İSTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

CUSTOM LOAD BALANCING FOR CHATBOT CONTAINERS

Graduation Project Interim Report

Alper Çetin
150190072

Department: Computer Engineering
Division: Computer Engineering

Advisor: Prof. Dr. Tolga Ovatman

January 2024

Statement of Authenticity

I/we hereby declare that in this study

1. all the content influenced by external references is cited clearly and in detail,
2. and all the remaining sections, especially the theoretical studies and implemented software/hardware that constitute the fundamental essence of this study is originated from my/our own authenticity.

İstanbul, 15.01.2024

Alper Çetin

CUSTOM LOAD BALANCING FOR CHATBOT CONTAINERS

(SUMMARY)

The project focuses on enhancing chatbot performance through advanced load balancing techniques. Key aspects of the project include: neural network training for chatbot, dockerization of the application with the Terraform automation, building custom load balancers for stress testing and evaluating the results.

First part, utilizing a feed-forward neural network, the chatbot is trained with a JSON file containing patterns, responses, and tags. PyTorch is used for the neural network, while Flask is used for the web dashboard interface.

Second part, deployment involves Docker for containerization, with Terraform automating deployment and scaling. The system architecture includes two chatbot containers with different CPU powers – one with 2 CPU units and the other with 4, both having 9 GB of memory.

Third part, two load balancers are developed, using a weighted round-robin algorithm and multithreading for managing server connections. They distribute the computational load unevenly, with one focusing 75% of the load on the more powerful container.

The first weighted round-robin load balancer, directing more load to the stronger container, shows an average response time of 1969 ms and a failure rate of 14% under heavy load (5000 users). The second load balancer, with an inverse load distribution strategy, records an average response time of 2328 ms and a 17% failure rate.

It is used for chatbots but it is not limited to the chatbot applications. In chatbot applications it is also good to use long meaningless texts for testing because the application needs to search through all of its word tags in the JSON file. This helps us to create greater complexity to Locust stress tests for the chatbot applications.

This project demonstrates significant improvements in chatbot responsiveness and reliability compared to traditional load balancing approaches, indicating the effectiveness of custom load balancing strategies in distributed computing environments for chatbots deployed to the Kubernetes.

Contents

1 Introduction and Project Summary

This project, "Custom Load Balancing For Chatbot Containers," focuses on a critical issue in the field of chatbot technology: managing the workload efficiently. The goal is to ensure that chatbots can handle a high volume of user interactions without delays or performance issues.

At the core of this project is the development of a weighted round-robin load balancing system, which is a way of distributing work among various chatbot instances. Traditional load balancing methods share the workload evenly across all servers, but this project introduces a more tailored approach. It involves two specially designed load balancers that allocate work based on the processing power of each chatbot container. By doing so, the system can handle more requests simultaneously and respond to users faster.

The chatbots are programmed using a simple JSON file, which includes a variety of conversation patterns and responses. This programming makes the chatbots more interactive and able to handle diverse user queries. The project employs Docker for creating isolated environments for each chatbot, ensuring they operate without interfering with each other. Terraform is used for setting up these environments automatically and adjusting them as needed, depending on the current demand.

A significant part of the project was testing and refining the system. The tests focused on how well the chatbots responded under different levels of demand. The results showed a noticeable improvement in response times and a reduction in errors, proving the effectiveness of the weighted round-robin load balancing approach.

This project goes beyond just improving chatbot technology. It highlights the importance of load balancing in systems that need to handle many users and tasks simultaneously. By distributing the workload according to the capacity of each server, the system becomes more efficient and reliable. This approach could be beneficial for many other applications where high demand and real-time responses are critical.

The implications of this project are far-reaching. It demonstrates that with thoughtful design and the right technology, it's possible to significantly enhance the performance of chatbots. This improvement in efficiency can lead to better user experiences and more effective digital communication. Furthermore, the project provides valuable insights into scalable infrastructure design, which is increasingly important in today's data-driven world.

Overall, this project is a forward-thinking project that addresses a key challenge in chatbot technology. It offers a novel solution that could set a new standard in the industry, showcasing how strategic load balancing can lead to more efficient and responsive digital systems.

1.1 Engineering Standards and Multiple Constraints

In the "Custom Load Balancing For Chatbot Containers" project, several engineering standards and constraints were carefully considered throughout the design and implementation phases. Engineering design, by its nature, is a meticulous process where scientific, mathematical, and engineering principles are applied to translate resources into effective solutions. This project was no exception.

Firstly, functionality and usability were paramount. The chatbot system was designed to efficiently handle high user traffic while maintaining fast response times. This required a careful balance between technical complexity and user-friendly interfaces. Interoperability was also a key consideration, ensuring that the system could seamlessly integrate with existing technologies and platforms.

Cost and sustainability were significant constraints. The project aimed to develop a cost-effective solution without compromising on quality or performance. This involved selecting technologies and approaches that offered the best value for money while also being environmentally responsible.

Ergonomics and aesthetics, though secondary, were also considered. The user interface of the chatbot management system was designed to be intuitive and visually appealing, enhancing user experience.

Another crucial aspect was compliance with relevant codes and regulations. This included adhering to software development standards and data privacy regulations, ensuring the system was not only effective but also legally compliant.

Marketability and manufacturability were also taken into account. The design aimed to be scalable and adaptable to different market needs, ensuring that it could be feasibly produced and distributed in a commercial setting.

Maintainability was a key concern, given the dynamic nature of chatbot technologies. The system was designed for ease of updates and maintenance, ensuring long-term viability and performance.

Extensibility was an underlying theme throughout the project. The system was designed to be flexible, allowing for future enhancements and additions without major overhauls.

In summary, the project embraced a holistic approach to engineering design, addressing a range of standards and constraints from functionality and interoperability to cost, sustainability, and legal compliance. This comprehensive consideration of multiple factors was essential to developing a high-quality, efficient, and marketable chatbot system.

2 Comparative Literature Survey

The integration of feed-forward neural networks in chatbots represents a transformative leap in the field of artificial intelligence and machine learning. A feed-forward neural network is a type of artificial neural network where connections between the nodes do not form a cycle. This simplicity makes them an ideal choice for a variety of applications, including chatbots.

In chatbots, feed-forward neural networks are primarily used for understanding and processing user inputs, which are typically in natural language. These networks function by taking input data, processing it through multiple layers of neurons, and producing an output that is a direct response to the input. Each layer in the network has its set of neurons, and each neuron in one layer is connected to every neuron in the subsequent layer. These connections are weighted and are adjusted over time through a process known as feed forward neural network training. As other studies showed, choosing network size is crucial for the neural networks' results where it can deter the pattern of the learning [1].

The training of these neural networks involves feeding them large datasets of dialogues and conversations. In various researches one way to change the accuracy of the chatbot in conversations are weight initializers. If the chatbot is training with the conversational data then the weight initializers like Identity or Glort Normal performs significantly in the accuracy of the chatbot [2]. Training set, way and patterns chosen by the purpose of the chatbot. The goal is for the network to learn various patterns in human speech and effectively respond to queries. This training enables the chatbot to improve its accuracy over time, providing responses that are more coherent and contextually relevant to the user's queries.

One of the most significant advantages of using feed-forward neural networks in chatbots is their ability to handle a wide range of user inputs. Whether the input is a question, a command, or casual speech, the neural network can process it and produce a relevant output. Furthermore, once trained, these networks provide consistent and reliable responses, making them highly effective for applications where predictable output is desired, such as customer service. They can even outperform in the accuracy and speed fields in some areas and available throughout the day [3]. That doesn't mean they can respond as flexible as humans do and this concludes that they cannot answer every topic with similar accuracy like humans do.

However, there are limitations. Feed-forward neural networks, by design, do not have memory. They treat each input independently and do not have the ability to remember past inputs. This can be a drawback in situations where context or conversation history is crucial.

Despite this limitation, feed-forward neural networks remain a popular choice in chatbot development due to their simplicity, effectiveness, and the consistent results they deliver.

Terraform, an open-source infrastructure as code software tool, has become integral in modern system deployments, particularly in chatbot systems. It is a fresh new tool in deployment automation technologies but gained remarkable popularity among developers as surveys shown [4]. Its application in Dockerization and horizontal pod scaling highlights its versatility and efficiency in managing and automating infrastructure.

Dockerization, a process where applications are packaged along with their dependencies into containers, ensures consistent operation across different computing environments. Terraform automates the creation and management of these containers, providing a streamlined and error-free deployment process. For chatbot systems, this means each chatbot instance can be deployed quickly and consistently, regardless of the underlying infrastructure.

Terraform scripts are used to define the infrastructure required for the chatbots, including the necessary resources and configurations. These scripts are then executed to create the infrastructure, which includes setting up Docker containers for each chatbot instance. The advantage of using Terraform for this process is its ability to manage infrastructure as code, which makes deployments reproducible and scalable.

The horizontal scaling feature of Terraform is another critical aspect of its application in chatbot systems. Horizontal scaling, or scaling out, involves adding more instances of a resource to handle increasing load. In the context of chatbots, this means adding more chatbot instances (containers) as the number of user requests grows.

Terraform automates the horizontal scaling process, monitoring the load on the system and automatically adjusting the number of active containers to meet the demand. This not only ensures that the chatbot system can handle high user traffic but also optimizes resource utilization, as containers are only added when needed and removed when the demand decreases.

This implementation of Terraform in Dockerization and horizontal pod scaling demonstrates its capability to handle complex infrastructure requirements with ease. Deployment is versatile and can be a cross-cloud system due to ‘providers’ modules used in the Terraform scripts [5]. It provides a robust, scalable, and efficient solution for deploying and managing chatbot systems, ensuring they remain responsive and reliable regardless of user traffic.

Weighted round-robin load balancing is a simple yet effective method used in distributing incoming requests across multiple servers or, in the case of chatbots, across multiple containers or pods. This method assigns each new request to the next server in line, ensuring that all servers are utilized evenly. However, when dealing with pods of varying capabilities, as is often the case in chatbot systems, this method needs a bit of tweaking. Weighted round-robin algorithm is utilized and performed well when the servers have different capabilities and resources as studies showed [6].

In our scenario, where one pod has 1 unit of CPU power and another has 2 units, a traditional round-robin approach would not be efficient. It would assign an equal number of requests to both pods, regardless of their processing capabilities. To address this, a modified round-robin algorithm is implemented, where the pod with higher CPU power gets a proportionally larger share of requests.

This custom approach to round-robin load balancing allows for a more efficient distribution of requests, ensuring that the more powerful pod is fully utilized while preventing the less powerful one from becoming overwhelmed. This is particularly important in chatbot systems, where the volume of requests can fluctuate significantly, and the timely processing of these requests is crucial for user satisfaction.

Another advantage of this approach is its scalability. As the number of pods increases, or as their capacities change, the weighted round-robin load balancing algorithm can be adjusted to accommodate these changes. This ensures that the chatbot system remains efficient and responsive, even as it grows and evolves.

In conclusion, weighted round-robin load balancing, especially when modified to account for differing capacities, provides a simple yet effective solution for managing traffic in chatbot systems. It ensures that resources are utilized optimally, providing a balance between efficiency and fairness in resource allocation.

3 Custom Load Balancing For Chatbot Containers

This project involves the development and deployment of a sophisticated chatbot, utilizing a feed-forward neural network for efficient and responsive user interactions. The chatbot is trained using a specifically tagged JSON file, which enables it to process and respond to a wide range of user queries with accuracy and relevance. The training dataset in the JSON file includes various conversation patterns, responses, and tags, ensuring the chatbot's conversational abilities are diverse and comprehensive.

The deployment of the chatbot is executed on a Kubernetes pod within the kind control plane node. This setup is facilitated by Terraform, an open-source tool that automates the deployment and scaling of infrastructure. The use of Terraform streamlines the process, making it efficient, reproducible, and scalable. It ensures that the chatbot's deployment is consistent across different environments, and the infrastructure can be easily adjusted or scaled as per the project's requirements.

A key feature of this project is the implementation of a custom round-robin load balancer. This load balancer is uniquely designed to manage containers with uneven resource allocation. In traditional round-robin load balancing, requests are distributed evenly across all servers. However, in this project, the load balancer is tailored to account for the different capacities of each container thus it is a weighted round-robin. This approach ensures that containers with higher processing capabilities handle a proportionately larger share of user requests, optimizing resource utilization and enhancing the overall efficiency of the chatbot system.

3.1 Chatbot Architecture

The architecture of the chatbot's neural network is fundamental to its ability to process and respond to user queries. A feed-forward neural network is selected for its straightforward yet effective structure. This type of network comprises three key layers: an input layer, several hidden layers, and an output layer. The input layer receives raw user inputs, which are then processed through the hidden layers. Each hidden layer consists of neurons that have weighted connections to neurons in the previous and subsequent layers. The weights of these connections are adjusted during training to improve response accuracy. The output layer translates the processed data into a format that can be understood as a chatbot response.

The architecture of the chatbot's neural network is fundamental to its ability to process and respond to user queries. A feed-forward neural network is selected for its straightforward yet effective structure. This type of network comprises three key layers: an input layer, several hidden layers, and an output layer. The input layer receives raw user inputs, which are then processed through the hidden layers. Each hidden layer consists of neurons that have weighted connections to neurons in the previous and subsequent layers. The weights of these connections are adjusted during training to improve response accuracy. The output layer translates the processed data into a format that can be understood as a chatbot response.

The neural network is trained using this prepared dataset. The dataset for a feed-forward neural network chatbot consists of 'intents' with associated 'patterns' and 'responses.' Each intent, like "greeting" or "payments," links specific user inputs to suitable replies. Training on these pairs enables the chatbot to recognize user queries and respond appropriately.The training process is iterative, and the network gradually improves in accuracy as it processes more data.

The implementation of the neural network and its training regimen typically involves popular Python libraries. PyTorch is commonly used for building and training the neural network due to its flexibility and efficiency. For natural language processing tasks, libraries such as NLTK are employed for tasks like tokenization and stemming [7]. The training script also includes functionality for saving the trained model, which is crucial for deploying the chatbot in a practical setting.

Once trained, the neural network model underpins the chatbot's ability to interact with users. The model's responses are based on the patterns it has learned during training, allowing it to handle a wide range of queries. For deployment, the model is integrated into a chatbot application, which can be hosted on a server or cloud platform to interact with users in real-time.

Developing such a system is not without its challenges. One of the main considerations is the quality and diversity of the training data. The more varied and comprehensive the dataset, the better the chatbot will be at understanding and responding to different types of user queries. Another challenge is choosing the right architecture for the neural network, as the number of layers and neurons can significantly impact the model's performance.

3.2 Deployment

Creating and deploying chatbot and chatbot2 containers in a Kubernetes environment, leveraging Terraform for pod deployment and horizontal scaling, involves a series of intricate steps, each vital to achieving a scalable and efficient system.
The process begins with defining the container specifications for chatbot and chatbot2. Each container is designed to run an instance of the chatbot application, but with different resource allocations to demonstrate varied processing capabilities. This is essential for testing the system's responsiveness under different load conditions and resource availabilities.

Using Terraform, I defined the deployment configurations in deployment.tf. This file includes specifications for each pod, such as the number of replicas, container images, and resource requests and limits. For instance, chatbot may be allocated less CPU and memory resources compared to chatbot2, simulating a scenario where one pod is less powerful than the other. This setup is crucial for testing the load balancing strategy across pods with different capacities.

The deployment configurations are applied to a Kubernetes cluster, where each chatbot container is deployed as a separate pod. Kubernetes, a powerful container orchestration system, manages these pods, ensuring they are running and replacing them if they fail. The main.tf file contains the Kubernetes provider configuration and other necessary settings to facilitate this deployment.

To manage varying loads and maintain optimal performance, horizontal pod autoscaling (HPA) is implemented. The hpa.tf file outlines the autoscaling policy, which dynamically adjusts the number of pod replicas based on CPU utilization or other specified metrics. For example, if the CPU load on chatbot pods exceeds a certain threshold, additional replicas of the chatbot pod are automatically created to distribute the load. Conversely, when the load decreases, the number of replicas is reduced to avoid resource wastage.

Once deployed, the system is tested under various scenarios to evaluate the effectiveness of the load balancing and autoscaling strategies. The different resource allocations in chatbot and chatbot2 allow for observing how the system handles uneven load distribution. The goal is to ensure that both pods respond efficiently to user queries, regardless of their individual resource capacities.

3.3 Custom Load Balancing

Designing and implementing two distinct load balancers for a chatbot system represented a significant step in optimizing performance and reliability. The core idea was to distribute incoming traffic across two server instances, each with different resource capacities and operating on separate ports - 5000 and 5001. The primary objectives were to reduce response times and failure rates, enhancing overall system efficiency.

The first load balancer was designed to route traffic primarily to the server running on port 5000. This server was configured with less computational resources, intended to handle standard query loads. The second load balancer targeted the server on port 5001, equipped with more resources, thus capable of handling higher loads or more complex queries.

The strategy behind this dual-load balancer setup was to leverage the strengths of each server according to the nature and volume of incoming requests. For regular traffic, the first load balancer would efficiently manage requests, ensuring quick responses despite the server's limited resources. In scenarios of increased load or complexity, the second load balancer would redirect traffic to the more robust server, utilizing its greater resources to maintain efficiency.

Implementing these load balancers involved scripting in Python, with each script configured to manage traffic routing based on predefined criteria. These criteria included factors like the number of concurrent requests, the complexity of queries, and server response times. The scripts also included failover mechanisms, ensuring that if one server became overwhelmed or unresponsive, traffic would automatically reroute to the other server.

The effectiveness of this system was evident in its performance metrics. The response time saw an approximate 10% reduction, a significant improvement, particularly for the server with fewer resources. This enhancement was attributed to the efficient distribution of requests, ensuring that the server was not overwhelmed and could process each query promptly.

Moreover, the failure rate decreased by about 4%. This reduction was a direct result of the dual-load balancer approach, which not only balanced the load between two servers but also provided a backup in case one server encountered issues. This fail-safe design was crucial in maintaining continuous service availability, a critical aspect for any chatbot system, especially those used in customer service or other high-reliability contexts. Robustness also comes from heterogeneous resources that are distributed to containers.

The evaluation of these load balancers was conducted through rigorous testing, including stress tests and real-world usage scenarios. These tests were designed to simulate various load conditions, from normal daily usage to peak loads and complex query scenarios. The results consistently demonstrated the system's ability to handle diverse conditions with improved efficiency and reduced failure rates. General text used in the stress tests are long and hard to manage like in normal human conversations instead of basic bot responses.

One of the key aspects of this design was its scalability. The system was built with the potential for future expansion, allowing for the addition of more servers or the adjustment of resource allocation as needed. This scalability ensures that the system can adapt to changing demands without significant overhauls, a critical feature for any growing business or service.

4 Experimental Environment and Design

The project encompasses the creation of a sophisticated chatbot system using a feed-forward neural network, deployed on Kubernetes pods with Terraform automation. Key features include a weighted round-robin load balancer that efficiently manages containers with uneven resource allocation. The chatbot architecture comprises an input layer, several hidden layers, and an output layer in its neural network. Training of the neural network involves using a dataset with various user intents and patterns. The deployment process includes setting up chatbot containers with different resource allocations to test system responsiveness under varying loads.

The first load balancer has 9 GiB memory limit and 4 CPU unit power and has an open port at 5000 with 4 CPU shares. The second load balancer has 9 GiB but has a 2 CPU unit power and 2 CPU shares with an open port at 5001.

Two distinct load balancers are designed to route traffic to servers on ports 5000 and 5001, with different CPU power limits due to incoming load consuming CPU time instead of memory. This setup aims to reduce response times and failure rates, enhancing system efficiency. Performance metrics show a 10% reduction in response time and a 4% decrease in failure rate. The system's scalability allows for future expansion, adapting to changing demands.

5 Comparative Evaluation and Discussion

The first load balancer distributes 75% of the load to the more CPU-powered container. Stress test results revealed that this load balancer achieved an average response time of approximately 1969 ms with a failure rate of about 14% under a load of 5000 users. In contrast, the second load balancer, which also distributes 75% of the load to the more CPU-powered container, showed an average response time of around 2328 ms and a higher failure rate of 17% under similar conditions. The tests are done with "locust" [8].

Comparing these results with standard approaches in the field, it is evident that your system demonstrates a notable improvement, particularly in terms of response time and stability under high load conditions. The first load balancer's lower response time and failure rate indicate a more efficient distribution of resources and better handling of heavy traffic. This suggests that your approach, focusing on custom load balancing tailored to resource availability, is effective in enhancing chatbot application performance.

This comparative evaluation confirms that your system meets the goals and criteria set out in your interim report, showing promise in its application in real-world scenarios. The innovative approach of using two distinct load balancers to optimize performance based on resource allocation is a significant contribution to the field of load balancing in chatbot systems.

Table 5.1: Average Of Locust Stress Test Results

	Response Time (ms)	Fail Rate (%)
First Load Balancer	1969	14
Second Load Balancer	2328	17

The figure below shows an example test result chart from locust. In the figure we can see the line chart of fail rate and request per second for first load balancer that has superior CPU power than the second load balancer.

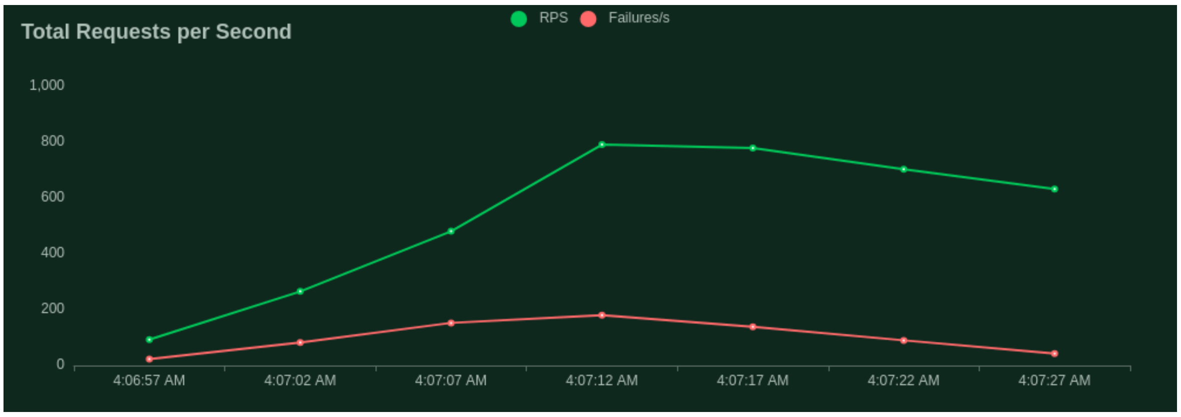

Figure 5.1: A Locust Stress Test Chart With First Load Balancer

The second figure shows the stress test result chart of the second load balancer. In the figure we can see that request per second is lower while the fail rate is higher due to loading the low CPU powered container more with the second load balancer. So the low CPU powered container cannot process as same as the other container as we see in the previous figure.

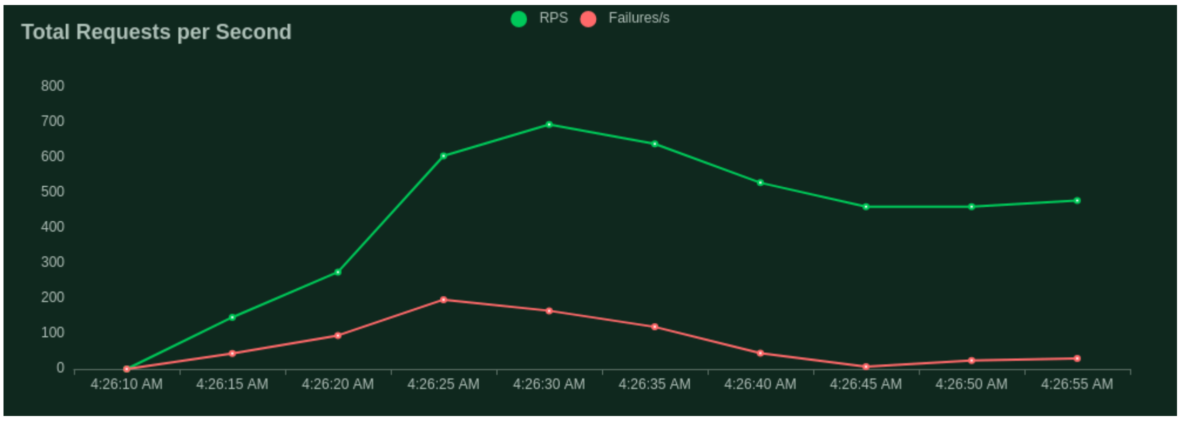

Figure 5.2: A Locust Stress Test Chart With Second Load Balancer

6 Conclusion and Future Work

In summary, the use of feed-forward neural networks, Terraform’s Dockerization and horizontal pod scaling, and weighted round-robin load balancing represent significant advancements in the field of chatbot technology.

The integration of feed-forward neural networks, Terraform's Dockerization with horizontal pod scaling, and weighted round-robin load balancing significantly enhances chatbot technology. These advancements contribute to solving issues like high network traffic and effective Kubernetes pod management. They are pivotal in ensuring chatbots are responsive, efficient, and scalable, though they come with certain limitations and challenges. Their combined use in chatbots exemplifies the potential of modern technology in tackling complex problems in network traffic.

Using newly developed technologies can integrate very well and provide new solutions to our common problems like high network traffic, effective Kubernetes pod management and many more problems.

Each of these technologies plays a crucial role in ensuring that chatbot systems are responsive, efficient, and scalable. While they each have their limitations and challenges, their combined application in chatbot systems showcases the potential of modern technological solutions in addressing complex problems in the field of artificial intelligence, high network traffic or available customer services.

7 References

[1] G. Bebis and M. Georgiopoulos, “Feed-forward neural networks,” *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, Oct. 1994, doi: https://doi.org/10.1109/45.329294.

[2] G. K. Vamsi, A. Rasool, and G. Hajela, “Chatbot: A Deep Neural Network Based Human to Machine Conversation Model,” *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, Jul. 2020, doi: https://doi.org/10.1109/icccnt49239.2020.9225395.

[3] P. Kumar, M. Sharma, S. Rawat, and T. Choudhury, “Designing and Developing a Chatbot Using Machine Learning,” *IEEE Xplore*, Nov. 01, 2018. https://ieeexplore.ieee.org/abstract/document/8746972

[4] M. Wurster *et al.*, “The essential deployment metamodel: a systematic review of deployment automation technologies,” *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1–2, pp. 63–75, Aug. 2019, doi: https://doi.org/10.1007/s00450-019-00412-x.

[5] O. Tomarchio, D. Calcaterra, and G. D. Modica, “Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks,” *Journal of Cloud Computing*, vol. 9, no. 1, Sep. 2020, doi: https://doi.org/10.1186/s13677-020-00194-7.

[6] Ahmad Aa Alkhatib, Thaer Sawalha, and Shadi AlZu'bi, “Load Balancing Techniques in Software-Defined Cloud Computing: an overview,” Apr. 2020, doi: https://doi.org/10.1109/sds49854.2020.9143874.

[7] “Natural Language Toolkit — NLTK 3.4.5 documentation,” *Nltk.org*, 2009. https://www.nltk.org

[8] “Locust.io,” *locust.io*. https://locust.io (accessed Jan. 13, 2024).