Alternative Visualisations of Distributed Tracing data in a complex, large-scale distributed system

Noah Santschi-Cooney

Final Year Project - BSc in Computer Science

Dr. Jason Quinlan

Department of Computer Science
University College Cork

3rd April 2020

Abstract

Modern Internet services are often implemented as complex, large-scale distributed systems. These applications are constructed from collections of software modules that could span many thousands of machines across multiple physical facilities. With the rise of modern Micro-service and Service-Oriented designs, traditional tooling used to monitor application behaviour is no longer viable, especially at scale.

To understanding the flow and life cycle of a unit of work performed in multiple pieces across various components in a distributed system, the concept of Distributed Tracing was born. Distributed Tracing was first introduced to the mainstream world in 2010 after the publication of Google's Dapper paper. Since then, various vendors have come out with their own Dapper-inspired services, most of them based off flame or timeline graphs.

The goal of this project is dual-faceted:

- Explore and research possible alternative uses and visualisation methods utilising data collected from distributed tracing clients.

- Implement one or more of the proposed alternatives.

Declaration of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award. I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: ....

Date: ....

Acknowledgements

Contents

Abstract **i**

Declaration of Originality **ii**

Acknowledgements **iii**

1 Introduction **1**
- 1.1 Problem 1
- 1.2 Debuggers 2
- 1.3 Distributed Tracing 3
- 1.4 Motivation & Goals 4
- 1.5 Project Summary 5

2 Background **6**
- 2.1 History 6
  - 2.1.1 Dapper 6
  - 2.1.2 OpenTracing 7
  - 2.1.3 OpenTelemetry 9

3 Design & Implementation **11**
- 3.1 Architecture Design 11
  - 3.1.1 Distributed Tracing API 11
  - 3.1.2 Supporting Services 11
  - 3.1.3 Backend API 12
  - 3.1.4 Debug Adapter 15
  - 3.1.5 Service Topology Frontend 20
- 3.2 Implementations 24
  - 3.2.1 Backend API 24

3.2.2 Debug Adapter 24
3.2.3 Service Topology Frontend 24

4 Evaluation **25**

5 Conclusion & Future **26**
5.1 Conclusion 26
5.2 Future Work 26
5.2.1 OpenTelemetry 26

Bibliography **27**

1. Introduction

1.1 Problem

Within the last decade, the way modern applications are being built and deployed has changed dramatically. With the shift from collocation to cloud computing, virtual machines to containerization technologies, monoliths to micro-services and beyond, software developers have been able to adjust to the monotonical increase in internet traffic, shipping highly scalable, efficient and reliable software that meets the ever-demanding needs of their customers with the slew of emerging technologies.

While this shift has undoubtedly solved many issues with regards to scaling services in terms of both maintainability as feature sets increase and in keeping up with an every larger number of online users, it has introduced a whole new suite of problems that needed to be addressed in terms of reliability and application monitoring. With the splitting of monolithic applications into micro-services, the failure points are extended to issues in the network, including but not limited to network congestion, DNS resolution errors etc. Developers are ever more inclined to code failure resilience into their applications, falling back gracefully in apprehension of unforeseeable failures.

As these new distributed system architectures evolved and became ever more widespread, traditional application monitoring tools consistently fell short of providing developers and systems operators with the means to gain introspection into systems and their failures in production scenarios. Traditional monolithic systems often utilized logging and metrics to gain introspection into the application and for alerting on rules respectively. For such systems, these process-scoped measures often provided good insight into a system, correlating logs on their thread identifier/name as each thread would handle a single request sequentially. As these systems adopted asynchronous execution models, where a request's lifetime may not be confined to a single thread, the previous approach no longer works, making observing the behaviour of such systems very difficult unless developers annotated logs with request-scoped identifiers. The final evolution of concurrency in application systems is commonly referred to as *distributed concurrency*. This is often associated with micro-services, in which a request is no longer constrained to being executed in a single process, but may span multiple processes and even servers. Figure 1.1 highlights this evolution, from simple, single threaded applications, through to micro-service-like architectures.

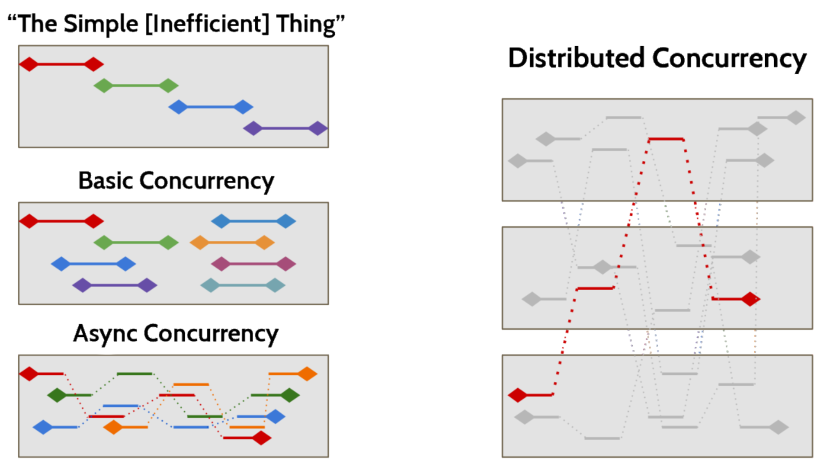

Figure 1.1: Evolution of concurrent systems.

1.2 Debuggers

In traditional single process applications, debugger tools, both standalone and bundled with integrated development environments (IDEs), are invaluable in their use of isolating bugs in codebases of any size. They have the capability to give complete overview of stack and heap allocated variables as well as being able to set breakpoints to step through code. Figure 1.2 highlights the various insights and utilities provided by such tools, including the display of call stacks, local and global variables as well as various utilities to step through code at the line and function levels.

However, it is often infeasible to use them in production scenarios due to their nature of halting complete execution of the process. This makes it unsuitable for debugging issues that manifest in production that developers are finding it difficult to reproduce in development scenarios, as is often a

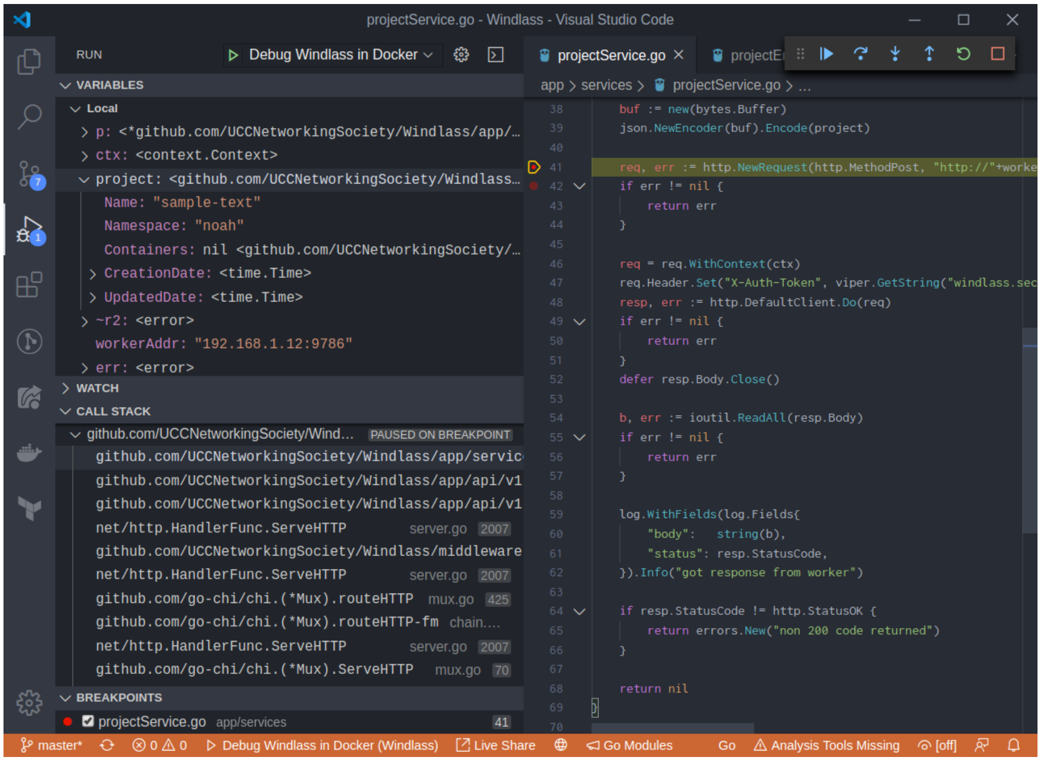

Figure 1.2: Screenshot of the Visual Studio Code debugger in action. Clockwise, shown are an expandable list of local and global variables, the currently open file view with the line currently halted on highlighted along with controls for stepping and finally the function call stack.

common scenario due to subtle parity differences between development and production systems.

1.3 Distributed Tracing

As traditional tooling is not designed to accommodate for this distributed concurrency system, new methodologies were needed to regain observability into the systems. Observing single systems individually, as was done with traditional tooling, no longer painted the full picture of a request as it travels through multiple system components. Distributed tracing systems and platforms build upon the concepts of reconstructing a request from a series of event streams from each component involved in the request, with distributed context propagation and aggregation, building causality graphs from a request-centric point of view.

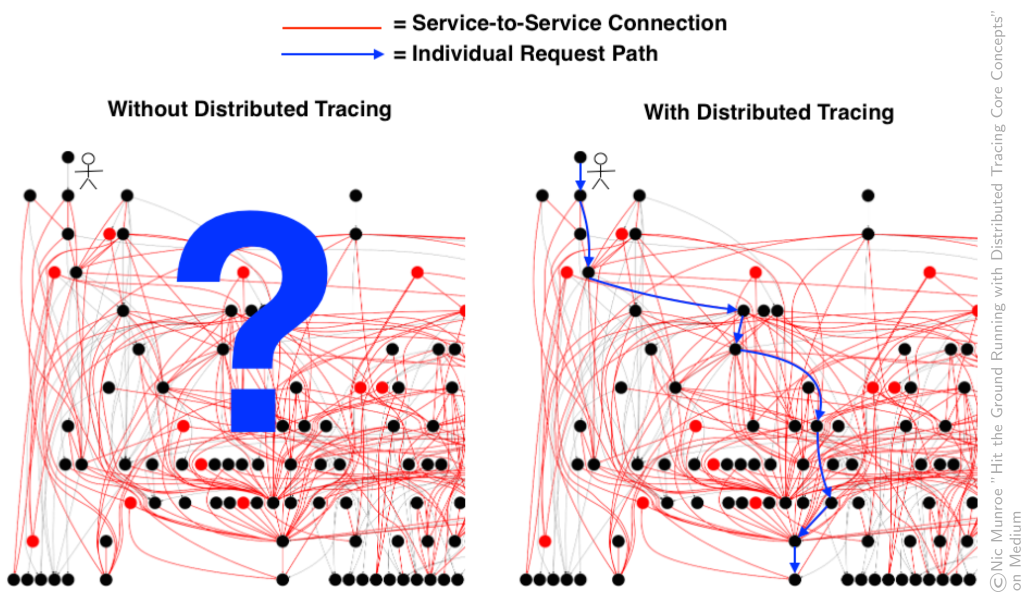

Figure 1.3: Depiction of a complex distributed system with much inter-connectivity between services.

Code is instrumented at various points of interest, recording annotated events with metadata such as the user ID associated with the request, SQL statements being executed on a database etc. These events are often shipped to a collector/exporter, from which they are either stored in a database or sent to a hosted vendor, such as LightStep or Honeycomb, after which they can be queried, retrieved and displayed.

1.4 Motivation & Goals

As distributed tracing is still a relatively new idea and only as of recently gathering mainstream interest in the industry, research and advancements on the topic are as of yet still sparse. Current vendors often provide a limit set of capabilities and operations that can be performed on the data output from instrumented distributed systems, most commonly simple expandable *gantt charts* or, less commonly, simple, mostly static, service dependency graphs that offer little value and utility.

To further research in this field, the project will attempt to explore alternative and hopefully improved ways of consuming and presenting the data from instrumented applications. Two ideas were planned to be explored and, if possible, implemented as proof of concepts:

- Advancements in Service Topology/Dependency graphs
- Editor Debugger integration

The viability and findings of both explored options will be discussed, with performance benchmarks where relevant being presented to highlight the feasibility of different approaches

1.5 Project Summary

This project builds upon the concepts of distributed tracing, exploring ways to provide novel and high-value derivable ways of visualizing and presenting distributed tracing data to developers. Modern standards, tools and integrations will be utilized to test the viability of less common and unexplored visualisations of distributed tracing data.

In Chapter 2, the history of distributed tracing will be introduced, while also covering some common vocabulary relevant to the topic and where they originated. It will also cover some of the standards that this project builds around. In Chapter 3, the project architecture design choices will be discussed and how they impacted the project, ranging from the frontend frameworks chosen to the backend API and supporting services that power the various implementations. Finally, the different visualisations will be evaluated on the value the provide as well as the feasibility of utilizing them in real-world scenarios. Chapters 5 and 6 will draw the writeup to a conclusion, detailing the closing thoughts and putting forward ideas for future work on the ideas explored in this project.

2. Background

2.1 History

2.1.1 Dapper

Released in April 2010, Google published a paper describing the design decisions behind an in-house implementation of distributed tracing, named Dapper. It is commonly believed that this paper describes the common ancestor to many tools that implement a form of distributed tracing.

The Dapper paper introduces some of the core primitives that underpin modern day standards. Most notable are the concepts of a directed acyclic graph (DAG) called a *trace tree* and its nodes, which are referred to as *spans*. The trace tree forms a relationship between spans, not unakin to a tree of stack frames that may be generated by gathering stack frames over time, albeit generally at a much higher level than at the level of individual subroutine calls.

Figure 2.1 illustrates a trace tree with five spans. Each span is shown to contain 3 specific pieces of metadata alongside the start and end timestamps necessarily to reconstruct the temporal relationships: a human-readable *span name*, an integer *span ID* and an integer *parent ID*. The latter two data points are used to reconstruct the relationship between individual spans. A span without a parent ID becomes the *root span* of a trace tree. Not shown is another important but, as of right now, not relevant piece of metadata, the *trace ID*, which is common amongst all spans within a single trace tree.

As described thus far, Dapper trace trees allow for a detailed view of the relationships of distributed systems within Google. When using this data for debugging or performance analysis, it can often be convenient or even necessary to have additional context surrounding a trace tree or its individual spans. As such, the paper describes a simple API through which application developers can provide a combination of two types of annotations: timestamped textual annotations and key-value, allowing for defining arbitrary equivalence classes between traces which can be operated upon in the analysis tools.

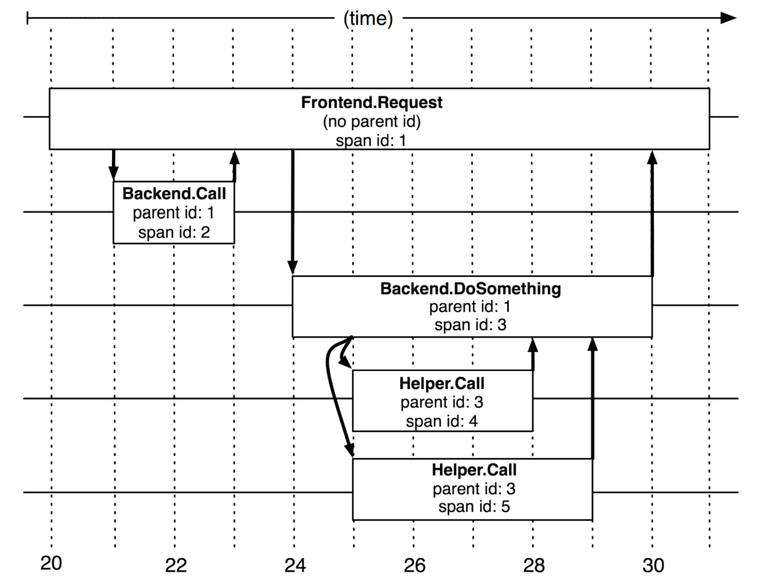

Figure 2.1: The relationships between traces in a trace tree.

2.1.2 OpenTracing

OpenTracing[3] project's inception came about in October 2015, it has since become a project under the Cloud Native Computing Foundation in 2016, created to standardize a set of vendor neutral and programming language agnostic application programming interfaces (APIs) for instrumenting code for distributed tracing. Heavily inspired by the Dapper paper, it borrows many of the nouns and verbs outlined in the Dapper paper, including *traces* and *spans*. Dapper's timestamped annotations are referred to as *logs* in the OpenTracing specification, while the key-value pairs are named *tags*.

The OpenTracing API also specifies how a trace cross process boundaries, so that spans created in different processes can be associated with a common trace tree. This was named the *span context* and at its most basic level contains the overlying trace ID as well as the current span ID. With this, new spans generated across process boundaries have the ability to to specify their parent span as well as their common trace, without propagating an entire span, which may prove costly as more tags and logs are attached to a span.

Figure 2.2 shows a timeline based visualisation of where the different components of the OpenTracing API interface are utilized in the larger picture of creating a span through use of distributed context propagation in the span context construct to build the span tree across process and network boundaries.

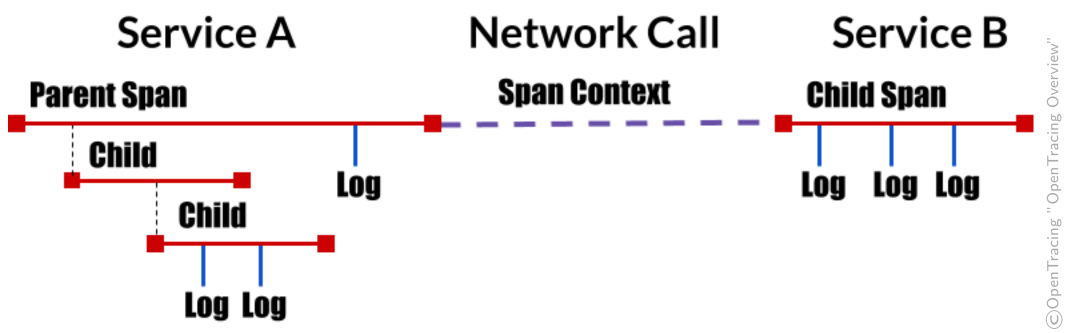

Figure 2.2: Infographic visualizing the different components that make up the OpenTracing API interface and how they relate to different services and the network

As there are multiple output sinks which can consume OpenTracing data, from self hosting services such as Jaeger to hosted vendors like LightStep, and given that different platforms may have different, vendor-specific options for operations such as access control, authorization etc, vendors provide different mechanisms and attributes for creating instances of OpenTracing API *tracers* implementations.

Listing 2.1: Example Golang snippet of instatiating a Datadog OpenTracing compatible tracer.

```go
package main

import (
    "gopkg.in/DataDog/dd-trace-go.v1/ddtrace"
    "gopkg.in/DataDog/dd-trace-go.v1/ddtrace/opentracer"
    "github.com/opentracing/opentracing-go"
)

// Start a Datadog tracer, optionally providing a set of options,
// returning an opentracing.Tracer which wraps it.
t := opentracer.New(
    tracer.WithAgentAddr("host:port"),
    tracer.WithServiceName("sample-text"))

    // Use it with the OpenTracing API, setting it as global.
opentracing.SetGlobalTracer(t)
```

Listing 2.2: Example Golang snippet of instatiating a Jaeger OpenTracing compatible tracer.

```go
package main

import (
    "github.com/uber/jaeger-client-go"
    "github.com/uber/jaeger-client-go/transport"
    "github.com/opentracing/opentracing-go"
)

// Start a Jaeger tracer, supplying the sampling strategy and the
// reporter configuration.
t, closer := jaeger.NewTracer(
    "sample-text",
    jaeger.NewConstSampler(true),
    jaeger.NewRemoteReporter(transport.NewHTTPTransport("host:port")))
defer closer.Close()

// Use it with the OpenTracing API, setting it as global.
opentracing.SetGlobalTracer(t)
```

2.1.3 OpenTelemetry

The OpenTelemetry[4] project came about as a result of the merging of two previous projects, namely the previously mentioned OpenTracing project as well as OpenCensus project. The OpenCensus project originated from Google and had many similar goals to OpenTracing. Alongside having an interface for distributed tracing gathering, it also supported instrumenting applications to output application metrics data. To reduce the fragmentation in having two independent APIs for distributed tracing, the two projects decided to merge into one standard going forward. At the time of writing, support for OpenTelemetry is still very sparse, due to the fact that it is still a very new specification set, while still being largely backwards compatible with both OpenTracing and OpenCensus, providing API bridges to maintain compatibility.

The OpenTelemetry API improves upon OpenTracing by introducing a set specification for context propagating header keys used to identify specific values, created as a W3C specification. In OpenTracing API implementations, different vendors would use different keys to denote values such as the trace ID etc in, for example, HTTP headers. This would break the context chain if different codebases used different vendor implementations in a service dependency graph. By demoting vendor libraries to providing simple *exporters* that define how distributed tracing data is exported to backend systems rather than having them provide tracer implementations like the way it was done with the OpenTracing API, the OpenTelemetry project achieves a better level of interoperability between codebases, where the OpenTelemetry *collector* can act as the common sink for the different services, becoming the source of truth regarding the eventual backend system to which the distributed tracing data will be exported to.

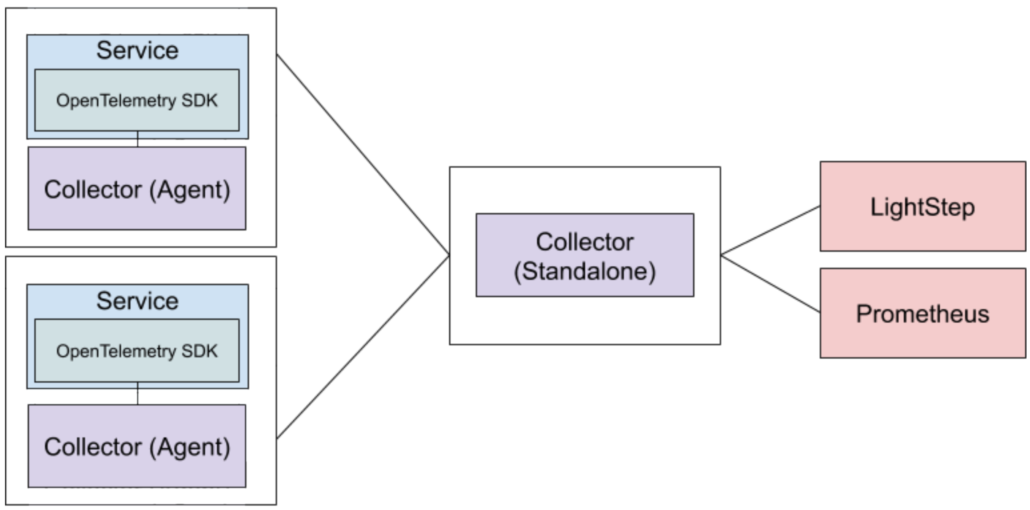

Figure 2.3: A high level overview of a typical OpenTelemetry setup, with services hooking into the OpenTelemetry SDK to output telemetry to OpenTelemetry Collectors, which themselves forward data to the standalone Collector sink, which is configured to send metrics to a *Prometheus* server and distributed tracing data to the LightStep API.

3. Design & Implementation

In this section, the different technical aspects will be covered, design decisions and components that played a role throughout the projects lifecycle. This will include third party services and the roles they played.

3.1 Architecture Design

3.1.1 Distributed Tracing API

The OpenTracing API interface was chosen as the foundation for this project. This decision was made due to the large language support and comprehensive open source tooling built around the OpenTracing API. It is also widely supported by many third party vendors, including LightStep, DataDog and Honeycomb amongst numerous others.

OpenTelemetry was initially considered as an alternative choice instead of OpenTracing, but was ultimately decided against due to it still being a very new standard, with OpenTracing having much more comprehensive support and documentation resources from both application libraries and distributed tracing tools. This lowered the productivity barrier, as the OpenTelemetry backing components, including the collector, introduce more complexity than what exists in a simple development-oriented OpenTracing setup.

Potential explorations around the OpenTelemetry API are discussed in the *Future Work* section.

3.1.2 Supporting Services

As both explored ideas will be interacting with distributed tracing data, there are two pieces to the puzzle of having a set of traces to work with. Firstly, a way of collecting trace data from applications is needed, and secondly, the database into which the data is persisted.

The OpenTracing platform, Jaeger, was chosen for this project. Jaeger is wholly self-hostable, providing a convenient setup for single-machine development purposes with a single, all-inclusive binary available from the distribution archives, as well as a pre-built *Docker* image published to DockerHub. The full Jaeger package, available in whole in the aforementioned all-inclusive formats, includes the trace collector, trace search and visualisation user interface and agent.

By default, the all-in-one Jaeger distribution stores all trace data in-memory. For convenience to persist data in between machine reboots, the JSON document search engine database, *Elasticsearch*, was chosen as the storage backend for Jaeger, as one of two possible choices alongside *Apache Cassandra*, a column store document database. The Backend API, discussed later in this chapter, interfaces with the Elasticsearch database as the source of truth for the trace data. A complementary user interface for Elasticsearch, maintained by the developers of Elasticsearch, *Kibana*, was used throughout the development of the project to view the raw trace data as it is represented in the database.

3.1.3 Backend API

To query for and transform the distributed tracing data stored in Elasticsearch, an HTTP API was developed for the two codebases of the ideas explored to interface with. It implements a GraphQL HTTP API server, the concepts of which are explained in detail below, to fetch the trace data from Elasticsearch.

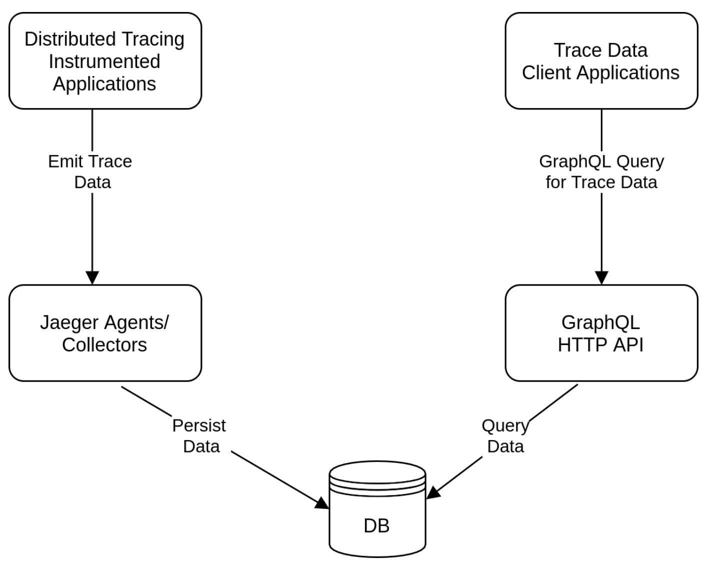

Figure 3.1: Basic diagram illustrating the architecture design for a sample system, including applications instrumented to emit trace data, the Jaeger agents and/or collectors, the database in which trace data is persisted and the GraphQL HTTP API that the clients such as those implemented as part of this project interact with to obtain trace data.

GraphQL

As an alternative to traditional RESTful HTTP services, GraphQL is both a query language and a server-side runtime for executing queries against a defined data schema laid out by the type system. The type system can be shared between both servers and clients alike, allowing for a common, known data schema. The data schema consists of user defined object types representing the data, as well as two special types: the *Query* type and the *Mutation* type. These define the entrypoints into the GraphQL server, allowing for the fetching and modification of data, and can take a defined set of arguments as defined in the schema, which may be defined as being optional.

Listing 3.1: The base GraphQL schema, defining a query and data types for the trace data.

```
type Query {
    findTrace(traceID: String): Trace
}

type Trace {
    traceID: String!
    spans: [Span!]!
}

type Span {
    traceID: String!
    spanID: String!
    parentSpanID: String
    duration: Int!
    startTime: Int!
    operationName: String!
    serviceName: String!
    logs: [LogPoint!]
    tags: [Tag!]
}

type LogPoint {
    timestamp: Int!
    fields: [LogPointField!]!
}

type LogPointField {
    key: String!
    type: String!
    value: String!
}

type Tag {
    key: String!
    type: String!
    value: String!
}
```

One of GraphQLs improvements over traditional REST is the ability to specify fields to *resolve* in the server-side runtime engine. Besides preventing under- and over-fetching, this also allows for the ability to selectively augment the returned data, adding extra fields or even changing existing ones on-demand, all within the same query while not incurring the costs for clients that do not request them. Listing 3.2 shows an example of selective field resolving. The second query also requests the `spans` field, which could have a range of effects on the server e.g. performing an extra SQL `JOIN` statement, while also resulting in potentially more data being sent down the wire.

Listing 3.2: GraphQL query to fetch a trace object and the span ID of each of its spans vs a query to fetch the trace object, its spans and every tag of every span.

```
{
    query findTrace(traceID: "asdf") {
        traceID
        spans {
            spanID
        }
    }
}

{
    query findTrace(traceID: "asdf") {
        traceID
        spans {
            spanID
            tags {
                key
                value
            }
        }
    }
}
```

Interfacing with the Database

As the decision was made to use the Jaeger tracing platform backed by Elasticsearch for this project, the backend server must interface with Elasticsearch to be able to serve the data to the API clients that make up this project. Under these circumstances, this project could be potentially used in production by teams that host a Jaeger tracing platform on their infrastructure with Elasticsearch. The backend must simply support connecting to the database, building JSON queries at either an abstracted or low level and finally being able to execute those queries against the database. Elasticsearch has first-party libraries for a large number of languages, and given Kotlin's interoperability with Java, the backend makes use of the Java Elasticsearch client library. Adding support for Apache Cassandra database would be trivial, however outside of the scope of this project.

While the Jaeger tracing platform is commonly employed by development teams, vendor hosted distributed tracing systems also have seen large adoption due to the fact that it allows teams to not worry about maintaining an instance of Jaeger in their infrastructure, either due to convenience, monetary costs needed to host the Jaeger platform or having to maintain extra services. These vendors often do not provide a public API for querying and fetching trace data. Discussion around possible solutions for these scenarios can be found in the future works section. The rest of this report works under the assumption of the Jaeger tracing platform being employed, either as the sole distributed tracing system or complementary to a vendor hosted solution.

3.1.4 Debug Adapter

For this part of the project, it was decided to develop the idea of integrating the telemetry from instrumented applications into the debugger API of Visual Studio Code. Visual Studio Code was chosen as the editor for which the integration would be built due to its extensive extensibility and first class Debug Adapter Protocol support. Built in Typescript, a Javascript superset with type annotations, the service implements the Debug Adapter Protocol to bridge between the trace data stored in Elasticsearch and the editor to provide many of the same features one would expect from traditional debugger tools such as the GNU Project Debugger (GDB).

Debug Adapter Protocol

Figure 3.2 displays the general architecture of how editors and tools utilize the programs that implement the Debug Adapter Protocol to interact with lower level debug runtimes, such as GDB etc. Each editor or tool would contain a small, lightweight shim, one per debug adapter, that launches the debug adapter with the appropriate configuration, potentially with user supplied configuration data, before handing off to the in-editor/tool generic debugger that interacts with the now running debug adapter through the Debug Adapter Protocol.

The Debug Adapter for this part of the project combines combines the three necessary parts in the one codebase, the shim, debug adapter and debugger runtime, for convenience in developing the proof of concept. It builds off the Visual Studio Code mock debug adapter, a sample codebase implementing the shim, debug adapter and a dummy debugger runtime in one. Heavily inspired by asynchronous event-based programming as is commonplace in the NodeJS ecosystem, the debug adapter invokes asynchronous methods on the debugger runtime when it receives requests from the editor or tool in the Debug Adapter Protocol format, upon which the debugger runtime may emit events that the debug adapter translates into Debug Adapter Protocol events for the editor or tool to consume.

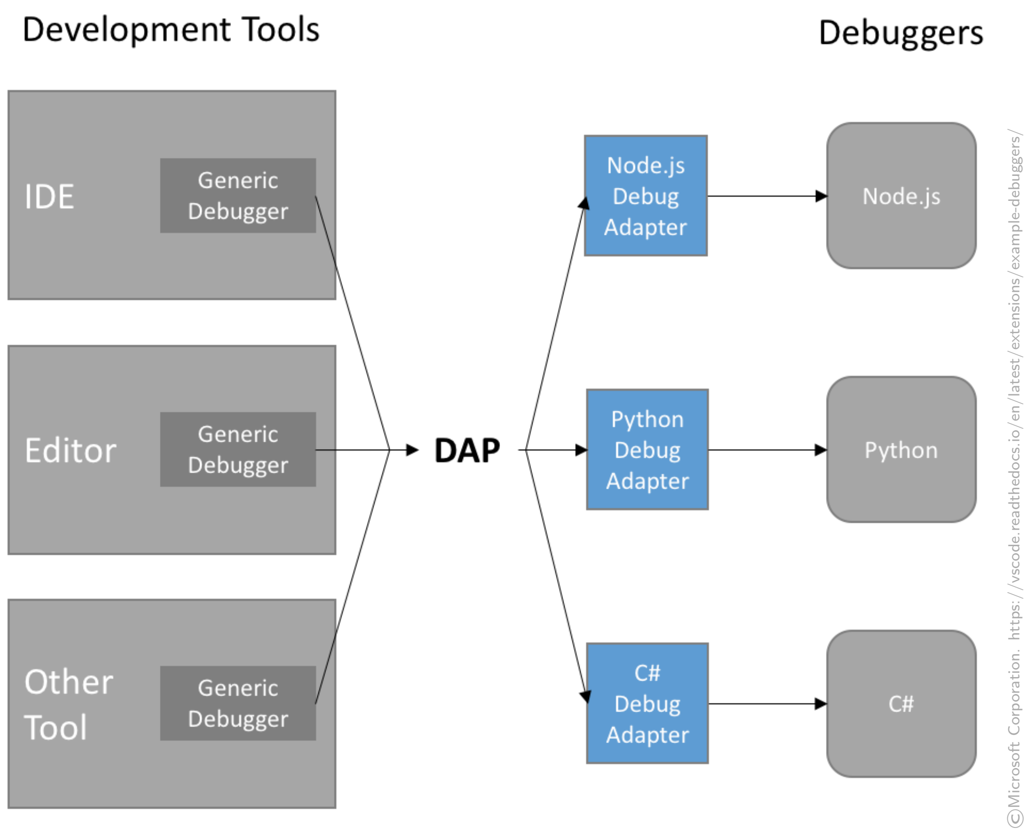

Figure 3.2: Diagram displaying the relationship between Editors and Tools, the Debug Adapter Protocol, Debug Adapters and the possible Debug Runtimes. Not shown are the editor/tool dependent shims. Each *Development Tool* contains a debugging utility that speaks the Debug Adapter Protocol to communicate with various Debug Adapters.

Concept

Conceptually, the role of the debug adapter created for this project is simple: given a trace ID, it should fetch the spans and their metadata for the trace ID, load the source code files for each span and allow the user to step through the code on a span-by-span basis (or more granular). If successful, the user would be able to see the source code file and line in which the current span was started and step back and forth between spans and their respective source code files and lines. Given the focus around non-monorepo codebases, it should also have the ability to jump to and from files outside the currently opened codebase, assuming that the codebases for the other services also reside on the user's machine.

The main approach to making this set of requirements possible is by utilizing the tag constructs defined in the OpenTracing specification to carry additional runtime or compile time information that provides enough information to the debugger runtime. This can be done by implementing shims that wrap OpenTracing trace implementations with methods that add the required information as tags to newly created span instances. When the debugger runtime queries the backend API, it can request an additional field on the span types that the backend GraphQL resolver resolves by transforming the information gathered by the shims into a format more useful to the debugger runtime than the raw collected information. This results in being able to interface with the same GraphQL query as the project idea that follows this part, while not incurring the transformation costs in the following idea.

In practice, there are a number of challenges that arise in various parts of this concept; gathering the information required to pinpoint the source file and line varies in format across different programming languages, amount of stack trace information that is to be or that can be gathered and its formats across different programming languages, mapping different services to the location in the filesystem of their codebase as well as handling spans emitted from services not appropriately instrumented for this to work. There are different trade-offs with various approaches to these problems that will be outlined.

Collecting and Parsing Runtime Data

When dealing with collecting information on the source file and line number at which a span was started, there are two main approaches with highly contrasting pros and cons. The first approach is to simply query the runtime, where applicable, for the file path and line number associated with the function that invoked the span creation. This is often a relatively cheap operation, with a small footprint in the amount of data that a span is tagged with. The main downside of this approach is that while it results in a small amount of extra data attached to a span, this reduced amount of information reduces the capabilities of the debugger runtime. With this information, the debugger runtime is limited to jumping between those single, individual points in the code at which the spans are created (this could be expanded to also wrap the log-point calls to do the same).

The second approach involves querying the runtime for an entire stacktrace leading up to this point. With this approach, there is considerably more data, with all the function calls of the stack frame tree branch leading up to the span creation being gathered, allowing for the possibility to step back and forth at the function level rather than simply at the span level. In both these instances, the data is attached to spans as tags. This approach was chosen to be explored for the project due to the larger potential that accompanies the greater amount of data. For further processing outlined in the following paragraph, the programming language from which the span originates is added as a tag to the span. There are a few downsides to this approach and possible workarounds that will also be discussed.

In many language implementations of the Jaeger tracer, tag values are limited in length to either 256 or 1024 characters. When dealing with stacktraces, this limit is too restrictive for storing entire stacktraces. This is due to an outstanding issue in Jaeger client implementations when using UDP as the transport protocol, as span sizes would exceed the maximum UDP packet size of 65535 bytes[5]. This makes using HTTP as the span transport method a hard requirement to avoid UDP packets being dropped, causing entire spans to be lost. The client implementations also provide an interface for changing the max tag value length, which must be utilised to prevent the truncating of the stacktraces. As this number cannot be set on a per-span basis, it must be set at a reasonably large number ahead of time based on previous examples of possible stacktrace sizes or the maximum value of an integer for the given platform. Another downside to full stacktraces is that the procedure for getting an entire stacktrace and transforming them can often be considerably more expensive than adding simple tags to a span. Unfortunately, sampling rules in the Jaeger client implementations, the processes by which it is decided whether or not spans are exported, are only applied to log points and the final reporting of spans. This results in the potentially expensive operation of gathering stacktraces to be run even if the span will never be reported. Despite implementing head-based sampling, where it is know when the root span is created whether or not the trace will be reported, the API does not expose an interface for querying this fact. For this reason, it was important to make the operation as cheap as possible, leaving as much of the stacktrace transformations as possible to query time, when the debugger runtime requests a trace from the backend.

Interfacing GraphQL API with Debug Adapter

With the stacktrace stored in the database as a span tag in string format, the backend GraphQL schema is amended by adding a field to the span type with an associated resolver and types that define a stacktrace and its individual stack frames. This resolver reads the stored stacktrace and the programming language associated with the span to process it into a stacktrace object usable for the debugger runtime.

As each language has a different format for its stacktraces, a stacktrace parser must be created for each language supported. These will parse the raw stacktrace strings and returns them as a parsed and formatted stacktrace object to the GraphQL resolver. Listing 3.3 shows the updated GraphQL schema with the unchanged sections filtered out.

Listing 3.3: GraphQL schema updated with stack- trace and frame objects, unchanged fields and objects filtered.

```
type Span {
    // ...
    stacktrace: StackTrace
}

// ...

type StackTrace {
    stackFrames: [StackFrame!]!
}

type StackFrame {
    packageName: String!
    filename: String!
    line: Int!
}
```

In this format, the debugger runtime has a more language agnostic representation of the stacktraces collected over the lifetime of a trace. There are still differences in how absolute local paths are resolved from this information in order to load the correct file in the editor from the users local filesystem, from differences in how codebases are structured into different packages for encapsulation and modularisation, as well as where dependencies are stored in the filesystem. For simplicity, resolving dependencies was not included in this project outside of the standard library for languages where implementation was trivial.

Alongside showing a stacktrace and the line in the file where the current span was created (or in the case of traditional debuggers, the line it has halted execution on), editors often have a panel dedicated for displaying local and global variables for the current stack frame and execution context. Largely due to performance reasons, it is not possible to attain this level of information on local and global variables when in the distributed tracing context unlike when using a halting debugger, assuming that *DWARF* symbols (*Debugging With Arbitrary Record Formats*, a standardized debugging data format embedded in executable binaries)[6] are still baked into the binary, which may not always be the case, as it is not uncommon for binaries to be stripped of DWARF symbols to reduce their size. Instead of this, the different types off tags and other key-value pairs can be displayed in said panel in place of variables. This works best when traces are heavily tagged with rich, high cardinality metadata which gives greater insight into a service's behaviour, both through the debugger interface and other means, and larger dimensionality with which to reduce a set of traces down to those with a common set of tags that exhibit certain behaviour.

3.1.5 Service Topology Frontend

In the service topology part of the project, the aim is to explore how to increase the value derivable from graph representations of the topology of a service architecture, at both an individual trace level and in aggregate. A common visualisation in many vendors in basic formats, they are commonly based on a type of directed graph called a *force-directed graph* due to its aesthetically pleasing properties. In the following section, the two main classes of service topology visualisations will be introduced, monitoring oriented and metrics based visualisation as well as observability oriented and event based visualisations, with the differences between the two being highlighted with an emphasis on the need for the latter type.

A basic service topology graph can give a quick overview of the dependencies between different services in a distributed system. However, it provides little value as an observability tool for debugging production issues or finding internal services that are returning errors for certain customers. For developers and administrators to derive tangible value from such a visualization, there is a need for a more dynamic system. The ability to generate views based on arbitrary attributes would allow for users of the system to create more personalized and focused graphs based on their needs, such as grouping by high cardinality fields like end-user IP address, user identifiers as well as metrics such as error rates, response times etc[7].

While they are commonly found in distributed tracing vendor services, the implementations are often very simplistic. A common example of such an implementation begins with a basic, often directed graph of each service as defined by the service name metadata passed into a tracer implementation. From this, the relationships between graph nodes is drawn by querying a spans service name and that of its parent span, which gives both the relationship and the directionality of said relationship. Nodes and edges are then often annotated with various metrics derived from the various span fields and tags, with average or percentiles of span durations as well as error rates and number of requests per unit of time being the standard figures on display. Figure 3.3 is an example of such an implementation from The RisingStack's NodeJS monitoring platform. Often misrepresented as features of *observability* platforms or services, graphs of such kind are more aptly described as features of *monitoring* platforms or services. The line between the two terms is often intentionally blurred by vendors which provide monitoring services looking to sell to companies who are in need of observability services. Monitoring is commonly described as being how **known-unknowns** are handled, including defining thresholds for values on metrics such as error rates or response times, most commonly working with pre-aggregated data of low dimensionality to give a quick, overview of the system in aggregate. Such systems are ideally tailored towards answering the questions one knows to ask beforehand, hence known-unknowns, which can range from questions such as does the database server require more CPU power or if a recent deployment of a service was correlated with an increase in API errors.

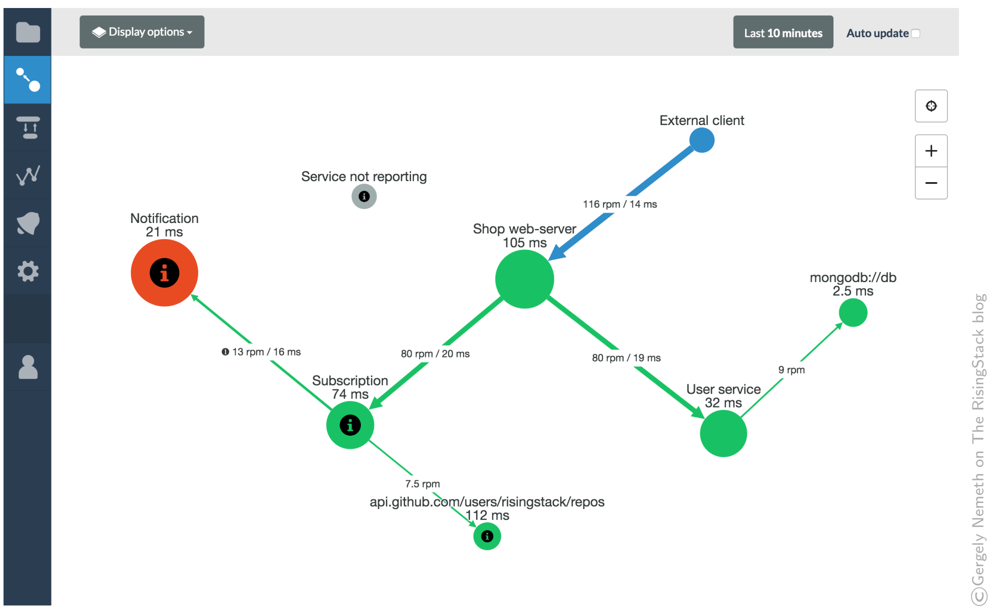

Figure 3.3: Example of the Microservices Topology visualization from RisingStack's monitoring platform, showing a basic topology overview with key metrics based on traces.

In contrast, observability is all about the **unknown-unknowns**, being able to ask different, unforeseen questions to a highly dimensional dataset with high cardinality fields that allow for arbitrary operations on raw, non-aggregated events, with the ability to drill down on individual events in the set from an aggregate overview. This allows one to receive answers to the questions on the behaviour of a system, in arbitrary aggregate down to an individual level. This sets the tone for what an advanced service topology visualisation should provide to allow for it to be distinguished from standard monitoring alternatives. Building off an event-based, non-aggregated dataset, the service topology graph will start out in a similar fashion to the aforementioned metrics based implementations, however in contrast deriving metrics at query time from the events. This will serve as the foundation from which additional operations and features will build off.

Filtering

The first vital operation that this visualisation should have is the ability to filter out traces from the working dataset arbitrarily. This allows us to reduce the amount of noise affecting the metrics on display at the edges and nodes, giving users the metrics specific to the new, filtered dataset view. As the user filters on various fields of arbitrary cardinality such as specific error codes, company identifiers or even performs filter operations on derived metrics such as the requests to a service with response times above a certain threshold, the metrics displayed on the graph such as HTTP error codes, response times etc that are displayed in the graph update dynamically to reflect the new working set consisting of the filtered entries from the original dataset, allowing for questions such as "*Is a particular error associated with any particular service or node?*" or "*What proportion of companies from a certain state with a certain cookie set are experiencing slow response times?*" to be asked without having instrumented for those particular questions, while being visualised in a service centric style.

Grouping

Grouping of data by an arbitrary field is the next operation that provides high value. This operation is a powerful method by which users can discover common traits of events exhibiting certain behaviour as well as the distribution of errors or other field values across events. By virtue of the form of the visualisation, groupings of event objects are displayed inherently by service identifier/name. This trait extends to grouping by additional fields, resulting in groupings always being in conjunction with service identifiers. The outcome of this is a topology view with a node for every service identifier/grouped field value combination.

As an example, assume a small scale service topology of 5 services for a business with 10 customers. The services consist of a database, an API gateway and three internal services. A user sees that the 99th percentile response time between two services, A and B, is abnormally high. To see if this problem is isolated to a specific customer or if it is happening to everyone, they could perform a group operation on the customer identifier. The topology graph updates, with every service node being multiplied by ten in count e.g. ten nodes for service A, with combinations ranging from {service name: A, customer ID: 0} to {service name: A, customer ID: 9} and so on, along with the accompanying edges and metrics for each combination. With the response times now separated by customer identifier, the user can see the distribution of the high response times, whether or not it was isolated to specific customers or not.

For the trivial example given, the graph already expands from five nodes to 50 nodes after grouping by customer identifier. For a company with more customers, this would become unwieldy in size and should therefore be used in conjunction with the aforementioned filtering capabilities to bring the graph down to a manageable size by excluding grouping combinations where the key metrics are not above or below a defined threshold.

These capabilities can also be expanded to reduce noise in the case of high cardinality service identifiers e.g. if using a serverless platform such as AWS Lambda in which each invocation may have a locally unique identifier, the service topology graph may explode with a large number of one-time nodes. A form of conditional grouping, such as grouping all nodes where the service identifier begins with a given prefix, could vastly reduce the noise from a graph featuring many nodes from ephemeral services as with serverless platforms or in companies with high deployment frequencies where service identifiers are uniquely suffixed on every deployment.

3.2 Implementations

3.2.1 Backend API

It was decided that Kotlin would be the language used for the backend after evaluating the different Elasticsearch and GraphQL libraries available, of which there were libraries that utilized the Kotlin *Domain Specific Language* (DSL) to define the queries and schemas respectively.

Listing 3.4: Kotlin snippet of defining the GraphQL schema using the Kotlin DSL.

```
val schema = KGraphQL.schema {
    query("findTrace") {
        resolver { traceID: String ->
            esRepo.getTraceByID(traceID)
        }.withArgs {
            arg<String> { name = "traceID"}
        }
    }

    type<Span> {
        property<StackTrace>("stacktrace") {
            resolver { span ->
                StackTrace.fromSpan(span)
            }
        }
    }
}
```

Listing 3.5: Comparison between Elasticsearch query using Kotlin DSL and the query in its JSON representation, where ⟨traceID⟩ refers to a variable storing the trace identifier.

```
// Kotlin DSL
val query = bool {
    must {
        term { "traceID" to <traceID> }
    }
}

// JSON Representation
{
    "bool": {
        "must": {
            "term": { "traceID": "<traceID>" }
        }
    }
}
```

3.2.2 Debug Adapter

3.2.3 Service Topology Frontend

4. Evaluation

5. Conclusion & Future

5.1 Conclusion

5.2 Future Work

5.2.1 OpenTelemetry

Bibliography

[1] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag, *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Google, Inc. 2010 https://research.google/pubs/pub36356/

[2] Mike Barry, Brian Card, *Visualizing MBTA Data.* 10th June 2014 https://mbtaviz.github.io/

[3] Benjamin H. Sigelman (co creator), *The OpenTracing project.* October 2015 https://opentracing.io/

[4] OpenTelemetry Authors, *The OpenTelemetry Project.* 19th April 2019 https://opentelemetry.io/

[5] YurzShkuro, *Jaeger Span Tags and UDP packet sizes.* 31st May, 2018 https://github.com/jaegertracing/jaeger-client-go/pull/316#issuecomment-393386461

[6] Michael J.Eager, *Introduction to the DWARF Debugging Format.* Eager Consulting, April 2012 http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[7] Cindy Sridharan, *Distributed Tracing - we've been doing it wrong.* Medium, 2nd July 2019 https://medium.com/@copyconstruct/distributed-tracing-weve-been-doing-it-wrong-39fc92a857df

[8] Mike Barry, Brian Card, *Visualizing MBTA Data.* 10th June 2014 https://mbtaviz.github.io/