Graph-based Anti-Pattern Detection in Microservice Applications

Amund Lunke Røhne
TNO-ESI & University of Amsterdam
the Netherlands
televisorpaper@gmail.com

Ben Pronk
TNO-ESI
the Netherlands
ben.pronk@tno.nl

Benny Akesson
TNO-ESI & University of Amsterdam
the Netherlands
benny.akesson@tno.nl

***Abstract*—Features of microservice architectures, such as scalability, separation of concerns, and their ability to facilitate the rapid evolution of polyglot systems, have made them popular with large organizations employing many software developers. However, the features that make them attractive also create complexity and require maintenance over the evolution of an application, especially concerning application decomposition. Microservice architecture decomposition evolves together with the application and is prone to errors referred to as architectural anti-patterns over its lifetime. These can be difficult to detect and manage because of their informal natural language definitions and a lack of automated tooling.**

This paper addresses this challenge by proposing an automated methodology for detection of architectural anti-patterns related to microservice dependencies. As a part of this methodology, a novel Granular Hardware Utilization-Based Service Dependency Graph (GHUBS) model is automatically inferred from telemetry data. Three commonly known anti-patterns have been formalized and algorithms provided to detect them in the model. The methodology is supported by an open-source tool that automatically detects and visualizes anti-patterns. The proposed methodology and tool are validated using both synthetic data and a case study of a popular microservice benchmarking suite, showing that instances of the formalized anti-patterns can be successfully detected.

***Index Terms*—microservice, software architecture, anti-pattern, telemetry, architectural smells**

I. Introduction

Microservice application architectures have become increasingly popular over recent years because of their ability to accelerate development in big organizations with many teams of developers, as opposed to traditional monolithic designs. Modeling organizations and software after business processes have become the norm, and microservice architectures excel at facilitating this kind of rapid workflow [1]. However, while microservice applications provide these benefits, they also tend to grow very large, sometimes in the range of 100 - 1000 services [2]. The size and complexity of microservice applications make it difficult for developers and architects to get a solid understanding of an application in its entirety [3]. This lack of understanding can be very damaging to quality assurance processes, such as debugging, performance optimization, and maintainability across multiple services and teams [4].

Ensuring that a system remains scalable, maintainable, and performance-optimized at every step of the development process is crucial, as the costs associated with a project are directly related to the quality of the application. Furthermore, microservice architectures are already susceptible to high degrees of network overhead, which can be further exacerbated by poor microservice application decomposition [5].

The literature on microservice architecture anti-patterns, also referred to as bad smells [6, 7], shows that anti-patterns in many cases are indicative of poor decomposition. These works often rely on domain knowledge for both detection and mitigation. The same literature also identifies anti-patterns that are strictly related to microservice dependency relationships, such as the Cyclic Dependency, Inappropriate Service Intimacy, Microservice Greedy, and Megaservice anti-patterns. However, these anti-patterns are described in natural language, making their definition subject to interpretation and detection difficult to automate. As a result, detection of these anti-patterns is done manually, which is time-consuming and error-prone in microservice applications with hundreds of services. This challenge is exacerbated, as microservices tend to evolve rapidly and independently, making maintenance and quality assurance a continuous process [8]. This paper addresses these challenges through the following two research questions:

RQ1 How can we automatically infer a model that supports detection of architectural anti-patterns that can be expressed in terms of microservice dependency relationships?

RQ2 How can we use such a model to automatically detect instances of known microservice architectural anti-patterns, such as Cyclic Dependencies, Inappropriate Service Intimacy, Microservice Greedy, and Megaservice?

Our research aims to reduce the time and cost related to detection of architectural anti-patterns through a novel methodology in which instances of architectural anti-patterns are automatically detected from traces and metrics gathered from the running microservice application are provided to support reasoning about possible mitigations. The five main contributions of this paper are: 1) a novel Granular Hardware Utilization-Based Service-Dependency Graph (GHUBS) model based on directed multi-graphs, which is automatically inferred using telemetry data, 2) formal definitions of three well-known microservice architectural anti-patterns, 3) algorithms for automatically detecting the formalized patterns in

the GHUBS model, 4) implementation of the proposed method in an open-source proof-of-concept tool called Televisor [9], and 5) validation of the methodology using both synthetic data and a case study from the popular DeathStarBench benchmarking suite [10].

In Section II, we describe the background material of this paper. Section III describes related work. Section IV introduces a new GHUBS model for microservice architecture analysis. Section V formalizes three anti-patterns from the literature and describes algorithms for automatically detecting them. We validate the methodology on a case study using our proof-of-concept tool Televisor in Section VI. We then discuss threats to validity in Section VII, before conclusions and future work are presented in Section VIII.

II. Background

This section introduces the necessary concepts to understand the contributions of this research. First, we introduce the concept of observability and the three main types of telemetry data in microservice architectures. We then introduce service dependency graphs, a commonly used data structure for capturing service dependencies that serves as a base for the GHUBS model proposed in this work.

A. Observability in Microservice Applications

Observability can be defined as: "a measure of how well internal states of a system can be inferred from knowledge of its external outputs" [11]. There are many commercial [12] and open-source tools for application performance monitoring and observability. Microservice architectures arguably need more of this kind of tooling than monolithic applications because of their inherently distributed nature [4]. Static analyses and end-to-end testing of microservice applications are usually not feasible because of their scale and polyglot nature. As such, we employ tooling to monitor applications and gather telemetry at runtime. The three main types of telemetry data are traces, metrics, and logs, each representing different information about our application. This work focuses on metrics and traces.

Metrics or more concretely *time-series metrics* is a "measurement of a service captured at runtime. The moment of capturing a measurement is known as a metric event, which consists not only of the measurement itself but also the time at which it was captured and associated metadata" [13]. There are many different kinds of time-series metrics in a microservice application. However, some of the more common metrics to gather are hardware utilization metrics, such as CPU, memory, and network utilization. While the means of exposing this information is highly dependent on the application infrastructure, tools like Prometheus [14] are used to store data in time-series databases. Prometheus, being the most prominent time-series metric database, also provides a query language for retrieving and manipulating the data, making it convenient for developers to create further tooling around.

Traces describe a sequence of process execution in an application and are composed of a finite set of *spans* [13]. Formally, we can define a set of traces

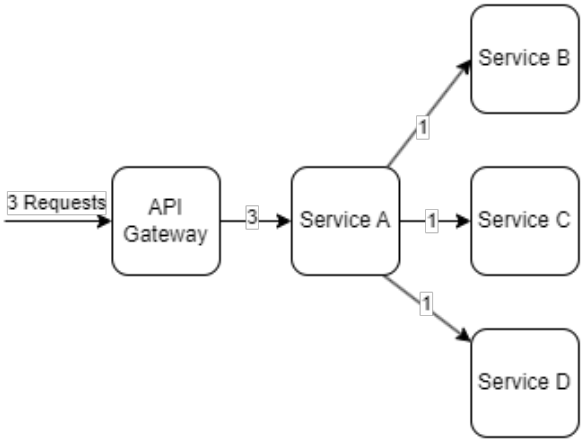

Fig. 1: Service Dependency Graph

$TR = \{tr_1; tr_2; :::; tr_{n-1}; tr_n\}$, where tr is defined as a tuple $tr := (id; SP)$ composed of an unique character string id and a set of spans SP, where the set $SP = \{sp_1; sp_2; :::; sp_{n-1}; sp_n\}$ and sp is a defined as another tuple $sp := (id; service; operation; parent; start; end)$. Here, $service$ is the service that emitted the span, $operation$ is an identifier for the human-readable operation name, $parent$ is the id of the parent span that is responsible for the execution of the child span and $start; end$ are timestamps defining the start and end timestamps of the span, respectively. Distributed tracing through context propagation allows us to see the causal connections between spans [15], and annotates every span with a link to their parent's span ID. This way a trace can cover many microservices allowing us to gain more insight into our system. Tools like Dapper [3], Zipkin, and Jaeger [16] allow developers to query spans and traces and use them to troubleshoot their applications.

B. Service Dependency Graphs

Service Dependency Graphs (SDG) [17], depicted in Fig. 1, are typically represented as a directed graph, including every relationship and a count for the number of times they have been called on each edge. They are a popular way of visualizing the topology of a given microservice application. SDGs have several features that are beneficial in the detection of anti-patterns. However, they are limited in granularity. One example of this is the lack of separation between application traces. Furthermore, SDGs prevent us from distinguishing individual spans or operations between microservices. Without this kind of information, the traditional SDG makes it difficult to say much about the relationships between individual microservices, making it more suitable for application-wide analysis. SDGs do not provide detailed information about the individual microservices that compose an application and are strictly concerned with relationships. This makes it difficult to reason about how the dependency relationships affect the health of the microservices.

III. Related Work

Anti-patterns are generally defined in natural language. A few notable exceptions that can be detected through static code analysis include the Cyclic Dependency anti-pattern and Hard-Coded Endpoints [6, 7], the latter referring to a lack of service discovery in an application. This lack of formal definitions of anti-patterns allows for subjective interpretation, which is detrimental to the detection and mitigation process as a whole. It creates inconsistencies between different research addressing the same or adjacent anti-patterns, and while facilitating discussion, does little to support the creation of universal good practices across microservice applications. This furthermore makes automatic detection of such anti-patterns very difficult.

There are several approaches for detecting anti-patterns in microservice applications using static analysis. The approach in [18] analyses compiled Java projects to derive an SDG. The work in [17] generates SDGs utilizing reflection. The SDGs produced in these works have the limitations discussed in Section II-B, which is detrimental to detecting the kinds of anti-patterns considered in this work. [19] uses a monolith API interface definitions (OpenAPI schemas) to derive decomposition suggestions based on matching natural language identifiers to an existing reference dictionary that groups related concepts. This makes the work only applicable to applications using OpenAPI schemas. In [20], researchers detect the Cyclic Dependency smell by extending an abstract syntax-tree-based tool created for detecting technical debt in monolithic Java applications. They were also able to detect instances of the Hardcoded Endpoints and Shared Persistence smells [6]. Again, this limits the applicability of the method as it is language-specific. In [21], structural coupling is computed in Java-based microservice applications by gathering coupling metrics found in the service classes. The researchers also chose to represent and visualize this in a directed graph, named a Structural Coupling graph. The metrics used here are always system-dependent, making it difficult to reason objectively about the severity of the structural coupling in a given system.

Despite their lack of general applicability, static analysis methodologies do have the advantage in that they are usually more performant, and can guarantee complete coverage of the application in question. This is not always the case with telemetry-based analysis. Telemetry-based approaches, such as [22, 23], use distributed tracing built into microservice applications to generate a service-dependency graph at runtime. This approach is preferable, as it allows us to analyze microservice applications regardless of the programming languages used in their creation. Furthermore, most observability tooling is built around the increasingly popular OpenTelemetry [13] standard, making it highly applicable.

In [24], an SDG is created based on service mesh logs, provided by telemetry tooling, containing information about inbound and outbound requests to determine dependency relationships. The researchers use the generated SDG to detect five anti-patterns, of which three are calculated using "anti-pattern metrics". These anti-patterns are ranked purely on application metrics without any indication of their severity, making it difficult to tell if the microservice application is unhealthy. Furthermore, this approach cannot detect the types of anti-patterns considered in this work.

IV. Granular Hardware Utilization-Based SDG

We proceed by describing the automatic inference of a novel model that facilitates detection of anti-patterns related to microservice dependencies, addressing RQ1. To capture the information we need, we have extended the traditional SDG with new features to accommodate the limitations discussed previously. This includes increasing granularity and mapping the microservices to their respective hardware utilization metrics. The result is the Granular Hardware Utilization Based SDG (GHUBS) model.

A. Increasing Granularity

The anti-patterns we wish to detect require us to distinguish between different spans and operations, something the SDG prevents us from doing. To tackle this granularity issue, we need to increase the number of edges that usually exist in an SDG. Instead of restricting the number of edges between two services to one per direction, we create an edge for each unique span between two services. The resulting graph shows not only which microservices are dependent on each other, but the actual individual operations that cause the dependencies. This is instead of having a single edge between services, regardless of the number of unique spans between them. We can tell if a span is unique by looking at its source and destination service, as well as its human-readable name and identifier that indicates its function.

Fig. 1 and 2 show the difference in an example with three external requests coming into a microservice application. Fig. 1 is the traditional SDG, where the three requests are grouped from the *API Gateway* to *Service A* and we can see the number of executions on each span. Fig. 2 on the other hand, shows us that the three requests are unique and labeled *R1*, *R2*, and *R3*, respectively. In fact, it can show us the trace in its entirety in the GHUBS model and how it is distinct from the other traces.

B. Adding Utilization Metrics

We are annotating the GHUBS model with hardware utilization metrics to support developers when thinking about different anti-pattern mitigation strategies, such as merging or splitting services. There are a few different approaches to do this. We can measure the utilization metrics of the platform the microservices are running on; the utilization metrics of the cluster the microservices are contained within, or we can measure the utilization metrics on a per microservice basis. We choose to record the utilization metrics on a per-microservice basis. While this will be an abstraction of the hardware platform, it will serve as a more accurate representation of the state of the microservices.

When it comes to recording utilization metrics, we need to be aware of what metrics we are collecting. In our case,

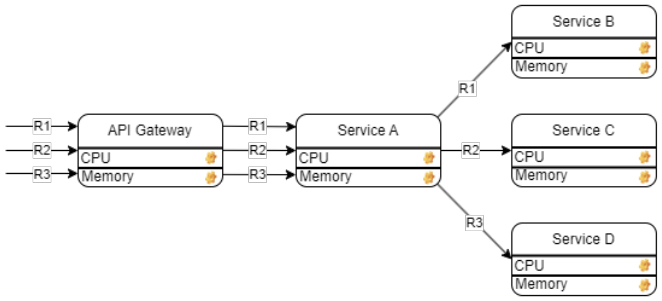

Fig. 2: GHUBS Model with Increased Granularity and Hardware Utilization Metrics

we want a conservative view of the system when performing validation, while still getting rid of rare outliers. The 99.7% percentiles, hereafter referred to as "tail", are hence suitable values. With that in mind, we wish to record six utilization metrics in the same time frame that we gather the traces for the construction of the GHUBS model. The metrics are the tail, mean, and standard deviation CPU and memory utilization. However, the GHUBS model is not limited to these metrics and can incorporate whatever metrics are most relevant to the work at hand. Some immediate examples could be network utilization per microservice, span latency between microservices on a request, and end-to-end latency for an entire request.

In addition to increased granularity, Fig. 2 shows the hardware utilization metrics visualized as gear icons included in the GHUBS model.

C. Formalizing the GHUBS Model

With our addition of potentially parallel edges to the traditional SDG, the proper formalism for the GHUBS model is a directed multi-graph, more commonly found in the domain of networking. As such, we can take advantage of the established language for directed multi-graphs to define our anti-patterns. That is, define our bad smells according to patterns of edges and vertices in the GHUBS model. We let V be the set of all nodes in the directed multi-graph, where each node $v \in V$ represents a service. A node v is a tuple and contains a name and utilization metrics and is defined as $v := (name; utilization)$. Let E be the set of all edges on a request, where each edge $(u; v; r) \in E$ represents a dependency from service u to service v, and r represents the unique operation name of the edge from u to v. Then, we let $(i; E) \in R$, where i is an identifier for the request, and R is a set of all requests in the GHUBS model. The GHUBS model can now be defined as a directed graph $G := (V; R)$, with nodes V and requests R.

D. Automatic Inference of the GHUBS Model

Manually modeling complex evolving systems is time-consuming and error-prone and is often a barrier to adoption of model-based methodologies [25, 26]. Data-driven design where models are automatically inferred from operational data has been proposed as a way to mitigate this problem [27]. To address this aspect of RQ1, Listing 1 shows the pseudo-code for inferring the GHUBS model from the trace definition in Section II. We define our function with a set of traces, V, as a parameter on Line 1. On Line 2, we define our set R that will group edges by their requests. We iterate over all traces on Line 3 and define sets of edges on Line 4. On Line 5, we iterate over the spans in every trace. Next on Line 6, we define $u; v; r$ which will contain the service where the edge comes from, the service where the edge ends, and an identifier making the edge unique in case of multiple parallel edges. On Line 7, we add the new edge to a set E, which is grouped by the request name of the trace on Line 8. Finally, we package both the set V with our services, and the set R with our requests in G and return the GHUBS model on Lines 10 and 11.

```
1  function ghubs(TR):
2      V = fg, R = fg
3      forEach(trace tr 2 TR):
4          E = fg
5          forEach(span sp 2 of tr.SP):
6              u,v,r = sp.from, sp.to, sp.operation
7              V = V [ sp.service
8              E = E [ (u,v,r)
9          R = R [ (tr.id, E)
10
11     return (V, R)
```

Listing 1: Creation of GHUBS Model

Note that while deriving the GHUBS model from an existing system through telemetry, it is also possible to manually create a model of an intended software architecture during system design. This is done by specifying the microservices, the requests in the application, and the dependency relationships in those requests. Having the ability to do this allows developers to evaluate their design before development begins, or test a particular idea using artificial data.

V. Formalization and Detection of Anti-Patterns

Having defined our new fine-grained GHUBS model, we proceed by formalizing architectural anti-patterns and showing how they can be detected in the model, addressing RQ2. The three example dependency-based anti-patterns we are looking for are *Inappropriate Service Intimacy*, *Megaservice*, and *Microservice Greedy* [7]. The methodology also supports the well-known Cyclic Dependency anti-pattern, although it is not included here for brevity. For more information about the formalization and detection of this anti-pattern, refer to [28].

To automatically detect architectural anti-patterns, we need formal descriptions that can be compared to the structure of software systems inferred from available telemetry data. Note that a single anti-pattern definition can be interpreted in several different ways. This is due to the inherent vagueness of the natural language definitions. Considering that, we have defined one instance of each of the three anti-patterns according to our interpreted understanding of the source material [6, 7].

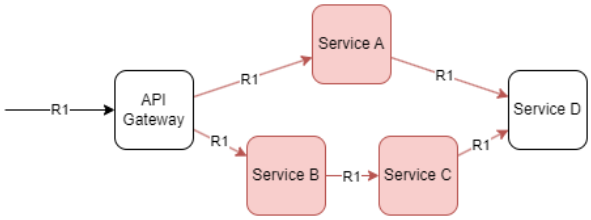

Fig. 3: Instances of Inappropriate Service Intimacy

A. Inappropriate Service Intimacy

Following the definition of the Inappropriate Service Intimacy anti-pattern where: "The microservice keeps on connecting to private data from other services instead of dealing with its own data" [7], we are looking for microservices that are dependent on each other or on a specific service in such a way that the separation of concerns between the services becomes blurred. However, the definition does not tell us anything about the number of microservices that are included. Neither does it define the interval of these connections or if it matters. The definition of what can be considered private data of a microservice is also ambiguous.

The instance of the anti-pattern that we seek to detect is when we have multiple microservices executing in parallel between a *diverging node* and a *converging node* on the same request. The same or related data is shared across multiple services, removing the separation of concerns. We consider this divergence to be problematic as it can result in race conditions and unnecessary network communication. If the processes are synchronous, it could also cause a tail latency bottleneck. An example of services with the Inappropriate Service Intimacy anti-pattern can be seen in Fig. 3, with the responsible services marked in red in the GHUBS model.

The figure shows an instance of the Inappropriate Service Intimacy anti-pattern we wish to detect. In formal terms, we are first looking for a node that has two or more inbound edges. If we find such a converging node, we want to detect all other nodes on the paths between the converging node, and a diverging node with two or more outbound edges. In nested instances, the top level is considered. The nodes between the diverging and converging nodes are marked as responsible for the anti-pattern. In formal notation, the Inappropriate Service Intimacy smell is on a single request, $r; E\ 2\ R$, for some request identifier r, where two or more sequences of nodes S contain $d; v_1; v_2; :::; v_k\ 2\ V$ such that $(v_i; v_{i+1}; 1)\ 2\ E$ for $i = 1; :::; k\ \ 1$ and $(v_k; c; 1)\ 2\ E$, where $d; c\ 2\ V$ and are the diverging and converging node, respectively.

Listing 2 shows the pseudo-code for detecting instances of Inappropriate Service Intimacy. We begin by defining our function and supplying a single request as a parameter on Line 1. Then, we use a function to find the diverging and converging nodes in the request on Lines 2 and 3. We create a set for our results on Line 5. On Lines 7 and 8, we iterate over the diverging and converging nodes and find all sequences between them on Line 9, excluding the diverging and converging nodes. If there is more than one sequence we add the node names from those sequences to our results set as a single set element on Lines 10 and 11. Finally, we can return our result of a nested set with sets of responsible node names on Line 13.

Fig. 4: Instance of the Microservice Greedy Anti-Pattern

```
1  function detectInnappropriateServiceIntimacy(R):
2      ds = divergingNodes(R.E)
3      cs = convergingNodes(R.E)
4
5      result = fg
6
7      forEach(divergingNode d 2 ds):
8          forEach(convergingNode c 2 cs):
9              S = getSequencesBetween(d,c)
10             if (jSj > 1):
11                 result = result [ [S...]
12
13     return result
```

Listing 2: Detection of Inappropriate Service Intimacy

B. Microservice Greedy

For the *Microservice Greedy* smell, we are trying to find services that only serve a singular purpose, and might be redundant in the microservice application architecture. In [7], the definition given is: "Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages." Such services are usually a result of poor planning or inconsiderate additions of new functionality [7]. Whatever the case for introduction, they increase complexity and decrease application maintainability needlessly. An example of the anti-pattern is shown in Fig. 4, where the responsible service is marked in red.

As opposed to our other detection methods, in the case of the *Microservice Greedy* anti-pattern, we look at all requests aggregated. This is to prevent flagging microservices that are only responsible for a singular function in an individual request but have responsibilities in other requests. Every node that is detected as such is marked as responsible for the anti-pattern. In terms of our GHUBS model, create a union of all edges in every request, $e_1\ [\ e_2\ [\ e_3\ [\ :::\ [\ e_n\ =\ E$, where $i; e\ 2\ R$. We define a function $g : V\ \ !\ E$ that takes a node as input and returns a set of inbound connections to that node.

$$g(v^0) = f(u; v; r)\, j\, (u; v; r)\ 2\ E; v = v^0 g$$

To check if a node only has a single inbound connection, evaluate $jg(v)j$. If $jg(v)j\ =\ 1$, then we can conclude that v has only a single inbound connection. In turn, v also has the *Microservice Greedy* anti-pattern.

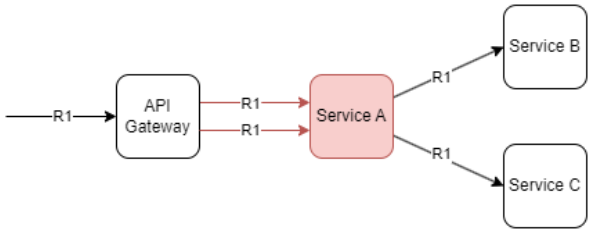

Fig. 5: Instance of the Megaservice Anti-Pattern

Listing 3 shows the pseudo code for the Microservice Greedy detection algorithm. First, we define our function and take in a set of all the requests R and another set of all the services V as parameters. Thereafter, we take the union of the requests and retrieve all of the edges on Line 2. On Line 3, we define a variable that will be an array containing the edges responsible for the anti-pattern. On Line 5, we iterate over all of the services $v \in V$. If we find that exactly one edge ends in v on Line 6, we append that edge to our results, and finally return the result on Line 9.

```
1  function detectMicroserviceGreedy(R, V):
2      E' = fe ∈ E | ∃(i, E) ∈ Rg
3      result = fg
4
5      forEach(node v ∈ V):
6          if (|g(v, E')| == 1):
7              result = result ∪ g(v, E')
8
9      return result
```

Listing 3: Detection of the Microservice Greedy Anti-Pattern

C. Megaservice

The Megaservice anti-pattern is defined as: "A service that does a lot of things. A monolith." [7]. The number of functions a microservice has to serve before it is considered a Megaservice is not clear following the definition, but we assume the functions are unrelated. The instance of the anti-pattern shown in Fig. 5 clearly shows the redundant execution between the "API Gateway" and "Service A", where "Service A" is marked as red and considered responsible. This could for instance be a remnant of a legacy monolithic application that has been transitioned into a microservice application, where a **single request** requires executing two operations to complete the desired functionality. This kind of relationship can cause data races and other unwanted behavior in the subsequent microservices and makes the Megaservice a greater point of failure and potential bottleneck.

Formally, the instance of the Megaservice anti-pattern we are looking to detect is when a node has two or more inbound edges coming from a second node on a single request. To check if we have such redundant edges, we defined a function $f : V \quad V \; ! \; E$ that takes in a pair of nodes and returns a set of edges between them.

$$f(u^0; v^0) = f(u; v; r) \, j \, (u; v; r) \in E; v = v^0 \;\; \& \;\; u = u^0 g$$

Where $i; E \in R$ for some request identifier i. To check if there are two or more redundant edges between two services u and v, we evaluate $jf(u; v)j$. If $jf(u; v)j \quad 2$, then we can conclude that there are two or more redundant edges between u and v. This in turn means that node v suffers from the Megaservice anti-pattern according to our definition.

Listing 4 shows us the pseudo-code for our Megaservice detection algorithm. On Line 1, we define our function with a single request for its parameter. We retrieve the nodes on the request on Line 2, and we define our result variable on Line 3, which is a nested set containing sets with the responsible edges. Lines 5 and 6 iterate over all nodes. If we find that we have more than two edges between nodes u and v on Line 7, we add node v to our result array on Line 8. Finally, the function returns the result array which will contain all edges pointing to a node that inhibits the Megaservice anti-pattern.

```
1  function detectMegaservice(R):
2      V = getNodesOnRequest(R)
3      result = fg
4
5      forEach(node u ∈ V)
6          forEach(node v ∈ V)
7              if (|f(u,v)| >= 2):
8                  result = result ∪ f(u,v)
9      return result
```

Listing 4: Detection of the Megaservice Anti-Pattern

VI. Implementation and Validation

To support the use of the proposed methodology, we have created a proof-of-concept tool that supports the application of the methodology to microservice applications. The tool has been validated with synthetic data, as well as through two case studies on two microservice applications from a well-known open-source microservice application suite.

A. Implementation

The methodology described in Sections IV and V was implemented as an application by the name Televisor (Telemetry-Advisor) [9]. The architecture of the tool comprises two modules, as shown in Fig. 6. The first module, which we refer to as the backend module, is responsible for pulling telemetry from a microservice application, creating the GHUBS model, and detecting the anti-patterns. The second module, the frontend, is responsible for displaying the detected anti-patterns and relevant metrics to the developer in a human-friendly manner. The developer uses these insights to reason about suitable anti-pattern mitigations. The backend module was written in Go, and the frontend module is web-based and was written in Typescript. The microservice applications were instrumented with cAdvisor for collecting utilization metrics.

B. Validation

To validate our methodology, we manually created synthetic GHUBS models with instances of all three formalized patterns using the mechanism described in Section IV-D. This allowed

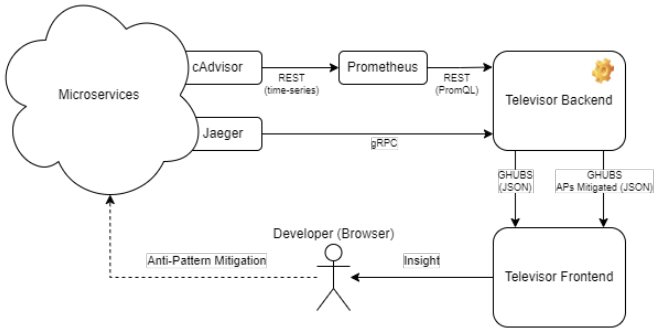

Fig. 6: Architecture of the Televisor tool

us to verify that the detection algorithms presented in Section V work correctly. All instances of the anti-patterns inserted in this synthetic model were correctly detected without any false positives.

In addition to validation with synthetic models, we also performed two case studies with benchmark applications found in the DeathStarBench microservice application suite [10]. To establish a ground truth, we first manually inspected the dependency relationships on each request to determine the presence of anti-patterns. For complex applications, this is very time-consuming, but both applications are small enough to allow manual validation in a reasonable time. From our manual analysis, we conclude that the Social Network application exhibits six instances of the Microservice Greedy anti-pattern, and the Media Application application exhibits one instance of Inappropriate Service Intimacy and three instances of the Microservice Greedy anti-pattern. A possible reason for the prevalence of the Microservice Greedy anti-pattern is that DeathStarBench addresses the need for microservice benchmarks with a larger number of services, which may have resulted in an unnecessarily fine-grained decomposition. Because of space limitations, we only present detailed findings from the Media Application in this paper. For results and validation of the Social Network application, refer to [28].

The scope of the Media Application, as described in the DeathStarBench paper, is as follows: "The application implements an end-to-end service for browsing movie information, as well as reviewing, rating, renting, and streaming movies" [10]. After running the included *wrk2* benchmarking tool, which executes all publicly exposed routes on the API Gateway concurrently over a set amount of time, the Media Application had six requests to analyze. Three instances of the Microservice Greedy anti-pattern, and one instance of the Inappropriate Service Intimacy anti-pattern were detected on a single request. This is consistent with the results of our manual inspection and no false positives were detected. The request in question is on the *n*wrk2–api*n*review*n*compose route. Fig. 7 shows the GHUBS model of the Media Application with the edges involved on that request. The services that exhibit the Microservice Greedy anti-pattern are highlighted in red, while the services that have the Inappropriate Service Intimacy anti-pattern are highlighted in purple.

To reason about how to mitigate the detected anti-patterns, we manually look at the dependency relationships and hardware utilization metrics in the GHUBS model and analyze a Jaeger trace of the /wrk2–api/review/compose request to argue about temporal behavior. Fig. 8 shows the Jaeger trace. First, we consider the three detected instances of the Microservice Greedy anti-pattern. The anti-pattern is detected because the three services review–storage–service, movie–review–service and user–review–service are only ever called by a single request from the compose–review–service. It is apparent that the dependency relation between the user–review–service and the movie–review–service waiting for a callback from the review–storage–service is not ideal. The three services perform strictly related tasks and are tightly coupled, making them suitable for a merger without the danger of creating a Megaservice. Furthermore, the three services all interact with separate databases, however, they are only invoked on this particular request. As the user–review–service and the movie–review–service are also dependent on data from the review–storage–service, there is strictly no need for separate databases. In fact, merging the services and utilizing a single database with multiple schemas would facilitate the use of database transactions, enhancing data integrity and possibly increasing maintainability by simplifying error handling. Looking at the hardware utilization metrics of the responsible services in the GHUBS model, we see that the CPU and memory tail utilization are very low. This indicates that we are unlikely to encounter any performance issues by attempting a merge. With this in mind, we believe that merging the services responsible for the Microservice Greedy smell in the Media Application is beneficial.

The Inappropriate Service Intimacy pattern involves movie–id–service, text–service, unique–id–service, user–service, and rating–service. The Jaeger trace shows unnecessary latency between the services involved. The incoming data on the request diverges into five different services and they all converge on the same service. Additionally, the service data boundaries do not seem to follow business processes, and certainly not Domain Driven Design [29]. The hardware utilization metrics in the GHUBS model show us that the offending microservices have low tail CPU and memory utilization metrics. This could be indicative of a merge being possible without performance concerns. However, because of the number of services involved in this anti-pattern, we do not think that a single merge is feasible. We could attempt recomposing the system after business processes. For instance, merging the services into three new services: a movie–service, user–service, and a review–service. Now, we could reduce the number of operations on this request to three. We can make a call to the new movie–service to retrieve the movie ID, at the same time retrieve or authenticate the user in the user–service and lastly upload our review in the review–service. This case study demonstrates how our methodology automatically detects instances of anti-patterns and supports the developer in reasoning about different mitigation strategies.

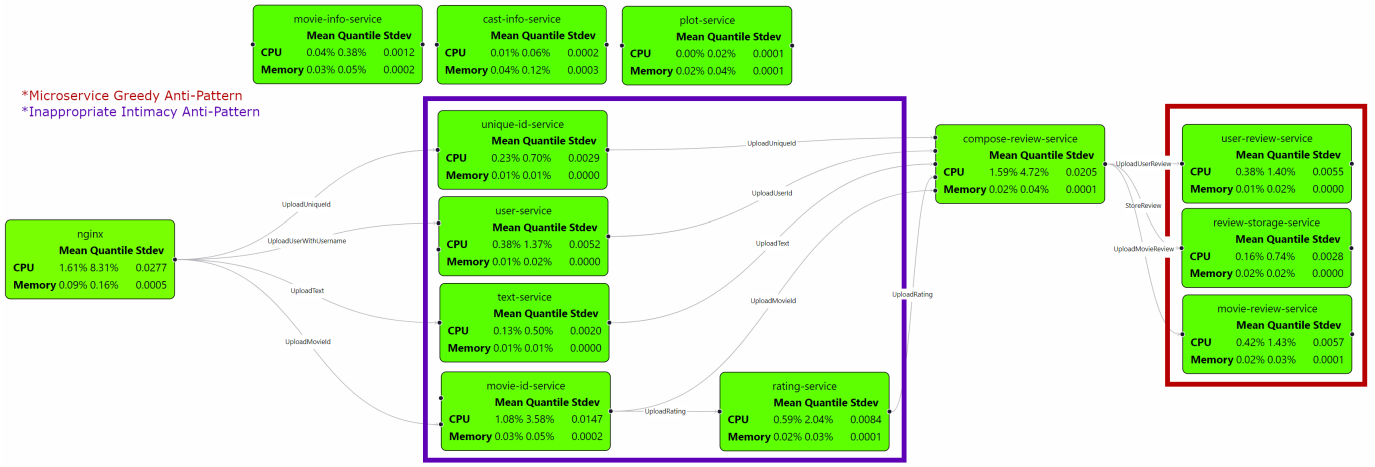

Fig. 7: Media Application Anti-Patterns

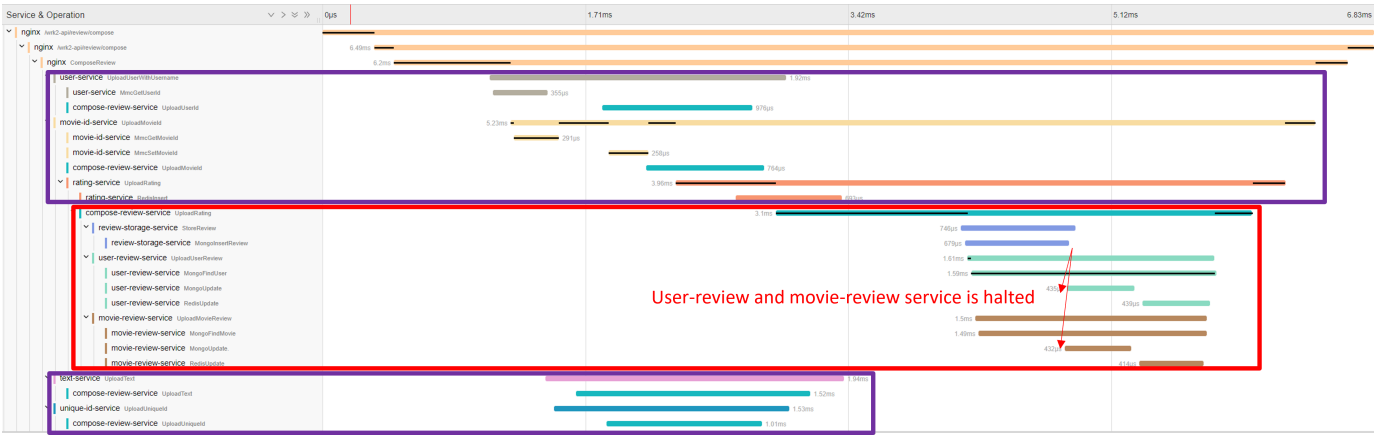

Fig. 8: Media Application Jaeger Request Trace

VII. Threats to Validity

This section discusses the threats to validity of the methodology proposed in this paper.

A. Internal validity

We proceed by discussing the internal validity of our research results, i.e. the extent to which they are valid within the scope of this study. A concern with any telemetry-based methodology is that the workload has to cover all paths through the application to get a complete view. If this is not the case when using our methodology, certain dependency relations between microservices may not be included in the GHUBS model, resulting in certain instances of architectural anti-patterns not being detected. How to create a workload that provides sufficient coverage is highly application-dependent and should be done with the help of domain experts. As explained in Section VI-B, we have mitigated this risk in our research by using the wrk2 benchmarking tool, which is provided as a part of DeathStarBench and executes all publicly exposed routes on the API Gateway.

The ability of our methodology to detect anti-patterns depends on how their definitions in natural language are interpreted and formalized. While we do believe that the formal definitions we have created are valid, they do not exclude other valid definitions of the same anti-patterns. Furthermore, the set of anti-patterns described here should serve as a way of thinking about architectural smells in general. Developers can hence formalize additional anti-patterns that can be expressed in terms of service dependencies and application metrics and provide an algorithm that can detect them in the GHUBS model. In most cases, the effort involved for an experienced developer to add support for an additional anti-pattern in the Televisor tool should be a couple of hours.

B. External validity

External validity considers the validity of our results beyond this study. The methodology and supporting tool have been validated using both synthetic data and two case studies based on applications from a commonly used microservice benchmark suite, providing initial evidence of the applicability and usefulness of our approach in this context. However, further validation is required with larger applications from different domains to establish how the methodology generalizes.