



An Empirical Study on Microservice Software Development

Francisco Ramírez ^{‡*}, Carlos Mera-Gómez [†], Rami Bahsoon* and Yuqun Zhang^{‡§}

[‡] Southern University of Science and Technology of China

Email: 11756009@mail.sustech.edu.cn, zhangyq@sustech.edu.cn

*School of Computer Science,

University of Birmingham, Edgbaston, B15 2TT, UK

Email: {fmr067, r.bahsoon} @cs.bham.ac.uk

[†] ESPOL Polytechnic University, Escuela Superior Politécnica del Litoral, ESPOL,

Facultad de Ingeniería en Electricidad y Computación,

Campus Gustavo Galindo Km 30.5 Vía Perimetral, P.O. Box 09-01-5863, Guayaquil, Ecuador

Email: cjmera@espol.edu.ec

[§] Corresponding author

Abstract—Microservice is an approach to software development, in which an application is designed and constructed to maximise the benefits of modularisation. This approach improves the testing of applications, where modularization can limit the propagation of bugs and facilitates their detection. Though the microservices paradigm has the potentials to ease the automation of bugs detection and fixing, the process is still less understood by the microservices community. To bridge this gap and accelerate such an understanding, we extracted posts from Stack Overflow to identify the most commonly discussed issues in microservice development and testing and we categorised the concerns. Our results indicate that (i) missing parameters and operations are the most common concerns in service routing; (ii) wrong versions of libraries, annotations, protocols, and clusters appear as the main concerns on service discovery; (iii) the absence of authorisation operation and web tokens exposed for a long time are the main concerns related to service authentication and authorisation; and (iv) the absence of configuration parameters for cache and inadequate patterns for long-running transactions emerge as trending concerns in service invocation. We analyse our findings and provide suggestions for future research.

I. INTRODUCTION

Microservice is an approach towards software development, in which an application is designed and constructed leveraging the benefits of modularisation through a set of small and highly cohesive services that coordinate and interact each other using lightweight mechanisms [1], [2]. This approach enables fast development of applications and has the potentials to localise bugs, limits their propagation within the system, and facilitates their automated detection and repair [3]–[6]. However, bug localisation/debugging are complex due to efforts required to trace the failed requests among a number of log files spread on different containers [7]–[9]. The process of locating/fixing bugs is eventually one of the core purposes of adopting microservices (i.e. designing for failure/recovery); the effectiveness and efficiency of the process are important for ensuring availability of the system, compliance with its Services Level Agreements and maintaining its revenue streams [10].

Although the automation of bug correction helps to reduce the time during testing stage; in practical terms, detecting bugs on running microservices may take days or even weeks [11]. For example, a high number of microservices implies a high number of vulnerability points, which means a large attack surface [12]; also, a bug propagation across dependent microservices [10] complicates the identification of the root of the issue. Moreover, the effectiveness of existing tools in detecting bugs varies with the context [11].

Developer’s information sourcing is commonly used in practice for learning about development issues and their solution [13]; this is simply through exploring the posts and the suggested answers, comments and coding practices [14]. However, our research has shown that systematic compilation and classification of microservices issues, including bugs, their symptoms and their root causes have been lacking.

To bridge this gap, the novel contribution of this paper is an empirical study that analyse 406 posts from Stack Overflow [15], one of the largest source for developers [14], where we looked at the bugs, their symptoms, root causes. We have then categorised the posts in terms of the development lifecycle phases, quality attributes, software construction activities and fault classes [16]–[18]. We chose Stack Overflow for the following reasons: (i) it is the most important online platform for sharing development knowledge and experience; and (ii) many microservice architects, developers, and operators interact through this platform to shape and evolve their practices [19]. For each of the collected posts from Stack Overflow, we inspected the full thread from the question to every corresponding response and comment.

Despite the significance of Stack Overflow as a source of common concerns in microservice development, only one previous work [20] has reported a taxonomy on microservices by using topic modelling over Stack Overflow posts. Its taxonomy captures technical and conceptual topics based on words frequency. However, its study excludes all the comments, which provide clarification and relevant information about

the posts [21]. Different from that work, our study identifies common development concerns from technical posts and categorises them considering microservices lifecycle, software construction, quality attributes, and fault classes. Moreover, previous empirical studies on microservices have either (i) introduced algorithms using a reduced number of issues to evaluate their debugging approaches [9], [11], [22]–[24]; or (ii) interviewed a limited number of developers to identify undesired practices [25] or code smells [2] in microservice-based applications. Different from those efforts, our study presents the collection of issues from an open data source to discuss concerns and draw conclusions about the construction of microservices from community perspective.

The remainder of this paper is organized as follows. Section II describes the life cycle of microservices at runtime, whereas the design of our study is presented in Section III. Section IV presents the results of this study. Section V discusses the threats to validity of our study and future research works, followed by a discussion of related works in Section VI. Finally, Section VII summarizes our work.

II. THE LIFE CYCLE OF MICROSERVICES AT RUNTIME

The life cycle of a microservice-based system at runtime is triggered by a request (e.g. Hypertext Transfer Protocol/HTTP, Advanced Message Queuing Protocol/AMQP) and this life cycle is composed of four system activities: service routing, service discovery, service authentication & authorization, and service invocation. Figure 1 shows the process that starts with service routing and ends with service invocation, which calls the microservices and return the required response.

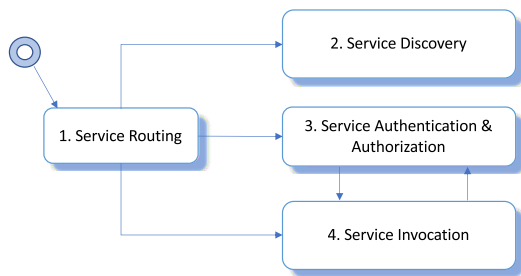


Fig. 1. Microservice-based system lifecycle at runtime

Service routing has an entry point that routes all requests to other microservices [26]. Specifically, the API gateway is the component that receives an initial request and examines the availability of a microservice, selects its right version and collects the responses of the required microservice [27]. In case of asynchronous responses, API gateways implement *polling* for capturing them. Frameworks such as *Kong*, *JHipster*, and *Swagger* [28] help to construct gateways. The service routing communicates with all the other life cycle activities.

Service discovery collects microservices health status (i.e. location, IP address, and ports of available instances) and keeps this information in a component called register, which

returns these details when requested. Additionally, load balancer, databases, repositories, and other components are initialized in this activity. These components may use a configuration manager to initialise and adjust the system performance depending on changing environments or workloads.

Service authentication & authorization validates user credentials and permissions. If a request requires authentication, the user credentials are validated. If a request requires authorization, the access permissions are also checked. Depending on the implementation, this activity communicates to the service invocation activity to propagate access permissions to other microservices. This activity needs the use of several security protocols such as: (i) *JSON Web Token (JWT)* [28], which is an open source JSON-based standard [28] that enables tokens as a medium for secure message transmission; and (ii) *OAuth* [29], a standard authorization protocol focused on simple client development for Web applications.

Service invocation is the activity in which one or more microservices are invoked. These invocations are performed in either a synchronous or an asynchronous manner. The former use invocation patterns such as *aggregation*, *chain*, *proxy* [30]; whereas the latter use tools based on messaging broker, streaming, and cache to send data to the invoked microservices. Regarding the origin of the service requests, most of these are received from the service routing, but some other requests come from the service authentication & authorization due to permissions propagation. For the requests that come with a token to establish a secure invocation, the validity of the token is checked before processing the request.

III. STUDY DESIGN

In this section, we present the key aspects of our study design, which follows well-established guidelines on systematic studies [31] [32].

A. Research Question

The purpose of this study is to gain insights into main concerns in microservices development. This study is an effort to (i) prevent junior developers from introducing accidental errors during the implementation of distributed patterns; and (ii) suggest future research on static code analysis to recognise and fix the misuse of annotations. In this study, we investigated the following research question (**RQ**): *Which are the most commonly discussed development concerns of the microservice community?* By answering this question, we aim to provide empirical evidence of existing microservice development concerns classified according to different software engineering aspects. Additionally, our findings provide specific implications for future research on complex systems.

B. Search and Selection Process

Figure 2 shows the stages of our search and selection process and the number of Stack Overflow posts at the end of each stage. For a better control on the characteristics of the posts, we considered the following stages in the design of our study: initial search, accepted answer criteria, conceptual

and technical criteria, merging and duplication removal, and application of selection criteria.

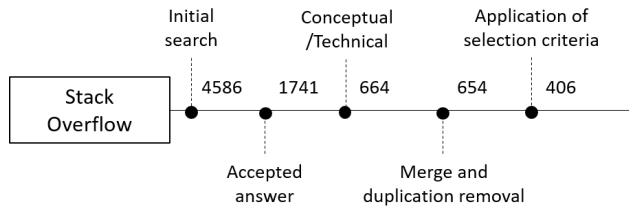


Fig. 2. Numbers and stages of our search and selection process

1. Initial search. In this stage, we performed searches on Stack Overflow [31] since it is one of the largest and most popular online Question & Answer (Q&A) forums [14], in which developers post programming issues [13]. We used the tag *microservices* as the main criterion and we limited our search to posts dated until December of 2019.

2. Accepted answer criteria. A post could have multiple answers; however, the owner of the post can select only one answer as accepted. We selected the post with an accepted answer by using Query Stack Overflow [33], which is a website to execute simple Structured Query Language (SQL) statements against public data from Stack Overflow.

3. Conceptual and technical criteria. We manually classified the posts in two main categories, namely conceptual and technical. On one hand, a post is considered as conceptual when the question is too general and non-technical details are found in the post, answers or comments. On the other hand, a post is considered as technical when the question is more specific and their answers/comments contain technical text. For the next stage, we only chose posts that were categorised as technical.

4. Merging and duplication removal. Moderators on Stack Overflow mark posts as closed when they are duplicated, off-topic, or unclear. Based on this, we excluded closed and duplicated posts. Additionally, we only collected posts suggested as main threads.

5. Application of selection criteria. We collected all the posts and manually filtered them according to the following inclusion (I) and exclusion (E) criteria:

- I1) *Microservices* posts created until December 2019.
- I2) Posts with an accepted answer.
- I3) Technical posts for microservice implementation.
- E1) Posts with duplicated content
- E2) Posts with a status of closed.

C. Data Extraction and Synthesis

In this stage, (i) we classified the posts according to the activities in the microservices life cycle, quality attributes, software construction activities, and fault classes; and (ii) we collected data for our study such as publication year, bug symptom, root cause, and system activity.

The data synthesis involved a collection and a summary of the data extracted from the posts [34]. It was aimed at understanding and analysing posts on microservices development.

Specifically, we performed content analysis (categorization of posts) and narrative synthesis (explanation of findings coming from the content analysis).

For the narrative synthesis, we created groups based on similar contexts by using the extracted bug symptom(s) and root cause from each post. We discussed the grouping of each post until we reached a consensus.

D. Replicability of Our Study

For the sake of the replicability of our study, we provide a replication package¹ for interested readers. The replication package includes: the research protocol, the SQL scripts, the raw data with the list of the retrieved posts, the extracted and categorised data, and the scripts for generating the information charts.

IV. RESULTS

We present our observation of posts over the years in Section IV-A and the main findings per life cycle activity in Section IV-B. Additionally, in Section IV-C, we categorise the posts based on quality attributes [16], software construction activities [17] and fault classes [18]. Since referencing hundreds of posts occupies a significant space, we decided to identify the posts using the letter *P* (from post) followed by a unique hexadecimal number (e.g. P1B).

A. Years vs Life Cycle Activities

Figure 3 shows the distribution of posts on microservice development over the years. Although only a small number of post were created during 2014, this was the year when large organizations become interested on microservices [35]. Moreover, the trend indicates that the activity of the community in microservice-related topics has been steadily growing among the years, except 2019 which has a reduction of 27% of the posts in comparison to 2018.

We also noticed that the life cycle activity with the highest number of posts is *service discovery* (161/406), followed by *service invocation* (136/406), *service routing* (69/406), and *service authentication & authorization* (40/406).

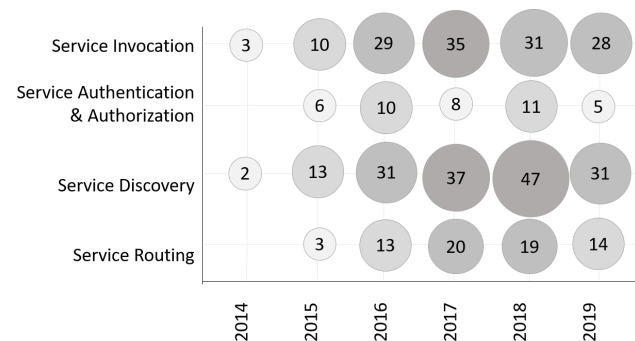


Fig. 3. Life cycle activities over time

¹<http://www.research.propio.click/paper-empirical/replication-package/>

B. Findings per Life Cycle Activities

We categorise the posts based on their pertinence to *service routing*, *service discovery*, *service authentication & authorization*, and *service invocation*.

1) **Service Routing:** Table I shows the service routing distribution. The main tasks in service routing are *client discover* 35/59, followed by *send request* 18/59 and *polling response* 6/59. *Client discover* refers to diverse issues that occur in the implementation of clients intended to discover microservices; for instance, a multipart request for uploading files requires to discover a microservice with an overridden UTF-8 configuration (P7D). *Send request* refers to issues that occur when routing requests such as configuration of cross-origin in Zuul (P12B) [28]. *Polling response* refers to issues that occur when receiving an asynchronous response; for instance, an interruption in the navigation flow of a movie web application caused by the lack of a mechanism to periodically observe a resource status (PE1).

We observed that 40% of the posts in client discover are related to missing parameters or operations. The former, missing parameters, refers to the absence of parameter names and values during the invocation of scripts or configuration files; whereas the latter, missing operations, refers to the absence of sentences for adding specific behaviour such as reporting a health status.

Regarding missing parameters, we found that environment variables, security on network access, framework settings, health metrics, and values of annotations are the most common concerns. For instance, the variables *HOST* and *PORT* require valid values to connect a remote machine (P59, P85, PD7), a configuration management as Zookeeper is required to hold the parameters for frameworks such as Zuul, Consul, gateways (P3E, PC6), and annotations such as *ConsulConfiguration* and *EnableSwagger* requires values to fetch data, generate documentation and others (PF6).

In relation to missing operations, we found that the most common operations include the implementation of variables, getters, setters, and rules for service discovery. For instance, Swagger requires *ApiImplicitParam* on getters and setters to generate documentation (PA7), service discovery clients ignore the different DNS ports to get a list of microservice instances from SRV records (P11, P55, P89), and developers miss rules for checking the version/health/context of microservices in Zuul, Nginx or vector clocks (P15, P76, PCB).

We observed that 56% of the posts in *send request* are related to wrong configuration, 22% of posts are related to missing publish events and missing format for case class, and 6% of posts refer to missing annotations. For instance, the wrong configuration issues include the settings of environment (e.g. Eureka, queue messaging, replication), relative paths, security properties, retries (e.g. waiting time for next retry, maximum number of retries) (P36, P39, P4F, P8F, PA8, PAA, PD4 P124, P12F, P143). In the case of missing publish events and missing format for case class, we observed that ActiveMQ, RabbitMQ, Kafka, Spring Framework and Spray JSON are

required components. Unknown Exception occurs when the CrossOrigin annotations are missed.

Finding 1: *Missing parameters emerges as one of the most popular issues on service routing. Results suggest API gateway frameworks require those parameters for configuration.*

Implication: *Developers need to fix missing parameters and environment variables for Somata, Cassandra, and Kong [28], which may contribute to reduce bugs at the compilation stage.*

Finding 2: *Missing operations emerges as a popular topic on service routing. Among these kinds of operations, we have ApiImplicitParam for the generation of documentation, SRV record to list microservices, and monitor of the health status.*

Implication: *Developers need a deeper understanding of frameworks such as Consul, Mesos-DNS, Nginx, Swagger, and Zuul [28]. A guide of missing operations in these frameworks could help to mitigate part of these issues.*

2) **Service Discovery:** Table I also presents the distribution of posts related to service discovery activity. The main tasks in service discovery are *configuration* 110/142, followed by *start* 17/142 and *registry* 15/142. *Configuration* refers to the lack of values, parameters, or annotations; for instance, if a parameter discovery is disabled then localhost and port between Java Spring and Consul are unresolved (P23) [28]. *Start* refers to issues that occur when starting components such as using a wrong package version of JHipster when launching a microservice (P112). *Registry* refers to issues that occur when components interact against the registry service; for instance, Eureka is a registry service [28] and Eureka clients are still working despite a Eureka server stops (P121).

Regarding the configuration task, 24% of the posts are related to the usage of frameworks such as Spring, Netflix, Cassandra, Zookeeper, GemFire and others [28]. For example, we have issues related to (i) lack of values for configuration of fallback, and circuit breaker and lack of parameters for dependency injection or binding exception; (ii) wrong version of libraries; (iii) annotations of Spring Boot, Lombok and GemFire are missed/misused [28]; (iv) data serialization provides communication outside private networks; and (v) missing fault-tolerant clusters to avoid single point of failure. For instance, query performance is very low when key filter is not an enumeration (P3A, P57, P67, P6C, PFA), BindingException occurs when server port is absent of the Java Virtual Machine (JVM) parameters (P22, P2E, P80, PBC, PD1, PE4, P11D, P134), unsupported class version occurs when using a higher version of servo-core dependency (P33, P51, P6B, P97, PA9, P92), *UnknownHostException* may occur when *RestTemplate* miss the *Autowired* annotation (P47, P3C, PC9, PD0), front-end microservices should return a *ResponseEntity* serialized as JSON to include the HTTP status code (PB2, PD8, P17, P93), and a system lose events if a central messaging hub has no cluster and becomes unavailable (P2, P98, PE5, PB1).

Regarding the start task, 47% of the posts are related to the usage of libraries for dependency (P7, PB3, P112), for instance, bootRepackage is no longer supported by JHipster version 5+; container initialization due to wrong login pa-

TABLE I
CLASSIFICATION OF POSTS

System Activity	Activity Task	Task Description	Issue Category	Ratio
Service routing	Client discover	Issues in the implementation of clients that discover microservices	Missing parameters Missing operations	9.9%
	Send request	Wrong settings for routing of request	Wrong configuration of environment Missing publish event Missing serialise mechanism Missing annotations	5.4%
	Polling response	Incomplete asynchronous response	Missing response (message, record id, promise or link) Missing event listener	1.7%
Service discovery	Configuration	Missing values, parameters or annotations for settings of components	Wrong configuration of frameworks Wrong usage of parameters Wrong version of dependencies Missing annotations	30.3%
	Start	Failures when starting components	Wrong version of dependent libraries Wrong login options Missing operations for services	5.7%
	Registry	Issues during interaction with registry service	Load balancing error caused by different version of microservices	3.7%
Service authentication & authorization	Authentication	Incomplete validation of a user account	Missing operations Missing Web tokens	5.2%
	Authorization	Deny the access to resources	Missing operations Missing Web tokens Missing event messages	3.4%
	Grants	Incomplete adoption of user roles to keep authentication & authorization	Wrong data usage High workload High coupled microservices	1.2%
Service invocation	Asynchronous	Errors when calling microservices simultaneously	Common cache errors Missing patterns for distributed systems	21.9%
	Synchronous	Interruption of calling microservices sequentially	Cross-origin resource sharing (CORS) Broken contracts Outdated databases	8.1%
	Token validation	Errors when validating secure tokens	Missing implementation Missing Token Unsecure zone	3.4%

rameters (PBB, PE0), for instance, docker containers require the option `--with-registry-auth` to forward credentials to other nodes; and operations for a high number of services such as backup/restoration, logs, and scaling (P18, P35, PF2), for instance, if there is no configuration for timeouts and retries then the system could have scaling troubles.

In relation to the registry task, 40% of the posts are concerned about a load balancing between different versions of microservices (P16, P108, P10D), for instance, load balancer considers a version of microservices unless the older version is marked as OUT OF SERVICE; and detecting the connectivity of services and components (P13, P1E, P69), for instance, if health checking is not enabled then a system cannot detect when microservices are down.

Finding 3: *Wrong version of libraries is one of the top issues on service discovery. Consequently, compilers do not find the requested classes in the library.*

Implication: *Developers may benefit from the usage of project management tools like Apache Maven, or build automation tools like Gradle [28]. Both kind of tools offer support for library and dependency management.*

Finding 4: *Annotations, protocols and clusters appear among the top issues on service discovery. According to our survey, microservice-based applications benefit from (i) the usage of NonNull annotation to warn about the presence of null values; (ii) the adoption of JSON protocol for external communications; and (iii) the activation of clusters with dynamic IP.*
Implication: *Developers could reduce these issues by using static code analysis to recognise the misuse of annotations, protocols and clusters.*

3) **Service Authentication & Authorization:** Table I illustrates the task distribution of service authentication & authorization. The main tasks in service authentication & authorization are *authentication* 18/37, *authorization* 14/37, and *grants* 5/37. *Authentication* refers to issues that occur when trying to validate an user account (P48). *Authorization* refers to issues that occur when trying to provide access permissions to specific functionality (PE8). *Grants* refer to issues that occur when adopting the user roles to avoid repeating authentication & authorization processes (PB).

We observed that 44% of the posts in authentication & authorization are related to missed operations and usage of web tokens. On one hand, missed operations refer to the absence

of sentences for the usage of user credentials, configuration, and others. On the other hand, web tokens refer to the time life of tokens and excessive validation.

Regarding missed operations, we found concerns related to the usage of user credentials, configuration to avoid failures such as unique constraints, and missing annotations. For instance, requests return 401 unauthorized response when credentials are unavailable into Eureka or Spring Cloud [28] (P46, P52, PBE, PFE); transactions could include the username field as part of unique constraints (P3F, P48, PAC); and clients should use *RestTemplate* with the *LoadBalanced* annotation to support multiple instances of microservices (P62).

In relation to exposing web tokens, there are two concerns: (1) a slow response of the overall system due to multiple validations of a token (P9, P1D, P2A, PF7), for instance, developers could pass valid JWT token to other microservices; and (2) tokens that live longer than expected (P5F, P8B), for instance, developers could pass short live tokens between microservices that stay in the same domain.

Finding 5: *The absence of authentication & authorization operations as JNDI lookup with connection factory causes issues when getting user credentials in a centralized solution.*
Implication: *Developers could choose a fully decentralized solution that includes Zookeeper, ZeroMQ, or Consul [28] to reduce the response time of overall system.*

Finding 6: *Exposing the web tokens for a long time makes a microservice-based application vulnerable to attacks and hacks.*
Implication: *Developers could use short timelife tokens to minimize risks.*

4) **Service Invocation:** Table I depicts the task distribution of service invocation. The main tasks in service invocation are *asynchronous* 78/118, followed by *synchronous* 26/118 and *token validation* 14/118. *Asynchronous* refers to issues that occur when simultaneously calling microservices like using eventual consistency pattern to handle inconsistent states over multiple responses (P19). *Synchronous* refers to issues that occur when sequentially calling microservices like using the right *HttpHeaders* to send JSON data for *HttpEntity* instances (P1C). *Token validation* refers to issues that occur when a token has been received by microservices and needs an internal validation; for instance, a token is validated when passed by the header between microservices (P4E).

Regarding the asynchronous task, 29% of the posts are related to common cache errors and missing patterns for distributed systems. The most common cache errors include missing Terracotta servers for backing up the cache [28] (P95), missing shared cache layer on top of a database to respond with latest data even when dependant microservices are inactive (P50, P11A, P5D), wrong cache parameters for the avoidance of data duplication, reduction of unnecessary communication, and validation of outdated data (P3, P1B, P7A, PED, PF3). In respect of the missing patterns for distributed systems, we observed that sagas, event-store, eventual

consistency, aggregator, reporting, shared database, and produce/consumer are the most mentioned patterns. For instance, a sagas pattern is used to get fresh data for completion of complex processes by keeping a trace of events (PBF, PDA, PC4, PE2). The event-store is used to replay events when the system provides outdated user information (PBF, PDA, PC4, PE2). The other design patterns are combined for easy and fast access to connected data (P19, P4D, P8E, P96, P135).

Regarding the synchronous task, 31% of the posts are concerned about (i) missed HTTP headers for microservices in different domains; (ii) change of bounded context that results in broken contracts; and (iii) data synchronization after data structure modification. For instance, Cross-origin resource sharing (CORS) filter is required in Zuul, Azure Service Fabric and request headers with Access-Control-Allow-Origin is also required on the client side (P34, P74, P75), bounded contexts of microservices change according the data structure dependency (P6F, PA3, PF9), and microservices without versioning produce error after data structure modification (P107, P111).

Finding 7: *The absence of configuration parameters for cache are one of the most common issues on service invocation. Our study indicates that communication processes among microservices require those parameters for an optimal performance.*
Implication: *Developers could use configuration parameters of cache (e.g. Ehcache enable with Terracotta server) in external files such as YML configuration files.*

Finding 8: *Patterns for long running transactions are another common issue on service invocation. Results suggest that transaction patterns reduce code duplication on services.*
Implication: *Developers need to focus on transaction patterns such as SAGAS, try/cancel/confirm, and event-store to maintain data consistency across microservices.*

C. Further Categorisation

1) **Quality Attributes:** Quality attributes are the system characteristics assessed by a quality management team to judge the quality of software [16]. Table IIa shows the distribution of posts in terms of quality attributes. The main attributes that we identify are *reliability* 105/406 and *security* 90/406, followed by *performance* 65/406 and *availability* 64/406. Posts related to different quality attributes are presented in a category called *Others* 83/406. *Security* refers to issues that are related to security quality attribute; for instance, microservices inside a private network with IP addresses that expose them to public networks (P32, P98, PAF). *Reliability* refers to issues that are related to reliability quality attribute like missing handlers, retries, publish events or incomplete rollback that affect the operation time of a system (P3C, P8, PF, P10). *Performance* refers to issues that are related to performance quality attribute; for instance, response time is affected when no cache is implemented (P4, P105, P7A, P3, P1B). *Availability* refers to issues that stop the execution of a system; for instance, Zuul filter launches error when missing the http header X-Forwarded-Host (P87, PA3, PC5, PE3).

2) **Software Construction:** Construction of software refers to the elaborated creation of software by combining coding,

TABLE II
FURTHER CATEGORISATION

Quality attributes	#Posts (%)
Availability	64 (16%)
Performance	65 (16%)
Security	90 (22%)
Reliability	105 (26%)
Others	83 (20%)

(a) Quality Attributes

Software construction	#Posts (%)
Coding and debugging	128 (32%)
Detailed design	57 (14%)
Integration	152 (37%)
Unit testing	6 (01%)
Others	63 (16%)

(b) Software Construction

Fault class	#Posts (%)
Input/output faults	65 (16%)
Logic faults	171 (42%)
Interface faults	92 (23%)
Data faults	48 (12%)
Others	30 (07%)

(c) Fault Class

verification, debugging, unit testing, integration testing, and debugging [17]. Table IIb presents a distribution of posts in terms of software construction activities. The main tasks in software construction are *integration* 152/406 and *code & debugging* 128/406, followed by *detailed design* 57/406, *unit testing* 6/406 and *others* 63/406. *Integration* refers to issues that occur during the integration of multiple components such as missing operations to keep a clean state of data(P2C), single-sign-on with discovery (P1A, P23) or aggregator with integration platform (in P4C, P4D). *Code & debugging* refers to issues related to missing code for cache (P3), handling events (in P19, P3C), overriding methods (in P6C, P7B, P86), and annotations to validate and enable transactions (in PC9, PD0). *Detailed design* refers to issues concerned about (i) approaches to build reports based on information from multiple microservices (P43); (ii) event store with the usage of cache to keep updating the user information (P79, P7A); and (iii) usage of data-streaming with eventual consistency to decouple the database and transfer data (P7C).

3) **Fault classes:** Fault classes are a classification of bugs that can be detected by static analysis with the intention of detect anomalies introduced in a piece of code [18]. Table IIc shows a distribution of posts in terms of fault classes. The main fault classes that we identified are *logic faults* 171/406 and *interface faults* 92/406, followed by *input/output faults* 65/406, *data faults* 48/406, and *others* 30/406. *Logic faults* refers to issues when code statements are right written but they produce a wrong behaviour; for instance, developers have logic concerns on reading/writing a model, changing versions of microservices, making notifications, implementing events, tolerant clusters, JSON protocols and others (P41, P6B, P73, P93, PA6, PAF, PB1, PB2). *Interface faults* refer to issues that occur during the interconnection of components like adding identifiers, implementing failure recovery, aggregated patterns, and others (P6, P8, P4C, P66, P8E, P91). *Input/output faults* refers to issues given wrong values of process inputs/outputs like missing or using wrong version of libraries, location of parameters, and others (P31, PA2, P123, P33, P2E, PD1). *Data faults* refers to issues when accessing/formatting/storing data like optimising cache, managing persistent messages, using decorators and others (P3, P50, P7F, PD5, P109).

V. THREATS TO VALIDITY

Theoretical validity. To reduce potential biases, we iteratively refined our posts classification, we aligned the extraction

process with our research question, and we also used descriptive statistics to show the syntheses of collected data.

Internal validity. To mitigate the correctness of our posts classification, we rigorously defined a research protocol by using guidelines from Di Francesco et al. [1]. Another threat is a personal bias during the manual classification, but we mitigate it by involving more than one author during the process of analysis and synthesis of posts.

External validity. We mitigate the generalizability of obtained results by (i) applying the best practices coming from two different guidelines on systematic studies [31], [32], and (ii) using a big datasource that provides a variety of posts [14]. Another threat is the number of databases to collect the posts, but we mitigate it by selecting one of the largest database used by developers [13], [20].

VI. RELATED WORK

While microservice systems have been widely studied under various domains [36]–[39], there exists only one previous work that refers to Stack Overflow for extraction of topics in microservice development. Bandeira et al. [20] implemented a topic model, which is an unsupervised machine learning that detects word or phrase patterns in Stack Overflow posts to build a general taxonomy of subjects on microservices. The topic model uses a textual content to build the frequency of each word. However, it excludes relevant information from comments and possible key answers from the authors of posts. Stack Overflow comments provide clarification and guide the authors to improve the posts [21]. Our research differs from their work in two dimensions. First, the classification framework is based on recognized aspects of software engineering such as software quality attributes, software construction activities, and fault classes. Second, we partially follow systematic mapping guidelines with the corresponding replication package to increase the reliability and replicability of our study results.

Regarding the origin of the data in previous empirical studies on microservices, they either: (i) introduced algorithms using a reduced number of faults or issues to evaluate their debugging approaches [9], [11], [22]–[24]; or (ii) interviewed a limited number of developers to identify undesired practices [25] or code smells [2] in microservice-based applications. In this context, to the best of our knowledge, our work constitutes the first attempt to reveal active development concerns

from Stack Overflow as a big data source, relevant for the development community, which may serve as a baseline for future empirical works.

VII. CONCLUSIONS

We surveyed Stack Overflow to provide a classification of posts based on microservice life cycle activities at runtime, namely service routing, service discovery, service authentication & authorization, and service invocation. The results of our study indicate that (i) missing parameters and operations are the most common concerns in service routing; (ii) wrong versions of libraries, annotations, protocols, and clusters appear as main concerns on service discovery; (iii) the absence of authorisation operation and web tokens exposed for a long time are the key concerns related to service authentication and authorisation; and (iv) the absence of cache parameters and inadequate patterns for long running transactions emerge as trending concerns in service invocation. Additionally, our study reveals that the development concerns are mostly focused on security and reliability. Integration testing and logic faults also represent a significant number of posts.

Based on our findings, we suggested potential research directions and identified some implications of our work such as (i) the usage of short timelife token to reduce applications vulnerabilities; (ii) the usage of static code analysis to recognise misuses of annotations, protocols and clusters; (iii) the storing of configuration parameters in external files to improve the cache performance; and (iv) the use of transaction patterns to maintain data consistency across microservices.

ACKNOWLEDGEMENTS

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Shenzhen Peacock Plan (Grant No. KQTD2016112514355531), and Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. JCYJ20170817110848086).

REFERENCES

- [1] P. Di Francesco and et al., "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," in *ICSA*, 2017.
- [2] P. Jamshidi and et al., "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.
- [3] M. Mazzara and B. Meyer, Eds., *Present and Ulterior Software Engineering*. Springer, 2017.
- [4] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of microservices-enabled fog applications," *Concurr. Comput. Pract. Exp.*, vol. 31, no. 22, 2019. [Online]. Available: <https://doi.org/10.1002/cpe.4436>
- [5] T. Zheng, Y. Zhang, X. Zheng, M. Fu, and X. Liu, "Bigvm: A multi-layer-microservice-based platform for deploying saas," in *Fifth International Conference on Advanced Cloud and Big Data, CBD 2017, Shanghai, China, August 13-16, 2017*. IEEE Computer Society, 2017, pp. 45–50. [Online]. Available: <https://doi.org/10.1109/CBD.2017.16>
- [6] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "Smartvm: a sla-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019. [Online]. Available: <https://doi.org/10.1007/s11280-018-0562-5>
- [7] A. Saba, "4 challenges you need to address with microservices adoption," 2016. [Online]. Available: <https://bit.ly/35Mfr4O>
- [8] A. Whitepaper, "Challenges of microservices." [Online]. Available: <https://amzn.to/3a0XRgl>
- [9] V. Heorhiadi and et al., "Gremlin: Systematic Resilience Testing of Microservices," in *ICDCS*. Nara, Japan: IEEE, 2016.
- [10] Y. Gan and et al., "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices," in *ASPLOS*, 2019.
- [11] X. Zhou and et al., "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE TRANSACTION ON SOFTWARE ENGINEERING*, p. 1, 2018.
- [12] C. Esposito and et al., "Challenges in Delivering Software in the Cloud as Microservices," *IEEE Cloud Computing*, pp. 10–14, 2016.
- [13] S. M. Nasehi and et al., "What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow," in *ICSM*. IEEE, 2012.
- [14] C. Rosen and et al., "What are mobile developers asking about? A large scale study using stack overflow," *Empirical Software Engineering*, 2016.
- [15] Jeff Atwood and Joel Spolsky, "StackOverflow," 2008. [Online]. Available: <https://stackoverflow.com/>
- [16] I. Sommerville, "Software engineering (tenth edn. global edition)," 2016.
- [17] P. Bourque, R. E. Fairley et al., *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3*. IEEE Computer Society, 2014.
- [18] P. Jorgensen, "Software testing, a craftsman approach," 2014.
- [19] N. Meng and et al., "Secure coding practices in java: Challenges and vulnerabilities," in *40th ICSE*. Washington, DC, USA: IEEE, 2018.
- [20] A. Bandeira and et al., "We Need to Talk about Microservices: an Analysis from the Discussions on StackOverflow," in *MSR*, 2019.
- [21] Jeff Atwood and Joel Spolsky, "StackOverflow Comment Everywhere," 2008. [Online]. Available: <https://stackoverflow.com/help/privileges/comment>
- [22] A. Panda, M. Sagiv, and S. Shenker, "Verification in the Age of Microservices," in *HotOS*, vol. 7. Whistler, BC, Canada: ACM, 2017.
- [23] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta Debugging Microservice Systems," in *ASE 2018*, vol. 18. ACM, 2018.
- [24] X. Zhou, X. Peng, T. Xie, and et al., "Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs," in *ESEC/FSE*, 2019.
- [25] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, pp. 56–62, 2018.
- [26] H. Upreti, "4 Essential Strategies for Testing Microservices," 2018.
- [27] N. Alshuqayran and et al., "A Systematic Mapping Study in Microservice Architecture," in *2016 IEEE SOCA*. Macau, China: IEEE, 2016.
- [28] [n.d.], "Common components used in Microservices," 2020. [Online]. Available: <https://bit.ly/3o2uHEJ>
- [29] RFC6749, "OAuth," 2006. [Online]. Available: <https://oauth.net/2/>
- [30] A. Gupta, "Microservice Design Patterns," 2015. [Online]. Available: <https://goo.gl/pd5d1x>
- [31] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," pp. 2049–2075, 2013.
- [32] K. Petersen and et al., "Guidelines for conducting systematic mapping studies in software engineering: An update," 2015.
- [33] [n.d.], "Query Stack Overflow," 2020. [Online]. Available: <https://data.stackexchange.com/stackoverflow/queries>
- [34] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.
- [35] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in *CLOSER 2016*. Rome, Italy: ACM, 2016.
- [36] J. Hua, Y. Zhang, Y. Zhang, and S. Khurshid, "Edsketch: execution-driven sketching for java," *STTT*, vol. 21, no. 3, pp. 249–265, 2019. [Online]. Available: <https://doi.org/10.1007/s10009-019-00512-8>
- [37] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
- [38] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, 2019, pp. 169–180. [Online]. Available: <https://doi.org/10.1145/3293882.3300574>
- [39] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 132–142. [Online]. Available: <https://doi.org/10.1145/3238147.3238187>