1.Experiment Result Table (Result)

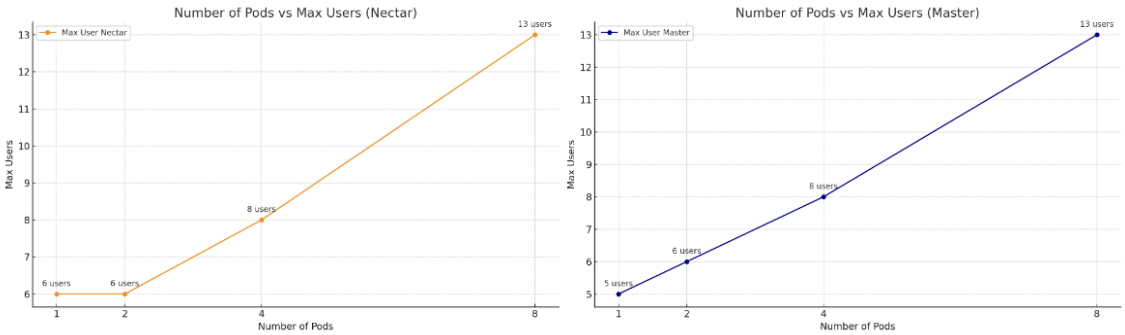
	Sending from Nectar instance		Sending from Master instance	
Number of Pods	**Max User**	**Avg.Response Time (ms)**	**Max User**	**Avg.Response Time (ms)**
1	6	3134.74	5	2862.66
2	6	1858.45	6	1778.42
4	8	1308.43	8	1273.25
8	13	1127.32	13	1109.23

2.Explanation of results and observations in experiment (Explain the result)

2.1 Methodology

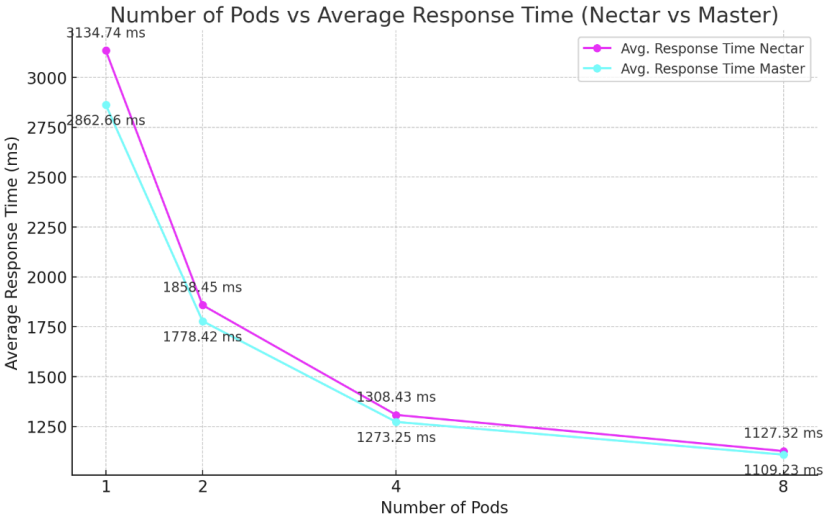
To determine the maximum number of users a server can successfully handle, a criterion has been established: if there are **no failures (0% failure rate) for 3 minutes** after reaching the maximum number of users, the server is considered capable of handling that specific user load successfully.

2.2 Observations



(figure 1: Max users for Nectar and Master when pods differs)

As shown in the figure 1, When utilising the Nectar instance, increasing the number of pods to 1, 2, 4, and 8 corresponded to a maximum user capacity of 6 users when 1 or 2 pods are deployed. The max user handle then experienced a modest increase to 8 users with 4 pods deployed and a substantial jump to 13 users when expanded to 8 pods. Conversely, with using Master instance, the maximum user capacity began at 5 users with a single pod deployment. It then increased to 6 users with 2 pods, to 8 users with 4 pods, and ultimately reached the capacity to handle 13 users upon scaling up to 8 pods. It can be concluded that there is a positive correlation between the number of deployed pods and the maximum user capacity, with increases in pod count leading to higher user accommodation.[1]



(figure 2: Number of pods / Response time for Nectar and Master)

From figure 2, we can observe that when the numbers of pods are increased, the average response time is decreased. For instance, when sending requests using nectar, when pods are increased from 1 to 8, the average response time decreased from 3134.7ms when using a

single pod to 1127.32ms when 8 pods are deployed. The average response time nearly decreased 3 times when the number of pods increased by 8 times. And this also happens in sending requests from master vm. Therefore we can conclude that there is a negative correlation between number of pods and average response time, with increase in pod count leading to a lower response time.[2]

In both request sending methods, the K8S cluster could handle a maximum of 13 users when operating with 8 pods, a maximum of 8 years with 4 pods and 6 users with 2 pods.

A notable difference is observed at the single pod configuration: requests sent through the Master instance accommodated a maximum of only 5 users, whereas the Nectar instance, when accessed via an external IP, was able to handle 6 users.[3]

From figure 2, the Nectar instance typically exhibits higher average response times compared to the Master instance. For instance, at a deployment of 8 pods with a maximum of 13 users, the average response time recorded for the Nectar instance is 1127.32 ms, marginally higher than the Master's 1109.23 ms. With 4 pods handling 8 users, the average response time for Nectar is 1308.43 ms compared to 1273.25 ms for the Master VM. A more pronounced difference is evident when 2 pods are deployed for 6 users; the Nectar's average response time is 1858.45 ms, which is noticeably higher by 80 ms than the Master's 1778.42 ms. Consequently, it can be deduced that the Nectar instance has a higher average response time than the Master VM when processing requests.[4]

2.3 Explanation and analysis of the observed result

Phenomenon [1] shows that deploying more pods results in an increased capacity to handle more users. This is generally true because the more pods there are, the greater the computing resources available. The Kubernetes cluster distributes requests to each pod via kube-proxy to achieve load balancing, thereby reducing the risk of pod crashes.

Before the experiment, it is expected to see a more significant increase in maximum users as the number of pods increased. For example, if one pod can handle 6 users, then theoretically, two pods should handle 12 users. However, the results did not reflect this expectation, as the maximum number of users remained the same when the pod count was increased from 1 to 2. Additionally, the maximum number of users only increased by 2—from 6 to 8—when the pod count rose from 2 to 4. There was a slight increase from 8 users to 13 users when the number of pods increased from 4 to 8.

The unexpected results could be due to inefficiencies in the load balancing managed by Kubernetes' kube-proxy. Although requests are distributed across pods, this distribution might not be perfectly efficient in achieving true balance. As a result, this could trigger a chain reaction where the failure of one pod affects the load balancing of other nodes, potentially causing the entire system to fail or leading to a higher failure rate.

Phenomenon [2] demonstrates that deploying additional pods leads to a reduction in average response time. This outcome occurs because when only a few pods are deployed, each one must handle a high volume of requests, resulting in substantial memory and CPU usage. Consequently, this heavy workload can slow down response times. By increasing the number of pods, the workload is distributed more evenly, which reduces the burden on individual pods and results in faster response times.

Phenomenon [3] illustrates that when only a single pod is deployed, the maximum user capacity differs between the Nectar and Master VM instances, with the Nectar instance able to handle one additional user compared to the Master VM. One possible reason for the lower user capacity could be that the locust script running on the VM consumes memory, leaving less available space for the container pod to use, thus limiting the number of users it can handle. Another potential reason could be limitations of the experiment itself; since it was only repeated five times, this might not have provided enough data to yield accurate results.

For phenomenon [4] shows that sending requests from nectar instances resulted in higher average response time compared to sending requests locally in the master instance. Because the Nectar instance sends requests through the network to the public IP of the master node, differences in geographical location between the two cloud instances—Nectar and Oracle—could result in increased network latency. Therefore, the transmission of requests over greater distances, potentially across different geographical regions, is likely to introduce more delays compared to the local transmission of requests using the Master VM.

3.Challenges in Distributed Systems

3.1 Failure handling

One of the challenges we encountered during this project was handling failures**.** For example, when there is a high volume of incoming requests, a pod is likely to crash due to overload and insufficient memory. Consequently, the requester may not receive the expected detection results because the server crashes.

To address these issues, Kubernetes is deployed. It enhances failure handling by providing redundancy through a built-in proxy that automatically reroutes incoming requests to other healthy pods when one crashes. Additionally, Kubernetes offers recovery capabilities through the use of a deployment.yml file, which can automatically restart failed pods to restore full operation and recreate pods if they are accidentally deleted. Furthermore, Flask provides appropriate HTTP status codes to the requester, ensuring that the server remains tolerant and does not crash entirely.

3.2 Concurrency

Another challenge we faced was managing concurrency when multiple requests attempted to access the same resource simultaneously. For example, the object detection model, which consumes a significant amount of memory, was initially declared as a local variable and was loaded over and over with each request. This setup limited the server to handling only one user's request at a time due to each pod having restricted resources (512 MB of RAM and 0.5 CPU). To address this, we modified the model to be a global variable, meaning it only loads once when the web service is initialised.

However, this adjustment led to an issue where all users accessed the model simultaneously on each thread, resulting in concurrency problems. The predictions were incorrect and mixed up due to these concurrency issues.

To resolve these issues, the system incorporates the Python threading lock library. This library provides a powerful and customizable lock that can be applied within functions to regulate access to code blocks. Specifically, we implemented a lock in the do_prediction function, ensuring that only one thread can use the model at a time to obtain prediction results. Other threads must wait for the currently active thread to finish. By doing so, the prediction results returned to being normal and accurate. Additionally, this approach optimised memory usage, as the model no longer needed to be loaded multiple times.

3.3 Openness

The system also addresses the challenge of openness, which refers to the system's ability to extend its software resources in the future. For instance, the requirements for this project might change to accommodate more users or to add more functionality to the web service.

To tackle this issue, the system utilises cloud instances that can be upgraded with additional CPU cores or RAM. Moreover, the use of Kubernetes allows for the expansion of the number of pods to accommodate more users. Additionally, the RESTful API ensures that the communication mechanism remains simple and straightforward, facilitating the future addition of resources to expand the functionalities of the web service.