 users. For each chart, we report the minimum, 50^{th}, 75^{th}, 95^{th}, and 99^{th} PCTs, and maximum values, along with their means and standard deviations (SD). These obtained RT values are illustrated through color-coded bar plots to distinguish between their execution scenarios, with distinct fill patterns to differentiate them from their ground-truth references.

Figure 10 shows the performance overhead produced by the data exchange backend agent. The results reveal the agent's sensitivity to both the number of users of our GWT application and the size of the data being serialized. Notably, as the number of users and the size of the data exchanged between the client and the server increase, the agent's overhead escalates significantly. This divergence is evident in the considerable difference between the performance overhead mean values for 1 user (7.14%) compared to 100 users (29.13%) and 250 users (30.37%). The proximity of the mean values for the two latter workloads can be attributed to server capacity constraints. Indeed, with larger data serialization demands, server capacity diminishes. As such, we recommend performing stress tests on the server to determine the new capacity, accounting for the number of concurrent users when using this agent. In contrast, we find that requests involving small data experience minimal impact from changes in the number

of users and execution scenarios, as reflected by the RT of HTTP requests located in the first quarter of RT values.

The serialization cost exhibits a positive correlation with data size and serialization frequency. We observe a substantial overhead for slower HTTP requests situated in the fourth quarter of RT values, even for 1 user. This effect amplifies for the last 5% of these requests, mainly due to the size of the data being serialized upon tracing the arguments and return objects of the corresponding RPC services. As the number of users increases, so does the frequency of serialization.

Upon investigating the size of the serialized data, we pinpoint bottlenecks related to the return objects of RPC services. To understand their size, we show character counts for these return objects across all RPC services within the three execution traces in Figure 11. The results highlight substantial variations in object sizes, with a noteworthy standard deviation of 38770. This issue stems from application design rather than the agent itself. Therefore, we report this concern to our development team, as it not only affects the agent's performance but also impacts the application performance overall.

For a more tangible understanding of data serialization costs, we measure the time taken by the agent to serialize the 10 largest return objects. Figure 12 shows the degradation of serialization costs for the same return objects over time for a 250-user workload. The results reveal a notable escalation in serialization costs, particularly after about 100 users, reaching extreme levels as the number of users nears server capacity.

In summary, due to the data exchange agent's heightened sensitivity to data size, prudent usage is advised, especially when dealing with large objects. Given our agent's intended use case: constructing an oracle for input/output data in RPC service test automation, we recommend running this agent selectively, targeting specific subsets of application users at different intervals to prevent server overload. Furthermore, we advise to avoid serializing slow HTTP requests as much as possible. These recommendations remain applicable until we find an alternative serialization approach (e.g., one that works offline) that allows greater flexibility for running our agent without these restrictions.

> This data exchange agent exhibits sensitivity to the size of serialized data. While serialization costs remain reasonable for small objects, practitioners should be careful when using it with large objects.

F. Threats to Validity

1) Internal Threats to Validity: We identify three internal threats to validity as follows.

- The obtained performance evaluation results are based on virtual workloads of our application. To ensure they represent real-world application loads as much as possible, we consider running the application

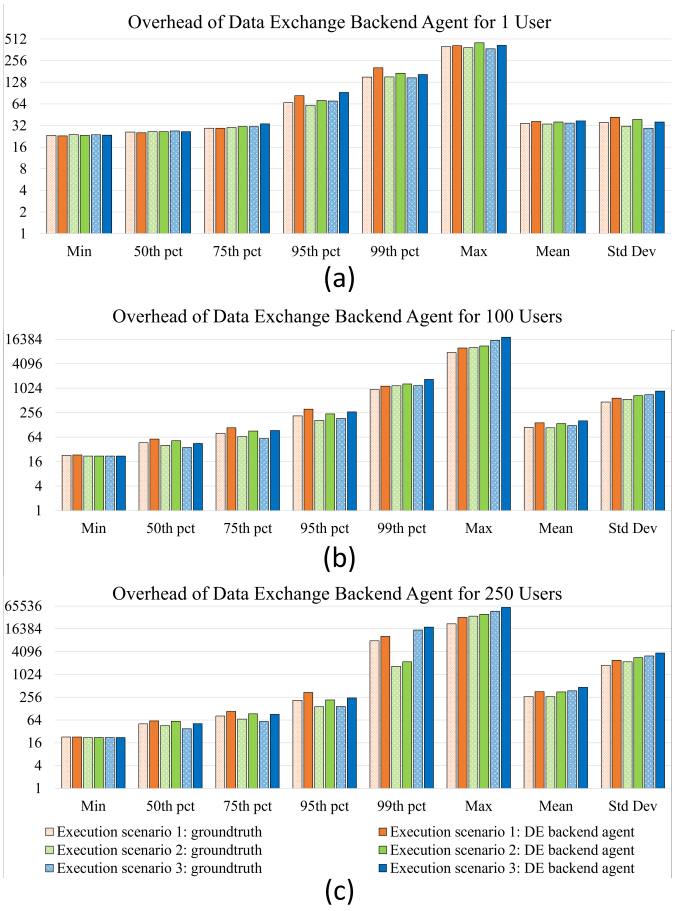

Fig. 9. Performance of the data exchange backend agent

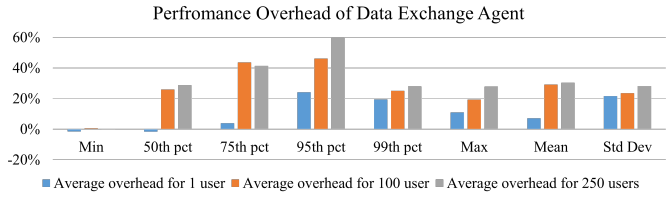

Fig. 10. Overhead of the data exchange backend agent

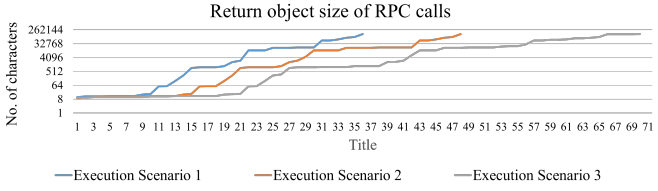

Fig. 11. Size of return objects of RPC calls for Execution Scenario 3

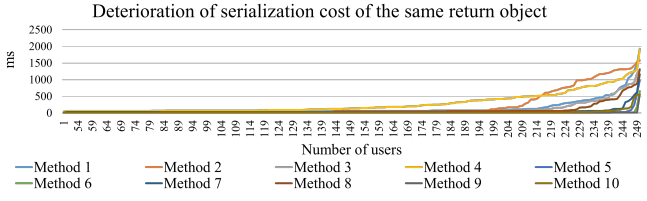

Fig. 12. Serialization cost deterioration for the same return objects

an average of 10 times for each execution scenario (different HTTP requests) and each number of users to calculate our performance metrics. This results in a total of 360 runs.
- The performance metrics are calculated based on Gatling. Thus, their accuracy is contingent upon Gatling's accuracy. However, Gatling is commonly used by the industry community to conduct performance load testing, such as *Adobe*, *SNCFConnect*, *Criteo*, *Sophos* and *thefork* [40], indicating an acceptable accuracy.
- Our performance overhead evaluation results are affected by the deployment infrastructure underlying Gatling and our application. To mitigate this impact, we isolate our application's deployment environment within a Microsoft Sandbox virtual machine on a dedicated server. This means that the resources of this sandbox are only used by our application. This arrangement ensures that the sandbox's resources are exclusively dedicated to our application, minimizing any variation in the deployment environment's impact on our results.

2) External Threats to Validity: We identify three external threats to validity as follows.

- The results are based on our GWT applications at Berger-Levrault. These applications are developed using GWT version 2.8.2 and Spring for the frontend and backend applications, respectively. This means that these results may be limited by these frameworks. However, the performance overhead evaluation results can be generalized to other applications in different domains that use the same frameworks. For example, the backend agents work seamlessly when automatically instrumenting another non-GWT application at Berger-Levrault, as its backend is also based on Spring.
- The performance overhead evaluation focuses on OpenTelemetry agents. Other telemetry agents, such as Elastic APM, may exhibit different performance overheads. Therefore, in our future work, we intend to assess other telemetry agents within the same evaluation environment. This will allow us to compare these agents based on their performance overhead.
- The obtained performance overhead percentages depend on the initial speed of the application under study. As such, faster or slower applications may yield different ground-truth response times, potentially leading to different percentage values.

V. Lessons Learned and Discussion

A. Manual Instrumentation of GWT Applications

Although automatic instrumentation serves our frontend telemetry objectives, we provide manual instrumentation capabilities to our development team. The idea is to allow them to explicitly inject instrumentation code into GWT applications, to collect specific telemetry data to gain fine-grained insight into critical parts of the application, specific workflows, or custom functionalities.

For the backend applications, the experiment shows that manual instrumentation using the Java agent is straightforward. Within the method body, we use the OpenTelmetry APIs[3] to create a span and add the desired attributes to that span.

However, manually instrumenting the front-end application is challenging because the GWT source code is in Java, while the executable code that runs in the browser is in JavaScript and is not human-readable. In fact, when we try to implement manual instrumentation using the Java OpenTelemetry agent's API, the GWT compiler fails to transpile it into executable JavaScript. This is because the OpenTelemetry Java agent is primarily focused on instrumenting applications that run natively in the Java Virtual Machine (JVM) [41]. Thus, the main challenge is to find a way to resolve the incompatibility between the GWT application source code, written in *Java*, and our agent's JavaScript implementation. Consequently, we have to resort to a bridging interface that allows us to import our defined tracer provider from the agent's JavaScript instrumentation file into the application's Java source code, and use it to create a tracer from which manual traces can be generated.

To solve this issue, we rely on JavaScript Native Interface (JSNI) [42]. This technology is provided by GWT to embed JavaScript code into the Java source code of a GWT application. In the context of our GWT application, the idea is to define a native manual instrumentation method that is parameterized based on the developers needs. The role of this method is to act as an interface between Java and JavaScript, where JSNI syntax will be used. For example, create a span for such events. The native method can subsequently be invoked like any other Java method from the application's source code where it's needed, while having access to the JavaScript agent.

Moreover, this manual instrumentation method requires the import of our agent's tracer provider object, through which it can access a tracer to create the span.

Unfortunately, while JSNI provides a way to interact with JavaScript code, it doesn't natively support modern JavaScript features such as module import/export syntax. The only way to do this in JSNI is to reference the object as a property of the global window object $wnd in the browser's JavaScript environment. Thus, JavaScript objects are often referenced in JSNI through $wnd. As such, we stored our tracer provider object as a property of $wnd to enable its access through JSNI and its use in the body of our native manual instrumentation method. However, using global variables for JavaScript integration has some limitations, primarily related to naming conflicts, code isolation, and security leaks [43]. As a result, it is possible but not recommended.

[3]https://opentelemetry.io/docs/instrumentation/java/manual/#create-spans

B. Data Serialization

We face three main issues with data serialization. These are the serialization cost, lazy loading of properties and nested Data Transfer Objects (DTOs).

1) Serialization Cost: GWT frontend applications rely on the Remote Procedure Call (RPC)mechanism to invoke server-side methods. The invocation of these methods involves data exchange based on shared DTO Java objects. Thus, if we want to instrument the client-server data exchange, we must serialize these Java objects into JavaScript Object Notation (JSON) in order to include them in the generated traces. The cost of this serialization process can be significantly affected by the size of these objects. Our evaluation of the performance overhead shows that serializing small objects is acceptable, while serializing large objects has some impact. Therefore, we plan to to investigate other serialization strategies and libraries in our future work.

2) Lazy loading of proprieties: The GWT framework uses lazy loading to delay loading certain DTO objects until they are actually needed. This helps improve the performance of GWT applications by minimizing initial load time and loading resources only when they are needed. This technique is widely used in our applications. However, it poses a problem when serializing DTO objects that are not fully loaded. Among the most commonly used serialization libraries such as *Gson* and *Jackson*, we find that *Xstream* supports resolving the serialization of lazy loading objects internally, without the need to modify the application source code. Jackson does not support resolving the serialization of lazy loading objects by default. Instead, it requires modifying the implementation of the DTO Java classes by adding some Java annotations that tell Jackson to ignore these properties during serialization. Since one of our objectives is to minimize changes to the application source code, we use *Xstream*.

3) Nested DTO Java Objects:: We find that the implementation of our applications contains nested DTO objects. Nested DTO objects are DTOs that contain other DTOs as properties, forming a nested structure. For example, in a library management system, we have two DTOs: BookDTO and AuthorDTO. Each BookDTO contains information about a book, including its title, ISBN, and author represented by the AuthorDTO. Each AuthorDTO contains information about the author, such as their name, biography, and the list of books represented by a list of BookDTOs. In this example, we have a circular structure, which is considered a challenge for data serialization tools. For instance, Jackson will get stuck in an infinite serialization process, unless we use some annotations to tell it to stop after one round in the current cycle. We find that *Xstream* automatically resolves the serialization of nested DTO objects without any changes to the application source code.

C. Cross-Origin Resource Sharing (CORS):

CORS is a security feature implemented by web browsers that restricts web pages from making requests to a domain other than the one that served the original page. This is done to prevent malicious scripts from stealing data or performing actions on behalf of a user without their consent. Consequently, CORS-related errors can occur and must be handled appropriately to allow data export when the application and collector are in different domains. This can be addressed by defining a CORS policy in the OpenTelemetry Collector configuration to explicitly include the application's URLs in the list of origins from which the collector is allowed to accept the exported data. However, this can be difficult to achieve if the collector configuration is inaccessible or unavailable. Furthermore, this approach can become error-prone and difficult to maintain if many applications are intended to send their telemetry data to the collector.

VI. Related Work

In this section, we discuss prior studies on telemetry approaches and the performance overhead evaluation of telemetry agents.

A. Existing Telemetry Approaches

We classify existing telemetry approaches based on the target application to be instrumented by a given telemetry approach, the approach itself, and the potential use cases of the collected telemetry data.

Our findings highlight different levels of similarities and differences between the instrumented applications. Regarding the studied platforms, we mainly identify web [14]–[16], [21], [44]–[47], desktop [14], [48], and IoT [21], [44], [48], [49] applications. These applications are developed based on different architectural styles such as Monolithic [48], N-Tiers [14], Follower-Leader [50] and Microservices [16], [21], [45], [46], [51]. In terms of their implementation, they adopt different technologies, such as Vaadin [14], Spigo [16], AngularJS [46], and Vue [52] for the frontend and Spring Cloud [49], Spring Boot [46], [53] and Node.js [46], [47] for the backend. Moreover, they can be community-based [46], industrial [16], [21], [47], [51], benchmark-oriented [44], [45], [48] or used for demonstration purposes [14]. However, none of these approaches tackle industrial GWT-Spring applications.

Furthermore, the telemetry approaches used to instrument these applications collect various telemetry data types, including traces [16], [45], [47], metrics [14], [44], [46], [54] and logs [46], [55]. Some approaches collect traces [16], [21], [45], metrics [44], or logs [55] exclusively, while others collect multiple telemetry data types [46], [51], [54]. This data is generated based on OpenTelemetry [16], [21], [48], Elastic APM [51], [52], [55], Jaeger [45], [46], or custom-built [44] instrumentation agents. Some of these approaches resort to AspectJ [14] or Spring Boot Sleuth [53] for the instrumentation of their target applications. To collect the generated telemetry data and transmit it to the telemetry backend, the examined approaches rely on telemetry data collectors provided by OpenTelemetry [21], [47], [56], Jaeger [16], [47] or Fluentd [14]. To store,

index, analyze, and visualize their collected telemetry data, many approaches relied on one or multiple telemetry backends, including the Elastic Stack [14], [51], [54], [55], Jaeger [16], [45]–[47], Zipkin [48], [53], and Prometheus [47], [48], [54]. While we rely on Elastic Observability as our telemetry backend, several approaches rely on its foundational predecessor - the Elastic Stack, as constituents of their telemetry backends. However, none of the existing approaches used OpenTelemetry exclusively for an end-to-end instrumentation of the target applications and for the collection of the generated telemetry data.

The telemetry data collected by these approaches is used for several purposes, such as constructing call graphs [15], [45], anomaly detection [47], infrastructure monitoring [44], [49], application performance monitoring [47], [54], [55], performance analysis [44], security monitoring [45], root cause analysis [46], [47], bottleneck detection [16], [53] and user interaction [14].

Compared to our telemetry objectives, none of the existing approaches collect telemetry data related to user navigation of GWT applications, client-sever data exchange based on the arguments and return objects of RPC services, and user identities. For example, Suonsyrja et al. [14] address user navigation partially, by solely handling click events, which makes up only a subset of the user-generated events traced by our approach. This approach also needs to modify the application source code during instrumentation. Casse et al. [16] and Lui et al. [15] address RPC calls between application components by tracing their RPC invocation chains. However, this can be only used to build call graphs between misconceives, but not at the class and method level.

Despite the architecture similarities we share with existing approaches, they still diverge from our work in terms of the applications they target for instrumentation, the instrumentation approach, and the objectives and uses cases for which the telemetry data should be generated. In our case, we seek to generate telemetry data in alignment with our defined objectives, including user identity, frontend user navigation, backend actions, and client-server data exchange, without source code modification and with minimal performance overhead. Our approach relies on OpenTelemetry for instrumentation and collection and Elastic Observability as a telemetry backend. We evaluate our approach using legacy GWT-Spring applications at Berger-Levrault.

B. Performance Overhead Evaluation of Telemetry Agents

In the literature, to the best of our knowledge, we identify five studies that evaluate the performance overhead of telemetry agents. We classify these studies based on their goals, benchmark applications, evaluation methodology, evaluation metrics, and their obtained results.

Based on their goals, these studies can be classified into approaches that evaluate only the performance overhead of OpenTelemetry Java agents [21], [23], compare OpenTelemetry with InspectIT and Kieker Java agents [48], and evaluate the performance overhead of other agents like Dapper [57], ROS2_tracing [58]. None of these approaches evaluate the overhead of frontend agents, like the JavaScript ones.

Existing approaches rely on different types of benchmark applications, including cloud-based microservice [21], [23], [57], J2SE [48], ROS 2 [58], and Raspberry Pi 4 [48] applications. None of these approaches rely on GWT-Spring applications.

Similar to our evaluation methodology, these studies measure the performance of the application with and without the agents. They compare the evaluation metrics following these two scenarios to determine the additional overhead that might be caused by the agents.

Existing studies rely on a variety of evaluation metrics such as latency [21], [57], [58], CPU utilization [23], [57], method execution time [48], logging to disk [48], regular text logging [48], throughput [57] and memory usage [59]. Each of these metrics provides indications about different aspects of the application performance. Despite its importance, the response time metric is not considered in any of these existing studies.

The results vary depending on the different benchmarks and evaluation metrics used. Shuvo [21] and Reichelt et al. [48] find between 9% and 16% additional performance overhead in latency caused by OpenTelemetry Java agents. As for the OpenTelemetry exporter overhead, DoorDash Engineering [23] finds that peak CPU usage increases by 33% when the OpenTelemetry exporter is enabled.

Although we share a similar evaluation methodology with these studies, they differ in terms of target telemetry agents, benchmark applications and evaluation metrics. In our case, we evaluate the performance overhead of OpenTelemetry Java and JavaScript agents using a GWT-Spring web application based on response time and percentage difference metrics. Consequently, it is reasonable to expect different performance evaluation results that are not comparable to theirs.

VII. Conclusion & Future Works

In this paper, we report our industrial experience in instrumenting three real, large-scale, industrial legacy web applications based on our telemetry approach at Berger-Levrault. Our approach is motivated by the need of our internal stakeholders (e.g., product managers, product owners, architects) to support the maintenance of these applications. This is based on collecting and analyzing real-time data about the frontend user navigation, the backend actions, the client-server data exchange, and the identity of the end-user initiating them. This allows us to measure the user experience, build a dependency call graph of how different components interact, and create a corpus of backend service input/output data for test case generation.

To obtain the necessary data, we automatically instrument our GWT-Spring applications to generate traces of user navigation, backend actions, client-server data exchange, and the identity of the end user initiating them.

We are able to perform this telemetry with minimal performance overhead and no changes to the application source code by extending the automatic instrumentation capabilities of OpenTelemetry agents. We use an OpenTelemetry Collector to collect traces generated by different sources, then we store and analyze them in Elastic Observability.

Furthermore, we empirically evaluate the performance overhead of our agents by comparing the response time of the GWT-Spring application with and without running these agents. We simulate a real-world environment using Gatling to generate virtual workloads for our application. The results show that there is no significant performance overhead when using our agents. However, the data exchange agent is sensitive to the size of the serialized data. The serialization cost is reasonable for small objects, but practitioners should be cautious when using it with large objects.

To help other researchers and practitioners implement their telemetry approaches for industrial contexts, we discuss lessons learned about overcoming some technical challenges we faced during the industrialization of our telemetry approach. Such technical challenges pertain to data serialization (large objects, lazy loading of properties, and nested Data Transfer Objects), Cross-Origin Resource Sharing, and manual instrumentation of GWT frontend code using JavaScript Native Interface.

As future work, we plan to experiment our telemetry agents with other types of web applications (e.g., Angular, React, Sprint Boot) to generalize the results. We would also like to empirically compare OpenTelemetry, Elastic APM, and Spring Cloud Sleuth agents to help practitioners choose the best agent for their use cases based on their performance overhead and the types of collected telemetry data. Furthermore, we want to show some use cases of the collected telemetry data by constructing a dependency call graph of user navigation and developing an automated testing approach for RPC/REST services based on their traced input/output data.

References

[1] H. M. Sneed, "Integrating legacy software into a service oriented architecture," in *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 2006, pp. 11–pp.

[2] S. Adjoyan, A.-D. Seriai, and A. Shatnawi, "Service identification based on quality metrics object-oriented legacy system migration towards soa," in *SEKE: Software Engineering and Knowledge Engineering*. Knowledge Systems Institute Graduate School, 2014, pp. 1–6.

[3] H. Mili, I. Benzarti, A. Elkharraz, G. Elboussaidi, Y.-G. Guéhéneuc, and P. Valtchev, "Discovering reusable functional features in legacy object-oriented systems," *IEEE Transactions on Software Engineering*, 2023.

[4] B. Verhaeghe, A. Shatnawi, A. Seriai, N. Anquetil, A. Etien, S. Ducasse, and M. Derras, "Migrating gui behavior: from gwt to angular," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 495–504.

[5] B. Verhaeghe, A. Shatnawi, A. Seriai, A. Etien, N. Anquetil, M. Derras, and S. Ducasse, "From gwt to angular: An experiment report on migrating a legacy web application," *IEEE Software*, vol. 39, no. 4, pp. 76–83, 2021.

[6] "Berger-levrault," 2023. [Online]. Available: https://www.berger-levrault.com

[7] "Gwt." [Online]. Available: https://www.gwtproject.org/

[8] I. Boukhraouba *et al.*, "From user activity traces to navigation graph for software enhancement: An application of graph neural network (gnn) on a real-world non-attributed graph," in *ACM International Conference on Information and Knowledge Management (CIKM2023)*, 2023.

[9] A. Shatnawi, H. Mili, G. El Boussaidi, A. Boubaker, Y.-G. Guéhéneuc, N. Moha, J. Privat, and M. Abdellatif, "Analyzing program dependencies in java ee applications," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 64–74.

[10] G. Darbord, A. Etien, N. Anquetil, B. Verhaeghe, and M. Derras, "A unit test metamodel for test generation," in *International Workshop on Smalltalk Technologies*, 2023.

[11] "Apache log4j," 2023. [Online]. Available: https://logging.apache.org/log4j/2.x/

[12] "OpenTelemetry: OTLP Specification 1.0.0," 2019. [Online]. Available: https://opentelemetry.io/docs/specs/otlp/

[13] "Elastic Observability: Transform Your Data into AI-Powered Insights," 2023. [Online]. Available: https://www.elastic.co/observability

[14] S. Suonsyrjä and T. Mikkonen, "Designing an Unobtrusive Analytics Framework for Monitoring Java Applications," in *Software Measurement*, ser. Lecture Notes in Business Information Processing, A. Kobyliński, B. Czarnacka-Chrobot, and J. Świerczek, Eds. Cham: Springer International Publishing, 2015, pp. 160–175.

[15] H. Liu, J. Zhang, H. Shan, M. Li, Y. Chen, X. He, and X. Li, "JCallGraph: Tracing Microservices in Very Large Scale Container Cloud Platforms," in *Cloud Computing – CLOUD 2019*, ser. Lecture Notes in Computer Science, D. Da Silva, Q. Wang, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 287–302.

[16] C. Cassé, P. Berthou, P. Owezarski, and S. Josset, "A Tracing Based Model to Identify Bottlenecks in Physically Distributed Applications," in *2022 International Conference on Information Networking (ICOIN)*, Jan. 2022, pp. 226–231, iSSN: 1976-7684. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9687217

[17] "Telemetry: Summary of concept and rationale," p. 13455, Dec. 1987.

[18] "OpenTelemetry: Instrumentation for JavaScript Applications," 2019. [Online]. Available: https://opentelemetry.io/docs/instrumentation/js/

[19] G. Leffler, "OpenTelemetry and observability: What, why, and why now?" Sydney: USENIX Association, Dec. 2022.

[20] "IBM: What is OpenTelemetry?" 2023. [Online]. Available: https://www.ibm.com/topics/opentelemetry

[21] G. K. Shuvo, *Tail Based Sampling Framework for Distributed Tracing Using Stream Processing*, 2021. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-306699

[22] "Splunk," 2023. [Online]. Available: https://www.splunk.com/

[23] "DoorDash Engineering: Optimizing OpenTelemetry's Span Processor for High Throughput and Low CPU Costs," 2023. [Online]. Available: https://doordash.engineering/2021/04/07/optimizing-opentelemetrys-span-processor/

[24] "HTMLElementEventMap | typescript - v3.7.7," 2020. [Online]. Available: https://microsoft.github.io/PowerBI-JavaScript/interfaces/_node_modules_typedoc_node_modules_typescript_lib_lib_dom_d_.htmlelementeventmap.html

[25] "OpenTelemetry: Tracing API," 2019, section: docs. [Online]. Available: https://opentelemetry.io/docs/specs/otel/trace/api/

[26] "OpenTelemetry: Tracing SDK," 2019, section: docs. [Online]. Available: https://opentelemetry.io/docs/specs/otel/trace/sdk/

[27] "Nodejs," 2023. [Online]. Available: https://nodejs.org/en

[28] "Babeljs," 2023. [Online]. Available: https://babeljs.io/

[29] "Webpack," 2023. [Online]. Available: https://webpack.js.org/

[30] D. Gomez Blanco, *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization*. Berkeley, CA: Apress, 2023. [Online]. Available: https://link.springer.com/10.1007/978-1-4842-9075-0

[31] "OpenTelemetry: Collector," 2019. [Online]. Available: https://opentelemetry.io/docs/collector/

[32] "SigNoz: an open-source observability tool," 2023. [Online]. Available: https://signoz.io/

[33] "Zipkin: a Distributed Tracing System," 2023. [Online]. Available: https://zipkin.io/
[34] "Elastic Stack: An Overview," 2010. [Online]. Available: https://www.elastic.co/guide/en/starting-with-the-elasticsearch-platform-and-its-solutions/current/stack-components.html
[35] "Elasticsearch," 2010. [Online]. Available: https://www.elastic.co/
[36] "Amazon Web Services (AWS)," 2006. [Online]. Available: https://aws.amazon.com/
[37] "Grafana Labs: Observability Survey 2023: Key findings and analysis on the state of observability," Apr. 2023. [Online]. Available: https://grafana.com/observability-survey-2023/?pg=blog&plcmt=body-txt
[38] "SigNoz: Blog - Top 11 Observability Tools in 2023," Sep. 2023. [Online]. Available: https://signoz.io/blog/observability-tools/
[39] T. J. Cole and D. G. Altman, "Statistics notes: What is a percentage difference?" *Bmj*, vol. 358, 2017.
[40] "Gatling: professional load testing tool." [Online]. Available: https://gatling.io/
[41] "OpenTelemetry: Instrumentation for Java," 2019. [Online]. Available: https://opentelemetry.io/docs/instrumentation/java/
[42] "JSNI," 2023. [Online]. Available: https://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html
[43] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated Analysis of Security-Critical JavaScript APIs," in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 363–378, iSSN: 2375-1207.
[44] R. Brondolin and M. D. Santambrogio, "A Black-box Monitoring Approach to Measure Microservices Runtime Performance," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 34:1–34:26, Nov. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3418899
[45] S. Jacob, Y. Qiao, Y. Ye, and B. Lee, "Anomalous Distributed Traffic: Detecting Cyber Security Attacks Amongst Microservices Using Graph Convolutional Networks," *Computers & Security*, vol. 118, p. 102728, Jul. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822001237
[46] A. Bento, J. Correia, J. Duraes, J. Soares, L. Ribeiro, A. Ferreira, R. Carreira, F. Araujo, and R. Barbosa, "A Layered Framework for Root Cause Diagnosis of Microservices," in *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, Nov. 2021, pp. 1–8, iSSN: 2643-7929. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9685494
[47] G. Y. Kusuma and U. Y. Oktiawati, "Application Performance Monitoring System Design Using Opentelemetry and Grafana Stack," *Journal of Internet and Software Engineering*, vol. 3, no. 1, pp. 26–35, Nov. 2022, number: 1. [Online]. Available: https://journal.ugm.ac.id/v3/JISE/article/view/5000
[48] D. Reichelt, S. Kühne, and W. Hasselbring, "Overhead comparison of opentelemetry, inspectit and kieker." in *SSP*, ser. CEUR Workshop Proceedings. CEUR Workshop Proceedings, 2021.
[49] R. Kang, Z. Zhou, J. Liu, Z. Zhou, and S. Xu, "Distributed Monitoring System for Microservices-Based IoT Middleware System," in *Cloud Computing and Security*, ser. Lecture Notes in Computer Science, X. Sun, Z. Pan, and E. Bertino, Eds. Cham: Springer International Publishing, 2018, pp. 467–477.
[50] J. Zhou, Z. Chen, H. Mi, and J. Wang, "MTracer: A Trace-Oriented Monitoring Framework for Medium-Scale Distributed Systems," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, Apr. 2014, pp. 266–271. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6830915
[51] S. de Vries, F. Blaauw, and V. Andrikopoulos, "Cost-Profiling Microservice Applications Using an APM Stack," *Future Internet*, vol. 15, no. 1, p. 37, Jan. 2023, number: 1 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1999-5903/15/1/37
[52] O. Ritari, "Monitoring a Kubernetes Application," Ph.D. dissertation, Karelia University of Applied Sciences, 2019, accepted: 2019-12-13T06:46:22Z. [Online]. Available: http://www.theseus.fi/handle/10024/266071
[53] S. D. Mallanna and M. Devika, "Distributed Request Tracing using Zipkin and Spring Boot Sleuth," *International Journal of Computer Applications*, vol. 175, pp. 35–37, Aug. 2020.
[54] R. Boncea, A. Zamfirou, and I. Bacivarov, "A Scalable Architecture for Automated Monitoring of Microservices," vol. 18, Sep. 2018.
[55] A. Tiwari and D. Mane, "Application Performance Monitoring Using Log File on ELK Stack," *IRJET*, Jan. 2020. [Online]. Available: https://www.academia.edu/44341224/IRJET_Application_Performance_Monitoring_Using_Log_File_on_ELK_Stack
[56] A. Ellis, "Emplacing New Tracing: Adding OpenTelemetry to Envoy," Master's thesis, 2022. [Online]. Available: https://www.proquest.com/openview/31eb07ac06dd2110c49e0b51ca355ffc/1?pq-origsite=gscholar&cbl=18750&diss=y
[57] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. K. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:14271421
[58] C. Bedard, I. Lutkebohle, and M. Dagenais, "ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, jul 2022. [Online]. Available: https://doi.org/10.1109%2Flra.2022.3174346
[59] R. J. Rodríguez, J. Artal, and J. Merseguer, "Performance evaluation of dynamic binary instrumentation frameworks," *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 12, pp. 1572–1580, 12 2014.

Anas Shatnawi received his Ph.D. degree in computer science from University of Montpellier, France, in 2015. He is now a senior research engineer at Berger-Levrault. He was a researcher at Sorbonne University, University of Milano-Bicocca and University of Quebec at Montréal. His research interests include software reuse, reengineering, reverse engineering, and empirical software engineering. He has published many papers in various international journals and conferences on these topics.

Bachar Rima is a Ph.D. student at University of Montpellier and Laboratory of Computer Science, Robotics, and Microelectronics (LIRMM), France. His current research focuses on software observability and logging. His broader research interests include design patterns, software architectures, software maintenance & evolution, and prompt engineering. He works closely with Berger-Levrault in a mutually beneficial collaboration that allows the research to involve real industrial cases and Berger-Levrault to benefit from the applied research.

Zakarea AL SHARA received his Ph.D. degree in software engineering University of Montpellier, France, in 2016. He is now an assistant professor in the Department of Software Engineering at Jordan University of Science and Technology. He is the team leader of collaboration with the Compact Muon Solenoid (CMS) and the European Organisation for Nuclear Research (CERN) in Geneva, Switzerland. His current research interests include software maintenance, evaluation, architecture, and modeling. He has published many papers in various international journals, conferences and workshops on these topics.

Gabriel Darbord is a Ph.D. student in the Evref team at Inria Lille - Nord Europe, France. His research focuses on automatic test generation. His broader research interests include software maintenance and evolution, and software modeling. He works closely with Berger-Levrault in a mutually beneficial collaboration that allows the research to involve real industrial cases and Berger-Levrault to benefit from applied research.

Djamel Seriai is an Associate Professor at University of Montpellier and a member of the Laboratory of Computer Science, Robotics, and Microelectronics (LIRMM), France. He is also the co-head of the software engineering master and a senior software architect with more than 25 years of experience as an engineer, architect, and project manager in the software development and maintenance field, and more than 20 years of experience in innovation related to applied research projects.

Christophe Bortolaso is the head of research at Berger-Levrault, France. He obtained his Ph.D. in computer science from University of Toulouse III in 2012.