1. What performance metrics would you look for when evaluating microservices? How would you tell bad performance from good performance?

The overall health of a microservice affects its performance. To be able to list the metrics we can use to measure microservice performance, we need to first define what to measure, and at what levels and areas to make the measurements in. The general health of a system/service constitutes of several things and one or more of these can be measured using one or more of the following metrics/techniques.

Factors that constitute the health of a Microservice? *(What to measure?)*

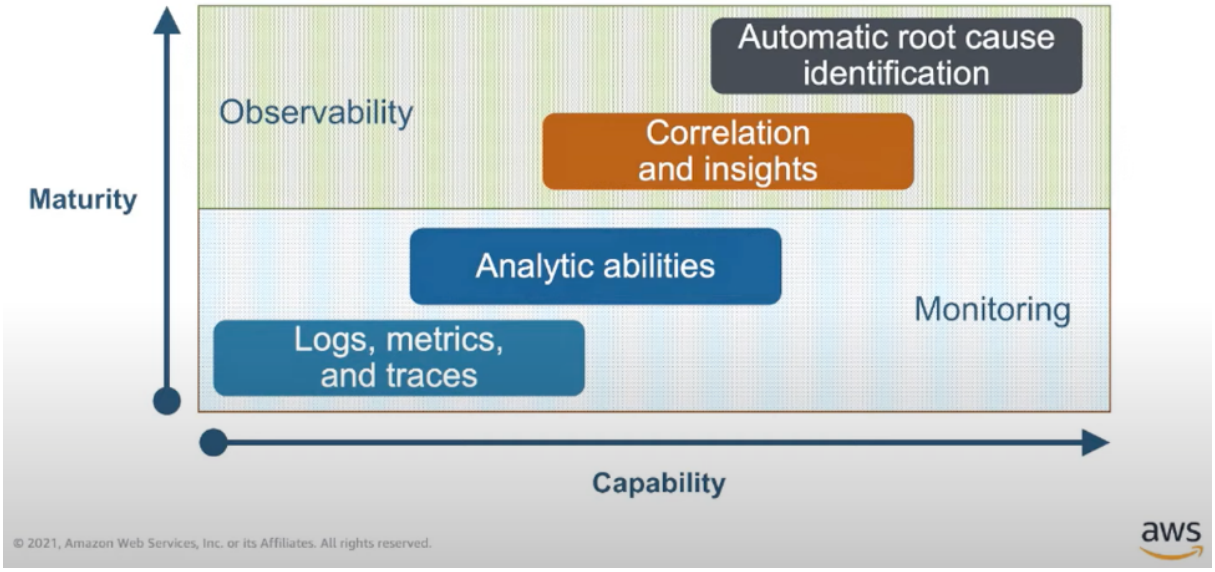
- Reliability
- Throughput (Number of requests or transactions per second)
- Service response time (The time between client sending a request and receiving a response)
- Availability
- Fault Tolerance
- Saturation (Or the amount of load on the service)
- Scalability
- Latency (Duration for which a request is waiting to be handled)
- Resiliency/Recoverability (example - Chaos Monkey by Netflix)
- Security (OWASP microservice security threats [5], Authentication, Authorization, Data encryption at REST and in-transit, network security, ACLs, Resistance to Denial of service attacks, etc.)
- Stateful-ness v/s Statelessness (i.e. is the Microservice able to honor the stateful or stateless nature of the transactions?)
- SLA/SLO contracts
    - TODO – DDIA 99% concepts etc.
- Number and severity of Errors/Failures
- System up-time v/s downtime
- Number of Error Budget breaches [6]
- Hardware/node resources health
- Staleness of data returned by the service or application lag which lies beyond the acceptable standards for the service (where applicable)
    - Example – data staleness introduced by the lag introduced at the database layer due to replication across read/write replicas, network delays, etc.

Two levels of Microservice health evaluation:

- MONITORING –
    - Tells whether the Microservice is working as expected
    - Example tool – Prometheus
    - Involves
        - End user experience monitoring

    - Service interaction monitoring
    - End-to-end performance monitoring
    - Service health monitoring
- OBSERVABILITY
    - Helps you understand WHY a Microservice is not working as expected
    - Example - How to implement Observability in AWS? [8]

Both Monitoring and Observability can be implemented at various levels of the 'Observability Maturity Model' [1] using different tools and techniques.



We can either directly add instrumentation to the microservice code or use third party tools to set them up for monitoring and observability (e.g. APM libraries with Elasticsearch/New Relic, Logstash, custom or third party instrumentation APIs, etc.).

Performance metrics to look for to measure the health of a Microservice:

- **Service Response Time**
    - The time between client sending a request and receiving a response. Includes network delays and queueing delays as well.
    - It is not a single number, but a distribution of values you can measure [4].

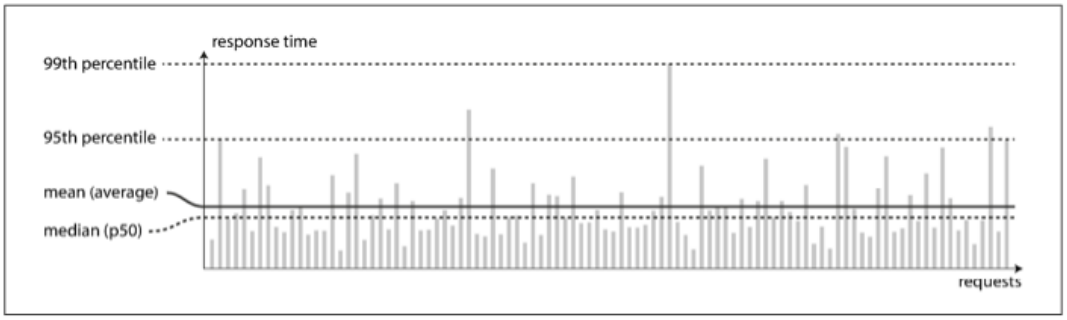


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

…Contd.

- **Throughput**
  - Constitutes the number of requests per second

- **Scalability**
  - Number of transactions per second, the number of requests per second, and the latency of transactions

- **Saturation (Or the amount of load on the service)**
  - Metrics provided by the RED method, Golden signals, APM and Observability

- **Latency**
  - Metrics provided by utilizing Instrumentation techniques, RED method, Golden signals, APM and Observability

- **Number and severity of Errors/Failures**
  - Both at the service mesh level and individual service level

- **Service downtime v/s up-time ratio**
  - As the name suggests, impacts availability.

- **SLIs, SLOs and Error Budgets (to measure service reliability, availability, etc.) [6]**
  - **SLI**
    - Is a quantifiable measure of service reliability
    - SLI = (Good Events/Valid Events) * 100
    - Is expressed as a percentage
  - **SLO**
    - Is a reliability target defined as an objective for each SLI.
    - Should capture performance and availability levels that, if barely met, would keep the typical customer of a service happy
    - Tip - Measure SLO achieved and try to be slightly over target
  - Services need SLOs!
  - **Error Budgets**
    - An SLO implies an acceptable level of unreliability. This is a budget that can be allocated and measured as a service metric!
    - An **error budget** is the amount of error that your service can accumulate over a certain period of time before your users start being unhappy. You can think of it as the pain tolerance for your users, but applied to a certain dimension of your service: availability, latency, and so forth. [6]
    - Remaining service error budget drives prioritization of engineering effort.
    - Example - Imagine that we are measuring the availability of a microservice. The availability is measured by the amount of requests responded with an error, divided by all the valid requests the service receives, expressed as a percentage. If we decide that the objective of that availability is 99.9%, the

…Contd.

error budget is 0.1%. We can serve up to 0.1% of errors (preferably a bit less than 0.1%), and users will happily continue using the service.

- **APDEX score [3]**
  - Apdex is an industry standard to measure the satisfaction of users based on the response time of applications and services
  - The Apdex score is a ratio of satisfied and tolerating requests to the total requests made.

$$\text{Apdex Score} = \frac{\text{SatisfiedCount} + (\text{ToleratingCount} / 2)}{\text{TotalSamples}}$$

  - A percentage representation of this score is known as the Health Indicator of a service.
  - Each *satisfied* request counts as one request, while each *tolerating* request counts as half a *satisfied* request.
  - Apdex specifications recommend the following Apdex Quality Ratings by classifying Apdex Score as
    - Excellent (0.94 – 1.00)
    - Good (0.85 – 0.93)
    - Fair (0.70 – 0.84)
    - Poor (0.50 – 0.69)
    - Unacceptable (0.00 – 0.49)
  - A service with higher traffic flow is an indication that this experience is impacting a significant number of users on the service mesh.

- **Platform metrics**
  - Monitoring platform metrics is critical to keeping microservices infrastructure running smoothly.
  - These include-
    - Number of requests per second/minute
    - Failed requests per second
    - Average response time per service endpoint
    - Distribution of time required for each request
    - Success/failure/error rates

- **Resource metrics**
  - CPU and memory utilization
  - Host count
  - Live threads or number of threads spawned by a service
  - Heap usage etc.

- **Golden signals/metrics**
  - Availability (e.g. Percentage of errors on total requests)
  - Health (e.g., Time to Live pings and other health checks)
  - Request Rate (Number of requests per second)
  - Saturation (Load on the system – e.g., Queue depth or available capacity)
  - Utilization (e.g., CPU or memory usage as a percentage)
  - Error rate
  - Latency (measured in 95^{th} or 99^{th} percentile as explained above)

- **Metrics related to the RED method**
  - RED (Rate, Errors, Duration)
  - Rate – Number of requests service is handling per second
  - Error – Number of failed requests per second
  - Duration – The amount of time each request takes

- **Security related metrics**
  - OWASP microservice security threats [5], Authentication, Authorization, Data encryption at REST and in-transit, network security, ACLs, Denial of service attacks, etc.)

- **Distributed tracing in case of Service Mesh [9]**
  - Distributed tracing provides a way to monitor and understand behavior by monitoring individual requests as they flow through a mesh. Traces empower mesh operators to understand service dependencies and the sources of latency within their service mesh. All the above metrics can be measured via distributed tracing.
  - Example tool – Jaeger [10]

Health of a service

- While health relates to a service, we can also analyze the interactions between two services and calculate the health of the interaction. This health calculation of every interaction on the mesh helps us establish a critical path, based on the health of all interactions in the entire topology.
- An unhealthy service participating in a high throughput transaction could lead to excessive consumption of resources.
- Tuning service that is a part of a high throughput transaction offers exponential benefits when compared to tuning an occasionally used service.

Bad v/s good performance?

- When a service does not meet the established SLIs, SLOs or Error budgets as defined above, we can say that the service is unperforming. Error budgets can then be used to improve the golden signals and performance defined above.

…Contd.

Other comments

- To be able to measure the performance of the Microservices, the load parameters on the services need to be determined. Once these are defined, then we can discuss Microservice scalability and the associated performance related questions [4].
    - Examples –
        - Requests per second or throughput handled by the Service Mesh (also by each microservice)
        - Maximum number of concurrent requests per second
        - Ratio of reads v/s writes to a database
- Microservice performance evaluation can be performance on several levels:
    - Service Mesh level (Overall health of the *Service Mesh* than individual services)
    - Microservice level
    - Application level (relevant low level metrics within Microservices)
        - Application Performance Monitoring (APM) and Dashboards – APDEX Score, Custom acceptable thresholds, etc.
        - Example third party APM libraries are provided by Elasticsearch, New Relic etc., as an example.
- This is a broad topic but the above should serve as a starting point for measuring performance of microservices.

2. What type of testing would you put your microservices through? How does testing affect the validity of the microservice?

Types of Microservice testing

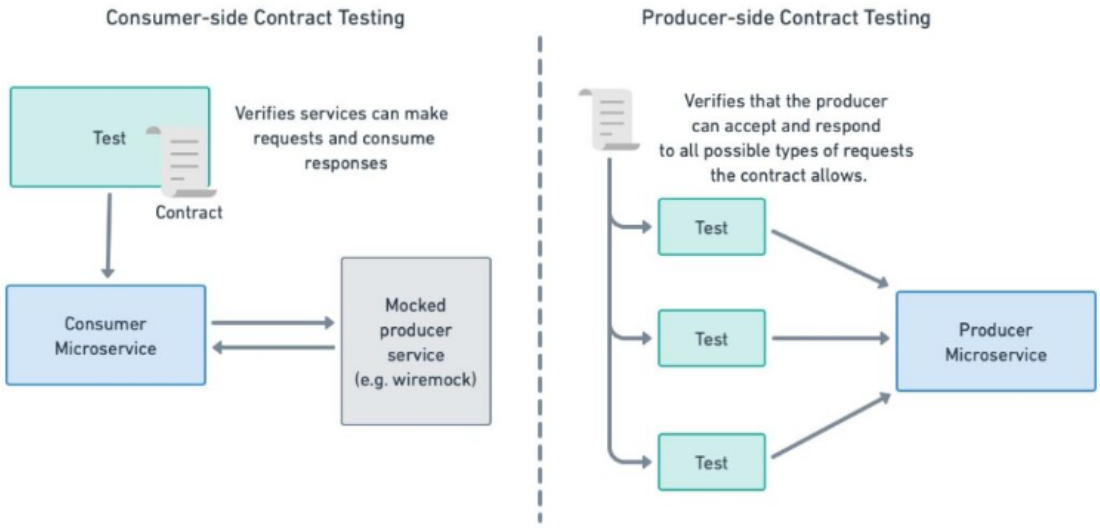
- Base testing
    - Unit tests, Integration tests, Component tests, Contract tests, Automated API testing, etc. for individual services
    - Automated API testing to ensure overall microservices work well together
    - End-to-end testing
- Load testing
    - The Pareto principle or 80/20 rule for performance testing [14]
        - 80% of the effects drive from 20% of the causes
    - Involves measuring the RED metrics and Golden signals by applying indeterministic amount of simulated system load.
- Resiliency testing
    - Increase the rate of faults by triggering them deliberately [4]
    - E.g. Chaos Monkey by Netflix to test Resiliency/Recoverability [13]

Testing strategies for Microservices [16][17][18]

- **Unit testing**
    - Solitary unit tests
        - Mocking, stubbing is used to deterministically test the service and to isolate it from external dependencies.

- Sociable unit tests
  - Sociable tests are allowed to call other services.
  - Are indeterministic in nature due to external dependencies but more reliable.

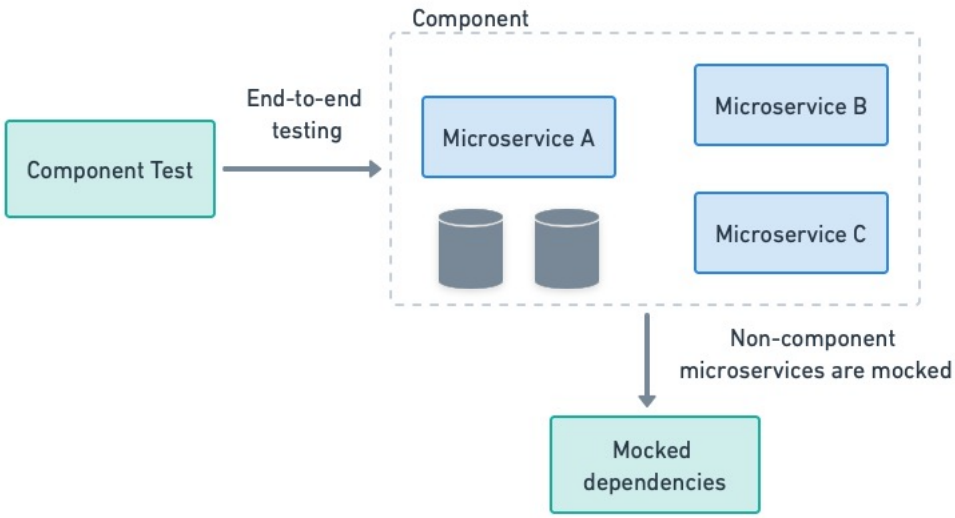
- **Contract testing**
  - They do not thoroughly test a service's behavior; they only ensure that the inputs and outputs have the expected characteristics and that the service performs within acceptable time and performance limits.
  - Contract tests should always run in continuous integration to detect incompatibilities before deployment.
  - Consumer side contract tests
    - The microservice connects to a fake or mocked version of the producer service to check if it can consume its API
  - Producer side contact tests
    - Emulates the various API requests clients can make, verifying that the producer matches the contract.



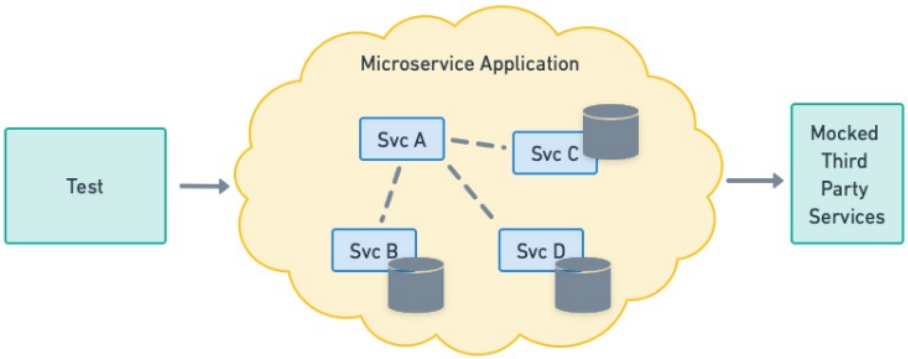
- **Integration testing**
  - Goal is to identify interface defects by making microservices interact.
  - Service business logic is not to be tested here. Their purpose is to make sure that the microservices can communicate with one another and their own databases.

- **Component testing**
  - A component is a microservice or set of microservices that accomplishes a role within the larger system.
  - Is a type of acceptance testing in which we examine the component's behavior in isolation by substituting services with simulated resources or mocking.
  - They are more thorough than integration tests because they test all happy and unhappy paths.

…Contd.



- **End-to-end testing**
  - Should cover all the microservices in the application using the same interfaces that users would–often with a combination of UI and API tests.
  - Test environment would include all the third-party services that the application usually needs, but sometimes, these can be mocked to cut costs or prevent abuse.



- **Feature flags**
  - A feature flag or feature toggle is a change or feature written inside conditional code. Developers can turn the feature on or off depending on their testing requirements while the application is running and in use.
  - Once integrated, a feature flag allows you to turn on a feature for a select group of users.

- **Traffic shadowing**
  - With traffic shadowing or mirroring, the router duplicates incoming traffic to an already-released service and gives the copy to another service. The request and response mechanism between the user and the existing service remains intact. On the other hand, the second service with a copy of the traffic contains new features

that require testing. Consequently, it does not interfere with the existing process. Instead, the copy is used to test its functionality.
    - There is no tangible impact on the existing services.
    - The new version's responses, which are not sent to users, can also be compared to those of the production service.
    - Traffic shadowing can be used with other deployment techniques like blue-green or canary deployments.

- **A/B Testing**
    - A/B testing, also known as split testing, refers to a randomized experimentation process wherein two or more versions of a service feature are shown to different segments of users at the same time to determine which version leaves the maximum impact and drives business metrics.
    - A/B testing is one of the components of the overarching process of Conversion Rate Optimization (CRO), using which you can gather both qualitative and quantitative user insights.
    - This test can be combined with blue-green or canary deployments as they handle the actual feature deployments that this strategy tests. After comparing the versions shown to the groups, the one that has performed better can be pushed to release for all users.

3. What steps would you follow to deploy a microservice?

The steps to deploy a microservice depend on the deployment strategy being used.

Types of Microservice deployment strategies [19][16]

- **Blue-Green deployment**
    - Uses 2 identical but distinct PROD environments.
        - Blue – Has new code but inactive
        - Green – Has old code and is active
    - Smoke tests are run in the Blue environment and when all tests pass, the load balancer is slowly/gradually allowed to redirect partial traffic from Green to Blue environment. If there are no issues, all of the traffic is redirected and the Green environment resources are decommissioned.
    - The old environment serves as a backup in case a rollback is needed.

- **Canary deployment**
    - Also protects against release related risks.
    - A canary deployment operates within the same microservice or infrastructure. The developers roll out a new service or application version with changes only to a fraction of the end-users.
    - The impact is temporary and minimal as the experiment is only run on a subset of users with most remaining unaffected. Once they are working and have passed verification, you can scale up the changes

- **Dark launching**
  - A dark launch is a technique that deploys updates to microservices catering to a small percentage of the user base. It does not affect the entire system. When we dark launch a new feature, we will initially hide it from most end users.
  - Feature toggles are a good way to release the service updates gradually and these can easily be turned on or off.
  - When the microservice has been tested and found to be suitable under realistic loads, it is activated to serve traffic from the entire production environment.

- **Staged release**
  - The staged release deployment strategy for microservices involves gradually releasing microservices to one environment at a time. For example, the development team first releases the microservices to the testing environment and later to production.

- **Rolling deployment**
  - Services that don't have to be constantly available can swap in the new code for the old one and then be restarted.

- **A/B testing** *(to determine which features to deploy)*
  - Explained above in detail, A/B testing helps make a determination of which features to deploy.

References

[1] Cloud Native Observability with AWS -https://www.youtube.com/watch?v=UW7aT25Mbng
[2] What is a Service Mesh - https://www.nginx.com/blog/what-is-a-service-mesh/
[3] APDEX score for measuring Service Mesh health - https://tetrate.io/blog/the-apdex-score-for-measuring-service-mesh-health
[4] Designing Data Intensive Applications - https://dataintensive.net/
[5] OWASP Microservice Security Threats - https://lalverma.medium.com/microservices-owasp-security-threats-eabcd836e08b
[6] Google SRE Error Budgets - https://cloud.google.com/blog/products/management-tools/sre-error-budgets-and-maintenance-windows
[7] The Art Of Service SLOs by Google - https://sre.google/resources/practices-and-processes/art-of-slos/
[8] Observability in AWS - https://aws.amazon.com/blogs/big-data/part-1-microservice-observability-with-amazon-opensearch-service-trace-and-log-correlation/
[9] Istio Observability - https://istio.io/latest/docs/concepts/observability/
[10] Service Mesh monitoring - https://sysdig.com/blog/monitor-istio/
[11] RED method - https://www.infoworld.com/article/3638693/the-red-method-a-new-strategy-for-monitoring-microservices.html
[12] Latency analysis for microservices - https://www.linkedin.com/pulse/latency-analysis-microservices-evrim-%C3%B6z%C3%A7elik/
[13] Netflix Chaos Monkey - https://netflix.github.io/chaosmonkey/

[14] When should I start load testing? - https://techbeacon.com/app-dev-testing/when-should-i-start-load-testing
[15] Canary deployments, A/B testing and microservices - https://blog.getambassador.io/canary-deployments-a-b-testing-and-microservices-with-ambassador-f104d0458736
[16] 5 testing strategies for deploying Microservices - https://devops.com/5-testing-strategies-for-deploying-microservices/
[17] Testing Microservices 12 useful techniques - https://www.infoq.com/articles/twelve-testing-techniques-microservices-intro/
[18] Testing strategies for microservices - https://semaphoreci.com/blog/test-microservices
[19] Microservice deployment pattern that improve availability - https://www.opslevel.com/blog/4-microservice-deployment-patterns-that-improve-availability

…Contd.