Theoretical Tasks

1. **Scenario: The web app is not reachable; what would you do? Write a detailed action plan.**

Assumptions

Webserver: Apache / Nginx

hostname: myweb

Fully qualified domain name (FQDN): myweb.turbit.com

IP address: 192.168.56.10

DNS resolves correctly

The problem is not on the network

The first question to ask is: “Is the problem on my end or the remote end

i. Is the Remote Port Open?

Let us assume that we can route to the machine but can't access the web server on port 80; the next test is to see whether the port is even open. One way to do this is to try telnet:

telnet 192.168.56.10

if the message returns Unable to connect to remote host: Connection refused, then either the port is down (Apache isn’t running on the remote host or isn’t listening on port 80) or the firewall is blocking access. If telnet connects, we don’t have a networking problem.

Another option is to use *nmap* (if installed), which is able to detect firewalls

nmap -p 80 192.168.56.10

nmap will report if a port is closed or behind a firewall. If the port is actually closed, nmap reports it as closed, but if behind a firewall, *nmap* reports it as filtered.

ii. Test the remote host locally

At this point, we know that the problem could either be a network issue or that the problem is on the host itself. If we are satisfied that the problem is not on the network, then we can log in to the web server and test whether port 80 is listening. For this, we use the *netstat* command:

The fourth column of the response from the above command tells us the local address on which the host is listening, while the final column tells us which process has the port open. If we do not see the web server process, we need to restart it.

If the process is running and listening on port 80, we might want to check if some firewall is in place. For this, we use the iptables command to list all the firewall rules:

sudo /sbin/iptables -L

The output of this command tells us whether the default policy is set to ACCEPT or DROP.

Should there be a firewall blocking port 80 for instance, then we need to modify the firewall rules to allow port 80.

iii. Testing from the command line

Once we are sure that the web server is listening on the correct port, the next troubleshooting step is to confirm that the web server actually responds to requests. For this, we use a command line tool like curl and telnet:

curl http://www.turbit.com

The output of this command should be the default web page created by the web server. We can also get information about the HTTP status code for a request by using the command below:

curl -w "%{http_code}\n" http://www.turbit.com

The output of this command lists the status code of the request at the bottom. With this extra information, we can tell what class the status code returned belongs to, whether it is a success, a client, or a server error.

iv. Check web server logs/ web server statistics for any errors, exceptions, or abnormal behaviour.

Both Apache and Nginx store their logs in custom directories under /var/log/apache2 (or apache or httpd, depending on your distribution) and /var/log/nginx, respectively. These logs can give us information relating to a high-load problem.

v. Check for web server configuration problems using commands like:

sudo nginx -t

vi. Check for web server permission problems:

One more check is to ensure that all subprocesses that do the work of serving content have appropriate permissions, say for instance, each file that we would like to serve must be readable by the www-data or apache-user. For instance, we can read from the logs that Nginx attempted to open /var/www/nginx-default/index.html, but permission was denied. At this point, we could check out the permissions of that file and confirm that it wasn't readable by the www-data user in which Nginx runs as using the command below:

ps -ef | grep nginx

What we have to do here is then to fix the permission problems if they are not the required ones.

HIGH LEVEL CHECKS

If we're using a load balancer, we check its health status in order to ensure backends or targets are healthy.

It is important to also do an infrastructure health check using commands like *free -m, df -h*, to ensure the servers are not running out of CPU, memory, and disk space.

We ensure that container orchestration systems like Docker Swarm or Kubernetes are healthy in a containerised environment.

Finally, we want to engage with stakeholders or users about the issue and provide updates on resolution timelines.

2. **What would be your strategy to migrate from Docker Swarm to Kubernetes?**

vii. Assessment & Pre-planning (key considerations):

Evaluate the existing Docker Swarm infrastructure. Understand the services, networks, volumes, configurations, secrets, and other resources. Compare the existing setup with Kubernetes objects like Pods, Deployments, Services, ConfigMaps, Secrets, Persistent Volumes, etc. The idea here will be to run Kubernetes in parallel with Swarm, and then gradually migrate traffic. Then we finally retire swarm.

viii. Training:

Ensure that the team is familiar with Kubernetes concepts. This might involve training sessions or workshops.

ix. Setup the Kubernetes Cluster:

Depending on our choice, we could either set up a managed Kubernetes service (like GKE, EKS, AKS) or install Kubernetes manually on our infrastructure. An interesting approach would be to manage the desired state of the environment with git (GitOps approach using an Argo CD operator). Each app in the suite will have an independent Helm chart, and the deployments will be performed by the Argo CD operator.

x. Recreating Configurations:

Although not Docker-native, there are a lot of additional tools and integrations to make the migration from Docker Swarm to Kubernetes smooth. One major tool is kompose, which can translate Docker Compose files into Kubernetes manifests. Several key concepts can be mapped between Docker Swarm and Kubernetes.

xi. Deploy the app in a staging environment on Kubernetes.

Traffic control will be key here. The percentage of apps deployed in Docker Swarm vs Kubernetes is key to avoiding a “bad” migration. As the percentage is slowly ramped up over days and weeks, we can compare behaviour and performance between the two systems. If there is a major problem with the

Kubernetes side of things, sending all traffic to Swarm is easy. Adequate testing will be crucial in this situation.

xii. Data Migration:

If the application uses stateful services, plan for data migration. This can involve moving data from Docker volumes to Persistent Volumes in Kubernetes.

- **How would you monitor a large infrastructure to ensure it is healthy?**

i. Using a Service Mesh to Improve Observability and Management:

A service mesh (e.g. Istio) is an infrastructure layer that controls and observes service communication. It offers several capabilities, such as observability, security, resilience and traffic management. Istio can be deployed in Kubernetes in a separate Namespace. Istio also supports several integrations with several open-source projects, for instance:

- Kiali, which provides observability to the service mesh and can help us visualise what is going on in the mesh
- Tracing, which handles and visualises distributed tracing information.
- Prometheus, which performs data ingestion and storage for metrics.
- Grafana, which can be used to visualise performance metrics and data collected by Prometheus

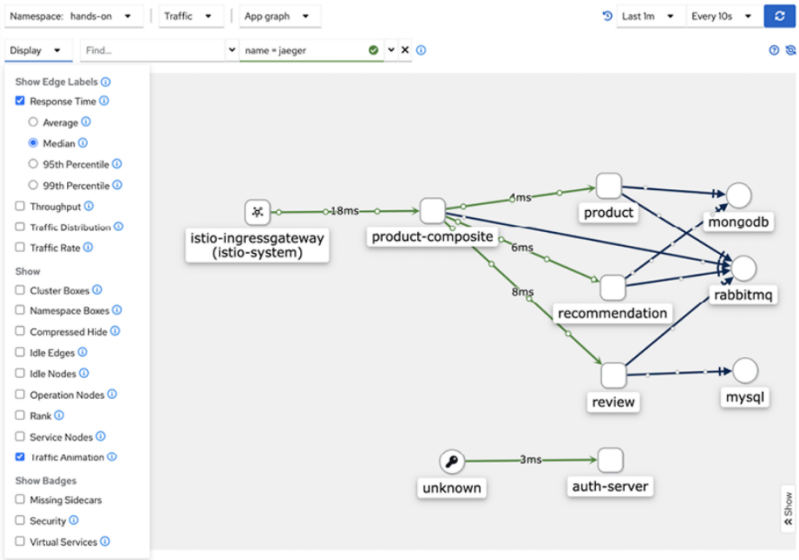

Figure 1. Kiali graph showing the hands-on Namespace (Microservices with Spring Boot – Magnus Larsson)

ii. Centralized Performance Monitoring:

Implement a robust monitoring stack using a layered approach, which will allow us to gather metrics from all our services. We shall then use those metrics to gather insight and predict system behaviour. Our monitoring stack will consist of metrics, logs and traces, each feeding into its dashboard to

aggregate data from multiple services. This allows you to set up automated alerts and look into all the collected data to investigate any issues or better understand system behaviour. Metrics will enable monitoring, whereas logs and traces will enable observability.

iii. Centralized Logging:

Implement a centralised logging solution like the EFK (Elasticsearch, Fluentd, Kibana). Elasticsearch is a distributed database with capabilities for searching and analysing large datasets. Fluentd is a data collector that can be used to collect log records. It will collect logs from containers that run in Kubernetes and send them to Elasticsearch after processing, while Kibana is a graphical frontend to Elasticsearch that can be used to visualise results and run analyses of the collected log records. The EFK stack will be deployed on Kubernetes. With this setup, we can perform root cause analysis using Kibana to determine the reason for an error in any of our deployed services.

iv. Alerting:

Once we are collecting metrics and storing them, we can set up alerts for when values deviate from what we consider normal for a given metric for instance, an increase in the time taken to process a given request or abnormal variation in a counter. We can then set up an alert to send a message whenever our threshold is exceeded. Multiple channels such as slack, email, webhooks or pagerduty can be used.

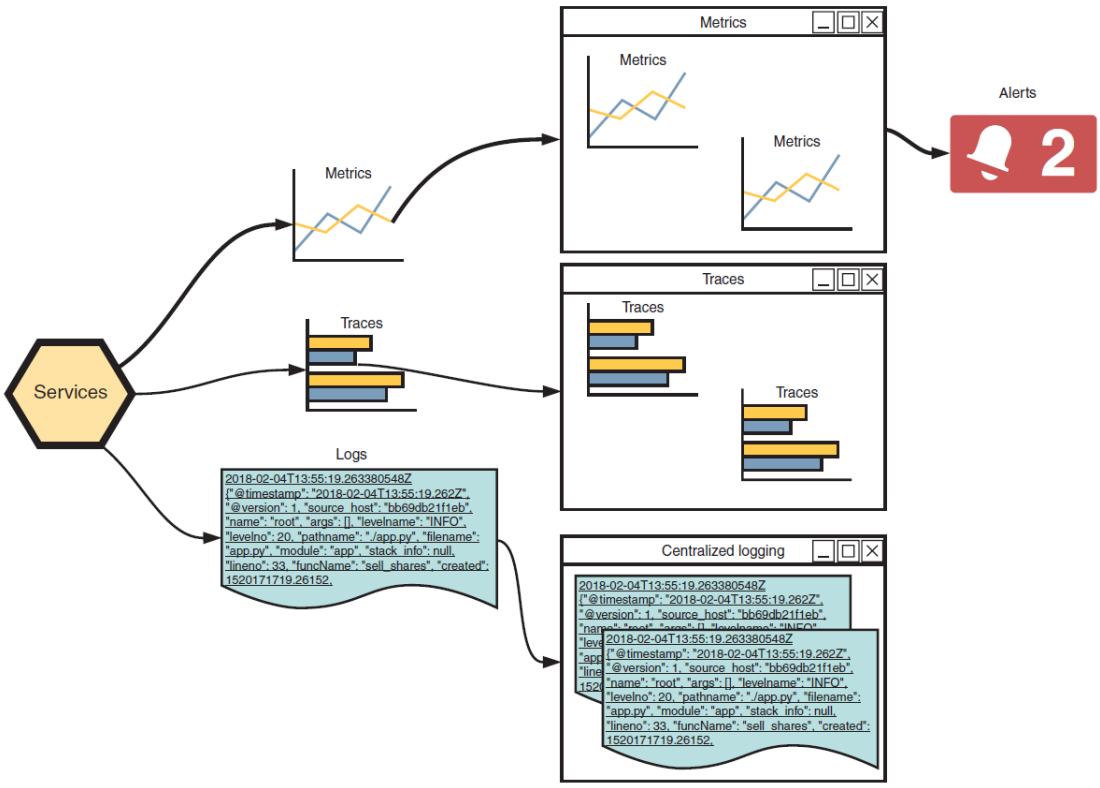

Figure 2. Components of a monitoring stack — metrics, traces, and logs — each aggregated in their own dashboards (Microservices in action – Morgan Bruce and Paulo Pereira)

v. Infrastructure / Configuration as Code (IaC):

Use infrastructure as code (IaC) tools like Terraform or configuration as code tools like Ansible. This ensures infrastructure consistency and reproducibility and can also help detect drifts from desired states. Use automated testing and continuous integration/continuous deployment (CI/CD) pipelines to catch issues early.

vi. Network Monitoring:

Monitor network traffic, bandwidth usage, and latency.