# Dynamic Analysis for Detecting Microservices Antipatterns

Leonardo H. de Braz
Instituto de Computação,
Universidade Estadual de Campinas
Campinas, Brazil
lhleonardo05@gmail.com

Breno B. N. de França
Instituto de Computação,
Universidade Estadual de Campinas
Campinas, Brazil
bfranca@unicamp.br

Bruno B. P. Cafeo
Instituto de Computação,
Universidade Estadual de Campinas
Campinas, Brazil
cafeo@unicamp.br

## ABSTRACT

The increasing adoption of microservices architecture in software development, driven by the need for flexible and scalable systems, has made it imperative to develop a strategy for monitoring, verifying, and validating the system's overall health at various levels, including architecture, design, and source code. Unfortunately, as software evolves, it becomes susceptible to recurrent poor solutions to problems that can negatively impact the system's health, also known as antipatterns. Cataloged techniques for detecting antipatterns in diverse architectural models employ different approaches, which involve both static analysis of code and environment, as well as dynamic analysis of data collected during execution. However, detecting antipatterns in an execution environment, particularly in the context of microservices, becomes challenging due to the architecture's inherent distributed nature and operational complexity, which present several difficulties in detecting antipatterns related to system visibility, communication, scope recognition, performance, and other challenges inherent to microservices. This work proposes an approach to detect primary runtime antipatterns, including four different strategies, and evaluates its effectiveness by applying it to a real-life open-source microservice application.

## KEYWORDS

Antipatterns, Dynamic Analysis, Microservices, Spinnaker, Software Architecture

## 1 Introduction

The ever-growing demand for software systems has led to various approaches to defining and maintaining them, including managing and evolving their software architectures. Consequently, planning and considering the architecture have become essential not only for the application's health but also for effective system management. Thus, various architectures have emerged to meet specific needs and scenarios, such as Microservices, Event-Driven, Hexagonal, and Command-Query Responsibility Segregation, among others.

In the microservices architecture, strongly adopted by companies such as Amazon, Netflix, and Meta, a microservice is a small, autonomous unit based on business capabilities and bounded contexts [18]. It is well-suited for delimited contexts and business capabilities, and with a granularity that supports distributed teams.

The literature attributes several claimed benefits to this architecture, including independent deployment and scaling, fault isolation, technology agnosticism, improved team productivity, business functionality segregation, resilience, and support for continuous integration and delivery [18, 20, 48]. However, its widespread adoption has made software quality assurance essential, as inadequate modeling, suboptimal architectural planning, and the misalignment between system evolution and initial design decisions can hinder maintainability, scalability, and long-term adaptability [6, 44].

A significant concern in microservices is the presence of *antipatterns*, which are indicators of potential problems at various abstraction levels, including code, design, and architecture [8, 22]. These can negatively impact system quality, including increased technical debt, performance bottlenecks, service coupling, reduced scalability, and difficulties with maintenance and deployment. Over time, these issues may hinder the system's evolution, increase operational costs, and compromise the user experience. Identifying and handling antipatterns is essential not only to maintain system quality and robustness but also to support the long-term sustainability and agility of the software architecture.

Although several detection methods, such as static code analysis, run-time monitoring, and graph-based techniques, have been proposed to identify antipatterns in microservice systems [3, 5, 11], each has its strengths and limitations, depending on the specific application context [11]. For instance, static analysis can be effective in detecting antipatterns, such as API Versioning and Hard-coded Endpoints, as they are directly related to the source code and static artifacts. Still, it may be incorrect or inefficient in detecting Bottleneck Services or Cyclic Dependency antipatterns, as they are often related to aspects observed at runtime. Tools such as MARS [49] achieve high detection rates for the Cyclic Dependency antipattern using static analysis.

In this scenario, dynamic and graph-based approaches offer better insights into runtime behaviors and interactions, making them suitable for diverse environments. While they can complicate data collection and instrumentation, their adaptability is a significant advantage. In contrast, static analysis requires specific tools and deeper integration with the codebase, resulting in higher adaptation costs. Furthermore, the complexity and volume of services pose challenges to the scalability and accuracy of both methods.

In this way, our primary concern is related to detecting microservice antipatterns, which necessitates information from dynamic analysis. This is associated with the lack of antipattern detection strategies or low accuracy of these strategies in the context of microservice applications [33].

This research aims to define stack-agnostic detection strategies for four microservice antipatterns using dynamic analysis: Bottleneck Service, Cyclic Dependency, Not Having API Gateway, and Service Chain. It investigates the limitations of static analysis, explores the advantages of dynamic detection, and addresses current gaps in accuracy and coverage within existing tools.

The other sections are organized as follows. Section 2 presents the theoretical background, highlighting microservices and their antipatterns. Section 3 discusses related work and outlines how our approach differs from theirs. Section 4 presents the research method. Section 5 presents the findings of a literature review, situating our work within the state of the art. Section 6 describes the criteria used for selecting microservice antipatterns targeted for detection, along with the design of the proposed strategies. Section 7 focuses on explaining the detection approach and the four strategies. Section 8 presents the evaluation results in a real-life project. Section 9 discusses threats to validity. Finally, Section 10 presents our conclusions and outlines future work.

## 2 Theoretical Background

### 2.1 Microservices

Microservices Architecture (MSA) structures applications as small, independent services communicating through lightweight protocols [12, 18]. MSA offers several potential benefits, including modeling services around business capabilities, enabling Continuous Integration/Delivery (CI/CD), lightweight communication, small scopes, and fault tolerance through decentralization [10]. Therefore, each microservice can be independently created, tested, deployed, maintained, scaled, and configured.

These characteristics benefit developers, who should be divided into teams with specific purposes [31]. Moreover, the MSA can help create a small, tested code base, as the scope is defined to resolve a particular purpose [55].

Although this style is widely recognized for its benefits and advantages, its implementation is challenging and may result in several problems as well [20]. Challenges include management and availability of microservices on the internal network, communication between services, performance optimization, load balancing, information sharing, and scope delimitation [2]. However, the benefits are still considered more significant than the challenges in several contexts.

In summary, microservices architecture can be a strategic choice for improving the performance of large-scale applications, although it presents complex challenges. Still, it is essential to carefully evaluate its implementation to maximize its benefits and reduce its drawbacks [48].

### 2.2 Antipatterns in Microservices

Architectural challenges can arise in MSA, especially in scenarios involving system evolution or the transition from monolithic architectures [6, 9, 41]. Studies have cataloged common antipatterns related to security, testability, and maintainability [5, 7, 13, 39, 47, 50].

These catalogs focus on defining the antipatterns, highlighting the main issues they cause, describing common scenarios, and presenting approaches to mitigate them. For instance, we specify some common and famous antipatterns [3, 34, 47, 49]:

- **Wrong Cuts**: Services based on technical layers instead of business capabilities;
- **API Versioning**: Lack of semantic versioning in APIs;
- **Cyclic Dependency**: Circular communication between microservices;
- **Hard-coded Endpoints**: Use of fixed addresses and ports for service communication;
- **Shared Persistence**: Multiple services accessing the same database;
- **Shared Libraries**: Reuse of common libraries across services;
- **No API Gateway**: External systems directly communicate with internal services;
- **Hub-like Dependency**: Central service with too many dependencies, risking maintainability and reliability;
- **Mega Service**: A service with too many responsibilities;
- **Nano Service**: Overly small services leading to excessive communication.

The literature on microservice antipatterns presents several methods and tools that focus on specific detection strategies, including rule-based, graph-oriented, Design Structure Matrix (DSM), and model-driven approaches [33, 39]. These strategies can be applied at different stages of the development life cycle, such as code review, static code analysis, during runtime, or observability-based triggers [33].

## 3 Related Work

Several research papers present antipatterns catalogs (Section 3.1) and solutions for detecting antipatterns in microservices (Section 3.2). Next sections also describe the differences between related work and our proposal.

### 3.1 Antipattern Descriptions and Catalogs

There are several catalog proposals focusing on microservice antipatterns and their effects. Taibi and Lenarduzzi [47] identified eleven antipatterns through interviews and qualitative analysis, detailing their impact, relevance, and associated effort.

Muntaz *et al.* [33] conduct a systematic review mapping 105 antipatterns to detection tools, revealing gaps in tool coverage and highlighting undetectable smells.

Zhong *et al.* [56] analyzed six antipatterns and their influence on maintainability using surveys, interviews, and static analysis of 118 microservices. Schirgi and Brenner [44] organized antipatterns by type and discussed their implications on system quality, focusing on awareness of detection techniques.

Bogner*et al.* [6] conducted a systematic literature review, categorizing research on maintainability assurance for service-based and microservice systems, identifying 36 antipatterns, with a focus on limited studies that exclusively focused on microservices.

Parker*et al.* [36] conducted a systematic mapping study examining the visualization of microservice antipatterns at runtime through dynamic analysis, highlighting the absence of a unified tool that combines detection and visualization.

Also, Pigazzini *et al.* [9] detailed and implemented detection strategies for microservice smells, such as cyclic dependencies, hard-coded endpoints, and shared persistence, using static analysis as an extension of the Arcan tool.

Furthermore, it is worth highlighting the presence of the same antipatterns across different studies, demonstrating that these issues are not exclusive to the catalogs mentioned above. For instance, the *Cyclic Dependency* antipattern is defined and appears in several studies, including [1, 3, 6, 19, 36, 38, 40, 52, 56]. Similarly, the *Bottleneck Service* is identified in [1, 36], while *Shared Persistence* is present in [1, 3, 6, 9, 36, 52]. Lastly, the antipattern *Not Having an API Gateway* appears in [3, 6, 36, 53, 54].

The primary reason for focusing the study on these antipatterns is the difficulty in detection that arises when dynamic analysis concepts are involved. Often, authors achieve results through static analysis, but with certain limitations (e.g., limited to a single tool, language, requiring manual action, or not accurately identifying patterns). Despite their significant mention and presence in the literature, as shown in the following section, we highlight some common detection problems and outline potential solutions in the following sections.

## 3.2 Antipatterns Detection

Several studies have investigated the detection of antipatterns using various strategies. Bacchiega et al. [3] propose Aroma, a dynamic analysis tool that detects Not Having an API Gateway, Shared Persistence, and Cyclic Dependency by reconstructing architectural graphs from Zipkin traces. The approach is limited to data collected via Zipkin, interacting directly with the Zipkin API and its resources, such as the collector. It does not support the integration of other observability sources, which constrains its flexibility and restricts detection to what can be inferred from tracing alone. Additionally, it was evaluated only on toy projects, limiting the assessment of its applicability to real-world scenarios.

Farsi et al. [15] detect the Cyclic Dependency antipattern using graph algorithms over a statically defined, manually constructed design-time dependency graph. The approach targets only this single smell and also has the same limitation of relying on toy projects for its assessment.

Pigazzini et al. [39] detect Cyclic Dependency, Hard-Coded Endpoints, and Shared Persistence using a modified version of the Arcan tool for static analysis. The approach focuses on Java-based systems and leverages multiple sources, including Dockerfiles, communication patterns specific to Spring Boot, regular expressions to locate IP addresses within code, and configuration files to identify shared persistence. Despite its thorough code-level inspection, the method is stack-dependent, lacks support for dynamic behavior, and was evaluated only on toy examples.

Tighilt et al. [49] developed MARS, an open-source tool for detecting microservice antipatterns through static analysis. It processes source code and configuration artifacts to build a language-agnostic system model based on a dedicated metamodel. MARS is capable of identifying 16 antipatterns, including Cyclic Dependency, Shared Persistence, and Hardcoded Endpoints. Its detection of Cyclic Dependencies is limited to direct cycles between service pairs due to scalability concerns. The tool was validated on a large dataset of open-source Java microservices. Although MARS covers a wide range of antipatterns, it relies solely on static analysis. While sufficient for cases like Shared Libraries or missing CI/CD, it may overlook hidden interactions. For example, Cyclic Dependencies are detected only when explicit service calls are present, which is barely the usual way of inter-service communication. Also, it ignores indirect or side-effect-based communications.

Fontana et al. [17] introduce Arcan, a static detection tool based on system reconstruction from decompiled Java artifacts, designed to identify Unstable Dependency, Hub-like Dependency, and Cyclic Dependency using communication graphs and statistical validation. Although the term "architecture" in this work refers primarily to the internal structure of a single Java application (rather than the broader architecture of distributed microservices), the tool has been frequently referenced in microservice-related studies [4, 17, 40] as a foundation or inspiration for antipatterns detection methods, but it is all static analysis.

## 4 Research Method

Our research method is structured into two major phases, each designed to systematically support the development and validation of dynamic detection strategies for microservice antipatterns.

The first phase focused on establishing a comprehensive understanding of the current landscape of microservice antipatterns and their corresponding detection strategies. To achieve this, we conducted a Literature Review (LR) to identify existing microservice antipatterns and categorized their detection approaches (e.g., rule-based, static, dynamic, or hybrid methods). Based on this mapping, we ranked them according to the effectiveness and maturity of their existing detection strategies, as well as their susceptibility to dynamic detection when static analysis was not successful. Antipatterns with no or insufficient dynamic detection support were prioritized as primary candidates for the development of our strategies. This process provided a clear scope and justified the selection of specific antipatterns as targets for dynamic analysis.

In the second phase, we designed and implemented a set of detection strategies that were tailored to the selected antipatterns. These were based on dynamic analysis techniques and designed to be independent of the programming language, framework, or specific libraries of the application under evaluation. We first applied our strategies in controlled environments using selected open-source applications, chosen for their modular microservice architecture and the availability of instrumentation points (e.g., tracing tools like Zipkin or Jaeger). Subsequently, we applied it to a more complex application scenario, utilizing a real, open-source microservice application within a realistic and production-like microservice environment. This step aimed to assess the scalability, generalizability, and practical relevance of the proposed strategies.

Throughout both phases, we performed manual validation of our results to verify the accuracy of the detection results. This included comparing the output with expected results based on known system behavior, architectural documentation, or previous manual analyses. This step ensured that the final results are not only theoretically grounded but also practically reliable.

## 5 Microservices Antipatterns Detection in Literature

We conducted a systematic mapping study [37] to identify a set of microservice antipatterns detectable primarily through dynamic analysis. It allowed us to identify and analyze strategies for detecting microservice antipatterns that rely on observing the system using dynamic analysis or based on runtime information. Unlike approaches that analyze source code for antipattern detection, dynamic-based strategies rely on collecting and analyzing metrics and behaviors of the system during or after execution, including monitoring network traffic, resource utilization, and communication patterns between services. The final mapping results are defined in Table 1.

This study examines the detection of antipatterns in microservice architectures using dynamic analysis strategies. We identify the microservice antipatterns detected and discuss the tools and technologies used to support runtime detection.

In addition, the study highlights existing gaps in the literature and identifies limitations of current approaches, providing directions for future research on enhancing antipattern detection in real-world microservice systems.

We developed and refined a search string to identify relevant studies on microservices or service-oriented architectures that address architectural issues, including antipatterns and detection strategies, particularly those involving dynamic or runtime analysis. Additionally, we selected four control papers [4] [57] [1] [40] as benchmarks to validate the search strategy and selection criteria, ensuring the rigor and reliability of the results obtained, and to serve as standards for quality and relevance in evaluating works found in the literature review.

Applied to the title, abstract, and keywords, the final search string is organized into concepts separated by the AND operator and their synonyms separated by the OR operator:

( microservices OR services OR msa ) AND ( architectur* ) AND ( antipattern OR anti-pattern OR smells OR challeng* OR gap OR impediment ) AND ( runtime OR dynamic ) AND ( detection OR identification OR recognition )

To reduce bias and increase the confidence of the final sample of papers, we previously defined inclusion (IC) and exclusion (EC) criteria to establish the review scope, which we also validated against the control papers. The IC focuses on studies that discuss microservice architecture (IC1), address antipatterns and problems in microservice architectures (IC2), present a catalog or identification of antipatterns in microservices (IC3), or discuss current antipattern detection strategies (IC4). The EC, on the other hand, exclude studies that are short papers (less than four pages) (EC1), are proceedings preface or other entries that do not represent full research articles (EC2), are duplicated reports published in different platforms (EC3), lack experimental or empirical evidence (EC4), have not been peer-reviewed (EC5), do not discuss microservice architecture, antipatterns, or antipattern detection (EC6), or were published before 2011, as the term "microservices" was not widely adopted before that year (EC7).

A total of 371 papers were initially returned through a Scopus [1] query. Following a thorough evaluation based on predefined criteria, one paper was excluded for being too short (EC1), 62 were excluded for being poorly indexed (EC2), 247 were excluded for not discussing microservices or antipatterns (EC6), and three were excluded due to being published before 2011 because the term and concept of microservices were not yet widely established. (EC7). This process resulted in 58 remaining papers. Additionally, 41 papers did not meet the key inclusion criteria related to the discussion of microservice architecture and antipatterns together (IC1, IC2), resulting in a final sample of 17 relevant papers.

The evaluated studies reveal that 40% are case studies, which is the most common research method in this field. The articles highlighted 17 microservice applications as examples, with a notable presence of 12 toy projects, primarily used for educational or illustrative purposes. Additionally, these systems are proprietary and used internally within organizations, in contrast to the two open-source applications, SiteWhere [45] and Spinnaker [46], designed for public and community use. This variety of examples underscores the diverse scale and complexity of microservices research, ranging from simplified models to real-world industrial applications.

[1]https://www.scopus.com/

**Table 1: Microservice antipatterns and detection strategies in the Literature**

| Name | Static detection | Runtime detection |
|---|---|---|
| Bottleneck Service | [36][32][1] | - |
| API Versioning | [30][53] | [1] |
| Cyclic Dependency | [57][4][16][30][40][53] | [36][1] |
| Endpoint-based Service Interaction | [36] | - |
| ESB Usage | [30][53] | - |
| Hard-coded endpoints | [30][40][53] | - |
| Innapropriate Service Intimancy | [30][53] | - |
| Microservice Greedy | [30][53] | - |
| Nano Service | [36][32] | - |
| Not Having API Gateway | [4][30][53] | [36] |
| Service Chain | [36] | - |
| Shared Libraries | [30][40][53] | - |
| Shared Persistence | [4][30][53] | [36][1] |
| Too Many Standards | [30][53] | - |
| Wrong Cuts | [30][53] | - |
| Long Chain of Responsability | [32] | - |
| Megaservice | [32] | - |
| Retiring Components | [14] | - |
| Team coupling | [14] | - |
| Hub-like dependency | [57] | - |
| Unbalanced API | [14][1] | - |
| Unstable Dependency | [57] | - |
| Concern Overload | [57] | - |
| Scattered Funcionality | [57] | - |

In total, we identified 24 antipatterns across the selected papers. Each one has its own characteristics and presents a specific detection strategy. Table 1 outlines the relationship of each antipattern and the frequency with which static and dynamic detection methods were observed. Additionally, we note that, as we searched for antipatterns in the context of dynamic analysis, there may be other antipatterns that are explored exclusively with static analysis, which fall outside our mapping scope.

This review highlights significant limitations in existing antipattern detection approaches for microservices. Most strategies rely heavily on static analysis, which, while helpful in evaluating code structure and architectural conformity, struggles to capture the dynamic behavior of microservices during runtime. Dynamic analysis is rarely used in isolation; in a few cases, it is combined with static data extraction to create intermediate models or support validation.

Many studies also focus on simplified systems that fail to reflect the complexity of real-world enterprise architectures, limiting the applicability of their findings. Although some tools claim to support dynamic detection, they often depend on specific frameworks or programming languages, require prior static analysis, or only consider individual services without an architectural overview.

The complexities of building, testing, deploying, and operating microservices applications, as mentioned in Section 2.1, also pose several challenges to conducting empirical studies like in this paper, as they require a significant effort to perform dynamic analysis on entire real-life applications. This may also explain why the majority of static analysis studies in the literature.

While a few antipatterns have been studied using hybrid strategies that include dynamic observation, the lack of approaches that operate entirely at runtime represents a critical research gap. This gap is significant given the emergent behaviors typical in microservices. These findings emphasize the need for more effective and scalable dynamic detection strategies that can operate independently of static assumptions.

## 6 Microservices Antipatterns Selection

We selected antipatterns based on a set of questions designed to evaluate their relevance and feasibility for dynamic detection. These questions helped assess the effectiveness of detecting antipatterns through both static and dynamic analysis, with answers marked as 'YES' or 'NO' based on thorough studies captured in the literature review (Section 5). The questions evaluated the following aspects:

(1) *Q1: Is there evidence of effectiveness in detection through static analysis?* This assesses if static analysis can effectively identify the antipattern by examining the code, without execution.
(2) *Q2: Does the static detection mechanism reveal statically generalizable characteristics?* This evaluates whether the static detection mechanism can be generalized across different code contexts for consistent antipattern identification.
(3) *Q3: Is the antipattern only observable through the execution of the application?* This investigates whether the antipattern's characteristics can only be detected during runtime, highlighting behaviors that manifest during execution.

To select the antipatterns targeted for dynamic detection, all those presented in Table 1 of Section 5 were evaluated based on the questions outlined in this section, as listed in Table 2.

The researchers in this study evaluated each antipattern using the defined questions. The selected ones were those for which we responded 'NO' to question Q1 and Q2, and 'YES' to question Q3. The selected antipatterns are: Cyclic Dependency, Bottleneck, Service Chain, and Not Having API Gateway.

## 7 Detection Approach

### 7.1 Steps

The proposed dynamic detection approach operates independently from the target application, designed to be mostly technology-agnostic, relying only on the system's deployment architecture and infrastructure (e.g., service distribution, orchestration layer, network boundaries, and tools). The primary objective is to collect data from system artifacts, such as logs, request traces, access audits, and network traffic logs, to feed the detection algorithms.

Detection focuses on runtime system behavior, including response times, resource usage, and dependency chains, by analyzing collected logs, traces, and network traffic. This data is processed and used by detection algorithms to identify antipatterns in the microservice architecture. The detection process consists of three stages (as in Figure 1):

(1) Extract data from the system's architecture (several sources)
(2) Process and handling the extracted data

**Table 2: Analysis of Antipatterns for Dynamic Strategies**

| Antipattern Name | Q1 | Q2 | Q3 | Status |
|---|---|---|---|---|
| Bottleneck Service | N | N | Y | Selected |
| API Versioning | Y | Y | N | Excluded |
| Cyclic Dependency | N | N | Y | Selected |
| ESB Usage | N | Y | N | Excluded |
| Hard-coded Endpoints | Y | Y | N | Excluded |
| Inappropriate Service Intimacy | Y | Y | N | Excluded |
| Microservice Greedy | N | Y | N | Excluded |
| Nano Service | Y | Y | N | Excluded |
| Not Having API Gateway | N | N | Y | Selected |
| Service Chain | N | N | Y | Selected |
| Shared Libraries | Y | Y | N | Excluded |
| Shared Persistence | Y | Y | N | Excluded |
| Too Many Standards | Y | Y | N | Excluded |
| Wrong Cuts | N | N | N | Excluded |
| Megaservice | Y | Y | N | Excluded |
| Hub-like Dependency | Y | Y | N | Excluded |
| Unstable Dependency | Y | Y | N | Excluded |
| Concern Overload | Y | Y | N | Excluded |
| Scattered Functionality | Y | Y | N | Excluded |

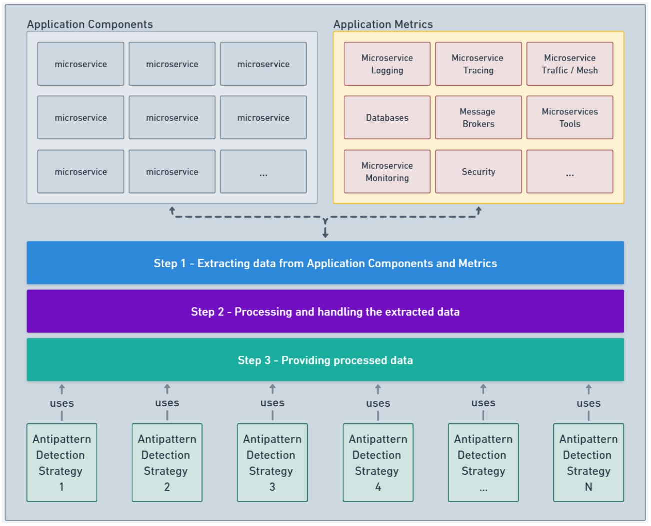
Application Components
microservice
microservice
microservice
microservice
microservice
microservice
microservice
microservice
...
Application Metrics
Microservice Logging
Microservice Tracing
Microservice Traffic / Mesh
Databases
Message Brokers
Microservices Tools
Microservice Monitoring
Security
...
Step 1 - Extracting data from Application Components and Metrics
Step 2 - Processing and handling the extracted data
Step 3 - Providing processed data
uses
uses
uses
uses
uses
uses
Antipattern Detection Strategy 1
Antipattern Detection Strategy 2
Antipattern Detection Strategy 3
Antipattern Detection Strategy 4
Antipattern Detection Strategy ...
Antipattern Detection Strategy N

**Figure 1: Proposed solution in the antipatterns detection architecture**

(3) Provide processed data as input for detection strategies.

The detection steps do not infer particular extraction strategies; data collection and handling are the responsibility of the preparation input process.

### 7.2 Detection Strategies

*7.2.1 Bottleneck Service.* The Bottleneck antipattern refers to a situation in which a single service within a system limits the overall performance and scalability of that system. This occurs when the service handles too many tasks, processes a large volume of requests, or performs complex operations that exceed its capacity [35]. Consequently, it can cause delays or significant slowdowns, forcing other services or components to wait for this service to complete its tasks. This antipattern typically arises when responsibilities are not properly distributed among services or when a service becomes a central point of dependency, leading to inefficiencies and congestion within the system [36].

To detect a bottleneck, it is essential to understand the architecture's limitations. This detection strategy requires two key inputs: a threshold mapping and a list of calls to be analyzed. The threshold mapping defines the maximum allowable response time (in milliseconds) for different protocols. Any values that exceed this threshold will be regarded as indicative of a bottleneck issue. Calls can be gathered from any source or method, as long as they provide the information specified by the threshold mapping. The structure of input parameters is:

```
threshold = {           calls = [{
    "protocol-1": 100,      "origin": "string",
    "protocol-2": 100,      "address": "string",
    "protocol-3": 100,      "protocol": "string",
    "...,                   "target": "string",
    "protocol-N": 100       "duration": 100
}                       }]
```

The main idea of this strategy (Algorithm 1) is to iterate through the list of incoming calls and validate each access between them. For each call, we identify the threshold limit and the call origin to support building a dependency graph. A dependency graph is constructed incrementally by registering relationships between components. If the duration of a call exceeds the threshold limit for its protocol, the call is identified as a bottleneck and added to the result list. This approach allows the detection to operate over any structured call data, as long as it includes the necessary variables.

The algorithm's output is a list of bottleneck problems based on connection pairs. Each entry in the issue list contains essential data from the mapped request, as well as the threshold value that was exceeded. This facilitates the visual representation and mapping of requests, enabling a clear understanding of the connections that surpass predefined thresholds.

**Algorithm 1** Detection Strategy - Bottleneck Service

```
Require: $threshold = protocol, duration$ ▷ Mapping to define the thresholds
Require: $calls$ = [...]                ▷ List of all calls to analyze
1:  $DG \leftarrow$ GenerateEmptyGraph()      ▷ Dependency Graph
2:  $BP \leftarrow \emptyset$   ▷ Initialize Bottleneck Problems with a empty List
3:  for all $call \in calls$ do
4:      if $call.origin$ = empty $\vee$ $call.target$ = empty then
5:          continue          ▷ Skip invalid or unmapped requests
6:      end if
7:      $source \leftarrow call.origin$
8:      $current \leftarrow call.target$
9:      $protocol \leftarrow call.protocol$
10:     $duration \leftarrow call.duration$
11:     $tLimit \leftarrow threshold[protocol]$
12:     if $current \notin DG[source]$ then
13:         $DG[source] \leftarrow +\!\!+[current]$   ▷ Add a new edge between current and source
14:     end if
15:     if $duration > tLimit$ then
16:         $aux \leftarrow (current, source, protocol, duration)$
17:         $BP \leftarrow +\!\!+[aux]$
18:     end if
19: end for
20: return $BP$
```

### 7.2.2 *Cyclic Dependency.*

The cyclic dependency antipattern occurs when services circularly depend on each other, creating a cycle. This leads to increased complexity, maintenance challenges, and potential deadlocks [16, 47]. It can also cause performance issues as services may end up waiting for each other, leading to delays and inefficient resource use. To avoid this, it is essential to design services with clear boundaries and ensure dependencies flow in a one-way direction.

To detect cyclic dependency, the strategy requires a call list input. It is necessary to receive a list of mapped calls from a system, with information obtained from a source. From these calls, the focus is on constructing a dependency graph that includes both direct and indirect dependencies, based on the extracted input. The structure of the input is:

```
calls = [{
    "origin": "string",
    "address": "string",
    "protocol": "string",
    "target": "string"
}]
```

After building a dependency graph, any graph algorithm that can detect cycles in a directed graph (digraph) could be used (as the DetectCycles in Algorithm 2). In our implementation, we use the depth-first traversal algorithm provided by the *Networkx* [23] library. The pseudocode is defined below:

**Algorithm 2** Detection Strategy - Cyclic Dependency

```
Require: $calls$ = [...]                ▷ List of all calls to analyze
1:  $DG \leftarrow$ GenerateEmptyGraph()      ▷ Dependency Graph
2:  for all $call \in calls$ do
3:      if $call.origin$ = empty $\vee$ $call.target$ = empty then
4:          continue          ▷ Skip invalid or unmapped requests
5:      end if
6:      $source \leftarrow call.origin$
7:      $current \leftarrow call.target$
8:      if $current \notin DG[source]$ then
9:          $DG[source] \leftarrow +\!\!+[current]$   ▷ Add a new edge between current and source
10:     end if
11: end for
12: return DetectCycles($DG$)
```

#### 7.2.3 *Not Having API Gateway.*

Refers to the absence of a central API Gateway to manage and route requests from external clients to microservices. This also refers to managing incoming requests externally from the architecture without control (i.e., directly accessing a microservice). Therefore, without an API Gateway, clients must interact directly with multiple services, resulting in decentralized access management and difficulty in maintaining communications (between or external) [4, 47].

In this strategy, our purpose is different from those in the literature. We are not strictly focused on identifying the "absence of an API Gateway," but instead on highlighting any incoming access to microservices that occurs outside a mapped area. For example, there are detection strategies that check if more than one API Gateway or entry point is detected [4]. However, we assume that an architecture may have multiple "entry points" without necessarily being a problem. In other words, the algorithm aims to detect potential "API Gateways" that were not planned or mapped, which is a more comprehensive approach to addressing the semantics involved in this antipattern.

To support this strategy, a different input structure is required compared to the previous ones. Specifically, it is necessary to define a mapping of calls based on a unique identifier, referred to as 'id', which groups all related calls in the exact order they occurred. Additionally, a predefined list of values corresponding to API Gateway components must be provided, as it will be used to validate the detection process. The input's structure is:

```
                          calls = [{
                              "origin": "string",
ValidGateways = [             "address": "string",
    'gateway1',               "id": "string",
    'gateway2',               "protocol": "string",
    '...'                     "target": "string",
]                             "duration": 100
                          }]
```

During the execution of Algorithm 3, a dependency graph is constructed for each call identifier and its associated list of calls, representing the architectural relationships between them. Any entry-point vertex in this graph—i.e., a node with no incoming edges and only outgoing edges—is classified as a potential API Gateway issue.

This algorithm can operate in two distinct ways, following the same logic. First, when no values are provided for the list of API Gateways, the algorithm is designed to detect decentralized incoming calls and highlight them as potential issues. Second, when the desired gateway is mapped, the algorithm validates that it is not acting as the central point of calls to the system. In this way, we can work effectively to address two distinct problems within a single idea.

**Algorithm 3** Detection Strategy - Not Having API Gateway

```
Require: $ValidGateways \leftarrow [...]$
Require: $calls \leftarrow \{'id' : [...]\}$                ▷ calls grouped by its id
1: $AGP \leftarrow \emptyset$ ▷ Initialize API Gateway problems with a empty list
2: $DG \leftarrow$ GenerateEmptyGraph()                 ▷ Dependency Graph
                                          ▷ Initialize Dependency Graph
3: for all $call \in calls$ do
4:     if $call.origin$ = empty $\lor$ $call.target$ = empty then
5:         continue            ▷ Skip invalid or unmapped requests
6:     end if
7:     $source \leftarrow call.origin$
8:     $current \leftarrow call.target$
9:     if $current \notin DG[source]$ then
10:        $DG[source] \leftarrow +\!\!+[current]$            ▷ Add a new edge
   between current and source
11:    end if
12: end for
                                       ▷ Detect API Gateway Problems
13: for all $node \in DG$ do
14:     if $\neg HasIncomingEdges(DG, node) \land node \notin ValidGateways$ then
15:         $AGP \leftarrow +\!\!+[node]$  ▷ Add a new API Gateway Problem
16:     end if
17: end for
18: return $AGP$
```

#### 7.2.4 *Service Chain.*

To detect the Service Chain antipattern, two input values are required: a threshold value to define how many calls are necessary to be considered as a "chain" and a list of calls. The threshold is a simple integer that specifies the minimum number of sequential calls for a chain to be considered problematic. When a list of calls associated with a specific span identifier (referred to as 'id') contains more items than the defined threshold, a new occurrence of the Service Chain is detected. The input structure is:

```
ThresholdCalls = 3

calls = [{
    "origin": "string",
    "address": "string",
    "protocol": "string",
    "target": "string",
    "duration": 100
}]
```

The final result of this algorithm is a mapping structure. Each key represents a specific service chain, and the corresponding value contains the list of calls involved in that chain along with the number of times it has occurred. The algorithm is defined below.

## 8 Results

### 8.1 Evaluation Project

In this work, we chose Spinnaker [46], an open-source, multi-cloud continuous delivery platform for releasing software changes with high velocity and confidence, as the evaluation case. Initially developed by Netflix for AWS, Spinnaker has gained significant adoption and now provides native support for multiple cloud providers, including AWS, OpenStack, Kubernetes, Google Cloud Platform, and Microsoft Azure [42]. We selected Spinnaker due to its real-world relevance and adoption, which makes it a suitable subject for evaluation, unlike toy examples often found in related work. Also, Spinnaker was built using microservices.

**Algorithm 4** Detection Strategy - Service Chain

```
1:  function GenerateChainKey(calls)    ▷ Generate a hashable key to identify the current chain
2:  end function
Require: $ThresholdCalls \leftarrow 2$ ▷ Threshold to consider as a chain
Require: $calls \leftarrow \{'id' : [...]\}$    ▷ calls grouped by its id
3:  $SC \leftarrow \{\}$ ▷ Initialize Service Chain problems with a empty list
4:  $DG \leftarrow$ GenerateEmptyGraph()    ▷ Dependency Graph
                                          ▷ Initialize Dependency Graph
5:  for all $callId \in calls$ do
6:      $count \leftarrow$ Count($calls[callId]$)
7:      if $count >= ThresholdChain$ then, $currentChain \leftarrow calls[callId]$
8:          $chainKey \leftarrow$ GenerateChainKey($currentChain$)
9:          if $chainKey \in SC$ then
10:             $occurrences \leftarrow SC[chainKey].ocurrences + 1$
11:             $SC[chainKey] \leftarrow (currentChain, occurrences)$
12:         else
13:             $SC[chainKey] \leftarrow (currentChain, 1)$
14:         end if
15:     end if
16: end for
17: return $SC$
```

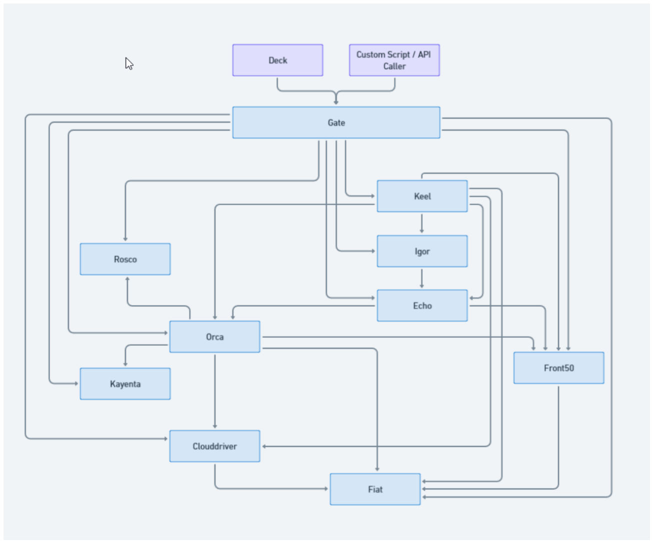

**Figure 2: Architectural definition of Spinnaker, adapted from the official documentation [46]**

We also selected Spinnaker due to the maturity of its documentation. All operational flows and architectural decisions are well-documented in its official resources. Figure 2 presents the architectural diagram of the microservices that compose Spinnaker, along with the communication flows established between them. Spinnaker has 10 microservices: Gate, Keel, Igor, Echo, Orca, Kayenta, Rosco, Clouddriver, Front50, and Fiat 2.

Spinnaker was deployed in a Kubernetes environment and fully configured using its official CLI tool, Halyard, following the steps provided in the official documentation [46]. A warm-up phase was executed by invoking the following functionalities to stimulate inter-service communication as defined by their respective implementations. We executed various operations from the Spinnaker GUI, including creating a project, building a pipeline, executing a pipeline, managing pipeline states, enabling webhooks communication using Jenkins [27], configuring OAuth Authentication [24] with the GitHub provider [21], and more. The idea was to exercise every service (node coverage) in the architecture.

## 8.2 Data Extraction

For data collection, we employed Istio [26] as the service mesh. In a microservices architecture, a service mesh is a dedicated infrastructure layer that manages communication between services [29]. It ensures the reliable delivery of requests across the complex network of services that make up a modern cloud-native application.

Istio relies on the sidecar proxy pattern, where a lightweight proxy (Envoy) is deployed alongside each service instance. This architecture enables transparent interception of all inbound and outbound traffic, allowing fine-grained observability, routing, and security without modifying application code [28, 43].

## 8.3 Validation

To validate the detection strategies, each of them was implemented using the Python programming language [51]. In general, the process involved reading the Sidecar Envoy logs provided by Istio [26], processing their values to populate the expected inputs for each strategy, and generating graphical visualizations using the Matplotlib [25] and NetworkX [23] libraries.

*8.3.1 Cyclic Dependency.* Figure 3 illustrates a dependency graph of the microservices in Spinnaker, constructed from values collected through dynamic analysis. All edges in the graph are directed, indicating communication between microservices. Red edges represent a detected cycle. These cycles are not necessarily present within the same request trace. Instead, the cycle was identified by analyzing multiple requests collectively, without considering any unique trace identifiers. The nodes in the figure represent: (1) the names of the microservices, all prefixed with spin, and (2) the IP address 192.168.15.8, which symbolizes the address of the Jenkins server, installed outside the Kubernetes cluster where Spinnaker is deployed.

In total, five communication cycles were identified among Spinnaker's microservices. After removing both the 'spin-' prefix and the '.spinnaker' suffix, and standardizing names with capital initials, the following cycles were observed:

- `Clouddriver` ↔ `Fiat`
- `Echo` → `Front50` ↔ `Fiat` ↔ `Igor` → `Echo`
- `Igor` → `Front50` ↔ `Fiat` ↔ `Igor`
- `Igor` ↔ `Fiat`
- `Front50` ↔ `Fiat`

These cycles highlight potential feedback loops in the system's architecture, identified through dynamic analysis across multiple request traces, without requiring all nodes to appear in a single

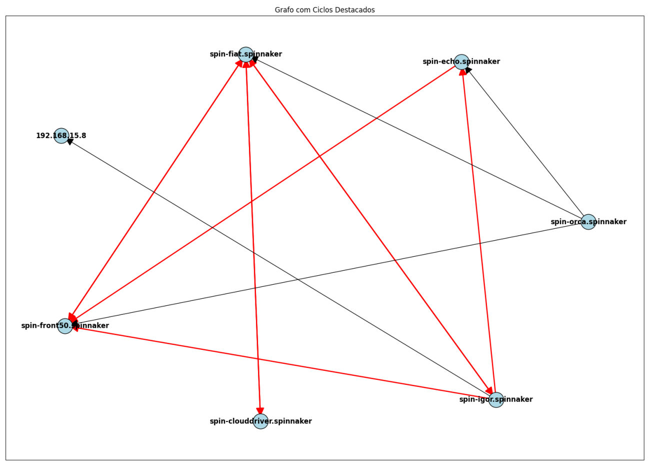

**Figure 3: Result of Cyclic Dependency strategy**

execution path. We also note that these cycles are not present in the documented architecture, indicating that they are unintentional.

*8.3.2 Bottleneck Service.* For the bottleneck detection, a threshold was configured based on the expected limits for the test scenarios. In this case, the threshold for HTTP and HTTPS requests was set to 100 milliseconds. Based on this configuration, the detection result is presented in Figure 4.

This value represents an average over multiple requests. The identified bottleneck occurs in the communication from Igor → Echo. If the threshold had been set lower, the outcome would likely have been different. Nevertheless, the 100-millisecond mark was chosen as it is considered a good response time between service requests.

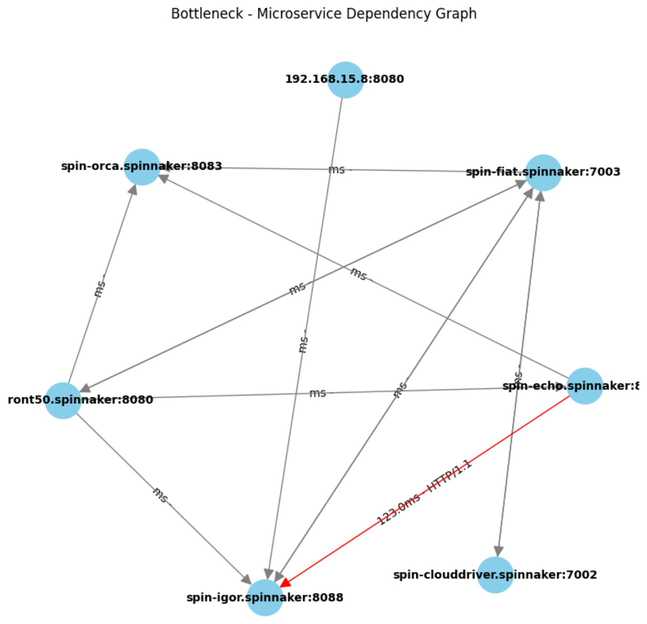

**Figure 4: Result of Bottleneck Service strategy**

Other tools could enhance the measurement of time consumption and bottlenecks in microservices, but they were not utilized in this study. One limitation is the default behavior of the Istio service mesh sidecar, which only captures traffic at the HTTP level. This limitation means that internal calls or operations using non-HTTP protocols, such as database drivers or messaging systems like Kafka, are not fully visible in Istio's telemetry.

Consequently, the time spent in these internal components may be underrepresented or completely omitted from the trace data. While there are specialized sidecars and instrumentation approaches designed for specific technologies, they were beyond the scope of this work. This limitation restricts the ability to measure end-to-end latency with high granularity, potentially leading to an incomplete identification of performance bottlenecks.

*8.3.3 Not having API Gateway.* For this detection strategy, the application Gate was established as the central API gateway for input into the strategy. All evaluations were carried out based on the previously mapped list of service calls. The dynamic analysis revealed no direct access to microservices from entry points other than Gate. Although Gate receives requests from various sources, including external clients and the Kubernetes internal gateway, none of these calls bypass the applications. This observation indicates that the architectural behavior illustrated in Figure 2 aligns with the anticipated design principles.

Concerns initially arose regarding the Fiat microservice, which is responsible for managing authentication and authorization. It queries a user's access permissions for accounts, applications, and service accounts. When the OAuth feature [24] with GitHub [21] was activated, concerns arose that the login confirmation process might bypass Gate. However, this behavior was thoroughly verified, confirming that all related traffic indeed passed through the designated gateway as expected.

*8.3.4 Service Chain.* During the detection process for the Service Chain antipattern, we executed workflows designed to interact with the system holistically. However, we did not identify any service call chains when applying a threshold of 3 hops. This threshold was chosen to filter out simple point-to-point communications and concentrate specifically on more extended sequences of interactions that may indicate potential performance or design issues, such as cascading service calls.

While communication between microservices was present, it did not form a chain of three or more distinct services participating in the same request or trace. When we lowered the threshold to 2, we detected numerous interactions, confirming that the services are indeed communicating with each other. However, these instances do not meet the minimum criteria to be classified as a chain and therefore do not provide evidence of the antipattern.

This outcome suggests two possibilities. First, the application may not exhibit the Service Chain antipattern under normal operational conditions. Second, and more likely given the nature of microservice systems, the testing scenarios may not have adequately stimulated the complex interactions between services. Service chains often emerge under realistic workloads or production-like behavior; therefore, the absence of a detected chain may be due to limitations in test coverage, data volume, or a lack of concurrent or chained business processes triggered during execution.

Furthermore, some services may be communicating outside the context of a single trace, which hinders the detection of chains that span across asynchronous or decoupled flows. This raises an important point about the necessity of complete trace propagation across services and the need for consistent instrumentation to be in place.

## 9 Threats to Validity

This study recognizes some potential threats to its validity. In terms of internal validity, the identification of antipatterns depended exclusively on runtime data; however, they are automatically collected and processed. Furthermore, the thresholds and parameters chosen for the detection strategies were established based on anticipated performance targets, yet they may not adequately capture all operational scenarios.

Regarding construct validity, the instrumentation approach employed Istio's service mesh and primarily focused on HTTP traffic. This methodology may have missed relevant interactions occurring over other protocols, possibly leading to incomplete detection in certain instances.

Regarding external validity, the evaluation was conducted on a single real-world system, Spinnaker. Although this system is well-established and widely used, it may not fully represent the entire range of microservice architectures and deployment contexts. Additionally, although all the automated approach is executed in silico, i.e., with no human intervention, the detected antipatterns were not externally validated by system maintainers, independent experts, or through historical operational data. This absence of external validation limits the generalizability and corroboration of the findings.

Regarding reliability, the automated procedures are all available as notebooks at Zenodo, ensuring transparency. From data collection to antipatterns detection, almost everything is automated, except for the manual exercising of the application (which can be automated in the future) to generate logs and the interpretation of results. This way, all the steps throughout the study can be reviewed externally.

## 10 Conclusions

In this work, we developed an approach with four strategies for detecting antipatterns in microservices, chosen based on a review and validation of existing literature. We focused on dynamic detection methods for the following antipatterns: Cyclic Dependency, Not Having an API Gateway, Service Chain, and Bottleneck Service. These strategies are independent of the technology stack used in the applications, relying solely on the environment and potential data sources regarding the system.

We developed a proof-of-concept solution that can evaluate runtime data collected from Spinnaker [46], an open-source multi-cloud management platform widely used by real-world companies. Additionally, we identified actual issues within Spinnaker, such as unmapped cyclic dependencies, as illustrated in Figure 2 from the tool's official website.

Unlike the approaches found in the literature, which often remain limited to synthetic projects or restricted scopes, this work applies well-established detection strategies to a real-world, production-grade system (Spinnaker). This practical application in a realistic scenario constitutes the main original contribution of this study, enabling the evaluation of the strategies under real operational conditions and allowing the identification of concrete limitations and impacts. As a result, we can observe aspects that are rarely reported in related work.

To address validation concerns, it is essential to assess the performance of the proposed detection strategies in other real-world applications, regardless of whether they are commercial or open-source. We also recommend exploring alternative methods for extracting runtime data beyond the Service Mesh discussed in this work, including data from database communications, messaging systems, proprietary protocols, Application Performance Monitoring tools, and more. Additionally, our validation relied on interactions with the system under test, which may not have triggered all possible execution paths. As a result, some issues could have remained undetected simply because the corresponding flows were not exercised during experimentation. Future work should consider a more comprehensive system stimulation to ensure broader coverage of potential antipattern occurrences.

For future work, we propose validating these detection strategies in other architectural contexts, not limited to microservices, as the selected antipatterns are not desirable in other contexts as well. For example, we can identify architectural issues in hybrid environments that incorporate microservices, serverless architectures, monoliths, and different approaches. Furthermore, we suggest developing a centralized tool to manage detections, visualize findings, and log historical occurrences, thereby enhancing the governance of antipatterns in software architecture. Antipatterns can be detected in hybrid environments that combine MSA with Serverless, Monoliths, and other approaches. Moreover, we suggest the development of a centralized tool to manage detections, visualize them, and log the occurrence history, thereby improving the governance of antipatterns in software architecture.

## ARTIFACT AVAILABILITY

The artifact regarding the Spinnaker logs and a notebook with Python implementation of the strategies, as well as the charts generation, is available at Zenodo: https://doi.org/10.5281/zenodo.15506161.

## REFERENCES

[1] Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. 2022. Using Microservice Telemetry Data for System Dynamic Analysis. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 29–38. doi:10.1109/SOSE55356.2022.00010

[2] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 44–51. doi:10.1109/SOCA.2016.15

[3] Paolo Bacchiega, Ilaria Pigazzini, and Francesca Arcelli Fontana. 2022. Microservices smell detection through dynamic analysis. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 290–293. doi:10.1109/SEAA56994.2022.00052

[4] Paolo Bacchiega, Ilaria Pigazzini, and Francesca Arcelli Fontana. 2022. Microservices smell detection through dynamic analysis. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 290–293. doi:10.1109/SEAA56994.2022.00052

[5] David Baum, Jens Dietrich, Craig Anslow, and Richard Müller. 2018. Visualizing Design Erosion: How Big Balls of Mud are Made. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. 122–126. doi:10.1109/VISSOFT.2018.00022

[6] Justus Bogner, Tobias Boceck, Matthias Popp, Dennis Tschechlov, Stefan Wagner, and Alfred Zimmermann. 2019. Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 95–101. doi:10.1109/ICSA-C.2019.00025

[7] John Brondum and Liming Zhu. 2012. Visualising architectural dependencies. In *2012 Third International Workshop on Managing Technical Debt (MTD)*. 7–14. doi:10.1109/MTD.2012.6226003

[8] William J. Brown. 1998. *AntiPatterns: Refactoring software, Architectures, and projects in crisis*. John Wiley amp; Sons.

[9] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. 2018. Migrating towards Microservices: Migration and Architecture Smells. In *Proceedings of the 2nd International Workshop on Refactoring* (Montpellier, France) *(IWoR 2018)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3242163.3242164

[10] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2019. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 150 (2019), 77–97. doi:10.1016/j.jss.2019.01.001

[11] Jamilah Din, Anas Bassam AL-Badareen, and Yusmadi Yah Jusoh. 2012. Antipatterns detection approaches in Object-Oriented Design: A literature review. In *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*. 926–931.

[12] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 195–216. doi:10.1007/978-3-319-67425-4_12

[13] Ulf Eliasson, Antonio Martini, Robert Kaufmann, and Sam Odeh. 2015. Identifying and visualizing Architectural Debt and its efficiency interest in the automotive domain: A case study. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 33–40. doi:10.1109/MTD.2015.7332622

[14] H. Fang, Y. Cai, R. Kazman, and J. Lefever. 2023. Identifying Anti-Patterns in Distributed Systems With Heterogeneous Dependencies. *Proceedings - IEEE 20th International Conference on Software Architecture Companion, ICSA-C 2023* (2023), 116–120. doi:10.1109/ICSA-C57050.2023.00035 cited By 0.

[15] Hassan Farsi, Driss Allaki, Abdeslam En-Nouaary, and Mohamed Dahchour. 2022. A Graph-based Solution to Deal with Cyclic Dependencies in Microservices Architecture. In *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)*. 254–259. doi:10.1109/FiCloud57274.2022.00042

[16] H. Farsi, D. Allaki, A. En-Nouaary, and M. Dahchour. 2022. A Graph-based Solution to Deal with Cyclic Dependencies in Microservices Architecture. *Proceedings - 2022 International Conference on Future Internet of Things and Cloud, FiCloud 2022* (2022), 254–259. doi:10.1109/FiCloud57274.2022.00042 cited By 0.

[17] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. 2016. Automatic Detection of Instability Architectural Smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 433–437. doi:10.1109/ICSME.2016.33

[18] Martin Fowler and James Lewis. 2014. https://martinfowler.com/articles/microservices.html

[19] E. Gaidels and M. Kirikova. 2020. Service Dependency Graph Analysis in Microservice Architecture. *Lecture Notes in Business Information Processing* 398 LNBIP (2020), 128–139. doi:10.1007/978-3-030-61140-8_9 cited By 5.

[20] Martin Garriga. 2018. Towards a Taxonomy of Microservices Architectures. In *Software Engineering and Formal Methods*, Antonio Cerone and Marco Roveri (Eds.). Springer International Publishing, Cham, 203–218.

[21] GitHub Inc. 2024. GitHub: Where the world builds software. https://github.com/. Accessed: 2025-07-06.

[22] Mouna Hadj-Kacem and Nadia Bouassida. 2019. Towards a taxonomy of bad smells detection approaches. *ICSOFT 2018 - Proceedings of the 13th International Conference on Software Technologies* (2019), 164 – 175. doi:10.5220/0006869201640175

[23] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. *Proceedings of the 7th Python in Science Conference (SciPy)* 2008 (2008), 11–15. https://conference.scipy.org/proceedings/scipy2008/paper_2/

[24] D. Hardt. 2012. The OAuth 2.0 Authorization Framework. Internet Engineering Task Force (IETF) Request for Comments. https://datatracker.ietf.org/doc/html/rfc6749 Accessed: 2025-07-06.

[25] John D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95.

[26] Istio Authors. 2024. Istio: Connect, secure, control, and observe services. https://istio.io/. Accessed: 2025-07-06.

[27] Jenkins Project. 2024. Jenkins: The leading open source automation server. https://www.jenkins.io/. Accessed: 2025-07-06.

[28] Matt Klein. 2024. Envoy: Modern, High Performance C++ Proxy. https://www.envoyproxy.io/. Accessed: 2025-07-06.

[29] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 122–1225. doi:10.1109/SOSE.2019.00026

[30] L. Liu, Z. Tu, X. He, X. Xu, and Z. Wang. 2021. An Empirical Study on Underlying Correlations between Runtime Performance Deficiencies and 'Bad Smells' of Microservice Systems. *Proceedings - 2021 IEEE International Conference on Web Services, ICWS 2021* (2021), 751–757. doi:10.1109/ICWS53863.2021.00103 cited By 2.

[31] Bahgat Mashaly, Sahar Selim, Ahmed H. Yousef, and Khaled M. Fouad. 2022. Privacy by Design: A Microservices-Based Software Architecture Approach. In *2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. 357–364. doi:10.1109/MIUCC55081.2022.9781685

[32] R. Matar and J. Jahic. 2023. An Approach for Evaluating the Potential Impact of Anti-Patterns on Microservices Performance. *Proceedings - IEEE 20th International Conference on Software Architecture Companion, ICSA-C 2023* (2023), 167–170. doi:10.1109/ICSA-C57050.2023.00044 cited By 0.

[33] Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. 2021. A systematic mapping study on architectural smells detection. *Journal of Systems and Software* 173 (2021), 110885. doi:10.1016/j.jss.2020.110885

[34] H. Mumtaz, P. Singh, and K. Blincoe. 2021. A systematic mapping study on architectural smells detection. *Journal of Systems and Software* 173 (2021). doi:10.1016/j.jss.2020.110885 cited By 11.

[35] Francis Palma and Naouel Mohay. 2015. A study on the taxonomy of service antipatterns. In *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*. 5–8. doi:10.1109/PPAP.2015.7076848

[36] G. Parker, S. Kim, A.A. Maruf, T. Cerny, K. Frajtak, P. Tisnovsky, and D. Taibi. 2023. Visualizing Anti-Patterns in Microservices at Runtime: A Systematic Mapping Study. *IEEE Access* 11 (2023), 4434–4442. doi:10.1109/ACCESS.2023.3236165 cited By 1.

[37] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology* 64 (2015), 1–18.

[38] I. Pigazzini, F. Arcelli Fontana, and A. Maggioni. 2019. Tool support for the migration to microservice architecture: An industrial case study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11681 LNCS (2019), 247–263. doi:10.1007/978-3-030-29983-5_17 cited By 14.

[39] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. 2020. Towards Microservice Smells Detection. In *Proceedings of the 3rd International Conference on Technical Debt* (Seoul, Republic of Korea) *(TechDebt '20)*. Association for Computing Machinery, New York, NY, USA, 92–97. doi:10.1145/3387906.3388625

[40] Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. 2020. Towards Microservice Smells Detection. In *Proceedings of the 3rd International Conference on Technical Debt* (Seoul, Republic of Korea) *(TechDebt '20)*. Association for Computing Machinery, New York, NY, USA, 92–97. doi:10.1145/3387906.3388625

[41] Francisco Ponce. 2021. Towards Resolving Security Smells in Microservice-Based Applications. In *Advances in Service-Oriented and Cloud Computing*, Christian Zirpins, Iraklis Paraskakis, Vasilios Andrikopoulos, Nane Kratzke, Claus Pahl, Nabil El Ioini, Andreas S. Andreou, George Feuerlicht, Winfried Lamersdorf, Guadalupe Ortiz, Willem-Jan Van den Heuvel, Jacopo Soldani, Massimo Villari, Giuliano Casale, and Pierluigi Plebani (Eds.). Springer International Publishing, Cham, 133–139.

[42] Pethuru Raj and Anupama Raman. 2018. *The Hybrid Cloud: The Journey Toward Hybrid IT*. Springer International Publishing, Cham, 91–110. doi:10.1007/978-3-319-78637-7_5

[43] Louis Ryan, Shriram Rajagopalan, et al. 2018. *Istio: A Platform for Connecting, Securing, and Managing Microservices.* Technical Report. Google, IBM, Lyft. https://istio.io/latest/docs/concepts/what-is-istio/ Accessed: 2025-07-06.

[44] Thomas Schirgi and Eugen Brenner. 2021. Quality Assurance for Microservice Architectures. In *2021 IEEE 12th International Conference on Software Engineering and Service Science (ICSESS)*. 76–80. doi:10.1109/ICSESS52187.2021.9522227

[45] SiteWhere Project. 2025. SiteWhere: Open Platform for the Internet of Things (IoT). https://github.com/sitewhere/sitewhere. Accessed: 2025-07-06.

[46] Spinnaker Authors. 2024. Spinnaker: Open Source Continuous Delivery Platform. https://spinnaker.io/. Accessed: 2025-07-06.

[47] Davide Taibi and Valentina Lenarduzzi. 2018. On the Definition of Microservice Bad Smells. *IEEE Software* 35, 3 (2018), 56–62. doi:10.1109/MS.2018.2141031

[48] Johannes Thönes. 2015. Microservices. *IEEE Software* 32, 1 (2015), 116–116. doi:10.1109/MS.2015.11

[49] Rafik Tighilt, Manel Abdellatif, Imen Trabelsi, Loïc Madern, Naouel Moha, and Yann-Gaël Guéhéneuc. 2023. On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. *Journal of Systems and Software* 204 (2023), 111755. doi:10.1016/j.jss.2023.111755

[50] Catia Trubiani, Anne Koziolek, Vittorio Cortellessa, and Ralf Reussner. 2014. Guilt-based handling of software performance antipatterns in palladio architectural models. *Journal of Systems and Software* 95 (2014), 141–165. doi:10.1016/j.jss.2014.03.081

[51] Guido Van Rossum and Python Software Foundation. 2023. Python: An interpreted, high-level and general-purpose programming language. https://www.python.org/. Accessed: 2025-07-06.

[52] Andrew Walker, Dipta Das, and Tomas Cerny. 2020. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Applied Sciences* 10, 21 (2020). doi:10.3390/app10217800

[53] A. Walker, D. Das, and T. Cerny. 2020. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences (Switzerland)* 10, 21 (2020), 1–20. doi:10.3390/app10217800 cited By 12.

[54] A. Walker, D. Das, and T. Cerny. 2021. Automated Microservice Code-Smell Detection. *Lecture Notes in Electrical Engineering* 739 LNEE (2021), 211–221. doi:10.1007/978-981-33-6385-4_20 cited By 3.

[55] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. 2020. A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software* 170 (2020), 110798. doi:10.1016/j.jss.2020.110798

[56] Chenxing Zhong, Huang Huang, He Zhang, and Shanshan Li. 2022. Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation. *Software: Practice and Experience* 52, 12 (2022), 2574–2597. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3138 doi:10.1002/spe.3138

[57] Chenxing Zhong, Huang Huang, He Zhang, and Shanshan Li. 2022. Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation. *Software: Practice and Experience* 52, 12 (2022), 2574–2597. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3138 doi:10.1002/spe.3138