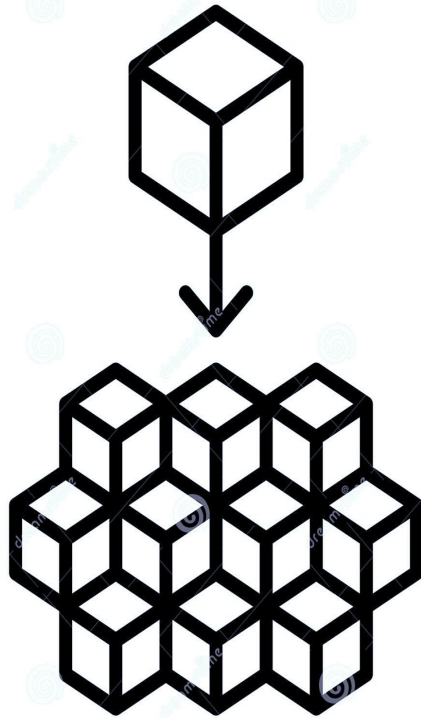


Microservices Reliability

January 23, 2024

Dr Veena Mendiratta

Adjunct Professor, Northwestern University
Researcher, ex Nokia Bell Labs



Agenda

- Introduction
- Domain Oriented Architecture - Example I
- Resilience with Service Mesh and Sidecars
- Reliability Models - Example II

Introduction

- Overview
- Benefits & Challenges

Microservices Architectural Style

- An approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API.
- These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery.
- There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies.

- James Lewis and Martin Fowler (2014)

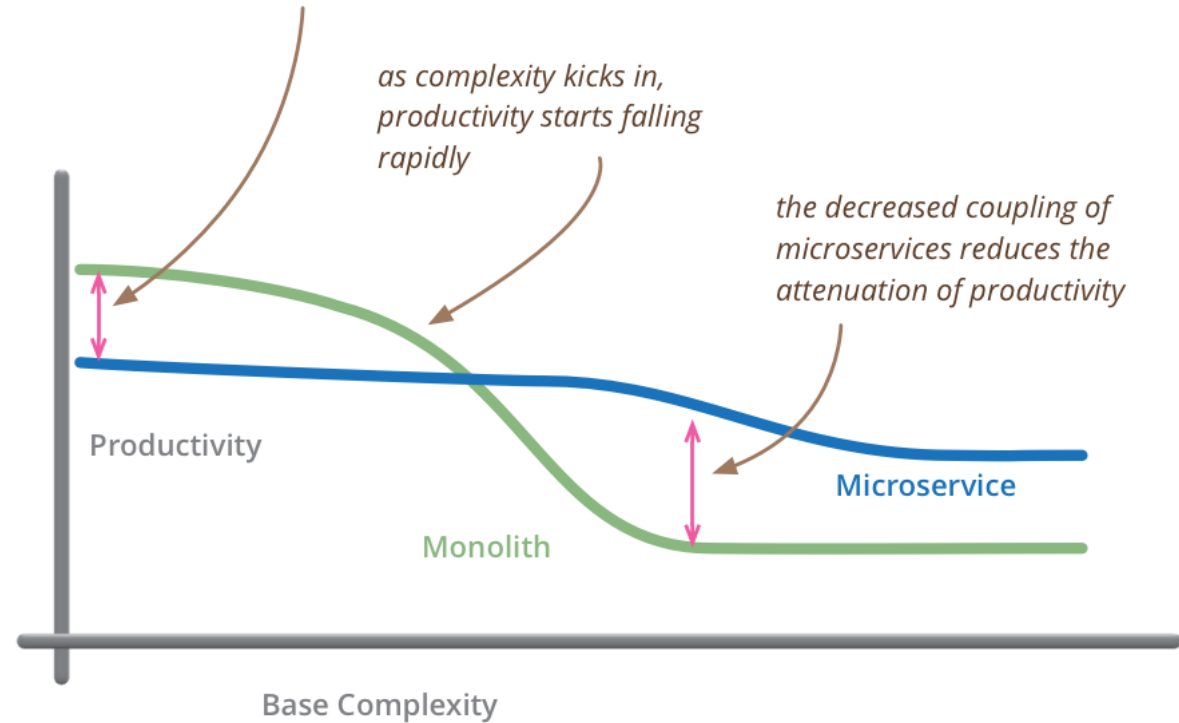
What is a Microservice?

"A particular way of designing software applications as suites of independently deployable services"

□ Martin Fowler

<https://martinfowler.com/bliki/MicroservicePremium.html>

for less-complex systems, the extra baggage required to manage microservices reduces productivity



but remember the skill of the team will outweigh any monolith/microservice choice

What is a Microservice?



What would such a single thing be?

... a queue processor – something that's reading a message from a queue, performing a piece of business logic, and then passing it on
... or it might be something that's cross-functional such as logging, security, service registration, etc.

- A microservice, is a small application that can be deployed **independently**, scaled independently, and tested independently and that has a single responsibility.
 - ranging from a couple of hundred up to a couple of thousand lines of code.
- It is a single **responsibility** in the original sense that it's got a single reason to change and/or to be replaced.
- But the other axis is a single responsibility in the sense that it **does only one thing** and one thing alone and can be easily understood

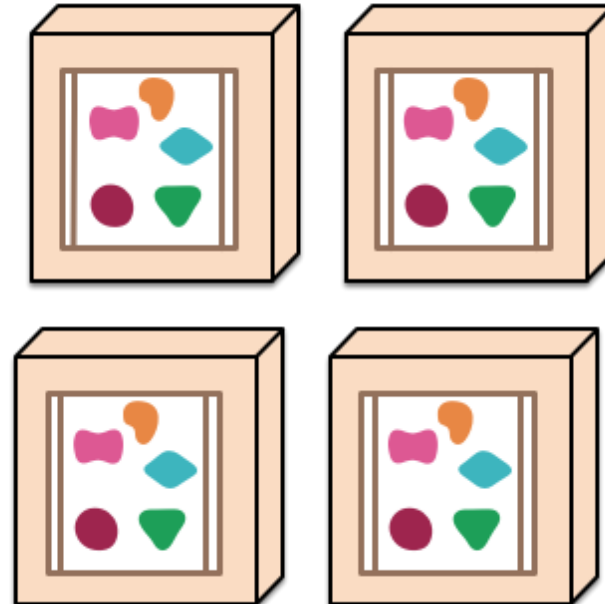
Thönes, Johannes. "Microservices." *IEEE software* 32.1 (2015).

Monolith vs Microservices Architecture

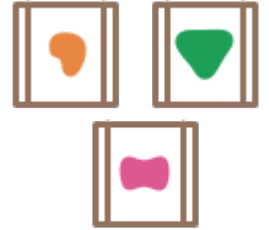
A monolithic application puts all its functionality into a single process...



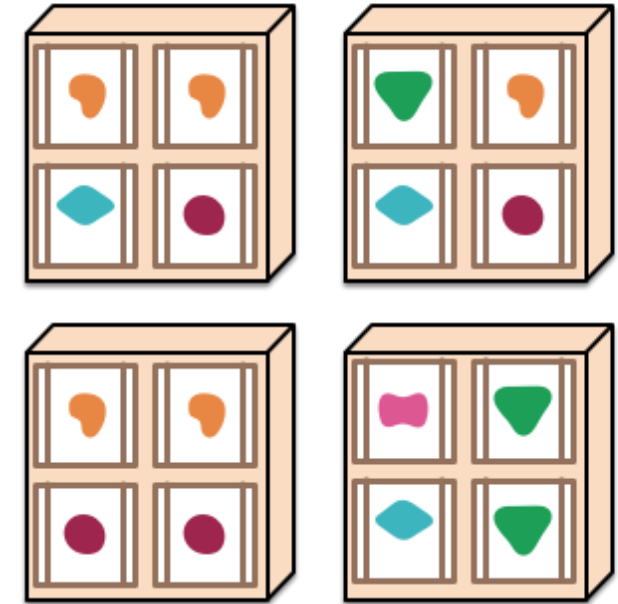
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...

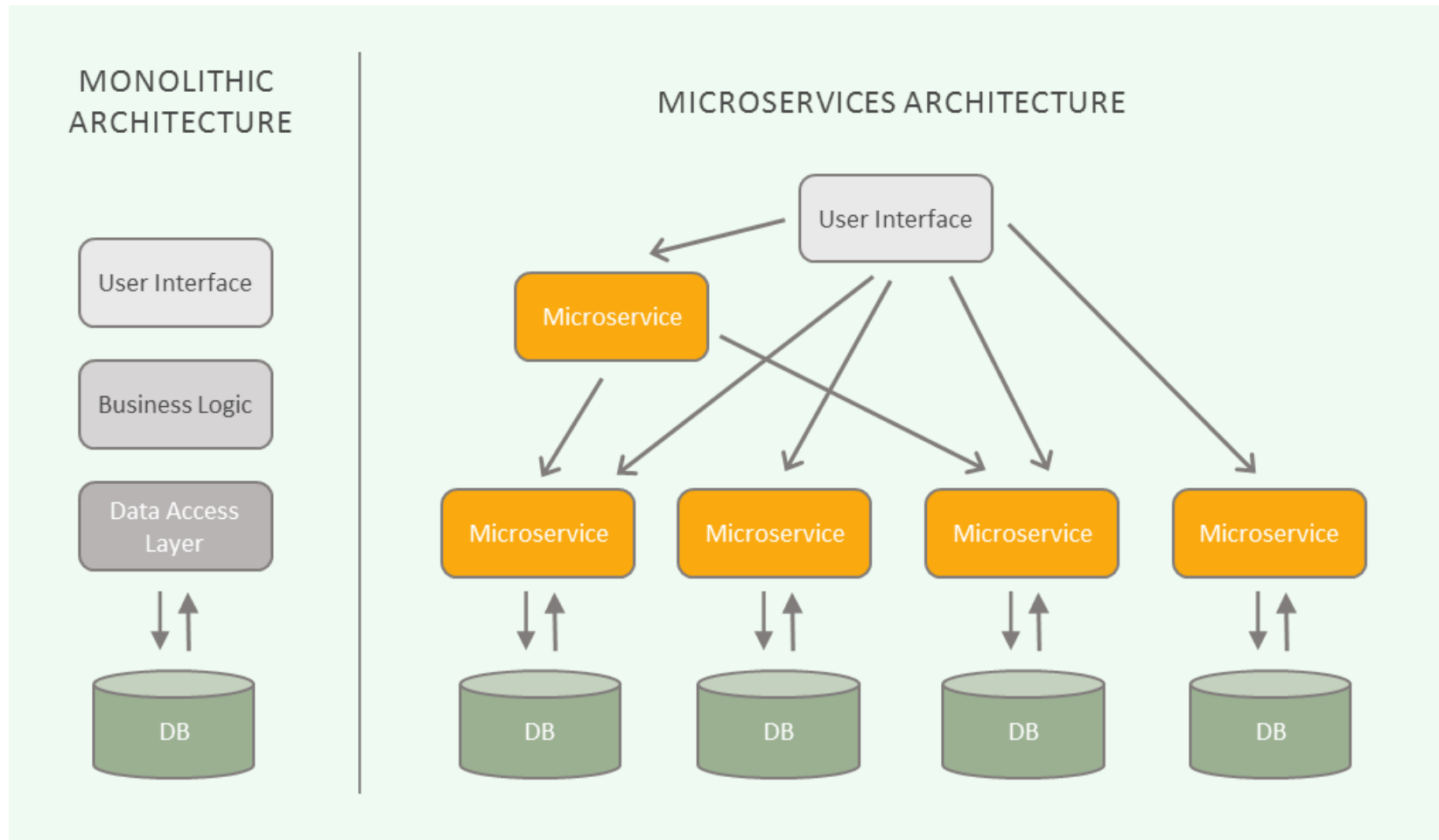


... and scales by distributing these services across servers, replicating as needed.



<https://martinfowler.com/tags/microservices.html>

Monolithic vs Microservices Architecture



A major industry shift for applications

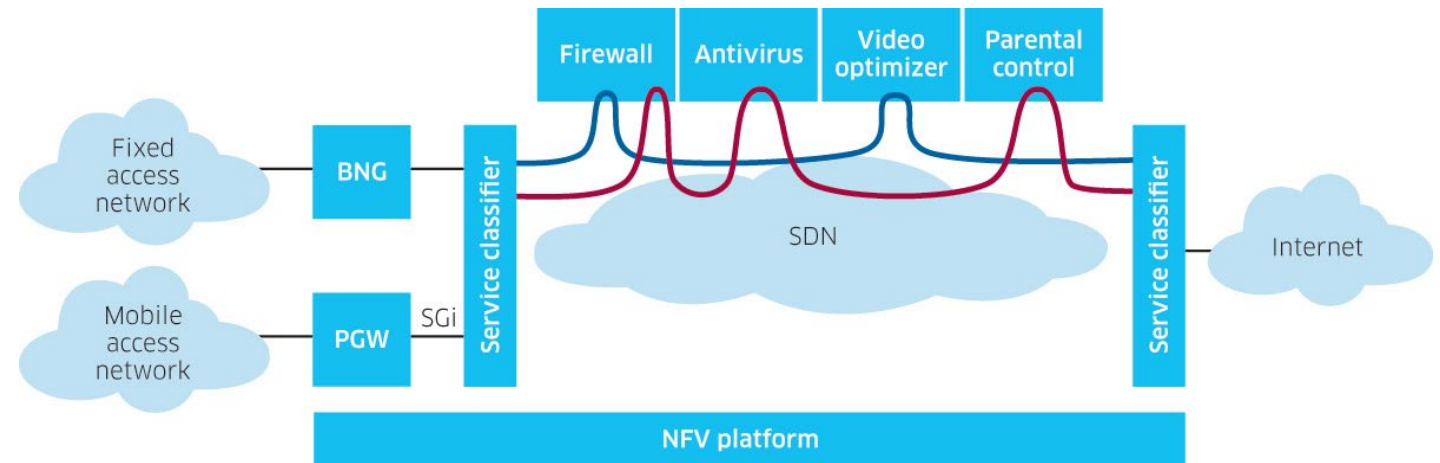
- From monolithic legacy applications
→ microservices architectures
- Functionality is split among hundreds or thousands of small and decoupled microservices
 - Communication is via inter-process protocols (e.g., http, tcp, gRPC)
 - Thus, forming a distributed system
- Part of a larger shift towards a DevOps culture, where development and operations teams work closely to support an application over its lifecycle
- Each microservice can be independently updated after deployment
- Enables selective and flexible updates to functionality or scaling in real-time on operational systems
- Popular open-source technologies such as Docker, Kubernetes, Istio, etc., are used to deploy, scale and manage microservices.



Network service chaining

- Create a virtual chain of connected network services, such as, firewalls, load balancers, network address translation, intrusion protection, etc. using SDN.
- Can be used to set up catalogs of connected services that enable the use of a single network connection for a range of services.
- Provisioning can be done via the orchestration layer.
- Enables the automation of virtual network connections to handle traffic flows for connected services. For example, an SDN controller can apply a service chain to different traffic flows depending on the source, destination or type of traffic.

Network Service Chaining Example (microservices)



Microservices

Claimed Benefits

Developer independence, isolation, scalability, lifecycle automation, relationship to business

- Applications easier to build and maintain
- Independent code evolution
- Teams can deploy and push independently
- Polyglot: each service can be in a different language
- Better resource management
- Independent allocation of resources
- Independent scaling of resources

Challenges

Implementation of cross-cutting concerns implemented using a microservices chassis framework

- Logging,
- Security
- Audit trails
- Service registration,
- Service discovery,
- Configuration management

Microservices are hosted and available over the network with a well-defined interface via a remote procedure call.

- When the number of microservices increases (possibly thousands) this adds complexity, performance overhead, and reliability concerns (discussion in Uber example)

→ Better resilience and fault tolerance – **it depends**

Microservices

Provide benefits ...

- Strong Module Boundaries: Microservices reinforce modular structure, which is particularly important for larger teams.
- Independent Deployment: Simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong.
- Technology Diversity: With microservices you can mix multiple languages, development frameworks and data-storage technologies.

... but come with costs

- Distribution: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure.
- Eventual Consistency: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone needs to manage eventual consistency.
- Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly.

<https://martinfowler.com/articles/microservice-trade-offs.html#summary>

Microservices: Reliability Impacts

Plus

- Separation of concerns
 - each service represents a set of narrowly scoped functionality
- Smaller modules for testing
- Isolation, minimizes error propagation
➔ higher reliability

Minus

- Increased complexity
 - due to large number of microservices
- Many modules and many interfaces for testing

Microservices □ a distributed system

- Typical assumptions for distributed systems that result in errors and failures
 - network is reliable and secure, delay is zero, bandwidth is infinite, topology does not change, there is a central administrator, transport cost is zero, network is homogeneous.
- Reliability problems that are more specific to microservices include:
 - communication failure between microservices that can impact the entire application
 - the risk of failure in the domain of integrations
 - problems maintaining consistent data since each service manages its own database
 - managing service configurations in a consistent way
 - testing microservices is more complex than in monolithic applications
- Failures of microservices:
 - hardware – the underlying infrastructure on which a service operates
 - communication – between different services
 - dependencies – failure within dependencies of a service
 - internal – errors within the code of the service, such as defects

Rotem-Gal-Oz, A., 2006. Fallacies of distributed computing explained. URL <http://www.rgoarchitects.com/Files/fallacies.pdf>, 20.
Dragonì, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L., 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pp.195-216.

Introducing Domain-Oriented Microservice Architecture (Uber)

<https://eng.uber.com/microservice-architecture/>



Example 1

Uber: Motivation for Microservices Architecture

Uber had primarily two monolithic services (circa 2012-2013) and ran into many of the operational issues that microservices solve

- **Availability Risks.** A single regression within a monolithic code base can bring the whole system (all of Uber) down.
- **Risky, expensive deployments.** These were time consuming to perform with the frequent need for rollbacks.
- **Poor separation of concerns.** Difficult to maintain good separation of concerns with a huge code base. In an exponential growth environment, expediency led to poor boundaries between logic and components.
- **Inefficient execution.** These issues combined made it difficult for teams to execute autonomously or independently.

Microservices Architecture: *flexibility* and *autonomy*

As Uber grew from 10s to 100s of engineers with multiple teams, the monolithic architecture tied the fate of teams together and made it difficult to operate independently.

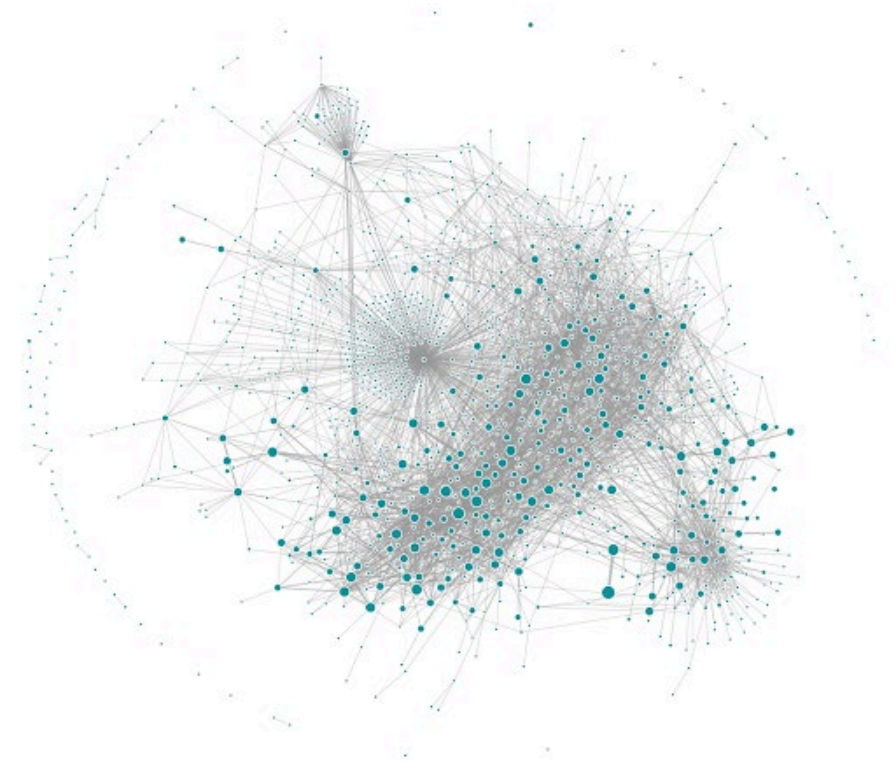
<https://eng.uber.com/microservice-architecture/>

- **System reliability.** Overall system reliability goes up in a microservice architecture. A **single service** can go down (and be rolled back) without taking down the whole system.
- **Separation of concerns.** Architecturally, microservice architectures force you to ask the question “why does this service exist?” more clearly defining the roles of different components.
- **Clear Ownership.** Becomes much clearer who owned what code. Services are typically owned at the individual, team, or org level.
- **Autonomous execution.** Independent deployments + clearer lines of ownership unlock autonomous execution by various product and platform teams.
- **Developer Velocity.** Teams can deploy their code independently, which enables them to execute at their own pace.

Dealing with Complexity

Motivation: as the company grew larger, 100s of engineers to 1000s, system complexity became an issue.

- Microservice architecture trades a single monolithic code base for black boxes whose functionality can change at any time and cause unexpected behavior.
- Engineers had to work through ~50 services across 12 different teams to investigate the root cause of a problem.
- Understanding dependencies between services became difficult, as calls between services can go many layers deep. A latency spike in the n th dependency can cause a cascade of issues upstream. Visibility into what is happening is impossible without the right tools, making debugging difficult.



Domain-Oriented Microservice Architecture (DOMA)

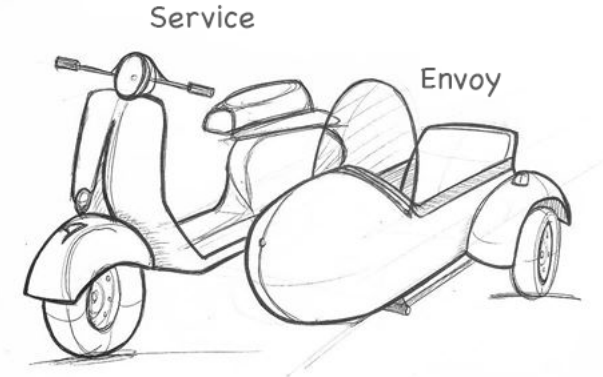
Core principles and terminology

- Domains: collections of related microservices.
- Layers: collections of domains, and that establish what dependencies the microservices within that domain are allowed to take on.
- Gateways: Interfaces for domains that are treated as a single point of entry into a layer.
- A domain is agnostic to other domains: a domain should not have logic related to another domain hard coded inside of its code base or data models.
- Extension architecture: to support well defined extension points within the domain for a team needs to include logic in another team's domain, for example, custom validation logic.

Layers

- **Infrastructure layer.** Provides functionality that any engineering organization could use. Uber's answer to the big engineering questions, such as storage or networking.
- **Business layer.** Provides functionality that Uber as an organization could use, but that is not specific to a particular product category or line of business such as Rides, Eats, or Freight.
- **Product layer.** Provides functionality that relates to a particular product category or LOB, but is agnostic to the mobile application, such as the "request a ride" logic which is leveraged by multiple Rides facing applications (Rider, Rider "Lite", m.uber.com, etc).
- **Presentation.** Provides functionality that directly relates to features that exist within a consumer-facing application (mobile/web).
- **Edge Layer.** Safely exposes Uber services to the outside world. This layer is also mobile application aware.

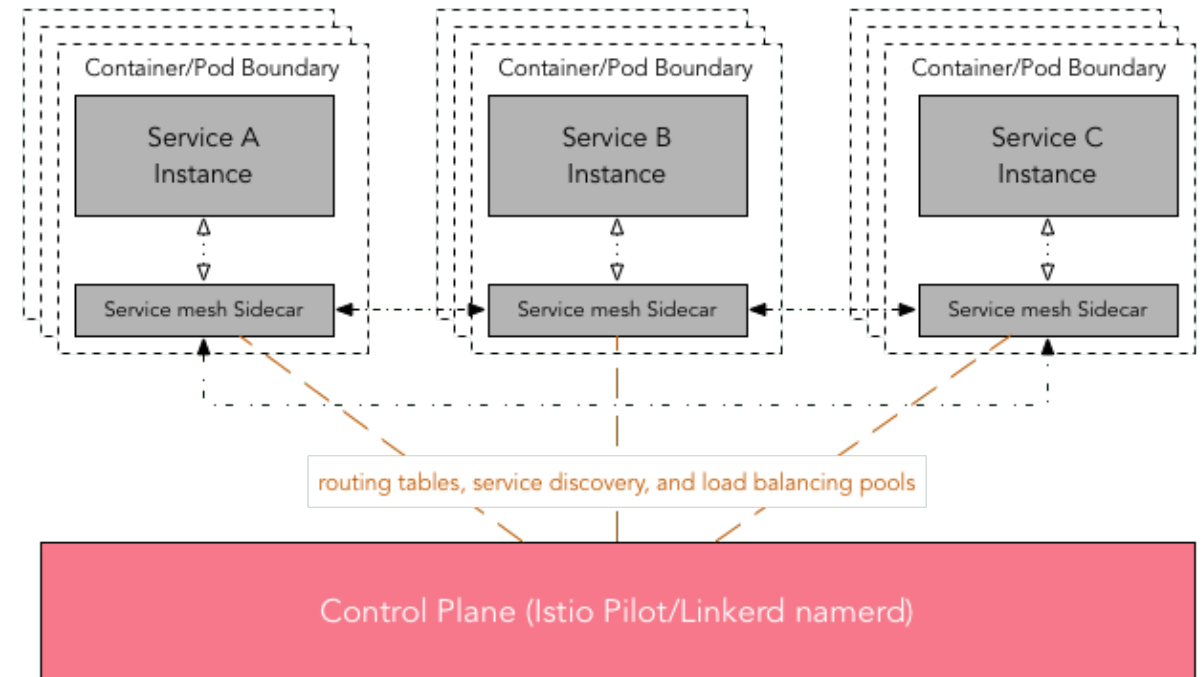
Resiliency with Service Mesh and Sidecars



<https://www.thoughtworks.com/insights/blog/building-service-mesh-envoy-0>

Resiliency with service meshes

- How to ensure **resilience when microservices fail**?
- Cascading failures: microservice fails → microservices that have sent it a request wait for a long time for a response → impacts app availability
- Has a microservice **actually failed**?
 - Provably impossible to know for distributed systems
- Solution: **service meshes** monitor microservices and govern communication among them
- Each microservice instance has an associated service mesh **sidecar** that monitors it
 - Sidecars communicate with each other and the control plane



Istio

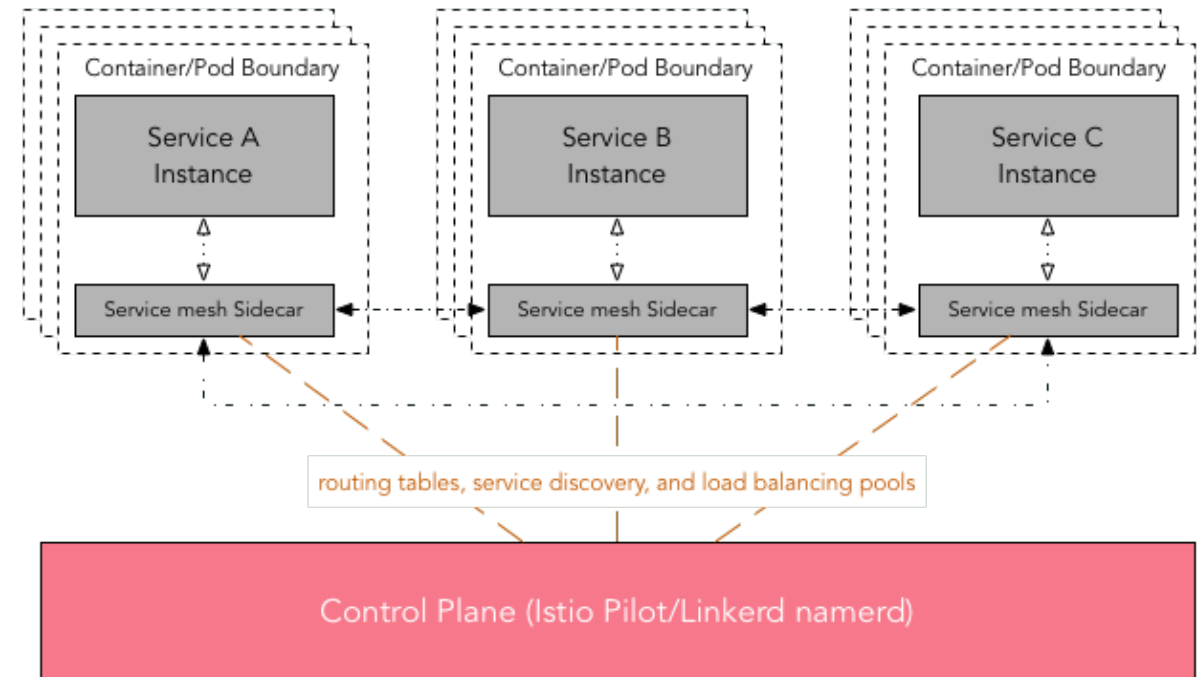


linkerd



Service mesh

- A **platform layer** on top of the infrastructure layer that enables managed, observable, and secure communication between individual services.
- Developed to provide a **microservice-agnostic** framework for ensuring resiliency.
- Obviates 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 **sidecars**.
- **Decouples** application or business logic from network functions.



Sidecar

- Communicate with the service mesh control plane, which makes decisions about the communication and routing between microservices.
- A separate container that runs alongside an application container in a pod.
- Responsible for offloading (data plane) functions required by all apps within a service mesh, such as SSL/mTLS (for authentication), traffic routing, high availability; and implementing circuit breakers.
- Managed by some type of control plane within the service mesh.

<https://www.c-sharpcorner.com/article/microservices-design-using-sidecar-pattern//>

Sidecar Pattern

- Deploy application into a separate process or container to provide isolation and encapsulation.
- Pattern can enable applications to be composed of heterogeneous components and technologies.

~ Watchdog pattern



Sidecar Pattern Notes

Benefits of Using a Sidecar Pattern:

- Reduces the complexity in the microservice code by abstracting the common infrastructure-related functionalities to a different layer.
- Reduces code duplication in a microservice architecture since you do not need to write configuration code inside each microservice.
- Provide loose coupling between application code and the underlying platform.

How Does the Sidecar Pattern Work?

- The service mesh layer can live in a sidecar container that runs alongside your application. Multiple copies of the same sidecar are attached alongside each of your applications.
 - All the incoming and outgoing network traffic from an individual service flows through the sidecar proxy. Thus, the sidecar manages the traffic flow between microservices, gathers telemetry data, and enforces policies. The service is not aware of the network and knows only about the attached sidecar proxy.
- ➔ Essence of how the sidecar pattern works – it abstracts away the network dependency to the sidecar.

Resilience with service meshes and sidecars

- Programmable via declarative APIs
- Can be deployed and updated at run-time of the microservices
- Circuit breakers are used to implement resilient applications
- Attributes in the individual sidecars are configured with timeout values and retry values to 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.
- 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 noncritical microservice if it is likely it has failed.
- Such circuit breakers prevent cascading failures.

Resilience with service meshes and sidecars

Techniques

- **Timeouts** to detect microservices delays or failures
- **Circuit breakers** to determine likely microservice failure and trip accordingly
- **Critical** microservice determined to have failed
→ notify dependent microservices
(abort current request)
- **Non-critical** microservice determined to have failed → continue in a degraded mode

Circuit breaker tripping - when?

- **Quick** to decide microservice failure
→ may trip unnecessarily;
Current requests are aborted, **lowering availability**
- **Slow** to decide microservice failure
→ may wait too long to detect failure;
resulting in cascading failures, **lowering availability**
- **Modeling** to determine interval

Reliability Models for Microservices Architectures

Example II

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

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.

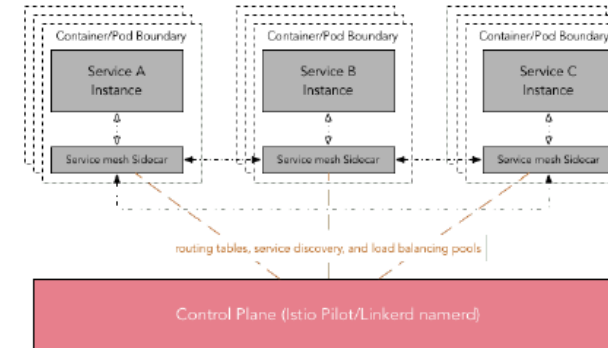


Fig. 1. Service mesh architecture [5]

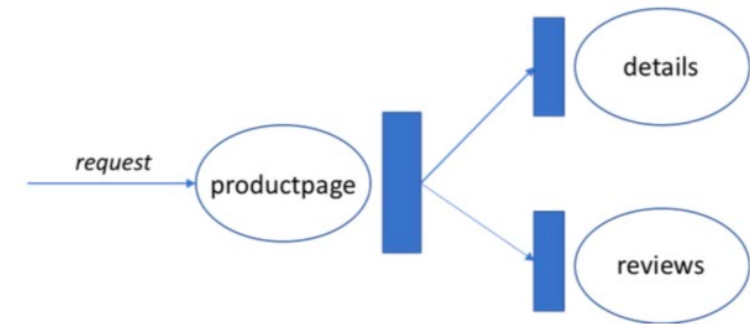
Jagadeesan, L.J. and Mendiratta, V.B., 2020, October. When Failure is (Not) an Option: Reliability Models for Microservices Architectures. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 19-24). IEEE.

Modeling Framework

- Modeling framework for service meshed **microservices availability**
 - Based on Continuous Time Markov Models (**CTMC**)
 - **Micro-model** for microservices design (**Prism tool**)
 - **Macro-model** for thousands of microservices at scale (R package **markovchain**, SHARPE)
- Accounts for tradeoffs in **circuit breaker settings** and availability
- Supports **critical and non-critical microservices**, and degraded operation
- Associated analyses of **availability** under varying assumptions

Micro-model of Istio *bookinfo* example

- Model of a small set of microservices and service mesh sidecars, useful for design
 - Functionality and failure behavior of microservices, and circuit breakers in sidecars
 - Degraded mode → failed non-critical microservices are bypassed by service mesh
 - Represented as a set of concurrent communicating Continuous Time Markov Models (CTMCs)
 - Analysis of availability with varying parameters
- Case study: Bookinfo example of Istio service mesh (book ordering application)
 - Processes one incoming request at a time, accepts incoming request if not busy, drops new requests if busy
 - Critical: details; Non-Critical: reviews
 - Model and analysis using PRISM stochastic model checker
 - Analyzed steady-state probability of accepted, completed, and degraded requests with increasing arrival rate



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

Analysis Results: Micro-model

STEADY-STATE PROBABILITY OF ACCEPTED REQUESTS

Circuit breaker rate μ_{cb}	Arrival rate γ_{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

STEADY-STATE PROBABILITY OF COMPLETED REQUESTS

Circuit breaker rate μ_{cb}	Arrival rate γ_{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

Prob of request acceptance and completion are lower with higher request arrival rate
INCREASES with circuit breaker rate
DECREASES with circuit breaker rate

STEADY-STATE PROBABILITY OF COMPLETED OR DEGRADED REQUESTS

Circuit breaker rate μ_{cb}	Arrival rate γ_{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

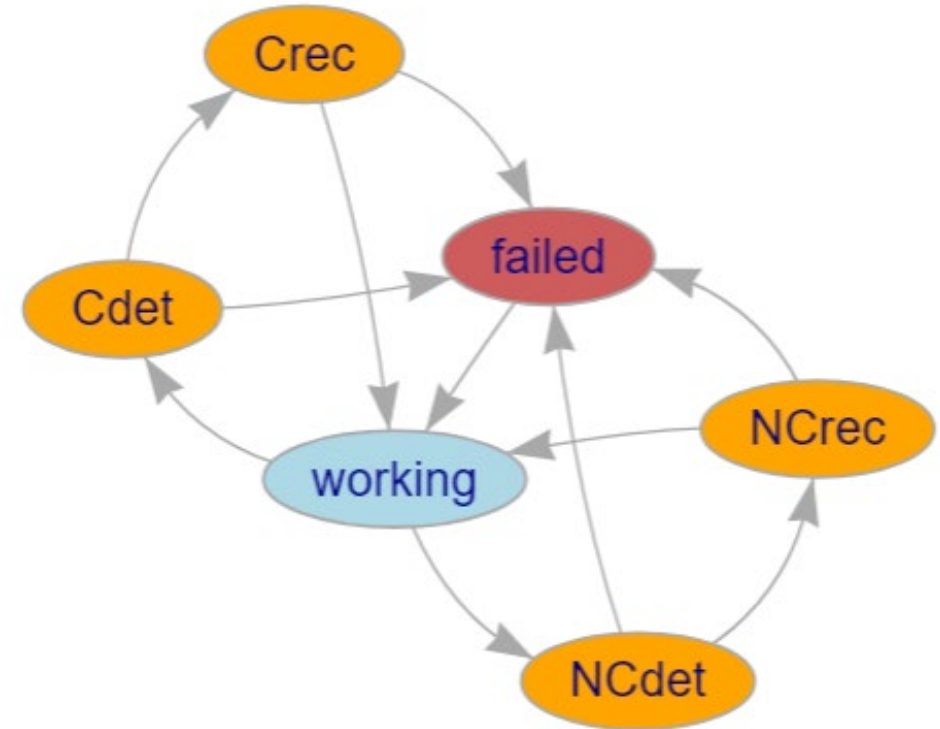
Prob of successful or degraded request
 completion **DECREASES** with circuit breaker rate

Key observations:

- Faster circuit breakers **increase** the probability that an incoming request is accepted for processing
- ... but **decrease** the probability that requests complete successfully (fully or degraded)

Macro Model - overview

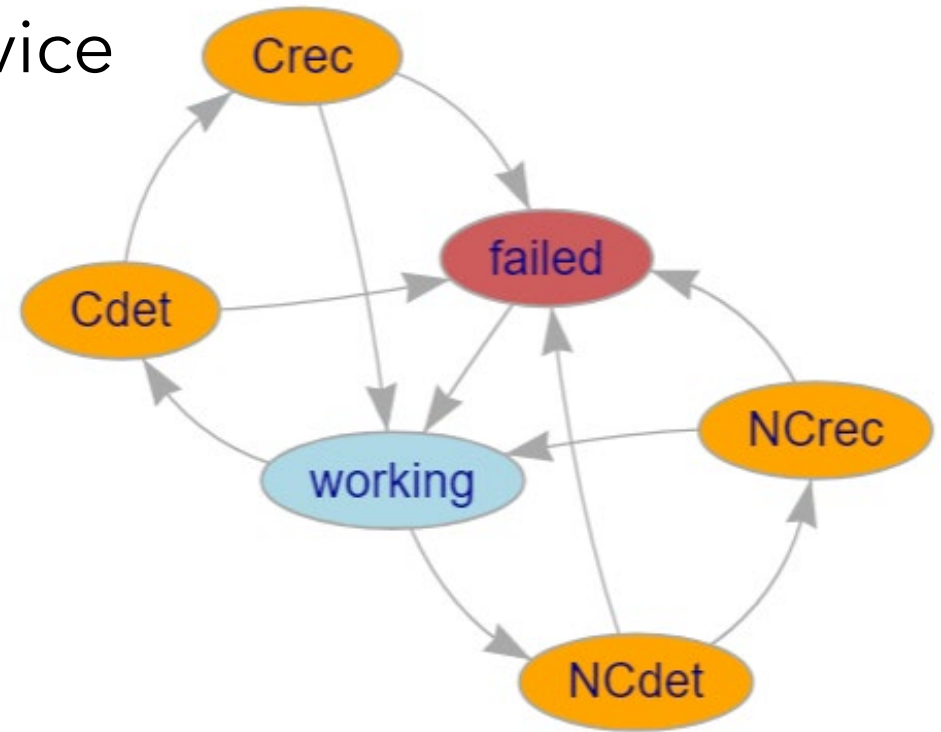
- Model of **thousands of microservices** in a service mesh
 - Includes failure/detection/recovery
 - Includes **critical and non-critical** microservices
- Represented as a CTMC
- Analysis using R, under varying parameters
- Analyzed probabilities of being in:
 - working state,
 - expected down time, and
 - expected degraded time



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

Macro Model – composite service

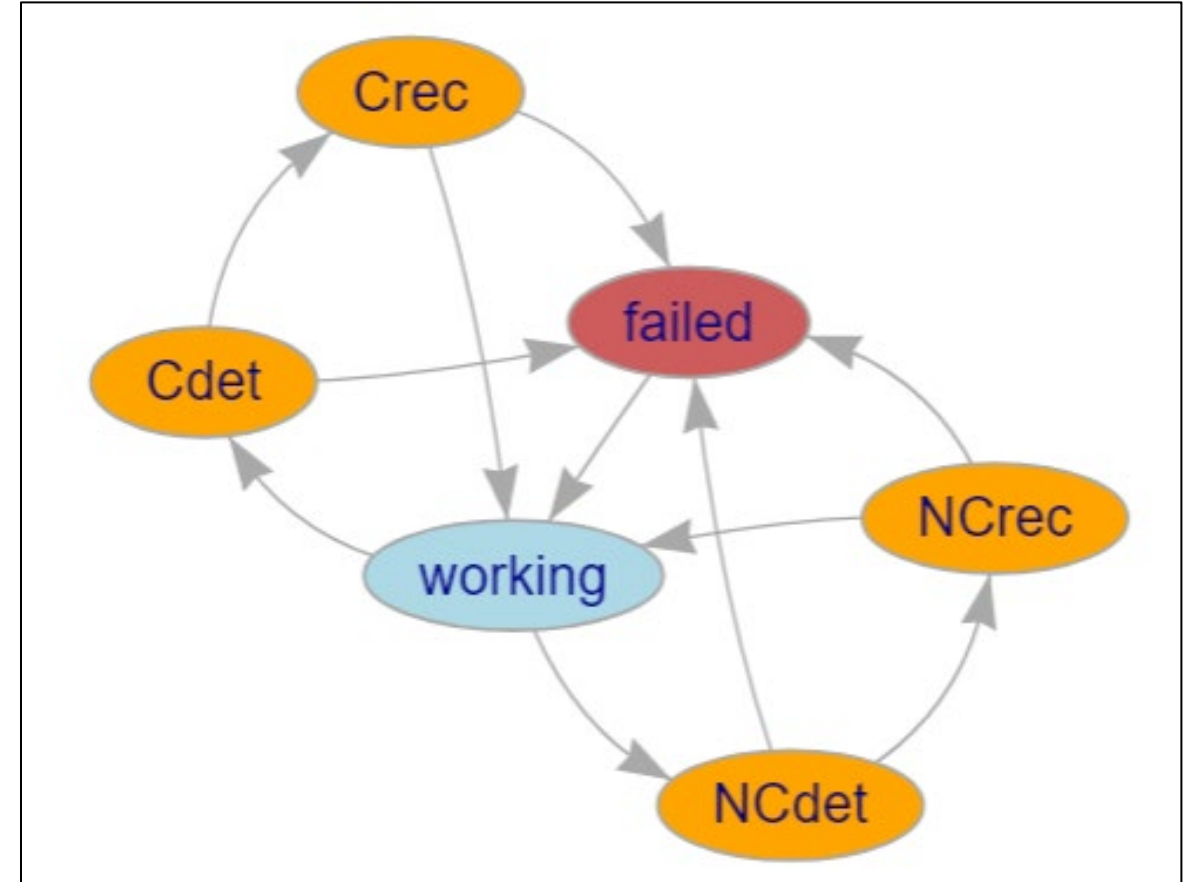
- Model Objective: assess the availability of large scale microservices deployments for a range of parameter values.
- Microservices: critical (CM) or non-critical NCM)
- Composite service: comprised of CMs and NCMs with the following properties:
 - Possible states: up, down, up with degraded functionality
 - Up iff all microservices are up
 - Down iff one or more CMs has failed or is in the recovery state
 - Up with degraded functionality iff all CMs are up and one or more NCMs has failed or is in the recovery state



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

Macro Model – state transitions

- State (*working*) represents the normal state.
- 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 λ_C and λ_{NC} respectively.
- Successful detection leads to a transition to the recovery states *Crec* and *NCre* 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* state at a rate μ_R which represents the off-line long recovery/repair rate.

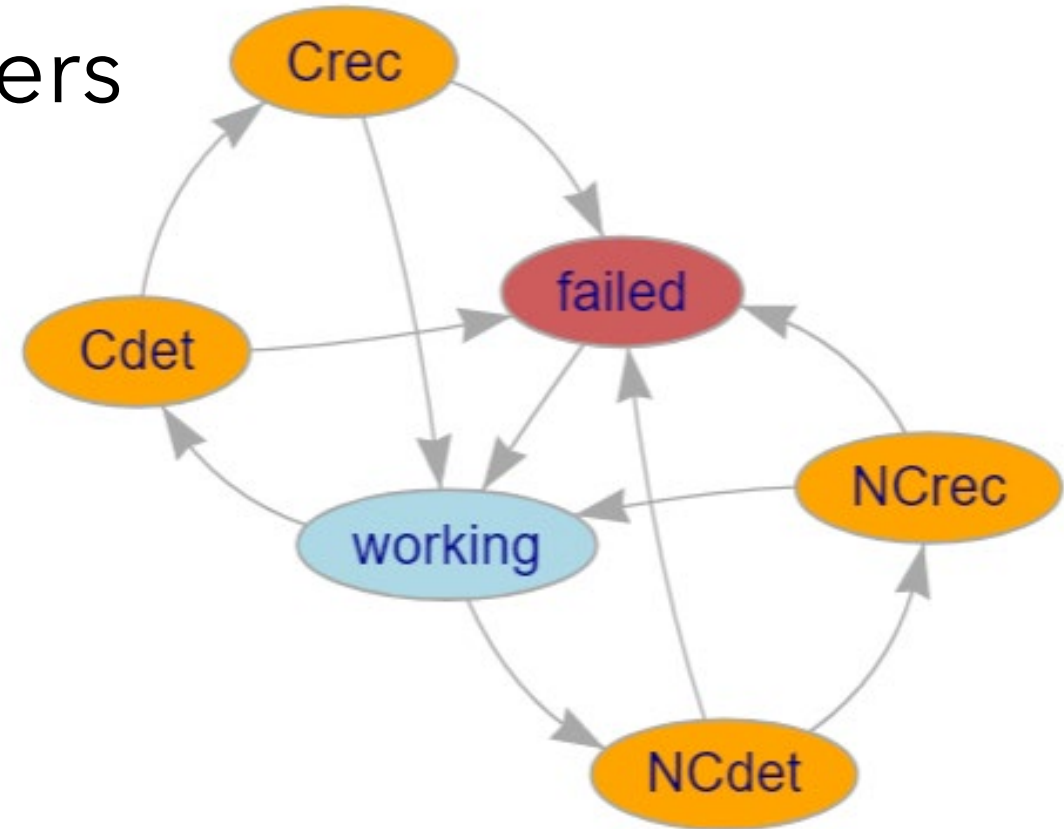


working	normal working state, all services up
Cdet	critical services detection, failed state
Crec	critical services recovery, failed state
NCdet	non-critical services detection, degraded state
NCre	non-critical services recovery, degraded state
failed	service unavailable, failed state

Macro Model - parameters

- p_C proportion of microservices that are critical (varied between 0.1 and 0.5)
 λ composite service failure rate (100/year)
 λ_C critical microservices failure rate ($p_C \lambda$)
 non-critical λ_{NC} critical microservices failure rate ($(1 - p_C) \lambda$)
 μ_{Cd} critical microservices failure detection rate (varied between 60 and 240 per hour)
 μ_{NCd} non-critical microservices failure detection rate (varied between 60 and 240 per hour)

 μ_{Cr} critical microservices failure recovery rate (varied between 60 and 240 per hour)
 μ_{NCr} non-critical microservices failure recovery rate (varied between 60 and 240 per hour)
 μ_R long recovery/repair rate – after failed detection or recovery (3/hour)
 δ probability of successful microservices failure detection (varied between 0.95 and 0.99)
 ρ probability of successful microservices failure recovery (varied between 0.95 and 0.99)



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

Analysis of Results: Macro-model

EXPECTED PROBABILITY IN WORKING STATE

μ_{det}	Critical Services Proportion					
and	$\delta=0.95, \rho=0.95$			$\delta=0.99, \rho=0.99$		
μ_{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

EXPECTED DOWNTIME, MINUTES/YEAR

μ_{det}	Critical Services Proportion					
and	$\delta=0.95, \rho=0.95$			$\delta=0.99, \rho=0.99$		
μ_{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

EXPECTED DEGRADED TIME, MINUTES/YEAR

μ_{det}	Critical Services Proportion					
and	$\delta=0.95, \rho=0.95$			$\delta=0.99, \rho=0.99$		
μ_{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

Key observations

- Decrease in detection and recovery rates lead to
 - decrease in expected probability of working state
 - increase in expected downtime and degraded time
- Higher proportion of critical to non-critical services
 - Does not affect probability of working state
 - Increases expected downtime
 - Decreases expected degraded time

Summary

- Modeling framework and associated analyses for microservices and service mesh
 - Micro-level and macro-level
 - Addressed tradeoffs in resilience □ micro-level and macro-level
- Future work
 - Deterministic distributions
 - Transient failures

Microservices Reliability

- Configuration of Attributes



Examples

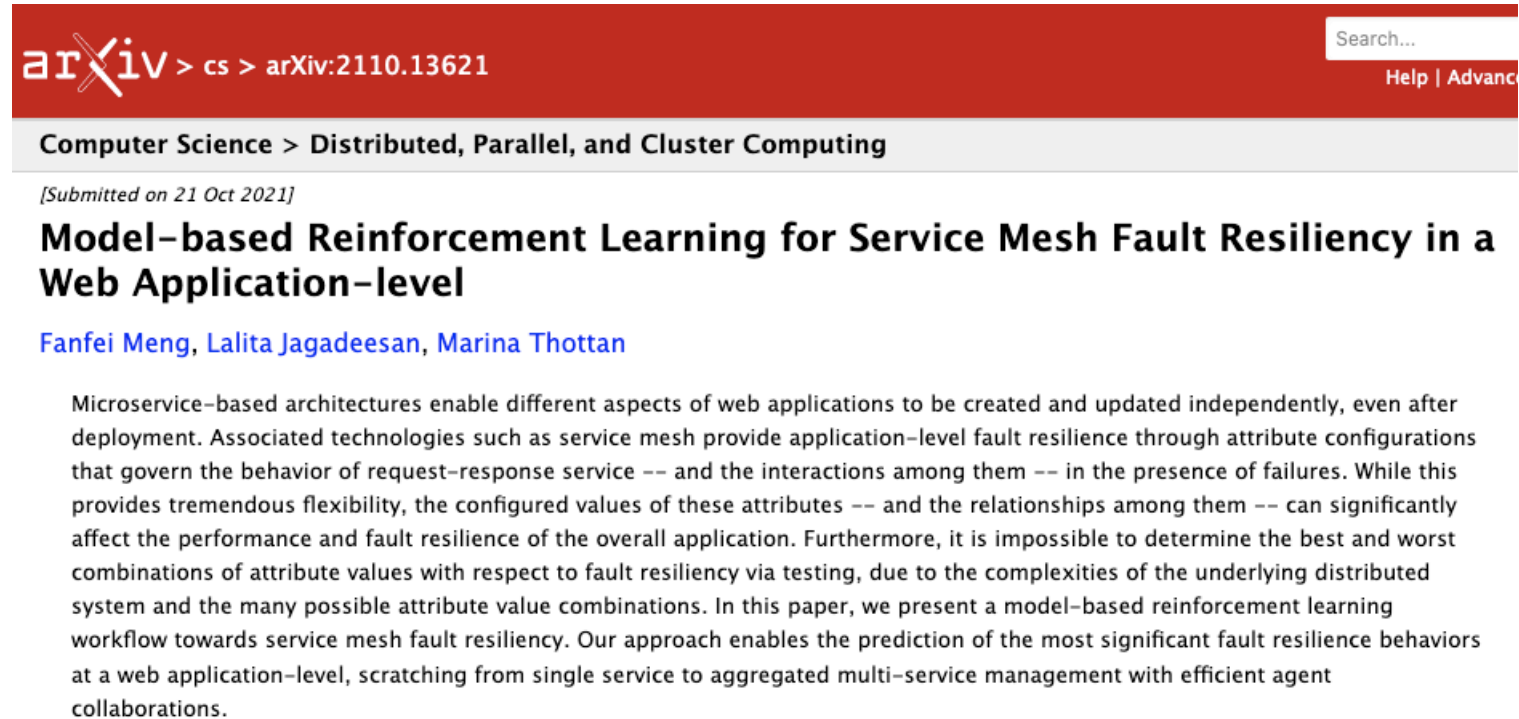
Microservices Reliability

– Configuration of Attributes

- Microservice-based architectures enable applications to be created and updated independently.
- Service meshes provide fault resiliency through attribute configurations that govern application-level behavior in response to failures, e.g., circuit breaking, request retry.
- Configured values of these attributes can affect the performance and fault resilience of the overall application.
- Given the large number of possible attribution combinations it can be a challenge to determine the optimal values, or the “worst combination of values” for testing.
- Nor is a static combination of attribute values appropriate for the dynamic environment of microservices.
- Proposed solutions:
 - experimental approaches
 - machine learning approaches

ML Approach – Reinforcement Learning (RL)

- Model-based reinforcement learning that determines the combinations of attribute and load settings that result in the most significant fault resilience behaviors at an application level.
- The configuration settings that yield the “worst-case” rewards give insight into which combinations of configurations should be tested rigorously during load testing to ensure robust fault recovery.



The screenshot shows the arXiv preprint interface. At the top, the arXiv logo is followed by the path 'cs > arXiv:2110.13621'. A search bar is on the right. Below the header, the category 'Computer Science > Distributed, Parallel, and Cluster Computing' is displayed. The submission date '[Submitted on 21 Oct 2021]' is shown. The title 'Model-based Reinforcement Learning for Service Mesh Fault Resiliency in a Web Application-level' is prominently displayed. The authors 'Fanfei Meng, Lalita Jagadeesan, Marina Thottan' are listed below the title. The abstract text describes how microservice-based architectures enable independent creation and updates of web applications, and how service mesh provides application-level fault resilience through attribute configurations. It notes that while this provides flexibility, the configured values and relationships can significantly affect performance and fault resilience, making it difficult to determine the best and worst combinations of attribute values through testing alone. The paper presents a model-based reinforcement learning workflow to predict the most significant fault resilience behaviors at a web application-level, starting from single service to aggregated multi-service management with efficient agent collaborations.

Meng, F., Jagadeesan, L. and Thottan, M., 2021. Model-based reinforcement learning for service mesh fault resiliency in a web application-level. *arXiv preprint arXiv:2110.13621*.

Experimental Approach

– Self-Adaptive Circuit Breaking and Retry

How should the retry mechanism be configured to improve the resiliency of a microservice when circuit breaking is used?

- Based on experiments, developed a controller that improves throughput by dynamically adjusting the number of retry attempts and retry timeout.
- One of the key metrics that the retry controller considers is the **service carried response time**, which has a direct impact on the user experience.
- Monitoring the response time allows the controller to adjust the retry configuration and prevent users from experiencing long wait times or timeouts due to failed requests.
- Additionally, the number of failed requests and the circuit breaker state is taken into account to avoid overloading the service with excessive retry requests.
- By dynamically adjusting the retry configuration based on these performance metrics, the controller can optimize the performance, reliability, and availability of the service.

2023 IEEE International Conference on Cloud Engineering (IC2E)

Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry

Year: 2023, Pages: 32-42

DOI Bookmark: [10.1109/IC2E59103.2023.00012](https://doi.org/10.1109/IC2E59103.2023.00012)

Authors

[Mohammad Reza Saleh Sedghpour](#), Umeå University, Department of Computing Science, Umeå, Sweden

[David Garlan](#), Carnegie Mellon University, School of Computer Science, Pittsburgh, USA

[Bradley Schmerl](#), Carnegie Mellon University, School of Computer Science, Pittsburgh, USA

[Cristian Klein](#), Umeå University, Department of Computing Science, Umeå, Sweden

[Johan Tordsson](#), Umeå University, Department of Computing Science, Umeå, Sweden



DOWNLOAD PDF



SHARE ARTICLE



GENERATE CITATION

Abstract

Microservice-based architectures consist of numerous, loosely coupled services with multiple instances. Service meshes aim to simplify traffic management and prevent microservice overload through circuit breaking and request retry mechanisms. Previous studies have demonstrated that the static configuration of these mechanisms is unfit for the dynamic environment of microservices. We conduct a sensitivity analysis to understand the impact of retrying across a wide range of scenarios. Based on the findings, we propose a retry controller that can also work with dynamically configured circuit breakers. We have empirically assessed our proposed controller in various scenarios, including transient overload and noisy neighbors while enforcing adaptive circuit

Sedghpour, Mohammad Reza Saleh, David Garlan, Bradley Schmerl, Cristian Klein, and Johan Tordsson. "Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry." In *2023 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 32-42. IEEE, 2023.

References

- Microservices Guide
- Thönes, Johannes. "Microservices." *IEEE software* 32.1 (2015): 116-116.
- Introducing Domain-Oriented Microservice Architecture
- Building a Service Mesh with Envoy
- Sidecar Design Pattern In Microservices
- Jagadeesan, L.J. and Mendiratta, V.B., 2020, October. When Failure is (Not) an Option: Reliability Models for Microservices Architectures. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 19-24). IEEE.