applied sciences

MDPI

Article

Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study

Andrew Walker, Dipta Das and Tomas Cerny *

Computer Science, Baylor University, One Bear Place #97141, Waco, TX 76798, USA; andrew_walker2@baylor.edu (A.W.); dipta_das1@baylor.edu (D.D.)

* Correspondence: tomas_cerny@baylor.edu

Received: 23 September 2020; Accepted: 27 October 2020; Published: 3 November 2020

Abstract: Microservice Architecture (MSA) is becoming the predominant direction of new cloud-based applications. There are many advantages to using microservices, but also downsides to using a more complex architecture than a typical monolithic enterprise application. Beyond the normal poor coding practices and code smells of a typical application, microservice-specific code smells are difficult to discover within a distributed application setup. There are many static code analysis tools for monolithic applications, but tools to offer code-smell detection for microservice-based applications are lacking. This paper proposes a new approach to detect code smells in distributed applications based on microservices. We develop an MSANose tool to detect up to eleven different microservice specific code smells and share it as open-source. We demonstrate our tool through a case study on two robust benchmark microservice applications and verify its accuracy. Our results show that it is possible to detect code smells within microservice applications using bytecode and/or source code analysis throughout the development process or even before its deployment to production.

Keywords: microservice; cloud-computing; code smells; code-analysis; quality assurance

1. Introduction

Microservices are becoming the preeminent architecture in modern enterprise applications [1]. There are several advantages to utilizing this architecture, which have led to its rise in popularity [2]. The distributed nature of a microservice-based application allows for greater autonomy of developer units. While this provides greater flexibility for faster delivery, improved scalability, and benefits in existing problem domains [3], it also presents the opportunity for code smells to be more readily created within the application. This is especially true since distinct teams manage different distributed modules of the overall system.

Code smells [4] are anomalies within codebases. They do not necessarily impact the performance or correct functionality of an application. They are patterns of poor programming practice and deteriorate program quality. They can affect a wide range of quality attributes in a program including reusability, testability, and maintainability. If code smells go unchecked in a microservice-based application, the benefits of using a distributed development process can be mitigated. It is therefore crucial that the code smells in an application are appropriately detected and managed [4,5].

Microservices present a unique situation when it comes to code smells due to the distributed nature of the application. Microservice-specific code smells often focus on inter-module issues rather than an intra-module issue. Traditional code-smell detecting tools cannot detect code smells between discrete modules, so these issues go unchecked during the development process. This paper shows that, when we augment static code analysis to recognize enterprise development constructs, we can effectively detect code smells in distributed microservice applications.

We share a case study targeting eleven recently identified code smells for this architecture to demonstrate our approach. Furthermore, we develop a prototype code smell detector for microservices and share it with the community as open source. Our prototype is based on code analysis and recognizes Java code along with Enterprise Java platform constructs and standards [6–10]. Next, it identifies the eleven microservice code smells targeted in this paper.

The rest of the paper is as follows. Section 2 assesses related work for code smells detection and the shortcomings for distributed systems. Section 3 introduces the code smells used in this paper. Section 4 describes the static code analysis of enterprise systems. Section 5 proposes our solution for automatic code-smell detection for microservices. Section 6 verifies our approach on two existing microservice benchmark applications. Lastly, Sections 7 and 8 conclude the work and highlight future perspectives.

2. Related Work

Although first defined by Fowler [4] as problems in code caused by poor design decisions, code smells have evolved in the world of modern software engineering to encompass much more. Code smells can be defined as "*characteristics of the software that may indicate a code or design problem that can make software hard to evolve and maintain*" [11].

Code smells are not necessarily a problem but rather an indicator of a problem. They can be seen as code structures that indicate a violation of fundamental design principles and negatively impact design quality [12]. Urgent maintenance activities prioritizing feature delivery over code quality often lead to code smells [13]. Thus, code smells are codebase anomalies and do not necessarily impact the performance or correct functionality; they are poor programming practice patterns. Code smells can affect a wide range of areas in a program, including reusability, testability, and maintainability.

Gupta et al. [14] underlined that it is essential to identify and control code smells during the design and development stages to achieve higher code maintainability and quality. However, even if developers are not invested in fixing them, code smells do matter to the overall software maintainability [15–17]. Furthermore, if left unchecked, code smells can begin to impact the overall system's architecture [18]. Code smells can be deceptive and hide the true extent of their 'smelliness' and even carry into further refactorings of the code [18,19]. Frequently code smells are also related to anti-patterns [20] in an application.

Code-smell correction is a necessary process for developers [21], but it is often pushed aside. A study by Sae-Lim et al. [21] found that the most prevalent factor towards developers addressing code smells is the importance of the issue and the relevance of the issue to the task they were working on. Another study by Peters et al. [22] found that, while developers are frequently aware of the code smells in their application, they do not care about actively fixing them. Most of the time, the code smells are fixed accidentally through unrelated code refactoring [4]. Much has been done in research to address the problem of code smells, and many studies have been performed, exploring how code smells are created [19], managed [23], and fixed in industry [24].

Tahir et al. [25] studied how developers discussed code smells in stack exchange sites and found that these sites work as an informal crowd-based code smell detector. Peers discuss the identification of smells and how to get rid of them in a specific given context. Thus, the question is how to detect and eliminate them in a given context. They found that the most popular smells discussed between developers are also shown to be most frequently covered by available code analysis tools. It is also noted that, while Java support is the broadest, other platforms, including C#, JavaScript, C++, Python, Ruby, and PHP, are lacking in support. Concerns were also raised that there is a missing classification for how harmful smells are on a given application.

Some researchers would argue that developers do not have the time to fix all smells. For instance, Gupta et al. [14] identified 18 common code smells and the driving power of these code smells to improve the overall code maintainability. The effect is that developers could refactor one of the smells

with higher driving power, rather than address all smells in an application, and still significantly improve code maintainability.

One of the first attempts at automatic code-smell detection came from Emden and Mooden [26], who defined an automated code-smell detection tool for Java. Since then, the field of code-smell detection has continued to grow. Code smell tools have been developed for *high level design* [27–29], *architectural smells* [30–32], and *language-specific code smells* [33–35], measuring not just code smells but also the quality [14,36] of the application. The field of automatic code-smell detection continues to evolve with an ever-changing list of code smells and languages to cover.

It is common to identify code smells in monolithic systems using code-analysis. For instance, tools such as SpotBugs [37], FindBugs [38], CheckStyle [39], or PMD [40] can detect code patterns that resemble a code smell. Anil et al. [41] recently analyzed 24 code smells detection tools. While the tools correctly mapped the code smells in an application, they are limited to a single codebase, and so they become antiquated as modern software development tends towards microservice architectures.

While extensive research has been done to define and detect code smells in a monolithic application, little has been done for distributed systems [42]. It would be possible for a developer to run code-smell detection on each of the individual modules, but this does not address any code smells specific to microservice architecture.

In a distributed environment, in particular microservices, there have been multiple code smells identified. In one study [43], these smells include improper module interaction, modules with too many responsibilities, or a misunderstanding of the microservice architecture. Code smells can be specific to a certain application perspective, including the *communication perspective*, or in the *development and design process* of the application. These smells can be detected manually, which usually requires assessing the application and a basic understanding of the system, but this demands considerable effort from the developers. With code analysis instruments, smells can be discovered almost instantly and automatically with no previous knowledge of the system required. However, we are aware that no tool can detect the code anomalies that can exist between discrete modules of a microservice application.

3. Microservice Code Smell Catalogue

For this paper's purposes, we reused the definition of eleven microservice specific code smells from a recent exploratory study by Taibi et al. [43]. They used existing literature and interviews with industry leaders to distill and rank these eleven code smells for microservices. The code smells are briefly summarized as follows:

- *ESB Usage (EU)*: An Enterprise Service Bus (ESB) [2] is a way of message passing between modules of a distributed application in which one module acts as a service bus for all of the other modules to pass messages on. There are pros and cons to this approach. However, in microservices, it can become an issue of creating a single point of failure, and increasing coupling, so it should be avoided. An example is displayed in Figure 1.
- *Too Many Standards (MS)*: Given the distributed nature of the microservice application, multiple discrete teams of developers often work on a given module, separate from the other teams. This can create a situation where multiple frameworks are used when a standard should be established for consistency across the modules.
- *Wrong Cuts (WC)*: This occurs when microservices are split into their technical layers (presentation, business, and data layers). Microservices are supposed to be split by features, and each fully contains their domain's presentation, business, and data layers.
- *Not Having an API Gateway (NAG)*: The API gateway pattern is a design pattern for managing the connections between microservices. In large, complex systems, this should be used to reduce the potential issues of direct communication.

- *Hard-Coded Endpoints (HCE)*: Hardcoded IP addresses and ports are used to communicate between services. By hardcoding the endpoints, the application becomes more brittle to change and reduces the application's scalability.
- *API Versioning (AV)*: All Application Programming Interfaces (API) should be versioned to keep track of changes properly.
- *Microservice Greedy (MG)*: This occurs when microservices are created for every new feature, and, oftentimes, these new modules are too small and do not serve many purposes. This increases complexity and the overhead of the system. Smaller features should be wrapped into larger microservices if possible.
- *Shared Persistency (SP)*: When two microservice application modules access the same database, it breaks the microservice definition. Each microservice should have autonomy and control over its data and database. An example is provided in Figure 2.
- *Inappropriate Service Intimacy (ISI)*: One module requesting private data from a separate module also breaks the microservice definition. Each microservice should have control over its private data. An example is given in Figure 3.
- *Shared Libraries (SL)*: If microservices are coupled with a common library, that library should be refactored into a separate module. This reduces the fragility of the application by migrating the shared functionality behind a common, unchanging interface. This will make the system resistant to ripples from changes within the library.
- *Cyclic Dependency (CD)*: This occurs when there is a cyclic connection between calls to different modules. This can cause repetitive calls and also increase the complexity of understanding call traces for developers. This is a poor architectural practice for microservices.

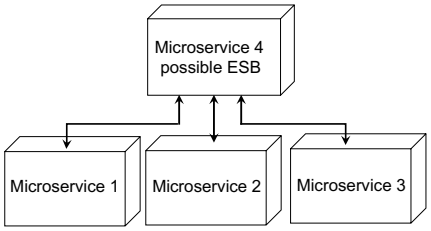

Figure 1. Example ESB usage.

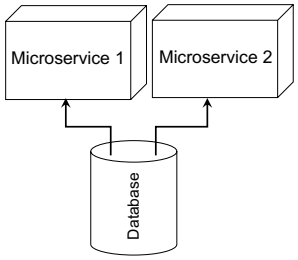

Figure 2. Shared persistency.

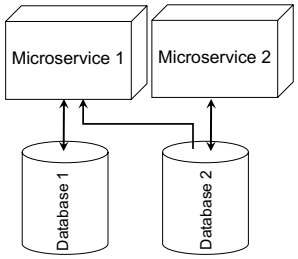

Figure 3. Inappropriate service intimacy.

To highlight the gap in microservice code smells, we took a list of existing state-of-the-art architecture-specific code-smell detection tools from a previous and recent study [42] and verified whether they detect any of the previously mentioned microservice-specific code smells. We chose these tools as they were compiled to study the existing state of the art of architecture smell detection tools and were shown to meet a minimum threshold of documentation and information about the tool. We compile our results in Table 1. The closest tool, called Arcan, was recently published [44] and only detected three of the smells.

Table 1. Comparison of architectural code smell detection tools.

Tools / Smell	1 EU	2 MS	3 WC	4 NAG	5 HCE	6 AV	7 MG	8 SP	9 ISI	10 SL	11 CD
AI Reviewer [45]											X
ARCADE [46]											X
Arcan [44]					X			X			X
Designite [47]											X
Hotspot Detector [48]											X
Massey Architecture Explorer [49]											X
MSA Nose	X	X	X	X	X	X	X	X	X	X	X
Sonargraph [50]											X
STAN [51]											X
Structure 101 [52]											X

(EU), ESB Usage; (MS), Too Many Standards; (WC), Wrong Cuts; (NAG), Not Having an API Gateway; (HCE), Hard-Coded Endpoints; (AV), API Versioning; (MG), Microservice Greedy; (SP), Shared Persistency; (ISI), Inappropriate Service Intimacy; (SL), Shared Libraries; (CD), Cyclic Dependency.

4. Code Analysis and Extension for Enterprise Architectures

Static code analysis [53] is one of the most important software development topics, primarily its role in detecting bugs in a system. However, as with most other problem domains, there exist gaps around enterprise architectures. The two static code analysis processes, source code and bytecode analysis, ultimately create a representation of the application. This is done through several processes, including recognizing components, classes, methods, fields, or annotations, tokenization, and parsing, which produce graph representations of the code. These include Abstract Syntax Trees (AST), Control-Flow Graphs (CFG) [54–56], or Program Dependency Graphs (PDG) [57,58].

Bytecode analysis [59] uses the application's compiled code and is useful in uncovering endpoints, components, authorization policy enforcements, classes, and methods. It can augment or derive CFG or AST [60–62]. However, the disadvantage is that not all languages have a bytecode.

In *source code analysis* [63], we parse through the source code of the application without having to compile it into an immediate representation. Many approaches exist to do this; however, most tools tokenize the code and construct trees, including AST [57,58], CFG [54–56], or PDG [64,65].

However, limits exist with these representations in encapsulating the complexity of enterprise systems. To mitigate the shortcomings of existing static code analysis techniques on enterprise systems, we augment the current techniques to recognize enterprise standards [6,66]. A more realistic representation of the enterprise application can be constructed with aid from either source code analysis or bytecode analysis. This includes a tree representation, detection of the system's endpoints, and a communication map's construction. These augmented representations along with metadata have been successful in other problem domains including security [67], networking [68] and semantic clone detection [69].

The following section shows how we can use these representations and metadata for a more thorough analysis of code smells in microservice applications.

5. Proposed Solution to Detect Code Smells

Previous studies have shown that, without readily available information about the code smells and easy integration into the software development pipeline, the smells are often not addressed. Thus, our approach uses static-code analysis for fast and easily-integrated reports on the code smells in an application. To cover a wide variety of possible issues within a microservice application, as well as the different concerns (application, business, and data) that the identified smells cover, we must statically analyze a couple of different areas of an application. Our approach specifically involves the Java Enterprise Editions platform because of its rich standards for enterprise development. In fact, we include Spring Boot (https://spring.io/projects/spring-boot) and Java EE (https://docs.oracle.com/javaee). However, alternative standard adoptions exist also for other platforms. Next, these standards can promote to UML [70,71], which shows platform-independence. Furthermore, extending our tool to another language would be trivial since we utilize an intermediate representation for analysis, as explained below.

The core of our solution is an automated derivation of *a centralized view of the application*, also sometimes referred to as Software Architecture Reconstruction [72–74]. To begin with, we individually analyze each microservice in the application. Once each module is fully analyzed, it can be aggregated into a larger service mesh. Then, the full detection can be done on the aggregated mesh.

Our analysis process's first step is to generate *a graph of interaction* between the different microservices. This involves exploring each microservice for a connection to another microservice, which is usually done through a REST API call. The inter-microservice communications are realized using a two-phase analysis: scanning and matching. In the first phase, we scan each microservice to list all the REST endpoints and their specification metadata. This metadata contains the HTTP type, path, parameter, and return type of the endpoint. Additionally, the server IP addresses (or their placeholders) are resolved by analyzing application configuration files that accompany system modules. These IP addresses, together with the paths, define the fully-qualified URLs for each endpoint. We further analyze each microservice to enumerate all REST calls along with request URLs and similar metadata. We list these endpoints and REST calls based solely on static code analysis, where we leverage the annotation-based REST API configuration commonly used in enterprise frameworks. We match each endpoint with each REST call across different microservice modules based on the URL and metadata in the second phase. During matching, URLs are generalized to address different naming of path variables across different microservice modules. Each resultant matching pair indicates inter-microservice communication.

Afterward, the underlying *dependency management configuration* file is analyzed for each of the different microservices (e.g., pom.xml file for maven). This allows us to find the dependencies and libraries used by each of the applications. Lastly, the application configuration, where developers define information such as the port for the module, the databases it connects to, and other relevant environment variables for the application, is analyzed.

The overall architecture of our proposed solution is shown in Figure 4. The resource service module takes the path of the source files and extracts metadata from those files. These metadata are then fed into the entity service and API service modules, which produce descriptions of entities and definitions of API endpoints, respectively. The REST discovery service module takes the definitions of the API endpoints and resolves inter-microservice communications. Once the processing of each module is done, we begin the process of code-smell detection. In the following text, we provide details relevant to each particular smell and its detection.

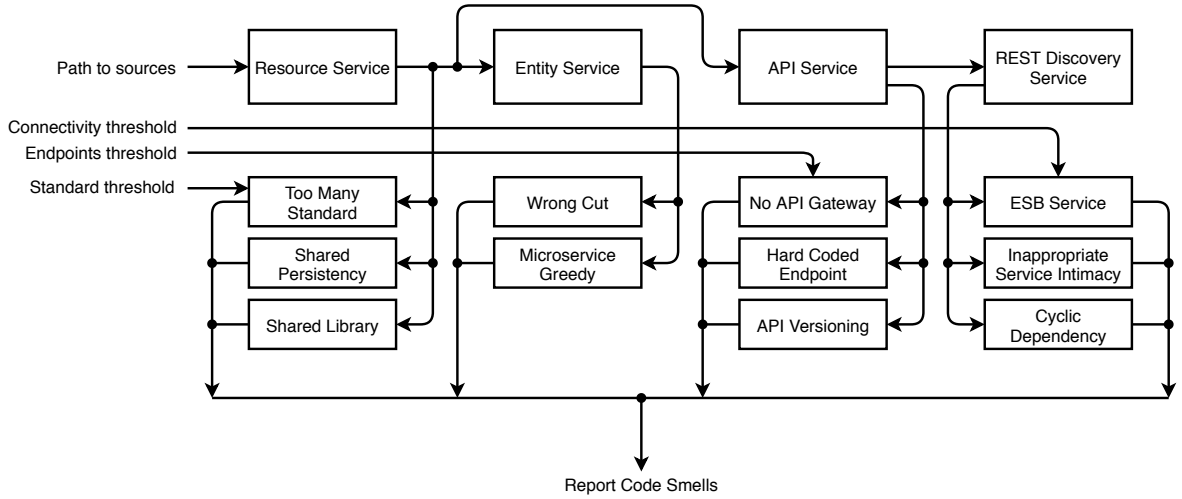

Figure 4. MSANose architecture diagram.

5.1. ESB Usage

To detect if an Enterprise Service Bus (ESB) is being used, we start by tallying up all of the incoming and outgoing connections within each module of the system. We define an ESB as a module with a high, almost outlier, number of connections and a relatively equal number of incoming and outgoing connections. Additionally, an ESB should connect to nearly all the modules.

5.2. Too Many Standards

Detecting if too many standards are used in an application is a tricky problem since it is entirely subjective on how many standards is "too many." Additionally, there are very good reasons developers would choose different standards for different system modules, including speed, available features, and security [67]. We tally the standards used for each of the layers of the application (presentation, business, and data). The user can configure how many standards is too many for each of the respective sections.

5.3. Wrong Cuts

Wrong cuts depend on the business logic and, therefore, nearly impossible to automatically detect without extrapolating a deep understanding of the business domain. However, we would expect to see an unbalanced distribution of artifacts within the microservices along with the different layers of the application (presentation, business, and data). To detect an unbalance presentation microservice, we look for an abnormally high number of front-end artifacts (such as HTML/XML documents for JSP). For the potentially wrong cut business microservices, we look for an unbalanced number of service objects, and, lastly, for wrongly cut data microservices, we look for an unbalanced number of entity objects. To find microservices with this smell, we look for outliers in the number of the specified artifacts within each microservice. Next, we report the possibility of a wrongly cut microservice to the user. We define an outlier count of greater than two times the standard deviation away from the average count of the artifacts in each microservice, which is seen in Equation (1).

$$2 * \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{X})^2}{n-1}} \tag{1}$$

5.4. Not Having an API Gateway

Not having an API gateway is something that is not always possible to determine from code analysis alone. It is especially the case as cloud applications increasingly rely on routing

frameworks such as AWS API Gateway (https://aws.amazon.com/api-gateway/), which uses an online configuration console and is not discoverable from code analysis, to handle routing API calls. In the study by Taibi et al. [43], it was found that developers could adequately manage up to 50 distinct modules without needing to rely on an API gateway. For this reason, if the scanned application has more than 50 distinct modules, we include a warning message in the final report that they will likely want to use an API gateway. This is not classified as an error but rather a suggestion for best practice.

5.5. Shared Persistency

Shared persistency happens when two or more modules of the application share the same relational database. An example of this can be seen in Figure 2. This is detected by parsing the application's configuration files and finding the submodules' persistence settings location. For example, in a Spring Boot application, the application YAML file is parsed for the datasource URL. Then, the persistence of each module is compared to the others to find shared datasources.

5.6. Inappropriate Service Intimacy

Inappropriate service intimacy can appear in a couple of different ways. It is defined as one microservice requesting the private data of another microservice. An example of this can be seen in Figure 3. One of the ways we detect this is as a variant of the shared persistency problem. Here, instead of sharing a datasource between two or more modules, a module is directly accessing another's datasource in addition to its own. This is detected in a similar way as shared persistency; however, once a duplicate datasource is found, if the module also has its own private datasource, then it is an instance of inappropriate service intimacy. Another way in which we search for inappropriate service intimacy is to look for two modules with the same entities. If one of those modules is only modifying/requesting the other's data, we define it as inappropriate service intimacy.

5.7. Shared Libraries

To detect shared libraries, the dependency management files are scanned for each module of the application to locate all shared libraries. Clearly, some shared outside libraries will exist among the microservices; however, the focus should be on any in-house libraries. Developers can then decide to extract into a separate module if necessary to bolster the application against changes in the libraries.

5.8. Cyclic Dependency

To find all cycles between modules, we use a modified depth first search [75]. First, we extract the REST communication graph for the microservice mesh. In the graph, each vertex represents a microservice, and each edge represents a REST API call. Then, we run our cyclic dependency detection algorithm on the graph. We maintain a recursive stack of vertices while traversing the graph. Since the graph is unidirectional (client to server), we mark it as a cycle if a vertex already exists in the stack. The algorithm is presented in Listing 1 and Listing 2.

Listing 1. Find all cycles.

```
boolean isCyclic() {

// Mark all the vertices as not visited
// and not part of recursion stack
boolean[] visited = new boolean[V];
boolean[] recStack = new boolean[V];

// Call the recursive helper function to
// detect cycle in different DFS trees
for (int i = 0; i < V; i++)
if (isCyclicUtil(i, visited, recStack))
return true;

return false;
}
```

Listing 2. A Helper Function to Find All Cycles.

```
boolean isCyclicUtil(int i,
boolean[] visited,
boolean[] recStack) {

// Mark the current node as visited
// and part of recursion stack
if (recStack[i])
return true;

if (visited[i])
return false;

visited[i] = true;

recStack[i] = true;
List<Integer> children = adjList.get(i);

for (Integer c: children)
if (isCyclicUtil(c,visited,recStack))
return true;

recStack[i] = false;

return false;
}
```

5.9. Hard-Coded Endpoints

Hard-coded endpoints are found during the bytecode analysis phase of the application. Using the bytecode instructions, we can peek at the variable stack and see what parameters are passed into the function calls used to connect to other microservices. In the case of Spring Boot, for example, we look at any calls from RestTemplate. We then link the passed address back to any parameters passed to the function or any class fields to find the path parameters used. Our system tests for both hardcoded port numbers and hardcoded IP addresses. Both should be avoided to make scalability of the system easier in the future.

5.10. API Versioning

To find unversioned APIs in the application, we first find all fully qualified paths for the application. For example, the Spring Boot code in Listing 3 would produce a fully qualified API path of "/api/v1/users/login". Each API path is matched against a regular expression pattern `.*/v[0-9]+(.?[0-9]*).*` to locate the unversioned paths. All unversioned APIs are reported back to the user.

Listing 3. Example Spring Boot API.

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController{

@Autowired
private UserService userService;

@Autowired
private TokenService tokenService;

@PostMapping("/login")
public ResponseEntity<?> getToken(...){
return ResponseEntity.ok(...);
}
...
}
```

5.11. Microservice Greedy

To find superfluous microservices, we find a couple of different metrics for each microservice. This includes front-end files (e.g., HTML, CSS, and javaScript), service objects, and entity objects in the application. Then, we find outliers, if any exist, as potential microservice greedy modules. We define outliers in the same way as when finding a wrongly cut microservice using Equation (1). However, we focus only on those that are outliers due to being undersized, as opposed to too large.

6. Case Study

Based on the described approach, we developed a prototype tool called MSANose (https://github.com/cloudhubs/msa-nose). This tool accepts Java-based microservice projects and performs static analysis of microservice modules. From the individual modules, it extracts the interaction patterns. It combines the partial results from each module to derive a single overall holistic view of the distributed system.

MSANose utilizes the system's derived centralized perspective to perform the eleven distinct detections mentioned above. The tool's outcome is a report containing a list of microservice code smell patterns with references to the offending modules and code. In the next section, we describe a case study to demonstrate our approach and the developed prototype tool MSANose.

Recent efforts [76] to catalog microservice testbed applications have found a lack of applications that adhere to the guidelines for testbeds outlined by Aderaldo et al. [77]. We processed the benchmarks list and concluded that these are insufficient in size, nature, and state to study code smells. We introduce two testbed systems to verify the effectiveness of our application.

6.1. Train Ticket

To test our application, we chose to run it on an existing microservices benchmark, the Train Ticket Benchmark [78]. We chose this benchmark since it is a reasonable size for a microservice application and would provide a good test of all of our application conditions. This benchmark was designed as a model of real-world interaction between microservices in an industry environment. Next, it is one of the largest microservice benchmarks available. This benchmark consists of 41 microservices and contains over 60,000 lines of code. It uses either Docker (https://www.docker.com/) or Kubernetes (https://kubernetes.io/) for deployment which relies on either NGINX (https://www.nginx.com/) or Ingress (https://kubernetes.io/docs/concepts/services-networking/ingress/) for routing.

Before running our application on the testbed system, we manually analyzed the testbed for each of the eleven microservice code smells. This was performed as follows: first, through manual tracing of REST calls, and then through the cataloging of entities and endpoints within the system. We utilized two student researchers familiar with research into enterprise systems to ensure our manual assessment accuracy. We show the results of our manual assessment in Column 2 of Table 2.

After taking the results manually, we ran our application on the testbed system. Column 3 of Table 2 is a quick overview of the results from running our application on the testbed. The application took just 10 s to run on a system with an Intel i7-4770k and 8 Gb of RAM. This includes the average time (taken over ten runs) to analyze the source code and compile bytecode of the testbed application. An individual breakdown of the times for each of the code smells available in Column 4 of Table 2.

Table 2. Case study in Train Ticket benchmark [78].

Smell	Manual	MSANose	Time (ms)
ESB Usage	No	No	1
Too Many Standards	No	No	213
Wrong Cuts	0	2	1487
Not Having an API Gateway	No	No	1
Hard-Coded Endpoints	28	28	1
API Versioning	76	76	1981
Microservice Greedy	0	0	2093
Shared Persistency	0	0	123
Inappropriate Service Intimacy	1	1	1617
Shared Libraries	4	4	237
Cyclic Dependency	No	No	1
Total			7755

We tested the testbed for potential ESBs with a connectivity threshold on the microservices of 80%, which could be adjusted by the user. Our application reported no potential ESBs, which matched our earlier manual assessment of the system.

Further, the testbed is built with Spring Boot, MongoDB, and uses static hosting for the front-end. We confirmed this through manual verification and the publicly available design documents (https://github.com/FudanSELab/train-ticket) for the system. Our application correctly identified these standards. We test for too many standards specifically within each layer (presentation, business, and data), and so no layer was beyond our threshold of two standards.

Wrong cuts were one of the most difficult to discern within the testbed. Since the testbed used static files for the front-end, we could only detect microservices that were wrongly split based on the business and data layers. We manually searched for wrongly cut microservices, but the testbed was designed well and did not have any that we could manually determine were wrongly cut. However, our application found two potential wrongly cut microservices in the system. To find the wrong cuts on the business and data layers, we looked at the distribution of services and entities within the microservices. We know that any microservice that is wrongly cut with on the business layer would have excess amounts of services, and any cut wrongly on the data layer would have excess entities.

To discover the entity objects as part of the system's data model, we utilized standard annotations [6] for entities and *@Document* annotation for MongoDB entities. We then matched any object we found in the system against the previously known entities and on name similarity match. To calculate name similarity, we used the WS4J project (https://github.com/Sciss/ws4j). Based on this, we could determine that object was an entity in its microservice.

The two potential wrong cuts we found were both microservices with an unusually high number of entity objects. Since the application only has 41 microservices, we did not report a need to use an API gateway; however, we can verify using the publicly available design document, as seen in Figure 5, that an API gateway is nonetheless used.

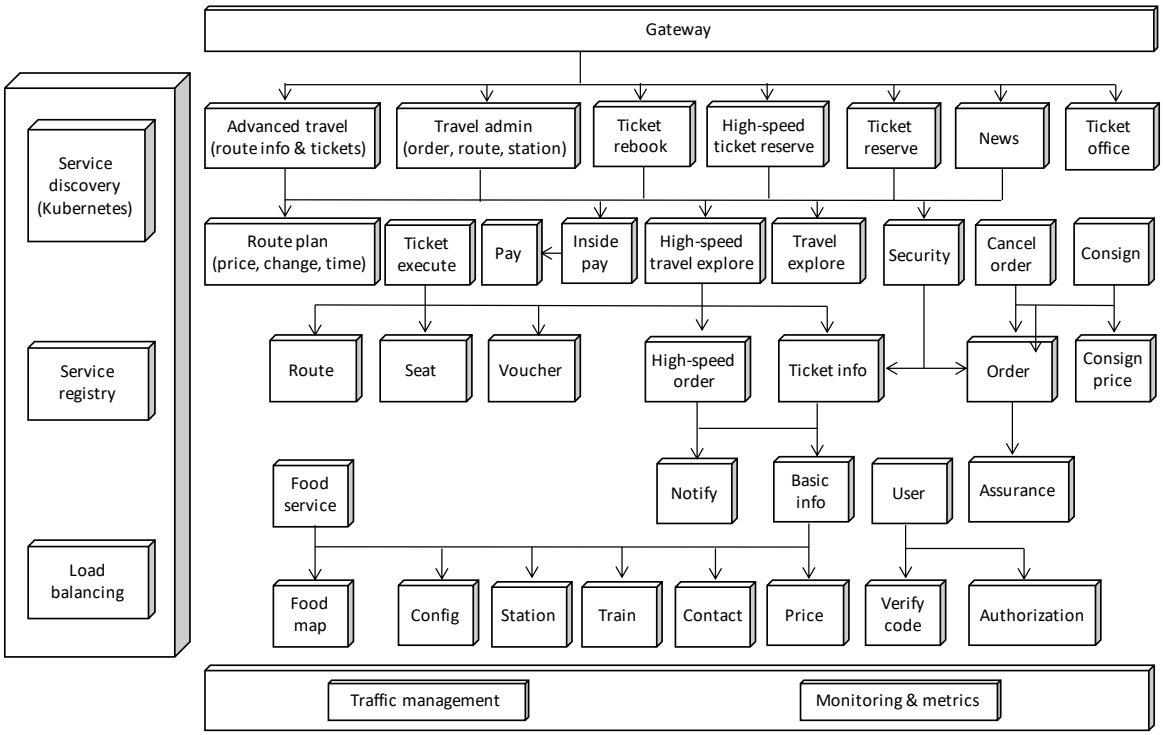

Figure 5. Train Ticket testbed architecture diagram.

Of the 28 hard-coded endpoints we found, all of them were hardcoded port numbers. This is still an issue for the application, as it makes it significantly harder to scale and change the microservices later.

Of the 76 API endpoints that were unversioned, most were found in the admin modules, and some miscellaneous, non-data endpoints throughout the other modules.

There were no shared persistencies among the microservices, as each microservice had its own database. We could manually verify one inappropriate service intimacy in the system, which our tool correctly found. The modules *ts-admin-route-service* and *ts-route-service* both use the exact same entities, and *ts-admin-route-service* solely requests/modifies the private data of *ts-route-service* as opposed to using its own data.

Of the four shared libraries our application found, only one was widespread. The library was also not a public library, but an in-house library developed for the system. This is a perfect example of a library that is too coupled to the microservices and should be refactored. Lastly, no cyclic dependencies were found, which matches both what we found in our manual testing and the architecture diagram in Figure 5.

6.2. Teacher Management System

The Teacher Management System (TMS) (https://github.com/cloudhubs/tms2020) is an enterprise application developed at Baylor University for Central Texas Computational Thinking, Coding, and Tinkering to facilitate the Texas Educator Certification training program. The TMS application consists of four microservices: user management system (UMS), question management system (QMS), exam management system (EMS), and configuration management system (CMS). All of the microservices are developed using the Spring Boot framework and structured into the controller, service, and repository layers. It uses Docker for application packaging, Docker-compose (https://docs.docker.com/compose/) for deployment, and NGINX for routing.

Similar to the first case study, we manually analyzed the testbed for each of the eleven microservice code smells. Then, we ran our MSANose application on the TMS testbed. It took around 2 s to run the benchmark on a 2.9 GHz Intel Core i9 computer with 32 GB of RAM. Table 3 shows the results of the

manual assessment and results from MSANose side-by-side. An individual breakdown of the times for each of the code smells is listed in Column 4 of Table 3.

Table 3. Case study on TMS benchmark.

Smell	Manual	MSANose	Time (ms)
ESB Usage	No	No	1
Too Many Standards	No	No	66
Wrong Cuts	0	0	279
Not Having an API Gateway	Yes	No	1
Hard-Coded Endpoints	2	2	1
API Versioning	62	62	546
Microservice Greedy	0	0	271
Shared Persistency	0	0	60
Inappropriate Service Intimacy	0	0	1
Shared Libraries	2	2	47
Cyclic Dependency	No	No	1
Total			1074

We ran our MSANose tool on the TMS testbed for potential ESBs with a connectivity threshold on the microservices of 80%. Our tool reported no potential ESBs, which can be verified using the REST communication diagram shown in Figure 6. At first look, one might think of the CMS as an ESB since most of the outgoing connections are from CMS. However, ESBs are simply routers with some intelligence like data conversion and filtering. Thus, ESBs typically have a high number of incoming and outgoing connections compared to microservices. In Figure 6, we can see that CMS does not have any incoming connection. This indicates CMS is not an ESB and our tool produced the correct result for ESB detection.

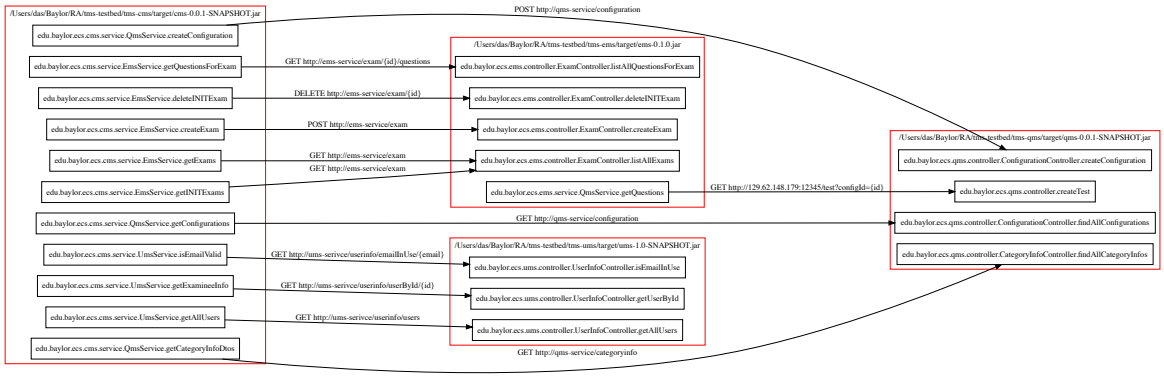

Figure 6. Inter microservice REST communications of TMS application.

Our manual assessment found that the testbed is built with Spring Boot, PostgreSQL with JPA is used for persistence, and static hosting is used for the front-end. Our application correctly identified these standards. We ran our tool for too many standards for each of the presentation, business, and data layers. No layer was beyond our threshold of two standards.

Since the application has only four microservices, we did not explicitly report for an API gateway. However, our manual assessment found that no API gateway was used. Our tool did not report any possible wrong cuts. We manually checked for access amount of services in the business layers and access amount of entities in the data layers. Our manual assessment confirmed that the application was well designed, and there were no possible wrong cuts.

Our tool found two hard-coded endpoints, both of them were in EMS and pointing to QMS. Both of these endpoints were hardcoded IP, which makes it harder to scale the application. From our manual assessment, as shown in Figure 6, we found QMS is also accessed from CMS using a non-hard

coded endpoint, which is considered to be the best practice. Thus, those hardcoded endpoints were probably mistakenly left unnoticed during the application development process.

The MSANose tool did not report any shared presidencies for the testbed, and we confirmed it by identifying that each microservices had its own database. There was no inappropriate service intimacy. We manually checked all entities and did not found any pair of entities that are exactly the same.

Our tool identified 62 unversioned API endpoints, which were verified by our manual assessment. The further assessment found that the application did not use any API versioning at all, which is critical for client-side code stability. Two shared libraries were identified, which matches the count of our manual assessment. However, both of those libraries are related to the Spring Boot framework. Thus, it is one of the false-positive warnings reported by our tool and can be ignored safely. Lastly, no cyclic dependency was found, which matches our manual assessment.

Our tool correctly analyzed both testbed systems and successfully identified the applications' microservice code smells. Code smells do not always break the system or cause system-crashing bugs, but they are problems nonetheless and are indicators of poor programming practice. As a system grows organically, as the testbed applications have done over the past couple of years, these smells can easily work their way into the system. Our tool can assist developers in locating code smells in their enterprise application, as well as providing a catalog of the smells and their common solutions as they attempt to fix them.

6.3. Validity threats

One of the main validity threats is the three code smells microservice greedy, wrong cuts, and too many standards. While these code smells are specifically defined as to what they are, they do not have an established system for detection or solution [43]. We used our discretion, along with knowledge of enterprise architecture, to determine how our application would detect those, but it is ultimately up to interpretation. Below, we also address the internal and external threats to validity.

6.3.1. Internal Threats

We ran our application ten times to validate that the times we record in Tables 2b and 3b for our system's running time to avoid an unusual system deviation. Our application is tested against manually gathered results. To mitigate potential error when collecting the results, we had multiple researchers gather the results and matched them. We used these results to validate the correctness of our system.

Our application uses several thresholds for determining different code smells, which are documented with the results. These thresholds are required to estimate the severity of certain code smells and set our detection algorithms' tolerance. We used 80% connectivity threshold for detecting *ESB Usage*. A threshold of 50 microservices was used to report *not having an API gateway*. The default values of those thresholds were originally proposed by Taibi et al. [44] through an extensive survey among industry specialist. However, these thresholds could be adjusted by the user, which would produce different results.

6.3.2. External Threats

Our application was run on two open-source benchmark applications that were similar to real-world conditions to make our test as applicable as possible. Our analysis utilizes established enterprise standards. Thus, if applications follow the best practice standards, they are analyzable by our system.

Both of the benchmark applications used in the case study are primarily written in Java. However, microservice architecture usually follows polyglot programming styles. We utilized bytecode and source code analysis in our tool to show that it can support both interpreted and compiled languages. We used Java Parser and Javassist for parsing Java source code and bytecode, respectively. Similar parsing tools are available for other modern languages; for example, Python and Golang have a built

in parser package to obtain AST from the source code. Provided that a language has an appropriate parser, our tool can be extended to support a wide variety of languages used in MSA.

We designed generic interfaces to analyze and detect code smells. In our case study, we chose our first benchmark application (Train Ticket) randomly from a list of exclusively designed applications for benchmarking [78]. Then, we implemented those interfaces for Spring boot and enterprise Java since the chosen benchmark follows these standards. To verify our prototype is not application-specific, we chose our second benchmark application (Teacher Management System) that follows similar standards. However, there might exist different standards in other languages. To address this, we need to implement the interfaces for those specific standards.

Modern cloud-native microservices are usually packaged as containers and deployed using orchestration platforms such as Kubernetes, Cloud Foundry, Docker Swarm, etc. During containerization, source codes are not typically included; only the compiled artifacts such as JAR or EXE files are added into the containers. It is still possible to perform bytecode analysis for containerized microservices by extracting the bytecode artifacts (e.g., JAR file) from container layers [53]. For this approach, we also need to analyze the deployment configuration files to identify service names associated with each microservices [53]. However, additional hard-coded dependencies in container images might require further research to identify them properly. In addition, for compiled languages, source code analysis is not possible within a containerized environment.

7. Future Trends

The area of microservice verification has only recently begun to be thoroughly explored. This means that an enterprise system's typical problem domains, such as security [67], data constraints, and networking [79], have only a surface-level examination for verification. The problem domain for code smells is not different. Although this work is based on established code smells from industry advice and examination [43], there exists the possibility to expand the pool of code smells for microservice-based applications. For monolithic systems, there exists hundreds of code smells in a multitude of languages. Definitions of those code smells can be adjusted to make them appropriate for MSA through an extensive survey among industry specialists. For example, the *Artificial Coupling* and *Hidden Dependencies* smells described in [4] can be interpreted for microservice level coupling instead of class-level coupling. In addition, similar to the study described in [80], a taxonomy of code smells can be done exclusively for MSA.

Our implementation has a clear separation between the metadata extraction and code-smell detection, where each detection algorithm is implemented in separate modules. Thus new detection mechanisms can be easily plugged in as a discreet module without affecting the existing metadata extraction and detection algorithms.

Similarly, this research could be expanded into other languages and enterprise standards. In addition, exploration for containerized microservices along with rigorous deployment configuration analysis can be done for cloud-native applications [53].

Our open-source tool can be integrated into the software development lifecycle. For instance, it can be added to the CI/CD pipeline to run an automatic screening test before performing the deployment. Further, it can be used to accelerate the code review process. These adoptions will reduce the manual efforts and human errors of code reviewers and DevOps engineers resulting in a shortened release and update cycle of a microservice applications along with improved code quality.

8. Conclusions

In this paper, we discuss the nature of code smells in software applications. Code smells, which may not break the application in the immediate time-frame, can cause long-lasting problems for maintainability and efficiency later on. Many tools have been developed which automatically detect code smells in applications, including ones designed for architecture and overall design of a system. However, none of these tools adequately address a distributed application's needs, specifically a

microservice-based application. To address these issues, we draw upon previous research into defining microservice specific code smells to build an application capable of detecting eleven unique microservice-based code smells. Our prototype application, MSANose, is open-source and available at https://github.com/cloudhubs/msa-nose. We ran our application on two established microservice benchmark applications and compared our results to manually gathered ones. We show that it is possible, through static code analysis, to analyze a microservice-based application and accurately derive microservice-specific code smells.

For future work, we plan to assess more application testbeds. Moreover, we plan to continue our work on integrating the Python platform to our approach since there are no platform-specific details, and most of the enterprise standards apply to across platforms. We also plan to detect code clones in distributed enterprise microservice applications in future work.

Author Contributions: Conceptualization, A.W.; methodology, A.W.; software, A.W. and D.D.; validation, A.W., D.D., and T.C.; formal analysis, A.W. and D.D.; investigation, A.W. and D.D.; resources, A.W. and D.D.; data curation, A.W. and D.D.; writing—original draft preparation, A.W.; writing—review and editing, A.W., D.D., and T.C.; visualization, A.W. and D.D.; supervision, T.C.; project administration, T.C.; and funding acquisition, T.C. All authors have read and agreed to the published version of the manuscript.

Funding: This material was based on work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. NGINX, Inc. The Future of Application Development and Delivery Is Now Containers and Microservices Are Hitting the Mainstream. 2015. Available online: https://www.nginx.com/resources/library/app-dev-survey/ (accessed on 27 March 2020).
2. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual Understanding of Microservice Architecture: Current and Future Directions. *SIGAPP Appl. Comput. Rev.* **2018**, *17*, 29–45. [CrossRef]
3. Walker, A.; Cerny, T. On Cloud Computing Infrastructure for Existing Code-Clone Detection Algorithms. *SIGAPP Appl. Comput. Rev.* **2020**, *20*, 5–14. [CrossRef]
4. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2018.
5. Yamashita, A.; Moonen, L. Do developers care about code smells? An exploratory survey. In Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 14–17 October 2013; pp. 242–251.
6. DeMichiel, L.; Shannon, W. JSR 366: Java Platform, Enterprise Edition 8 Spec. 2016. Available online: https://jcp.org/en/jsr/detail?id=342 (accessed on 27 March 2020).
7. DeMichiel, L.; Keith, M. JSR 220: Enterprise JavaBeans Version 3.0. Java Persistence API. 2006. Available online: http://jcp.org/en/jsr/detail?id=220 (accessed on 27 March 2020).
8. Bernard, E. JSR 303: Bean Validation. 2009. Available online: http://jcp.org/en/jsr/detail?id=303 (accessed on 27 March 2020).
9. DeMichiel, L. JSR 317: JavaTM Persistence API, Version 2.0. 2009. Available online: http://jcp.org/en/jsr/detail?id=317 (accessed on 27 March 2020).
10. Hopkins, W. JSR 375: JavaTM EE Security API. Available online: https://jcp.org/en/jsr/detail?id=375 (accessed on 27 March 2020).
11. Fontana, F.A.; Zanoni, M. On Investigating Code Smells Correlations. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011; pp. 474–475.
12. Suryanarayana, G.; Samarthyam, G.; Sharma, T. *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2014.

13. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. When and Why Your Code Starts to Smell Bad. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 403–414.
14. Gupta, V.; Kapur, P.; Kumar, D. Modelling and measuring code smells in enterprise applications using TISM and two-way assessment. *Int. J. Syst. Assur. Eng. Manag.* **2016**, 7. [CrossRef]
15. Yamashita, A.; Moonen, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 682–691.
16. Moonen, L.; Yamashita, A. Do Code Smells Reflect Important Maintainability Aspects? In Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 306–315. [CrossRef]
17. Yamashita, A.; Counsell, S. Code smells as system-level indicators of maintainability: An empirical study. *J. Syst. Softw.* **2013**, *86*, 2639–2653. [CrossRef]
18. Macia, I.; Garcia, J.; Daniel, P.; Garcia, A.; Medvidovic, N.; Staa, A. Are automatically-detected code anomalies relevant to architectural modularity? An exploratory analysis of evolving systems. In Proceedings of the AOSD'12—11th Annual International Conference on Aspect Oriented Software Development, Potsdam, Germany, 25–30 March 2012. [CrossRef]
19. Counsell, S.; Hamza, H.; Hierons, R.M. The 'deception' of code smells: An empirical investigation. In Proceedings of the ITI 2010—32nd International Conference on Information Technology Interfaces, Cavtat/Dubrovnik, Croatia, 21–24 June 2010; pp. 683–688.
20. Sehgal, R.; Nagpal, R.; Mehrotra, D. Measuring Code Smells and Anti-Patterns. In Proceedings of the 2019 4th International Conference on Information Systems and Computer Networks (ISCON), Mathura, India, 21–22 November 2019; pp. 311–314.
21. Sae-Lim, N.; Hayashi, S.; Saeki, M. How Do Developers Select and Prioritize Code Smells? A Preliminary Study. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 484–488.
22. Peters, R.; Zaidman, A. Evaluating the Lifespan of Code Smells using Software Repository Mining. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, Szeged, Hungary, 27–30 March 2012; pp. 411–416.
23. Oliveira, R. When More Heads Are Better than One? Understanding and Improving Collaborative Identification of Code Smells. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), Austin, TX, USA, 14–22 May 2016; pp. 879–882.
24. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Penta, M.D.; De Lucia, A.; Poshyvanyk, D. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Trans. Softw. Eng.* **2017**, *43*, 1063–1088. [CrossRef]
25. Tahir, A.; Dietrich, J.; Counsell, S.; Licorish, S.; Yamashita, A. A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites. *Inf. Softw. Technol.* **2020**, *125*, 106333. [CrossRef]
26. Van Emden, E.; Moonen, L. Java Quality Assurance by Detecting Code Smells. In Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), Richmond, VA, USA, 28 October–1 November 2002; p. 97.
27. Alikacem, E.H.; Sahraoui, H.A. A Metric Extraction Framework Based on a High-Level Description Language. In Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, Edmonton, AB, Canada, 20–21 September 2009; pp. 159–167.
28. Marinescu, R.; Ratiu, D. Quantifying the quality of object-oriented design: The factor-strategy model. In Proceedings of the 11th Working Conference on Reverse Engineering, Delft, The Netherlands, 8–12 November 2004; pp. 192–201.
29. Rao, A.; Reddy, K. Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix. In *Lecture Notes in Engineering and Computer Science*; Newswood Limited: Hong Kong, China, 2008; Volume 2168.
30. Moha, N. Detection and correction of design defects in object-oriented designs. In Proceedings of the OOPSLA '07, Montreal, QC, USA, 21–25 October 2007.
31. Moha, N.; Guéhéneuc, Y.G.; Meur, A.F.; Duchien, L.; Tiberghien, A. From a Domain Analysis to the Specification and Detection of Code and Design Smells. *Form. Asp. Comput.* **2010**, *22*. [CrossRef]

32. Moha, N.; Guéhéneuc, Y.G.; Meur, A.F.L.; Duchien, L. A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering, Budapest, Hungary, 29 March–6 April 2008; Lecture Notes in Computer Science; Springer International Publishing: Berlin/Heidelberg, Germany, 2008; Volume 4961, pp. 276–291. [CrossRef]
33. Moha, N.; Gueheneuc, Y.; Duchien, L.; Le Meur, A. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.* **2010**, *36*, 20–36. [CrossRef]
34. Khomh, F.; Di Penta, M.; Gueheneuc, Y. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In Proceedings of the 2009 16th Working Conference on Reverse Engineering, Lille, France, 13–16 October 2009; pp. 75–84.
35. Moha, N.; Gueheneuc, Y.; Leduc, P. Automatic Generation of Detection Algorithms for Design Defects. In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan, 18–22 September 2006; pp. 297–300.
36. Marinescu, R. Measurement and quality in object-oriented design. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 26–29 September 2005; pp. 701–704.
37. SpotBugs. 2019. Available online: https://spotbugs.github.io (accessed on 27 March 2020).
38. Pugh, B. FindBugs. 2015. Available online: http://findbugs.sourceforge.net (accessed on 27 March 2020).
39. CheckStyle. 2019. Available online: https://checkstyle.sourceforge.io (accessed on 27 March 2020).
40. PMD: An Extensible Cross-Language Static Code Analyzer. 2019. Available online: https://pmd.github.io (accessed on 27 March 2020).
41. Mathew, A.P.; Capela, F.A. An Analysis on Code Smell Detection Tools. In Proceedings of the 17th SC@ RUG 2019–2020, Groningen, The Netherlands, 17 May 2020; p. 57.
42. Azadi, U.; Arcelli Fontana, F.; Taibi, D. Architectural Smells Detected by Tools: A Catalogue Proposal. In Proceedings of the 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), Montreal, QC, Canada, 26–27 May 2019; pp. 88–97.
43. Taibi, D.; Lenarduzzi, V. On the Definition of Microservice Bad Smells. *IEEE Softw.* **2018**, *35*, 56–62. [CrossRef]
44. Pigazzini, I.; Fontana, F.A.; Lenarduzzi, V.; Taibi, D. Towards Microservice Smells Detection. In Proceedings of the 42nd International Conference on Software Engineering, Seoul, Korea, 24 June–16 July 2020.
45. Logarix. AI Reviewer. Available online: http://aireviewer.com (accessed on 21 September 2020).
46. Le, D.M.; Behnamghader, P.; Garcia, J.; Link, D.; Shahbazian, A.; Medvidovic, N. An Empirical Study of Architectural Change in Open-Source Software Systems. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 235–245.
47. Sharma, T. Designite—A Software Design Quality Assessment Tool. In Proceedings of the 2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE), Austin, TX, USA, 17 May 2016. [CrossRef]
48. Mo, R.; Cai, Y.; Kazman, R.; Xiao, L. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In Proceedings of the 2015 12th Working IEEE/IFIP Conference on Software Architecture, Montreal, QC, Canada, 4–8 May 2015; pp. 51–60.
49. Dietrich, J. Upload Your Program, Share Your Model. In Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, Tucson, AZ, USA, 19–26 October 2012; Association for Computing Machinery: New York, NY, USA; pp. 21–22. [CrossRef]
50. von Zitzewitz, A. Mitigating Technical and Architectural Debt with Sonargraph. In Proceedings of the 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), Montreal, QC, Canada, 26–27 May 2019; pp. 66–67.
51. Bugan IT Consulting UG. STAN. Available online: http://stan4j.com (accessed on 21 September 2020).
52. Headway Software Technologies Ltd. Structure101. Available online: https://structure101.com (accessed on 21 September 2020).
53. Cerny, T.; Svacina, J.; Das, D.; Bushong, V.; Bures, M.; Tisnovsky, P.; Frajtak, K.; Shin, D.; Huang, J. On Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices. *IEEE Access* **2020**. [CrossRef]

54. Kumar, K.S.; Malathi, D. A Novel Method to Find Time Complexity of an Algorithm by Using Control Flow Graph. In Proceedings of the 2017 International Conference on Technical Advancements in Computers and Communications (ICTACC), Kochi, India, 22–24 August 2017; pp. 66–68. [CrossRef]
55. Ribeiro, J.C.B.; de Vega, F.F.; Zenha-Rela, M. Using Dynamic Analysis Of Java Bytecode For Evolutionary Object-Oriented Unit Testing. In Proceedings of the 25th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC), Belem, Brazil, June 2007.
56. Syaikhuddin, M.M.; Anam, C.; Rinaldi, A.R.; Conoras, M.E.B. Conventional Software Testing Using White Box Method. *Kinet. Game Technol. Inf. Syst. Comput. Netw. Comput. Electron. Control* **2018**, *3*, 65–72. [CrossRef]
57. Roy, C.K.; Cordy, J.R.; Koschke, R. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* **2009**, *74*, 470–495. [CrossRef]
58. Selim, G.M.K.; Foo, K.C.; Zou, Y. Enhancing Source-Based Clone Detection Using Intermediate Representation. In Proceedings of the 2010 17th Working Conference on Reverse Engineering, Beverly, MA, USA, 13–16 October 2010; pp. 227–236. [CrossRef]
59. Albert, E.; Gómez-Zamalloa, M.; Hubert, L.; Puebla, G. Verification of Java Bytecode Using Analysis and Transformation of Logic Programs. In *Practical Aspects of Declarative Languages*; Hanus, M., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 124–139.
60. Keivanloo, I.; Roy, C.K.; Rilling, J. SeByte: Scalable clone and similarity search for bytecode. *Sci. Comput. Program.* **2014**, *95*, 426– 444. [CrossRef]
61. Keivanloo, I.; Roy, C.K.; Rilling, J. Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web Reasoning. In Proceedings of the 6th International Workshop on Software Clones, Zurich, Switzerland, 4 June 2012; IEEE Press: Piscataway, NJ, USA, 2012; pp. 36–42.
62. Lau, D. An Abstract Syntax Tree Generator from Java Bytecode. 2018. Available online: https://github.com/davidlau325/BytecodeASTGenerator (accessed on 27 March 2020).
63. Chatley, G.; Kaur, S.; Sohal, B. Software Clone Detection: A review. *Int. J. Control Theory Appl.* **2016**, *9*, 555–563.
64. Gabel, M.; Jiang, L.; Su, Z. Scalable Detection of Semantic Clones. In Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; ACM: New York, NY, USA, 2008; pp. 321–330. [CrossRef]
65. Su, F.H.; Bell, J.; Harvey, K.; Sethumadhavan, S.; Kaiser, G.; Jebara, T. Code Relatives: Detecting Similarly Behaving Software. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; ACM: New York, NY, USA, 2016; pp. 702–714. [CrossRef]
66. Makai, M. Object-Relational Mappers (ORMs). 2019. Available online: https://www.fullstackpython.com/object-relational-mappers-orms.html (accessed on 27 March 2020).
67. Walker, A.; Svacina, J.; Simmons, J.; Cerny, T. On Automated Role-Based Access Control Assessment in Enterprise Systems. In *Information Science and Applications*; Kim, K.J.; Kim, H.Y., Eds.; Springer: Singapore, 2020; pp. 375–385.
68. Trnka, M.; Svacina, J.; Cerny, T.; Song, E.; Hong, J.; Bures, M. Securing Internet of Things Devices Using the Network Context. *IEEE Trans. Ind. Inform.* **2020**, *16*, 4017–4027. [CrossRef]
69. Svacina, J.; Simmons, J.; Cerny, T. Semantic Code Clone Detection for Enterprise Applications. In Proceedings of the 35th ACM/SIGAPP Symposium On Applied Computing, Brno, Czech Republic, 30 March–3 April 2020.
70. Torres, A.; Galante, R.; Pimenta, M.S. Towards a uml profile for model-driven object-relational mapping. In Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering, Fortaleza, Brazil, 5–9 October 2009; pp. 94–103.
71. Cerny, T.; Cemus, K.; Donahoo, M.J.; Song, E. Aspect-driven, data-reflective and context-aware user interfaces design. *ACM SIGAPP Appl. Comput. Rev.* **2013**, *13*, 53–66. [CrossRef]
72. Rademacher, F.; Sachweh, S.; Zündorf, A. A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems. In *Enterprise, Business-Process and Information Systems Modeling*; Nurcan, S., Reinhartz-Berger, I., Soffer, P., Zdravkovic, J., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 311–326.

73. Alshuqayran, N.; Ali, N.; Evans, R. Towards Micro Service Architecture Recovery: An Empirical Study. In Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 30 April–4 May 2018, pp. 47–4709.
74. Granchelli, G.; Cardarelli, M.; Di Francesco, P.; Malavolta, I.; Iovino, L.; Di Salle, A. Towards Recovering the Software Architecture of Microservice-Based Systems. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 46–53.
75. Tarjan, R. Depth-first search and linear graph algorithms. In Proceedings of the 12th Annual Symposium on Switching and Automata Theory (SWAT 1971), East Lansing, MI, USA, 13–15 October 1971; pp. 114–121.
76. Márquez, G.; Astudillo, H. Identifying Availability Tactics to Support Security Architectural Design of Microservice-Based Systems. In Proceedings of the 13th European Conference on Software Architecture, Paris, France, 9–13 September 2019; pp. 123–129. [CrossRef]
77. Aderaldo, C.M.; Mendonça, N.C.; Pahl, C.; Jamshidi, P. Benchmark Requirements for Microservices Architecture Research. In Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, Buenos Aires, Argentina, 22 May 2017; pp. 8–13.
78. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Xu, C.; Ji, C.; Zhao, W. Benchmarking microservice systems for software engineering research. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May–3 June 2018; Chaudron, M., Crnkovic, I., Chechik, M., Harman, M., Eds.; ACM: New York, NY, USA, 2018; pp. 323–324. [CrossRef]
79. Smid, A.; Wang, R.; Cerny, T. Case Study on Data Communication in Microservice Architecture. In Proceedings of the Conference on Research in Adaptive and Convergent Systems, Chongqing, China, 24–27 September 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 261–267. [CrossRef]
80. Mantyla, M.; Vanhanen, J.; Lassenius, C. A taxonomy and an initial empirical study of bad smells in code. In Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 22–26 September 2003; pp. 381–384. [CrossRef]