SaaS Observability on the Microsoft Power Platform and its Performance Impacts

Robert Pröll
Kupp Software
Linz, Austria
ropr@kuppsoft.com

Markus Weninger
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
markus.weninger@jku.at

***Abstract*—Modern Software as a Service (SaaS) platforms, such as the Microsoft Power Platform, simplify maintenance efforts through abstraction of underlying layers such as managing server infrastructure or OS-level configurations. Nevertheless, they still allow custom code extensions to extend and customize the application behavior. Yet, this implies restricted control and limited insights with respect to performance.**

Often, built-in performance reportings do not provide enough detail for efficient problem root cause analysis. Open standards, such as OpenTelemetry, make it easier to implement a more advanced aligned tracing ecosystem. Yet, while monitoring on traditional, non-cloud systems has been thoroughly studied, collecting telemetry data on SaaS systems faces many more complex challenges such as sandboxing, e.g., limited API access, or collecting end-to-end traces through multiple distributed entry points. Besides platform experience, deep technical knowledge about serialization, transmission, and caching costs is essential to understand the process and assess the performance overhead introduced by collecting telemetry data.

In this work, we discuss common challenges regarding SaaS observability and the performance impact that telemetry data collection introduces. For this, we modify Intermediate Language (IL) code, i.e., already compiled code, on the Microsoft Power Platform to collect telemetry data for each method execution. A key finding is that a significant and often unreported portion of performance overhead originates from the SaaS platform itself, a crucial distinction from the overhead introduced by telemetry collection. Finally, we propose a concept for adaptive telemetry collection based on execution plan prediction, which aims to balance observability detail with performance overhead by estimating monitoring costs.

***Index Terms*—SaaS (Software as a Service), Power Platform, Observability, Monitoring, Telemetry, Performance Overhead, Instrumentation, Distributed Tracing**

I. Introduction

The basic principle of SaaS is that the provider offers a running application (such as a CRM system [1], [2]) and covers all related operational tasks, including OS-level updates and database backups. At the same time, the customer must have enough flexibility to extend and customize the application through custom code (also called *plugins*). The critical need for robust performance monitoring in SaaS software is well-documented [3], yet acquiring suitable telemetry for these processes remains a significant challenge in restrictive platforms.

In this paper, we focus on the Microsoft Power Platform (PP), a widely used SaaS environment, and how to collect telemetry data using OpenTelemetry and IL-code injection. Although IL-code injection is a well established way to monitor applications, there are many additional challenges when it is used within a SaaS environment. Besides discussing platform limitations, we focus our research on quantifying and mitigating observability impact under SaaS restrictions. Since we cannot call certain APIs directly (due to sandboxing, etc.), many existing tools can not be directly applied in SaaS environments. Thus, we have explored various ways to provide a solution which works in a restricted SaaS environment, but also handles the issues caused by distributed entry points. We apply IL modifications, a very powerful tool, the implementation efforts and performance degradation risks are much higher than alternative ways such as collecting logs.

A major difference to traditional on-premise applications is starting the telemetry data collection. Since it is highly important to ensure that a full trace (i.e., a collection of all called methods and API endpoints) is collected for every user request, we have to find the earliest possible starting point (called *entry point*) for the telemetry collection process, and combine it with all subsequent method calls. In a "traditional" web application, starting a trace is normally done via instrumenting the first loaded script/code on the client- and server side. Within SaaS environments these scripts are unmodifiable, which implies that we can only add telemetry collection at later entry points.

In PP, we can trace *platform events* (e.g., the update of a user account, or creation of a contact). Since multiple such events could occur for a single user request, their separate traces have to manually be connected based on a common ID in the right order.

Learning: SaaS have distributed entry points

Conclusion: No global entry point means different sub-traces must be linked based on a common unique identifier (Correlation ID).

Summarized, our contributions encompass:

- an example implementation of the various phases of an instrumented plugin's execution (section III)
- a discussion of challenges in the field of SaaS observability (section IV)
- experimental analysis of common performance impact

This is the author's version of the work. The definitive version will be published in the Proceedings of the 13th IEEE International Conference on Cloud Engineering (IC2E 2025).

factors added by telemetry data collectors (section V), including event-to-code-level walkthroughs based on concrete use cases for common scenarios, including overhead and root cause analysis (subsection V-A)

II. Background

Typically, SaaS applications are structured via layers as shown in Figure 1. The access restrictions for each layer are of particular importance here: Starting from the bottom (at low-level components such as DB/Infrastructure) up to the first application layer (out-of-the-box functionality), all components are exclusively managed by the provider.

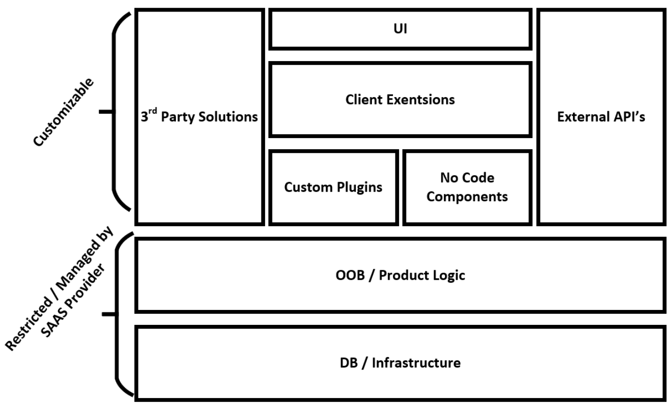

Fig. 1. A high-level overview of common SaaS Application Layers, grouped by customizable and restricted areas.

This means it is not possible to install any additional software to collect telemetry data directly within these layers.

On top of these restricted layers, customers can build custom logic. Since customer code (so-called *Custom Plugins*) could be a potential risk for the SaaS platform (with regard to, e.g., security, hardware resources, etc.), a sandboxed execution context is enforced. A sandbox blocks many system APIs (such as directly accessing memory resources, runtime telemetry events and similar), which causes traditional telemetry data collectors (code libraries which collect telemetry events during runtime) to fail.

Learning: Unable to install traditional observability tools in SaaS environments

A key observation is that traditional observability tools cannot be used due to limited API access and missing permissions. We have to resort to low-level instrumentation techniques such as IL modifications.

Although a few basic concepts such as logging [3], watchdog services, etc. are parts of modern observability concept implementations, it is important to extend those concepts with more advanced (and more detailed) approaches. This section introduces the concept of traces (with references to the OpenTelemetry Protocol) and the Microsoft Power Platform.

A. Observability in Modern Software Systems

IT observability mostly refers to tracing specific actions from the initial trigger point (e.g., a user clicks on a button on a webpage) to every reached action in other systems, including recording of various context-specific parameters and metadata, such as server IP or input parameters. That means that one trace, i.e., a hierarchically structured recording of actions with a single root entry, includes all information needed to investigate a complete transaction including all intermediate steps/systems. Although traces can be enhanced with manually created logs, traditional observability software uses mostly automatic instrumentation to generate traces [4]–[6].

We want to achieve the same level of auto-instrumentation on SaaS environments. As discussed in detail by Bissig [7], the most common way to collect monitoring data is to use automatic code instrumentation provided by various libraries. Yet, this approach is not suitable for most SaaS environments (such as Microsoft Power Platform), due to dependencies on certain restricted APIs.

B. OpenTelemetry (Concepts, Protocol)

The OpenTelemetry Protocol (OTLP) is an open industry standard [8] to define how different telemetry data collectors can interact with each other. OTLP uses a single container for each collection of related actions, called a trace. A trace contains multiple recorded actions (e.g., time-measured units of work [9] such as method calls), called spans.

C. Power Platform (PP)

Microsoft Power Platform (PP) is a SaaS ecosystem for business applications, mainly CRM and ERP enterprise applications. Like many other SaaS applications, PP is event-driven (with multiple handlers for various events) instead of a single-entry point (e.g., traditional applications that always start at their `main` method).

An update to an account could be such an event, as well as any other create, delete, retrieve operation to a single or multiple record(s). Each of these events will then trigger their registered custom code. These code points are then the earliest, top-most code locations to start telemetry collection. Note: The term *plugin* usually refers to *custom code*, we use this definition throughout this work.

Since code-based components (written in C#) are only available in compiled form on PP, Intermediate Language [10] statements must be injected to collect telemetry data. Installing "regular" tools, such as the OTel library for .NET applications, would require access to various restricted APIs (e.g., the File API or the .NET Metric APIs)

Learning: Only Intermediate Language provides enough flexibility

We do not have access to the C# source code, only the compiled .dll is available. Modification of this compiled code can only be done via IL code.

III. Observing SaaS Environments

User plugins are stored as .dll files within PP, which means they can be updated at any time (i.e., deployment). Based on that, we implemented a process to download changed .dlls, instrument all methods within it, and then upload the instrumented .dll-Files back to PP. We need to repeat that process every time a new version of a .dll has been updated.

The instrumentation process itself does not trigger any tracing. Traces are generated in multiple phases when user plugins are executed, as shown in Figure 2.

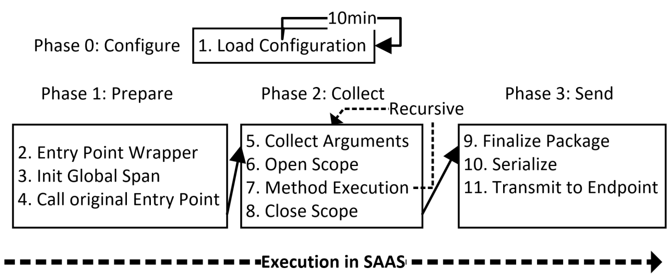

Fig. 2. Basic workflow of instrumented plugins, grouped by the 4 phases: (0) configure, (1) prepare, (2) collect and build, and (3) send telemetry data.

First, we *configure* the tracing by loading configurations. This phase is executed after an assembly update (i.e., when a plugin is first loaded) and then every 10 minutes to refresh the configuration. Secondly, we *prepare* the plugin execution (starting the trace). Then we *collect* data about method calls and lastly *send* the collected data. The *prepare* stage and *send* stage are only executed once per plugin execution (at the start and the end respectively), while *collect* is performed for every (recursively) called method. We explain each phase and their steps in the following sections.

Compared to traditional, on-premise solutions with access to hardware metrics such as CPU utilization or memory utilization [7], SaaS environments solely focus on source code execution times (since no hardware info is available due to limited API access). In SaaS environments, the telemetry collector logic must be adjusted to handle technical limitations, namely that we have no access to a shared file system and that we have strict limited hardware resources (e.g., memory is limited to $\tilde{2}56$ MB per assembly on PP). These adjustments include, for example, storing configuration in memory instead of configuration files and carefully limiting the number of live objects used for building the traces.

A major consideration is also that code instrumentation always adds additional runtime costs caused by the collector code statements (e.g., time spent on recording timestamps, collecting arguments, intercepting exception handlers, and so on). A performance runtime overhead of 5% to 10% caused by telemetry collection code is typically considered acceptable [11]. We will examine detailed performance metrics from various use cases executed within a SaaS environment.

A. Phase 0: Configure

Capabilities to configure the collector (for example providing API keys or setting the level of tracing detail) from the outside are essential. Due to their distributed nature, SaaS environments are not suitable to store configurations as plain files. Rather, an HTTP call to obtain the configuration file from a remote location has to be issued. Since such calls can be expensive, we try to keep the number of configuration refreshes low.

Step 1 - Load configuration: An assembly (.dll) contains one or more plugins. The first executed plugin within an assembly (loading the assembly) initially loads the configuration. It then stores the configuration values as static object to make the configuration accessible to all other plugins within the same assembly on subsequent plugin executions. It is important to prevent other plugins from the same assembly from running in parallel during that phase, otherwise the configuration would be unnecessarily loaded multiple times.

Usually we try to keep the configuration as long as possible, however we need to check for configuration changes periodically. For traditional systems it is easy to control how long the configuration will be kept until it reloads (e.g., using a timer to reload every 10 minutes). However, in PP only an event can trigger code executions. Thus, every assembly needs to wait for the next event to decide if the configuration should be reloaded. Each assembly stores its own configuration (i.e. as a static object), that means less active assemblies will not immediately retrieve the new configuration. In addition to that, three additional things must be considered for PP:

1) Clustering & Geographic replication: Usually assemblies are loaded on servers within the same region, but in failover scenarios they can also be loaded in another region. → Configuration must be reloaded in new region.
2) Assembly recycles: Once an assembly has been loaded, it usually stays within the AppDomain for a while but an assembly reload can occur anytime. → Assembly reload timing is not predictable.
3) Object instance disposal: PP recreates plugin object instances at any time. Plugins must be stateless and are not thread-safe. → Object-level variables are discarded after each plugin execution, thus configuration must be stored in a static context.

Alternative implementations using bi-directional connections (e.g., WebSockets) are also not available due to timeout restrictions, which will kill the plugin execution after a specific time. In case of PP this happens after 2 minutes for synchronous plugins [12].

Learning: Configuration changes are not applied immediately

Conclusion: A platform event or assembly recycle is needed to check for configuration changes. Changes may not be synchronized between assemblies, as each updates its static configuration object independently.

B. Phase 1: Prepare

In the case of PP, plugins must implement a given interface with one generic method `Execute`, as shown in Listing 1. Via the `IServiceProvider` instance important context variables such as event name, etc. can be retrieved. Listing 2 shows a basic implementation of the plugin interface and how to retrieve objects from the `ServiceProvider` instance. When a plugin is executed, this phase handles the initial trace bootstrapping (startup logic such as allocation of memory).

```
public interface IPlugin {
  void Execute(IServiceProvider serviceProvider);
}
```

Listing 1. The default entry point for PP plugins provided by the SDK. Every plugin must implement `Execute(IServiceProvider)`.

```
public class VeryBadAccountPlugin : IPlugin {
  void Execute(IServiceProvider serviceProvider) {
    var demo =
        serviceProvider.GetService(typeof(IDemoService));
    demo.SomeLongRunningOperation();
  }
}
```

Listing 2. A simple plugin implementation without instrumentation

Step 2 - Entry point replacement: When reloading a .dll, we instrument every method call by injecting additional code before and after calls to open and close spans. A simple tracing approach is to override the original interface method implementation with some telemetry collector bootstrapping code, as exemplary shown in Listing 3. The original method body will be moved to a new method `Execute_Wrapped` (line 7), which will then be called by the overridden body in `Execute` (line 4). Our telemetry collector (implemented in our custom class `ReportService`) uses 3 methods to manage the spans: `Init` (to open the first span and set initial variables), `Report` (to open a span) and `Close` (to close the span). These methods must be developed with great care. The implementation details of `ReportService` will not be discussed due to closed source restrictions, however the simplified versions of the 3 main methods are shown in upcoming examples. Besides the complexity of how intermediate language code has to handle try/catch/finally blocks (there are various restrictions and special instructions defined by the official .NET specification [10]), memory leaks are a high-risk. Due to the low limits of shared memory, a few unnecessary references to otherwise "garbage objects" held by the telemetry collector can easily lead to `OutOfMemoryException`. PP wraps these kind of exceptions, so it can be very hard to understand the issues and identify the root cause. Often the only information displayed is *"Unexpected Error. Try again"*.

```
1  public class VeryBadAccountPlugin : IPlugin {
2    public void Execute(IServiceProvider serviceProvider) {
3      ReportService.Init(this, "f5a29daf");
4      Execute_Wrapped(serviceProvider);
5      ReportService.Close("f5a29daf");
6    }
7    public void Execute_Wrapped(IServiceProvider
         serviceProvider) {
8      Type type1 = typeof (IDemoService);
9      ReportService.Report(
10       "System.IServiceProvider.GetService(System.Type)",
11       "0b7f9dd5"
12     );
13     object demo = serviceProvider.GetService(type1);
14     ReportService.Close("0b7f9dd5");
15     ReportService.Report(
16       "IDemoService.SomeLongRunningOperation()",
17       "0b7f9dd6"
18     );
19     demo.SomeLongRunningOperation();
20     ReportService.Close("0b7f9dd6");
21   }
22 }
```

Listing 3. A simplified plugin implementation after instrumentation.

Step 3 - Init global span: We need to initialize a data structure to store the collected spans and additional details (such as a `traceID` and the start time). This step will also create the first span (global span), which will then be the parent for all direct calls within `Execute_Wrapped`. Our instrumentation does not take place on source code level but rather injects multiple IL statements into already compiled assemblies. For instance, Listing 4 is what our instrumentation tool adds to the assembly to add the `ReportService.Report(...)` call (our method to open a span) from line 9 in Listing 3 .

```
...
IL_000c: ldstr
      "System.IServiceProvider.GetService(System.Type)"
IL_0011: ldstr "0b7f9dd5"
IL_0016: call void ReportService::Report(string, string)
...
```

Listing 4. Parts of the instrumented plugin.

Step 4 - Call original entry point: Call the original `Execute` method, which is now `Execute_Wrapped`. See Listing 3, line 4.

C. Phase 2: Collect

During the *Collect* phase, information about every method call is collected recursively by calling the instrumented methods. This could lead to deep call stacks which must be considered in regards to stack-overflows / memory limitations. A simplified description of our actual implementation is that we count the call stack depth and sum up the total memory allocation of collected spans. Both values can then be compared to constant thresholds defined by us (i.e., a maximum of 1MB of total memory for all spans and a maximum depth level of 16) to ignore call stacks out of these limits. Since such optimizations are not trivial, we discuss them in more detail in subsection IV-A.

Step 5 - Collect arguments: Although the argument name for a given method could be retrieved via reflection, it is much faster to use hardcoded argument names injected while code instrumentation (See Listing 3, line 10). It is also very important to avoid any side effects that could be caused by, for example, the serialization of objects.

Step 6 - Open span: Creates a new span as child of either the global span opened at step 3 or the out instrumented method.

Step 7 - Execute original method: Calling the original method (here e.g. `GetService()`). In case of user-defined methods, this results in a call of another instrumented method (back to step 5). The execution of the original method can result in many different scenarios, such as exceptions or timeouts, which has to be handled gracefully.

Step 8 - Close span: Stop measurement and cleanup.

D. Phase 3: Send

The last phase could be reached either via a successful full execution (i.e., the last statement within the entry point body is reached), or when an exception occurs within the plugin. In both cases, the complete trace will be sent.

Step 8 - Finalize package: This happens after closing the global span (no further actions needed) or due to a thrown exception (remaining open spans must be closed recursively). It is essential to consider edge cases such as plugin timeouts. In the event that the plugin has been canceled by the platform due to an execution duration timeout (2 minutes [12]) the close span call (injected code) will not be executed. That would mean static resources would remain in memory until the next assembly recycle period. This can be circumvented by monitoring logic, which closes the global span automatically if the total duration comes close to 2 minutes.

Step 9 - Serialize: The built trace must be prepared to be sent to the endpoint. gRPC (acronym for gRPC Remote Procedure Calls) [13] has emerged as a viable option, but normal compressed JSON is also possible. It's also important to consider the collector's total memory consumption versus performance. Assuming that all arguments of every called method are being held until this step, it could be more efficient to already serialize and release them in an intermediate step, allowing the GC to clean memory earlier. This is discussed in subsection IV-A.

Step 10 - Transmit: The serialized trace will be sent to an intermediate OTel endpoint. The intermediate endpoint is needed to post-process the data, before it can be send to the final OTel endpoint. Afterwards all resources related to this specific trace will be released. Together with step 9 this is the most time-consuming part. More details based on actual numbers will be shown shortly.

IV. Further Challenges When Observing SaaS Environments

In addition to the limitations already mentioned, there are a few other differences to conventional systems. These secondary constraints are not usually considered directly, as they do not occur in proof of concept implementations. However, supposing that the entire software development process (develop, test, release) is considered through to production, these borderline cases must also be addressed.

A. Memory limitations

We can split the memory used for telemetry data into two groups: (1) objects that live on a global level (e.g., configuration) that are kept alive independent of plugin executions, and (2) objects that are only used per plugin executions (e.g., spans). The first group thus poses a fixed amount of memory which reduces the remaining working set.

Available memory: Listing 5 shows the behavior of PP in terms of memory management, it limits the memory per assembly. That means if an assembly holds a static object with a size of 256MB (line 2), any further memory allocation request (line 6) will automatically lead to an exception.

```
1 public class AccountPlugin {
2   static bytes[] configuration = new bytes[256 * 1024 *
        1024];
3   static bytes[] tracingData = new bytes[];
4
5   public void Execute(IServiceProvider provider) {
6       byte[] data = new byte[1];
7   }
8 }
```

Listing 5. A simple plugin which would throw an out of memory exception.

Thus, our space for optimizations is the trace collector, more specifically how long we hold argument objects. Instead of keeping alive all method call arguments until serializing the complete trace at the end (see Step 9 in section III), we can already serialize arguments directly within `ReportSerivce.Open()`. That makes it possible to release the arguments right after the call to the actual method at the cost of increasing the size of tracingData (line 5) due to new string values (serialized argument values).

In general, the main problem is missing information an remaining memory due to API limitations. Our solution is to use details from previous executions to make decision whether arguments should be collected. We call this estimated execution plan (EEP), which will be introduced in section VI. The EEP also includes information about exception that occured in previous executions (such as `OutOfMemoryException`), which can further inform how and when to serialize arguments.

B. Assembly loading times

Most SaaS platforms do not share technical details about their code loading process, but for PP (based on .NET Framework) it can be assumed that AppDomains are used with many policies via the normal .NET AppDomain Policy API. Given this assumption, we assume that the whole AppDomain must be recreated to be able to reload assemblies (AppDomains do not support reloading of separate assemblies). This fact must be considered when setting the configuration reload times.

C. Discrepancies between local server times

Multiple plugins, triggered by the same event, should appear in the same trace. Yet, the nature of distributed systems causes time differences when using hardware-based measuring methods and different machines (see C# Stopwatch). These differences must be fixed during post-processing.

D. Restricted access

Classic instrumentation tools would instrument classes (e.g. `System.Net.Http.HttpClient`) provided by system libraries. For security & performance reasons many system libraries (such as .NET Core CLR [14]) are managed by the platform and cannot be modified. Our workaround is to identify certain API (e.g. `System.Net.Http.HttpClient`) and inject method calls to a telemetry collector right before the actual method execution.

E. Assembly size

A common restriction in SaaS environments is the single DLL policy, that means a single DLL cannot reference other DLLs directly (excluding SDK DLLs, which are automatically resolved by the platform). There are few workarounds such as ILMerge [15], a tool published by Microsoft R&D to merge two .DLL files into one single file. For PP the assembly size limit is 8 MB. Including common libraries like gRPC, Newtonsoft JSON, etc. could exceed the limit quite easily supposing that the original assembly size is taken into account. Regarding monitoring / collector code implementations, this could be a hard blocker assuming that there are dependencies to 3rd party libraries. So, the telemetry collector code should (in the best case) not use any libraries at all.

F. Deployment

In the case of PP, deployments are done via simple zip package uploads, which include all components. The downside of this is that instrumented DLLs will be overridden. This stops the monitoring process. A straightforward way to work around this problem is to periodically compare plugin registration metadata (i.e. last modified on date) with the timestamp (saved in a database within our platform) of the last instrumentation of that specific assembly.

V. Common Performance Impact Factors

In this section, we explore common performance impact factors such as the influence of environment patterns (e.g., plugin communication) or serialization overhead introduced by telemetry collectors. All measurements were taken under controlled system conditions of a normal PP environment:

- EMEA Region (MS Cloud in West Europe) (Scope is server side only, client region has no impact.)
- Normal production license (MS uses scale groups to handle very high consumption customers. Here, the default scale group is used.)
- 100 sample size (total 200 runs, 100 skipped warmup runs, then filtered by the same server, to ensure integrity.)

A. Typical SaaS application / environment patterns

In this section, we compare the performance metrics of two typical SaaS application pattern: (1) A single simple plugin, i.e., user code, is triggered based on a platform event, and (2) a plugin is triggered based on a platform event, which in turn triggers another plugin to be executed.

1) Simple single execution: Performance metrics reported in this section are based on the plugin shown in Listing 6). This plugin is triggered every time a user account has been updated, and it only logs that it has been executed.

```
class BadAccountPlugin : IPlugin {
  // ... auxilliary helper property TracingService ...
  // ... auxilliary helper function GetTarget() ...
  protected override void Execute(IServiceProvider
       serviceProvider) {
    TracingService.Trace("Demo Plugin started.");
    var target = GetTarget<Account>();
  }
}
```

Listing 6. Simple plugin, executed every time a user account is updated.

Typically, when plugins are executed on PP, multiple parties are involved, as shown in Figure 3. The *event framework* is a PP-internal engine that is responsible for handling requests

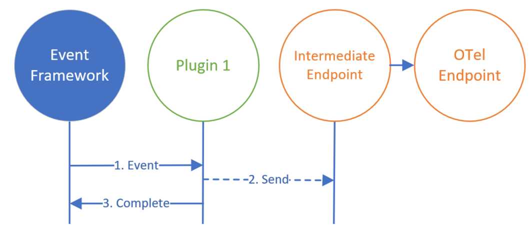

Fig. 3. An illustration of how a simple plugin is executed: PP generates an event, triggers the plugin, in which we then collect and send telemetry data.

and events (such as "Update Account") and triggering relevant components such as *plugins*. Plugins (such as Listing 6 contain the user-defined code (which we instrument using IL modification). Our instrumentation builds its trace, sends data to an *Intermediate Endpoint*, i.e., a webservice used to post-process OTel data, which is then stored in a final *OTel Endpoint*.

a) Telemetry Overhead: In a first analysis, we inspect the performance overhead introduced by our instrumented telemetry code (screenshots shown in this section show the data of the average run of our 100 test runs).

Type Name	Execution Duration
Golet.Plugins.Accounts.BadAccountPlugin,	109

Fig. 4. Execution duration reported by PP for original (i.e., non-instrumented) simple plugin.

Type Name	Execution Duration
Golet.Plugins.Accounts.BadAccountPlugin,	140

Fig. 5. Execution duration reported by PP for instrumented simple plugin.

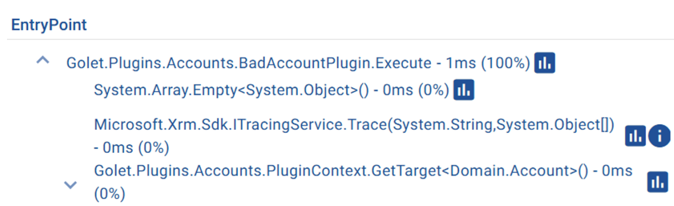

Fig. 6. Execution breakdown *based on our traces* with runtime details for every method call inside the simple plugin.

Figure 4 shows 109ms reported by PP for executing the non-instrumented plugin, i.e., without our tracing IL modifications. The reported 109ms include platform overhead by the event framework such as time spent on event creation.

Figure 5 shows 140ms reported by PP for executing the instrumented plugin, i.e., with our tracing IL modifications including transmission to the OTel endpoint. Thus, on average, our tracing introduces *31ms (28.44%) overhead on average* for a simple plugin according to PP.

Figure 6 shows how our tool can now display the recorded trace. Our traces are started and closes again within the

`Execute` method of the plugin. We can see that the `Execute` method actually only took 1ms (with the methods called within all took less than 1ms, shown as 0ms). Yet, as shown in Figure 5, PP reports 140ms for executing the plugin. As calculated above, 31ms are spent on our tracing. The remaining reported 108ms are thus spent within PP's event framework and are out of our control for further optimization.

b) Assembly Loading: The numbers in the previous section reported on already loaded assemblies. Figure 7 shows that PP reports 1,171ms for *freshly loading* and executing the simple, non-instrumented plugin, while Figure 8 shows that PP reports 1,218ms for freshly loading and executing the simple, instrumented plugin. Executing the instrumented plugin also includes (in addition to recording and sending a trace) initially loading our configuration file.

Type Name	Execution Duration
Golet.Plugins.Accounts.BadAccountPlugin,	1,171

Fig. 7. Execution duration reported by PP for original (i.e., non-instrumented) simple plugin when assembly has to be freshly loaded.

Type Name	Execution Duration
Golet.Plugins.Accounts.BadAccountPlugin,	1,218

Fig. 8. Execution duration reported by PP for instrumented simple plugin when assembly has to be freshly loaded.

Thus, on average, our tracing (including a configuration reload) introduces *47ms (4.01%) overhead on average* when freshly loading and executing a simple plugin according to PP. Comparing these 47ms overhead to the 31ms reported for already loading plugins leaves us with the insight that reloading our configuration takes approximately 16ms.

2) Multiple chained executions: This second version of the plugin (see Listing 7) actually performs work: it updates the address of all contact information (private, business, etc.) related to the updated account.

```
1  class BadAccountPlugin : IPlugin {
2    // ... auxilliary helper property Service and
         TracingService ...
3    // ... auxilliary helper function GetTarget() ...
4    protected override void Execute(IServiceProvider
         serviceProvider) {
5      TracingService.Trace("Demo Plugin started.");
6      var target = GetTarget<Account>();
7
8      using (var eContext = new ServiceContext(Service)) {
9        var contacts = (from e in eContext.ContactSet where
             e.AccountId.Id == target.Id select
             e).ToArray();
10       foreach (var contact in contacts) {
11         eContext.Detach(contact);
12         contact.EntityState = null;
13         contact.Address1_Line1 = target.Address1_Line1;
14         Service.Update(contact);
15       }
16     }
17   }
18 }
```

Listing 7. Plugin updates the adress of all contacts upon account update.

In addition to this, this scenario involves a second plugin that is executed every time a contact is updated (more specifically, the `Update` call on the contact in line 15 in Listing 7 triggers this contact plugin). This flow of operations is also shown in Figure 9.

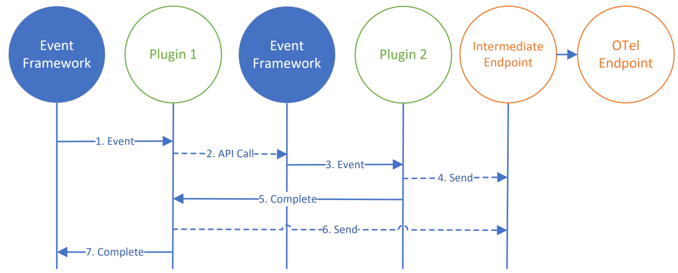

Fig. 9. An illustration of our chained executed workflow: Upon account update, PP generates an event, triggers the first plugin, which in turn triggers further PP events that lead to another plugin being executed.

For reasons of simplicity, the contact plugin, i.e., the second plugin, in this scenario only performs `Thread.Sleep(400)` within its `Execute` method. Since our example account has two associated contacts, each execution of the account plugin leads to two executions of the contact plugin (the second call to the contact plugin is not shown in Figure 9).

a) Telemetry Overhead: We inspect the performance overhead introduced by our instrumented telemetry code.

Time	Type Name	Execution Duration
8:02 PM	Golet.Plugins.Contacts.BadContactPlugin, ...	421
8:02 PM	Golet.Plugins.Contacts.BadContactPlugin, ...	453
8:02 PM	Golet.Plugins.Accounts.BadAccountPlugin,...	1,890

Fig. 10. Execution durations reported by PP for original (i.e., non-instrumented) plugins in chained execution.

The bottom line of Figure 10 shows 1,890ms as the average duration reported by PP for executing the non-instrumented account plugin, i.e., without our tracing IL modifications. The durations of the two non-instrumented contact plugin executions are also reported by PP (454ms and 421ms).

Time	Type Name	Execution Duration
11:50 PM	Golet.Plugins.Contacts.BadContactPlugin, ...	484
11:50 PM	Golet.Plugins.Contacts.BadContactPlugin, ...	531
11:50 PM	Golet.Plugins.Accounts.BadAccountPlugin,...	1,953

Fig. 11. Execution durations reported by PP for instrumented plugins in chained execution.

The bottom line of Figure 11 shows 1,953ms as the average duration reported by PP for executing the instrumented account plugin, i.e., with our tracing IL modifications including transmission to the OTel endpoint. The durations of the two instrumented contact plugin executions are also reported by PP (531ms and 484ms).

Thus, on average, our tracing introduces *63ms (3.33%) overhead on average* for the "outer" account plugin, while it introduces *70ms (16%) overhead on average* for the "inner" contact plugin.

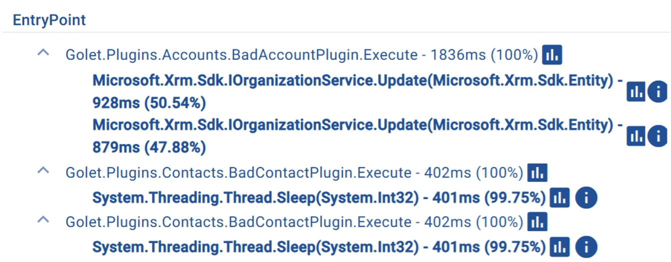

Fig. 12. Execution breakdown *based on our traces* with runtime details for every method call inside the plugins (Partially filtered, showing only durations longer or equal to 1ms).

When we again compare the durations reported by PP itself (Figure 11) with the numbers recorded in our trace (Figure 12), we again see that PP includes platform overhead in its reported numbers. Executing the contact plugin took 531ms and 484ms respectively according to PP, while at the bottom of our detailed breakdown we can see that the `Execute` method of our two contact plugin executions both times took 402ms. But we gain even more insights if we look inside the `Execute` method of our account plugin, i.e., the top half of our detailed breakdown. There we can see that updating the contacts actually took 928ms and 879ms respectively, i.e., nearly double as long as reported by PP. In Table I, we summarize all numbers reported by PP and our tool. Comparing *PP Original* and *PP Instrumented* gives us insight into how much overhead tracing and OTel communication is introducing. Comparing *PP Instrumented* with *Trace*, i.e., the numbers reported based on our traces, clearly shows that the time spent within the platform (for event creation, etc.) is much higher than reported by default PP.

TABLE I
OVERVIEW OF THE DIFFERENT REPORTED DURATIONS. PP ORIGINAL AND PP INSTRUMENTED INCLUDE (CERTAIN) PLATFORM OVERHEAD (POH).

Plugin	PP Original	PP Instrumented	Trace
Account	1,890	1,953	1,836 (only Execute, no POH)
Contact	453	531	928 (incl. POH) 401 (only Execute, no POH)
Contact	421	484	879 (incl. POH) 401 (only Execute, no POH)

Learning: Multiple layers of runtime: (1) Platform Overhead, (2) Telemetry Overhead, (3) Plugin Logic

Conclusion: While PP includes (some) platform overhead in its reportings, our instrumentation only measure within entry points. By inspecting calls that lead to further plugin executions, we can derive even more detailed platform overhead costs.

Based on the observed overheads, telemetry collector code could be considered as an acceptable extension in terms of runtime costs impact for most cases.

B. Serialization Overhead (Monitoring Impact)

To inspect how serialization impacts the overhead cause by telemetry collection, we analyzed the performance of JSON Serializer (C#) [16] used for serialization. We did not measure gRPC due to API limitations. Table II shows a comparison of different serialization metrics for common objects, extracted from code which was executed within a PP plugin.

TABLE II
SERIALIZATION OVERHEAD FOR DIFFERENT KINDS OF OBJECTS.

Object	Description	Size (Byte)	Avg* (ms)
`POCO`	Simple object with 5 properties	98	0.025
`Entity`	Common CRM object (Account). Similar to POCO but with 27 properties	1,687	0.049
`Entity[]`	10 entity objects	15,145	0.396

Learning: Arrays can have a massive impact to the performance.

Conclusion: Maximum serialization depth and limits for array (i.e. only include the first 5 items) must be implemented.

C. Further factors

This section contains performance factors that we have not yet measured in detail but still want to report on.

- Argument Size: Besides arrays, also base64 encoded attributes will result in a large JSON string. This leads to slower transmission to the OTel Endpoint.
- Memory Costs: SaaS is almost always a shared memory environment. That means the collector code is a high critical execution path as it stores data on an assembly level (static variables). With every new collected span, the total memory allocated by the collector is increased. This could lead to `OutOfMemoryExceptions`, so some spans might be dropped or reduced as described in subsection III-C.
- Data Privacy: A common requirement is to mask personal data before leaving the SaaS platform. A regex would be

the worst option here, as it is slow compared to simple checks such as string comparison. Simple string based wildcards (e,g, replace values for certain attributes called Name*), can often fulfill requirements without complex regex evaluations.

VI. Introduction of EEP / Proposal

At Kupp, the goal of an ongoing research project is to analyze, control and reduce observability overhead using automated optimizations based on code execution plans built upon historical tracing data. As a major prerequisite, a clear picture of all monitoring phases at run time in terms of time costs must be defined. This picture is based on the results presented in previous sections of this document. A tool to automatically generate an estimated execution plan (EEP) based on a given entry point (fully qualified method name) and .DLL file(s), which includes the potential time costs for each call including parameter serialization is the expected output of this project. As an extension to already established sampling techniques, this EEP can then be used to decide in real time whether a specific action (i.e. not tracing a method, reducing the detail of argument serialization, ...) should be applied, based on the impact to the overall execution costs. The acceptable overhead limit is set as a percentage relative to the normal execution time (for example, 10% acceptable overhead means based on a normal execution time of 100ms a maximum observability overhead of 10ms in total is allowed). Furthermore, the tool should continuously improve the accuracy of the EEP based on real tracing data.

A. Goals

It is important to highlight the difference between (1) the actual method execution duration and (2) the time used to collect telemetry data. For example, `Thread.Sleep(100)` runs for approx. 100ms (actual method execution duration) and has an instrumentation overhead of less than 1ms. This is due to the fast serialization of a single integer (method argument), however for larger argument objects (e.g., large arrays) this behavior changes. The following sections present more details based on Listing 8.

```
public class VeryBadAccountPlugin : IPlugin {
  public void Execute(IServiceProvider serviceProvider) {
    Thread.Sleep(500);
    var data = new int[5_000_000];
    Process(data);
  }
  private void Process(int[] data) { Process2(data);  }
  private void Process2(int[] data) {  }
}
```

Listing 8. Simple plugin used to show the difference between runtime costs and instrumentation overhead.

Summarized, the targeted benefit of such a tool is to have a way to automatically disable argument serialization or skip whole methods based on expected monitoring costs.

B. Real World Impact

There are two main reasons why this approach is a major improvement to the overall technology:

- Some libraries (such as Newtonsoft, a well known .NET JSON library) have very *deep call stacks*. Even though the instrumentation of simple methods (call to `ReportService.Report(..)`) itself does not heavily impact performance, this changes for methods with deep recursion. A simple approach would be to exclude calls to certain methods of well-known libraries. Yet, this would enforce us to maintain a list of such methods.
- The second problem is the *serialization of large objects*. While it is uncommon (but not impossible) to have very large argument arrays, large argument objects occur in many different scenarios (e.g., images). During the plugin execution, there is no straight forward way to determine the final transmission size of a given object without running the serialization. That means the most efficient way is to lookup the expected costs in the EEP to be able to decide whether serialization should be skipped or not.

C. Process

To enable enhancements based on EEP data, the instrumentation process must be adjusted from the original process (Figure 2) to an extended flow as shown in Figure 13. Traces collected within PP (bottom half) are now fed into an *external processing server* (top half) which is responsible for building and continuously updating the EEP. Published EEPs are then used in the telemetry *Collect* phase to decide whether certain operations should be skipped.

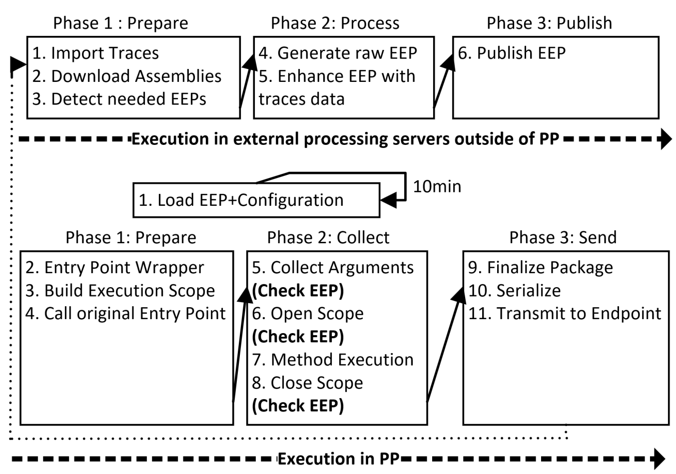

Fig. 13. EEP extended flow based on Figure 2.

a) External Processing Servers: In *Phase 1 (Prepare)*, (1) multiple traces from previous executions are imported to a temporary location, then (2) non-instrumented assemblies must be downloaded. It is important to ensure that the non-instrumented assemblies are used, otherwise the telemetry collector code would also be included in the EEP. The last step is (3) to detect which EEPs are needed depending on components included per assembly.

In *Phase 2 (Process)*, for each entry point (i.e., `Execute()`), (4) all reachable method calls are recursively collected. Circular dependencies are handled by tracking visited methods. (5) Existing trace data (e.g., average execution time) is then aggregated per method and added to the EEP.

Phase 3 (Publish) simply (6) publishes EEPs to make them available to the PP runtime (similar to the configuration data).

b) Execution in PP: The flow in PP has stayed the same as shown in Figure 2 and discussed in section III, except for newly introduced "Check EEP" steps. These steps are used to decide whether a certain operation can be skipped (e.g., whether argument serialization or recording the whole method should be skipped). For example, consider Listing 8 as an example to show the impact of using an EEP. Listing 9 shows simplified tracing data from a normal execution without optimizations based on EEPs. The example plugin first executes `Thread.Sleep(500)` (`TargetMethod` in Listing 9) which needs 519ms to complete (`TotalExecutionTime`), of which the serialization of the arguments (one `System.Int32`) needs less than 1ms (`ArgumentsTimeCosts` value of 0). The next statement (Listing 8, line 4) initiates an array of integers with a size of 5 million elements. This array is used as argument for the next method call to (`VeryBadAccountPlugin.Process()`). Executing this method takes 1,368ms, yet this includes 734ms for argument serialization. Thus, a notable result is that the serialization of an array with 5 million integer elements can easily take over half a second.

Listing 10 shows an EEP that has been built based on multiple full executions (traces) of the plugin. It uses average values based on these runs (in our case 5 executions, see Listing 10, line 4).

Listing 11 then shows a trace collected for the same plugin, yet applying optimizations based on an existing EEP. Considering an expected argument serialization cost of 614ms (see Listing 10, line 13), serializing the argument for the method call was skipped (thus no `Arguments` entry is shown on line 9 in Listing 11). A similar decision to not trace the array argument was made for the method `Process2(...)`, thus also no `Arguments` entry is shown there. This demonstrates the effectiveness of EEPs: by proactively identifying a high-cost serialization operation, the system could selectively disable it, reducing the execution time for that specific call from 1,368ms to just 7ms – a reduction of over 99%.

```
{
  "TargetMethod": "Thread.Sleep(System.Int32)",
  "Arguments": "[{\"millisecondsTimeout\":500}]",
  "Metadata": {"ArgumentsTimeCosts": 0},
  "TotalExecutionTime": 519
},
{
  "TargetMethod": "Process(System.Int32[])",
  "Arguments": "[{\"data\":[0,0,0...]}]",
  "ChildContexts": [{
    "TargetMethod": "Process2(System.Int32[])",
    "Arguments": "[{\"data\":[0,0,0,0...}]",
    "Metadata": {"ArgumentsTimeCosts": 632},
    "TotalExecutionTime": 634,
  }],
  "Metadata": {"ArgumentsTimeCosts": 734},
  "TotalExecutionTime": 1368
}
```

Listing 9. Simplified demo tracing data *without* EEP optimization. (Values are shortened for improved readability)

```
1  "AvgDuration": 1925200,
2  "EntryPoint": {
3    "Name": "VeryBadAccountPlugin.Execute",
4    "Hits": 5,
5    "MonitoringTimeCosts": 93,
6    "Children": [{
7      "Name": ".Thread.Sleep(System.Int32)",
8      "Hits": 5,
9      "MonitoringTimeCosts": 0,
10   },{
11     "Name": "Process(System.Int32[])",
12     "Hits": 5,
13     "MonitoringTimeCosts": 614,
14     "Children": [{
15         "Name": "Process2(System.Int32[])",
16         "Hits": 5,
17         "MonitoringTimeCosts": 676, ....
```

Listing 10. Estimated Exceution Plan (EEP) based on 5 excutions.

```
1  {
2    "TargetMethod": "Thread.Sleep(System.Int32)",
3    "Arguments": "[{\"millisecondsTimeout\":500}]",
4    "Metadata": {"ArgumentsTimeCosts": 0},
5    "TotalExecutionTime": 507,
6  },
7  {
8    "TargetMethod": "Process(System.Int32[])",
9    "ChildContexts": [{
10     "TargetMethod":"Process2(System.Int32[])",
11     "Metadata": { "ArgumentsTimeCosts": 0},
12     "TotalExecutionTime": 7,
13   }],
14   "Metadata": {"ArgumentsTimeCosts": 0},
15   "TotalExecutionTime": 7
16 }
```

Listing 11. Simplified demo tracing data for *with* EEP optimization.

D. *Further considerations*

Although this improvement looks like a straight forward implementation there are a few disadvantages and some side effects in terms of complexity. The configuration source must be extended to include published EEPs for each plugins. Consuming this service from a plugin adds additional runtime costs. Besides that, memory limits must also be taken into account. Especially for large plugins the EEP can grow fast as it must be a context-aware representation of the possible code flow. A simplification to a simple key-value store could easily lead to unwanted results. For example, a shared library method might be called with large, slow-to-serialize objects from one code path but with small, fast-to-serialize objects from another. A simple key-value store mapping the library method to an average cost would be inaccurate; the EEP must therefore be context-aware, tracking costs based on the call stack. Furthermore, changes to the plugin (i.e., deployments) must trigger a reset of the generated EEP.

VII. LIMITATIONS

While this work demonstrates the feasibility of advanced observability on the Power Platform, the practical implementation of real-world telemetry collectors must navigate several inherent limitations.

a) Platform-Specific Operational Constraints::

- Cloud Provider Adherence: Telemetry collection code must operate strictly within the Microsoft Power Platform's sandboxed environment. This entails respecting defined execution time and resource limits [12], utilizing only whitelisted APIs, and understanding that any API

not explicitly supported is effectively off-limits. This can impact choices for fundamental operations like precise time measurement or cryptographic functions.
- Serialization Choices: While gRPC offers performance benefits, the restriction on external libraries often makes it necessary to rely on standard platform capabilities like JSON serialization.
- Argument Collection Overhead: Capturing method call arguments can be resource-intensive. Strategies to optimize this, such as referencing or hashing repeated argument objects rather than full serialization, become important considerations to minimize runtime costs.

b) Complexities in Distributed Environments:

- Geo-Distribution and Endpoint Management: SaaS platforms frequently leverage CDNs and automated geo-failover. This requires robust management of OTel Endpoints to ensure continuous data flow and correct trace routing, especially when traces originate from different servers participating in the same overall user request.
- Trace Context Propagation and ID Management: Ensuring consistent Trace ID generation and propagation (e.g., aligning Power Platform-generated IDs with W3C Trace Context specifications [17], [18]) is crucial for end-to-end tracing, particularly when integrating with other systems. This may involve ID mapping or regeneration.
- Clock Synchronization: OTel does not inherently enforce clock synchronization across distributed systems. This can lead to challenges in accurately ordering events and may necessitate post-processing steps to align trace timelines derived from different sources.

c) Instrumentation Lifecycle and Integrity:

- Deployment Overwrites: Standard deployment mechanisms on the Power Platform will replace existing assemblies, thereby overwriting any IL-level instrumentation. Sustaining observability requires a process to re-apply instrumentation, ideally integrated into a CI/CD pipeline or triggered by deployment notifications.
- Code Signing Post-Instrumentation: The Power Platform mandates strong-name signing for assemblies. Any modification, including IL instrumentation, requires the assembly to be re-signed. Without access to the original signing certificates, this poses a significant challenge.

d) Understanding Platform-Induced Overhead: The SaaS provider's own internal monitoring and platform operations contribute an inherent "Platform overhead." This is largely opaque and uncontrollable by custom collectors but must be acknowledged when presenting performance data and distinguishing it from the overhead introduced by the telemetry.

VIII. Related Work

Monitoring in restricted environments: The problem that restricted environments cannot be auto-instrumented after the build phase is not unique to SaaS. For example, Dynatrace [19] reports that applications that should be shipped to app stores (such as the Google Playstore) must already include a dependency on the instrumentation library before they are compiled. This must be considered to make sure to not mix telemetry collected from development and production environments.

OpenTelemetry: In their work on *Assessing OpenTelemetry's Impact on Application Performance Monitoring*, Gartner Research [20] discusses the adoption of OpenTelemetry by APM vendors and provides insights into how and why to implement OpenTelemetry for observability in organizations. Further, in their similarly named work *Assessing OpenTelemetry's Importance to Application Performance Monitoring* [21] they evaluate the impact of OpenTelemetry on application performance monitoring, particularly for microservices, containers, and Kubernetes.

Tracing besides OpenTelemetry: Janes et al. [22] analyzed 30 different tracing tools with regards to their distinctive features, tool popularity, benefits and issues. One of these tools is the observability framework Kieker [6], [23] which was initially introduced as Kieker Monitoring Framework by van Hoorn et al. in 2012 [24]. Reichelt, Yang et al. [5], [25]–[27] have performed extensive work on measuring and reducing the overhead introduced by monitoring tools, mostly revolving Kieker. There are ongoing efforts to make Kieker and OpenTelemetry traces seamlessly compatible with each other [9], which opens interesting perspectives for future work.

IX. Future Work

Building on the findings of this paper, our future research will proceed along two primary avenues: the full implementation of our adaptive tracing proposal based on our EEPs and the generalization of our methodology to other SaaS platforms.

a) Adaptive Tracing: Our approach, using Estimated Execution Plans (EEPs), aims to keep monitoring overhead within a user-defined performance budget by dynamically adjusting telemetry collection – for instance by selectively disabling argument serialization. This concept of adaptive tracing is not new; for example Mertz and Nunes [28], [29] dynamically adjust sampling rates for certain parts of the code. We plan to build upon such work by exploring more sophisticated predictive models to improve EEP accuracy, potentially incorporating contextual information from the execution trace to make more intelligent real-time decisions.

b) Platforms beyond PP: A second research direction will focus on generalizing our IL-instrumentation methodology to other restrictive SaaS environments. This investigation will seek to identify the core, platform-agnostic principles of post-compilation instrumentation while documenting the necessary platform-specific adaptations for unique execution models, security sandboxes, and API limitations. This will require acquiring deep technical knowledge of each target platform, as details such as memory limits must be verified beyond lab environments to ensure practical viability. The ultimate aim is to develop a more generalizable framework for achieving deep observability in closed-box SaaS systems.

X. Conclusion

This paper examines the growing area of SaaS observability, introducing common problems such as sandboxing, limited API access, and distributed starting points that often make standard monitoring difficult or even impossible in these systems. We then present that detailed telemetry collection is possible in restricted platforms such as the Microsoft Power Platform by injecting Intermediate Language (IL) code. Our performance analysis reveals a key finding: a significant portion of execution overhead originates from the SaaS platform's own operations, an impact that is distinct from and often larger than the overhead introduced by telemetry collection.

By working through these specific SaaS challenges we've explained, our IL modification method becomes vital for getting deep diagnostic information where standard tools can't work. This helps companies stop treating their important applications like 'black boxes' where they can't see inside, providing the clarity needed for solid performance analysis and meeting compliance rules. The understanding gained from our thorough look at SaaS observability challenges, including the careful separation of telemetry versus platform costs, directly informs future improvements. This includes our proposed Estimated Execution Plan (EEP) for smarter telemetry, helping to create more efficient, clear, and compliant ways to observe various SaaS systems.

References

[1] Z. Deng and E. Hai, "Analysis of saas-based e-commerce platform," in *Proc. of the International Conference on E-Business and E-Government (ICEE)*. IEEE Computer Society, 2010, pp. 9–12. [Online]. Available: https://doi.org/10.1109/ICEE.2010.10

[2] Y. Chen, C. Wu, H. Chu, C. Lin, and H. Chuang, "Analysis of Performance Measures in Cloud-based Ubiquitous SaaS CRM Project Systems," *J. Supercomput.*, vol. 74, no. 3, pp. 1132–1156, 2018. [Online]. Available: https://doi.org/10.1007/s11227-017-1978-x

[3] R. Wang, X. Tian, and S. Ying, "Performance Issue Monitoring, Identification and Diagnosis of SaaS Software: A Survey," *Frontiers Comput. Sci.*, vol. 19, no. 1, p. 191201, 2025. [Online]. Available: https://doi.org/10.1007/s11704-023-2701-0

[4] T. Muhammad, Z. Halim, and M. A. Khan, "Visualizing Trace of Java Collection APIs by Dynamic Bytecode Instrumentation," *J. Vis. Lang. Comput.*, vol. 43, pp. 14–29, 2017. [Online]. Available: https://doi.org/10.1016/j.jvlc.2017.04.006

[5] D. G. Reichelt, L. Bulej, R. Jung, and A. van Hoorn, "Overhead Comparison of Instrumentation Frameworks," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering, (ICPE)*. ACM, 2024, pp. 249–256. [Online]. Available: https://doi.org/10.1145/3629527.3652269

[6] S. Yang, D. G. Reichelt, R. Jung, M. Hansson, and W. Hasselbring, "The Kieker Observability Framework Version 2," in *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2025, pp. 11–15. [Online]. Available: https://doi.org/10.1145/3680256.3721972

[7] N. Bissig, "Unified Application Observability in Heterogeneous Distributed Systems," 2024. [Online]. Available: https://opus4.kobv.de/opus4-hm/frontdoor/deliver/index/docId/590/file/Bissig_2024_MA.pdf

[8] OpenTelemetry Community, "OpenTelemetry Specification 1.40.0," December 2024. [Online]. Available: https://opentelemetry.io/docs/specs/otel/

[9] D. G. Reichelt, M. Hansen, S. Yang, and W. Hasselbring, "Interoperability From Kieker to OpenTelemetry: Demonstrated as Export to ExplorViz," *Softwaretechnik-Trends*, vol. 45, no. 1, pp. 20–22, 2025. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_45_Heft_1/SSP24_23_camera-ready_5445.pdf

[10] Ecma International. (2012, June) Common Language Infrastructure (CLI). [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf

[11] P. Lengauer, V. Bitto, and H. Mössenböck, "Accurate and Efficient Object Tracing for Java Applications," in *Proc. of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2015, pp. 51–62. [Online]. Available: https://doi.org/10.1145/2668930.2688037

[12] Microsoft. (2023) Analyze Plug-in Performance (Microsoft Dataverse) - Power Apps. [Online]. Available: https://learn.microsoft.com/en-us/power-apps/developer/data-platform/analyze-performance

[13] M. Z. Ain, R. Ardiansyah, S. A. Pratama, M. Akbar, and N. T. Lapatta, "Comparative Performance Analysis of GRPC and Rest API Under Various Traffic Conditions and Data Sizes Using a Quantitative Approach," *J. of Applied Informatics and Computing*, vol. 9, no. 2, pp. 450–457, March 2025. [Online]. Available: https://jurnal.polibatam.ac.id/index.php/JAIC/article/view/9276

[14] M. N. Team. (2025) CoreLib Design. [Online]. Available: https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/corelib.md

[15] Microsoft Research. (2024, December) ILMerge. [Online]. Available: https://github.com/dotnet/ILMerge

[16] J. C. Viotti and M. Kinderkhedia, "A Benchmark of JSON-compatible Binary Serialization Specifications," 2022. [Online]. Available: https://arxiv.org/abs/2201.03051

[17] OpenTelemetry. Context propagation. [Online]. Available: https://opentelemetry.io/docs/concepts/context-propagation/

[18] World Wide Web Consortium (W3C). (2021) Trace Context. [Online]. Available: https://www.w3.org/TR/trace-context/#interoperating-with-existing-systems-which-use-shorter-identifiers

[19] Dynatrace. (2025) Instrument Android Apps. [Online]. Available: https://docs.dynatrace.com/docs/observe/digital-experience/mobile-applications/instrument-android-app

[20] D. Hassemer. (2021, September) Assessing OpenTelemetry's Impact on Application Performance Monitoring. [Online]. Available: https://www.gartner.com/en/documents/4006096

[21] D. Cummings. Assessing OpenTelemetry's Importance to Application Performance Monitoring. [Online]. Available: https://www.gartner.com/en/documents/4429299

[22] A. Janes, X. Li, and V. Lenarduzzi, "Open Tracing Tools: Overview and Critical Comparison," *J. Syst. Softw.*, vol. 204, p. 111793, 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111793

[23] W. Hasselbring and A. van Hoorn, "Kieker: A Monitoring Framework for Software Engineering Research," *Softw. Impacts*, vol. 5, p. 100019, 2020. [Online]. Available: https://doi.org/10.1016/j.simpa.2020.100019

[24] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis," in *Third Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*. ACM, 2012, pp. 247–248. [Online]. Available: https://doi.org/10.1145/2188286.2188326

[25] D. G. Reichelt, S. Kühne, and W. Hasselbring, "Overhead Comparison of OpenTelemetry, inspectIT and Kieker," in *Short Paper Proceedings of Symposium on Software Performance (SSP)*, ser. CEUR Workshop Proceedings, vol. 3043. CEUR-WS.org, 2021. [Online]. Available: https://ceur-ws.org/Vol-3043/short3.pdf

[26] ——, "Towards Solving the Challenge of Minimal Overhead Monitoring," in *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2023, pp. 381–388. [Online]. Available: https://doi.org/10.1145/3578245.3584851

[27] S. Yang, D. G. Reichelt, and W. Hasselbring, "Evaluating the Overhead of the Performance Profiler Cloudprofiler With MooBench," *Softwaretechnik-Trends*, vol. 44, no. 4, pp. 15–17, 2024. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_44_Heft_4/SSP24_04_camera-ready_0409.pdf

[28] J. Mertz and I. Nunes, "Software Runtime Monitoring with Adaptive Sampling Rate to Collect Representative Samples of Execution Traces," *J. Syst. Softw.*, vol. 202, p. 111708, 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111708

[29] ——, "On the Practical Feasibility of Software Monitoring: A Framework for Low-impact Execution Tracing," in *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS@ICSE)*. ACM, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1109/SEAMS.2019.00030