1. Learning Summary

* For this week, I learned to how to scale with k8s and understand how we can setup initial requests for both cpu and memory usage as well as setting out a limit to cap these for each of the generated pod. In addition, I learned when is the best time for horizontal scaling or vertically through their pros and cons.
* I am confident that these knowledge will great helpful toward my future career where I not just stop at being a fullstack developer but as well as a devops because I could understand the whole process from production to deployment stage.

2. Lab Activities

Rebuilt the original NodeJS Application and pushed it to localhost:5000

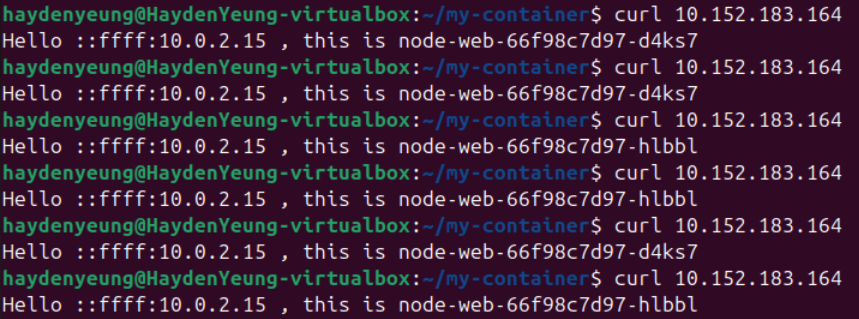


Created node-web.yaml and applied it

A screen shot of a computer

AI-generated content may be incorrect.

“Curl <Cluster-IP>” tests



Applied command “kubectl describe node haydenyeung-virtualbox”

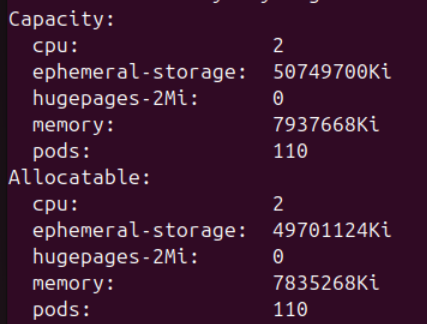
A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

It was found that those pods required 22% of CPU, and 3% of Memory



Pod Access & Checked on both CPU and Memory Usage

A screenshot of a computer program

AI-generated content may be incorrect.



Applied new changes to “node-web.yaml” and applied it

A screen shot of a computer program

AI-generated content may be incorrect.

Information of the new pod

A screenshot of a computer

AI-generated content may be incorrect.

Applied new changes to “node-web.yaml” – added memory: 10Mi

A screenshot of a computer screen

AI-generated content may be incorrect.

Used “kubectl describe pod …”

A computer screen shot of a program

AI-generated content may be incorrect.

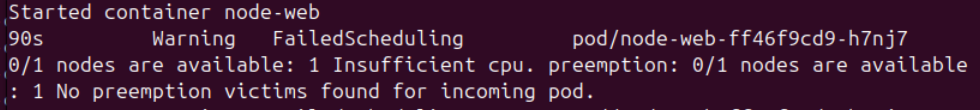
* It was found that this pod was kept on “CrashLoopBackOff’ because the instructed value for memory was too low.

Task 1 – Experiment with the resource requirements

A/ Both High: set CPU = 1000M, memory = 512Mi

A screenshot of a computer screen

AI-generated content may be incorrect.



* Due to high CPU requirement, one of the new pod remained in “pending” state due to insufficient CPU resource to handle another pod from running side-by-side with the first running one.

B/ Both Low: set CPU = 50M, memory = 5Mi

A screenshot of a computer program

AI-generated content may be incorrect.

* Because the CPU was set too low, thus pods cannot be created due to insufficient resource to “housing” them.

C/ Low Request, High Limit:

Requests:

* Cpu: 100m
* Memory: 10mi

Limits:

* Cpu: 500m
* Memory: 50mi

A screenshot of a computer screen

AI-generated content may be incorrect.

A computer screen shot of white text

AI-generated content may be incorrect.

Both pods were working well. QoS Class was found to be Burstable

Challeng Task

Re-edit “myapp.js”

* Create an array named memoryHog.
* For every request,1 string of 1MB of letter A (~1 million letters A) is added to this array.
* This array is not edit-able → memory keeping increasing with incoming requests.
* The console.log helps tell the current memory usage.

A screenshot of a computer program

AI-generated content may be incorrect.

Re-edit “node-web.yaml”

A computer screen shot of a program

AI-generated content may be incorrect.

Result:

A screen shot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

* It was found that upon reaching the CPU limit, curling became failed and after a short period of time, this pod this terminated and a new pod (with the same name) was created and took over the continuous process of “curling”.

Task 2 – Experiment with horizontal scaling

A black background with white text

AI-generated content may be incorrect.

Wrote autoscale according to instructions.

A screenshot of a computer

AI-generated content may be incorrect.

2 Replicas has been created, because, having additional replicas → it took longer time to requiring an additional replica.

3. Benefits of Scalable Resources for Applications

Scalable resources are critical for modern applications as they enable systems to adapt dynamically to varying workloads, ensuring optimal performance and cost efficiency. Scalability allows applications to handle increased user demand, such as during traffic spikes, by allocating additional resources, thereby maintaining responsiveness and preventing downtime (Buyya et al., 2018).

For example, in cloud environments like Kubernetes, scalable resources ensure that applications can access sufficient CPU and memory to meet demand without over-provisioning, which reduces costs. Additionally, scalability supports fault tolerance by redistributing workloads across resources when failures occur, enhancing reliability. This adaptability is particularly beneficial for applications with unpredictable usage patterns, such as e-commerce platforms during sales events, where scalable resources ensure seamless user experiences while optimizing operational expenses (Mell & Grance, 2011).

4. Vertical Scaling in Applications

Vertical scaling involves increasing the capacity of a single server by adding more resources, such as CPU, memory, or storage, to handle increased application demand. Applications implement vertical scaling by upgrading hardware or allocating additional virtual resources in cloud environments, allowing them to process more tasks without changing the application architecture (Buyya et al., 2018).

* Advantages include simplicity, as it requires minimal changes to application code, and lower latency, since all resources are co-located on a single machine. For example, a database application can benefit from vertical scaling by adding more memory to handle larger datasets.
* However, disadvantages include limited scalability, as there is a physical or virtual cap on how much a single server can be upgraded. Additionally, vertical scaling can lead to downtime during upgrades and creates a single point of failure, reducing fault tolerance (Mell & Grance, 2011).

This approach is less suitable for highly distributed applications requiring massive scalability.

5. Horizontal Scaling in Applications

Horizontal scaling involves adding more servers or instances to distribute an application’s workload across multiple nodes, commonly used in cloud-native environments like Kubernetes.

Applications achieve horizontal scaling by deploying additional containers or pods, managed by tools like the HorizontalPodAutoscaler, which adjusts the number of instances based on metrics like CPU utilization (Burns et al., 2019).

* Advantages include high scalability, as adding more nodes can handle virtually unlimited demand, and improved fault tolerance, as failures in one node do not disrupt the entire system. For instance, a web application can scale horizontally to handle traffic spikes during peak hours.
* However**,** disadvantages include increased complexity, as applications must be designed to handle distributed processing, often requiring load balancers and data consistency mechanisms. Additionally, horizontal scaling may introduce higher operational costs due to managing multiple instances (Buyya et al., 2018).

This approach is ideal for stateless applications but challenging for stateful systems.

References

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2019). Borg, Omega, and Kubernetes. *Communications of the ACM*, 62(5), 50-57. <https://doi.org/10.1145/3308560>

Buyya, R., Srirama, S. N., Casale, G., & Buyya, R. (2018). *Cloud computing: Principles and paradigms*. Wiley.  
Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6), 50. https://doi.org/10.6028/NIST.SP.800-145