# When Failure is (Not) an Option: Reliability Models for Microservices Architectures

Lalita J. Jagadeesan
*Nokia Bell Labs*
Naperville, IL 60563, USA
lalita.jagadeesan@nokia-bell-labs.com

Veena B. Mendiratta
*Nokia Bell Labs*
Naperville, IL 60563, USA
veena.mendiratta@nokia-bell-labs.com

***Abstract*—Modern application development and deployment is rapidly evolving to microservices based architectures, in which thousands of microservices communicate with one another and can be independently scaled and updated. While these architectures enable flexibility of deployment and frequency of upgrades, the naive use of thousands of communicating and frequently updated microservices can significantly impact the reliability of applications. To address these challenges, service meshes are used to rapidly detect and respond to microservices failures without necessitating changes to the microservices themselves. However, there are inherent tradeoffs that service meshes must make with regards to how quickly they assume a microservice has failed and the subsequent impact on overall application reliability. We present in this paper a modeling framework for microservices and service mesh reliability that takes these tradeoffs into account.**

***Index Terms*—microservices, service mesh, sidecars, circuit breakers, reliability, availability, resilience, reliability models, probabilistic model checking, PRISM.**

## I. Introduction

Microservices architectures are becoming widely adopted for the creation and deployment of applications and services. Typical applications based on these architectures are comprised of thousands of loosely coupled microservices that communicate through inter-process communication protocols such as http, tcp, gRPC, depending on the nature of the application. In contrast to legacy monolithic applications, functionality is divided among these microservices, enabling independent and rapid updates and scaling of different parts of an application, in response to real-time changes in application demand and changing expectations of functionality.

### *Service Meshes and Sidecars*

This loose coupling among rapidly changing microservice deployments, while enabling much speed and agility, faces the inherent and difficult challenges of distributed systems: microservices and the communication among them may silently fail, potentially significantly affecting the availability of the overall application. As such, to protect against such failures, each microservice must be protected from the failure of other microservices with which it communicates, potentially requiring changes to the microservices themselves. In response to this need, service meshes (e.g., Istio [1], Linkerd [2]) have been developed to provide a microservice-agnostic framework for ensuring resiliency. In particular, service meshes obviate the need for individual microservices to be aware of the failure of other microservices by providing an intelligent wrapper around microservices. This is implemented through the use of "sidecars" (e.g., Envoy [3], NGINX [4]) that monitor individual microservices. The sidecars communicate with the service mesh control plane, which makes decisions about the communication and routing amongst microservices. The service mesh and sidecar architecture is depicted in Figure 1.

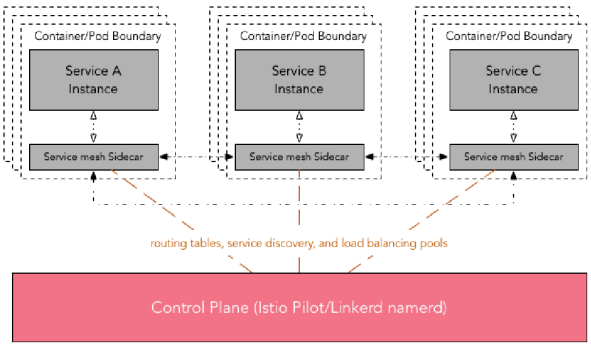


Fig. 1. Service mesh architecture [5]

The service mesh control plane and sidecars are programmable via declarative APIs and can be deployed and updated at run-time of the microservices; capabilities such as circuit breakers [6] are used to implement resilient applications. In particular, attributes in the individual sidecars are configured with timeout values and retry values to rapidly detect when the associated microservice has not received a response to a request within a specified timeframe, and to retry the request a specified number of times with potentially increasing timeout values. Attributes in the service mesh control plane are configured and pushed down to the sidecars to indicate how the communication among microservices should dynamically change when a given microservice is deemed to be unresponsive by its sidecar; for example, bypassing a non-critical microservice if it is likely it has failed. Such circuit breakers prevent cascading failures, where the failure of a single microservice causes the overall application to fail. They further provide a framework that permits certain non-critical microservices, such as logging, to be bypassed if they are unresponsive, allowing the overall application to work in a degraded mode, while shutting down the entire application if

978-1-7281-7735-9/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSREW51248.2020.00031

a microservice such as a critical database is deemed to have failed.

*Availability: Programmability, Models, and Impacts*

However, there are inherent tradeoffs in the specification of these circuit breakers with regards to how quickly they assume a microservice has failed and the subsequent impact on overall application reliability. If the determination that a microservice has failed is made too quickly, then service requests may be unnecessarily aborted, lowering overall application availability. On the other hand, if the determination that a microservice has failed is made too slowly, the failure of a single microservice can recursively cascade to its client microservices and cause the overall application to fail, again lowering application availability. For many types of applications, availability is a critical requirement, and hence such tradeoffs must be carefully considered. Further, as these applications typically consist of thousands of communicating microservices, these tradeoff must be understood at scale.

To this end, the main contribution of this paper is a modeling framework for the reliability of microservice-based applications, in which applications may operate in a degraded mode if non-critical microservices have been deemed to have failed. Our modeling framework consists of two parts: the first is applicable at the micro-level of detailed operational behavior of small sets of microservices communicating with one another via a service mesh; this is useful for iterative system design. The second part of our reliability framework is suitable for typical application deployments comprised of thousands of service meshed microservices. For both of these models, we present associated reliability analyses.

The remainder of the paper is organized as follows. Section II describes our micro-level reliability model for small sets of microservices, together with associated reliability analyses. Section III describes our reliability model for microservice-based applications at scale, together with associated reliability analyses. We discuss the implications of our models and analyses in Section IV. Related work is described in Section V.

## II. A MICRO-MODEL OF MICROSERVICES

We first describe our micro-level model of service meshed microservices. In our model, microservices, as well as service mesh sidecars, are represented as a set of concurrent continuous-time markov models (CTMCs) which communicate through multi-way synchronous message passing. We use the open-source PRISM probabilistic model checker [7] for our specification and analysis.

To illustrate our micro-level model, we use a variation of the *bookinfo* application [8] example from the Istio service mesh, representing a book information web application. In our modification of this service, we assume three microservices, as depicted in Figure 2: a main product page, a details module that provides information about the requested book, and a reviews module that provides information on book reviews. (We omit the ratings module from [8] for simplicity.) The rectangles represent sidecars in a service mesh.

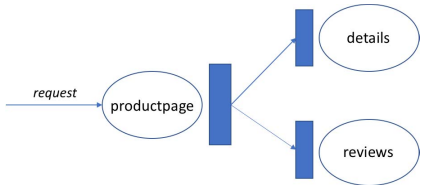


Fig. 2. Variation of bookinfo application

In this application, when a request arrives, it is first sent to the `productpage` microservice, which in turn concurrently calls the `details` and `reviews` microservices. When both of these microservices have completed, the `productpage` returns the requested information back to the requester. Communication among the microservices is governed by sidecars, which also implement circuit breakers. In addition to modeling this functionality, we also model microservices failures, detection, and recovery.

Our PRISM model of this modification and extension of the *bookinfo* application is given in Figure 3. The left hand side depicts the specification of the `productpage` microservice. The `processing` variable represents whether a request is currently being processed. The `done_d` and `done_rv` variables represent respectively whether the calls to the `details` and `reviews` modules have been completed for the request currently being processed. Further, this microservice can be in three possible states, represented as the `sp` variable: up, failed but not yet detected, and failed under repair.

An arriving request is *accepted* if and only if the `productpage` is up and is not busy processing another request, otherwise the arriving request is dropped. If it is accepted, the `details` and `reviews` microservices are concurrently called using the `rqd` and `rqv` messages. When both of these message synchronizations have completed, the `productpage` goes back to its initial state, ready to accept new requests. At any time, the `productpage` can fail, resetting all other state information to their initial values, and entering a state where detection and recovery take place; failure, detection, and recovery are modeled as exponentially distributed delays. The `details` and `reviews` modules respectively synchronize on the `rqv` and `rqv` messages, with exponentially distributed computation times, and can also independently fail.

We further introduce a sidecar for each microservice, which synchronizes on the messages of its associated microservice; hence there is multi-way synchronization among microservices and sidecars. We specify a circuit breaker `circuitb` for the `productpage` sidecar, which causes the `productpage` to drop the request that is currently being processed and reset to its initial state. Further, we assume that details are *critical*, but reviews are not, and that the request is considered to have been processed in a *degraded* manner if only the `details` computation has completed when a circuit breaker opens.

We then analyze availability through steady state rewards, as depicted in Figure 4. In particular, we analyze the probability

```
//main microservice
module productpage

    // currently processing a request
    processing: bool init false;

    // details have been completed for this request
    done_d: bool init false;

    // reviews have been completed for this request
    done_rv: bool init false;

    // 0 is up, 1 is failed not yet detected, 2 is failed under repair
    sp: [0..2] init 0;

    //if up and not currently processing a request, accept new request
    [request] (sp=0) & !(processing) -> (processing'=true);

    // drop incoming request because either process is busy or down
    [request] !(sp=0) | (processing) -> true ;

    // call details microservice if it and reviews not been called yet
    [rqd] (sp=0) & (processing) & (done_d=false) & (done_rv=false)
    -> (done_d'=true);

    // call details microservice if just it has not been called yet
    // computation on current request done, reset
    [rqd] (sp=0) & (processing) & (done_d=false) & (done_rv=true)
    -> (done_d'=false) & (done_rv'=false) & (processing'=false);

    // call reviews microservice if it and details has been called yet
    [rqv] (sp=0) & (processing) & (done_d=false) & (done_rv=false)
    -> (done_rv'=true);

    // call reviews microservice if just it has not been called yet
    // computation on current request then done, reset
    [rqv] (sp=0) & (processing) & (done_d=true) & (done_rv=false)
    ->  (done_d'=false) & (done_rv'=false) & (processing'=false);


    // process fails, wipe out all other state of this microservice
    [pfail] (sp=0) -> lambdapfail: (sp'=1) & (done_d'=false)
                    & (done_rv'=false) & (processing'=false);

    // failure detected
    [pdetected] (sp=1) -> mu_pdetect: (sp'=2);

    // failure repaired
    [prepaired] (sp=2) -> mu_prepair: (sp'=0);

    // circuit breaker triggered, reset all state of this microservice
    [circuitb] true -> (done_d'=false) & (done_rv'=false)
                     & (processing'=false);

endmodule
```

```
//details microservice
module details

    // up, detection, repair
    sd: [0..2] init 0;

    // compute and return details
    [rqd] (sd=0) -> tau_details: true;

    // failure, detection, recovery
    [dfail] (sd=0) -> lambdadfail: (sd'=1);
    [ddetected] (sd=1) -> mu_ddetect: (sd'=2);
    [drepaired] (sd=2) -> mu_drepair: (sd'=0);
endmodule

//reviews microservice
module reviews

    // up, detection, repair
    srv: [0..2] init 0;

    // compute and return reviews
    [rqv] (srv=0) -> tau_reviews: true;

    // failure, detection,recovery
    [rvfail] (srv=0) -> lambdarvfail: (srv'=1);
    [rvdetected] (srv=1) -> mu_rvdetect: (srv'=2);
    [rvrepaired] (srv=2) -> mu_rvrepair: (srv'=0);

endmodule

//sidecars
module productpage_sidecar

    //circuit breaker triggered on delay
    [circuitb] (withcircuitb) & (processing)
    -> mu_cb: true;

    [rqd] true -> true;
    [rqv] true -> true;

endmodule

module details_sidecar
    [rqd] true -> true:
endmodule

module
    reviews_sidecar = details_sidecar [rqd=rqv]
endmodule

// request arrival generator
module requests
        [request] true -> gamma_req: true;
endmodule
```

Fig. 3. PRISM specification of bookinfo service

```
//rewards
rewards "accrequest_rate"
        // incoming request accepted
        [request] (sp=0) & !(processing) : 1/(gamma_req);
endrewards


rewards "complrequest_rate"
        // computation on current request finished
        [rqd] (sp=0) & (processing) & !(done_d) & (done_rv): 1/(gamma_req);
        [rqv] (sp=0) & (processing) & !(done_rv) & (done_d): 1/(gamma_req);
endrewards


rewards "complordegradedrequest_rate"
        // computation on current request finished
        [rqd] (sp=0) & (processing) & !(done_d) & (done_rv): 1/(gamma_req);
        [rqv] (sp=0) & (processing) & !(done_rv) & (done_d): 1/(gamma_req);

        // if degraded states are allowed
        // then current request is considered completed or degraded
        // if details have been computed
        [circuitb] (allowdegrade=true) & (done_d) :1/(gamma_req);
endrewards
```

Fig. 4. Rewards specification

of an arriving request (a) being accepted (b) being accepted and fully processed or (c) being accepted and either fully processed or processed in a degraded manner.

We use the following parameter values for our analysis:

- $\lambda_{*fail}$ microservice failure rate (100/year)
- $\mu_{*detect}$ microservice failure detection rate (60 per hour)
- $\mu_{*repair}$ microservice failure recovery rate (60 per hour)
- $\tau_*$ computation rate of `details` and `reviews` (3600 per hour)
- $\mu_{cb}$ circuit breaker rate (varied as 0 and 240 per hour)
- $\gamma_{req}$ request arrival rate (varied as 12 and 108 per hour)

Our results are given in Tables I, II, and III.

Table I shows the steady-state probability that an incoming request will be accepted for processing. This measure is closely related to prevention of cascading failures as every request that is not accepted (i.e., dropped) affects the upstream process (or user): that is, the calling process (or user) will either wait for a reply, or timeout and retry, thus causing its own availability (or satisfaction) to deteriorate. Thus, recur-

TABLE I
STEADY-STATE PROBABILITY OF ACCEPTED REQUESTS

| Circuit breaker rate $\mu_{cb}$ | Arrival rate $\gamma_{req}$ of incoming requests 12 | 36 | 60 | 84 | 108 |
|---|---|---|---|---|---|
| 0 | 0.99447 | 0.98448 | 0.97477 | 0.96530 | 0.95602 |
| 60 | 0.99466 | 0.98492 | 0.97541 | 0.96610 | 0.95698 |
| 120 | 0.99478 | 0.98526 | 0.97593 | 0.96679 | 0.95783 |
| 180 | 0.99488 | 0.98555 | 0.97640 | 0.96743 | 0.95863 |
| 240 | 0.99498 | 0.98582 | 0.97685 | 0.96803 | 0.95938 |

TABLE II
STEADY-STATE PROBABILITY OF COMPLETED REQUESTS

| Circuit breaker rate $\mu_{cb}$ | Arrival rate $\gamma_{req}$ of incoming requests 12 | 36 | 60 | 84 | 108 |
|---|---|---|---|---|---|
| 0 | 0.99446 | 0.98447 | 0.97477 | 0.96529 | 0.95601 |
| 60 | 0.96985 | 0.96043 | 0.95120 | 0.94215 | 0.93329 |
| 120 | 0.94638 | 0.93738 | 0.92855 | 0.91990 | 0.91140 |
| 180 | 0.92383 | 0.91521 | 0.90676 | 0.89846 | 0.89031 |
| 240 | 0.90211 | 0.89386 | 0.88575 | 0.87779 | 0.86997 |

sively, dropped requests lead to cascading failures (or user dissatisfaction). Table II and Table III measure the probability of completion of requests, without and with degraded processing, respectively; both key measures of service availability.

As shown in Table I, for a given circuit breaker rate, the probability of accepting a request decreases as the request arrival rate increases, as the `productpage` is more likely to be busy when an incoming request arrives. For a given request arrival rate, increasing the circuit breaker rate increases the probability of acceptance, as the `productpage` is less likely to be waiting too long for the `details` or `reviews` microservices to recover from a failure. However, as shown in Table II, the probability that an accepted request will be successfully processed decreases significantly as the circuit breaker rate increases, as the processing of a request may be interrupted unnecessarily. Table III shows the amelioration that happens when degraded processing is taken into account.

These experiments demonstrate the inherent tension in the use of circuit breakers for ensuring availability: on the one hand, they decrease the probability that messages are dropped due to cascading failures of microservices, on the other hand, they decrease the probability of successful request processing.

## III. A MACRO-MODEL OF MICROSERVICES

The previous section described a reliability model for an application with a small set of microservices, suitable for iterative system design. However, due to state space explosion, this model cannot be used at scale for typical application deployments comprised of thousands of microservices, as described in Section I. In this section, we describe our Continuous–Time Markov Chain (CTMC) model that can be used to analyze the availability of such large deployments.

In order to work at scale, such deployments, in addition to using sidecars and service mesh, together with circuit breakers and timeouts, also use load-balanced replicated microservice instances as part of their resiliency architecture. Namely, a

TABLE III
STEADY-STATE PROBABILITY OF COMPLETED OR DEGRADED REQUESTS

| Circuit breaker rate $\mu_{cb}$ | Arrival rate $\gamma_{req}$ of incoming requests 12 | 36 | 60 | 84 | 108 |
|---|---|---|---|---|---|
| 0 | 0.99446 | 0.98447 | 0.97477 | 0.96529 | 0.95601 |
| 60 | 0.97814 | 0.96860 | 0.95927 | 0.95013 | 0.94118 |
| 120 | 0.96242 | 0.95323 | 0.94424 | 0.93541 | 0.92676 |
| 180 | 0.94721 | 0.93835 | 0.92966 | 0.92114 | 0.91277 |
| 240 | 0.93248 | 0.92393 | 0.91553 | 0.90729 | 0.89919 |

request to an application is first sent to a load balancer, which then routes the request to a microservice instance that is currently not under high load, as determined by its sidecar. If a specified percentage of these replicated microservice instances are deemed unresponsive, the entire load-balanced group of microservices is deemed to have failed. This "cluster panic" attribute is typically set at 50%, but is specifiable to any value.

The objective of the model is to assess the availability of large scale microservices deployments for a range of parameter values. Microservices are either critical (CM) or non-critical NCM) and are represented separately in the model. A composite service is comprised of the critical and non-critical services with the following properties:

+ A composite service can be in the following states: up, down, up but with degraded functionality
+ A composite service is up iff all microservices are up
+ A composite service is down iff one or more critical microservices has failed or is in the recovery state
+ A composite service is up with degraded functionality iff all critical microservices are up and one or more non-critical microservices has failed or is in the recovery state

For modeling abstraction we assume that a certain proportion of services are critical and the remaining ones are non-critical; the proportion is varied in the model to assess the impact on composite service availability. The model is built to represent single failure events. Figure 5 shows the macro reliability model of the microservices architecture where we model the failure events and the subsequent detection and recovery actions.

The model states are defined as follows:

| | |
|---|---|
| **working** | normal working state, all services up |
| **Cdet** | critical services detection, failed state |
| **Crec** | critical services recovery, failed state |
| **NCdet** | non-critical services detection, degraded state |
| **NCrec** | non-critical services recovery, degraded state |
| **failed** | service unavailable, failed state |

The model takes as inputs the failure rates of microservices in failures per hour. The detection/recovery/repair rates for the microservices are also specified in terms of events per hour. The probability of successful detection and recovery are included in the model. These reflect the coverage factors, which is the probability of successful (detection/recovery) given that an error has occurred.

The model runs as follows. The state (*working*) represents the state where all the microservices and the composite

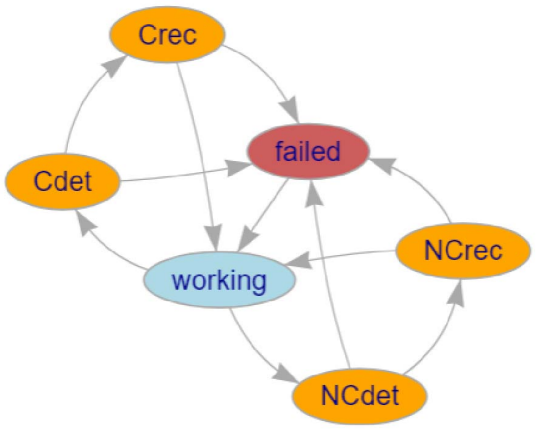


Fig. 5. Microservices reliability model

service are up. In the event of a microservice failure, the system transitions to the detection state — *Cdet* for a CM failure and *NCdet* for a NCM failure with transition rates $\lambda_C$ and $\lambda_{NC}$ respectively. Successful detection leads to a transition to the recovery states *Crec* and *NCrec* respectively with corresponding transition rates $\mu_{Cd}\rho$ and $\mu_{NCd}\delta$. If the detection is unsuccessful the system transitions to the *failed* state. Successful recovery leads to a transition to the normal working state *working* with transition rates $\mu_{Cr}\rho$ and $\mu_{NCr}\delta$ respectively. If the recovery is unsuccessful the system transitions to the *failed* state. The system transitions out of the *failed* state to the *working* at a rate $\mu_R$ which represents the off-line long recovery/repair rate. Depending on the implementation, offline repair of microservices can involve restarting the software, replacing the hardware unit, instantiating a new virtual machine, etc.

The parameters and the values used for the model inputs are listed below.

- $p_C$ proportion of microservices that are critical (varied between 0.1 and 0.5)
- $\lambda$ composite service failure rate (100/year)
- $\lambda_C$ critical microservices failure rate ($p_C\lambda$)
- $\lambda_{NC}$ critical microservices failure rate ($(1-p_C)\lambda$)
- $\mu_{Cd}$ critical microservices failure detection rate (varied between 60 and 240 per hour)
- $\mu_{NCd}$ non-critical microservices failure detection rate (varied between 60 and 240 per hour)
- $\mu_{Cr}$ critical microservices failure recovery rate (varied between 60 and 240 per hour)
- $\mu_{NCr}$ non-critical microservices failure recovery rate (varied between 60 and 240 per hour)
- $\mu_R$ long recovery/repair rate – after failed detection or recovery (3/hour)
- $\delta$ probability of successful microservices failure detection (varied between 0.95 and 0.99)
- $\rho$ probability of successful microservices failure recovery (varied between 0.95 and 0.99)

The CTMC model is solved using the markovchain package in R [9]. The model results for a range of parameter values are presented in Tables IV, V, and VI for Expected Probability in *working* State, Expected Degraded Time (minutes/year), and Expected Downtime (minutes/year) respectively. Degraded time represents the time spent in the detection and recovery states, *NCdet* and *NCrec*. Downtime represents the time spent in the states *Cdet*, *Crec* and *failed*. In Tables IV, V, and VI, $\mu_{det}$ refers to the detection rates $\mu_{Cd}$ and $\mu_{NCd}$ and $\mu_{rec}$ refers to the recovery rates $\mu_{Cr}$ and $\mu_{NCr}$.

TABLE IV
EXPECTED PROBABILITY IN WORKING STATE

| $\mu_{det}$ | Critical Services Proportion | | | | | |
|---|---|---|---|---|---|---|
| and | $\delta$=0.95, $\rho$=0.95 | | | $\delta$=0.99, $\rho$=0.99 | | |
| $\mu_{rec}$ | 0.1 | 0.3 | 0.5 | 0.1 | 0.3 | 0.5 |
| 240 | 0.99954 | 0.99954 | 0.99954 | 0.99983 | 0.99983 | 0.99983 |
| 120 | 0.99944 | 0.99944 | 0.99944 | 0.99974 | 0.99974 | 0.99974 |
| 80 | 0.99935 | 0.99935 | 0.99935 | 0.99964 | 0.99964 | 0.99964 |
| 60 | 0.99926 | 0.99926 | 0.99926 | 0.99955 | 0.99955 | 0.99955 |

TABLE V
EXPECTED DEGRADED TIME, MINUTES/YEAR

| $\mu_{det}$ | Critical Services Proportion | | | | | |
|---|---|---|---|---|---|---|
| and | $\delta$=0.95, $\rho$=0.95 | | | $\delta$=0.99, $\rho$=0.99 | | |
| $\mu_{rec}$ | 0.1 | 0.3 | 0.5 | 0.1 | 0.3 | 0.5 |
| 240 | 44 | 34 | 24 | 45 | 35 | 25 |
| 120 | 88 | 68 | 49 | 90 | 70 | 50 |
| 80 | 132 | 102 | 73 | 134 | 104 | 75 |
| 60 | 175 | 136 | 97 | 179 | 139 | 99 |

TABLE VI
EXPECTED DOWNTIME, MINUTES/YEAR

| $\mu_{det}$ | Critical Services Proportion | | | | | |
|---|---|---|---|---|---|---|
| and | $\delta$=0.95, $\rho$=0.95 | | | $\delta$=0.99, $\rho$=0.99 | | |
| $\mu_{rec}$ | 0.1 | 0.3 | 0.5 | 0.1 | 0.3 | 0.5 |
| 240 | 200 | 210 | 219 | 45 | 55 | 65 |
| 120 | 205 | 224 | 244 | 50 | 70 | 90 |
| 80 | 209 | 239 | 268 | 55 | 85 | 114 |
| 60 | 214 | 253 | 292 | 60 | 99 | 139 |

From Table IV we observe that the expected probability of being in the the *working* state decreases as the detection and recovery rate decreases, and does not change as the proportion of CM is varied from 0.1 to 0.5 (the variation along this dimension is seen in the downtime and degraded time).

We observe from Table V that the expected time in the degraded states increases as the detection and recovery rate decreases (i.e., it takes longer to detect and recover), and decreases as the proportion of CM is increased from 0.1 to 0.5 (the reason being that the detection and recovery states associated with the failure of a CM, *Cdet* and *Crec*, are down states and not degraded states).

Similarly, from Table VI we observe that the expected downtime increases as the detection and recovery rate decreases (i.e., it takes longer to detect and recover), and also

increases as the proportion of CM is increased from 0.1 to 0.5. In this case the probabilities of successful microservices failure detection and recovery, $\delta$ and $\rho$, have a significant impact on the expected downtime —dropping from a maximum value of 292 minutes/year to a maximum value of 139 minutes/year as the parameters $\delta$ and $\rho$ are increased from 0.95 to 0.99.

## IV. Discussion

We have described a modeling framework for the reliability of microservice-based applications, consisting of a micro-model for iterative application design and a macro-model for reliability analysis at scale. Our model takes into account the tradeoffs in service mesh frameworks for rapidly - but not too rapidly - detecting failures of microservices, preventing cascading failures and enabling degraded functionality. We have further presented associated reliability analyses.

In future work, we plan to extend our micro model to allow deterministic delays for circuit breakers, either using Erlang distributions or extending to generalized semi-markov processes. We also plan to extend our macro model to handle transient failures which may recover without intervention.

## V. Related Work

Microservices based applications are evolving rapidly and there is a growing body of research and development work on the subject addressing areas such as architecture issues including service mesh and sidecars, performance engineering, fault analysis and debugging, and run-time reliability estimation.

In [10], PRISM is used to analyze the behavior of the Retry and Circuit Breaker patterns captured as CTMCs to quantify the impact of choosing a particular resiliency pattern and its configuration on expected total time and expected contention time to perform a required number of successful invocations, in the context of a simple client-service interaction scenario. Our micro-model uses the same underlying formalism to model and analyze microservices and service mesh reliability mechanisms; however, we support multiple concurrent target microservices, and analyze steady-state availability measures including degraded microservice functionality.

The study by Zhang [11] presents principles to build a reliable microservice, and shows how Istio can improve reliability by providing observability and improving network resiliency, as well the limitations and performance impacts of Istio. An approach for the specification, aggregation, and evaluation of software quality attributes for the architecture of microservice-based systems is presented in [12] that allows developers to produce and evaluate architecture models of the system. The work by Zhou et al. [13] presents the results of an industrial survey and empirical study on the fault analysis and debugging of microservice systems.

The work in [14] presents an "in vivo" testing method to assess the reliability of an MSA application in operation and is based on an adaptive sampling strategy, leveraging monitoring data about microservices usage and failure/success of user demands. In [15], the authors present an interesting hybrid monitoring-testing approach for web services that includes a monitoring-based estimate built with a Bayesian model for reliability and performance assessment, a testing-based estimate based on an adaptive sampling strategy, and a strategy to continuously combine both estimates to trade-off accuracy and cost. Very recent work [16] presents a Microservice Resilience Measurement Model (MRMM) and framework, which can be used to elicit resiliency requirements and to quantify resilience metrics in a microservice-based system.

It has been acknowledged that a key problem in the engineering of microservices architecture applications is the estimate of their reliability, which can be difficult to perform prior to release due frequent upgrades, dynamic service interactions, and changes in how the applications are used [14]. To that end, our work presents a modeling framework for analyzing the reliability of microservices-based applications at both a micro and macro level during the early architecture/design phase, thereby complementing the works on assessing estimating the reliability during the testing and operational phases. We evaluate availability, unavailability as well as degraded state probability. Microservices are often deployed on the cloud, and studies on cloud system failures [17] have shown such systems are often likely to be in a "limped" or degraded state rather than be totally unavailable.

## References


[1] Istio. https://istio.io.
[2] Linkerd: Homepage. https://linkerd.io.
[3] Envoy Proxy - Home. https://www.envoyproxy.io.
[4] NGINX. https://www.nginx.com.
[5] A sidecar for your service mesh. https://www.abhishek-tiwari.com/a-sidecar-for-your-service-mesh/.
[6] Circuit Breakers and Microservices Architectures. https://techblog.constantcontact.com/software-development/circuit-breakers-and-microservices/.
[7] http://www.prismmodelchecker.org/manual/Main/Welcome.
[8] https://istio.io/latest/docs/examples/bookinfo/ https://istio.io/latest/docs/examples/bookinfo/.
[9] G. A. Spedicato, "Discrete time markov chains with r," *The R Journal*, 07 2017, r package version 0.6.9.7. [Online]. Available: https://journal.r-project.org/archive/2017/RJ-2017-036/index.html
[10] N. Mendonca, C. M. Aderaldo, J. Cámara, and D. Garlan, "Model-based analysis of microservice resiliency patterns," in *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2020, pp. 114–124.
[11] W. Zhang, "Improving microservice reliability with istio."
[12] M. Cardarelli, L. Iovino, P. Di Francesco, A. Di Salle, I. Malavolta, and P. Lago, "An extensible data-driven approach for evaluating the quality of microservice architectures," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1225–1234.
[13] 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.
[14] R. Pietrantuono, S. Russo, and A. Guerriero, "Testing microservice architectures for operational reliability," *Software Testing, Verification and Reliability*, vol. 30, no. 2, p. e1725, 2020.
[15] A. Guerriero, R. Mirandola, R. Pietrantuono, and S. Russo, "A hybrid framework for web services reliability and performance assessment," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 185–192.
[16] K. Yin, Q. Du, W. Wang, J. Qiu, and J. Xu, "On representing and eliciting resilience requirements of microservice architecture systems," *arXiv preprint arXiv:1909.13096*, 2019.
[17] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 1–16.