# Melange: A Hybrid Approach to Tracing Heterogeneous Distributed Systems

Gonçalo Garcia
Instituto Superior Técnico
Lisbon, Portugal
goncalotgarcia@tecnico.ulisboa.pt

## ABSTRACT

Software systems are becoming increasingly distributed to deal with the performance and availability requirements of modern users, as well as an ever-increasing amount of data collected from a global user base. While distributed systems pose numerous advantages, they present a challenge for the developers who wish to debug and analyze their performance. Some works have used distributed tracing to offset these difficulties by providing a high level view of the end-to-end execution of single requests, which can be annotated with debugging and performance metrics. However, most of these tools require the modification of the traced components which is often not viable or even possible. Other works have attempted to trace systems as sets of black-boxes, through statistical inference based on non-intrusive data-capturing, or by leveraging expert knowledge of the system's components. While these systems require no modification of the traced software, they are often slow, susceptible to false positives, or require very low-level knowledge of components which is difficult for users of closed source software. In this document, we present a survey of the current state-of-the-art in distributed tracing, and propose an alternative distributed tracing tool (that operates as a middleware layer) that combines source-code modification and passive meta-data capturing to implement distributed tracing in heterogeneous distributed systems containing black-boxes. We attempt to do this by leveraging commonly available open-source tools.

## KEYWORDS

distributed tracing, distributed systems, monitoring, instrumentation, performance engineering

## 1 INTRODUCTION

Modern software systems are growing in scale and, as teams become larger to accommodate the increase in system complexity, it becomes appealing to develop large-scale software as a set of smaller decoupled services with independent development and release cycles. Such architectures are designed with scalability and productivity in mind, as this separation of concerns no longer requires that developers have a low level understanding of all parts of the system.

While this approach to software development is beneficial, and often outright necessary, it adds a significant overhead to the debugging process. Standard halting debuggers are rarely helpful as they are mostly unable to track execution across process boundaries, and in the event that they are, this is usually reserved for highly homogeneous systems [4]. Even the infamous, but often useful "printf debugging" is of little help in this situation, as many times developers are forced to sift through a multitude of log files spanning across several services and physical machines, in order to track the origin of a single faulty request. The decoupling of modules and teams also poses a challenge to debugging as developers might have to follow bugs into other unknown (to them) modules [13].

Some works have attempted to solve the distributed debugging problem by recording a distributed checkpoint of the system state as the execution reaches breakpoints [22]. This has a similar effect to a halting debugger, as the developer is able to analyze the global state of the system at set points, except without interrupting the execution. Although these methods can be effective, they are quite expensive, and a lot of the faults associated to distributed systems are very dependent on timing, hardware and network state, making them hard to reproduce. In these situations it might be more useful to collect information about previously occurred faulty (and correct) requests as they pass through the system, a strategy known as distributed tracing.

Distributed tracing aims to provide a detailed view of the execution of any given request across all of the components of a system. It can be thought of as a centralized log containing causally ordered [10] entries from all components that took part in a single execution. This could very easily be expanded to visual tools that help developers understand and debug their systems. When developing a tracing enabled application it is possible to monitor services or components and pinpoint where faults occur.

Major companies like Google (Dapper [18]), Facebook (Canopy [13]), Etsy [20], JD (JCallGraph [11]) and others have developed their own custom distributed tracing tools, as their systems grew beyond what a single person or team could reasonably understand. However, implementing distributed tracing is, in most cases, not straightforward, which leads to most companies avoiding the endeavor.

### 1.1 Shortcomings in Distributed Tracing

Tracing platforms like the ones deployed in major technology companies are able to reconstruct the path of a request and causally associate debugging information to each point in its execution. To do this, however, developers must instrument their modules to propagate tracing specific meta-data through requests and thread synchronization points. This is possible because companies like Google and Facebook have perfect vertical integration of their stack of software components, and are able to modify them to fit their needs.

The instrumentation approach is straightforward and provides great visibility into the application, which lead to the development of Open-Source instrumentation frameworks that provide interoperability with the most commonly used tracing engines. However, these frameworks are tailored for RPC-based executions, and systems based on other paradigms might require extensive change in order to capture end-to-end traces [13]. Even in suitable programs, instrumentation may be undesireable to some as it clutters the source-code with tracing specific clauses adding significant complexity.

Companies that do not wish, or are unable, to rely on instrumentation may be able to introduce tracing into their platform by using common frameworks or middleware with support for distributed tracing (for example, Finagle [1], gRPC [2], Spring Framework [3]). The amount of readily available components that support tracing is ever increasing, but it is still a small minority, making this approach not viable for every use-case.

Most companies have systems which may be comprised of several black-box components which do not support distributed tracing and cannot be modified. In these cases teams may rely on tracing the in-house modules that interface with the black box components as an attempt to have some sort of observability into the system. This

[1]https://github.com/openzipkin/zipkin-finagle
[2]https://github.com/grpc-ecosystem/grpc-opentracing
[3]http://spring.io/projects/spring-cloud-sleuth

is not optimal because faults may be introduced by the black-boxes and therefore hidden from the trace.

## 1.2 Proposed Solution and Contributions

The problem we aim to solve in this work - tracing heterogeneous distributed systems - can be thought of as two distinct sub-problems with correspondingly different solutions. The first being "sustainable" instrumentation of modifiable (white-box) components, and the second tracing non-modifiable black-boxes.

To tackle the first problem, we propose Melange, a novel instrumentation framework that provides a clearer and less verbose API than current alternatives, while showing comparable functionality and performance. As for the second problem, we take a step back from the lower-level approaches of the literature and instead focus on leveraging Aspect-Oriented Programming to instrument the black-box's drivers, allowing us to extract uniquely identifying process-level meta-data that will label the client-side traces collected by Melange, and provide enhanced visibility into black-box clusters.

# 2 MELANGE

Melange[4] is comprised of an API and several implementations which provide different levels of tracing. To simplify development and to minimize the amount of dependencies placed on the users, each implementation was developed as a separate module that can have individual development and release cycles. Following the Interface Segregation Principle [15], we designed smaller APIs that can be used independently, or together if the instrumented application so requires.

In total there are three APIs: the **BaseTracing API** (used for single-threaded, single-process applications), the **TracingWithId API** (used for single or multi-threaded, single-process applications, that feature a uniquely identifying event ID.), and finally **TracingWithContext API** (Used for single or multi-threaded, single-process or distributed applications).

## 2.1 BaseTracing API

The BaseTracing API, described in Listing 1, is the simplest but it is also the least flexible of all the APIs we have developed. Through this API, context is passed between tracepoints by storing it in thread-local variables.

```
public interface Tracing {
    <R> R newTrace(Supplier<R> toTrace, String description);
    void newTrace(Runnable toTrace, String description);
    <R> CompletableFuture<R> newTraceAsync(Supplier<CompletableFuture<R>>
        ↪ toTraceAsync, String description);
    <R> Promise<R> newTracePromise(Supplier<Promise<R>> toTraceAsync, String
        ↪ description);
    <R> R addToTrace(Supplier<R> toTrace, String description);
    void addToTrace(Runnable toTrace, String description);
    <R> CompletableFuture<R> addToTraceAsync(Supplier<CompletableFuture<R>>
        ↪ toTraceAsync, String description);
    <R> Promise<R> addToTracePromise(Supplier<Promise<R>> toTraceAsync, String
        ↪ description);
    boolean isActive();
}
```

**Listing 1: BaseTracing API definition**

### 2.1.1 *Base Instrumentation Routines.*

Ignoring overloads, one can immediately split the API into two different semantic classes, which we will refer to as `newTrace` and `addToTrace` methods.

The **`newTrace`** methods, as the name suggests, signal the beginning of a new trace. These methods must not inherit context from previous traces in the caller thread. Additionally, the reference to the generated span must always be available to subsequent spans, including after it has finished (which might happen if the execution continues asynchronously).

The **`addToTrace`** methods add new spans to a previously started, overarching trace. In the BaseTracing API, spans created by calling `addToTrace` become children of the context stored in the caller thread's local memory, i.e., the span that is active during the beginning of this new span. Unlike their `newTrace` counterparts, the reference to these spans will be lost as soon as it finishes, and the most recent unfinished span (or the root) will become active.

### 2.1.2 *Capturing Asynchronicity.*

Melange provides overloads for Java's `CompletableFuture` interface, which is often used to execute long running computation asynchronously. Through our API, capturing the full duration of asynchronous computations running in a `CompletableFuture` is as simple as tracing a method returning `CompletableFuture` with the correct overload. Despite this simplicity, we are well aware that not all software systems have evolved to use this feature, and many legacy projects rely on custom `CompletableFuture`-like classes. Consequently, the library features introduced a simple `Promise` API, that allows custom classes to benefit from our library's functionality. These two features fulfill our of objective of easily tracing asynchronous behavior.

## 2.2 TracingWithId API

Commonly, high performance distributed systems have non-linear execution paths, where one request might be processed by multiple threads, not as a series of asynchronous jobs, but as a pipeline where each thread applies a transformation on the output of its predecessor. Most of the time, these pipelines rely on thread-pools to minimize the cost of thread creation. In environments such as this one, it is not possible to rely on thread-local variables to store a trace's context, as the same thread may be summoned to handle different parts of the pipeline, resulting in incorrect traces.

```
public interface TracingWithId {
    <R> R newTrace(Supplier<R> toTrace, String description, String eventId);
    void newTrace(Runnable toTrace, String description, String eventId);
    <R> CompletableFuture<R> newTraceAsync(Supplier<CompletableFuture<R>>
        ↪ toTraceAsync, String description, String eventId);
    <R> Promise<R> newTracePromise(Supplier<Promise<R>> toTraceAsync, String
        ↪ description, String eventId);
    <R> R newProcess(final Supplier<R> toTrace, final String description, final
        ↪ String eventId);
    void newProcess(final Runnable toTrace, final String description, final
        ↪ String eventId);
    <R> CompletableFuture newProcessFuture(final Supplier<CompletableFuture<R>>
        ↪ toTrace, final String description, final String eventId);
    <R> Promise<R> newProcessPromise(final Supplier<Promise<R>> toTrace, final
        ↪ String description,final String eventId);
    <R> R addToTrace(Supplier<R> toTrace, String description, String eventId);
    void addToTrace(Runnable toTrace, String description, String eventId);
    <R> CompletableFuture<R> addToTraceAsync(Supplier<CompletableFuture<R>>
        ↪ toTraceAsync, String description, String eventId);
    <R> Promise<R> addToTracePromise(Supplier<Promise<R>> toTraceAsync, String
        ↪ description, String eventId);
    TraceContext currentContextforId(final String eventId);
    boolean traceHasStarted(final String eventId);
}
```

**Listing 2: TracingWithId API definition**

In such situation, the frameworks must provide a way to explicitly pass context between instrumentation routines, usually by representing the context as an object. On one hand, this approach is flexible and easy for developers to understand, as it is no different from passing around data objects in code, but on the other hand it might require structural source code modification, such as adding tracing context parameters to method signatures. This is usually not optimal, as it tightly couples tracing instrumentation and application code (once again violating our first objective).

To avoid this tight coupling, we introduce the TracingWithId API, which is able to leverage application-specific event IDs to reconstruct traces that cross thread boundaries, thus reducing the need for structural source-code modification. Previous works [5] have also leveraged uniquely identifying event IDs for trace reconstruction purposes, as this is considered both a common feature, and an easy change.

The main change between the BaseTracing API and the TracingWithId API, as Listing 2 shows, is simply the addition of a new method argument for passing the event ID. This was intentional, as each API should be able to stand on its own and be used independently.

### 2.2.1 *Tracing across Threads.*

The naive approach to using event IDs for trace reconstruction

[4]https://github.com/feedzai/dist-tracing

would be to make every newly created span a child of the previous span associated to the event ID. However, this approach is flawed, and heavily dependent on the timing between operations. Consider an example where traced method A calls two traced methods B and C, the first one (method B) being asynchronously, and the second (method C) synchronously.

Executing this behavior multiple times could lead to three different traces:

**A → B - A → C** This is the correct trace and happens if method B finishes before method C starts, or vice-versa.

**A → B → C** Incorrect trace which will occurr if method B starts before C, but C starts before B has finished.

**A → C → B** Also incorrect and the opposite of the previous example, will happen if C starts before B and B starts before C finishes.

It should be obvious that a deterministic activation relationship between trace points should not result in a non-deterministic span-tree. For that reason, we cannot simply create a relationship between the new span and the most recent one.

After careful review of several programs and their expected traces we could conclude that when a span is created in a new thread and there is no previous trace-context in that thread (or the context is unreliable due to thread reuse) its parent must be the context active in the previous thread. We based our `TracingWithID` on the aforementioned approach, making sure that it is clear to developers that, when tracing a code-fragment, one is able to fetch the previous thread's context, instead of relying on the thread-local context, by passing an event ID as parameter.

Naturally, like most out-of-band solutions to this problem, ours is not infallible. In some cases where multiple asynchronous computations are started in sequence we will once again be presented with non-deterministic traces. In such situations the only solution is to explicitly propagate a context object, which leads us into our last API.

## 2.3 TracingWithContext API

The `TracingWithContext` API, was developed to tackle two situations: (1) tracing asynchronous executions where causality cannot be inferred using event IDs, and (2) provide the ability to serialize and deserialize the tracing state, so that it can be propagated between processes.

```
public interface TracingWithContext extends Tracing {
    <R> R newProcess(final Supplier<R> toTrace, final String description, final
        ↪ TraceContext context);
    void newProcess(final Runnable toTrace, final String description, final
        ↪ TraceContext context);
    <R> Promise<R> newProcessPromise(final Supplier<Promise<R>> toTrace, final
        ↪ String description, final TraceContext context);
    <R> CompletableFuture<R> newProcessFuture(final Supplier<CompletableFuture<R
        ↪ >> toTrace, final String description,final TraceContext context);
    <R> R addToTrace(Supplier<R> toTrace, String description, TraceContext
        ↪ context);
    void addToTrace(Runnable toTrace, String description, TraceContext context);
    <R> CompletableFuture<R> addToTraceAsync(Supplier<CompletableFuture<R>>
        ↪ toTraceAsync, String description, TraceContext context);
    <R> Promise<R> addToTracePromise(Supplier<Promise<R>> toTraceAsync, String
        ↪ description, TraceContext context);
    Serializable serializeContext();
    TraceContext deserializeContext(Serializable headers);
    TraceContext currentContext();
    TraceContext currentContextforObject(final Object obj);
}
```

**Listing 3: TracingWithContext API**

As with `TracingWithId` this API (shown in Listing 3) is very similar to `Tracing` with the exception of a new `TraceContext` parameter. Since the API needed to be completely vendor agnostic, and implementable on top of existing instrumentation frameworks like OpenTracing or OpenCensus, we created a `TraceContext` interface to not only represent the tracing context in our API, but to wrap the context representations of underlying implementations (e.g. OpenTracing's span).

## 2.4 Open API Extension

As we have shown, all of Melange's tracing APIs feature similar semantics for tracing code blocks, as in all cases the developer must be able to wrap the whole computation in an anonymous function. Often times this is not possible, and the clear start and end points of the computation occur in different methods. Examples of this behavior are visible when objects are queued during processing, or in callback-based asynchronous programming.

As stated previously, our work includes an extension to each tracing API that introduces explicit start and finish semantics. It works by internally associating the span to an object required for the computation, so that it can be retrieved later when it is time to finish the span. This is a very small incremental change that simply adds an Object parameter to each method representing the object to which we will associate the span.

Associating the span to a specific object reference allows Melange to retrieve this association in the future, even if the execution spans multiple threads. This feature can also be used to follow the full ramification of individual events, even if they are processed long after the call.

In our experience, this *enqueue → dequeue → store context → finish span → continue with previous context* instrumentation pattern proved useful in connecting the previously almost independent event arrival, preprocessing, storing and event processing loops.

## 2.5 Black Box instrumentation

In systems that are reliant on black-box components in their critical path, it is not unreasonable to assume that the external components could be the source of a slow or incorrect request. In many cases it is not possible to debug or inspect at the source-code of the black-box, and teams often have to resort to monitoring tools to get some insight on what is affecting their performance. If each black-box is, in itself, a distributed system, this might quickly become an exercise in futility, as it is unclear which node or replica was part of the execution of any given request.

In an attempt to ease this process we instrument the black box's client or driver, through the use of Aspect Oriented programming, in order to obtain process-level meta-data that can be used filter the monitoring results. We have found that the most useful piece of data is usually the IP address of the node responsible for executing the traced request, or any other uniquely identifying piece of information. Using AOP allows us to limit the overhead of upgrading black-box versions. Developers are weary of relying on forked versions of their open-source dependencies as the smallest upgrade in the original product could break the modified code, which results in one of two possible outcomes: (1) slow adoption of new features due to the added effort of fixing conflicts, or (2) upgrade avoidance and loss of new features, or worse, important security patches. However, in most projects, the probability of an upgrade breaking low-impact instrumentation code is small, therefore the upgrade burden is mostly related to the effort of merging the two code bases for even the smallest upgrade. With AOP the developers need not be concerned with the instrumentation code, as it is already self-contained in the form of an Aspect, making the upgrade process as simple as updating the version in the dependency manager's configuration and recompiling.

For the purposes of this project we have instrumented two black-box systems: Cassandra and RabbitMQ.

# 3 IMPLEMENTATION

In this section we will describe each implementation of the instrumentation framework's APIs

The module-based hierarchy of the tracing library is described in Figure 1. Since the BaseTracing API is simply an adaptation of the OpenTracing functionality to match our API semantics, in the following sections we will instead focus on the modules that contain Melange-specific functionality.

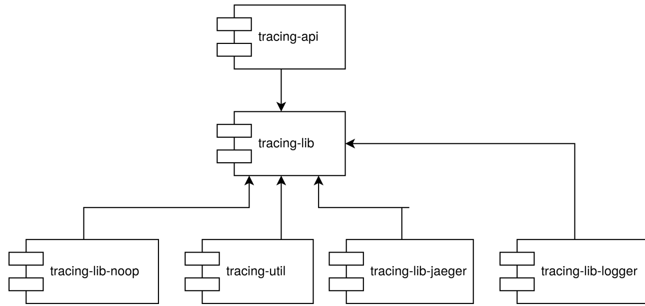


**Figure 1: Module-based decomposition of our tracing library.**

## 3.1 TracingWithId API Implementation

As stated in the API description, through the TracingWithId API, each new tracepoint will become a child of the tracing context stored, for the same eventID, in the previous thread.

To achieve these semantics, four questions needed to be answered:

**(i)** How to know which trace represents the current eventID?
**(ii)** How to know if the thread has changed?
**(iii)** How to obtain the previous thread's context?

To answer the **first question**, Melange makes use of two maps `traceIdMappings` and `spanIdMapping`. The first maps the application-specific eventID to its corresponding traceID. The latter maps a traceID to a stack of spans, where each entry represents the active span for each context switch the request went through. When a new trace is started (by calling `newTrace`), the span's traceID is parsed and the eventID is mapped to it. Afterwards, an empty stack is created and associated to the same traceID and the newly created root span is pushed on top of it. This span will never leave the stack so that orphan spans associated to the same eventID can bind to it.

To **detect if there was a context switch**, Melange relies on an OpenTracing feature called Baggage. Baggage, much like an annotation, is a key-value pair stored within the span, however, unlike annotations, baggage items can be retrieved through the instrumentation API. When a new span is created, the current thread's ID is attached to it as a baggage item. Every time a new span is created through this API, the threadID of the span at the top of this trace's stack is compared to the caller thread's ID. If it matches, the span is created as usual, otherwise the new span is pushed onto the stack. Later, when the span is finished, it is popped from the stack.

The answer to **question (iii)** follows naturally from the structures created to answer questions (i) and (ii), as one can assume that when calling `addToTrace`, the new span's parent is almost always the span that is at the top of the current trace's stack (barring the edge-cases described in the previous section).

The reader might be wondering how these mappings affect the memory usage and Garbage Collection (GC) of a long-running program. This is extremely important, as large GC pauses can severely impact the performance of latency-sensitive applications such as the ones developed by Feedzai. To counter the increased memory usage, we made use of Google's Guava Caches for our mappings, allowing us to set a maximum number of concurrent mappings and a time-based eviction policy, both easily configurable. Through this configuration users can establish an upper bound for the amount of tracing data stored in memory, and properly manage its lifetime. The object lifetime management is essential for performance, as modern GC algorithms can collect short-lived objects efficiently but collecting older generations still has a noticeable impact on performance.

## 3.2 TracingWithContext API Implementation

Similarly to the TracingAPI implementation, OpenTracing already supports propagating the tracing context as an object, was well as its serialization and deserialization. Therefore, this implementation is mainly a usability improvement.

### *3.2.1 Interoperability with Instrumentation Frameworks.*

As stated in the previous chapter, Melange provides a `TraceContext` interface that abstracts the underlying tracing framework's span implementation. This allows developers to implement the API on top of several instrumentation frameworks, allowing them to compare and chose whichever framework better fits their problem set. This interface should be parametrized with the span implementation used by the framework used as back-end. For OpenTracing Melange provides a custom SpanTracingContext class out of the box.

### *3.2.2 Serializing and deserializing context.*

Unlike other instrumentation frameworks Melange does not distinguish between propagation format, and the context is serialized as plain-text. This change frees the programmer from having to implement multiple interfaces for each propagation method. After experimenting with OpenTracing, we felt that the advantage of having different primitives for `TextMap` and `HTTPHeaders` was minimal.

## 3.3 Tracing-Util

The `tracing-util` module, like the name says, contains utilities designed to help introduce tracing into complex systems.

### *3.3.1 Trace-Util Singleton.*

As we have previously stated, in most systems it is not easy to propagate context objects through multiple levels of indirection. Our library attempts to ease that process by providing alternative means to perform the same tasks. However, to execute any action, the developers still need access to the tracer object wherever a tracepoint is needed. To simplify this access, we introduced a singleton object that accepts an implementation of our API, allowing it to be accessed anywhere.

### *3.3.2 Configuration Reloading Tracer.*

More often than not, developers wish to customize the the level or granularity of the tracing infrastructure, much like they would with a logging engine. Existing tools like OpenTracing allow developers to specify a sampling rate that can be adjusted on the fly, thus managing the relationship between observability and overhead. We decided to expand upon that, and provide the ability to, not only adjust the sampling rate, but to update the tracing-configuration at runtime, allowing developers to switch their tracer as they wish.

# 4 EVALUATION

In the previous chapter we introduced Melange and explained how it integrates with commonly used distributed tracing engines. We will now present and discuss the experimental results obtained in a plethora of tests designed to evaluate its performance and usability.

# 5 QUALITATIVE EVALUATION

Melange improves upon the current standards for tracing instrumentation in three main categories: **code simplicity**, **tracing asynchronicity** and **tracing black-boxes**.

## 5.1 Simplifying Instrumentation Code

OpenTracing is the most widely used framework for tracing instrumentation. It provides an API that, while powerful, is quite complex and verbose. To address the first issue, Melange explicitly labels instrumentation routines as **starting**, or **continuing** a trace. To better understand this, consider Listing 4 and Listing 5 where we describe the semantics for starting a new trace in OpenTracing and Melange, respectively.

The key difference in both examples, besides the reduced code footprint, is that in the OpenTracing example users must know

```
try(Scope s = tracer.buildSpan("Entry Point").ignoreActiveSpan()
                                             .startActive(true){
    longRunningThing();
}
```

Listing 4: Starting a trace with OpenTracing

```
tracer.newTrace(() -> longRunningThing(),"Entry Point");
```

Listing 5: Starting a trace with Melange

that newly created active spans automatically become children of the previous active span unless `ignoreActiveSpan()` is called. In our experience, this is far from clear, and caused a lot of confusion during our early OpenTracing experiments.

## 5.2 Tracing Asynchronous Executions

In the previous section we detailed the clarity and simplicity of our instrumentation middleware when compared to OpenTracing in single-threaded programs. The same can be said for inter-thread traces, where we are able to leverage the TracingWithId API. As shown in Listing 6 and Listing 7, not only does our library require less lines of code to trace the same execution, the OpenTracing version forces the developer to add a tracing-related parameter to the traced method, violating our **separation-of-concerns** design goal.

```
public void do(String eventID){
    try(Scope s = tracer.buildSpan("Do").startActive(true) {
        Span context = tracer.activeSpan();
        doAsync(eventID, context);
    }
}

@Async
public void doAsync(String eventID, Span context){
    try(Scope s = tracer.buildSpan("Do Async").asChildOf(context).
        ↪ startActive(true) {
        longRunningThing();
    }
}
```

Listing 6: Continuing an asynchronous trace with OpenTracing

```
public void do(String eventID){
    tracer.newTrace(() -> doAsync(eventID), "Do", eventID);
}

@Async
public void doAsync(String eventID){
    tracer.addToTrace(() -> longRunningThing(), "Do Async",
        ↪ eventID);
}
```

Listing 7: Continuing an asynchronous trace with the Melange

Additionally, Melange is able to follow requests across load balancing strategies like queuing. This is made possible by the Open extension discussed in the previous chapter. In Listing 8 we show an example of this behavior.

```
public void enqueueEvent(Event event){
    tracer.addToTrace(() -> queue.enqueue(event), event, "Event
        ↪ waiting in queue");
}

public void processEvent(){
    TraceContext ctx = tracer.currentContext();
    Event event = queue.dequeue();
    tracer.closeOpen(event);
    tracer.addToTrace(() -> doProcessing(event), "Do Processing",
        ↪ ctx);
}
```

Listing 8: Open extension for Tracing API

To capture this behavior through OpenTracing one would have to modify the application's source code to propagate an object representing the tracing context, much like the example in Listing 6. Modifying methods or classes with tracing parameters tightly couples the application and instrumentation code, making future changes more complicated. Furthermore, exposing OpenTracing related constructs like span or Scope prevents users from changing instrumentation frameworks easily, even if all frameworks follow the span-based model. As stated in the previous section, Melange is mostly self-contained, allowing developers to implement it on top of whatever instrumentation framework or engine they are familiar with.

## 5.3 Leveraging Black-Box Data

As discussed the previous chapter, our work includes aspects to instrument the Java drivers of RabbitMQ and Cassandra which augment the spans resulting from Melange. To do this, one must simply use the AspectJ compiler for their project, loading our aspects, or set up a project that instruments the drivers separately and subsequently load them into the application.

To better understand how this can be useful in practice regard the following example. Consider a system that stores data in a Cassandra cluster that is currently under heavy load making one of the nodes experience degraded performance. As requests are being sent to this system, most requests complete successfully and within the expected latency range, but a few are either failing, or slow. Obviously we would be able to monitor the resource usage of each node, but if all nodes are working equally it might be hard to find the culprit. The only way to detect the root cause of this problem, without simply increasing the capacity of the machines, would be to infer the node that is resulting in failures by matching logs or manually calculating the hash key of the failing requests and comparing it with the cluster's current data distribution.

A job like the one described above might take hours in a production environment, resulting in outages or poor performance, both of which can incur financial penalties for the company. Even if the system supported distributed tracing through OpenTracing, the resulting traces would not of be much help, as we would only be able to confirm that the spans representing database access are longer in certain requests. On the other hand, if the system was instrumented using Melange, and made use of the instrumentation aspects for the Cassandra driver, we would immediately know which node was being targeted by the failing requests, as the spans representing this computation would be enriched with the node's IP address.

Our AOP based approach is very resilient to upgrades when compared to a manual modification approach. We have tested all previous versions of the Datastax Cassandra driver for Java and our aspects were able to advise every version released in the past four years, failing for the first time with version 2.1.5 released in March of 2015 (the current version, 4.0.0, was released in May of 2019). The RabbitMQ driver showed even better results, failing only on version 2.3.1 released in February of 2011 after being tested on every version since the most recent 5.7.3 released in July of 2019.

## 5.4 Quantitative evaluation

In this section we will analyze how Melange fares in a sleuth of tests that will focus on performance, and code modification metrics.

The main test-cases for this evaluation are the products involved in a normal Feedzai deployment. However, as this is a very specific use-case, we have also instrumented two reference-applications for microservices development: Spring's Sockshop[5] and Micronaut's Petstore[6]. The benefit of this approach is that Melange has been tested in applications following two different and widespread architectures for distributed systems, reinforcing its versatility. Since the results were very similar for the two micro-services projects, in this paper we will focus on Petstore.

[5]https://microservices-demo.github.io/
[6]https://github.com/micronaut-projects/micronaut-examples/tree/master/petstore

## 5.5 Feedzai's Use Case

Feedzai uses machine-learning to detect fraudulent credit-card transactions. To do this its software must analyze each of its client's transactions. Naturally, this means clients will not confirm a transaction unless it has first been been approved by Feedzai's product which, in plain terms, means Feedzai's software runs on the critical path of payment processing systems that require it to process billions of dollars per day, with extremely strict SLAs.

Although distributed tracing provides numerous benefits, as described in Section 1, a system with the above requirements cannot afford its monitoring tools to add significant overhead. In the following sections we will show that the performance of applications instrumented with our middleware is comparable to the application's performance when traced with alternative instrumentation frameworks, and that, in the future, it could be deployed in production environments due to its ability to be enabled dynamically and the potential to further reduce its performance impact via sampling.

Furthermore, a regular deployment of Feedzai's system is comprised of several products, each containing hundreds of thousands of lines of code. This creates a need for a tracing instrumentation framework that does not require structural code modification resulting in code breaking changes. Below we will show that our tracing framework requires less lines of code and structural modifications per-tracepoint than current alternatives.

## 5.6 Code Evaluation

As source-code instrumentation is the main focus of this thesis we have also performed a thorough analysis of Melange's impact on the code-base it will instrument.

Evaluating how much a program was modified is not as clear cut as evaluating its performance. Therefore, we settled on metrics that are measurable and easily verified. After much consideration we decided to monitor the number of lines-of-code, method signatures and APIs modified, as well as the number of lines-of-code and classes added. Although we experimented with tools to automatically compute the differences in instrumentation between Melange and OpenTracing, the final results were manually calculated, as none of these tools provided the granularity we required. Since the metrics enumerated above can vary significantly with code-style changes, we enforced two rules: (i) K&R style brackets (i.e., opening bracket is on the same line as the method signature) as this is the style used by most Java books, and (ii) one statement per line, with no line wrapping.

### 5.6.1 *Code evaluation results and analysis.*

**Petstore** When analyzing the results in Table 1, it becomes clear that Melange requires more line modifications than OpenTracing to achieve the same purpose. This was expected, as our library traces code that is passed to it as a lambda, surrounding and modifying the original method call. However, this design decision results in much fewer lines added, when compared to the code instrumented with OpenTracing.

| Library | Lines Modified | Lines Added |
|---|---|---|
| Melange | 94 | 76 (68*) |
| OpenTracing | 83 | 171 (86*) |

**Table 1: Impact of Tracing at the line-of-code level on the Petstore project.** The * symbol stands for lines added for import statements.

The difference can be easily understood by reviewing the code constructs for tracing of single method calls. While OpenTracing requires that the code is enclosed in a try-with-resources block, which uses two additional lines (the try initialization, and the closing bracket) and an additional language construct, our library wraps the traced line of code. Even if we ignore the closing bracket in our calculation, the OpenTracing implementation requires that developers add another level of indentation to their code, specifically for tracing. As we have stated throughout this document, we believe that this leads to less readable code, as the level of indentations increases with the granularity of the trace.

Surprisingly, we see that the number of lines modified is not too dissimilar, with Melange modifying ninety-four lines when compared to OpenTracing's eighty-three. Strange though it may seem, it is easily explained by viewing the traced code. Since Micronaut's (the framework used) pattern for including custom HTTP headers in the request is to add another parameter to the declarative client's method signatures, each traced call will require a modification, regardless of whether we use OpenTracing or our library. Melange shines in this situation, as the cost of our instrumentation is amortized by the fact that the line would have to be modified anyway. In summary, for every line changed by Melange to trace a REST call, OpenTracing requires one line modification, two additional lines added and one extra level of indentation.

| Library | Sig. Changes | APIs Broken | Classes Added |
|---|---|---|---|
| Melange | 0 | 17 | 0 |
| OpenTracing | 0 | 17 | 1 |

**Table 2: Impact of Tracing at the structural level on the Petstore project**

Table 2 shows that some modification of the project's APIs and method signatures were necessary to propagate context between services. It must be said that this was only necessary due to the way Micronaut handles the addition of request headers. As we have said before, in Micronaut, when using the declarative REST Clients, headers are added to a request by passing an additional parameter to the client API. Finally, a new class was also added by the OpenTracing instrumentation to encapsulate the code for serializing and deserializing the tracing context.

**Feedzai's Streaming Engine** Due to the scale of Feedzai's codebases it is unfeasible to instrument every product with OpenTracing just for a simple comparison, hence we focused our efforts on instrumented the streaming engine, which is the most complex and prominent component of Feedzai's stack. Similarly to the Petstore and Sock-Shop, Table 3 shows an increase in modified lines of code when comparing Melange to Opentracing, but a vast decrease in the number of lines added. The magnitude of this decrease is caused by Feedzai's very asynchronous code, with a heavy emphasis for chained CompletableFuture objects, requiring a lot more OpenTracing code, as our library was designed for this type of environment.

| Library | Lines Modified | Lines Added |
|---|---|---|
| Melange | 20 | 31 (16*) |
| OpenTracing | 16 | 114 (32*) |

**Table 3: Impact of Tracing at the line-of-code level on Feedzai's Streaming Engine.** The * symbol stands for lines added for import statements.

In Feedzai's streaming engine, a request is represented by a Message class that is propagated throughout the execution. This proved valuable in reducing the amount of structural changes required by the OpenTracing instrumentation, as we could piggyback the tracing context on this Message, but certain situations still required method signature changes, especially for methods that are simply side-effects of the critical path and therefore do not require access to the Message object. However, this Message is an API which means that modifying it requires modifying all implementations, which we see as a reasonable trade-off.

| Library | Sig. Changes | APIs broken | Classes Added |
|---|---|---|---|
| Melange | 0 | 0 | 0 |
| OpenTracing | 5 | 1 | 1 |

**Table 4: Impact of Tracing at the structural level on Feedzai's Streaming Engine.**

Concluding, it is clear that, as proposed, our middleware requires less structural source-code changes, as well as fewer line additions with a comparable number of modifications. Additionally Melange provides additional features while maintaining the instrumentation advantages described above.

## 5.7 Performance

To analyze the performance of our instrumentation framework we will focus on request latency and throughput as well as CPU, Memory, Disk and Bandwidth usage. These metrics allow us to understand how the instrumentation affects the system's ability to respond to requests adequately, and without unreasonable resource requirements.

### *5.7.1 Performance Evaluation Methodology.*

To create a set of realistic benchmarks, we stress-test the projects by generating load in a way that simulates a real and highly demanding environment, as opposed to a purely synthetic micro-benchmark.

To benchmark Feedzai's products, we simulate a real environment by sending a fixed rate of 300 transaction authorization requests per second. For both micro-services projects we used Locust[7] to generate HTTP requests simulating active users on the website. We configured Locust with 200 clients, spawning at a rate of 5 clients per second. Each client awakes every second to execute 3 requests in sequence, on a round-robin basis.

There were a few differences in the environment between Feedzai's benchmarks and Petstore. This is because Feedzai's system, being comprised of real-world products, is more resource-intensive than simple micro-service reference projects, and thus requires more hardware.

**Feedzai**. To benchmark Feedzai's products we used two machines with the following specifications: Intel Xeon CPU E5-2680 v3 CPU (32 cores @ 2.50GHz), 120GB RAM and a 500GB 7200RPM HDD, running CentOS 7 (7-4.1708.el7). The containers were split across the two machines using Docker Swarm, leaving the orchestration to the tool, with no additional configuration. It is important to mention that this was a system implemented specifically for the purposes of this work, with the goal of representing the structure of a typical Feedzai deployment, but at a much smaller scale and with reduced hardware requirements. The environment was also not tuned or configured by experienced Feedzai engineers, meaning that the results shown in this document should only be used to compare the different tracing frameworks and are not indicative of the performance of Feedzai's software in the real world.

**Petstore** To benchmark PetStore, each service was deployed as a Docker container on a single, powerful machine that has the following specifications: Intel Xeon CPU E5-2680 v3 CPU (16 cores @ 2.50GHz), 64GB RAM and a 500GB 7200RPM HDD, running CentOS 7 (7-4.1708.el7)

To monitor the resource usage per-container, we also ran a dockerized monitoring infrastructure based on Prometheus[8], Cadvisor[9], Grafana[10] and Node Exporter[11].

### *5.7.2 Performance evaluation results and analysis.*

**Petstore** The performance test results indicated that the change in request latency when instrumenting Petstore with either Melange or OpenTracing was very small.

This not to say that there was no increase, in fact, Figure 5 shows that the median request was 0.4% slower with Melange and 15% faster with OpenTracing, while the 99th percentile data reveals the opposite, with Melange adding a 0.24% overhead and OpenTracing showing a 29.6% increase in latency. It is clear that our work cannot consistently outperform OpenTracing (as Melange is built on top of it), and that it is impossible for either framework to outperform tracing-less executions. This indicates that the results displayed were affected by experimental noise. While we are not discarding the clear overhead introduced by either tracing framework starting at the 99.9th percentile, it becomes difficult to differentiate between true overhead and noise. Running the same benchmark multiple times returned different, but equally inconsistent results, with neither framework coming out on top, which is indicative of limited overhead as there is not a consistent gap in performance between

[7]https://locust.io/
[8]https://prometheus.io/
[9]https://github.com/google/cadvisor
[10]Grafana
[11]https://github.com/prometheus/node_exporter

| Percentile | None | Melange | Op'Tracing |
|---|---|---|---|
| 50.000 | 372.1 | 372.5 | 365.5 |
| 99.000 | 663.5 | 665.0 | 693.8 |
| 99.900 | 759.5 | 1191.6 | 1277.7 |
| 99.990 | 1315.6 | 1353.7 | 1379.2 |
| 99.999 | 1398.0 | 1459.4 | 1448.8 |

**Table 5: Latency distribution of Petstore**

| Metric | None | Melange | Op'Tracing |
|---|---|---|---|
| Tput (t/s) | 557.0 | 557.7 | 555.6 |
| Max (ms) | 1538.1 | 1619.3 | 1522.7 |
| Disk (GB) | – | 5.6 | 5.4 |

**Table 6: Additional metrics for Petstore**

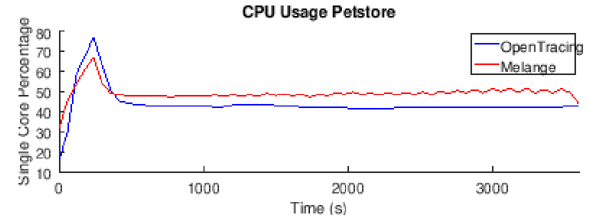


**Figure 2: CPU Usage of Petstore**

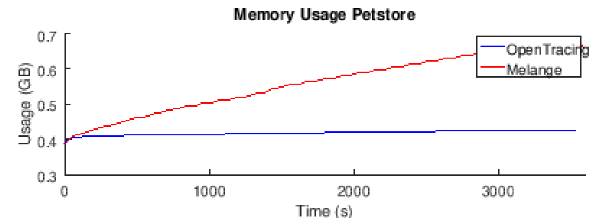


**Figure 3: Memory Usage of Petstore**

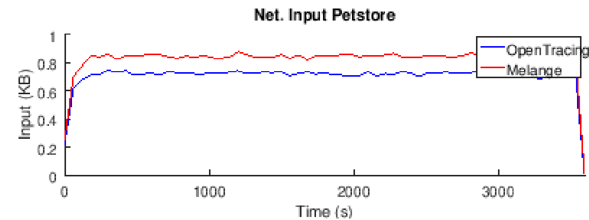


**Figure 4: Net Input of Petstore**

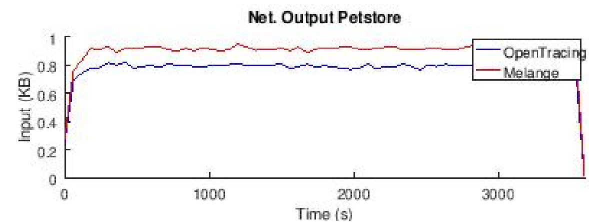


**Figure 5: Net Output of Petstore**

Melange, OpenTracing or neither. It is also worth mentioning that both tracers were configured to sample every request, which adds to the overhead but would rarely happen in production environments.

The monitoring data, unlike the performance metrics, shows a clear overhead when tracing with Melange, compared to OpenTracing. Figure 2 shows an 11% increase in CPU Usage, which we attribute to the additional span processing performed by our framework (detailed in the previous chapter). Melange uses 20% more memory than OpenTracing (shown in Figure 3) as our framework holds data in memory for cross-thread trace reconstruction. While this number may seem high, Melange's in-memory cache configuration was very conservative, and we hypothesize that the memory usage could be decreased with a more aggressive maximum size and expiration policy.

**Feedzai Streaming Engine.** In a latency sensitive application like Feedzai's Streaming Engine (FSE), slight differences in performance between instrumented and not instrumented runs will result in more noticeable overhead.

Unlike the previous projects, Table 7 shows that there is a clear overhead when tracing with Melange, as the median increases by 22% when sampling every request. While high, this overhead is consistent with the data collected in Google's Dapper evaluation [18]. Reducing the sampling rate to 50% decreases the latency, but unlike Dapper it does not decrease linearly. However, while there is a large percentual increase, the absolute overhead is simply 1 millisecond, which makes little difference in practice, as it would not be noticeable by the users.

| Percentile | None | Melange | Mel. (50%) | OT Noop |
|---|---|---|---|---|
| 50.000 | 6.8 | 8.3 | 8.1 | 7.6 |
| 99.000 | 9.8 | 18.1 | 16.7 | 12.1 |
| 99.900 | 17,5 | 62.8 | 58.3 | 23.6 |
| 99.990 | 61.5 | 87.2 | 85.1 | 58.2 |
| 99.999 | 116.7 | 130.2 | 135.8 | 89.5 |

**Table 7: Latency distribution of FSE**

| Metric | None | Melange | Mel. (50%) |
|---|---|---|---|
| Tput (t/s) | 297.3 | 295.2 | 295.8 |
| Max (ms) | 116.7 | 144.4 | 176.7 |
| Disk (GB) | – | 11 | 6.3 |

**Table 8: Additional metrics for FSE**

The largest overhead is found in the 99.9th percentile, as Melange shows latencies 3.8 times higher when sampling all requests and 3.4 times higher with a 50% sample rate. On the other hand, the overhead in the 99.999th percentile, as well as the increase in maximum request latency are quite comparable to the median, being only 12% and 24% higher, respectively. From this data we can extract two conclusions. First, while the overhead is large in the 99.9th percentile, the fact that the median and 99.999th percentiles show comparable increases indicates that the impact of our framework is constant, making the impact less noticeable in slower workloads.

Secondly, it shows that Melange's performance is not affected by load spikes, making it suitable for production workloads where establishing a hard maximum request latency is more important than low latencies across the whole percentile spectrum, or in performance engineering efforts.

Based on the above results we conjectured that the performance impact could be caused by either increased GC activity, or coarse-grained synchronization in Melange's caches. To test the first hypothesis, we first tested the system after setting the sampling rate to 0 to understand what is OpenTracing's contribution to GC activity and afterwards we experimented with more aggressive expiration policies for Melange's internal caches. To properly evaluate how the in-memory caches were affecting performance, we devised two additional tests. The first took advantage of a configuration property of the Guava Cache that allows the user to set the expected number of threads accessing the cache concurrently, which we set to 32 (the number of cores in the test machine). The second test involved replacing the caches by a HashMap with absolutely no thread synchronization (clearing the Hashmap when its size is larger than 50 entries).

Although setting Melange's sampling rate to 0 reduced the number of GC pauses that occurred during the test to a number closer to a tracing-less execution, none of the tests had any meaningful effect on the latencies in the most affected percentiles. Once the impact of GC and thread synchronization was disregarded, it became obvious that the overhead was being caused by the additional instrumentation code introduced by Melange and OpenTracing. While this is a straightforward conclusion, our experience with the two micro-services projects, as well as our analysis of the source-code for Jaeger's OpenTracing implementation pointed towards different sources, as the overhead seemed too large to be caused by the additional instrumentation. To confirm this we executed yet another test, in which we replace Jaeger's OpenTracing implementation by a no-op implementation of the API. After this test we were finally able to clarify whether the bulk of the impact was being caused by Melange or OpenTracing.

Replacing the OpenTracing code with a No-Op implementation showed significant latency changes in the 99.9th percentile, reducing the overhead from 270% to 30%. Obviously this is not useful in practice as it means that the system is not collecting any traces, but it shows that Melange has minimal effect on OpenTracing's performance, confirming what was theorized while discussing the Petstore and Sock-Shop tests. After further investigation we discovered that the sampling rate in Jaeger's OpenTracing implementation has little impact on the system's performance as sampling a request simply means that it is not propagated to the agent but the instrumentation code is executed regardless.

Finally, when analyzing the monitoring data, we see that the Streaming Engine shows 20% increase in CPU Usage when using Melange, even when sampling only 50% of the requests, as shown in Figure 6. This increase was expected as it is the combined impact of Melange's trace-reconstruction functionality, and OpenTracing's span generation and forwarding. Like in the two micro-service projects, the memory usage (depicted in Figure 7) is also larger, due to Melange's in-memory caches used for cross-thread tracing. Interestingly, when plotting the monitoring data for a single container, the Network Output graph in Figure 8 clearly shows the periodic propagation of batched spans, sent from the instrumentation code to the Jaeger Agent. As expected the Network Input remains the same due to each test having the same workload, as shown in Figure 8.

## 5.8 Final Remarks

After analyzing the evaluation data it becomes clear that Melange outperforms OpenTracing in both functionality and code modification metrics. The execution performance of both frameworks is also comparable, with neither showing a clear advantage over the other. However, without further performance improvement, Jaeger's OpenTracing implementation (and consequently Melange) is currently not suited for latency-sensitive systems in production environments that require extremely low latencies across the board. This result could, however, be improved by adding a more thorough sampler to Melange instead of relying on the one provided by OpenTracing.

Both frameworks can, on the other hand, be used to trace systems where consistent and spike-tolerant performance is more important than extremely low latencies in the lower percentiles. Additionaly, each framework can also be used for performance engineering, and in this regard Melange is superior as its small source-code footprint makes it advantageous for non production-critical uses.

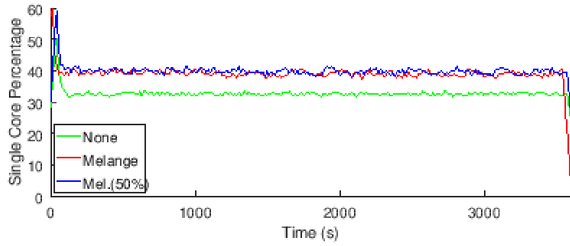


Figure 6: CPU Usage of FSE

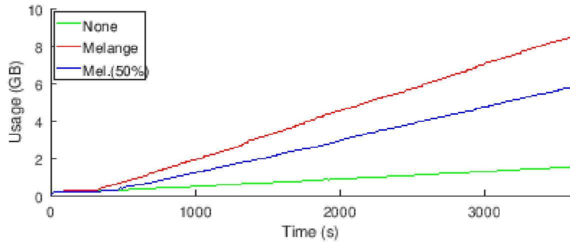


Figure 7: Memory Usage of FSE

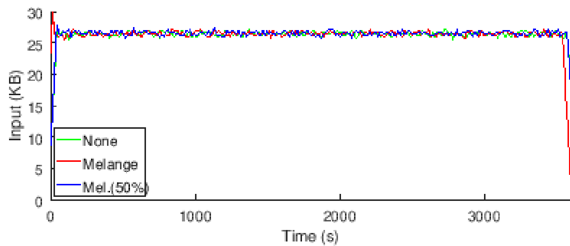


Figure 8: Net Input of FSE

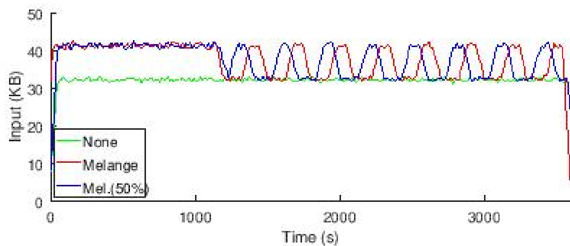


Figure 9: Net Output of FSE

## 6 MELANGE IN PRACTICE

As we benchmarked Feedzai's products the first objective was to establish a tracing-less baseline that could validate future performance results. This seemed simple at first, but the initial results quickly showed otherwise, as we were struck with latencies in the order of several seconds for the 99.9th percentile and an extremely low throughput. Naturally, these results were not expected, and after multiple tests with similar outcomes we enabled tracing through Melange's `LazyConfigurationTracer`. Analyzing the traces uncovered several issues that were impacting performance and had to be fixed in order to obtain the results presented in the previous section. Below we will present some examples.

### 6.1 Garbage Collection in Alert Monitor

When a request is processed by the Streaming Engine it is sent to a RabbitMQ queue that eventually leads to the Alert Monitor (AM) where analysts can manually approve transactions. At the time, the queue was quickly becoming full as the Alert Monitor was not able to deplete the queue at a fast enough rate, which in turn increased the backpressure when FSE was inserting.

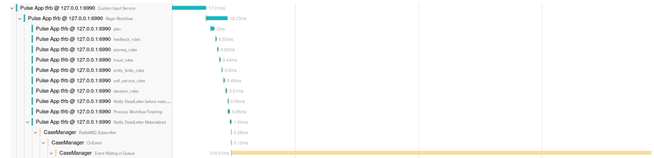

Figure 10: Requests queued in AM

The traces revealed that the requests that had been pulled from the queue spent most of their time in an unbounded in-memory queue in AM, as illustrated in Figure 10. After reviewing the configuration we realized that the process was being initiated with only 2GB of memory, which is extremely low. The constant GC pauses were preventing AM from draining the in-memory queue, which consequently increased the amount of memory even further.

Increasing the amount of memory available solved the problem and raised the overall throughput of the system.

### 6.2 Unnecessary Database Fetches

After studying the monitoring data, we were surprised by the amount of CPU being used by one of the databases – even more than the Streaming Engine. Auditing the streaming pipeline showed nothing that could cause an increase of such magnitude. After investigating multiple traces we noticed that every execution of the input service (the module tasked with input validation and requesting tokenization) was lasting as much time as the entire streaming pipeline (depicted in Figure 11).

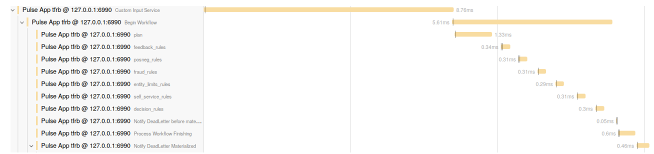

Figure 11: Long spans in FSE

After reviewing the configurations, we discovered that the benchmarking environment was configured to execute multiple data-fetches that were not needed, as the type of request used for the tests did not require the fetched data.

By disabling the unnecessary database accesses we were able to reduce the request latency even further and vastly diminish the CPU usage of the database.

## 7 CONCLUSION

This document introduced Melange, a new distributed tracing instrumentation framework that builds upon the current standard – OpenTracing. Melange was designed to maintain the same functionality as OpenTracing, under a much more concise API, and introduce new instrumentation techniques that allow it to efficiently trace highly asynchronous and heterogeneous systems. In parallel, we described the use of Aspect-Oriented Programming to instrument black-box drivers improving Melange's black-box tracing capabilities, by combining client-side traces with monitoring data. Melange was tested and evaluated in three different systems: two reference micro-service applications, and one real-world highly-performant and latency-sensitive system at Feedzai. It showed great results in source-code modification metrics when compared to

OpenTracing, as well as similar performance. The tests proved that Melange is perfectly suited for instrumenting production-level micro-services based systems. Our work also showed promising yet lacking results in Feedzai's tests, requiring further work before being deployed in most latency-sensitive production environments. At the same time, performance engineering and root-cause analysis becomes much simpler in tracing-enabled systems and while Melange is at too early a stage to make this impact visible in Feedzai's production environments, reviewing the execution logs of previous benchmarking and performance engineering efforts for Feedzai's clients, uncovered several instances of exploratory tests (analyzing the performance impact of enabling/disabling certain parts of the streaming pipeline, etc.) that would not be necessary if the application were tracing-enabled.

# 8 FUTURE WORK

While our work is a good first step into the combination of monitoring and tracing data in heterogeneous distributed systems, there is still a clear distinction between the monitoring and tracing workflows. The end-goal of this effort is to provide a unified tracing and monitoring view that can show, in the same window, the resources used to execute the portion of code represented by any given span. On a more immediate note, Melange requires additional performance engineering efforts to reduce its overhead to a level that could be confortably accepted by Feedzai, and other projects with similar latency requirements. Possible solutions to offset this overhead could be to implement a request sampler that does not rely on Jaeger's OpenTracing implementation, or to experiment with other tracing backends such as Zipkin that could have a more performant implementation of the OpenTracing standard. The AOP based approach imposes some requirements on the products that could be instrumented, mainly that the information that developers want to collect needs to be accessible in the caller thread as the meta-data could not be associated to a specific span if there are context switches. During the development of Melange, we discussed the possibility of solving this *uncertainty* problem by introducing a supervised machine-learning algorithm, that could analyze pre-existing correct traces and suggest the trace that corresponds to an "orphan" piece of meta-data extracted from a black-box.

# 9 RELATED WORK

Most of the works in the distributed tracing literature can be described by the design choices made to implement each of the following four layers:

**Data Extraction Layer**

Concerned with the techniques used to extract tracing data from running systems. Some works use **Static Instrumentation** [9, 13, 18] by modifying their code to extract or propagate tracing meta-data. Others are able to specify new tracepoints during runtime through **Dynamic Instrumentation** [8, 14]. Other works choose **Passive Data Collection** techniques like message capturing [1, 2] or log processing [5] so as to not modify the traced systems.

**Model Layer**

This layer describes models used to aggregate the output of multiple discrete tracepoints into higher level constructs that structure and simplify the analysis of full traces. Tracing systems mostly describe their traces as **span trees** [18] (simpler but less expressive) or **event-based directed acyclic graphs** [9, 13] (expressive but harder to implement and analyze).

**Trace Collection Layer**

Encompasses techniques for data collection and propagation, and strategies to minimize the performance impact of each. Existing works propagate tracing meta-data either **in-band** [12, 14] (less disk usage, but greater toll on the network) or **out-of-band** [3, 4, 9, 13, 18] (little network impact, but I/O intensive). To improve performance in-band propagation systems mostly use **immediate aggregation** [12, 14] to reduce the size of the messages, while out-of-band systems choose **sampling** [9, 11, 13, 18] to limit the I/O impact.

**Analysis Layer**

After traces are collected it is important to have strong analysis tools to extract knowledge from the information. Some works implement **manual techniques** like **query engines** [6, 14] or **visualization tools** [7, 17] to help the user understand the traces, while others leverage the user's existing knowledge for **automated tools** to perform **fault detection**[3, 4] and **root cause analysis**[16, 19, 21].

# REFERENCES


[1] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, 2003), SOSP '03, ACM, pp. 74–89.

[2] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 18–18.

[3] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2002), IEEE Comput. Soc, pp. 595–604.

[4] Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. Path-based Faliure and Evolution Management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (San Francisco, California, 2004), NSDI'04, USENIX Association, pp. 23–23. bibtex[numpages=1;acmid=1251198].

[5] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 217–231.

[6] Chun, B.-G., Chen, K., Lee, G., Katz, R. H., and Shenker, S. D3: Declarative Distributed Debugging. Tech. rep., University of California at Berkeley.

[7] Consens, M. P., and Mendelzon, A. O. Hy+: A Hygraph-based Query and Visualization System. In *SIGMOD Conference* (1993).

[8] Erlingsson, Ã., Peinado, M., Peter, S., Budiu, M., and Mainar-Ruiz, G. Fay: Extensible Distributed Tracing from Kernels to Clusters. *ACM Transactions on Computer Systems 30*, 4 (Nov. 2012), 1–35.

[9] Fonseca, R., Porter, G., Katz, R. H., and Shenker, S. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)* (Cambridge, MA, 2007), USENIX Association.

[10] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[11] Liu, H., Zhang, J., Shan, H., Li, M., Chen, Y., He, X., and Li, X. JCallGraph: Tracing Microservices in Very Large Scale Container Cloud Platforms. In *Cloud Computing âĂŞ CLOUD 2019*, D. Da Silva, Q. Wang, and L.-J. Zhang, Eds., vol. 11513. Springer International Publishing, Cham, 2019, pp. 287–302.

[12] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 589–603.

[13] Mace, J., and Kaldor, J. Canopy: An End-to-End Performance Tracing And Analysis System.

[14] Mace, J., Roelke, R., and Fonseca, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association.

[15] Martin, R. C. Design Principles and Design Patterns. 16.

[16] Mirgorodskiy, A. V., and Miller, B. P. Diagnosing Distributed Systems with Self-propelled Instrumentation. In *Middleware 2008*, V. Issarny and R. Schantz, Eds., vol. 5346. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 82–103.

[17] Reynolds, P., Edwin Killian, C., L. Wiener, J., Mogul, J., A. Shah, M., and Vahdat, A. Pip: Detecting the Unexpected in Distributed Systems. Symposium on Networked Systems Design and Implementation.

[18] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.

[19] Whittaker, M., Teodoropol, C., Alvaro, P., and Hellerstein, J. M. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proceedings of the ACM Symposium on Cloud Computing - SoCC '18* (Carlsbad, CA, USA, 2018), ACM Press, pp. 333–346.

[20] Wright, P. What Etsy Learned Building a Distributed Tracing System. In *Surge: The Scalability & Performance Conference* (2014).

[21] Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M., and Ma, W.-Y. Automated known problem diagnosis with event traces. *ACM SIGOPS Operating Systems Review 40*, 4 (Oct. 2006), 375.

[22] Zhonghua Yang, and Marsland, T. Global snapshots for distributed debugging. In *Proceedings ICCI '92: Fourth International Conference on Computing and Information* (Toronto, Ont., Canada, 1992), IEEE Comput. Soc. Press, pp. 436–440.