Classifying the Reliability of Microservice Architectures

Abstract—**Microservices are popular as they offer better scalability and reliability than monolithic software architectures. Reliability is improved by the 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 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 demonstrate that each point in MDSR corresponds to a known reliability pattern or bad smell. Hence MDSR provides a structured classification of microservice software with the potential to improve code quality.

We explore the reliability implications of the different MDSR classes by running the case study applications against a fault injector, and show the following. All applications fail catastrophically if a critical microservice fails. Applications survive the failure of individual minor microservice(s). The failure of any chain of microservices in JPyL & Hipster is catastrophic. Individual microservices do not necessarily have minor reliability implications.

Index Terms—**microservices, reliability, web architectures, patterns, bad smells, web applications**

I. Introduction

Microservices are an increasingly popular software architecture, in part because they are perceived to offer better scalability and reliability than monolithic architectures. Some microservices are *stateful*: they record some information, e.g. the participants in a web chat. Others are *stateless* and record no information, e.g. they simply accept requests and process them.

Monolithic architectures are prone to *catastrophic failure*, where the user-visible functionality is suddenly and permanently unavailable [1]. That is, it is common for the failure of a single monolithic component to cause the entire system to fail. Cascade failures in web application stacks are a well known example.

In contrast microservice architectures potentially provide *improved reliability* due to the loose coupling of the services. So if one microservice fails, others will remain available. This may cause a reduction in throughput but will most likely avoid catastrophic failure [2]. In the worst case scenario, the loose coupling of services enables graceful failure [3].

This is achieved based on the design principle that microservices are implemented as standalone, independent services [4]. However, many industrial large scale web applications consist of different types of microservices like *chained* microservices. For example, in the Netflix Titus Platform, hundreds of individual and chained microservices can be found [5].

Chained microservices are usually tightly coupled, e.g. by high-frequency API-based interaction sequences. Chained microservices make an application far less reliable because if any of the services fail the entire chain fails, and may induce catastrophic failure [6]. For example, in 2014 BBC experienced a critical database overload that caused many of its critical microservices to fail one after another [7]. In 2015, Parse.ly experienced several cascading outages in their analytics data processing backend due to a microservices message bus overload [8].

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 microservice fails, the entire application fails. 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 [6], only six of the possible eight points in the space are valid (Section IV).
2) Using three web applications we exhibit microservices that exemplify each of the six points in MDSR. The web applications are: (1) *Hipster-Shop*, a Google demo application; (2) *JPyL*, a Jupyter Notebook/Flask web stack; and (3) *WordPress*, the well-known content management system (Section II).
3) We demonstrate that *each* point in MDSR corresponds to a known microservice pattern or bad smell [9]. Crucially MDSR identifies classes of microservices prone to critical failure and providing a framework for analysing the properties of microservices, and chains of microservices, in a system. (Section V).
4) We explore the reliability implications of different MDSR microservice classes by running Hipster, JPyL & WordPress against a simple process level fault injector. Specifically we show the following. (1) *All* applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of a minor microservice, and the successive failure 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 VI).

II. Example Microservices Web Applications

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

A. Hipster-Shop

Hipster-Shop is an e-commerce application with 10 microservices (Figure 1) used by Google to demonstrate tools like Kubernetes Engine [10]. Users can perform activities like viewing products, adding items to cart and making purchases. The microservices are developed in different languages like Python, Go and Java with communication taking place via gRPC remote procedure calls.

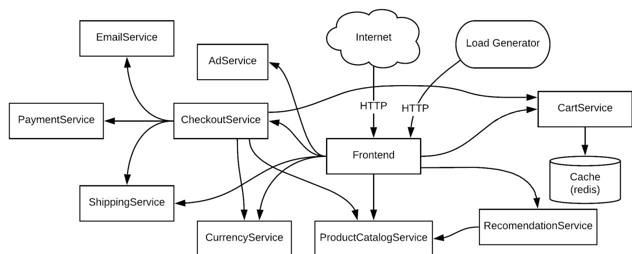

Fig. 1. Hipster-Shop Architecture Diagram[1]

B. JPyL: Jupyter/Python/Linux

The Jupyter Notebook framework is a web-based code development environment [11]. JPyL is a web stack that combines the popular Flask microservices web tier with monolithic Jupyter components (Figure 2). The Flask web tier has 7 microservices and is fairly conventional as outlined in Figure 2. Some are supported by a data store, e.g. current geolocation and IP address information are accessed via the userdata microservice that extracts the data from a MySQL database. Data is displayed on the webpage via the reverse proxy and port configuration microservices on port 10125. Crucially for reliability a backup URL port can be initiated via a redirect if the original port service is interrupted. Users will still be able to access the content and are automatically redirected to a different port.

JPyL features a Defense-in-Depth multi-layered security approach where a range of security mechanisms is deployed throughout the stack. The intention is that if an attacker penetrates one layer, another layer may thwart the attack. Each layer is handled by a specific Flask microservice or set of microservices. For example, security headers are processed by a Python Talisman microservice.

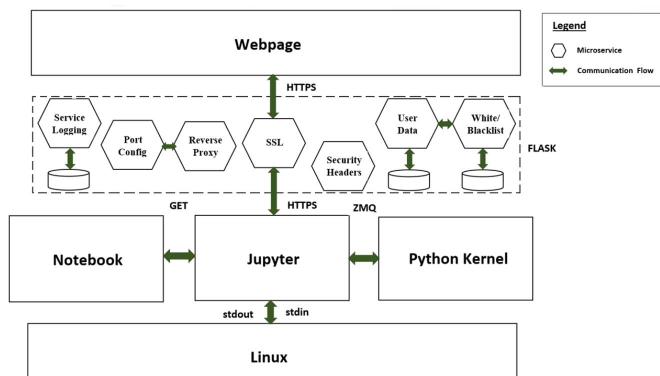

Fig. 2. JPyL Application Architecture

C. WordPress

WordPress is an open source web development and Content Management System (CMS) accounting for a significant proportion of online sites. One of the main reasons for the popularity of WordPress is its wide range of plugins that provide additional functionality [12].

As a standalone application WordPress has a monolithic architecture with core CMS components that communicate with a MySQL database. Microservices are, however, commonly integrated into WordPress to provide plugins that support additional functionalities—for example to provide facilities to post comments, allow subscription memberships, to search indexes, or to provide data analytics [13].

The application we study integrates microservice plugins that allow users to post comments on a blog. These services use the WordPress HTTP REST microservice that facilitates communication between the microservices plugins and the monolithic components as shown in Figure 3.

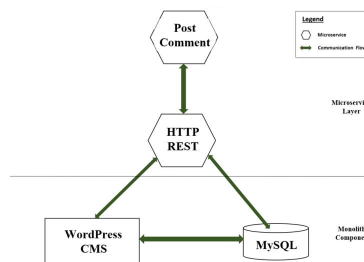

Fig. 3. Architecture of the WordPress Application Studied

III. Related Work

A. Existing Microservice Classifications

Fowler and Lewis identify the following three microservices design principles [14]:

[1]Source: https://github.com/GoogleCloudPlatform/microservices-demo

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 — should be via a REST API or message brokers.

Others have added other principles like reliability [6], and advocate using design patterns like timeouts, bounded retries, circuit breakers and bulkheads to tolerate failures [6]. 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 I.

1) Dependence: classifies a microservice by how tightly coupled it is with other microservices [5], [6], [15]. *Individual* microservices are loosely coupled to other microservices, with low dependency. Communication with other microservices is 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 [16]. Examples for JPyL include the SSL and Service Logging services (Figure 2) while Hipster-Shop includes Frontend and Adservice 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 with another service to function [6]. 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.

2) State: classifies a microservice by whether it preserves state between service requests. *Stateful* microservices require data storage, for example to record transactions or current actors [17]. 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) Combinations of properties, and inheritance of 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 (Figure 4).

B. Microservices Patterns and Bad Smells

Some design patterns capture reusable solutions to common microservice design challenges [9]. 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 [18], eliminating the single data store as a single point of failure (SPOF). While design patterns like Database-Per-Service help, they are not universal solutions. For example a single atomic operation often spans multiple microservices, and here additional techniques are required to ensure consistency across the data stores [19].

TABLE I
MICROSERVICES CLASSIFICATION CRITERIA

Classification	Properties	Description
Dependence	Individual	Loosely Coupled. Constant communication with other microservices not required.
	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 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.

Likewise microservice bad smells identify common designs that may cause issues [9]. Indeed [6] and the Fowler and Lewis design principles consider *all chained microservices* as bad smells and prone to reliability issues. For example the code snippet in Listing 1, reproduced from [6], shows an instance where a Cassandra microservice is dependent on the messagebus microservice to receive data. If the messagebus microservice is interrupted or fails, the Cassandra service will no longer receive the data it needs to function. This is an instance of the *Inappropriate Service Intimacy* bad smell [18].

Listing 1. Cascade failure between chained microservices arising from an *Inappropriate Service Intimacy* bad smell

```
Crash(cassandra)
  for s in dependents(messagebus):
     if not HasTimeouts(s, 1s)
        and not HasCircuitBreaker
        (s, messagebus, ...)):
        raise Will block on message bus
```

C. Microservices Reliability

There are substantial studies of the reliability of microservice software in both the academic [6], [20], [21] and grey literature [22], [23]. These reveal that the microservices 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. [24]

However, developers do not always implement the necessary design patterns to prevent microservices failure. Even if they do, they remain unaware whether their microservice can actually tolerate failures until it actually occurs [6]. Thus, the occurrence of failures in microservices should be expected at some point.

The literature distinguishes partial and catastrophic failures in microservices. Partial failures are typically temporary and recovery is automatic, e.g. the Docker container for some microservices may fail briefly, or a microservice without a load balancer can become overloaded [20]. Downtime can often be minimised if replacement microservice instance(s) are activated automatically [21]. 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 [1]. In microservices, catastrophic failures are often termed Interaction Faults [20]. 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 [20], 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 [6]. Moreover adding more microservices to the chain increases coupling [25] and the likelihood of catastrophic failure. If one service in the chain fails, there will be a cascade of failures of all services in the chain [6], [26].

A limitation of [6] and the Fowler and Lewis design principles is that they consider only dependence. We extend their work by simultaneously classifying dependence, state and reliability.

D. Detecting Failures and Identifying Causes

Detecting failure within microservices usually involves configuring a collection of microservices indicators (KPIs) to continuously monitor for root causes of failures. These are usually based on time series data. For example, the response level of a microservice is measured over a period of time to certain calls/requests from other services. The microservice will be deemed as having failed if it becomes anomalous [27].

In addition, diagnosing the severity and exact reason for a microservices failure in a large-scale service ecosystem is troublesome. The diagnosis usually requires domain and site-reliability knowledge as well as automated observability support [28]. Not all companies have such resources.

IV. Classifying the Reliability of Microservices

A. Critical and Minor Reliability

1) Critical vs Minor Microservices: To analyse the reliability of a microservice architecture we consider a reliability property alongside the established properties of state and dependency. Minor microservices provide non-essential functionality, and 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.

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.

It would be possible to consider partial failures as different from minor failures, but doing so requires distinguishing between minor and partial failures. For simplicity we select the binary minor/critical classification, although we shall see in section VI that some minor failures have very significant performance implications.

B. The Microservice Dependency State Reliability (MDSR) Classification

Combining reliability with the state and dependency classifications defines a three-dimensional space: our new Microservices Dependency State Reliability (MDSR) Classification. Figure 4 depicts the MDSR classification, and the first rows of the Patterns and Bad Smells tables below provide example microservices from the case study applications that correspond to the eight points in the classification space. For example the individual/stateful/minor exemplar is JPyL's Service Logging microservice.

Only six of the eight points in the space are valid, however, as microservice chains are necessarily critical [6]. We confirm this in our evaluation (Section VI) where, even though we attempt to recover reliability using of microservice patterns, the chains remain critical. That is, we implement a Database-Per-Service pattern for the chained/stateful UserData & White/Black Listing service and an API Gateway for the chained/stateless/ Product Catalog & Recommended service. In both cases the application fails catastrophically despite reporting only a "404 Service Not Found" error. Hence is not possible to find microservices that occupy the chained/stateful/minor or chained/stateless/minor classifications, i.e. the shaded columns in Figure 4.

The second rows of the Patterns and Bad Smells tables in Figure 4 show the errors produced when the example microservice at each classification point fails. For example when the individual/stateful/critical Frontend service fails, Hipster-Shop reports a "500 Internal Server Error" and fails catastrophically. In contrast when the individual/stateful/minor Service Logging service fails JPyL reports a "404 Service Not Found" and continues to operate.

V. Identifying Reliability Patterns & Bad Smells

MDSR classification makes it possible to analyse the expected properties of microservices and chains of microservices in an architecture. Specifically it helps to identify design patterns and bad smells in the architecture. To illustrate, the third row of the Patterns and Bad Smells tables in Figure 4 identify the microservice pattern or bad smell associated with each point in the classification space. Of the patterns and bad smells enumerated in [9], the classification identifies 4 out of 8 patterns and 3 out of 11 bad smells. The pattern and bad smell descriptions are summarised in Tables II & III.

In our applications individual/stateful microservices have only minor reliability implications if they implement a pattern. For example JPyL Service Logging is individual/stateful/minor and implements the Database-Per-Service pattern.

Most critical microservices are associated with known bad smells. 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.

Hence MDSR provides a useful analysis of microservice software, helping to identify bad smells to be considered for refactoring to improve reliability. MDSR incorporates the principle that chained microservices are critical, even if they implement patterns in an attempt to recover reliability.

TABLE II
MICROSERVICES PATTERNS DESCRIPTION [18]

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

TABLE III
MICROSERVICES BAD SMELLS DESCRIPTION [9]

Bad Smell	Description
SRP Violation[2]	Microservice performs more than a single functionality. Reason: Tight Coupling
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)
Cyclic Dependency	Constant call cycles between microservices Reason: Too much dependency

[2]Not listed in the Taibi bad smells classification.

VI. Evaluating the Reliability of Different Classes of Microservices

A. Experiment Design

We execute the Hipster, JPyL & WordPress web applications against a simple fault injector to investigate the reliability implications of different classes of microservices. All applications are executed on a typical server, i.e. a 16 core Intel server with 2TB of RAM 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, the experiments, and the fault injector are available in a Bitbucket repo[3].

Requests are generated to each application using wrk 1.2 [29] for sixty seconds with a simulated 100 concurrent users. The reported measurements are based on three consecutive benchmark executions, and identify the median.

This fault injector is implemented in Python using the Chaos Monkey Engine 1.1.0 [30] to terminate the process associated with a specific microservice at a specific time. As neither process nor termination time is selected at random, this is not a Chaos Monkey.

B. Critical Microservice Failure

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 chained/stateless/critical Port Config microservice in JPyL, the individual/stateless/critical SSL microservice in JPyL & the chained/stateful/critical Cart Service in Hipster.

Figures 5, 6, 7 & 8 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.

C. Minor/Individual Microservice Failure

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 following the Database-per-service design pattern.

As before, Figure 9 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.

[3]URL removed for double blind review.

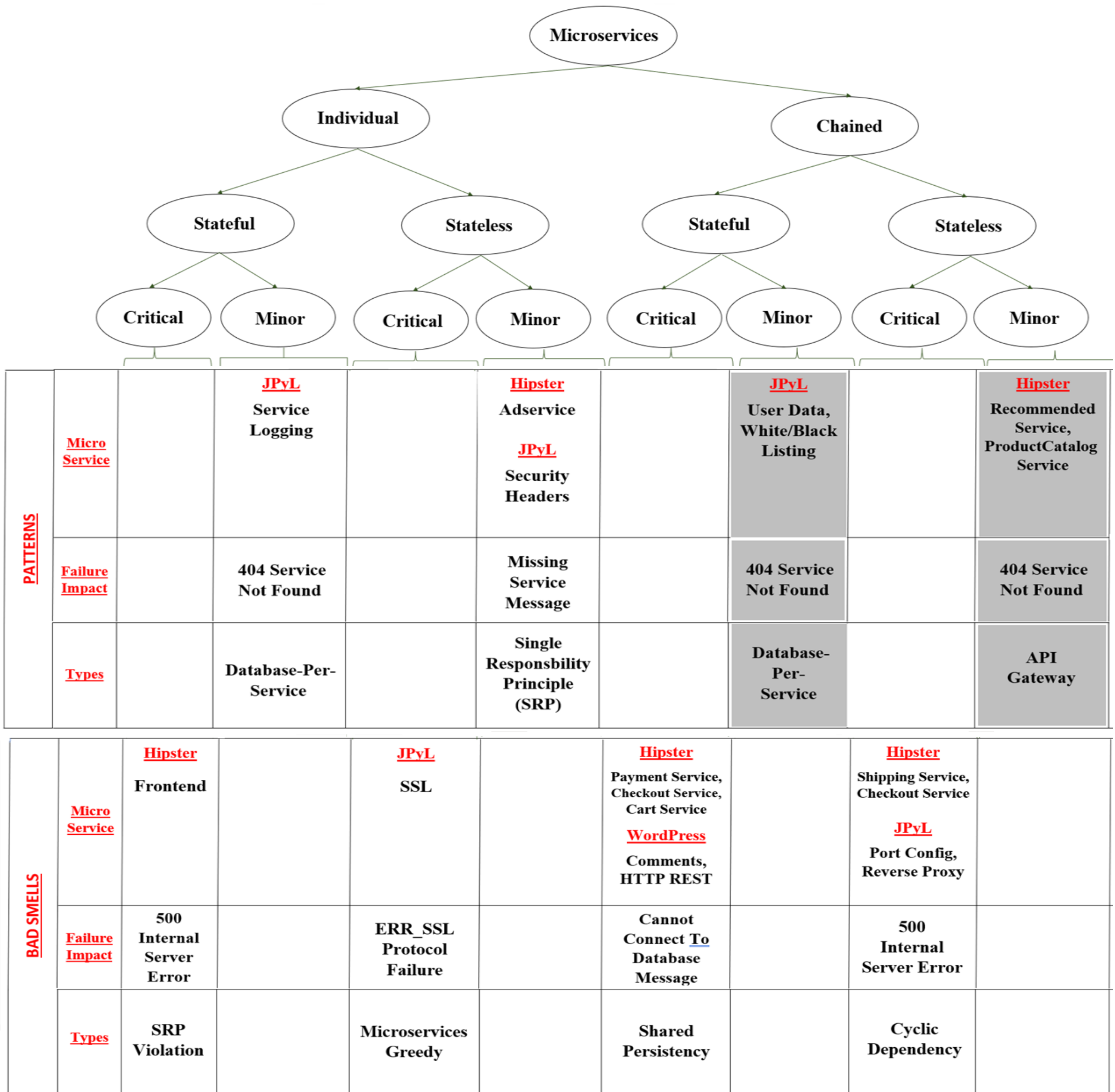

Fig. 4. Microservices Dependency State Reliability (MDSR) Classification

D. Critical/Chained Design Pattern Microservices Failure

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 IV).

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 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 11).

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

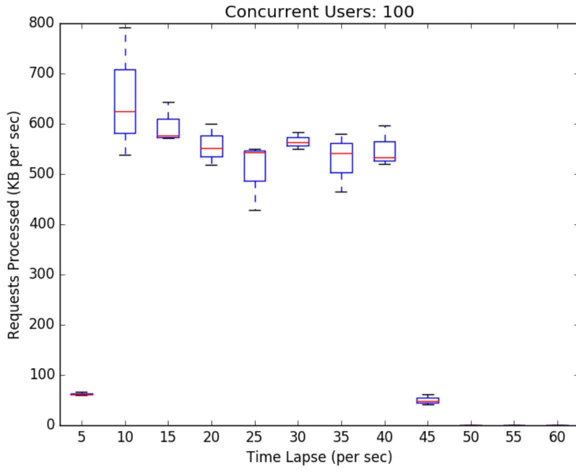

Fig. 5. A JPyL Critical Failure (Port Config & Reverse Proxy) at 43s.

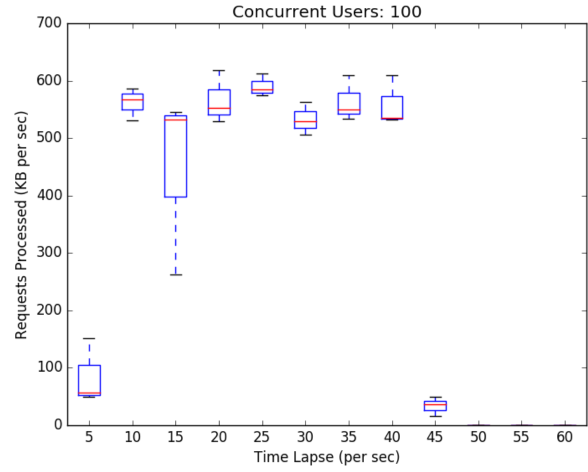

Fig. 7. A WordPress Critical Failure (HTTP REST & Comment) at 43s.

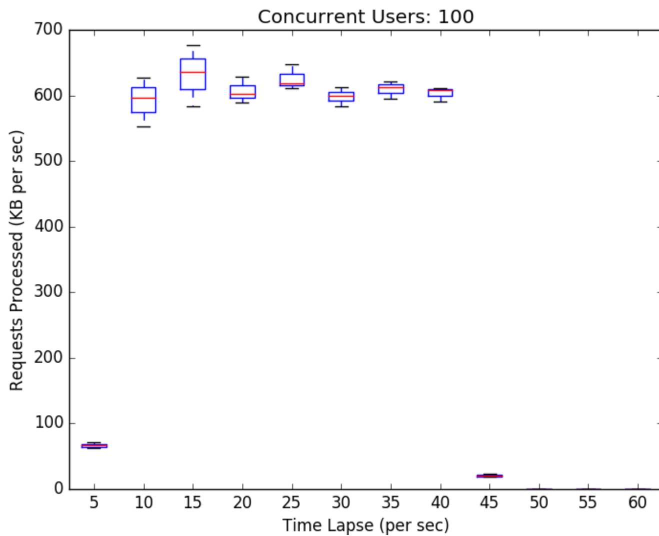

Fig. 6. A JPyL Critical Failure (SSL) at 43s.

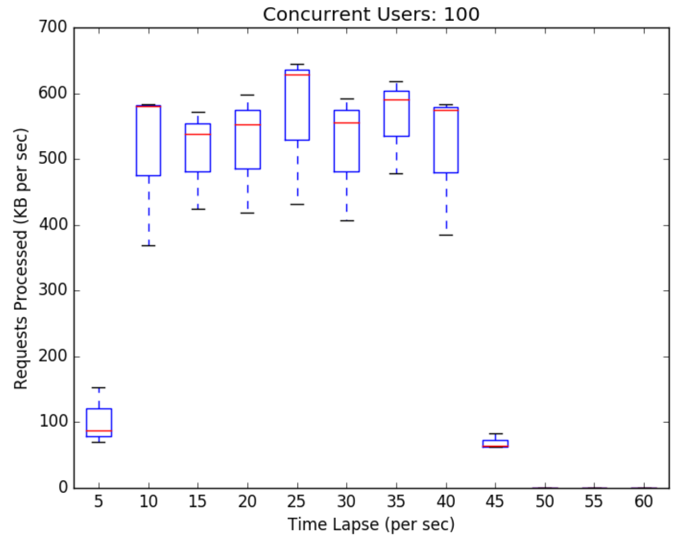

Fig. 8. Hipster Critical Failure (Cart & Payment Services) at 43s.

E. 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 12 plots JPyL throughput (Request KB/s) against time, and the service is running at around 600KB/s. When the fault injector kills the *individual*/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.

F. Evaluation Summary

The key findings from our evaluation are as follows. (1) All case study applications fail catastrophically if a critical microservice fails (Figures 5, 6, 7 & 8). (2) JPyL survives the failure of an individual/minor microservice (Figure 9), and even the successive failure of two individual/minor microservices (up to 40s in Figure 12) (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic: throughput being dramatically reduced (by 98%) (Figures 10, 11 and after 48s in Figure 12). (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 6).

VII. Conclusion

Microservices are commonly classified based on their dependence (chained/individual) or state (stateful/stateless). We

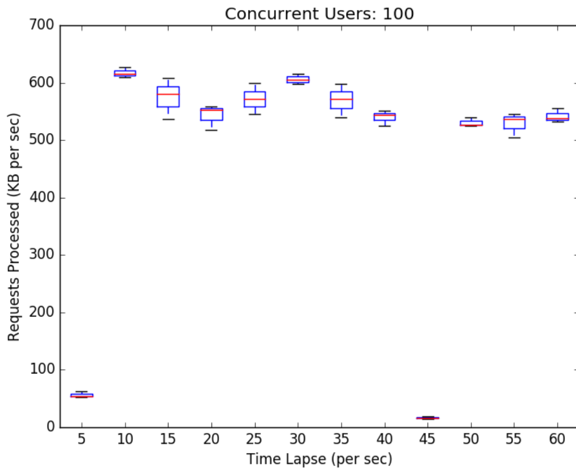

Fig. 9. JPyL Minor/Individual Failure (Service Logging) at 43s.

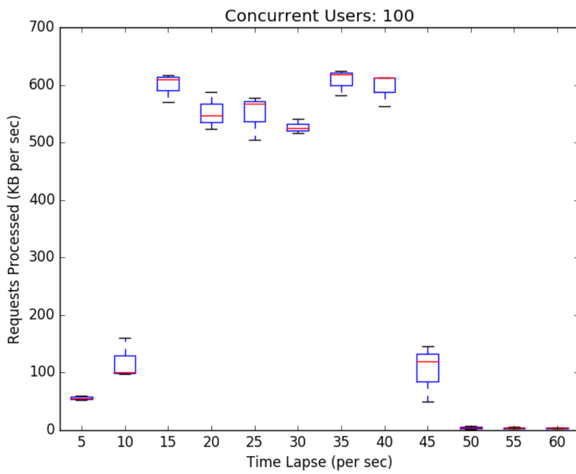

Fig. 10. JPyL Minor/Chained (User Data & White-Blacklisting) at 43s.

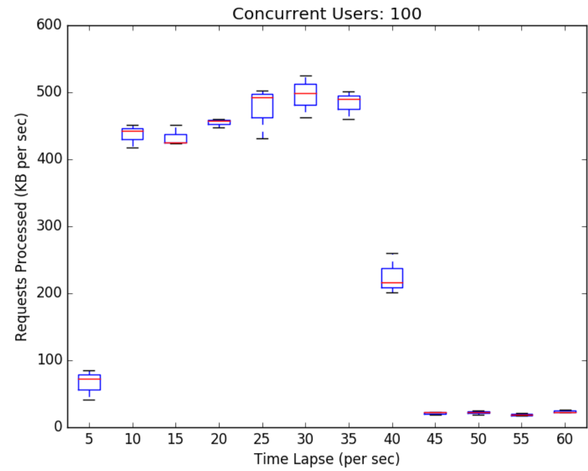

Fig. 11. Hipster Minor/Chained (Product & Recommended) at 43s.

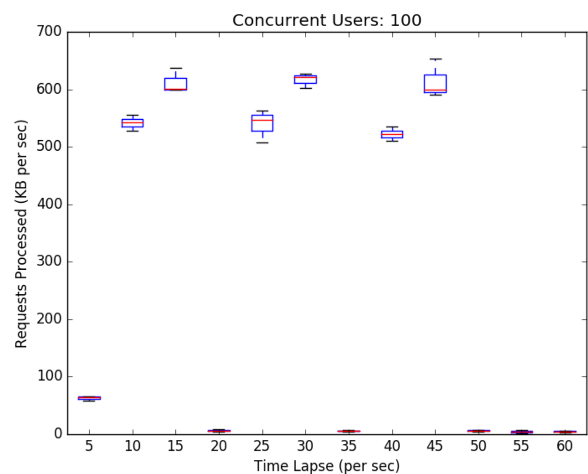

Fig. 12. JPyL Multiple Minor Failures at 16s, 32s, 48s.

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 IV). Using three established web applications we exhibit microservices that exemplify six of the eight points in MDSR. The remaining points in the space are not valid as microservice chains are necessarily critical as argued, e.g. by [6], and confirmed in our evaluation.

The tables in Figure 4 show that each point in MDSR corresponds to a known microservice pattern or bad smell [9]. 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 V).

We investigate the reliability implications of MDSR classes by running the case study applications against a simple fault injector under realistic workloads to show the following. (1) *All* applications fail catastrophically if a critical microservice fails. (2) Applications survive the failure of individual minor microservice(s). (3) The failure of any chain of microservices in JPyL & Hipster is catastrophic. (4) Individual microservices do not necessarily have minor reliability implications.

In **future work** we plan to investigate larger microservice-based systems to accumulate evidence for the effectiveness of our MDSR Classification in identifying reliability bad smells. We also seek to investigate whether the analysis could be automated, with a view to complementing Service Dependency Graphs (SDG) tools that map microservices relationships.

References

[1] E. Nikolaidis, S. Chen, H. Cudney, R. T. Haftka, and R. Rosca, “Comparison of probability and possibility for design against catastrophic failure under uncertainty,” *J. Mech. Des.*, vol. 126, no. 3, pp. 386–394, 2004.

[2] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges when moving from monolith to microservice architecture,” in *International Conference on Web Engineering*. Springer, 2017, pp. 32–47.

[3] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, “The pains and gains of microservices: A systematic grey literature review,” *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.
[4] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.
[5] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, “Using service dependency graph to analyze and test microservices,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2018, pp. 81–86.
[6] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, “Gremlin: Systematic resilience testing of microservices,” in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 57–66.
[7] R. Cooper, “BBC online outage on saturday 19th july 2014,” 2014, https://www.bbc.co.uk/blogs/internet/entries/a37b0470-47d4-3991-82bb-a7d5b8803771.
[8] A. Montalenti, “Kafkapocalypse: a postmortem on our service outage,” 2015, https://blog.parse.ly/kafkapocalypse/.
[9] D. Taibi and V. Lenarduzzi, “On the definition of microservice bad smells,” *IEEE software*, vol. 35, no. 3, pp. 56–62, 2018.
[10] Google, “Hipster-shop microservices demo,” 2021, https://github.com/GoogleCloudPlatform/microservices-demo.
[11] J. M. Perkel, “Why jupyter is data scientists’ computational notebook of choice.” *Nature*, vol. 563, no. 7729, p. 145, 2018.
[12] S. K. Patel, V. Rathod, and J. B. Prajapati, “Performance analysis of content management systems-joomla, drupal and wordpress,” *International Journal of Computer Applications*, vol. 21, no. 4, pp. 39–43, 2011.
[13] B. Williams, D. Damstra, and H. Stern, *Professional WordPress: design and development*. John Wiley & Sons, 2015.
[14] J. Lewis and M. Fowler, “Microservices: A definition of a new architectural term,” 2015, https://martinfowler.com/articles/microservices.html.
[15] S. E. Ghirotti, T. Reilly, and A. Rentz, “Tracking and controlling microservice dependencies,” *Communications of the ACM*, vol. 61, no. 11, pp. 98–104, 2018.
[16] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, “The evolution of distributed systems towards microservices architecture,” in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2016, pp. 318–325.
[17] A. Wu, “Taking the cloud-native approach with microservices,” *Magenic/Google Cloud Platform*, 2017.
[18] D. Taibi, V. Lenarduzzi, and C. Pahl, “Architectural patterns for microservices: A systematic mapping study.” in *CLOSER*, 2018, pp. 221–232.
[19] C. K. Rudrabhatla, “Comparison of event choreography and orchestration techniques in microservice architecture,” *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 8, pp. 18–22, 2018.
[20] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering*, 2018.
[21] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, “An architecture for self-managing microservices,” in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, 2015, pp. 19–24.
[22] E. Wolff, “Why microservices fail: An experience report,” Tech. Rep., 2018.
[23] D. Gupta and M. Palvankar, “Pitfalls and challenges faced during a microservices architecture implementation,” Tech. Rep., 2020.
[24] A. Power and G. Kotonya, “A microservices architecture for reactive and proactive fault tolerance in iot systems,” in *2018 IEEE 19th International Symposium on” A World of Wireless, Mobile and Multimedia Networks”(WoWMoM)*. IEEE, 2018, pp. 588–599.
[25] S. Eski and F. Buzluca, “An automatic extraction approach: Transition to microservices architecture from monolithic application,” in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, 2018, pp. 1–6.
[26] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 2015, pp. 583–590.
[27] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, “Localizing failure root causes in a microservice through causality inference,” in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 2020, pp. 1–10.
[28] W. Hasselbring and G. Steinacker, “Microservice architectures for scalability, agility and reliability in e-commerce,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 243–246.
[29] W. Glozer, “wrk http benchmarking tool on github,” 2019, https://github.com/wg/wrk.
[30] PyPi, “Chaos monkey engine project description,” 2017, https://pypi.org/project/chaosmonkey/.