

Classifying the Reliability of the Microservices Architecture

Adrian Ramsingh^a, Jeremy Singer^b and Phil Trinder^c

School of Computing Science, University of Glasgow, Glasgow, U.K.

Keywords: Microservices, Reliability, Web Architectures, Patterns, Bad Smells, Web Applications.

Abstract: Microservices are popular for web applications as they offer better scalability and reliability than monolithic architectures. Reliability is improved by loose coupling between *individual* microservices. However in production systems some microservices are tightly coupled, or *chained* together. We classify the reliability of microservices: if a minor microservice fails then the application continues to operate; if a critical microservice fails, the entire application fails. Combining reliability (minor/critical) with the established classifications of dependence (individual/chained) and state (stateful/stateless) defines a new three dimensional space: the Microservices Dependency State Reliability (MDSR) classification. Using three web application case studies (Hipster-Shop, Jupyter and WordPress) we identify microservice instances that exemplify the six points in MDSR. We present a prototype static analyser that can identify all six classes in Flask web applications, and apply it to seven applications. We explore case study examples that exhibit either a known reliability pattern or a bad smell. We show that our prototype static analyser can identify three of six patterns/bad smells in Flask web applications. Hence MDSR provides a structured classification of microservice software with the potential to improve reliability. Finally, we evaluate the reliability implications of the different MDSR classes by running the case study applications against a fault injector.

1 INTRODUCTION

Microservices are popular for web applications, since they may offer better scalability and reliability than monolithic components. Some microservices are *stateful*, recording data, e.g. participants in a web chat. Others are *stateless*, i.e. they simply accept requests and purely process them.

Architectures with monolithic components are prone to *catastrophic failure*, where user-visible functionality is suddenly and permanently unavailable (Nikolaidis et al., 2004). It is common for the failure of a single monolithic component to cause the entire system to fail (a cascade failure).

In contrast microservice architectures potentially provide *improved reliability* due to loose coupling of services. If one microservice fails, others will remain available. This may cause a reduction in throughput but will most likely avoid catastrophic failure. In the worst case scenario, the loose coupling of services enables graceful failure (Soldani et al., 2018).

This is achieved based on the design principle that


microservices are implemented as standalone, independent services (Jamshidi et al., 2018). However, many large scale web applications include *chains* of microservices where a set of services are closely dependent, e.g. Netflix Titus (Ma et al., 2018).


Chained microservices are tightly coupled, e.g. by high-frequency API-based interaction sequences. Chained microservices make an application less reliable because if any of the services fail the entire chain fails, and may induce catastrophic failure (Heorhiadi et al., 2016). For example, in 2014 BBC experienced a critical database overload that caused many critical microservices to fail one after another (Cooper, 2014). In 2015, Parse.ly experienced several cascading outages in its analytics data processing due to a microservices message bus overload (Montalenti, 2015).

This paper makes the following research contributions.

(1) We combine a *reliability* (minor/critical) classification with the established classifications of dependence (individual/chained) and state (stateful/stateless). If a minor microservice fails the application continues to function, although performance or functionality may be reduced. If a critical microser-

^a  <https://orcid.org/0000-0003-3501-902X>

^b  <https://orcid.org/0000-0001-9462-6802>

^c  <https://orcid.org/0000-0003-0190-7010>

vice fails, the application fails catastrophically. Combining reliability with state and dependence defines a new three dimensional space: the Microservices Dependency State Reliability (MDSR) classification. As microservice chains are necessarily critical (Heorhiadi et al., 2016), only six of the possible eight points in the space are valid. We outline a prototype static analyser that can identify all six MDSR classes. Applying the tool to 30 microservices from seven small Flask web applications reveals interesting statistics, e.g. the majority of services are chained (70%), and critical (77%) (Section 4).

(2) Using three web applications we highlight microservices that exemplify each point in MDSR. The web applications are: (1) *Hipster-Shop*, a Google demo application; (2) *JPyL*, a Jupyter Notebook/Flask web stack; and (3) *WordPress*, a content management system (Section 2).

(3) We show that each of the MDSR critical case study microservices exhibits a known bad smell (Taibi and Lenarduzzi, 2018). Likewise in each minor MDSR class the case study microservices follow a design pattern (Taibi and Lenarduzzi, 2018). We show that the prototype static analyser can identify three of six patterns/bad smells in Flask web applications. The analysis offers the opportunity to focus reliability engineering efforts early in the development cycle. That is, we propose *static* MDSR analysis as a complement to *dynamic* Service Dependency Graph (SDG) analysis (Ma et al., 2018) (Section 5).

(4) We explore reliability implications of different MDSR microservice classes by running the three web applications against a simple process level fault injector. Specifically we show: (1) *All* applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of a minor microservice, and successive failures of minor microservices. (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications (Section 6).

2 CASE STUDIES

We illustrate our new classification and analysis using three realistic microservice web applications. *Hipster-Shop* is a popular Google microservices demo web application; *JPyL* is a Jupyter/Flask microservices web application; *WordPress* is a widely-used Content Management System. The applications illustrate different aspects of real world microservice web applications, e.g. Hipster-Shop implements microservices in different languages, and both JPyL and Word-

Press combine monolithic and microservice components.

2.1 Hipster-Shop

Key attractions of microservices are decentralization and polyglotism. Here each service can be separately developed with appropriate programming languages and tools, promoting agile development (Zimmermann, 2017). Many developers claim this is why they prefer microservices (IBM, 2021).

The Hipster-Shop case study illustrates polyglot development with services developed in Python, Go and Java and communication via gRPC remote procedure calls. Hipster-Shop is an e-commerce application with 10 microservices (Figure 1) used by Google to demonstrate tools like Kubernetes Engine (Google, 2021). Users can perform activities like viewing products, adding items to cart and making purchases¹.

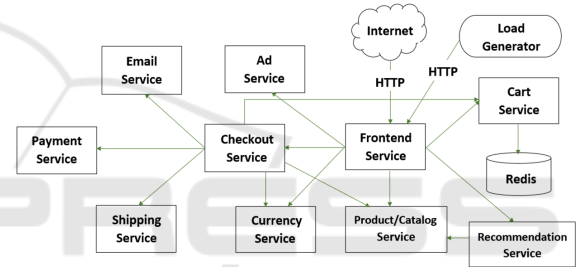


Figure 1: Hipster-Shop Architecture.

2.2 JPyL: Jupyter/Python/Linux

Migrating from a monolithic architecture towards a full microservices architecture is a gradual process. It is common for developers to initially integrate one or more microservice tiers that function alongside monolithic components, in a hybrid approach known as *microlith* (Soldani et al., 2018).

JPyL is a microlith web stack that combines the popular Flask microservices web tier with monolithic Jupyter components (Figure 2). The Flask tier has seven microservices and is fairly conventional as outlined in Figure 2.

Some are supported by a data store, e.g. current geolocation and IP address rely on the userdata microservice that interfaces with a MySQL database. Data is displayed on the webpage via reverse proxy and port configuration microservices on port 10125. Crucially for reliability a backup URL port can be initiated via redirect if the original port service is interrupted.

¹<https://github.com/GoogleCloudPlatform/microservices-demo>

Each service is handled by a specific set of Flask microservices². For example, security headers are processed by a Python Talisman microservice.

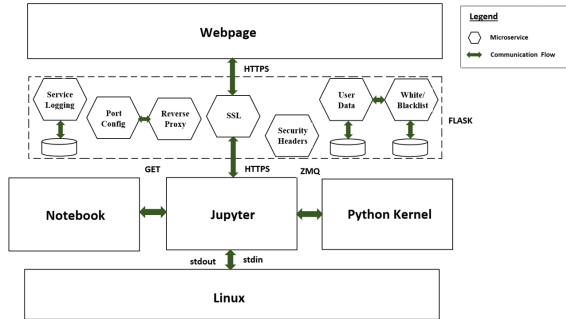


Figure 2: JPyL Application Architecture.

2.3 WordPress

WordPress is an open source Content Management System (CMS) used for many websites (Patel et al., 2011). As a standalone application WordPress has a monolithic architecture with core CMS components that communicate with a MySQL database. Microservices can be integrated with WordPress to provide plugins for additional features, e.g. to post comments, allow subscription memberships or search indexes (Cabot, 2018).

The application we study integrates microservice endpoints that allow users to post comments³. The service uses WordPress HTTP REST. It facilitates communication between the microservice and the monolithic components as shown in Figure 3.

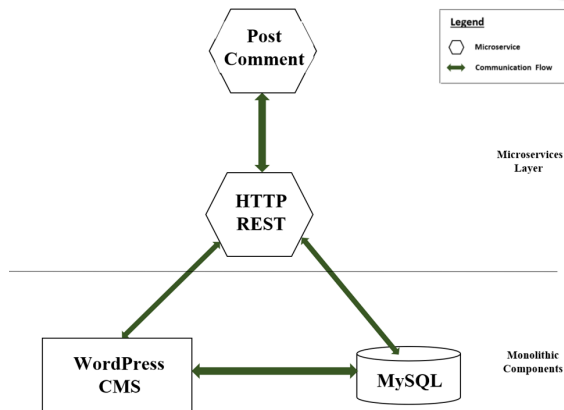


Figure 3: WordPress Application Architecture.

²<https://bitbucket.org/latent12/microproject/src/master/jpyl/>

³<https://bitbucket.org/latent12/microproject/src/master/wordpress/>

3 RELATED WORK

3.1 Existing Classifications

Lewis and Fowler identify three microservices design principles (Lewis and Fowler, 2015). (1) Independent services — each service should run in its own process and be deployed in its own container like one service per Docker container. (2) Single functionality — one business function per service. This is referred to as the Single Responsibility Principle (SRP). (3) Communication — often using a REST API or message brokers.

Others have added other principles like reliability (Heorhiadi et al., 2016), and advocate using design patterns like timeouts, bounded retries, circuit breakers and bulkheads to mitigate failures (Heorhiadi et al., 2016). However most assume that all microservices are individual, but in reality there are many types of microservices.

Microservices are commonly classified by their properties and we outline some key properties below, and summarise in Table 1.

3.1.1 Dependence

This property classifies a microservice by how tightly coupled it is with other microservices (Heorhiadi et al., 2016; Ma et al., 2018; Ghirotti et al., 2018). Coupling is the degree of dependence between software components like microservices, and there are different types like content, data and control coupling (Offutt et al., 1993). Software architects seek loose coupling, and this is often achieved for microservices through data coupling.

Individual microservices are loosely coupled to other microservices and communication with other microservices is typically via infrequent remote API calls. Many individual microservices express computations at a high level of abstraction, and provide their own built-in runtimes, functionalities and data stores (Salah et al., 2016). Examples for JPyL include the SSL and Service Logging services (Figure 2) while Hipster-Shop includes Frontend and Ad-service microservices (Figure 1).

In contrast *chained microservices* are tightly coupled with one or more other microservices. A chained microservice is reliant on some form of constant communication or a *chain of calls* with another service to function (Heorhiadi et al., 2016; Rossi, 2018). This is often due to control coupling where the chained services must request and marshal data between themselves (Offutt et al., 1993).

In JPyL, the Reverse Proxy service is dependent on the Port Configuration service to display data on

the webpage through port 10125. Specifically the Reverse Proxy must access the Port Configuration to determine which backup URL port to use.

3.1.2 State

This property classifies a microservice by whether it preserves state between service requests. *Stateful* microservices require data storage, for example to record transactions or current actors (Wu, 2017). In JPyL the Service Logging microservice is stateful: it logs the status of all microservices in the application in a MySQL database (Figure 2). Other microservices are *Stateless*, i.e. they maintain no session state. Such services typically accept requests, process them in a pure fashion, and respond accordingly. In JPyL the SSL microservice is stateless: it processes https requests but maintains no session data.

3.1.3 Combining & Inheriting Properties

Microservices may have any combination of properties, e.g. individual/stateful or chained/stateless. Properties may be inherited from other chained microservices, e.g. if any microservice is stateful then the entire chain is stateful. In Hipster-Shop although both Checkout and Payment microservices are stateless, their chain with Cart Services is stateful as Cart Services is stateful (Table 6).

3.2 Microservices Reliability

There are substantial studies of the reliability of microservice software in both the academic (Heorhiadi et al., 2016; Zhou et al., 2018; Toffetti et al., 2015) and grey literature (Wolff, 2018; Gupta and Palvankar, 2020). These reveal that reliability in the microservices architecture is not always attainable because the reliability design principle is not always followed. This principle states that a microservice should be fault tolerant so that in the case of failure, its impact on other services will be negligible (Power and Kotonya, 2018).

However, developers do not always implement the necessary design patterns or follow the Fowler and Lewis design principles (Section 3.1) to prevent microservices failure. Even if they do, they remain unaware whether their microservice can actually tolerate failures until it actually occurs (Heorhiadi et al., 2016). Thus, the impact of a microservices failure on an application is not always readily known beforehand.

Table 1: Microservices Classification Criteria.

Classification	Properties	Description
Dependence	Individual	Loosely Coupled. Limited communication with other microservices.
	Chained	Tightly Coupled. Constant communication with other microservices required.
State	Stateless	No data store. Does not maintain state.
	Stateful	Utilises data store. Maintains state.
Reliability	Critical	Supports core functionality. Service failure means that the application becomes suddenly and permanently unavailable.
	Minor	Supports non-essential functionality. Application continues to function despite service failure. Degradation in performance or graceful failure over a period of time.

3.3 Patterns and Bad Smells

Some design patterns capture reusable solutions to common microservice design challenges (Taibi and Lenarduzzi, 2018). For example the *Database-Per-Service* pattern prevents tight coupling by ensuring that multiple microservices are not dependent on a single data store. Instead, each service accesses its own private store (Taibi et al., 2018), eliminating the single data store as a single point of failure (SPOF). Some of the microservices patterns utilised in our evaluation are summarised in Table 2

Table 2: Microservices Patterns Description.

Pattern	Description
Database-Per-Design	Microservice accesses its own private data store.
API-Gateway	Microservice communication occurs through an API.
Single Responsibility Principle	Microservice performs a single functionality.

While design patterns like Database-Per-Service help, they are not universal solutions. For example a single atomic operation often spans mul-

multiple microservices, and here additional techniques are required to ensure consistency across the data stores (Rudrabhatla, 2018).

Likewise microservice bad smells identify common designs that may cause issues (Taibi and Lenarduzzi, 2018). Indeed (Heorhiadi et al., 2016) and the Fowler and Lewis design principles consider *all chained microservices* as bad smells and prone to reliability issues. Some of the microservices bad smells found in our evaluation are summarised in Table 3. Microservices Greedy, Shared Persistency and Cyclic Dependency are listed in (Taibi and Lenarduzzi, 2018), Chained Services is mentioned by (Heorhiadi et al., 2016) and SRP Violation is a well-known microservice bad smell.

Table 3: Microservices Bad Smells.

Bad Smell	Description
SRP Violation	Microservice performs more than one functionality. Reason: Microservice becomes more critical. Increases the probability of catastrophic failure in the application.
Microservices Greedy	Microservices created for every feature in an application. Reason: More microservices could lead to more points of failure.
Shared Persistency	Different microservices access the same data storage. Reason: Single Point of Failure (SPOF).
Chained Services	Microservices that depend on communication or data marshalling from other microservices. Reason: Tight Coupling.
Cyclic Dependency	Where there are cycles in the call graph, e.g. A to B, B to C and C to A. A subset of Chained Services. Reason: Too much dependency.

3.4 Failures in Microservices

Even if the microservices design principles are followed, failure is to be expected. There are two major reasons for these failures (Zhou et al., 2018). Functional Failures due to poor implementation e.g. a SQL column missing error is returned upon some data request. Environmental Failures due to misconfiguration of the infrastructure necessary to run the microservices efficiently e.g. microservices processing of requests is slow due to insufficient memory being made available in the Docker environment.

Partial failures are typically temporary and recovery is automatic, e.g. a microservice without a load balancer may be briefly overloaded (Zhou et al., 2018). Downtime can often be minimised if replace-

ment microservice instance(s) are activated automatically (Toffetti et al., 2015). Partial failure is considered acceptable in the design of microservices applications.

In contrast, catastrophic failure is considered unacceptable as it can lead to long downtimes without manual intervention (Nikolaidis et al., 2004). In microservices, catastrophic failures are often termed Interaction Failures (Zhou et al., 2018). Common causes are incorrect coordination or communication failure between microservices, e.g. asynchronous message delivery lacking sequence control or a microservice receiving an unexpected output in its call chain. The errors may be replicated in several microservice instances (Zhou et al., 2018), so even switching workload from a failed instance doesn't help as the new instance fails in the same way.

Chained microservices are especially prone to interaction faults because they violate the *Single Responsibility Principle (SRP)* and lead to brittle architectures (Heorhiadi et al., 2016). Moreover adding more microservices to the chain increases coupling and the likelihood of catastrophic failure (Rossi, 2018). If one service in the chain fails, there will be a cascade of failures of all services in the chain (Heorhiadi et al., 2016).

A limitation of (Heorhiadi et al., 2016) and the Lewis and Fowler design principles is that they consider only dependence. *We extend their work by simultaneously classifying dependence, state and reliability.*

3.5 Detecting Failures

Failures in microservice-based applications may arise from the microservices, or from the infrastructure services like libraries, containers like Docker, development frameworks like Flask, etc. Here we focus on failures in the microservices, and detecting these failures is often challenging.

A common approach is to configure a collection of microservices indicators (KPIs) to continuously monitor for the causes of failure. The KPIs are typically time series, e.g. the response time of a microservice to requests from other services. The microservice is identified as failing if it fails to meet the expected KPI (Meng et al., 2020).

Service Dependency Graphs (SDGs) can be used to dynamically detect microservices bad smells by mapping their node relationships (Ma et al., 2018). However, diagnosing the severity and reason for a failure in a large system is challenging. The diagnosis usually requires domain and site-reliability knowledge as well as automated observability support (Has-

selbring and Steinacker, 2017). Not all companies have such resources.

4 CLASSIFYING RELIABILITY

4.1 Critical vs Minor Reliability

To analyse the reliability of a microservice architecture we consider a microservice reliability property alongside the established properties of state and dependency.

Critical microservices provide core functionality to the application, and if such a service fails, the entire application fails catastrophically even if there are several instances of the microservice. In JPyL, the chained PortConfig to ReverseProxy services are critical because if the PortConfig service fails, the ReverseProxy service will not be able to determine the port to display data or access the URL backup port. As with other properties criticality is inherited within chains, so if any microservice is critical then the entire chain is critical.

Minor microservices provide non-essential functionality. The application continues to operate if they fail, although performance and/or functionality may be reduced. In Hipster-Shop, the Adservice microservice is minor because if it fails, the server returns a 404 status code indicating that the service is temporarily unavailable. The rest of the application continues to function normally.

It would also be possible to consider partial failures that eventually affect the operation of the system (Cristian, 1991). However, given the challenges of distinguishing between partial failures with different severities we adopt a binary minor/critical classification. As partial failures are considered as gray failures, severe partial failures are classified as critical, and low severity failures are classified as minor.

4.2 MDSR Classification

Combining reliability with the state and dependency classifications defines a three-dimensional space: our new Microservices Dependency State Reliability (MDSR) Classification Tree as shown in Figure 4. It can also be represented in tabular form as in Tables 5 and 6.

In MDSR chained microservices are necessarily critical as argued in (Heorhiadi et al., 2016), and confirmed in our evaluation (Section 6) even for chains that attempt to recover reliability using microservice patterns. As examples we implement a Database-Per-Service pattern for the chained/stateful UserData &

White/Black Listing service and an API Gateway pattern for the chained/stateless/ Product Catalog & Recommended service. In both cases the application fails catastrophically despite reporting only a "404 Service Not Found" error.

The fourth rows of Tables 5 and 6 show example microservices from the case study applications for each of the six MDSR classes. For example the individual/stateful/minor exemplar is JPyL's Service Logging microservice. The fifth rows of the tables show the error reported if the service fails.

4.3 Semi-automatic Classification

Static analysis of a set of microservices can automatically propose MDSR classifications for many microservices in an application. We demonstrate the principle with a prototype analyser that classifies all Python/Flask microservices in a source project⁴. The analyser tokenises the Python/Flask code, and identifies properties using keyword matches. As examples, the presence of keywords like "SQL" or "JSON" classifies a service as stateful; the presence of "request", "requests", "requests.get", "get" Flask keywords, or use of the "POST" or "GET" methods classifies a service as chained as they indicate a service is pushing data or requesting information from other microservices. Reliability is determined by the type of pattern or bad smell detected as discussed in Sections 5.2 & 5.3. Figure 5 shows a screenshot of the tool's output for JPyL.

The prototype analyser identifies all six MDSR classes. The analyser may, however, propose an incorrect classification, for example a stateless service may be incorrectly classified as stateful if a keyword like "SQL" appears in a comment. Similarly, a stateful microservice could be classified as stateless if it uses a persistent store that is not included in the current set of keywords.

Despite these limitations the analyser is effective in classifying microservices. For example Tables 5 and 6 show how it correctly classifies all of the JPyL microservices. We have also applied the analyser to a total of 30 microservices in a further six small Flask web application projects⁵. Manual inspection of 3 of the projects validates the properties identified by the

⁴<https://bitbucket.org/latent12/microproject/src/master/analyser/>

⁵GitHub Links: <https://github.com/IBM/worklog/tree/master/app>, <https://github.com/IBM/Flask-microservice>, <https://github.com/bakrianoo/Flask-elastic-microservice>, <https://github.com/airavata/Blitzkrieg>, <https://github.com/michaellitherland/Flask-microservice-demo>, <https://github.com/umermansoor/microservices>

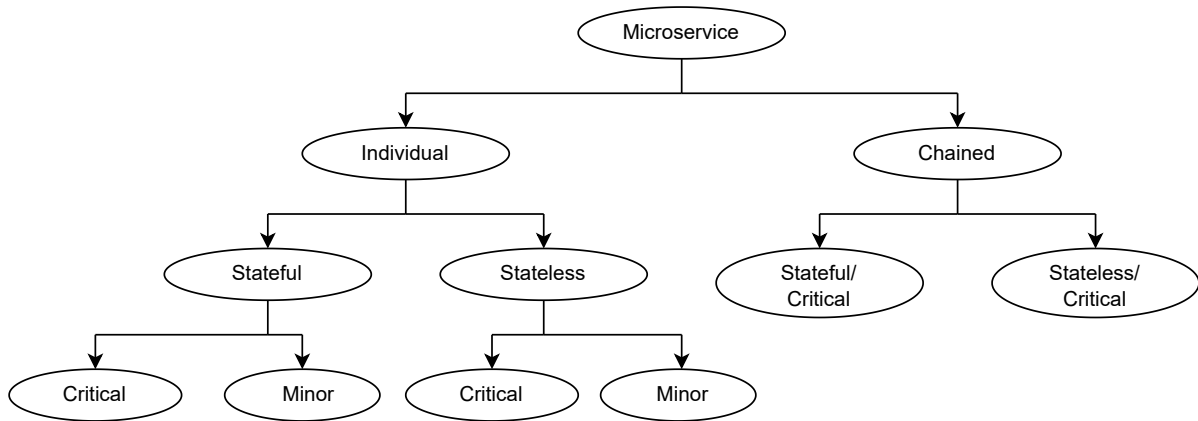


Figure 4: MDSR Classification Tree.

```

MDSR Classification Properties
('ssl.py', 'Individual', 'Stateless', 'N/A', 'Reason: No Assignment')
('listing.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('userinfo.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('proxy.py', 'Chained', 'Stateless', 'Critical', 'Bad Smell: Chained')
('logs.py', 'Individual', 'Stateful', 'N/A', 'Reason: No Assignment')
('portconfig.py', 'Chained', 'Stateless', 'Critical', 'Bad Smell: Chained')
('sechad.py', 'Individual', 'Stateless', 'N/A', 'Reason: No Assignment')

Proceeding To Analyse Microservices For Patterns/Bad Smells

MDSR Patterns/Bad Smells
('ssl.py', 'Individual', 'Stateless', 'Critical', 'Failure: ERR_SSL')
('listing.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('userinfo.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('proxy.py', 'Chained', 'Stateless', 'Critical', 'Bad Smell: Chained')
('logs.py', 'Individual', 'Stateful', 'Minor', 'Pattern: Database-Per-Service')
('portconfig.py', 'Chained', 'Stateless', 'Critical', 'Bad Smell: Chained')
('sechad.py', 'Individual', 'Stateless', 'Minor', 'Failure: 404')
  
```

Figure 5: MDSR Analyser JPyL Output.

analysers. As a further example the analyser output for the IBM worklog application is shown in Figure 6, and corresponds to the use case diagram provided by IBM⁶.

```

MDSR Classification Properties
('create_user_service.py', 'Individual', 'Stateful', 'N/A', 'Reason: No Assignment')
('user.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('login.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('get_worklog_data_service.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('reset_password_service.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('login_service.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')

Proceeding To Analyse Microservices For Patterns/Bad Smells

MDSR Patterns/Bad Smells
('create_user_service.py', 'Individual', 'Stateful', 'Minor', 'Pattern: Database-Per-Service')
('user.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('login.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('get_worklog_data_service.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('reset_password_service.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
('login_service.py', 'Chained', 'Stateful', 'Critical', 'Bad Smell: Chained')
  
```

Figure 6: MDSR Analyser IBM Worklogs Output.

Table 4 shows the number and percentage of each MDSR class of microservice that the analyser detects in the seven web applications, and key observations are as follows. The majority of services are chained (70%), and most are stateful (73%). 50% of services are from a single MDSR classification, i.e. chained/stateful/critical. Perhaps most startling is that 77% of

services are critical. We speculate that this reflects that the designers of these small web applications have not designed them to be reliable.

Table 4: MDSR Classifications of 7 small Flask Applications containing 30 Microservices.

Classification	No. Services	%
Individual	9	30
Chained	21	70
Stateful	22	73
Stateless	8	27
Critical	23	77
Minor	7	23
Individual/Stateful	7	23
Individual/Stateless	2	7
Chained/Stateful	15	50
Chained/Stateless	6	20
Individual/Stateful/Critical	2	7
Individual/Stateless/Critical	1	3
Chained/Stateful/Critical	15	50
Chained/Stateless/Critical	6	20
Individual/Stateful/Minor	5	17
Individual/Stateless/Minor	1	3

5 IDENTIFYING RELIABILITY

MDSR analysis provides information about the expected reliability of microservices and chains of microservices in an architecture. In general reliability engineering should focus on the 77% of critical microservices identified by the analysis in Table 4. More specifically the analysis can help identify design patterns and bad smells in the architecture. To illustrate, the sixth row of the Patterns and Bad Smell tables (Tables 5 & 6) identify the microservice pattern or bad smell associated with each point in the classifica-

⁶<https://github.com/IBM/worklog/blob/master/designs/>

tion space. Of the patterns and bad smells enumerated in (Taibi and Lenarduzzi, 2018) (and summarised in Tables 2 & 3) the case studies exhibit four out of eight patterns and three out of eleven bad smells.

Static analysis enables the early identification of patterns and bad smells, allowing developers to anticipate the types of failures, and their likely impact. Potentially this information allows developers to troubleshoot problems faster and prevent long application downtimes.

5.1 MDSR Patterns & Implications

The sixth row of the MDSR Patterns table (Table 5) identifies the microservice pattern exhibited by the case study example microservice, or microservice chain. In our case study applications, individual/stateful microservices and individual/stateless microservices have only minor reliability implications if they implement a pattern as shown in Table 5. For example JPyL Service Logging is individual/stateful/minor and implements the Database-Per-Service pattern.

5.2 MDSR Bad Smells & Implications

The sixth row of the MDSR Bad Smells table (Table 6) identifies the microservice bad smell exhibited by the case study example microservice, or microservice chain. Considering the Patterns and Bad Smells tables together (Tables 5 and 6) we see that the example case study microservices at each point in the MDSR classification exhibit either a design pattern or a bad smell. This is expected as microservice best practice applies patterns, while bad smells indicates places where design principles have not, or cannot be applied (Heorhiadi et al., 2016).

Bad smells identified by MDSR can be considered for refactoring to improve reliability. That is, most critical microservices are associated with known bad smells as shown in Table 6. For example the individual/stateless/critical SSL microservice in JPyL is an instance of Microservices Greedy, where there is a proliferation of microservices. Of course SSL need not be implemented as a microservice.

A key element of MDSR is that chained microservices remain critical, even if they implement good design patterns. For example, the chained/stateful/critical UserData & White/Black Listing microservices implement the Database-Per-Service pattern but still fail catastrophically as we show in Section 6.

5.3 Semi-automatic Pattern/Bad Smell Detection

The prototype MDSR analyser can detect the Database-Per-Service pattern, and both chained microservices and Shared Persistency bad smells. Shared Persistency is detected by determining whether any microservices share a data store. Currently the user must provide the analyser with the names of the data stores used in the application, e.g. `jpyl.micro` in JPyL. An enhanced analyser could parse the Flask code and extract the data store names from connection statements. The analyser counts the number of times each data store name appears in the microservices in the given directory. If the count is greater than 1, the microservices have a shared persistency bad smell. Microservices with a unique persistent store name implement a Database-Per-Service pattern.

There are some bad smells that the analyser isn't able to detect. Some of these, like Cyclic Dependency could be detected dynamically, perhaps using SDGs to examine the connection between services and the rate of communication (Ma et al., 2018; Omer and Schill, 2011). Other bad smells likely require human analysis, like Microservices Greedy and SRP violation.

The analyser also inspects Flask error handling codes to classify the reliability of a microservice. For example if a service returns a 404 error indicating that the service is not found or temporarily unavailable the failure is considered minor. In contrast a code like 415 indicates that there is a SSL Protocol Violation, and the service is critical because even if other services are available, the application cannot accept https requests, and has failed catastrophically.

Table 7 shows the number and percentage of bad smells and patterns detected by the analyser in the seven web applications. Key observations are as follows. (1) 17% of services implement Database-Per-Service. (2) As 70% of the services are chained (Table 4), they are the most common bad smells. This accords with, and provides evidence for, the claim in (Heorhiadi et al., 2016) that developers do not always implement reliability patterns.

6 EVALUATING RELIABILITY

We execute the Hipster, JPyL & WordPress web applications against a simple fault injector to investigate the reliability implications of different MDSR classes. All applications are executed on a typical server, i.e. a 16 core Intel server with 2TB of RAM

Table 5: MDSR Pattern Classification.

	Individual				Chained	
	Stateful		Stateless		Stateful	Stateless
	Critical	Minor	Critical	Minor	Critical	Critical
Microservices		JPyL Service Logging		Hipster Adservice JPyL Security Headers	JPyL UserData, White/Black Listing	Hipster Recommend, ProductCatalog
Failure Impact		404 Service Not Found		404 Service Not Found	404 Service Not Found	404 Service Not Found
Pattern		Database-Per Service		Single Responsibility Principle (SRP)	Database-Per Service	API Gateway

Table 6: MDSR Bad Smell Classification.

	Individual				Chained	
	Stateful		Stateless		Stateful	Stateless
	Critical	Minor	Critical	Minor	Critical	Critical
Microservices	Hipster Frontend		JPyL SSL		Hipster Payment, Checkout, Cart WordPress Comments HTTP REST	Hipster Shipping, Checkout JPyL PortConfig, ReverseProxy
Failure Impact	500 Internal Server Error		ERR_SSL Protocol Failure Error		Database Connection Error	500 Internal Server Error
Bad Smells	SRP Violation		Microservices Greedy		Chained, Shared Persistency	Chained, Cyclic Dependency

Table 7: MDSR Patterns/Bad Smells found in 7 small Flask Applications containing 30 Microservices.

Patterns/Bad Smells	No. Services	%
Database-Per-Service	5	17
Shared Persistency	2	7
Chained Services	21	70
Other	2	6

running Ubuntu 18.04. Hipster uses Minikube 1.19.0 and multiple languages including Python 3.6, Go 1.10 and C# 8.0. JPyL uses Jupyter Server 6.1, Python 3.6, MySQL 5.7 and Flask 1.1.2. WordPress v5.7.2 uses PHP 7.2 and MySQL 5.7. The code for all applications, tools and experiments are available⁷.

6.1 Critical Microservice Failures

Catastrophic failure is a major challenge for web applications and our case study applications are no

exception. To investigate the failure of critical microservices we target the chained/stateful/critical HTTP REST microservice in WordPress, the individual/stateless/critical SSL microservice in JPyL & the chained/stateful/critical Cart Service in Hipster.

Figures 7, 8 & 9 plot throughput (Request KB/s) against time. The red line in each box plot is the median throughput from three executions. Once established, all services have throughputs of approximately 600KB/s. When the fault injector kills the critical microservice at 43s the applications fail almost instantaneously: by 50s throughput is 0KB/s.

6.2 Minor/Individual Failures

Our first investigation of the failure of minor microservices uses an *individual* microservice. Specifically we target the individual/stateful/minor Service Logging microservice in JPyL. Recall that, although stateful, this microservice has a private store follow-

⁷<https://bitbucket.org/latent12/microproject/src/master/>

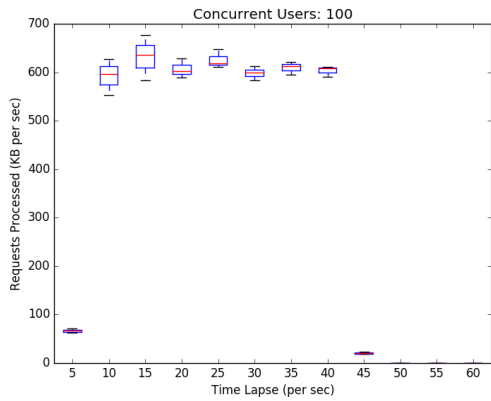


Figure 7: A JPyL Critical Failure (SSL) at 43s.

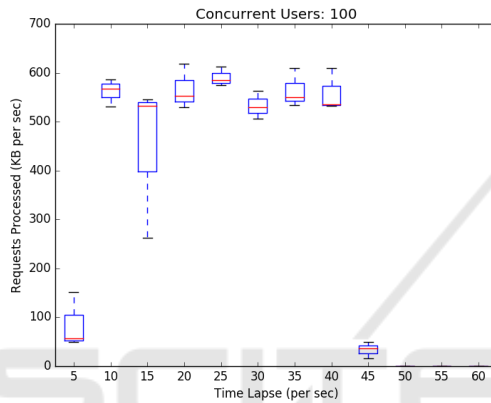


Figure 8: A WordPress Critical Failure (HTTP REST & Comment) at 43s.

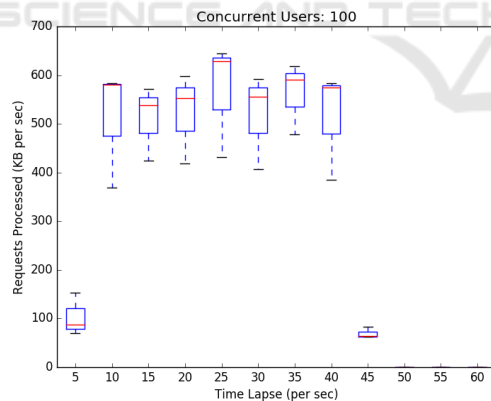


Figure 9: Hipster Critical Failure (Cart & Payment Services) at 43s.

ing the Database-per-service design pattern.

As before, Figure 10 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. Once the fault injector kills the critical microservice at 43s the application continues to serve pages, but throughput falls dramatically but briefly to around 2KB/s. By 50s the application is able to recover to a throughput of around 520KB/s.

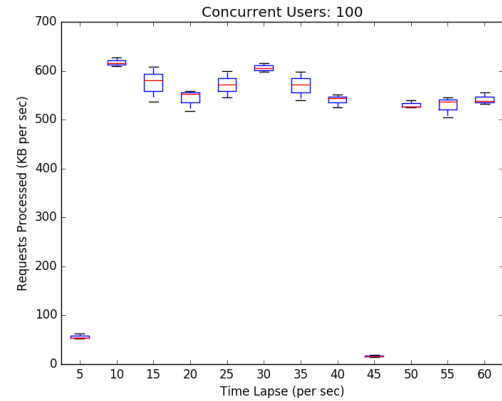


Figure 10: JPyL Minor/Individual Failure (Service Logging) at 43s.

6.3 Critical/Chained Failures

We next investigate the failure of critical/chained *chained* microservices. Specifically we target the chained/stateful/critical User Data & White/Black Listing microservices in JPyL and the chained/stateless/critical Product Catalog & Recommended microservices in Hipster. Both microservice chains implement patterns that aim to recover reliability (Section 4).

As before Figure 11 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. Once the fault injector kills the pair of microservices at 43s the application reports a *404 Service Not Found* error and continues to serve pages. However the throughput has fallen to around 2KB/s. That is the application is barely able to accept client requests or even load in a browser quickly. A similar failure is reported for Hipster when the Product Catalog service fails (Figure 12).

For realistic workloads the failure of chained/critical microservices even with pattern implementations has caused the applications to fail catastrophically!

6.4 Multiple Microservices Failure

Even if an application survives the failure of a single minor microservice, how will it cope when *multiple* microservices fail successively? To investigate the failure of multiple microservices in JPyL we target three microservices i.e. Service Logging, Security Headers & User Data – White/Black Listing. Specifically the fault injector kills these microservices in order at approximately 16s, 32s and 48s into the execution.

Figure 13 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. When the fault injector kills the *in*

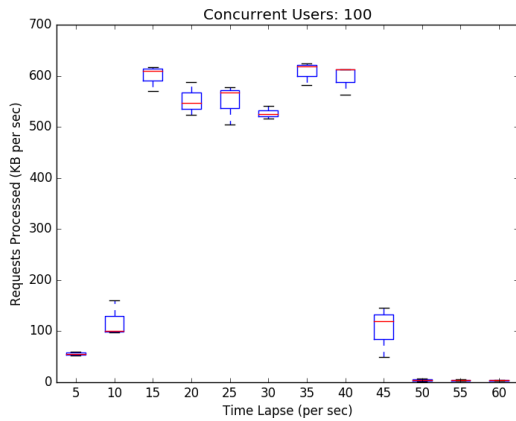


Figure 11: JPyL Chained (User Data & White-Blacklisting) at 43s.

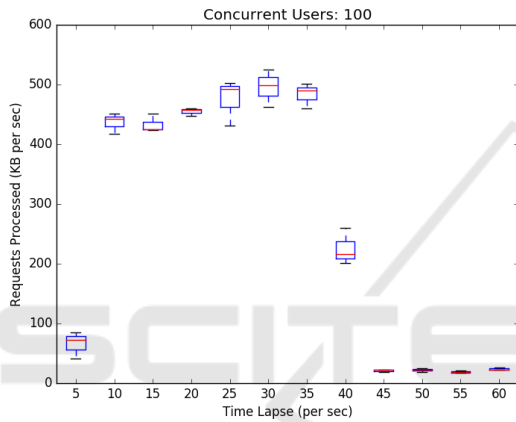


Figure 12: Hipster Chained (Product & Recommended) at 43s.

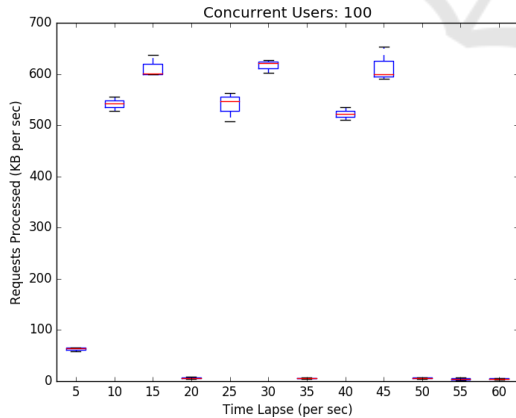


Figure 13: JPyL Multiple Minor Failures at 16s, 32s, 48s.

dividual/minor microservices the throughput drops briefly to around 2KB/s, but then recovers to around 600KB/s. As before, when the *chained*/critical microservice fails at 48s the application fails catastrophically.

6.5 Evaluation Summary

The key findings from our evaluation are as follows. (1) All case study applications fail catastrophically if a critical microservice fails (7, 8 & 9). (2) JPyL survives the failure of an individual/minor microservice (Figure 10), and even the successive failure of two individual/minor microservices (up to 40s in Figure 13) (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic: throughput being dramatically reduced (by 98%) (Figures 11, 12 and after 48s in Figure 13). (4) Individual microservices do not necessarily have minor reliability implications, e.g. the Hipster Frontend is individual/stateful/critical and the JPyL SSL is individual/stateless/critical (Figure 7).

7 CONCLUSION

Microservices are commonly classified based on their dependence (chained/individual) or state (stateful/stateless). We add a binary reliability classification, and combine it with the other classifications to define a three dimensional space: the MDSR Classification in Figure 4 (Section 4). Using three established web applications we exhibit microservices that exemplify the six MDSR classes. We outline a prototype static analyser that can statically identify all six classes in Flask web applications, and apply it to seven small web applications. Analysing the applications reveals that the majority of services are chained (70%), stateful (73%) and critical (77%) (Table 4). We speculate that the high percentage of critical services indicates that the applications are not designed for reliability.

We demonstrate that each of the MDSR critical case study microservices exhibits a known bad smell; and that each minor MDSR class in the case study microservices follows a design pattern (Taibi and Lenarduzzi, 2018). Across, the seven applications we find that 70% consist of the chained services bad smell by default while only 17% were implemented for resiliency as they consist of the Database-Per-Service pattern. Hence MDSR provides a framework to analyse the properties of microservices and chains of microservices in a system, identifying components to be considered for refactoring to improve reliability (Section 5).

In **future work** we plan to apply the MDSR classification to larger microservice systems, e.g. to Death Star⁸. It would be interesting to see whether MDSR could be extended to also classify failures in infrastructure services.

⁸<https://github.com/djmgit/DeathStar>

We also seek to enhance the analyser to make it more automatic, e.g. to automatically detect more persistent stores. Perhaps the analyser could suggest when some microservice resilience patterns, like bulkhead or load balancer, are implemented to reduce criticality? Finally we would like to investigate the potential of combining static MDSR analysis with dynamic SDG analysis.

REFERENCES

- Cabot, J. (2018). Wordpress: A content management system to democratize publishing. *IEEE Software*, 35(3):89–92.
- Cooper, R. (2014). BBC online outage on saturday 19th july 2014. <https://www.bbc.co.uk/blogs/internet/entries/a37b0470-47d4-3991-82bb-a7d5b8803771>.
- Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Comm. ACM*, 34(2):56–78.
- Ghirotti, S. E., Reilly, T., and Rentz, A. (2018). Tracking and controlling microservice dependencies. *Comm. ACM*, 61(11):98–104.
- Google (2021). Hipster-shop microservices demo. <https://github.com/GoogleCloudPlatform/microservices-demo>.
- Gupta, D. and Palvankar, M. (2020). Pitfalls and challenges faced during a microservices architecture implementation. Technical report.
- Hasselbring, W. and Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. In *IEEE Intl Conf on Software Architecture Workshops*, pages 243–246.
- Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. (2016). Gremlin: Systematic resilience testing of microservices. In *IEEE 36th Intl Conf on Distributed Computing Systems*, pages 57–66.
- IBM (2021). Microservices in the enterprise, 2021: Real benefits, worth the challenges. Technical report, Technical report, IBM.
- Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.
- Lewis, J. and Fowler, M. (2015). Microservices: A definition of a new architectural term. <https://martinfowler.com/articles/microservices.html>.
- Ma, S.-P., Fan, C.-Y., Chuang, Y., Lee, W.-T., Lee, S.-J., and Hsueh, N.-L. (2018). Using service dependency graph to analyze and test microservices. In *IEEE 42nd Annual Computer Software and Applications Conf*, volume 2, pages 81–86.
- Meng, Y., Zhang, S., Sun, Y., Zhang, R., Hu, Z., Zhang, Y., Jia, C., Wang, Z., and Pei, D. (2020). Localizing failure root causes in a microservice through causality inference. In *IEEE/ACM 28th Intl Symp on Quality of Service (IWQoS)*, pages 1–10.
- Montalenti, A. (2015). Kafkapocalypse: a post-mortem on our service outage. <https://blog.parse.ly/kafkapocalypse/>.
- Nikolaidis, E., Chen, S., Cudney, H., Haftka, R. T., and Rosca, R. (2004). Comparison of probability and possibility for design against catastrophic failure under uncertainty. *J. Mech. Des.*, 126(3):386–394.
- Offutt, A. J., Harrold, M. J., and Kolte, P. (1993). A software metric system for module coupling. *J. Systems and Software*, 20(3):295–308.
- Omer, A. M. and Schill, A. (2011). Automatic management of cyclic dependency among web services. In *2011 14th IEEE Intl Conf on Computational Science and Engineering*, pages 44–51.
- Patel, S. K., Rathod, V., and Prajapati, J. B. (2011). Performance analysis of content management systems-joomla, drupal and wordpress. *Intl J. Computer Applications*, 21(4):39–43.
- Power, A. and Kotonya, G. (2018). A microservices architecture for reactive and proactive fault tolerance in IoT systems. In *IEEE 19th Intl Symposium on "A World of Wireless, Mobile and Multimedia Networks"*, pages 588–599.
- Rossi, D. (2018). Consistency and availability in microservice architectures. In *Intl Conf on Web Information Systems and Technologies*, pages 39–55.
- Rudrabhatla, C. K. (2018). Comparison of event choreography and orchestration techniques in microservice architecture. *Intl J. Advanced Computer Science and Applications*, 9(8):18–22.
- Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., and Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. In *Proc. 11th Intl Conf for Internet Technology and Secured Transactions*, pages 318–325.
- Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *J. Systems and Software*, 146:215–232.
- Taibi, D. and Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62.
- Taibi, D., Lenarduzzi, V., and Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. In *CLOSER*, pages 221–232.
- Toffetti, G., Brunner, S., Blöchliger, M., Dudouet, F., and Edmonds, A. (2015). An architecture for self-managing microservices. In *Proceedings of the 1st Intl Workshop on Automated Incident Management in Cloud*, pages 19–24.
- Wolff, E. (2018). Why microservices fail: An experience report. Technical report.
- Wu, A. (2017). Taking the cloud-native approach with microservices. *Ma-genic/Google Cloud Platform*.
- Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., and Ding, D. (2018). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. on Software Engineering*.
- Zimmermann, O. (2017). Microservices tenets. *Computer Science-Research and Development*, 32(3):301–310.