Performance Test Models and Test Framework 4+1 Architecture

Created by Qidi Wang, last modified on Apr 01, 2022 08:33

Document Status

Document Status	Created/Changed by	Created/Changed on
Draft	@Qidi Wang	
Review		
Final		

Overview

Establish a Technical performance test strategy that focuses on evaluating the application performance and the system's Capability. Providing Performance test framework, performance analysis, performance optimizations, capacity planning. Ensure the Box products in the production meet the expected performance, reliability, availability, and Scalability (PRAS) level. This Architecture document provides the solution of Performance test framework and Performance Test process in Box.

Logical View

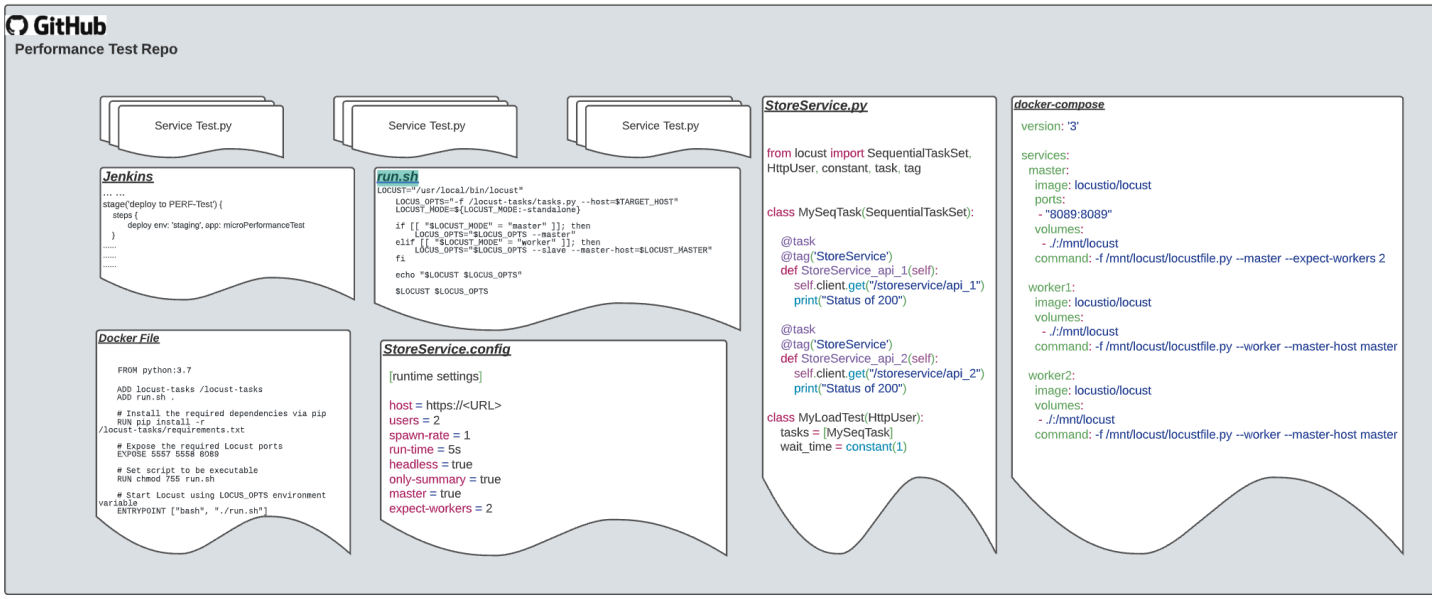
Performance test framework is responsible for providing enterprise level performance test solution to satisfy various teams to have continues performance test as part of code development phase that achieve "test early, test often and find issue early". The performance test framework that team is creating meets following mandatory requirements:

1. Easy to maintain the test script and test scenario
2. Test Early
3. Scaleable test infra & Automated test
4. Able to run different types of performance based on different requirement
5. Making Reporting of the process

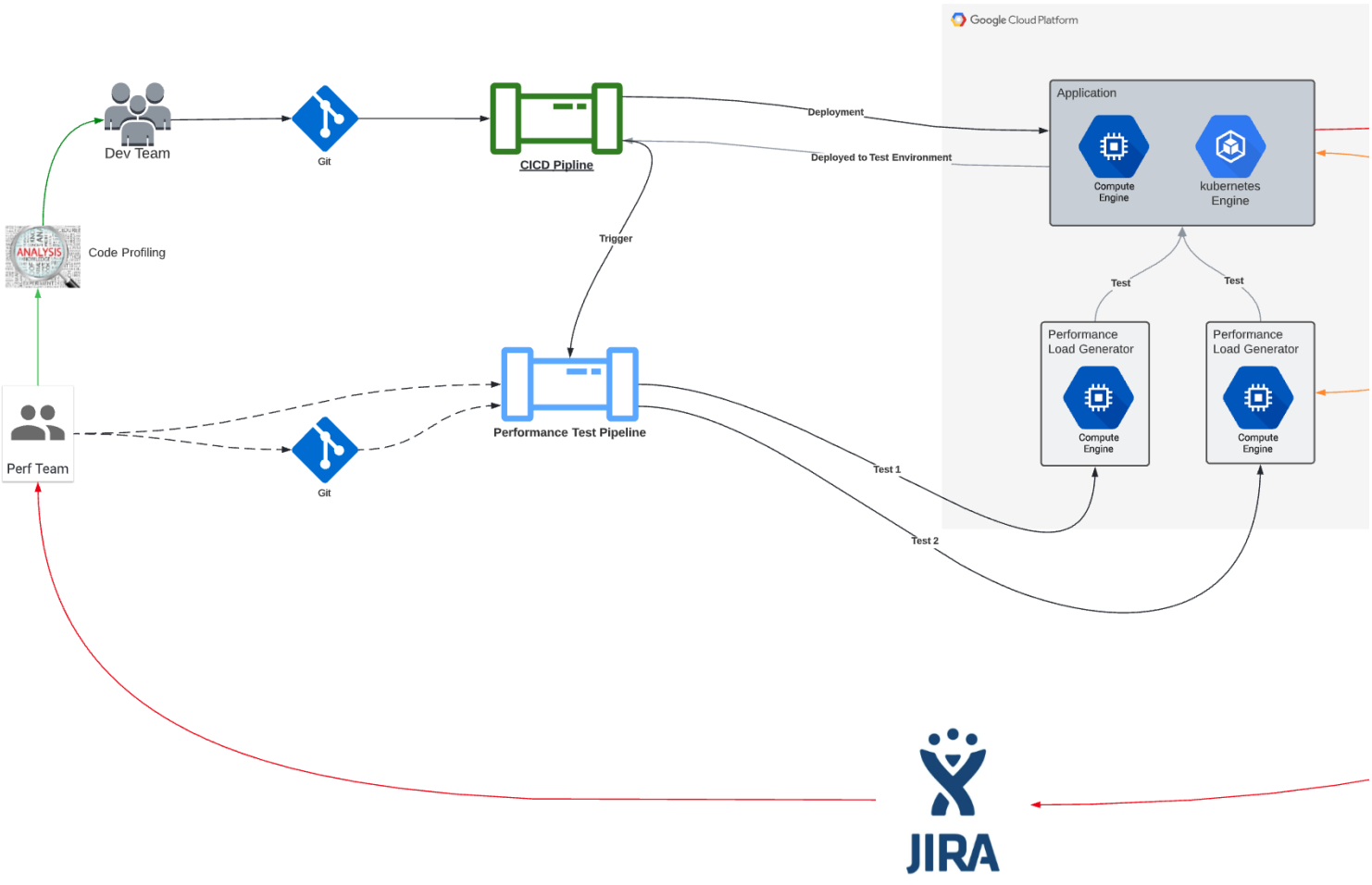
Development View

- Unit performance test and Integration performance test scripts run using Locust, an open-source tool written in python3. The reasons for using Locust for Unit and Integration performance test are:
  a. Test as a Code. Test scripts and test scenarios can be created and designed as python code that would be friendly for developers to maintain performance test scripts.
  b. Locust has a CLI. The command line with different arguments can execute the test for various requirements. Integration friendly with most of automation framework
  c. Distributed load testing
- Jmeter is preferred to support Regression level end-to-end load test and middleware/backend component tests(Kafka, SQL, MessageQ).
  a. Jmeter supports different protocols.
  b. Jmeter has the Capability to test Kafka, MySql DB directly.
  c. Recording function much easier for regression test script development and real-user scenario creation
- Question from Dev team: A few Dev teams have already started using Gatling for performance validation. Why can Gatling not be adopted?
  a. The biggest concern is the scalability of the Gatling tool to support distributed performance testing. Only Gatling Frontline (enterprise version) has the built-in feature of Clustering mode. Distributed performance testing is an essential requirement for the test framework, especially for any test requiring many connections.
  b. Gatling runs as a Java application that consumes more resources than Locust. Gatling itself will become the bottleneck of the performance test and report invalid response time that could cause misleading.
  c. Weak CLI capability.

	JMeter	Locust	Gatling
Operating System	Any	Any	Linux
Open Source	Yes	Yes	Yes
GUI mode	Yes, with non-GUI mode available	No	Yes
Support of “Test as Code”	Weak (Java)	Strong (Python)	Mid(scala)
In-built Protocols Support	HTTP, FTP, JDBC, SOAP, TCP, JMS	HTTP	HTTP
***CLI**	**No**	**Yes**	**No**
Recording Functionality	Yes	No	Yes
***Distributed Execution**	**Yes(but, complicated)**	**Yes**	**No (Yes for commercial version)**
Number of Concurrent Users	Thousands, under restrictions	Thousands	Thousands
Test Results Analyzing	Yes	Yes	Yes



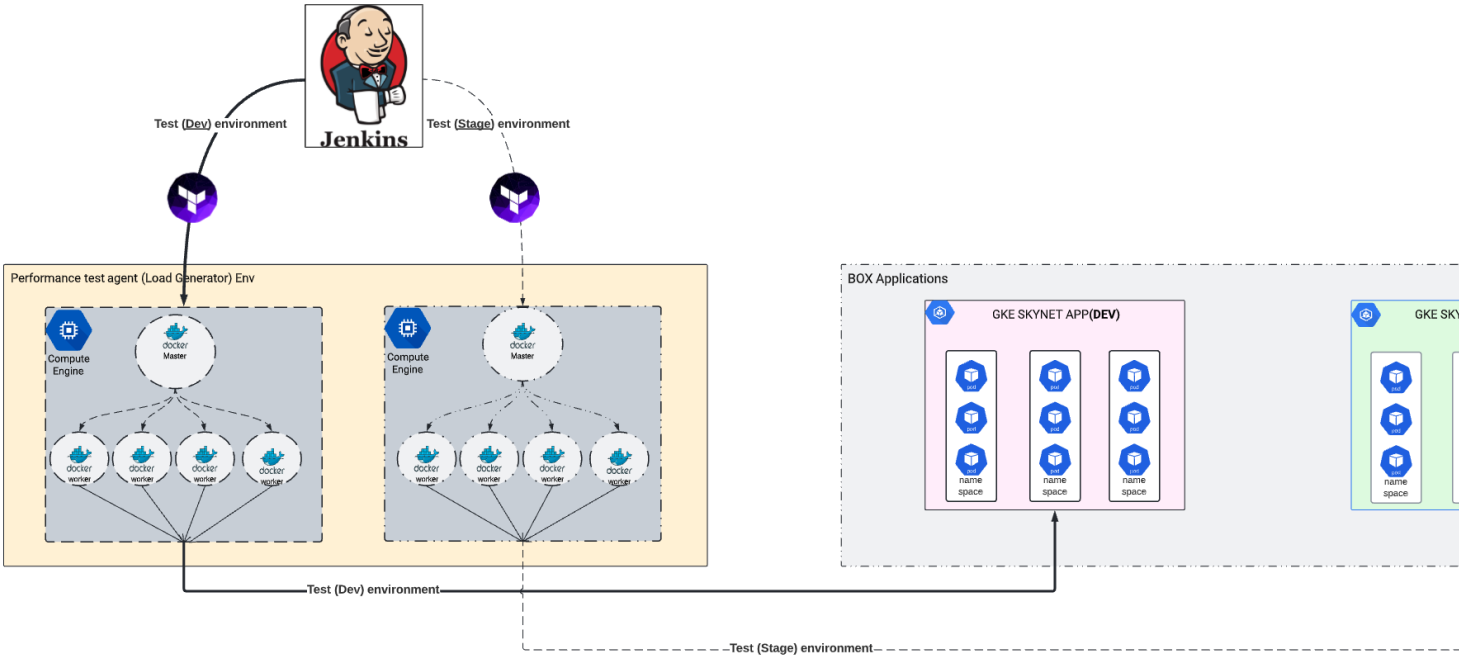
Process View:



Physical View:

- **PoC in Dev**
- Performance load generation instances should be set up in a dedicated env that allows the test requests sent to Dev, Stage, and PROD depending on performance test requirements. (considering to use PE env , or similar to PE env)
- Performance load generators should be able to scale up and down based on the concurrent number of test runs for different test purposes. Instances should be turned off after the test is completed to save the cost.

- Selecting Docker instead of using skynet Platform since Auto-scaling and Auto-Provisioning function is not available yet. In addition, The size of the GCE VM and number of docker container should be customized for different test scenarios. Docker is more flexible in this case.

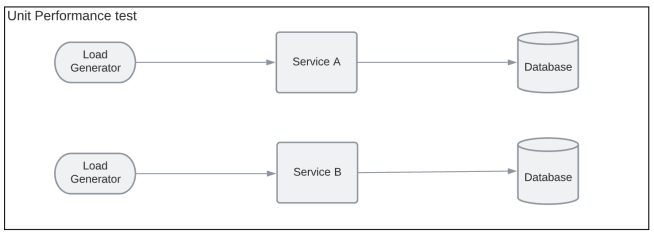


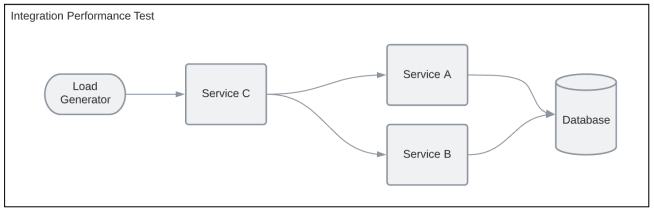
Scenarios:

Unit Performance Test & Integration Performance Test:

Continuous Performance Test(Regular release/code changes)

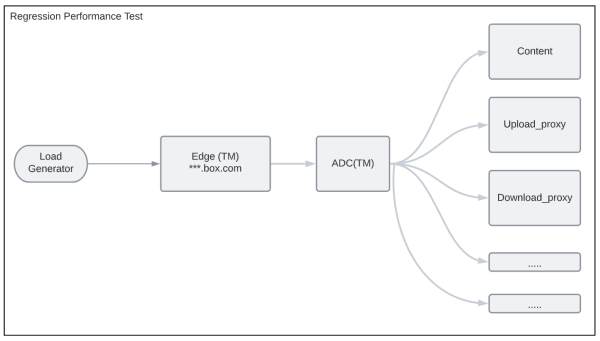
- To prevent performance issue occur post production deployment for any code change or product release, the Unit Performance Test and Integration Performance test should be part of the application release cycle in order to discover any performance issue and Potential risk before production deployment. The performance test should be auto-triggered when the new code deployed to the targeted Environment (Continuous Performance Test).
- Each Microservices performance test will bring up its own GCE VM, Turn off once the test is completed.
- Monitor the response time on the server end by using APM tools (SignalFx), Testing result can be monitored and stored in Wavefront.
- On the server side the SLI thresholds setup is required to determine the test is pass or fail (CPU, Memeory,TPS, 90 % Latency, Error rate),
- If one of the SLIs exceeds the threshold, the test will be marked as failed, generating a Jira ticket and assigned to a performance engineer for troubleshooting.
- Provide the performance tuning recommendations based on performance analysis, then cycle back to developers

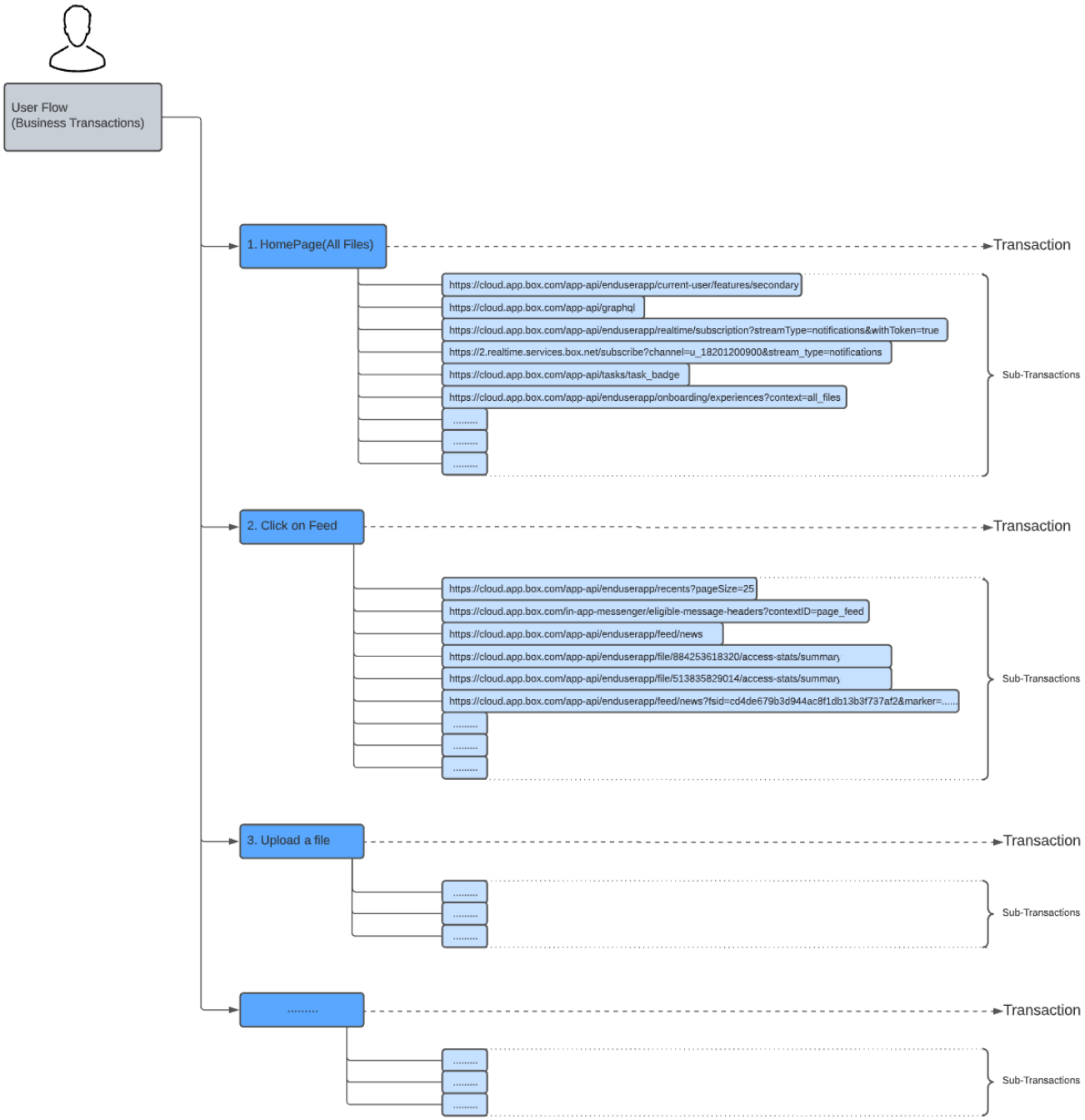




Regression performance test(End to End)

- The purposes of the Regression Performance test is not only to measure the response time of each page/business transaction sending request to Edge Traffic manager (Cloud.app.box.com) but also test how up stream applications leverage with down stream applications after the all code changes or new future announced (Thread pool, timeout, network latency, service mesh).
- Regression performance test also cover the Performance, Availability, Reliability and Scalability of the end to end system and System Capacity measurement/Estimation. Especially for the Centralized components, such as: API Gateway, Network bandwidth, Service mesh, Middleware(Kafka, MQ), Databases, Redis, Memcache etc.





***Capacity Planning ***

Capacity planning is one of the most critical components for the performance engineering team. Capacity planning is not only to estimate the size of the system is required to support the business growth (short term and Long term) but also to reduce or avoid resource-wasting. In addition, ensure the system is cost-efficient and scalable.

The tests described below are required to optimize the cost-efficiency and Capacity planning when team start work on the autoscaling.

1. Set the Threshold and the Capacity indicators:

- What is the Threshold for Capacity planning test?

- The threshold can be defined as a signal to stop the stress load test. It can be set as a service level indicator or infra level indicator. The test need to be terminated once one of the indicators meet the threshold,
    - Service level Indicator (Response time <= SLO , Error rate <= .5%)
- Infra level Indicator (CPU utilization <= 60%, IO <= 60%, Memory <= 70%)

- Understand the type of clusters to set the proper threshold and indicators:

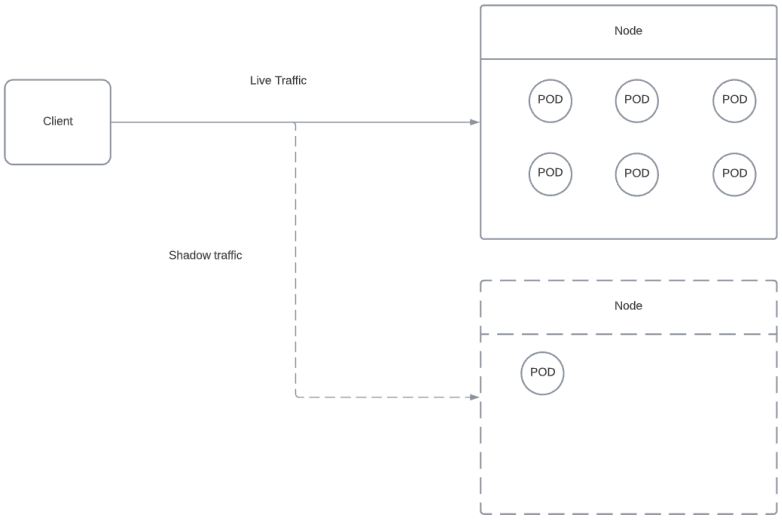
- For computer-intensive clusters, TPS(Transaction per second) is the indicator for measuring the Capability of the cluster. The threshold will focus more on the CPU utilization if the CPU is the initial bottleneck.
- For memory intensive Cluster, MB/S is the capacity indicator depending on the size of the data. The threshold will focus more on the data throughput and memory usage.

2. Analyze current production load and Estimate the expected load based on business growth

- The production load history will help analyze the periodical production load(Prefer to narrow it down to each service). For example: what are the Peak load and dip load during the day, week, and month. Understand what are factors affect the load.
- Note down the most recent peak and dip loads for the Autoscaling configuration.
- Estimate the short term and Long term load expectations for additional resource planning

3. Run application benchmark Tests(Single pod)

Benchmark tests help identify application performance bottlenecks in a single pod and appropriately tune the system and application configuration. The single pod benchmark is the cornerstone of capacity planning as most of the Capacity estimations depend on the benchmark data. Hence, better to have benchmark test run in PROD env with real traffic and actual data to get more accurate estimation data. **Istio Traffic Mirroring** is a powerful Capability that can shadow the traffic without impacting the critical path in production, validate the test results with the shadow traffic from the actual traffic. All the read operations can use the current consistent view of production data. The mutated data can go into a throw-away database whenever the test service makes a write operation. Here is the link for reference https://istio.io/latest/docs/tasks/traffic-management/mirroring/



The shadow traffic should be increased gradually until the throughput in the single pod dose does not increase any further. Then, Note down all the application behavior metrics and resource utilization metrics. Two scenarios:

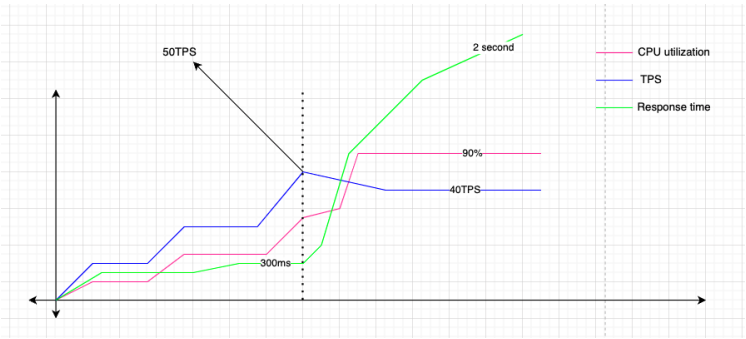
- If one of the hardware rescues is utilized to the maximum. Indicating the indicator which reached high utilization becomes a performance bottleneck
- If the throughput peaked even we keep increasing the shadow traffic, but utilization of the hardware is all within the threshold. Indicating the application tuning is required to use the available hardware resources better.

Example:

Service info:

- Test Service: "Gallery" service (Java Springboot Application)
- Peak throughput in Prod: 400 TPS
- Dip throughput in Prod: 200 TPS
- Pod size: 4 Cores, 4GB
- SLO: 500ms

The throughput stop increased any further at 40 TPS since response time kept increasing along with rising shadow traffic by turn up the weight on the Istio service mesh. The CPU started spiking up when the throughput reached 50 TPS.



Based on the snapshot above, The bottleneck of Gallery service in a single pod is 50TPS. However, considering all the threshold of CPU is 60 %. Therefore, the Maximum Capability of the Gallery service in the given size of the pod is 40TPS.

Hence, The resource estimation for Gallery services should be:

Max no. of pods = 2 * peak TPS / maximum capability of single pod = 800/40 = 20 (To measure the system is able to handle more unexpected traffic to make the system more scalable, set the maximum of the load as 2x peak load)

Min no. of pods = dip TPS / maximum capability of single pod = 200/40 = 5

The best practice for K8S resource allocation should be set as request = limit that guarantees to get. that will make it much easier for admin to manage the resources allocation and more efficiency. Hence, The total resources for this tenancy(namespace) should be :

Request the total number of CPU cores = max no. of pods * request.CPU = 80 Cores

Request total memory = Max no. of pods * request.memory = 80GB

NOTE: (A node pool may be shared by multiple tenancies. Resources should be divided proportionately based on the load volume, priority, and resources that ResourceQuota can handle.

4. Select the right worker note and tuning the resource assignment(Future scope):

"Total Request number of CPU cores" and "Request total memory" were calculated in the previous step will be the total capacity of the cluster.

There are two options to design this cluster:

Few Large nodes:

Pros:

- Lower costs per node
- Less management cost
- Run heave applications that require more resources

Cons:

- Kubelet and cAdviser overhead due to massive number of pods in Single node
- Low availability and reliability
- More Cost with Scaling up node

More Small Nodes

Pros:

- High availability and reliability
- More resilience system

Cons:

- Master node overhead due to a large number of nodes
- Kubelet and cAdviser overhead
- Ping pong effect may encounter with Autoscaling
- Pod limits on Small nodes

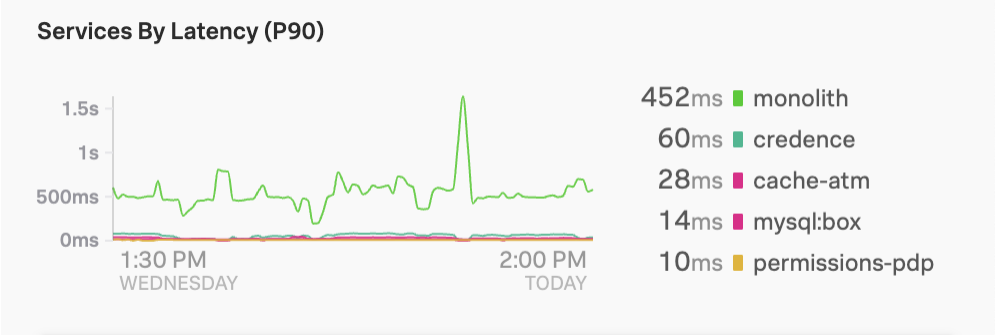
Hence, many factors affect which node types should be selected, especially when the Autoscaling cluster is enabled. Characteristics may change, including the type of the applications and the periodic load traffic.

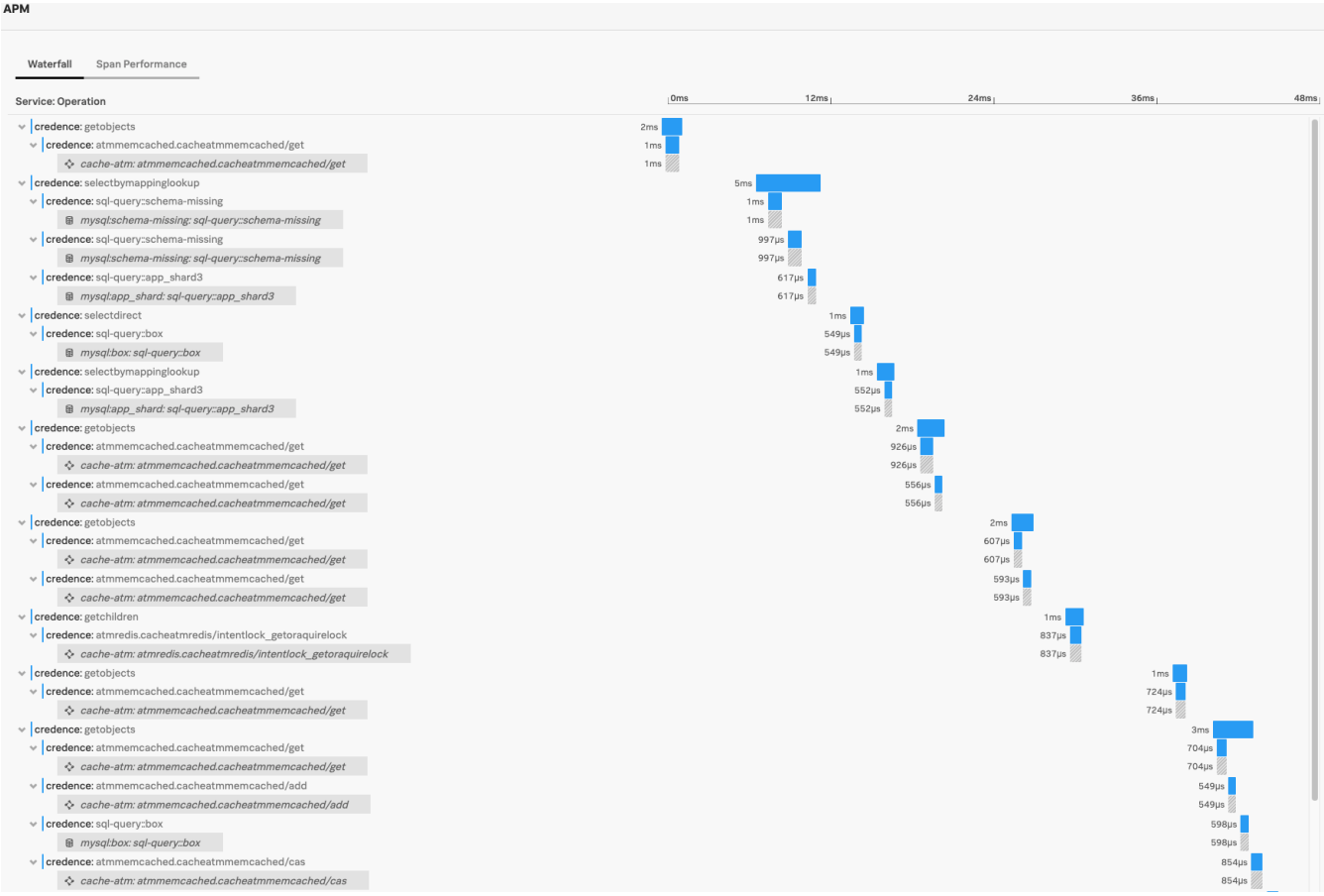
Performance issue analysis and debugging

Service response time tracing (APM tool)

Tool: Splunk SignalFx

Monitor overall services(APIs) response time behavior. Narrow down to the method which caused the slowness.





Code profiling :

Tools: Splunk SignalFx

Splunk SignalFx has "AlwaysOn Profiling" feature,

What you can do with AlwaysOn Profiling:

Here are some of the things you can do with AlwaysOn Profiling for Splunk APM:

- Perform continuous profiling of your applications. The profiler is always on once you enable it.
- Collect code performance context and link it to trace data.
- Analyze code bottlenecks that impact service performance.
- Identify inefficiencies that increase the need for scaling up cloud resources.

Here are some of the typical issues that AlwaysOn Profiling can identify:

- Slow or inefficient database queries
- Thread locks
- Thread pool starvation
- File system bottlenecks
- Slow calls to external services

[redacted]

No labels