TÉCNICO LISBOA

Melange: A Hybrid Approach to Tracing Heterogeneous Distributed Systems

Gonçalo Alexandre Torrão Garcia

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Dr. Nuno Diegues

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Nuno Miguel Carvalho dos Santos

November 2019

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus dois orientadores, Professor Luis Veiga e Doutor Nuno Diegues, pela disponibilidade para me aconselhar e pela ajuda a navegar todas as dificuldades e dúvidas que surgiram durante o decorrer deste trabalho. Queria também exprimir a minha enorme gratidão pela Feedzai por ter apoiado o projecto .

Nada do que fiz, e farei, seria possível sem o eterno apoio dos meus pais, que cometeram todos os esforços e sacrifícios necessários para me proporcionar tudo o que sempre precisei. Um enorme obrigado aos dois, e a toda a minha família que nunca deixou de me encorajar .

Aos meus amigos agradeço pelos incontáveis momentos de descontração e pelas conversas catárticas nos momentos mais difíceis. Aos meus colegas do Instituto Superior Técnico e do ISCTE-IUL, agradeço todo o esforço e empenho necessário para me ajudar a chegar a este ponto.

Queria também agradecer à Clara Quintans, pelo infindável apoio emocional, mas acima de tudo, por me mostrar que há sempre espaço para me esforçar mais, e para ir mais longe.

Não poderia ter estado rodeado de pessoas melhores.

Lisboa, December 9, 2019

Gonçalo Garcia

Resumo

Os sistemas de software estão, cada vez mais, a tornar-se distribuidos para suportar os requisitos de performance e disponibilidade dos utilizadores modernos, bem como a crescente quantidade de dados colecionada por uma base global de utilizadores. Apesar dos sistemas distribuidos apresentarem inúmeras vantagens, colocam em simultâneo alguns desafios para os programadores que necessitam de depurar e analizar a sua performance. Alguns projectos recorreram a *distributed tracing* para reduzir essas dificuldades proporcionando uma visualização de alto nível da execução completa de *requests* individuais, que pode ser anotada com informação de *debugging* e métricas de performance. No entanto, estas ferramentas requerem a modificação dos componentes que nem sempre é viável, ou sequer possível. Outros trabalhos procuraram conceptualizar os sistemas como conjuntos de *black-boxes*, e implementar *tracing* através de inferencia estatistica baseada em dados capturados de forma não intrusiva, ou baseando-se em conhecimento especializado dos componentes do sistema. Apesar destes sistemas não modificarem o software *traced*, são frequentemente lentos, susceptiveis a falsos positivos, ou requerem conhecimento de muito baixo-nível dos componentes, algo dificil de obter para utilizadores de software *closed-source*. Neste documento apresentamos um estudo do estado-da-arte do *distributed tracing*, e propomos uma nova *framework* de instrumentação para *tracing* de sistemas distribuidos que combina modificação do *source code* e extracção de dados com base em *Aspect-Oriented Programming*, para implementar *tracing* em sistemas distribuidos heterogéneros constituidos por *black-boxes*. Fazemos isto recorrendo a tecnologias *open-source*.

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 instrumentation framework that combines source-code modification and Aspect-Oriented Programming based data extraction to implement distributed tracing in heterogeneous distributed systems containing black-boxes. We do this by leveraging commonly available open-source tools.

Palavras Chave Keywords

Palavras Chave

Sistemas Distribuidos

Monitorização

Instrumentação

Keywords

Distributed Systems

Distributed Tracing

Performance Engineering

Monitoring

Instrumentation

Index

1 Introduction **1**

1.1 Motivation 1

1.2 Distributed Tracing 2

1.3 Common Shortcomings in Distributed Tracing 3

1.4 Proposed Solution and Contributions 4

1.5 Document Roadmap 5

2 Related Work **6**

2.1 Overview of Distributed Tracing 6

2.1.1 Data Extraction Layer 8

2.1.1.1 Static Instrumentation 9

2.1.1.2 Dynamic Instrumentation 10

2.1.1.3 Passive Data Collection 11

2.1.2 Model Layer 13

2.1.2.1 Span Tree Model 14

2.1.2.2 Event Based DAG Model 15

2.1.3 Trace Collection Layer 17

2.1.3.1 Data Collection 17

2.1.3.2 Performance Optimization 19

2.1.4 Analysis Layer 22

2.1.4.1 Manual Tools 22

2.1.4.2 Automated Tools 23

2.2 Relevant Systems 26

2.2.1 Dapper 26

2.2.2 Magpie 27

2.2.3 Canopy 29

2.2.4 Protractor 30

2.3 Analysis and Discussion 32

3 Architecture **33**

3.1 High-Level Architecture 33

3.1.1 Feedzai - Reference Traced System 33

3.1.2 Tracing Infrastructure 35

3.1.2.1 Trace Granularity 36

3.1.3 Tracing Architecture 37

3.2 Melange's Architecture 39

3.2.1 Architectural Objectives 39

3.2.2 Architectural Overview 40

3.2.3 API 40

3.2.3.1 Tracing API 40

3.2.3.2 TracingWithId API 42

3.2.3.3 TracingWithContext API 45

3.2.3.4 Open API Extension 45

3.2.4 Black Box instrumentation 46

4 Implementation **48**

4.1 API Implementations 48

4.1.1 Tracing-Lib 49

4.1.1.1 Tracing API Implementation 49

4.1.1.2 TracingWithId API Implementation 49

4.1.1.3 TracingContext API Implementation 50

4.1.2 Tracing-Lib-Jaeger 52

4.1.3 Tracing-Lib-Logger 52

4.1.4 Tracing-Lib-Noop 52

4.1.5 Tracing-Util 53

5 Evaluation **54**

5.1 Qualitative Evaluation 54

5.1.1 Simplifying Instrumentation Code 54

5.1.2 Tracing Asynchronous Executions 56

5.1.3 Leveraging Black-Box Data 57

5.2 Quantitative evaluation 58

5.2.1 Feedzai's Use Case 59

5.2.2 Code Evaluation 59

5.2.3 Code evaluation results and analysis 60

5.2.3.1 Petstore 60

5.2.3.2 Sock-Shop 62

5.2.3.3 Feedzai's Streaming Engine 63

5.2.4 Performance 64

5.2.4.1 Performance Metrics 64

5.2.4.2 Performance evaluation methodology 65

5.2.4.3 Performance evaluation environment 65

5.2.4.4 Performance evaluation results and analysis 67

5.2.5 Final Remarks 74

5.3 Melange in Practice 74

5.3.1 Profiling the Benchmarking Environment 74

5.3.1.1 Garbage Collection in the Streaming Engine 75

5.3.1.2 Garbage Collection in the Event Monitor 75

5.3.1.3 Unnecessary Database Fetches 76

5.3.2 Remarks on the practicality of Melange 77

6 Conclusion **78**

6.1 Concluding Remarks 78

6.2 Future Work 79

I Appendices **85**

A API Reference **86**

List of Figures

1.1 Example of a system suited for Distributed Tracing 2

1.2 Example of a trace from the system depicted in Figure 1.1 3

2.1 Layered decomposition of the techniques implemented in previous works 7

2.2 Instrumented example system 8

2.3 Example of a Pivot Tracing Query 11

2.4 Materialization of the model layer in an example system 14

2.5 Low-level view of a span 15

2.6 Example of span tree in Dapper 16

2.7 Evolution of the tracing model in Canopy 16

2.8 Example of out-of-band propagation 18

2.9 Example of in-band propagation 19

2.10 Example of a Hy+ visualization 24

3.1 High-Level overview of a Feedzai deployment 34

3.2 Sequence diagram depicting an end-to-end execution of a deployment 35

3.3 High-level overview of the tracing infrastructure. 36

3.4 High-level view of the components involved in distributed tracing of a Feedzai deployment 38

3.5 Hierarchical view of the tracing APIs 41

4.1 Module-based decomposition of Melange 48

5.1 Cumulative latency distribution function of the Petstore project 68
5.2 CPU Usage of the Petstore project 68
5.3 Mem Usage of the Petstore project 68
5.4 Net Input of the Petstore project 68
5.5 Net Output of the Petstore project 68
5.6 Cumulative latency distribution function of the Sock-Shop project 69
5.7 CPU Usage of the Sock-Shop project 69
5.8 Mem Usage of the Sock-Shop project 69
5.9 Net Input of the Sock-Shop project 69
5.10 Net Output of the Sock-Shop project 69
5.11 Cumulative latency distribution function of the Streaming Engine 70
5.12 CPU Usage of FSE 73
5.13 Mem Usage of FSE 73
5.14 Net Input of FSE 73
5.15 Net Output of FSE 73
5.16 Long spans in FSE 75
5.17 Requests queued in AM 76
5.18 Long running Input Service 76

List of Tables

2.1 Types of Instrumentation 13
2.2 Tracing Models 17
2.3 Propagation Scheme per System 20
2.4 Optimizations per System 22
2.5 Analysis Tools 26
2.6 Techniques per layer for each relevant system. 31

5.1 Impact of tracing at the line-of-code level on the Petstore project 60
5.2 Impact of tracing at the structural level on the Petstore project 61
5.3 Impact of tracing at the line-of-code level on the Petstore project 62
5.4 Impact of tracing at the structural level on the Petstore project 62
5.5 Impact of tracing at the line-of-code level on Feedzai's Streaming Engine 63
5.6 Impact of tracing at the structural level on Feedzai's Streaming Engine 63
5.7 Latency distribution of Petstore 68
5.8 Additional metrics for Petstore 68
5.9 Latency distribution of Sock-Shop 69
5.10 Additional metrics for Sock-Shop 69
5.11 Latency distribution of FSE 70
5.12 Additional metrics for FSE 70
5.13 Latency distribution of FSE for slower requests 71

5.14 Latency distribution of FSE with no-op OpenTracing implementation 72

5.15 Latency distribution of FSE with no-op OpenTracing implementation 72

1 Introduction

1.1 Motivation

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 [8]. 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 [21].

Some works have attempted to solve the distributed debugging problem by recording a distributed checkpoint of the system state as the execution reaches breakpoints [38]. 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.

1.2 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 [17] 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. To better explain how distributed tracing can be useful, we present the following example.

Consider a system (depicted in Figure 1.1) for a web application comprised of a reverse-proxy/load balancer that routes requests to one of several micro-services running the business logic that communicate through asynchronous RabbitMQ queues. Each microservice is free to use its own storage component, with some using Cassandra for fast access and others using PostgreSQL for ACID transactions. Now consider that a request sent to this system took 5 seconds to complete (1000 times slower than usual).

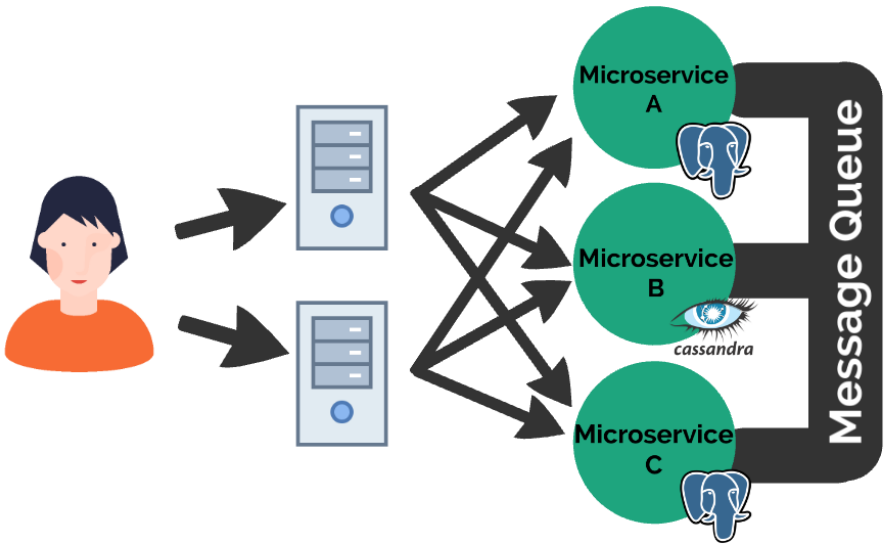


Figure 1.1: **Example of a system suited for Distributed Tracing**

As system administrators or developers, how could we understand what went wrong? First we would have to determine which components were included in the critical path of the request (which by itself is not an easy task [21, 31]), then we would have to go through each of the logs (which might have thousands of lines) to understand which components took longer than usual to compute the request. Only after we discovered the origin of the fault, could we then try to understand what caused the failure and, in some cases, there might not even be a failure. A certain component may respond slower due to high demand, maybe there was a long garbage collection pause, or perhaps the network was slow [31]. The whole process can take hours if not days, depending on how easy it is to access, and understand each component's logs.

If the system was tracing enabled, we could have a single visualization (such as the one shown in Figure 1.2) or file describing exactly which components were part of the request's execution, in what order they were executed and the latency of each computation. Better yet, developers could annotate traces with their own component specific log statements, providing application level observability.

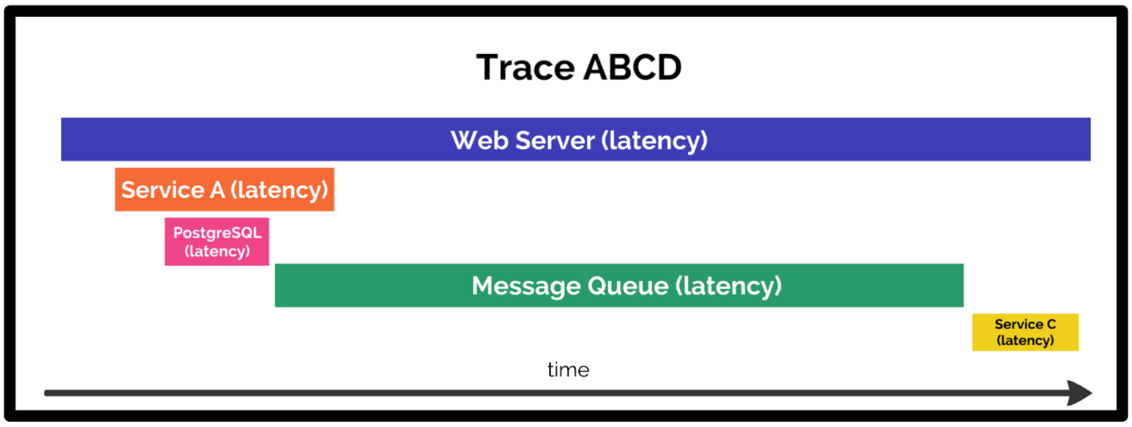


Figure 1.2: **Example of a trace from the system depicted in Figure 1.1**

Major companies like Google (Dapper [31]), Facebook (Canopy [21]), Etsy [36], JD (JCall-Graph [20]) 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.

1.3 Common 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 [21]. 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 is not optimal because faults may be introduced by the black-boxes and therefore hidden from the trace.

1.4 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

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

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.

Aside from our solution, this document contains a detailed overview of the state-of-the-art in distributed tracing, based on a novel layer-based framework developed to classify previous works. Additionally, we construe an exhaustive evaluation of our work (in comparison with OpenTracing[4]) when implemented on top of two reference micro-services projects, and a real industry product, in the form of Feedzai's transaction-fraud detection system.

1.5 Document Roadmap

The document is structured as follows. Chapter 2 is an analysis of the current state-of-the-art in distributed tracing. Chapter 3 describes the architecture of Feedzai's deployments, as well as of our tracing infrastructure and API. Chapter 4 describes how the features specified by Melange's API were implemented. In Chapter 5 we analyze the performance and usability of our middleware, comparing it with other alternatives. Finally, Chapter 6 includes our final remarks on the topic.

[4]https://opentracing.io/

2 Related Work

In this chapter we will discuss important endeavors into distributed tracing. Section 2.1 is an overview of the distributed tracing body of work. In Section 2.2 we describe the most relevant studies that attempt to accomplish objectives similar to ours, and in Section 2.3 we discuss the strengths and shortcomings of each one.

2.1 Overview of Distributed Tracing

To help systematize the discussion of the state-of-the art, we will first present a framework that models the components of a distributed tracing tool as a set of layers which we summarize in Figure 2.1.

Data Extraction Layer Concerned with the techniques used to extract tracing data from running systems. Some works use **Static Instrumentation** by modifying their code to extract or propagate tracing meta-data. Others are able to specify new tracepoints during runtime through **Dynamic Instrumentation**. Other works choose **Passive Data Collection** techniques like message capturing or log processing 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** (simpler but less expressive) or **event-based directed acyclic graphs** (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** (less disk usage, but greater toll on the network) or **out-of-band** (little network impact, but I/O intensive). To improve performance in-band propagation systems mostly use **immediate aggregation** to reduce the size of the messages, while out-of-band systems choose **sampling** 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** or **visualization tools** to help the user understand the traces, while others leverage the user's existing knowledge for **automated tools** to perform **fault detection** and **root cause analysis**.

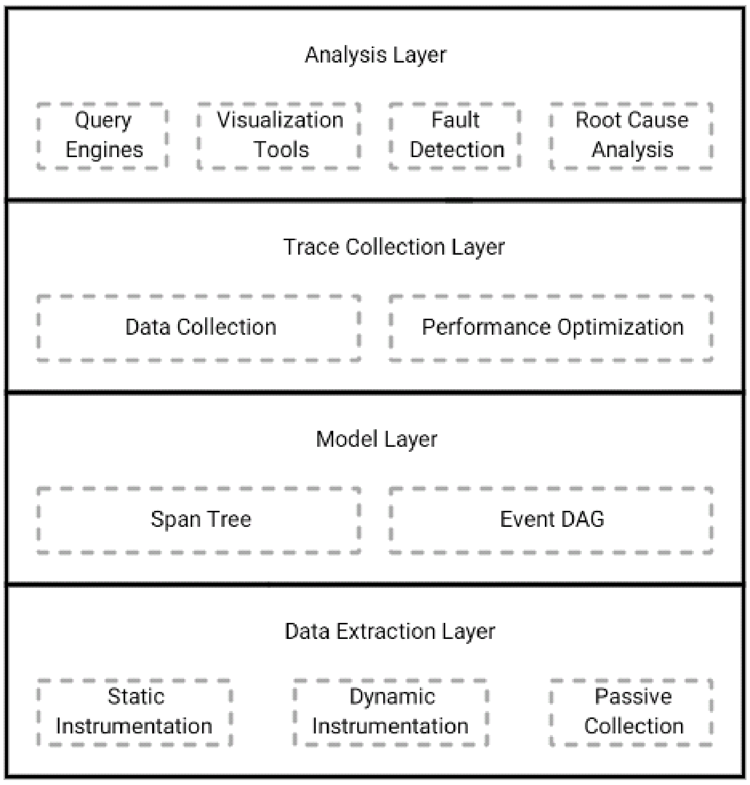


Figure 2.1: **Layered decomposition of the techniques implemented in previous works**

Every system discussed in this chapter implements at least one of the aforementioned layers, while most implement all four.

For each layer we will detail its importance to the overall framework and discuss the most common types of implementation.

2.1.1 Data Extraction Layer

The data extraction layer represents the mechanisms used to extract relevant metrics or data from running software in order to achieve end-to-end tracing. This can be achieved through instrumentation, or passive data capturing.

Instrumentation refers to the changes that must be applied to the traced software in order to propagate meta-data or expose tracing information. Generally tracing packages require the addition of tracing-specific constructs to achieve end-to-end observability. The yellow outline on the example system in Figure 2.2 represents the additional code constructs added to each component. The instrumentation can be static, i.e., developers implement permanent changes to the source code, or dynamic if the modifications can be applied, modified or removed during the execution of the traced software.

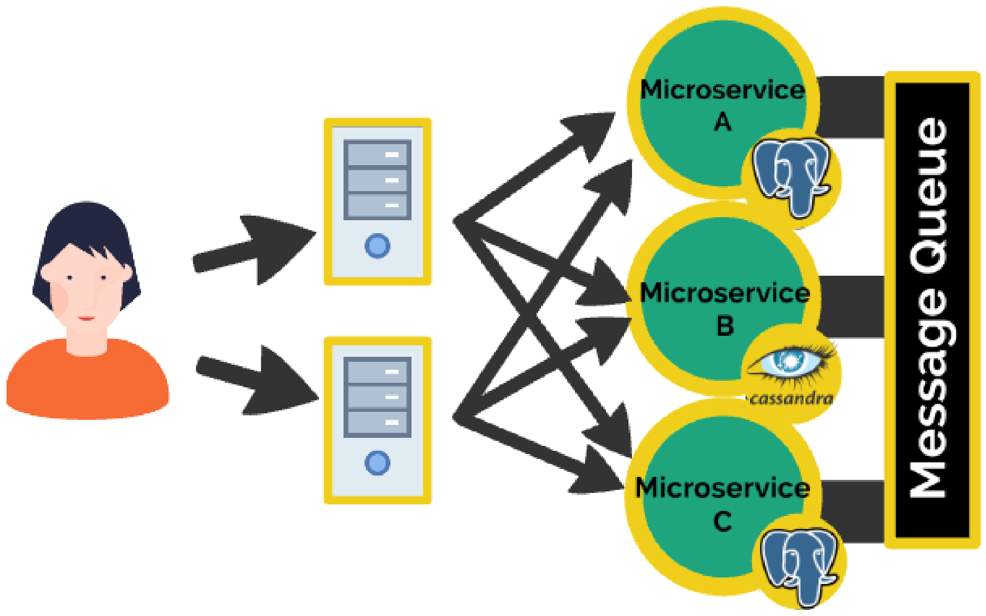


Figure 2.2: **Instrumented example system**

Some works such as Magpie [2], Protractor [6], and the work of Aguilera, et al [1] show that it is possible to extract the necessary information without instrumentation, by **capturing message exchanges**, **parsing logs**, or **analyzing the invocations of underlying kernel functions**.

We will further the discussion on the data extraction layer by describing techniques used to perform static and dynamic instrumentation, as well as passive data capturing.

2.1.1.1 Static Instrumentation

Static instrumentation is used in most production-ready tracing systems currently available [31, 14, 21]. This type of instrumentation requires permanent changes to be applied to the source code in order to propagate and extract tracing meta-data.

To simplify code instrumentation, tracing tools usually provide a high-level API to be used by developers. An example of these APIs is OpenTracing, which is used by Zipkin[1] and Jaeger[2], two systems inspired by Google's Dapper [31]. OpenTracing provides a language agnostic and vendor neutral instrumentation API, that allows developers to seamlessly switch between tracing tools that support it. A similar implementation of the same idea is OpenCensus[3], which provides APIs for extracting both tracing metadata and application metrics.

As the reader might have experienced, having instrumentation code intertwined with application code is undesirable, and teams may be inclined to look for alternatives. The simplest way to achieve this would be to instrument a lower level framework or library instead. That was the approach chosen by Google, who instrumented their gRPC[4] library to support the propagation of tracing information [31]. Zipkin users can also skip the instrumentation process by designing their interprocess communication around Finagle[5]. While this technique is usually applicable, often times the libraries or frameworks used have been previously compiled, and thus are harder to instrument. Modifying machine or byte code requires deeper knowledge that most developers might not have, prevents updates as the system has been modified, and may restrict what can be extracted from the system. In these situations developers may look for indirect ways of injecting instrumentation code. An interesting option is Aspect-Oriented Programming (AOP) [16], a paradigm that allows the separation of cross-cutting concerns by

[1]https://zipkin.io/
[2]https://www.jaegertracing.io/
[3]https://opencensus.io/
[4]https://grpc.io/
[5]https://twitter.github.io/finagle/

modifying code behavior without changing the source code itself. One common use of AOP is to weave logging code into application code at points that match certain criteria (pointcuts). This could very easily be extended for tracing by replacing the logging module with tracing specific functionality.

An important thing to consider when using static instrumentation is that it adds a non-negligible overhead in the form of probe effect, as the instrumentation code is evaluated for every request just like normal code. Furthermore, it forces developers to commit to a final set of instrumentation points when deploying. This is not optimal, as choosing what to instrument and where to apply the instrumentation is a delicate matter and striving for complete decoupling of code and instrumentation might result in the loss of application-level observability. For these reasons, researchers have studied ways of adding and removing instrumentation points to running systems.

2.1.1.2 Dynamic Instrumentation

Dynamic instrumentation tools have two major objectives: (i) having zero probe effect when disabled and (ii) providing developers with some way to instrument running systems, without defaulting to the modify-build-test loop. Systems that use these techniques can alter the code of running processes or OS libraries on the fly, allowing developers to dynamically add tracepoints to expose additional information. These tools are usually implemented through **probing** or **binary instrumentation** [33].

Probing tools have been available for operating systems for many years, in the form of DTrace [5] and SystemTap [26], which compile sets of *probes* or instrumentable points in the system that can be exposed and combined through custom queries. The aforementioned probes are usually defined by applications themselves, shared libraries, or kernel routines [33]. Scholten and Posthuma also proposed a tool based on probes and user queries called *recognizers* for event-based debugging as early as 1993 [30]. Probing tools may not allow the user to add arbitrary instrumentation points, but the simplicity of activating and combining well-defined and documented probes can make them easy to use. The availability of probes in widely used applications like PostgreSQL and MySQL can also reduce the barrier of entry for new users. As of right now the area of probe-based dynamic instrumentation tools seems to focus on single process tracing, however some discussion has been started in the DTrace community on extending DTrace to distributed systems [15].

When software probes are not available, or developers wish to add arbitrary instrumentation to running processes, they must resort to **binary instrumentation tools** (BIT).

BITs usually provide a tool specific API around which developers can design and implement their own tracing routines [33]. This is a departure from probing tools where most of the work might already be done by the software providers. However, the heightened barrier of entry might be worth overcoming for some developers that wish to trace less common execution paths. One of such tools is aptly called BIT [3] and it provides a Java API for instrumenting applications running on the JVM, other tools like Javassist[6] and Soot[7] provide similar features. Software such as Fay [13] and more recently Pivot Tracing [22] extended this notion to distributed systems. Both Fay and Pivot Tracing allow developers to execute queries on live systems much like DTrace or SystemTap. However they are not limited in number of probes available as they can instrument any portion of the program.

Although these tools offer more flexibility when deploying traceable software, they lack some of the expressiveness of their static counterparts. Most only allow developers to query and aggregate simple constructs such as variables or return values, and all of the above, with the exception of Pivot Tracing [22] (as shown in Figure 2.3), are unable to causally relate queries. They also require a relatively homogeneous environment, and Pivot Tracing even requires source-code access.

```
From incr In DataNodeMetrics.incrBytesRead
 Join cl In First(ClientProtocols) On cl -> incr
 GroupBy cl.procName
 Select cl.procName, SUM(incr.delta)
```

Figure 2.3: **Example of a Pivot Tracing Query.** This query sums the value of every iteration of the delta variable of the incrBytesRead method in Hadoop's DataNodeMetrics class and groups it the name of the client process. (Image from Mace, et al. [22])

2.1.1.3 Passive Data Collection

As we mention in Chapter 1, not all distributed systems are compatible with **annotation-based** tracing, since it requires (i) that we have access and ability to modify the source code, or (ii)

[6]http://www.javassist.org/
[7]http://sable.github.io/soot/

that it is compatible with dynamic instrumentation tools. Real-World systems are usually comprised of several external components for which the source code may not be available, or may not be modified as it can invalidate future updates and long term support.

Some research has been put into what is called **black-box** tracing, which attempts to passively capture execution information. One way to do this is by analyzing logs, Mystery Machine [9] requires developers to parse application logs to fit a predefined schema, then the extracted information is used to reconstruct causal paths and performance characteristics with techniques found in race detection literature. Aguilera, et al. [1] on the other hand, passively capture network traffic between system components, and successfully use data-science and signal processing techniques to reconstruct the request path. The work of Chen, et al. [7] uses a similar packet-sniffing based approach to reconstruct system dependencies. One other possibility is suggested by Magpie [2], which uses a user-defined schema to map low-level events such as system-function calls to higher level application specific constructs.

These tools are able to solve the problem of tracing non instrumentable applications, but not without some drawbacks. The reliance on iterative statistical techniques means that the performance does not scale well with large amounts of data [21], meaning a trace might take minutes or even hours to reconstruct. On the other hand, reducing the amount of data ingested by these systems might result in inaccurate reconstructions, with a high rate of false positives, or negatives. Other tools [2] may not rely on these techniques, but require low level knowledge of the application, which could only be possible through reverse engineering.

When tracing less common interactions, these libraries might not be available and the user must provide their own instrumentation. For instance, the developers of X-Trace [14], a distributed tracing system for networks, were forced to instrument the implementation of several networking protocols in order to achieve their goals. Choosing a more opinionated tracing model should make this process easier as it imposes more restrictions on what should be instrumented to fit the higher-level constructs.

Based on the possibilities described above, developers of distributed tracing frameworks must make an educated decision regarding what type of instrumentation to use, in order to reduce the barrier of entry for users, while still providing enough flexibility and usability. Table 2.1 summarizes the type of instrumentation used by each of the systems described in this section. We have excluded DTrace and SystemTap as they are not useful for distributed sys-

System	Static Ins.			Dynamic Ins.			Passive
	Library	Custom	Indirect	Probe	Binary	Indirect	
Dapper [31]	✓						
X-Trace [14]		✓					
Canopy [21]	✓						
Aguilera, et. al [1]							✓
Fay [13]					✓		
Mistery Machine [9]							✓
Pivot Tracing [22]					✓	✓	
Magpie [2]							✓

Table 2.1: **Types of Instrumentation** Shows the type of instrumentation used by each of the described systems. Neither implements Indirect Static Instrumentation nor Probe-based Dynamic Instrumentation, as those approaches are mostly reserved for non-distributed tracing

tems. A quick analysis of the table confirms that systems running in the production environments of major companies [31, 21] use static instrumentation for its low overhead, and due to the homogeneity of the environments. Unlike most systems which use a single technique, Pivot Tracing combines both indirect modification techniques (AspectJ[8]) with binary modification (Javassist) to accomplish its objective.

2.1.2 Model Layer

On top of the instrumentation lies the tracing model. This layer describes how the interactions between software components can be modeled. Simply put, this layer defines what to trace, and how to trace it (illustrated in Figure 2.4). Components of distributed systems are able to interact through several different paradigms. Some use simple message-passing, other base their communication around message queues or pub-sub systems, and many use remote procedure calls.

Architects must take into account which paradigms must be modeled, as a very abstract model is more expressive, but at the same time intensive to compute, and less efficiently stored. However, a more strict and specialized model may be simpler to implement but at the same time restrict the behaviors that can be represented [28]. This dichotomy is quite apparent in the two most common tracing models: **span trees** or **event-based directed acyclic graphs** (which we will refer to as event-based DAG for simplicity).

[8]https://www.eclipse.org/aspectj/

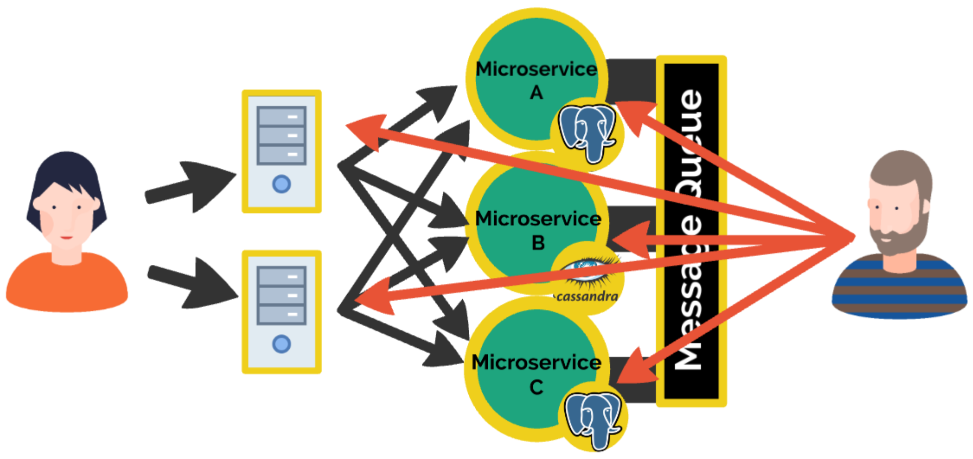


Figure 2.4: **Materialization of the model layer in an example system**

2.1.2.1 Span Tree Model

The span based model was born at Google and had the goal of easily representing RPCs. As the name suggests, this model represents traces as a directed tree of nodes called spans, which are simply a small structure that represents a fragment of a process' execution. As shown in Figure 2.5 each span contains only a trace ID, its own span ID, a parent ID and annotations which are nothing more than timestamped log messages. While most annotations are user defined, some are used by the tracing software in order to record the timestamps of the main actions that represent an RPC:

Client Send	When the client sends the RPC request to another service.
Server Receive	When the server receives the RPC request.
Server Send	When the service issues the response back to the client.
Client Receive	When the client receives the RPC response.

Figure 2.6 shows the span tree in greater detail. Spans without a parent are placed at the root of the tree, and calls to other services are placed on levels below their caller. When calling a remote service, a new span is generated and the previous span becomes its parent. To correlate causality across services, the trace and parent span IDs are propagated along with the request. Spans are displayed across time based on the timestamps of the actions mentioned above.

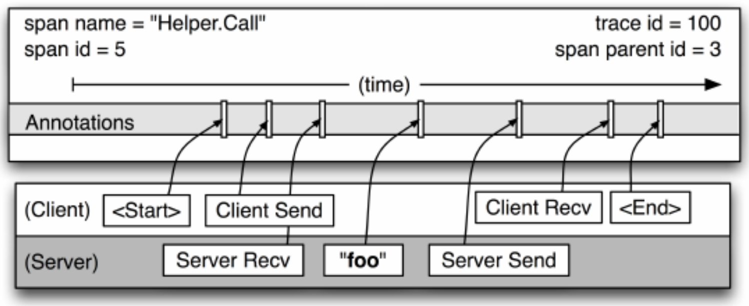


Figure 2.5: **Low-level view of a span.** (Image from Sigelman, et al. [31])

The span model is very easy to understand, and it quickly became popular in the open-source community. However, it is not very flexible, and its simplicity for representing RPCs quickly becomes a hindrance when modeling other paradigms [21]. For example, users of Zipkin and Jaeger have struggled with representing asynchronous executions, message queues, and multiple-parent causality [21, 28]. An example of the latter would be a system that waits for a quorum before responding to requests, where the response would only be as fast as the slowest response in the quorum. In this case the span-based model is not able to show all parental relations, which might hide a slow node in the trace.

2.1.2.2 Event Based DAG Model

The event based DAG model imposes no specific mechanism of interaction. This model was first described by X-Trace [14], and is common throughout the distributed tracing literature [22, 27]. It models events simply as a discrete point in time inside a computation, with edges representing a *happens-before* [17] relation between events. Note that unlike the edges between spans which represent the activation of a different service, the edges in the X-Trace event model represent a higher level causality relation, allowing events to have any number of parents [18, 28]. Other systems define their own event DAG models: Pip [27] groups sets of events as higher-level tasks, messages, and notices, which are analogous to Dapper's spans, edges and annotations. Others model sets of captured events as *flows* [25] or *paths* [8] of execution.

The span model has been proven to be less expressive than the event model [18], and it has been shown to be possible to create a tree of spans using the event based DAG model, by representing the timestamped actions in spans (Client Send, Server Receive, Server Send, Client Receive) as the start and end events of parent and child spans [18, 21]. Facebook developers

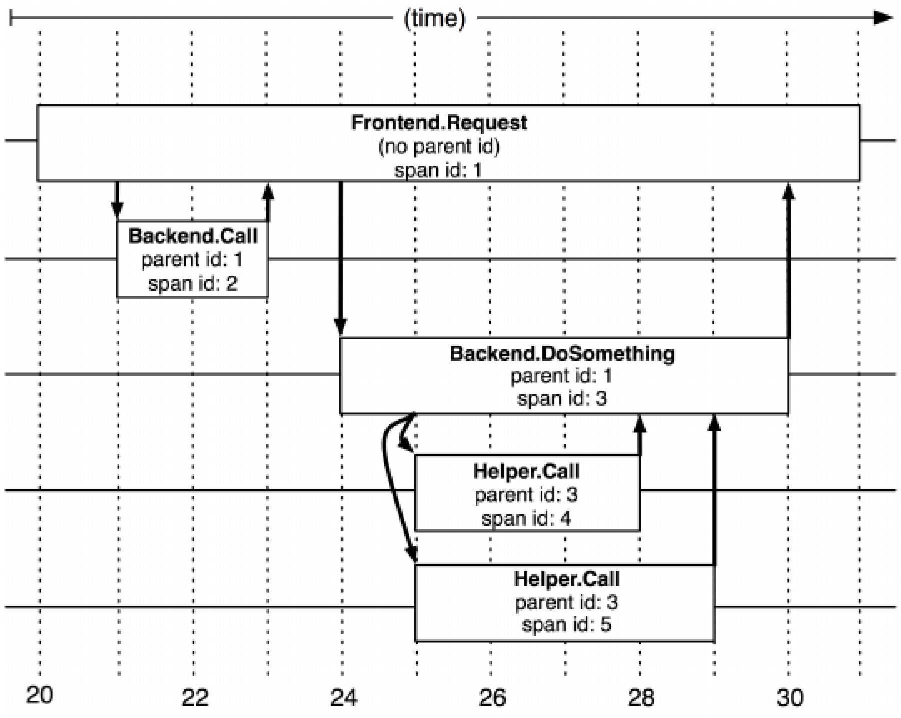


Figure 2.6: **Example of span tree in Dapper.** Rectangles represent spans and edges represent causal relationships (Image from Sigelman, et al. [31])

also showed that building a tracing backend on top of the event based DAG model allows for greater flexibility to modify the model independently of the instrumentation and vice-versa [21]. Figure 2.7 shows that evolution.

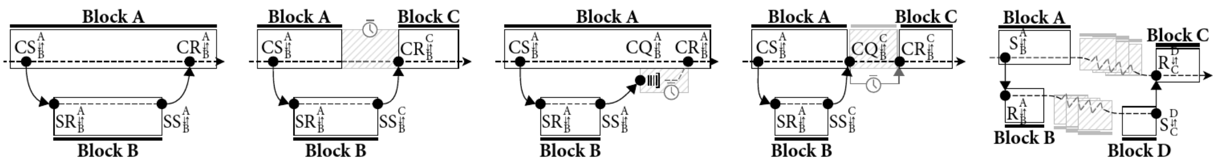


Figure 2.7: **Evolution of the tracing model in Canopy.** Initially traces were represented as trees of spans, similar to Dapper. With time, the model became more relaxed to represent new types of communication. (Image from Mace, et al. [21])

Table 2.2 describes the model used by each system mentioned above. It is worth noting that while Dapper is the lone user of the Span-Tree model, its popularity among the open-source community spawned multiple implementations, making this the model used by most readily available tracing systems.

System	Span Tree	Event DAG	Other
Dapper [31]	✓		
X-Trace [14]		✓	
Pip [27]		✓	
spTracer [25]			✓
Canopy [21]		✓	
Pinpoint [7, 8]			✓

Table 2.2: **Tracing Models.** Summarizes the type of model used by each of the described systems.

2.1.3 Trace Collection Layer

The Trace Collection layer abstracts the infrastructure and techniques required to efficiently execute distributed traces. There are two main design decisions that need to be considered when implementing this layer: **data collection** and **performance optimizations**.

2.1.3.1 Data Collection

As we mentioned before, during the execution of a trace several independent tracepoints are activated, exposing snippets of information which may be causally related, or propagating data necessary to reconstruct that causality. We have discussed in the Model Layer how meta-data can be used to reconstruct the causality relations. This section discusses a key part of the reconstruction: how the meta-data associated to one tracepoint can be propagated throughout the request path. This can be done either **out-of band** or **in-band**.

Out-of-band propagation (depicted in Figure 2.8) is used by most of the tracing systems studied in this document [31, 7, 14, 21, 29]. This scheme propagates only the causality tracking meta-data along with the request, writing all other information to the local disk. This technique reduces the impact of the tracing infrastructure on the network, as the size of the packets is only minimally increased. Since the trace information has to be persisted to disk, the overhead associated to this method of propagation is mostly concentrated on the CPU, memory, and disk usage.

X-Trace [14] developers use this form of propagation for two reasons. First, propagating the trace data along with the request means that it is lost in the event of a network failure. This is not tolerable as tracing periods of failure is especially important for future analysis.

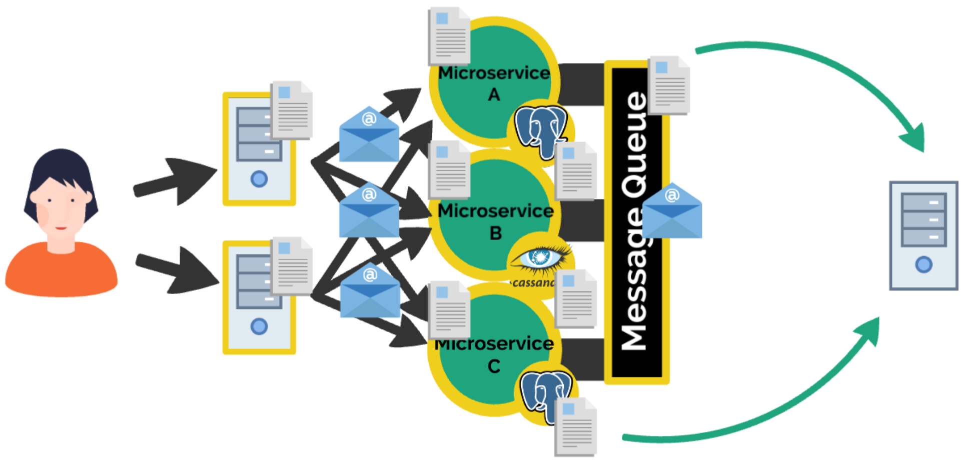


Figure 2.8: **Example of out-of-band propagation**

Secondly, increasing the size of the messages has a negative impact on the overall latency. The latter observation is reinforced by the work in [31] who note that the trace information often dwarfs the application data.

Developers of systems with non perfectly nested RPCs must also resort to out-of-band tracing in order to trace every operation [31]. Examples of non nestable RPCs are distributed components which use quorums and attempt to optimize response times. In this type of system, the response may be returned before all nodes respond to the caller. With in-band-propagation, the slower nodes would appear to not have responded at all.

In-band propagation is the other alternative in context propagation. This approach consists on sending the information captured at each tracepoint along with the requests, thus reducing disk usage, but increasing the size of the messages exchanged between components. This approach is depicted in Figure 2.9

Unlike out-of-band propagation, which needs an orthogonal infrastructure to collect the tracing data [14], in-band methods are able to aggregate and collect the information while the request is being executed on the same machine.

Pivot Tracing [22] provides an abstraction of meta-data collection called *baggage* which uses in-band propagation to causally relate queries in the request path. As we mentioned before, in-band propagation tends to increase the size of the messages, which increases request latency. However, Pivot Tracing developers claim that they have increased the efficiency of their *happened-before join*, by computing the causality for each query during the request's execution,

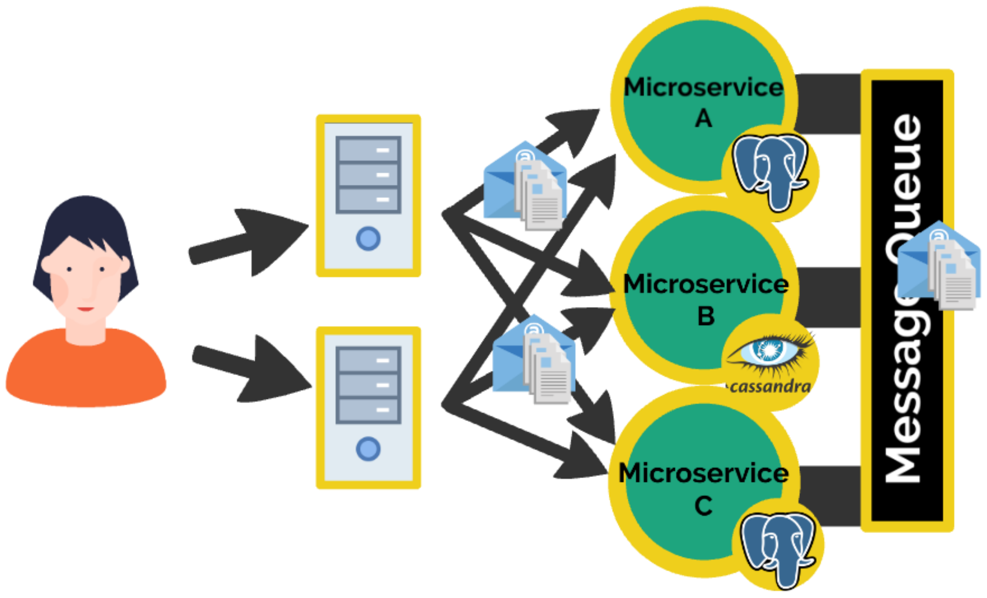


Figure 2.9: **Example of in-band propagation**

as opposed to cluster-wide *a posteriori* computation, thus reducing the amount of meta-data tuples emitted by the tracer.

Other works also use in-band data collection by aggregating performance counters in-memory, but avoid inflating messages between services by periodically reporting the data to a central collection service [23]. In this technique the causality tracking meta-data is still carried in the messages, but as we have stated previously, this has a negligible effect. While this might not seem to completely propagate information with the execution flow, surveys on this topic consider it to be an example of in-band propagation [28], as the relevant data is aggregated in-band, even though it is periodically reported.

We summarize the propagation techniques in Table 2.3. One can see the out-of-band collection is much more popular than its in band counterpart, perhaps due to its simpler implementation, which is closer to the logging solutions of current systems.

2.1.3.2 Performance Optimization

The effectiveness of tracing systems is very much dependent on their ability to capture non-ordinary requests. Since the large majority of requests do not exhibit faulty characteristics, it is important that the tracer can run permanently so as to not miss the few requests that stray from the norm [31]. For a monitoring tool to be allowed to run 24/7, system administrators must

System	Propagation	
	In-Band	Out-of-Band
Dapper [31]		✓
X-Trace [14]		✓
Pivot Tracing [22]	✓	
Pinpoint [8, 7]		✓
Canopy [21]		✓
Retro [23]	✓	

Table 2.3: **Propagation Scheme per System**

have confidence that its probe effect will not induce too much overhead on the monitored systems. For this reason, a lot of work has been put into studying how to improve the performance of data collection in distributed tracing software.

Due to the large differences between the inner workings of both data collection methodologies, it is reasonable to assume that the performance optimizations for in-band propagation models differ from their out-of-band counterparts. Therefore, while some techniques are applicable to both, we will present them separately and summarize the ones used by each system in Table 2.4.

Out-of-band propagation systems write the output of each executed tracepoint to disk to be reconstructed later. Naturally, the best way to optimize performance in this situation is to reduce the amount of disk writes that need to be performed. Generally, systems use **sampling** techniques to limit the amount of tracepoint executions by only tracing a percentage of the requests [31, 14, 21]. The decision to collect any given trace can be made at the **head**, **tail** of the trace or using a **hybrid** approach.

Head-based sampling is the most straightforward technique. When a request hits the system, the tracing component will randomly decide if the request should be traced based on a user-defined probability. Using this technique the developers of Dapper [31] were able to reduce the overall overhead to less than 1% on both latency and throughput from 16% on the latency and 1.5% on the throughput when no sampling is used.

Tail-based sampling requires tracing components to cache all traces into memory until completion. At the end of each request, the trace is only persisted if it exhibits some sort of anomaly or if it is important to what the system is trying to monitor. This is a somewhat smarter alternative to head based sampling, but has a larger resource footprint as it might have to cache hundreds or thousands of concurrent requests.

Hybrid approaches may provide the best of both worlds by using head based sampling, while still caching all requests for a small amount of time. This leaves some room for the tracing system to backtrack and collect traces that only became meaningful towards the end of their execution, while still maintaining less strain on the resources than tail-based sampling. This technique is refereed to as **Opportunistic Tracing** in [21], where it is only applied in systems with a small number of components. An interesting use of Hybrid sampling was introduced in JCallGraph [20], where only successful requests are sampled for load shedding, while all failures are collected.

In-band propagation based tools do not persist data to disk, therefore sampling based techniques may not be as useful. The major source of overhead in these tools is the amount of data stored in memory or packed in the messages. Since the captured data flows across the system, it is beneficial to greedily execute aggregations as soon as possible [28].

Pivot Tracing [22] does this by performing intermediate aggregation for queries that contain *Aggregate* or *Group By* clauses, i.e., maintaining a running sum or average, as opposed to performing the calculation later after all tuples have been collected. Retro [23] also performs some intermediate aggregation on its performance counters and reports them periodically to a central repository in order to reduce the load on the local node.

Another technique suggested by Pivot Tracing is query planning. This tactic is mostly applicable to tools that use dynamic instrumentation where queries are evaluated on the fly. The main goal is to push the operators in query joins as close as possible to the source tuples in situations where the operation is distributive. This approach is based on Fay [13] which uses a similar technique for merging tuple streams.

A special type of performance optimization in dynamically instrumented tracing systems is simply to reduce the amount of active tracepoints at any given time. This is applicable to both types of data collection as it reaches the goal of reducing the amount of collected information on a much more granular level. However, developers must be aware that removing tracepoints required for causality tracking may result in incorrect traces.

System	Sampling			Aggregation	Query Plan	Point Removal
	Head	Tail	Hybrid			
Dapper [31]	✓					
X-Trace [14]	✓					
Pivot Tracing [22]				✓	✓	✓
Fay [13]					✓	✓
Canopy [21]			✓			
Retro [23]				✓		
JCallGraph [20]			✓			

Table 2.4: **Optimizations per System**

Implementing the Trace Collection Layer requires certain infrastructure that varies wildly among tracing systems. For that reason we refrain from suggesting any sort of implementation or infrastructure design. Instead we will describe the infrastructures implemented by the most relevant works in the following section.

2.1.4 Analysis Layer

The topmost layer of our distributed tracing decomposition is reserved for analysis tools and strategies. As mentioned before, distributed tracing is only a means to an end, which is to have a high level debugging mechanism for distributed systems. This means that even the most robust tracing backend is worthless if the captured data is not made useful.

Commonly tracing systems defer the interpretation of the results to the user, by providing **manual tools** querying and/or data visualization engines, which allow for building high-level concepts from low-level data. Others attempt to simplify this process through **automated tools** for error detection and root cause analysis.

2.1.4.1 Manual Tools

Manual tools are useful for users who want to attain a better understanding of the traced system.

Querying Engines. The output of a distributed tracing backend comprises, more often than not, a large wealth of information. With that being said, it is only natural for developers to look for ways to narrow down or identify relevant information. The simplest approach is to provide a querying engine. Querying engines allow the user to filter and aggregate data points based on high-level queries.

A good example of such systems is D3 [10], which sits on top of standalone tracing backends like X-Trace [14]. This tool provides a declarative querying language that allows the user to perform operations ranging from simple filtering tasks, such as showing all events with a given trace ID, to more advanced tasks, similar to what-if analysis.

A special case of querying systems, are tools based on distributed instrumentation, such as Pivot Tracing [22] which allow the user to submit queries to a live system, while capturing new data points. The causality tracking properties of Pivot Tracing allow it to work with complex queries, unavailable in other querying tools, like aggregating the evolution of a service's throughput by client process.

A very helpful use of querying engines is to process data that can then be exported to plotting and graphing tools for visual interpretation.

Visualization Tools. Other tools make the data visualization their primary goal. While the most common visualization format for tracing information is plotting the evolution of parameters over time, many systems aim to provide new ways to view and interact with data abstractions. One of these tools is Hy+ [11], which models tracing data as a graph that they call *hygraph*. The defining feature of Hy+, shown in Figure 2.10, is the ability to encapsulate nodes and edges and view them as higher-level concepts, like communication behaviors. This abstracted view of the system can be useful for developers to visualize hard-to-detect faults like deadlocks.

2.1.4.2 Automated Tools

Automated analysis tools allow us to create models of execution that are verified against each trace.

Fault Detection. The obvious first use for automated analysis tools is automatic fault detection. Stop faults are easy to identify as a failure to complete a request is immediately visible to a user or automated tests. However, some faults can remain silent for large periods of time,

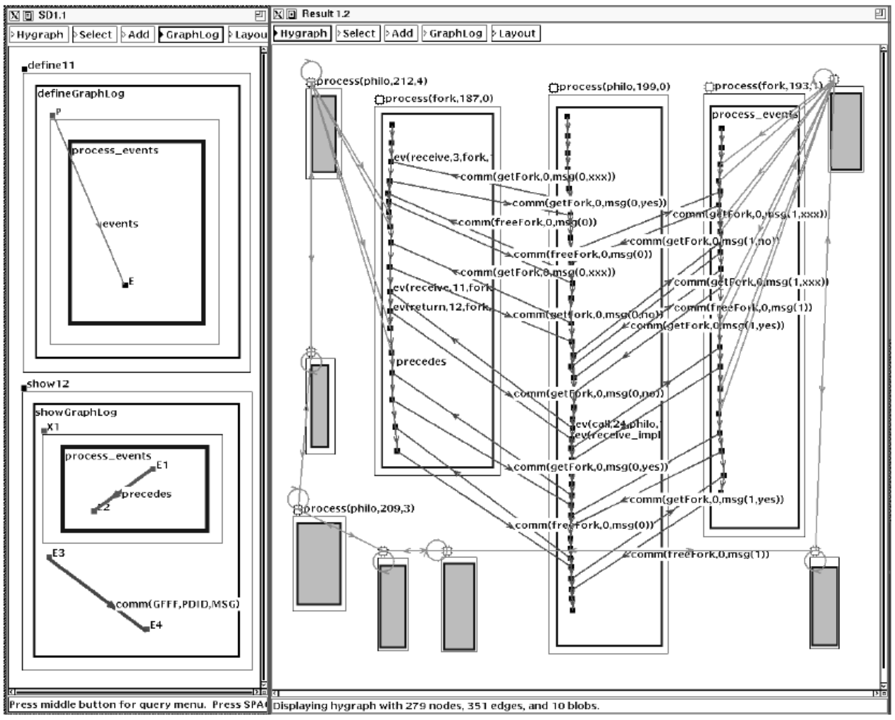


Figure 2.10: **Example of a Hy+ visualization** The left panes show abstraction definitions while the right pane shows the final hygraph (Image from Consens, et al. [12])

either due to automated recovery techniques or if they simply occur off the critical path. While silent faults are not noticeable to a user, system developers should still be interested in their detection, as they can leave the system in a working, but degraded state.

The developers of Pinpoint [7] designed a failure detector able to identify both internal (silent) and external (stop) faults. To log internal faults, Pinpoint instruments the traced J2EE platform to log exceptions that span across process boundaries. To detect external faults, Pinpoint uses an application layer packet sniffer that monitors TCP and HTTP failures which result in non-completed requests.

Pip [27], a distributed tracing system that focuses in fault detection, formalizes this approach through the use of a structured language for *expectations*. Pip users can write a program that specifies the expected structure of an execution of their software, with a good deal of granularity, and be notified if the expectation fails. This type of tool is useful for finding silent faults in high usage systems that can collect millions of traces in a single day.

Root Cause Analysis. Finding the root cause of faults is one of the most difficult and time consuming tasks for software developers, and the difficulty is only exacerbated when dealing with heterogeneous distributed systems.

With modern technology and algorithms it is possible to leverage the large amount of trace data collected from highly demanded systems to automatically approximate the root cause of faults, through machine learning, data science or clustering techniques.

Some systems attempt to compare faulty and correct executions in order to detect differences between the causal paths of requests. This approach was used by Mirgorodskiy, et al. [25] who perform always-on monitoring by first summarizing traces into a small and easy to compare format, and using a distance metric between pairs to identify the most deviant ones. The atypical traces are then combined by applying transformations that merge requests with similar paths together, thus presenting a set of possible root causes.

Pinpoint [8] uses a technique based on data clustering combined with automated failure detection. Traces are automatically marked as faulty or correct when the request is finalized, and clustering algorithms are applied on the resulting dataset to identify the combinations of components most highly correlated with request failures.

Other tools use datasets of past faulty requests and corresponding causes to build machine learning models which infer the cause and solution of recurring problems. One example is the work in [37] that captures traces of system calls which are then "cleaned", canonicalized and associated to their solution on a database. These traces are used to build a classifier that can analyze new traces and suggest root causes. When a user detects a known issue, they can use the provided troubleshooter which will automatically reproduce the problem and record a trace to be processed by the classifier. This technique differs from the others in the sense that the problem and cause are not unknown, but the space of possible causes is too big to be efficiently processed by a human.

A novel idea in this area is called Why-Across-Time Provenance [35], which is based on Why-Provenance, a concept used in the analysis of relational databases. In short, by analyzing trace data that flows in and out of a service, it is possible to produce the minimum set of inputs the caused a certain output, thus performing root cause analysis by discovering the traces that contain inputs that eventually lead to a faulty output. This tool could be used in environments with byzantine faults where components may not only stop but also respond with incorrect or arbitrary data.

The analysis tools described in this section are summarized in Table 2.5. The table shows that a lot of work is being put into automated analysis techniques made possible due to the recent advances in data-science and machine learning

System	Querying	Visualization	Fault Detection	Root Cause
D3 [10]	✓			
Pivot Tracing [22]	✓			
Hy+ [11]		✓		
Pinpoint [8, 7]			✓	✓
Pip [27]		✓		
spTracer [25]				✓
Yuan et. al [37]				✓
Whittaker et. al [35]				✓

Table 2.5: **Analysis Tools.**

2.2 Relevant Systems

In this section we will further discuss the systems that we found most relevant or closest to our proposed work. These systems were chosen because they introduced concepts used in our proposal [31], have similar goals and approaches [2, 6], or because they show that it is possible to implement distributed tracing in highly demanding production environments [31, 21]. Table 2.6 summarizes our description of each system at the end of this section.

2.2.1 Dapper

Since Dapper's [31] paper was released by Google it became one of the most influential systems in distributed tracing. As the authors state, a single request to Google's search engine can hit hundreds of machines across multiple services (search, advertising, news, videos), therefore having a way to observe the internal path of a request was a necessity.

Instrumentation. According to the authors of Dapper's paper, almost all services at Google communicate through a custom RPC library. Therefore, the developers instrumented the library to define spans around all RPCs. Additionally, span meta data is attached to each thread with thread-local variables in order to be propagated inside a service. When working with asynchronous computation, developers use a custom library to construct callbacks which attaches the span meta-data to the callback, allowing Dapper to follow the control paths transparently.

Model. Dapper introduced the span tree model. Since most services use RPC to communicate, this model is the optimal balance between ease of instrumentation, and accurate representation. While other activities such as SMTP, HTTP and SQL are also modeled by Dapper, they can be represented as simple activation of remote services and do not exhibit complex causality relations, making them a great fit for the span tree model.

Trace Collection. The developers of Dapper specifically use out-of-band propagation for two reasons: (i) trace data often contains thousands of spans, dwarfing RPC responses and possibly affecting network dynamics if sent in-band, and (ii) not all RPCs are perfectly nestable, and middleware systems who do not wait for all backends before responding would not be accurately represented.

To comply with Google's strict latency requirements, Dapper introduced head-based sampling as a way to reduce the overhead of the tracing mechanisms. Since services with more traffic are more susceptible to the tracing overhead, adaptive sampling is used so workloads with less traffic capture more traces and workloads with more traffic capture less traces. This decision hinges on the idea that ”interesting” requests will occur more often in high traffic situations, allowing the sampling rate to be reduced.

Infrastructure. During traces spans are immediately persisted to local log files. Periodically, Dapper daemons pull the logs and written to a BigTable cell (each trace is a BigTable row, with each column being a span). The entire process takes on average 15 seconds. To simplify the creation of analysis tools Dapper provides an API to simplify access to trace data.

Analysis. The Dapper UI allows users to access performance data by service, time window and execution patterns, in the form of charts or histograms. It is also possible to view each trace separately as a tree or more commonly a Gantt chart. The information can also be pulled through the Dapper API for automated analysis tools.

2.2.2 Magpie

Magpie [2] uses passive capturing of information to accurately attribute resource (CPU, memory, disk, network bandwidth) usage to individual requests in distributed systems.

Instrumentation. Magpie applies no instrumentation to the traced system. Instead it uses passive capturing on information such as system calls (to account CPU and I/O consumption) and network packets (for network bandwidth). To achieve this it uses the Event Tracing for Windows infrastructure (ETW) and WinPcap[9]. Some middleware is also instrumented to cover points were resources can be multiplexed or demultiplexed such as sockets.

Model. Magpie canonicalizes the event stream of requests into a binary tree structure where forks in the tree represent thread synchronization. When a thread blocks, a leaf node is created. This model is very similar to Pinpoint's [7, 8] path model (if the leaf nodes were discarded), and the span-tree model. This tree-based model is used so that the trees can be flattened using DFS and compared using distance metrics.

Trace Collection. The instrumentation used by Magpie outputs a tuple stream denoting ETW events. Since Magpie does not instrument the applications, the data collection is necessarily performed out-of-band. Magpie also does not apply any performance optimization techniques and relies on the performance of its underlying tracing instrumentation (WinPcap and ETW) to achieve low overhead.

Infrastructure. Each machine running Magpie is tasked with parsing the local event stream stemming from the instrumentation tools. The parser combines low-level events into the higher level model through the use of a user-defined application specific schema.

The schema defines which events can be joined based on attribute-value pairs, for example, some events can be joined because they share the same thread ID. Since resources can be reused (a thread in a pool can serve other requests), the schema also defines valid-intervals to restrict the periods in which events can be joined by a given pair. To define intervals, the schema specifies which event types indicate the start and end of a valid interval. Once the interval is closed, no more events can be added to that set via the same attribute-value pair. Valid intervals that share events are combined into a higher level graph. When a request graph is no longer accessible, i.e., all intervals are closed, the parser checks if the graph contains a unique seed event (specified in the schema) that identifies a request. If it contains the event the graph is stored, otherwise it is garbage collected.

To restore the causality relations the schema also defines synchronization statements that

[9]https://www.winpcap.org/

represent transfer of execution between threads or machines (via packets). For workloads that span several machines, the graph obtained in each machine is first canonicalized and stitched together via the synchronization constructs.

Analysis The developers of Magpie use clustering algorithms to aggregate requests. Since the requests can be flattened into a string, it is possible to use simple string-edit-distance metrics to compute the similarity. New requests are compared to the centroid of each cluster and aggregated with the most similar. This allows Magpie to identify workflows and visualize resource usage per workflow.

2.2.3 Canopy

Canopy [21] stemmed from Facebook's need to record causally related end-to-end performance data from browsers, backend services and mobile devices.

Instrumentation. Much like Google, Facebook uses mostly in-house tools in their development stack, and the causality tracking is handled by a low level library. The application-level instrumentation libraries encapsulate the event capturing by providing constructs for notating segments of processing, such as try-with-resources in Java. The instrumentation APIs are decoupled from the data model to lower the barrier of entry and allows for the introduction of new performance capturing APIs without invalidated legacy instrumentation.

Model. Canopy uses the event-based DAG as its trace model. However, to the users this event model is abstracted into a higher level representation to hide inconsistencies in the instrumentation APIs. The higher level model is composed of three constructs: execution units (a thread of execution), blocks (segments of computation within a unit) and points (instantaneous occurrences in a block). How each construct is used varies by task and instrumentation APIs are written with the high-level constructs in mind.

Trace Collection. Similarly to the previous systems, Canopy uses out-of-band propagation, and routes events to a distributed datastore via a Kafka-like message queue. To maintain good performance, the system uses a global token bucket that limits the rate at which traces can be executed. On a user-by-user basis, Canopy also provides a very maleable sampling API for more granular configuration of sampling rates. Canopy supports hybrid, or opportunistic tracing, allowing back-propagation of a sampling decision late in the request path.

Infrastructure. Facebook's tracing infrastructure under Canopy was designed to allow different teams to extract only the necessary performance information from each trace into personalized datasets. Since a trace may span components from many different teams this limits the amount of information that the analysis components will have to process. When an event is extracted, it is stored in a persistent database and into a in-memory cache where they are grouped by traceID. Presumably finished traces are taken from the cache (or the database) and processed into the higher-level model detailed above, and also annotated with performance data and causally linked. Afterwards traces are parsed by multiple threads that build independent feature datasets based on user-specific configuration.

Analysis For adequate data analysis Canopy provides multiple querying and visualization tools based on the underlying datastores, including SQL-like queries, performance graphs and Gantt charts. Users can explore individual traces, aggregate metrics or feature datasets manually or progamatically using iPython.

2.2.4 Protractor

Protractor [6] attempts to bridge the gap between tracing and profiling by including service specific profiling in traces.

Instrumentation. Protractor is built on top of a novel concept called a service mesh [19] which provides distributed tracing for free. A service mesh is a network of sidecar proxies that intercept all communication between components for load balancing and service discovery. The information captured can also be used to reconstruct a trace of the request's execution. In addition to tracing, profiling of each component is also used. Since all components run in the JVM, Protractor uses Java Flight Recorder which simply requires that programs are ran with special flags.

Model. Jaeger is used as a tracing backend. Since Jaeger is a Dapper clone it uses the span tree model. Protractor was built for microservices that communicate through REST APIs, making the span model a great fit.

Trace Collection. Data Collection is performed out-of-band by sending data to a component called mixer that also configures the proxies. The mixer reconstructs spans and also augments each span with the performance data. No mention is made of sampling techniques,

System	Data Extraction	Model	Trace Collection		Analysis
			Collection	Performance	
Dapper [31]	Static Instrumentation	Span Tree	O/B	Sampling†	Charts
Canopy [21]	Static Instrumentation	Event DAG	O/B	Sampling*	Charts Query Engine Programmatic API
Magpie [2]	Packet Capturing System Call Monitoring	Binary Tree	O/B	–	Trace Clustering
Protractor [6]	Network Data Capturing (Service Mesh)	Span Tree	O/B	Sampling†	Charts

Table 2.6: **Techniques per layer for each relevant system.** O/B stands for Out-of-Band, † represents head-based sampling, while * represents hybrid sampling

but the system only augments traces that were considered faulty, in order to reduce the amount of data storage necessary.

Infrastructure. Service meshes were designed for container orchestrators such as Kubernetes [4]. Each of the traced services runs on a separate Kubernetes pod together with two sidecars: the proxy (Envoy) and the profiler. Other pods include the Jaeger interface, collector, and the mixer (a control plane running on Istio[10]). Cassandra is used for storage but runs outside the cluster. When services interact, the proxies collect request attributes and route them to the mixer which is tasked with span reconstruction and fault detection/augmentation. Faults are detected by comparing the latency of a span with the previous 500 executions. If the request is not faulty, the span is augmented with the ID of the performance data on the database.

Analysis Protractor relies on manual analysis. It provides the normal Jaeger interface (a simple query engine for finding traces and a Gantt chart for each trace), which was modified to allow the download of profiling data from the database. The profiling data can be viewed through Java Mission Control.

[10]https://istio.io/

2.3 Analysis and Discussion

Each system detailed in the previous section has characteristics that influenced our proposed architecture. However, some shortcomings, design decisions or simply the lack of a proper release, impede them from being used for our goal of providing distributed tracing by combining open source components.

Dapper has inspired numerous open-source clones, like Zipkin and Jaeger. However, they require that the traced components or underlying libraries are instrumented. This is not possible in many cases as databases, message queues, or other services are proprietary and therefore unmodifiable black-boxes.

Canopy shows that the decoupling of model and instrumentation provides more flexibility for future changes, however no open-source implementations of this model are currently available. The opportunistic sampling described is also an interesting design choice, that open-source solutions do not provide. Canopy also suffers from the same problem as Dapper, where instrumentation is required to propagate data, making it unusable for systems of black boxes.

Magpie's passive data capturing based on a user-specified schema is a very interesting hypothesis, but it requires expert understanding of the systems interactions with low level kernel functions. On the other hand, its ability to attribute resource usage to specific threads of processes is a great aid for diagnosing system failures.

Protractor relaxes the low-level resource attribution of Magpie to service-wide performance profiling, but its reliance on Envoy[11] and Istio make it unusable for protocols other than HTTP. Many commonly used applications like Zookeeper, Cassandra or other databases communicate with their clients through custom protocols (instead of HTTP), which could be added in the future but only if the ability to propagate context through headers is provided.

The analysis of relevant systems is summarized in Table 2.6. It is clear that out-of-band propagation is the most common choice, along with sampling for performance. It is also noticeable that both systems that use non intrusive techniques rely on capturing network data.

[11]https://www.envoyproxy.io/

3 Architecture

The purpose of this chapter is to illustrate the high-level structure of a tracing enabled system, while taking the opportunity to detail the components of our main test case, Feedzai's fraud-detection pipeline, as well as the tracing-specific infrastructure. Afterwards we delve deeper into Melange's API design.

3.1 High-Level Architecture

This section follows the following structure: in Sections 3.1.1 and 3.1.2 we outline the high-level architecture of Feedzai's system and the tracing infrastructure, respectively, while Section 3.1.3 shows the full picture of a tracing enabled deployment of Feedzai's products.

3.1.1 Feedzai - Reference Traced System

Our instrumentation library does not make any assumptions regarding the architecture of the traced software, in general, Melange is simply targeted for highly distributed heterogeneous systems. However, since it was designed with the primary goal of fulfilling the tracing requirements of Feedzai's system, we will use it as a reference point. The system is comprised of the following components (depicted in Figure 3.1):

Streaming Engine	The Streaming Engine (FSE) processes each transaction as it reaches the system, and performs the bulk of the business logic. This logic runs on the critical path of the underlying system, forcing it to be extremely latency-sensitive (in the order of milliseconds in the 99.999 percentile).
Profile Storage	Distributed NoSQL database that stores customer profiles computed as part of Feedzai's business logic.

Tokenization Engine	It is tasked with the encryption and decryption of sensitive data that must not be stored in plaintext.
Token Storage	Database that stores the tokenized sensitive fields of previous transactions.
Event Monitor	Once the business logic is executed, some transactions may still require human confirmation. In the Event Monitor (FEM), analysts can view each transaction's metadata and approve or reject it.
Event Storage	Database that stores the events that require human approval.
Message Queue	Persistent distributed message queue that is used to route transactions from the streaming engine to the event manager and other unrelated Feedzai components.

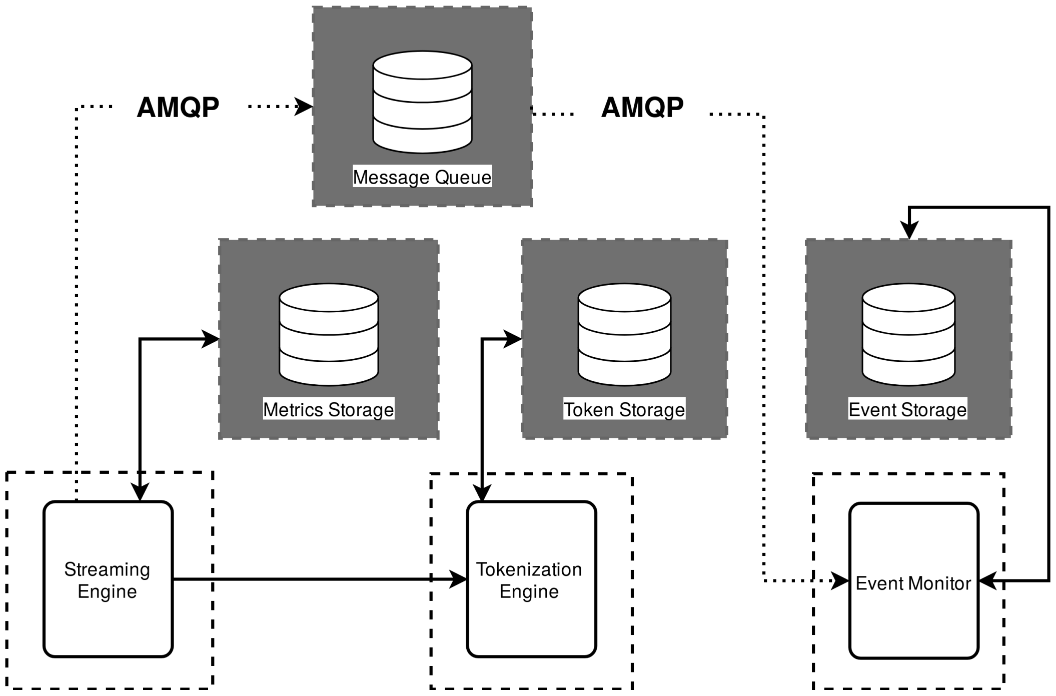


Figure 3.1: **High-Level overview of a Feedzai deployment.** Dotted boxes signify node boundaries, dotted arrows represent request forwarding and dark boxes stand for black-box components.

When a transaction reaches the streaming engine, it must first be tokenized in order to obfuscate any sensitive data before processing. This step is performed through HTTPS, as the tokenization engine exposes a simple REST API. Upon receiving the tokenized transaction, the Streaming Engine will execute the client-specific business logic, often querying the Profile Storage in the process. Once the transaction has been processed it is placed in an AMQP-based [34] persistent message queue, leading it to the Event Monitor. Here, the transactions are stored in a SQL database, from which they are later retrieved for human analysis. It should be mentioned that all of these components can, and will, be replicated and partitioned for better load-balancing and fault tolerance.

In Figure 3.2 we describe the aforementioned behavior as a sequence diagram. It is important to note that, while the overall processing sequence followed by transactions is the same across requests, certain steps might be omitted as the streaming pipelines are client-specific. Candidates for omission are Token Storage writes when the sensitive fields have been tokenized previously, or accesses to the Profile Storage in pipelines that do not require it.

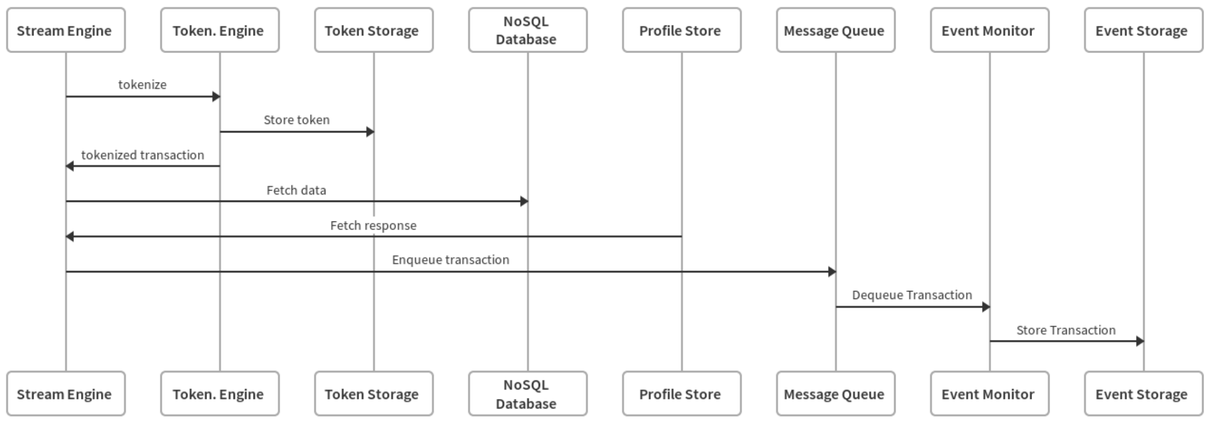


Figure 3.2: **Sequence diagram depicting an end-to-end execution of a deployment**

3.1.2 Tracing Infrastructure

The tracing infrastructure encompasses seven components, with some – Melange and the instrumented drivers – being the focus of our work, while others – Jaeger Tracing Engine, OpenTracing and Elasticsearch – are well known open-source technologies. Each component is described below:

Jaeger Agent	The agent resides on each node, collecting spans created by the tracing library and routing them in batches to the collector.
Jaeger Collector	The collector processes spans received from the agents, performs validation and stores them in Elasticsearch

Jaeger Query Engine	The query engine powers the UI used to analyze the traces.
Melange	Our library instruments each process' code so that specific blocks are traced and sent as spans to the Jaeger Agent.
OpenTracing	Melange is implemented on top of OpenTracing to maintain compatibility with standard tracing engines.
Instrumented Drivers	The instrumented drivers label the client-side spans with process-level meta-data extracted from the black-boxes.
Elasticsearch	Elastisearch is the database used to store the spans. It was chosen due to its powerful searching capabilities.

Figure 3.3 illustrates the relationships between the aforementioned components. While the diagram shows the Jaeger Collector, Query Engine and Elasticsearch in the same node, each component could be individually replicated for load balancing and fault-tolerance. Although it is not a requirement, the agent should always be present in whichever nodes the application processes reside, as to avoid the overhead of propagating spans through the network.

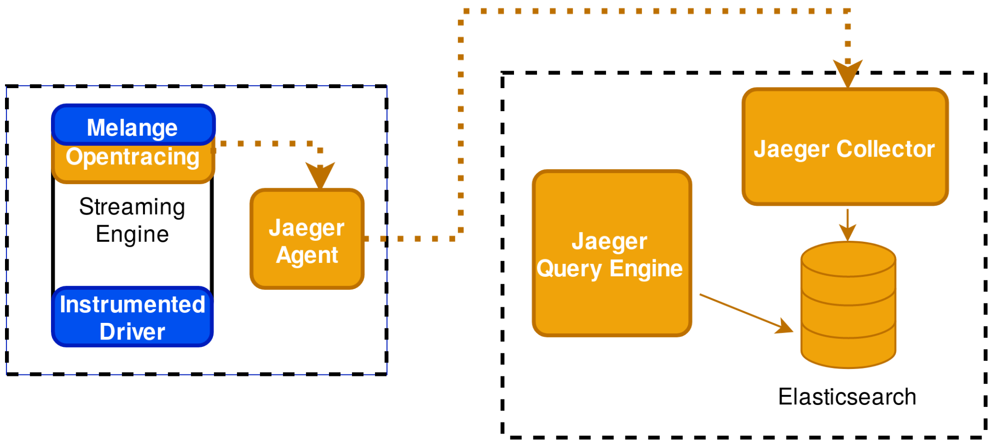


Figure 3.3: **High-level overview of the tracing infrastructure.** Dotted boxes signify node boundaries, dotted arrows represent request forwarding, orange components are newly introduced but external while blue components are newly introduced and newly developed.

3.1.2.1 Trace Granularity

When instrumenting any system, one must take into account the desired level of granularity. As each tracepoint adds a small overhead to the performance of the traced system, this becomes increasingly important in latency-sensitive systems like Feedzai's.

In the previous chapter, most of the discussed systems traced at the process level, for they usually followed service-oriented approaches. Tracing at this level unlocks information about the targeted service's specific replica or partition that handled the request. Feedzai's streaming engine executes a set of processing steps that vary for each specific use-case, and even though the engine is replicated and partitioned, each request is processed by a single engine (excluding replication). It is easy to understand that a failure or delay in certain pipeline steps could result in bad performance or failures in execution, which forces us to not only trace at the process level, but also at the level of the streaming pipeline. As this is much deeper into the system we chose to accept a larger overhead (in performance, and source-code modification) in exchange for richer traces.

3.1.3 Tracing Architecture

The final architecture of a traced deployment is built by combining the traced system's architecture with the tracing infrastructure and source-code instrumentation, as depicted in Figure 3.4.

Each of Feedzai's components is first instrumented with the tools provided by Melange. Processes that interact with components that are considered black-boxes, such as message queues or databases, are modified to do so through the instrumented drivers built to enrich the spans collected on the client-side. The data extracted by the instrumentation code is propagated asynchronously to a Jaeger agent running on each node. The spans sent to the Jaeger Collector by the Agents are then validated and stored in an Elastisearch cluster present in each environment. A cluster will provide better load balancing than a single node reducing the probability that traces will be discarded due to backpressure.

Although neither process has demanding hardware requirements, Jaeger Collector and Query Engine run in separate nodes. This ensures that the collector's performance is not affected even if several users decide to query the stored traces at once.

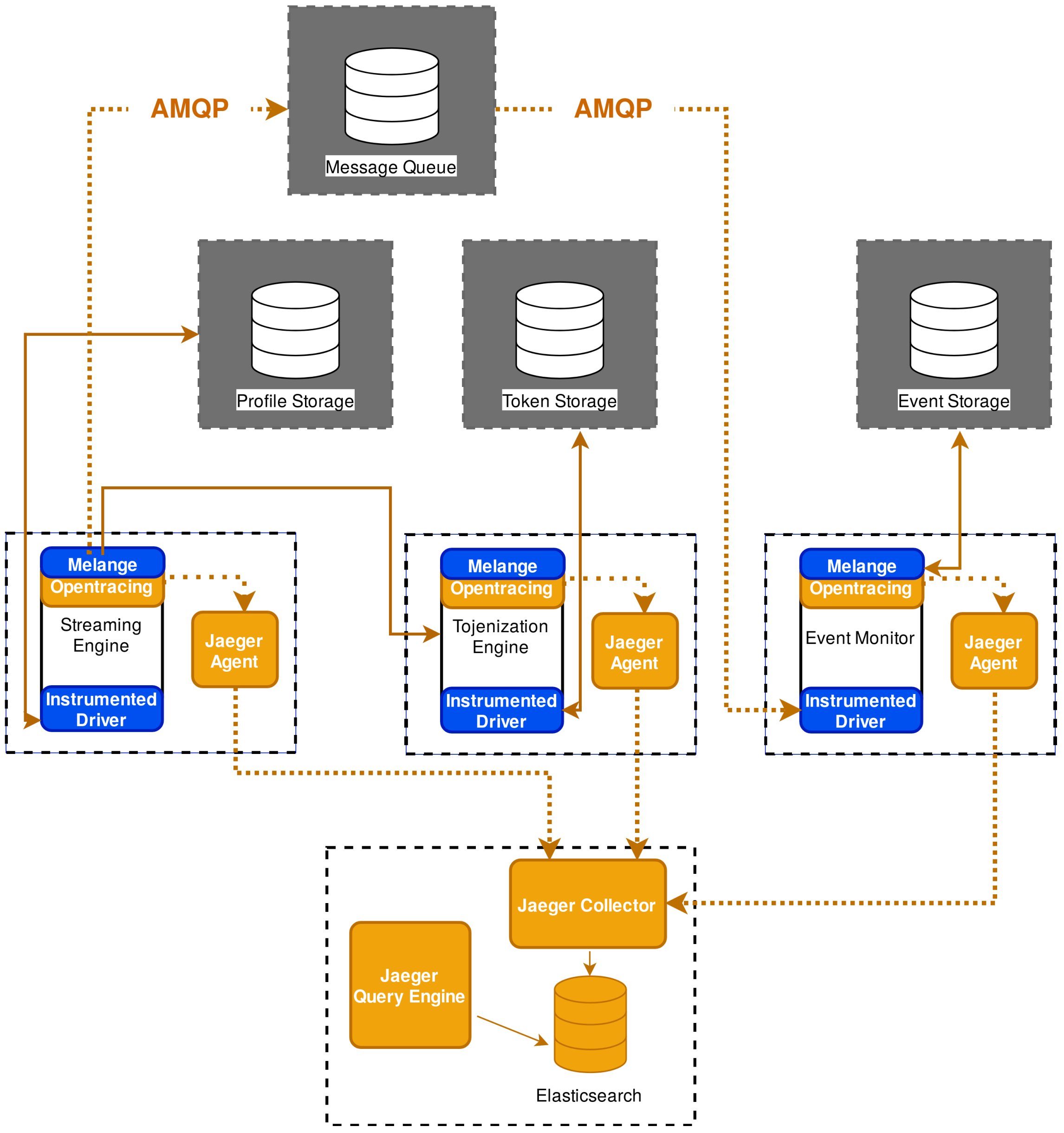


Figure 3.4: **High-level view of the components involved in distributed tracing of a Feedzai deployment.** Dotted boxes signify node boundaries, dotted arrows represent request forwarding, orange components are newly introduced but external while blue components are newly introduced and newly developed.

3.2 Melange's Architecture

In this section we will describe the internal structure of Melange's APIs.

3.2.1 Architectural Objectives

Before delving deeper into the library's structure, it is important to clarify the objectives placed on the final product. We have designed Melange's architecture to fulfill the following goals:

Separation of Concerns	The instrumentation code should be mostly self-contained to minimize the developer's cognitive load when determining the relationships between tracepoints. Understanding the application's code should be no more difficult with instrumentation than without.
Tracing Asynchronicity	In latency-sensitive applications, asynchronous programming is essential. The middleware must be able to trace asynchronous execution with minimal developer effort. This includes includes tracing events across layers of indirection, like when queued for asynchronous processing.
Black-Box Visibility	Melange must extract process-level meta-data about interactions with black-box components whenever possible.
Tracer Agnostic	The distributed tracing landscape is fast-moving and constantly evolving. Companies that leverage tracing for Application Performance Monitoring are becoming increasingly prevalent. Therefore, this middleware must not impose any restrictions on the underlying tracer implementation, other than it using the Span Model.
Low Overhead	To find the most elusive bugs or wrinkles in performance, distributed tracing software must always be enabled. Doing so carries a performance cost, thus, Melange must attempt to reduce the cost imposed on the traced application's performance to the bare minimum.

3.2.2 Architectural Overview

Our library 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. Melange is also completely Open-Source and available on Github[1].

3.2.3 API

Following the Interface Segregation Principle [24], we designed three smaller APIs that can be used independently, or together if the instrumented application so requires. The three APIs are to be used as follows:

BaseTracing API	Used for single-threaded, single-process applications.
TracingWithId API	Used for single or multi-threaded, single-process applications, that feature a uniquely identifying event ID.
TracingWithContext API	Used for single or multi-threaded, single-process or distributed applications.

Additionally, each API has been extended to allow tracing executions by manually starting and finishing spans. We call this extension **Open API** (prefixed by the original API name). Figure 3.5 displays the API hierarchy, and below, we will describe each one in depth.

3.2.3.1 Tracing API

The BaseTracing API is the simplest but it is also the least flexible of all the APIs we have developed. It is comprised of nine methods which are displayed in Listing 3.1. Through this API, context is passed between tracepoints by storing it in thread-local variables.

This API provides similar semantics as other tracing instrumentation libraries but under a more concise API that improves readability and reduces the impact of tracing on the codebase.

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

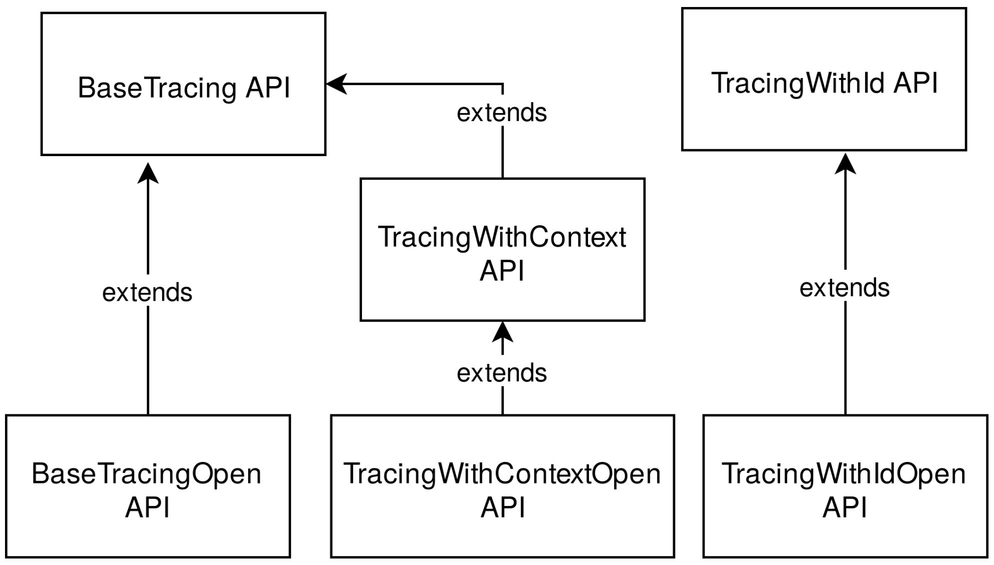


Figure 3.5: **Hierarchical view of the tsracing APIs**

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 its parent will become active.

```
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 3.1: **BaseTracing API definition.** Larger version in Listing A.1.

Capturing Asynchronicity As Listing 3.1 shows, 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 it, 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 (shown in Listing 3.2), that allows custom classes to benefit from our library's functionality. These two features fulfill our second proposed objective for this architecture: easily **trace asynchronous behavior**.

```
public interface Promise<T, P extends Promise<T, P, E>, E extends Throwable> {
   Promise<T> onCompletePromise(Consumer<T> callOnCompletion);
   Promise<T> onErrorPromise(Consumer<E> callOnError);
}
```

Listing 3.2: **Promise API definition**

3.2.3.2 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.

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 [9] have also leveraged uniquely identifying event IDs for trace reconstruction purposes, as this is considered both a common feature, and an easy change. The TracingWithId API is shown in Listing 3.3.

```
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 3.3: **TracingWithId API definition.** Larger version in Listing A.2.

The similarities between the Tracing API and TracingWithId API are obvious, with the main change between the two being 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.

Tracing across Threads. The naive approach to using event IDs for trace reconstruction 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 have designed a simplified tracing approach for such situations that relies on a per-request event ID. To understand our approach it is important to remember that the span-model records simple activation relationships, meaning that when tracing across threads the code-fragment that activated the context-switch is the same code-fragment that activated subsequent code-fragments in the new thread. Therefore, we can 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.

Without this API the only solution for tracing requests that are handled by multiple threads would be to explicitly propagate the tracing context in the form of an object. In most cases this would require structural code modifications in the form of adding tracing specific method parameters or instance fields, which would violate our first architectural objective.

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 (as displayed in Listing 3.4), we will once again be presented with non-deterministic traces.

```java
public void do(String eventID){
    tracer.newTrace(() -> doAsync(eventID), "Do", eventID);
    tracer.addToTrace(() -> doAsyncOther(eventID), "DoOther", eventID);
}
@Async
public void doAsync(String eventID){
    tracer.addToTrace(() -> longRunningThing(), "Do Async", eventID);
}
@Async
public void doAsyncOther(String eventID){
    tracer.addToTrace(() -> longRunningThingOther(), "Do Async Other",
        ↪ eventID);
}
```

Listing 3.4: **Tracing across threads with TracingWithId API**

3.2.3.3 TracingWithContext API

The `TracingWithContext` API, described in Listing 3.5, 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 3.5: **TracingWithContext API.** Larger version in Listing A.3.

As with `TracingWithId` this API is very similar to `BaseTracing` with the exception of the `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 the `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).

The `TraceContext` interface is purposely simple, and it is described in Listing 3.6.

```
public interface TraceContext<T> {
        T get();
}
```

Listing 3.6: **TraceContext API**

3.2.3.4 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 in Section 3.2.3, 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.

In Listing 3.7 we describe this extension for the BaseTracing API. This is a very small incremental change that simply adds an Object parameter representing the object to which we will associate the span.

```
public interface TracingOpen extends Tracing {
    <R> Promise<R> addToTraceOpenPromise(final Supplier<Promise<R>> toTraceAsync, final Object object, final String
         ↪ description);
    <R> CompletableFuture<R> addToTraceOpenFuture(final Supplier<CompletableFuture<R>> toTraceAsync, final Object object,final
         ↪  String description);
    void addToTraceOpen(final Runnable toTraceAsync, final Object object, final String description);
    <R> R addToTraceOpen(final Supplier<R> toTraceAsync, final Object value, final String description);
    void closeOpen(final Object object);
}
```

Listing 3.7: **Open extension for the BaseTracing API**

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* $\rightarrow$ *dequeue* $\rightarrow$ *store context* $\rightarrow$ *finish span* $\rightarrow$ *continue with previous context* instrumentation pattern proved useful in connecting the previously almost independent event arrival, preprocessing, storing and event processing loops. This, much like the TracingWithId API prevents unnecessary structural modification of the traced application that would be necessary when instrumenting with other frameworks.

3.2.4 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. When the application's code makes a call to the black-box via the programmatic client, the aspect code will intercept the execution and, by calling Melange's API, tag the active span with the extracted data.

At this point the reader may be curious as to why we chose the more complex Aspect Oriented Programming for instrumentation purposes (instead of simply modifying the code and recompiling) and the answer is simple: 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.

In some systems the dependency instrumentation can be made part of the compilation or deployment process of application, while in others it must be a separate process that is performed before compilation, replacing the non-instrumented dependencies. In either case, developers can use our instrumentation routines in their projects with no additional changes. The data extracted will be added as a tag to the span representing the black-box's execution.

For the purposes of this project we have instrumented two black-box systems: Cassandra[2] and RabbitMQ[3].

[2]https://github.com/GoncaloGarcia/Black-Box-Instrumentation/tree/master/cassandra
[3]https://github.com/GoncaloGarcia/Black-Box-Instrumentation/tree/master/rabbitmq

4 Implementation

In this chapter we will introduce the internal structure of Melange's modules, and detail how the semantics introduced in Chapter 3 were implemented.

The module-based hierarchy of the tracing library is described in Figure 4.1. In following sections we will further detail each module.

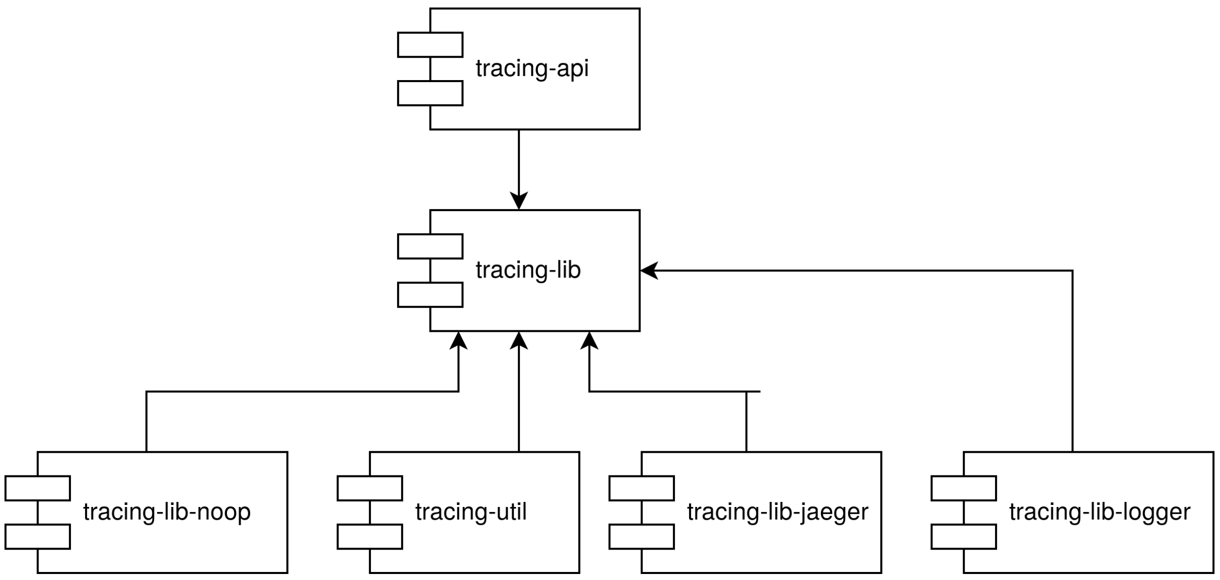


Figure 4.1: **Module-based decomposition of Melange**

4.1 *API Implementations*

In this section we will describe each implementation of the instrumentation API.

4.1.1 Tracing-Lib

The `tracing-lib` module contains the Open-Tracing based implementation of the Tracing API. This library provides all the data structures and algorithms required to use the features described in Section 3.2.3.

4.1.1.1 Tracing API Implementation

Implementing the base tracing API implementation was straightforward, as it maintains the same semantics as OpenTracing, changing only the API. Following this property, the implementation becomes an exercise in API call translation between Melange and the OpenTracing implementation of the developer's choice. Simply put, we have created a Decorator for the OpenTracing API, that provides a clearer and less verbose interface. However, it does not implement the full OpenTracing specification, providing only the features to trace single process programs and leaving the rest of the functionality to our other APIs.

4.1.1.2 TracingWithId API Implementation

As stated in the API description, through the TracingWithId API, instead of a single active trace per thread, there can be multiple active traces, each pertaining to one specific eventID.

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, that represents the current tracing scope for the request.

When a new trace is started, 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. This ensures that active span for the eventID is always at the top of 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.

4.1.1.3 TracingContext 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.

Interoperability with Instrumentation Frameworks As stated in the previous chapter, Melange provides an interface (depicted in Listing 3.6) 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 choose 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.

Serializing and deserializing context Unlike other instrumentation frameworks Melange does not distinguish between serialization format, and the context is serialized as plain-text. This change frees the programmer from having to implement multiple serialization methods. The only requirement for Melange's serialization is that the communication protocol supports custom headers, which we believe is a reasonable assumption, as common inter-process communication protocols such as HTTP or AMQP support this feature. In Listing 4.1 and Listing 4.2 we show two examples of the deserialization protocol, the first over HTTP and the second using AMQP.

```
RequestEntity.HeadersBuilder<?> builder = RequestEntity.get(url).accept(type);
tracer.serializeContext().forEach((k, v) -> builder.header(k, v));
restTemplate.exchange(builder.build(), type);
```

Listing 4.1: **Deserialization over HTTP:** Example uses the Spring Framework RestTemplate

```
Map<String, String> context = tracer.serializeContext();
AMQP.BasicProperties props = properties.builder().headers(context).build();
channel.basicPublish(topicName, routingKey, props, body);
```

Listing 4.2: **Deserialization over AMQP:** Using the RabbitMQ Java Client

The code for deserializing context in both environments is extremely similar. This was the main reason behind our decision to provide a single deserialization format. After experimenting with OpenTracing, we felt that the advantage of having different primitives for `TextMap` and `HTTPHeaders` was minimal.

Preserving the root span. In the previous section we discussed that the reference to the root span of a given trace should never be lost so that all subsequent spans would be part of the same trace, even if executed asynchronously. However, when crossing process boundaries, the reference is naturally lost. For such situations Melange introduces another API method named `newProcess`, that works similarly to `newTrace`, except it will become a child of the thread-local context (if available), or a context object passed as argument.

As OpenTracing is simply an API, some of the features necessary to implement our specified behavior is implementation-specific, therefore, this library cannot be used by itself, and developers will instead depend on one of the concrete implementations described below.

4.1.2 Tracing-Lib-Jaeger

This is the main implementation of our API for distributed tracing. Jaeger is an open-source distributed tracing engine based on Google's Dapper [31] that is compatible with OpenTracing. As we have previously stated, most of almost all of the functionality is implemented on top of OpenTracing as most tracing engines are compatible with it. Therefore this module contains only the Jaeger specific functionality, such as serialization formats and configuration. We chose Jaeger as our tracing engine for its ease of deployment, fast implementation of the OpenTracing specification, and more intuitive UI when compared to its main alternative, Zipkin.

4.1.3 Tracing-Lib-Logger

Most complex distributed systems will have some form of logging, or probing, to provide insight regarding the system's state. As workloads and complexity grow, it becomes unfeasible to manually check these logs, and teams often write custom log analysis tools for parsing and discovering abnormalities. Although we propose distributed tracing as an improvement upon existing logging tools, it is not reasonable to assume that developers would want to completely revamp their monitoring infrastructure, abandoning their tried and trusted tools, in favor of a new technology that might be foreign to them. For that reason we provide a logging-backed implementation of our API, that allows developers to continue working with their old tools and lowering the barrier of entry for introducing tracing into complex systems.

4.1.4 Tracing-Lib-Noop

Like in all other non-trivial software tools, it is inevitable that bugs or unexpected behavior will occur when tested in a real production environment. In performance-critical systems, a few bad requests might be enough to break SLAs which might have financial repercussion. Furthermore, in highly available distributed systems it is simply not possible to incur downtime to debug failures. Consequently, developers will be weary of introducing new monitoring tools into their production deployments [32], even when fully convinced of their benefits. To provide an additional level of safety, we introduced the No-Op Tracer into our library. As the name suggests, this implementation simply ignores every call to the API, making it as close to a non instrumented system as possible.

4.1.5 Tracing-Util

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

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.

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. One example of such would be to use the `logger-tracer` in high demand situations, which would still allow the system to probe each request, unlike reducing the sampling rate, but would have a smaller overhead when compared to distributed tracing. Another very important use is to enable the No-Op Tracer when a critical bug is discovered in the monitoring infrastructure, preventing monitoring-related downtime. To reduce the impact of tracing on the steady state performance of the system, one could even use this implementation to enable tracing only when performance is degraded or issues arise (for example when SLAs are being broken).

5 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.

The evaluation is structured as follows: In **section 5.1** we perform a qualitative evaluation, discussing the features created for Melange and how they compare against the instrumentation frameworks currently available on the market, **section 5.2** will present the quantitative evaluation consisting of performance benchmarks as well as source-code modification analysis, finally in **section 5.3** we will recount some case-studies that show the advantages of our middleware in practice.

5.1 Qualitative Evaluation

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

5.1.1 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 Listings 5.1 and 5.2 where we describe the semantics for starting a new trace in OpenTracing and Melange, respectively.

In Melange's example it is immediately clear that a new trace is starting. OpenTracing, on the other hand, requires some familiarity with its tracing API to understand what is being done. Additionally, the code becomes cluttered with tracing-related configuration, which violates our first design objective: **separation of concerns**.

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

Listing 5.1: **Starting a trace with OpenTracing**

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

Listing 5.2: **Starting a trace with Melange**

Comparing the semantics for continuing to a trace, as shown in Listings 5.3 and 5.4, yields similar results.

```java
try(Scope s = tracer
           .buildSpan("Subproblem")
           .startActive(true){
    longRunningThing();
}
```

Listing 5.3: **Continuing a trace with OpenTracing**

```java
tracer.addToTrace(() -> longRunningThing(),
                              "Subproblem");
```

Listing 5.4: **Continuing a trace with Melange**

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

Programs that rely on Java's `CompletableFuture` to encapsulate asynchronous computation also benefit from a smaller code footprint and increased encapsulation of tracing functionality when using Melange, as demonstrated in Listing 5.5 and Listing 5.6.

```java
Scope scope = GlobalTracer.get().buildSpan("Get Address").ignoreActiveSpan().asChildOf(span)
                                                             .startActive(true);
CompletableFuture<Resource<Address>> addressFuture = asyncGetService.getResource(item.address,
                                           new TypeReferences.ResourceType<Address>() {});
addressFuture.thenApply(future -> {
    scope.span().finish();
    return future;
});
```

Listing 5.5: **Tracing a CompletableFuture with OpenTracing**

```java
TraceUtil.instance().addToTraceAsync(() -> asyncGetService.getResource(item.address,
      new TypeReferences.ResourceType<Address>() {}), "Get Address", item.id.toString());
```

Listing 5.6: **Tracing a CompletableFuture with Melange**

Again, when using OpenTracing, the application code becomes much less readable, as developers that are unfamiliar with the idiosyncrasies of tracing Java programs will have to understand OpenTracing specific concepts like the difference between Scopes and Spans (i.e., finishing a Span and closing a Scope have different outcomes), which detract from the readibility of the code. When designing Melange, we sought to create a completely self-contained solution whenever possible, such that developers need not know the intricacies of the tracing models and their implementations.

5.1.2 Tracing Asynchronous Executions

Developers of latency-sensitive systems will, undoubtedly, resort to asynchronous and parallel programming to maximize the performance of their software. Consequently, tracing systems must provide a simple way to trace asynchronous computations.

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 Listings 5.7 and 5.8, 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 5.7: **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 5.8: **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 5.9 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 5.9: **Usage of the Open extension for the BaseTracing 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 5.7. 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.

5.1.3 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.

5.2 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[1] and Micronaut's Petstore[2]. 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.

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

5.2.1 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 Chapter 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.

5.2.2 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.

Code Evaluation Metrics. 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 on the following:

Lines Added	Aiming for a small number of lines added just for tracing increases the simplicity of the instrumented code.
Lines Modified	Often times adding new lines is not necessary and instead existing lines are modified, which also affects the readibility of the code.

Signatures Modified	Adding tracing-specific parameters to method signatures should be avoided as it tightly couples application and instrumentation code.
APIs Modified	Modifying APIs that interface with other programs or modules should be avoided at all costs so that the tracing dependencies can be confined to the modules that require them.
Classes Added	Adding new classes to encapsulate the tracing functionality is not necessarily bad, however, we propose a fully self-sufficient solution, hence we consider this a negative point.

Code Evaluation Methodology. 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.

5.2.3 Code evaluation results and analysis

5.2.3.1 Petstore

We will now discuss the results obtained from instrumenting Petstore. When analyzing the results in Table 5.1, 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 5.1: **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, exemplified in Section 5.1.1. 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	Signature Changes	APIs Broken	Classes Added
Melange	0	17	0
OpenTracing	0	17	1

Table 5.2: **Impact of tracing at the structural level on the Petstore project**

Table 5.2 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.

5.2.3.2 Sock-Shop

The results of the Sock-Shop code analysis are similar to the ones obtained for Petstore (as shown in Table 5.3). Instrumenting with Melange results in more modified lines-of-code, more so (in comparison with OpenTracing) than in Petstore, because Spring does not require changing REST clients in order to add custom headers. On the other hand, much fewer lines of code were added.

Library	Lines Modified	Lines Added
Melange	20	59 (24*)
OpenTracing	10	177 (80*)

Table 5.3: **Impact of tracing at the line-of-code level on the Sock-Shop project.** The * symbol stands for lines added for import statements.

The very clear disparity here was mostly caused by a single service (Orders). In this service certain methods run asynchronously through the use of a `CompletableFuture` (originally it was Spring's Async, but this was modified to simplify instrumentation). This means that a callback method must be attached to the `CompletableFuture` to finish the span once the method completes. While Melange does this automatically, OpenTracing does not have any specific functionality to handle this which forces developers to do it manually, leading to an increase in lines of code. While, in theory, this could be extracted to a method, it would almost be a re-implementation of our API and would result in tracing functionality creeping into the application's code).

Library	Signature Changes	APIs broken	Classes Added
Melange	12	0	0
OpenTracing	12	0	1

Table 5.4: **Impact of tracing at the structural level on the Sock-Shop project.**

Similarly to the PetStore instrumentation, the only structural difference between OpenTracing and Melange was that the former required adding a new class for context propagation between services. Unlike Petstore, however, this project had no broken APIs, due to the way the Spring Framework handles request headers. In Spring adding headers does not require modifying method signatures, however to access them one must add an additional parameter to that endpoint's controller. In other applications that do not abstract away the underlying

HTTP or AMQP libraries this step would not be necessary, which is demonstrated by our instrumentation of Feedzai's components.

5.2.3.3 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 5.5 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 5.5: **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	Signature Changes	APIs broken	Classes Added
Melange	0	0	0
OpenTracing	5	1	1

Table 5.6: **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 modifications with a comparable number of additions. Additionally Melange provides additional features while maintaining the instrumentation advantages described above.

5.2.4 Performance

5.2.4.1 Performance Metrics

To analyze the performance of our instrumentation framework we will focus on the following metrics:

Latency	Feedzai's applications are latency-sensitive, having strict SLAs that require low and consistent latency across the percentile spectrum.
Throughput	As stated above, Feedzai's systems must be able to handle millions of transactions per day. Therefore it must be able to sustain high throughput without worsening latency.
CPU Usage	It is inefficient to have unused hardware capacity when running software in production. Consequently, companies provision their hardware to handle the program's demands and not much else, which means that monitoring frameworks must not increase the CPU usage significantly so as to push the program's requirements over the edge.
Memory Usage	Our library is aimed at JVM applications, which means that we must keep memory usage to as slow as possible in order to avoid stop-the-world GC pauses.
Disk Usage	In high throughput systems, enabling tracing means collecting millions of traces and storing them in disk. Our library's trace size in disk must not outweigh its benefits forcing developers to disable it.
Bandwidth Usage	As previously stated, our library uses out-of-band propagation, meaning the trace data is not sent along with the requests. However, some meta-data must be propagated in-band to allow for trace-reconstruction. This meta-data must be small enough that it does not have significant impact on the network.

5.2.4.2 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 transaction authorization requests at a controlled rate. For both micro-services projects we simulate multiple users accessing the different pages of the website concurrently.

We chose not to micro-benchmark each component of either system for different reasons. Benchmarking each of Feedzai's products independently would be futile, as they are always deployed in tandem and each request passes through each product in the same order. For Petstore and Sockshop, there would be no difference between the micro and macro-benchmarks, since the requests are already evenly distributed among each micro-service.

To calculate the request latency we will measure the round trip time of a request sent to the system. The same data can be used to measure throughput, as we can simply divide the number of completed requests by the benchmark's duration. The resource usage of all systems will be measured by probes on each machine.

5.2.4.3 Performance evaluation environment

There were a few differences in the environment between Feedzai's benchmarks and Sockshop/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:

CPU	Intel Xeon CPU E5-2680 v3 @ 2.50GHz (32 cores)
Memory	120GB RAM
Disk	500GB 7200RPM HDD
OS	centos-release-7-4.1708.el7.centos.x86_64

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.

To monitor the resource usage per-container, we also ran a dockerized monitoring infrastructure based on Prometheus[3], Cadvisor[4], Grafana[5] and Node Exporter[6].

Petstore and Sockshop. To benchmark PetStore and Sockshop, each service was deployed as a Docker container on a single, powerful machine that has the following specifications:

CPU	Intel Xeon CPU E5-2680 v3 @ 2.50GHz (16 cores)
Memory	61GB RAM
Disk	500GB 7200RPM HDD
OS	centos-release-7-4.1708.el7.centos.x86_64

To inject data into the systems 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.

While an expected throughput of 200 client actions per second might seem low, it is important to consider that we are dealing with reference projects, that were created to display the capabilities of the chosen web frameworks. Naturally these projects were not designed for performance benchmarking, and as the load increases, the request latency also increases exponentially. We chose the maximum injector rate that yields stable results, as we believe an

[3]https://prometheus.io/
[4]https://github.com/google/cadvisor
[5]Grafana
[6]https://github.com/prometheus/node_exporter
[7]https://locust.io/

over-worked system does not provide an accurate comparison between traced and non-traced executions. We ran each benchmark for one hour and discarded the first 15% of the requests.

5.2.4.4 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. The latency distribution curves, plotted in Figure 5.1, show significant overlap between the three runs across the whole percentile spectrum, bolstering our claim of comparable performance.

This not to say that there was no increase, in fact, Table 5.7 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 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 5.2 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 5.3) 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.

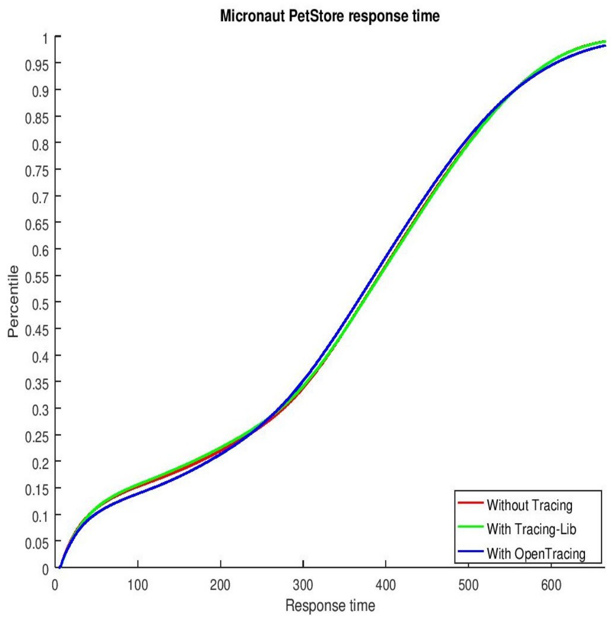


Figure 5.1: **Cumulative latency distribution function of the Petstore project**

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 5.7: **Latency distribution of Petstore**

Metric	None	Melange	Op'Tracing
Tput (t/s)	557.0	557.7	555.6
Min (ms)	5.0	5.1	4.9
Max (ms)	1538.1	1619.3	1522.7
Mean (ms)	345.0	344.5	345.7
Disk (GB)	–	5.6	5.4

Table 5.8: **Additional metrics for Petstore**

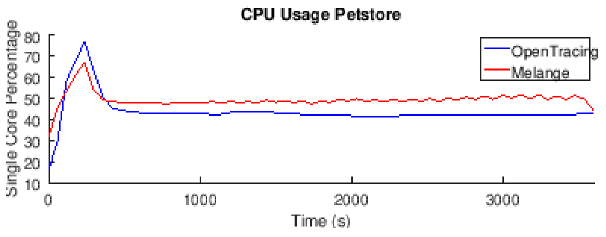


Figure 5.2: **CPU Usage of Petstore**

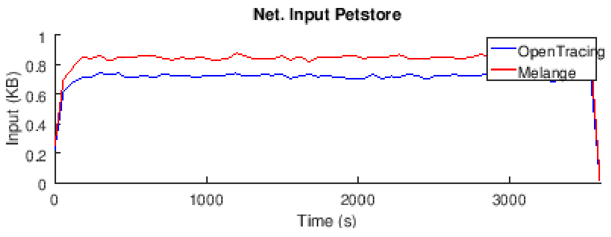


Figure 5.4: **Net Input of Petstore**

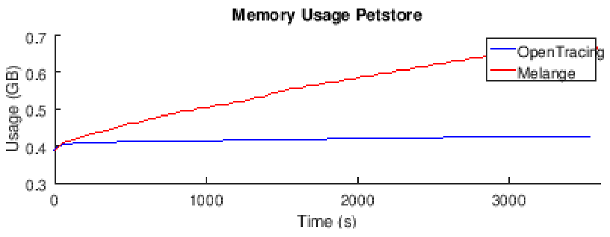


Figure 5.3: **Memory Usage of Petstore**

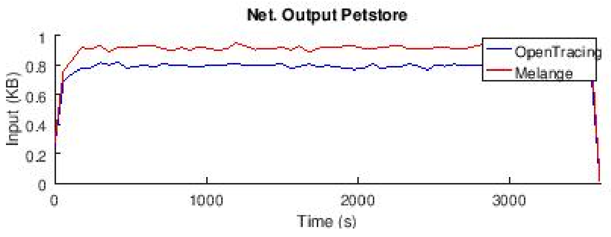


Figure 5.5: **Net Output of Petstore**

Sock-Shop. Comparably, Sock-Shop shows little tracing overhead, with the CDF in Figure 5.6 showing similar overlap and reinforcing our previous statements in the Petstore analysis.

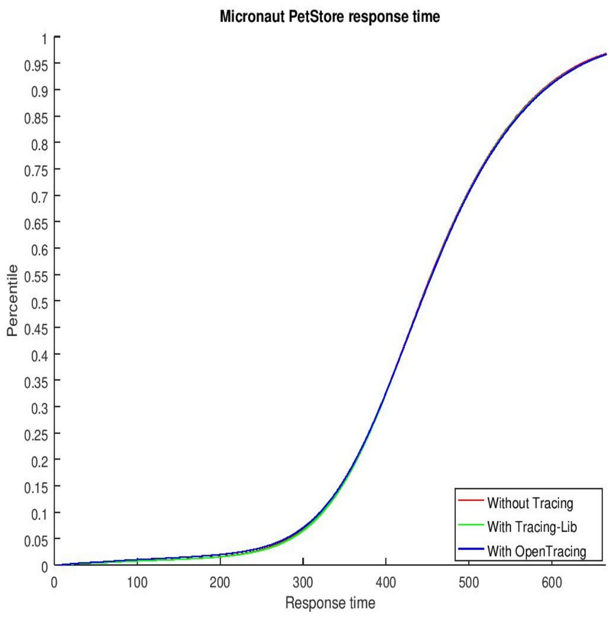


Figure 5.6: **Cumulative latency distribution function of the Sock-Shop project latencies**

Percentile	None	Melange	Op'Tracing
50.000	442.2	442.8	443.0
99.000	736.5	743.5	743.0
99.900	870.0	883.3	951.3
99.990	2903.7	2781.0	4490.0
99.999	4610.5	4433.9	5390.4

Table 5.9: **Latency distribution of Sock-Shop**

Metric	None	Melange	Op'Tracing
Tput (t/s)	638.0	636.8	636.2
Min (ms)	4.5	4.5	4.6
Max (ms)	4917.9	5373.3	5654.3
Mean (ms)	448.5	449.6	450.1
Disk (GB)	–	3.2	3

Table 5.10: **Additional metrics for Sock-Shop**

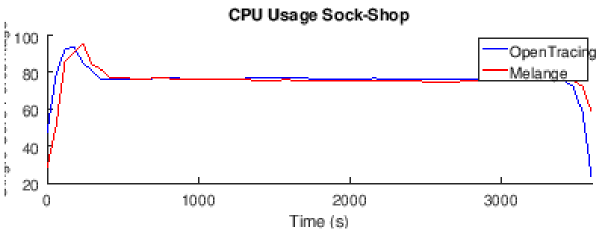


Figure 5.7: **CPU Usage of Sock-Shop**

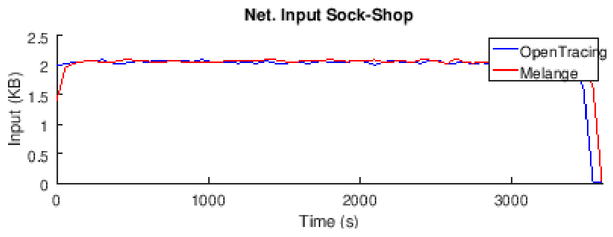


Figure 5.9: **Net Input of Sock-Shop**

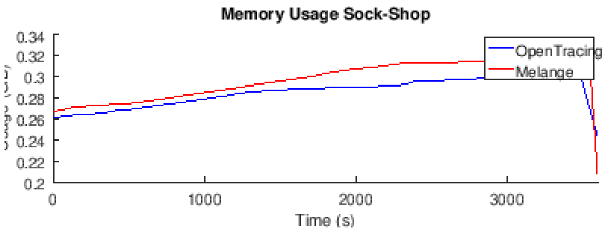


Figure 5.8: **Memory Usage of Sock-Shop**

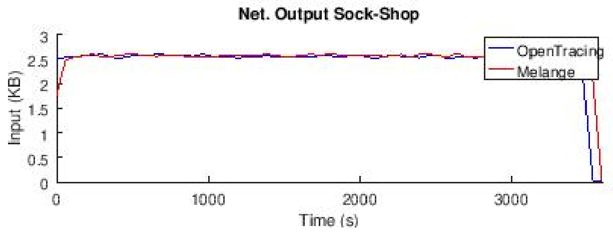


Figure 5.10: **Net Output of Sock-Shop**

Once again, while there is some variation between benchmark results for each framework, the median overhead remains rather low (between 1 and 2%), with the 99th percentile showing a 5% increase for either framework. As we move up to the smaller percentiles, the experimental noise between runs begins to impact the results.

Unlike Petstore, the monitoring data for Sock-Shop shows similar CPU Usage when tracing with OpenTracing and Melange, which can be explained by simpler instrumentation code that makes less use of Melange's trace reconstruction features. Melange's Memory Usage remains higher with a 12.5% increase in total usage. The network usage is very similar between Melange and OpenTracing, which is to be expected, as our work defers the span propagation functionality to OpenTracing. Increases in network usage would only be caused by additional meta-data in the spans such as tags or baggage.

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

Unlike the previous projects, 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 [31]. 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.

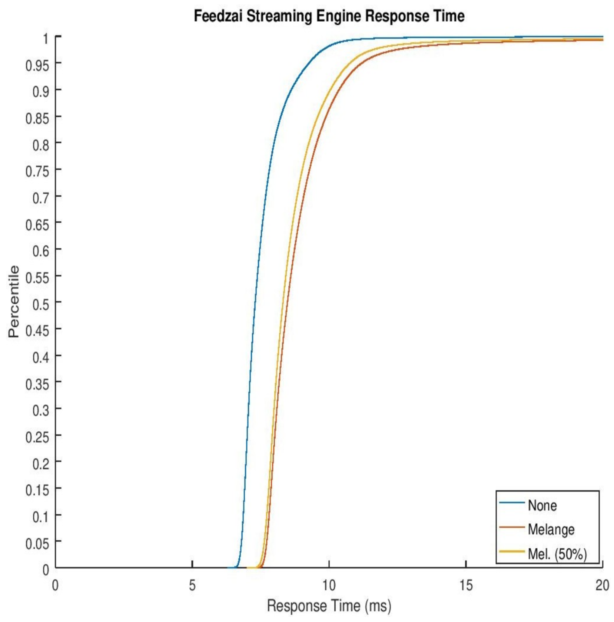


Figure 5.11: **Cumulative latency distribution function of FSE**

Percentile	None[a]	Melange	Mel. (50%)
50.000	6.8	8.3	8.1
99.000	9.8	18.1	16.7
99.900	17,5	62.8	58.3
99.990	61.5	87.2	85.1
99.999	116.7	130.2	135.8

Table 5.11: **Latency distribution of FSE**

Metric	None	Melange	Mel. (50%)
Tput (t/s)	297.3	295.2	295.8
Min (ms)	5.8	6.8	6.5
Max (ms)	116.7	144.4	176.7
Mean (ms)	7.1	8.9	8.8
Disk (GB)	–	11	6.3

Table 5.12: **Additional metrics for FSE**

[a]The data presented here is not indicative of Feedzai's performance in production environments and is merely a baseline for comparison of different tracing frameworks

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. This effect becomes clear when we configure the Streaming Engine to make requests slightly slower, as evidenced in Table 5.13, where the overhead shrinks to 16% in the median and 12% in the maximum.

Percentile	None	Melange	Mel. (50%)
50.000	7.3	8.4	8.3
99.000	10.4	16.9	14.1
99.900	19.2	71.9	63.5
99.990	74.7	112.5	100.4
99.999	123.8	138.6	134.1

Table 5.13: **Latency distribution of FSE for slower requests**

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.

Percentile	None	OpenTracing Noop
50.000	6.8	7.6
99.000	9.7	12.1
99.900	17.4	23.6
99.990	61.5	58.2
99.999	68.1	89.5

Table 5.14: **Latency distribution of FSE with no-op OpenTracing implementation**

Percentile	100%	50%	30%	10%	No Tracing
50	8.3	8.2	8.5	8.2	6.8
99	18.1	14.8	13.7	12.5	9.7
99.9	62.8	56.6	47.4	30.5	17.4
99.99	87.2	82.9	75.8	73.0	61.5
99.999	130.2	137.0	88.3	99.7	68.1

Table 5.15: **Latency distribution of FSE with no-op OpenTracing implementation**

Table 5.14 displays the results of the aforementioned test. 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. In an attempt to reduce overhead while regaining tracing functionality, we added simple sampling functionality to Melange, leading to the results in Table 5.15. Clearly there is a reduction in latency in tandem with lower sampling rates. While the relationship between the two isn't linear, further work could improve the performance of this feature.

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 5.12. 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 5.13) is also larger, due to Melange's in-memory caches used for cross-thread tracing, which can be severely reduced with a more aggressive cache eviction policy and lower sampling rate. Interestingly, when plotting the monitoring data for a single container, the Network Output graph in Figure 5.14 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 5.14.

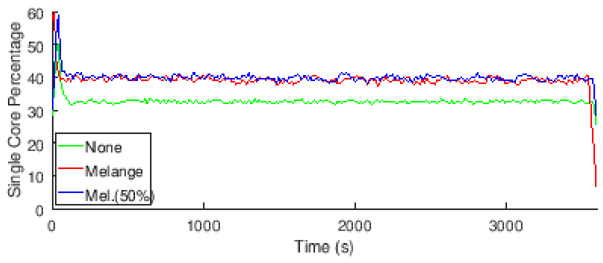


Figure 5.12: **CPU Usage of FSE**

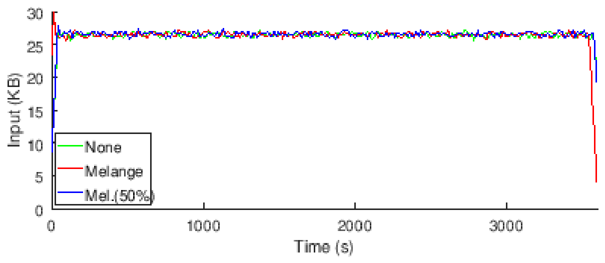


Figure 5.14: **Net Input of FSE**

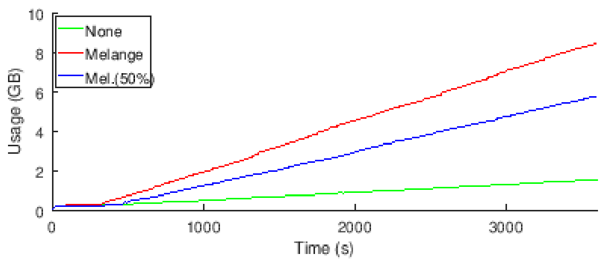


Figure 5.13: **Memory Usage of FSE**

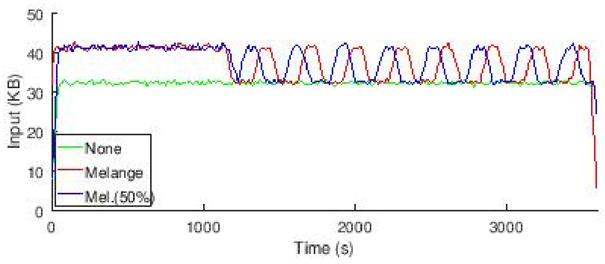


Figure 5.15: **Net Output of FSE**

5.2.5 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, OpenTracing (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.

5.3 Melange in Practice

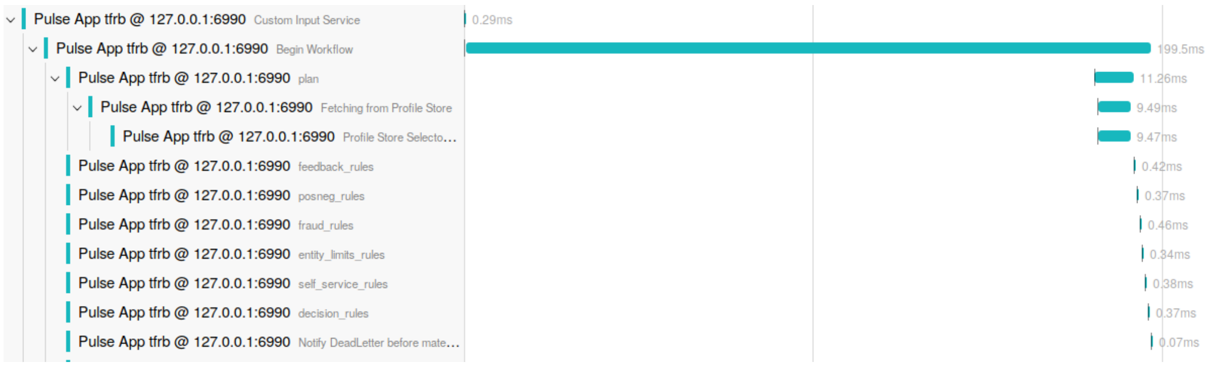
The end-goal of tracing distributed systems is usually to improve the developer's ability to debug any issue that might arise, whether those issues are structural faults or performance problems. Below are some examples of how we used Melange in a performance engineering scenario, where we were able to leverage the new tracing capabilities to fine-tune its own testing environment.

5.3.1 Profiling the Benchmarking Environment

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.

5.3.1.1 Garbage Collection in the Streaming Engine

The Streaming Engine's traces pointed not to a single common source of overhead, but to one of two problems. All traces either showed extremely long single spans that took up most of the duration (depicted in Figure 5.16), or equally long gaps between consecutive processing steps in the streaming engine.



```
Pulse App tfrb @ 127.0.0.1:6990 Custom Input Service                  0.29ms
  Pulse App tfrb @ 127.0.0.1:6990 Begin Workflow                      199.5ms
    Pulse App tfrb @ 127.0.0.1:6990 plan                              11.26ms
      Pulse App tfrb @ 127.0.0.1:6990 Fetching from Profile Store     9.49ms
        Pulse App tfrb @ 127.0.0.1:6990 Profile Store Selecto...      9.47ms
    Pulse App tfrb @ 127.0.0.1:6990 feedback_rules                    0.42ms
    Pulse App tfrb @ 127.0.0.1:6990 posneg_rules                      0.37ms
    Pulse App tfrb @ 127.0.0.1:6990 fraud_rules                       0.46ms
    Pulse App tfrb @ 127.0.0.1:6990 entity_limits_rules               0.34ms
    Pulse App tfrb @ 127.0.0.1:6990 self_service_rules                0.38ms
    Pulse App tfrb @ 127.0.0.1:6990 decision_rules                    0.37ms
    Pulse App tfrb @ 127.0.0.1:6990 Notify DeadLetter before mate...  0.07ms
```

Figure 5.16: **Long spans in FSE**

The gaps between streaming pipeline stages pointed towards excessive queuing happening due to lack of resources (requests can be queued between pipeline steps while waiting for threads to be freed), yet the monitoring dashboards showed low resource usage which contradicted our assumption.

Upon further analysis, we hypothesized that the long spans and long gaps could be two symptoms of the same illness: Garbage Collection. Pauses that occur during the execution of a traced method result in long spans or, if they happen between pipeline steps, a long break. Finally, after the size of Java's available heap space was raised the throughput increased while the latencies lowered across the full percentile spectrum.

5.3.1.2 Garbage Collection in the Event Monitor

After the request latency problem was resolved, the attention moved to throughput, which was being affected by the Event Monitor's performance. When a request is processed by the Streaming Engine it is sent to a RabbitMQ queue that eventually leads to the Event Monitor. At the time, the queue was quickly becoming full as the Event Monitor was not able to deplete the queue at a fast enough rate, which in turn increased the backpressure when FSE was inserting.

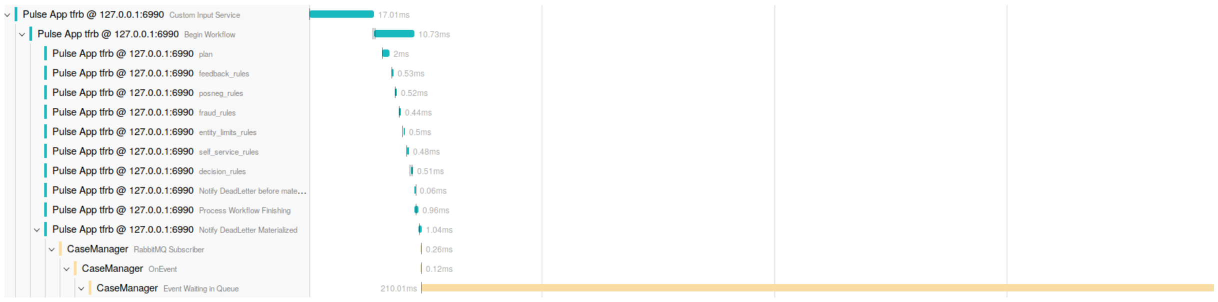

Pulse App tfrb @ 127.0.0.1:6990 Custom Input Service 17.01ms
Pulse App tfrb @ 127.0.0.1:6990 Begin Workflow 10.73ms
Pulse App tfrb @ 127.0.0.1:6990 plan 2ms
Pulse App tfrb @ 127.0.0.1:6990 feedback_rules 0.53ms
Pulse App tfrb @ 127.0.0.1:6990 posneg_rules 0.52ms
Pulse App tfrb @ 127.0.0.1:6990 fraud_rules 0.44ms
Pulse App tfrb @ 127.0.0.1:6990 entity_limits_rules 0.5ms
Pulse App tfrb @ 127.0.0.1:6990 self_service_rules 0.48ms
Pulse App tfrb @ 127.0.0.1:6990 decision_rules 0.51ms
Pulse App tfrb @ 127.0.0.1:6990 Notify DeadLetter before mate… 0.06ms
Pulse App tfrb @ 127.0.0.1:6990 Process Workflow Finishing 0.96ms
Pulse App tfrb @ 127.0.0.1:6990 Notify DeadLetter Materialized 1.04ms
CaseManager RabbitMQ Subscriber 0.26ms
CaseManager OnEvent 0.12ms
CaseManager Event Waiting in Queue 210.01ms

Figure 5.17: **Requests queued in Event Monitor**

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 EM, as illustrated in Figure 5.17. 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.

5.3.1.3 Unnecessary Database Fetches

After studying the monitoring data, we were surprised by the amount of CPU being used by the Profile Storage – 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 5.18).

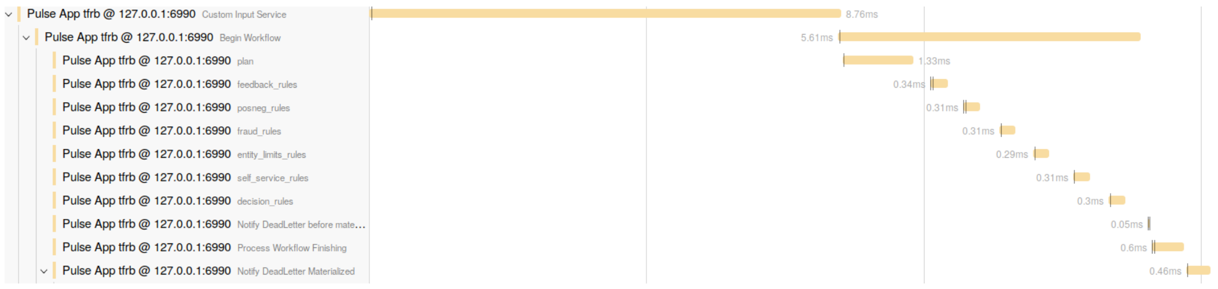

Pulse App tfrb @ 127.0.0.1:6990 Custom Input Service 8.76ms
Pulse App tfrb @ 127.0.0.1:6990 Begin Workflow 5.61ms
Pulse App tfrb @ 127.0.0.1:6990 plan 1.33ms
Pulse App tfrb @ 127.0.0.1:6990 feedback_rules 0.34ms
Pulse App tfrb @ 127.0.0.1:6990 posneg_rules 0.31ms
Pulse App tfrb @ 127.0.0.1:6990 fraud_rules 0.31ms
Pulse App tfrb @ 127.0.0.1:6990 entity_limits_rules 0.29ms
Pulse App tfrb @ 127.0.0.1:6990 self_service_rules 0.31ms
Pulse App tfrb @ 127.0.0.1:6990 decision_rules 0.3ms
Pulse App tfrb @ 127.0.0.1:6990 Notify DeadLetter before mate… 0.05ms
Pulse App tfrb @ 127.0.0.1:6990 Process Workflow Finishing 0.6ms
Pulse App tfrb @ 127.0.0.1:6990 Notify DeadLetter Materialized 0.46ms

Figure 5.18: **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 Profile Storage.

5.3.2 Remarks on the practicality of Melange

As it was made clear by our previous examples, performance engineering and root-cause analysis becomes much simpler in tracing-enabled systems. With Melange, identifying and understanding the source of performance issues in the testing environment was extremely straightforward as the sheer amount of causally-related data collected, and visually represented, allowed us to immediately detect patterns of irregular behavior. The addition of new functionality, like the ability to capture the time an object spent in a queue made itself useful from the start, as elaborated in Section 5.3.1.2, proving that our work further advances the practical benefits of distributed tracing in real-world scenarios.

While Melange is at too early a stage to make this impact visible in Feedzai's production environments as this thesis is being written, upon reviewing the execution logs of previous benchmarking and performance engineering efforts for Feedzai's clients, we discovered 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, severely shortening the performance-engineering feedback loop.

6 Conclusion

6.1 Concluding Remarks

Our work began by examining the difficulties experienced when tracing highly asynchronous and heterogeneous distributed systems, and showing a need for an instrumentation platform that could combine white-box and black-box tracing techniques.

We continued by reviewing the current state-of-the art in distributed tracing, from which we were able to extract a novel framework that can catalog each work. Our taxonomy describes each system as a set of layers required for tracing distributed systems (ranging from meta-data extraction, to trace analysis) and for each layer describes the most common design decisions, as well as their strengths and shortcomings.

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, while showing itself useful for performance engineering.

6.2 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. The next step in this regard would be to create a compatibility layer that could process data from the user's monitoring backends and generate dashboards based on the selected span's metadata.

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.

Bibliography

[1] Marcos K. Aguilera et al. “Performance Debugging for Distributed Systems of Black Boxes”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: ACM, 2003. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945454.

[2] Paul Barham et al. “Using Magpie for Request Extraction and Workload Modelling”. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004. URL: http://dl.acm.org/citation.cfm?id=1251254.1251272.

[3] Han Bok Lee and Benjamin Zorn. “BIT: A Tool for Instrumenting Java Bytecodes”. In: *USENIX Symposium on Internet Technologies and Systems* (Dec. 1997).

[4] Brendan Burns et al. “Borg, Omega, and Kubernetes”. In: *ACM Queue* 14 (2016). URL: http://queue.acm.org/detail.cfm?id=2898444.

[5] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. “Dynamic Instrumentation of Production Systems”. In: *USENIX Annual Technical Conference, General Track*. 2004.

[6] Robert Carosi. “Protractor: Leveraging Distributed Tracing in Service Meshes for Application Profiling at Scale.” carosi_protractor:_2018. KTH, School of Electrical Engineering and Computer Science (EECS), 2018.

[7] M.Y. Chen et al. “Pinpoint: Problem Determination in Large, Dynamic Internet Services”. In: *Proceedings International Conference on Dependable Systems and Networks*. International Conference on Dependable Systems and Networks. Washington, DC, USA: IEEE Comput. Soc, 2002. ISBN: 978-0-7695-1597-7. DOI: 10.1109/DSN.2002.1029005.

[8] Mike Y. Chen et al. “Path-Based Faliure and Evolution Management”. In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*. NSDI’04. San Francisco, California: USENIX Association, 2004. URL: http://dl.acm.org/citation.cfm?id=1251175.1251198.

[9] MIchael Chow et al. “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: USENIX Association, 2014. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow.

[10] Byung-Gon Chun et al. *D3: Declarative Distributed Debugging*. University of California at Berkeley.

[11] Mariano P. Consens and Alberto O. Mendelzon. “Hy+: A Hygraph-Based Query and Visualization System”. In: *SIGMOD Conference*. 1993.

[12] Mariano Consens, Masum Hasan, and Alberto Mendelzon. “Debugging Distributed Programs by Visualizing and Querying Event Traces”. In:

[13] Úlfar Erlingsson et al. “Fay: Extensible Distributed Tracing from Kernels to Clusters”. In: *ACM Transactions on Computer Systems* 30.4 (Nov. 1, 2012). ISSN: 07342071. DOI: 10.1145/2382553.2382555.

[14] Rodrigo Fonseca et al. “X-Trace: A Pervasive Network Tracing Framework”. In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, 2007. URL: https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework.

[15] Graeme Jenkinson. “Distributed DTrace”. In: *Dtrace.Conf*. 2008. URL: https://vimeo.com/173346403 (visited on 01/02/2019).

[16] Gregor Kiczales and Erik Hilsdale. “Aspect-Oriented Programming”. In: *ESEC / SIGSOFT FSE*. 2001.

[17] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (July 1, 1978). ISSN: 00010782. DOI: 10.1145/359545.359563.

[18] Jonathan Leavitt. *End-to-End Tracing Models: Analysis and Unification*. Brown University.

[19] Wubin Li et al. “Service Mesh: Challenges, State of the Art, and Future Research Opportunities”. In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019 IEEE International Conference on Service-Oriented System Engineering

(SOSE). San Francisco East Bay, CA, USA: IEEE, Apr. 2019. ISBN: 978-1-72811-442-2. DOI: 10.1109/SOSE.2019.00026.

[20] Haifeng Liu et al. “JCallGraph: Tracing Microservices in Very Large Scale Container Cloud Platforms”. In: *Cloud Computing – CLOUD 2019*. Ed. by Dilma Da Silva, Qingyang Wang, and Liang-Jie Zhang. Vol. 11513. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-23501-7 978-3-030-23502-4. DOI: 10.1007/978-3-030-23502-4_20. URL: http://link.springer.com/10.1007/978-3-030-23502-4_20 (visited on 10/25/2019).

[21] Jonathan Mace and Jonathan Kaldor. “Canopy: An End-to-End Performance Tracing And Analysis System”. In: Symposium on Operating Systems Principles. Oct. 28, 2018.

[22] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/mace.

[23] Jonathan Mace et al. “Retro: Targeted Resource Management in Multi-Tenant Distributed Systems”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. ISBN: 978-1-931971-21-8. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace.

[24] Robert C Martin. “Design Principles and Design Patterns”. In: (2000).

[25] Alexander V. Mirgorodskiy and Barton P. Miller. “Diagnosing Distributed Systems with Self-Propelled Instrumentation”. In: *Middleware 2008*. Ed. by Valérie Issarny and Richard Schantz. Vol. 5346. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-89855-9 978-3-540-89856-6. DOI: 10.1007/978-3-540-89856-6\\_5. URL: http://link.springer.com/10.1007/978-3-540-89856-6_5 (visited on 10/24/2018).

[26] V Prasad et al. “Locating System Problems Using Dynamic Instrumentation”. In: (Jan. 2005).

[27] Patrick Reynolds et al. “Pip: Detecting the Unexpected in Distributed Systems.” In: Symposium on Networked Systems Design and Implementation. Jan. 2006.

[28] Raja R. Sambasivan et al. “Principled Workflow-Centric Tracing of Distributed Systems”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC ’16*. The Seventh ACM Symposium. Santa Clara, CA, USA: ACM Press, 2016. ISBN: 978-1-4503-4525-5. DOI: 10.1145/2987550.2987568.

[29] Raja R Sambasivan et al. “Diagnosing Performance Changes by Comparing Request Flows.” In: USENIX Symposium on Networked Systems Design and Implementation. 2011.

[30] H. Scholten and J. Posthuma. “A Debugging Tool for Distributed Systems”. In: *Proceedings of TENCON ’93. IEEE Region 10 International Conference on Computers, Communications and Automation*. TENCON ’93. IEEE Region 10 International Conference on Computers, Communications and Automation. Beijing, China: IEEE, 1993. ISBN: 978-0-7803-1233-3. DOI: 10.1109/TENCON.1993.319956.

[31] Benjamin H. Sigelman et al. “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure”. In: (2010). URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[32] Richard Sites. “Data Center Computers: Modern Challenges in CPU Design”. Feb. 23, 2015. URL: https://cs.unc.edu/50th/alumni-speaker-series/dick-sites/.

[33] Justin Thiel. *An Overview of Software Performance Analysis Tools and Techniques: From GProf to DTrace*.

[34] Steve Vinoski. “Advanced Message Queuing Protocol”. In: *IEEE Internet Computing* 10.6 (Nov. 2006). ISSN: 1089-7801. DOI: 10.1109/MIC.2006.116.

[35] Michael Whittaker et al. “Debugging Distributed Systems with Why-Across-Time Provenance”. In: *Proceedings of the ACM Symposium on Cloud Computing - SoCC ’18*. The ACM Symposium. Carlsbad, CA, USA: ACM Press, 2018. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267839.

[36] Paul Wright. “What Etsy Learned Building a Distributed Tracing System”. In: *Surge: The Scalability & Performance Conference*. omniTI Surge. 2014. URL: https://www.youtube.com/watch?v=lKT2FOth4to (visited on 12/30/2018).

[37] Chun Yuan et al. “Automated Known Problem Diagnosis with Event Traces”. In: *ACM SIGOPS Operating Systems Review* 40.4 (Oct. 1, 2006). ISSN: 01635980. DOI: 10.1145/1218063.1217972.

[38] Zhonghua Yang and T.A. Marsland. “Global Snapshots for Distributed Debugging”. In: *Proceedings ICCI ‘92: Fourth International Conference on Computing and Information*. ICCI ‘92: Fourth International Conference on Computing and Information. Toronto, Ont., Canada: IEEE Comput. Soc. Press, 1992. ISBN: 978-0-8186-2812-2. DOI: 10.1109/ICCI.1992.227618.

I
Appendices

A API Reference

```java
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 A.1: **BaseTracing API definition**

lst:tracing

```java
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 A.2: **TracingWithId API definition**

```java
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 A.3: **TracingWithContext API**