CS436 Project

GCP Multi-Architecture Deployment for Tinode

Eren Çavuş
Ayhan Salih Öner
Gürsel Yiğit Pekgöz
Yarkın Akyosun

Github: https://github.com/onerayhan/cs436_project

Video: https://drive.google.com/file/d/1Ona6kbR2zL5qYEp_6rjE3SgQ0f8QFegu/view?usp=sharing

Introduction & Application Chosen

For our project, we successfully deployed the Tinode chat application on the Google Cloud Platform. Tinode is an open-source chat application that provides a comprehensive messaging solution with various features. It supports individual and group messaging, web calls, data exporting, location sharing, and more. The application is designed to be flexible and scalable, capable of running in different configurations to meet diverse needs. Our project journey started with a Docker Compose-based single VM deployment to understand the basic setup and functionality of Tinode. We then transitioned to a more robust and scalable GKE deployment. By leveraging GKE and its features like HPA, we enhanced the scalability, reliability, and overall quality of our Tinode chat application. In our project, we also utilized Terraform for infrastructure as code (IaC) to automate the setup of our Google Kubernetes Engine (GKE) cluster.

Technologies Used:

1. **Google Cloud Platform (GCP)**: Our cloud infrastructure provider, offering scalable and reliable cloud services.
2. **Docker**: Used for containerizing the Tinode application, ensuring that all dependencies and configurations are encapsulated within Docker images.
3. **Docker Compose**: Utilized for managing multi-container applications in a single VM setup.
4. **Google Kubernetes Engine (GKE)**: A managed Kubernetes service that allows us to deploy, manage, and scale containerized applications using Kubernetes.
5. **Horizontal Pod Autoscaler (HPA)**: A feature in Kubernetes that automatically adjusts the number of pods in a deployment based on observed CPU/memory usage or other select metrics.
6. **Terraform:** Automated GKE cluster setup with IaC. Efficiently managed cloud infrastructure for consistent, reproducible deployments, minimizing human error.
7. **Locust**: An open-source load testing tool used to simulate high traffic conditions and evaluate the performance and scalability of the application.

Deployment Configurations:

1. **Single VM Instance**: In this setup, the entire Tinode application, along with its dependencies, runs within a single virtual machine. Docker and Docker Compose are used to manage the application components, including the Tinode server and the MySQL database. This configuration is straightforward and easy to set up but lacks scalability and fault tolerance.

2. **3-Node Cluster Setup**: This setup distributes the application components across three nodes, providing improved performance and reliability. Each node can handle part of the load, ensuring that the application remains operational even if one of the nodes fails. This setup requires manual configuration and management but offers better resilience compared to a single VM instance.
3. **Google Kubernetes Engine (GKE) Managed Environment**: For our project, we chose to deploy Tinode on GKE. This managed environment offers advanced features such as automatic scaling, integrated monitoring, and logging. Using Kubernetes manifests (YAML files), we defined the desired state of our application components, including deployments, services, ConfigMaps, Secrets, and Persistent Volume Claims (PVCs). GKE manages the underlying infrastructure, allowing us to focus on the application itself.

Throughout our project, we gained hands-on experience with a variety of powerful tools and platforms essential for modern cloud-native development. We began by deploying applications using Docker, which allowed us to run virtual machines (VMs) from Docker images. This approach provided a straightforward way to containerize our applications, encapsulating all necessary dependencies and configurations to ensure consistency across different environments.

Transitioning from single VM deployments, we explored the scalability and management benefits of Kubernetes, particularly through Google Kubernetes Engine (GKE). Kubernetes enabled us to orchestrate our containerized applications, manage their lifecycles, and scale them efficiently using features like Horizontal Pod Autoscaling (HPA). We learned to define our application components using Kubernetes manifests, which specified deployments, services, ConfigMaps, Secrets, and Persistent Volume Claims (PVCs). This modular and declarative approach facilitated a robust, scalable, and resilient application deployment strategy.

To automate the provisioning and management of our infrastructure, we leveraged Terraform, an Infrastructure as Code (IaC) tool. Terraform allowed us to define our GKE cluster and associated resources in a version-controlled configuration file. We created and managed a GCP service account with the necessary permissions to deploy these resources. By using Terraform, we could easily replicate and manage our infrastructure, ensuring consistency and reducing the potential for human error.

Furthermore, we integrated monitoring and logging solutions to maintain visibility into our application's performance and health. Using tools like Google Cloud's operations suite, we implemented comprehensive monitoring and logging for our Kubernetes cluster, which was crucial for early detection of issues and effective troubleshooting.

Overall, this project provided us with a deep understanding of deploying, managing, and scaling applications in the cloud using Docker, Kubernetes, and Terraform. These experiences have

equipped us with the skills needed to build and maintain cloud-native applications, ensuring they are scalable, resilient, and efficient.

Docker-compose based deployment on single VM

In this deployment configuration, the Tinode chat application is orchestrated in a single virtual machine (VM) using Docker Compose. The setup includes multiple Docker containers, each serving a distinct role in the application architecture. The MySQL database container hosts the database required for Tinode, ensuring data persistence such as messages and user data. Tinode servers are deployed as separate containers, with each instance representing a distinct node in the chat application cluster. These servers facilitate messaging functionalities, group messaging, and other features provided by Tinode. The containers are interconnected within the Docker network adapter, allowing communication between them. Tinode servers communicate with the MySQL database using the MySQL container's hostname and port, ensuring data retrieval and storage operations. Additionally, there are also exporters for monitoring the Tinode application deployed alongside Tinode server instances. These exporters gather metrics and they can push operational monitoring data to external systems for monitoring and analysis, but we haven't utilized this functionality in our project. The “compose.yml” configuration defines the configurations for each container, ensuring they are deployed properly inside the VM environment. Overall, this deployment model offers a basic and manageable solution for hosting the Tinode chat application within a single VM instance. However, it is not a very realistic deployment model since it only contains a constant number of worker nodes and cannot scale with incoming traffic volume. For this reason, after gaining initial experience with compose and vm, we have switched to GKE.

GKE deployment

After initial tests with docker-compose based single VM deployment, we deployed the Tinode chat application on Google Kubernetes Engine (GKE). We designed an GKE based architecture for our application that would orchestrate components of the original project to achieve scalable deployment. Main component in our application that requires scaling is tinode-chat containers, since their processing need is directly proportional to incoming messaging request volume. We use horizontal pod autoscaler on ReplicaSet in Tinode service to achieve scaling.

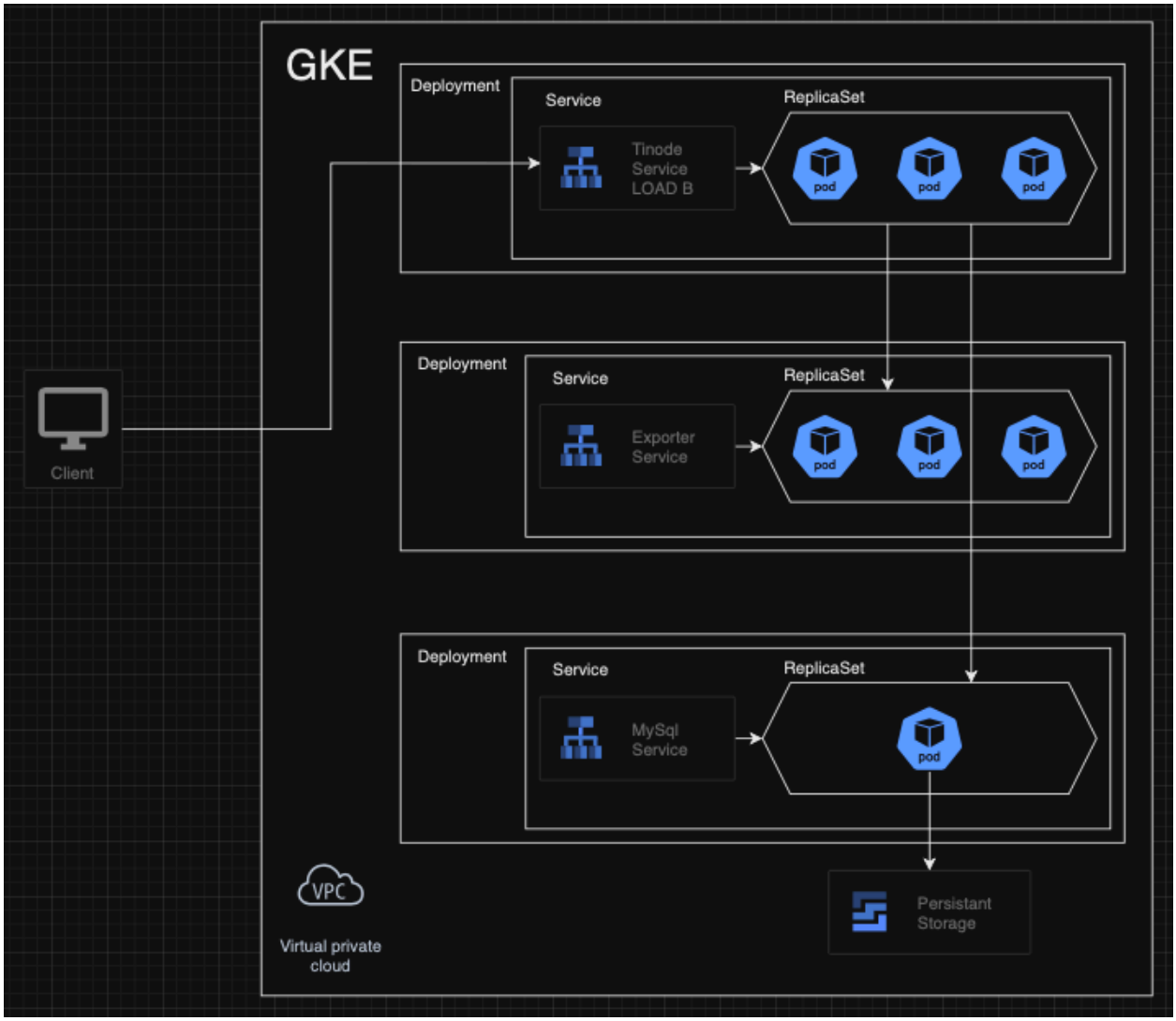

Figure [1]: GKE deployment architecture of Tinode

The deployment process began with the transfer of Kubernetes files into the Cloud Shell environment. These files, located in the kubernetes_files directory, contained essential configurations for various components of the application, including MySQL, Tinode chat servers, exporters for monitoring, and the Horizontal Pod Autoscaler (HPA). Execution of the deployment involved several steps. Firstly, the Compute Zone was set using the command gcloud config set compute/zone <zone>. Subsequently, a persistent disk was created to store MySQL data holding messages and other user data. The creation of the GKE cluster followed, with the cluster named tinode-cluster established to host the application components. Credentials for cluster access were obtained, and Kubernetes manifests for each component were applied using kubectl apply -f.

To verify the successful deployment, thorough checks were conducted on various aspects of the GKE cluster from the cloud terminal. This included verifying the status of cluster nodes, individual pods, and essential services using commands such as kubectl get nodes, kubectl get pods, and kubectl get svc. Once the deployment was verified, the external IP of the Load Balancer service was retrieved using kubectl get services. This IP served as the entry point for accessing the deployed Tinode chat application, allowing users to engage with the platform from our own browsers to test it by manually sending messages between team members to ensure intended functionality of the application. Efforts were made to ensure operational efficiency and cost-effectiveness by cleaning up unnecessary resources post-deployment. This involved deleting the GKE cluster using gcloud container clusters delete tinode-cluster and removing residual persistent disks using gcloud compute disks list and gcloud compute disks delete <disk-name> to make sure unnecessary cost was not incurred.

Terraform

We utilized Terraform to automate the deployment of a Google Kubernetes Engine (GKE) cluster and associated Kubernetes resources. This section provides an in-depth look at the Terraform deployment process, explaining its workflow, benefits, and structure.

How the Terraform Deployment Process Works

1. Service Account Creation:
    a. We started by creating a Google Cloud Platform (GCP) service account with roles necessary for managing Kubernetes and Compute Engine resources. This included roles such as Kubernetes Engine Admin, Compute Admin, and Service Account User.
    b. A key in JSON format was generated for the service account and saved as credentials.json in the root directory of our repository to enable Terraform to authenticate with GCP.
2. Initializing Terraform:
    a. The Terraform configuration was initialized using the command terraform init. This step set up the necessary backend and downloaded required provider plugins.

3. Applying the Terraform Configuration:
    a. We applied the Terraform configuration with terraform apply -var="project_id=your-gcp-project-id". This command executed the instructions in the main.tf, variables.tf, and outputs.tf files, which define the desired state of our infrastructure

.

4. Configuring kubectl:

    a. After the infrastructure was set up, we configured kubectl to interact with our newly created GKE cluster using gcloud container clusters get-credentials tinode-cluster --zone europe-west1-b --project your-gcp-project-id.

5. Verifying the Deployment:
    a. Verification steps included checking the status of nodes, pods, and services with commands like kubectl get nodes, kubectl get pods, and kubectl get svc.

6. Cleanup:
    a. To ensure resource efficiency and avoid unnecessary costs, the infrastructure could be torn down using terraform destroy -var="project_id=your-gcp-project-id".

Benefits of Using Terraform for Deployment

- Infrastructure as Code (IaC): Terraform allows us to define infrastructure in code, making it version-controlled, repeatable, and shareable. This approach reduces human error and improves collaboration.

- Consistency and Reproducibility: By using Terraform, we ensure that our infrastructure is consistent across different environments. This reproducibility is crucial for debugging and testing.

- Scalability: Terraform can manage infrastructure across multiple cloud providers and services, making it easier to scale and modify as project requirements evolve.

- Automation: Terraform automates the provisioning of resources, saving time and reducing manual intervention.

Structure of the Terraform Configuration

1. main.tf:
    a. The main.tf file contains the core configuration for creating the GKE cluster and other resources. It specifies the cluster's properties, such as the number of nodes, machine types, and networking configurations.
2. variables.tf:
    a. This file defines the input variables used in the Terraform configuration. Variables enhance the flexibility and reusability of the configuration by allowing different values to be passed in during execution
3. outputs.tf:
    a. The outputs.tf file specifies the outputs of the Terraform execution. These outputs can include information like the cluster name, endpoint, and other critical details needed for further configuration or integration.

4. Apply_manifests.sh:

a. This shell script (apply_manifests.sh) automates the application of Kubernetes manifests stored in the k8s directory. It runs kubectl apply -f commands to deploy resources like MySQL, Tinode, exporters, and the Horizontal Pod Autoscaler (HPA).

In summary, using Terraform to deploy our GKE cluster and Kubernetes resources streamlined our setup process, enhanced our deployment consistency, and provided a robust foundation for managing our cloud infrastructure efficiently. This approach not only simplified the initial setup but also ensured that our deployment could be easily scaled and maintained.

Testing

Testing was a crucial part of our project to ensure the Tinode chat application could handle varying loads and provide reliable performance. We utilized Locust for load testing and compared different deployment configurations to evaluate their effectiveness.

Single VM Instance

In the single VM instance setup, we used Docker Compose to manage the Tinode application and its dependencies. This setup was straightforward to deploy and allowed us to quickly get the application up and running. However, during load testing with Locust, we observed several limitations:

1. **Scalability**:
    - **Fixed Capacity**: The single VM instance was limited to the resources (CPU, memory, disk) of the single virtual machine. As the number of users and messages increased, the VM struggled to keep up, leading to performance degradation.
    - **Lack of Dynamic Scaling**: There was no mechanism to automatically increase resources based on demand, making it challenging to handle sudden spikes in traffic.
2. **Resource Utilization**:
    - **Resource Contention**: With all components (Tinode server, MySQL database) running on a single VM, they competed for the same resources, causing bottlenecks under high load.
    - **Single Point of Failure**: If the VM faced any issue, the entire application would go down, affecting all users.
3. **Performance**:
    - **Higher Latency**: Under heavy load, response times increased significantly, and the application sometimes failed to respond to user requests.
    - **Reduced Reliability**: The application became unreliable during peak usage periods, with higher error rates and slower performance.

3-Node Cluster Setup

The 3-node cluster setup improved upon the single VM instance by distributing the application components across three nodes, providing better performance and reliability:

1. **Scalability**:
    - **Distributed Load**: Each node handled a portion of the load, reducing the strain on individual nodes and improving overall application performance.
    - **Limited Manual Scaling**: While better than a single VM, this setup still required manual intervention to add or remove nodes based on demand.
2. **Resource Utilization**:
    - **Improved Efficiency**: By distributing components across multiple nodes, resource contention was minimized, and each node could operate more efficiently.
    - **Redundancy**: The failure of one node did not bring down the entire application, as other nodes could continue to handle requests.
3. **Performance**:
    - **Better Latency**: The distributed architecture reduced latency and improved response times compared to the single VM setup.
    - **Higher Reliability**: The application remained more stable under load, with fewer errors and better handling of user requests.

Despite the improvements, the 3-node cluster setup had limitations in scalability and required manual management, which could be cumbersome for large-scale deployments.

Google Kubernetes Engine (GKE) Managed Environment

Transitioning to GKE provided significant advantages in terms of scalability, resource management, and ease of operation:

1. **Scalability**:
    - **Automatic Scaling**: GKE, combined with Horizontal Pod Autoscaler (HPA), allowed the application to automatically adjust the number of pods based on CPU/memory usage or other metrics. We set up scaling policies that automatically adjust the number of pods. For example, if CPU usage exceeds 50%, HPA will scale out by adding more pods. Conversely, if CPU usage drops below 50%, HPA will scale in by reducing the number of pods. This dynamic scaling ensured that the application could handle high traffic loads efficiently.
    - **On-Demand Resource Allocation**: GKE could automatically provision additional resources when needed and scale back during periods of low demand, optimizing cost and performance.
2. **Resource Utilization**:
    - **Efficient Use of Resources**: By scaling up during peak times and scaling down during low traffic periods, GKE optimized resource usage and reduced costs.

    - **Managed Infrastructure**: GKE handled the underlying infrastructure, including node management, networking, and security, freeing us from these operational tasks.
3. **Performance**:
    - **Consistent Latency**: The dynamic scaling capabilities of GKE ensured that the application maintained low latency and high performance even under varying loads.
    - **High Availability**: With features like automated cluster management and integrated monitoring, GKE ensured high availability and quick detection of issues, enhancing overall reliability.

Load Testing with Locust on GKE:

1. **Setup**: We deployed Locust to simulate high traffic conditions.
2. **Metrics Collection**: Locust collected various performance metrics, including response times, error rates, and throughput.
3. **Results**: Under high traffic conditions, the application automatically scaled up by increasing the number of pods to handle the load. As traffic decreased, the application scaled down, conserving resources. This dynamic scaling ensured consistent performance and high availability.

Detailed Comparison:

- **Scalability**:
    - **Single VM Instance**: Limited to the resources of a single VM, unable to handle high traffic effectively, and lacked dynamic scaling.
    - **3-Node Cluster Setup**: Distributed load across three nodes, improving performance and reliability but required manual scaling.
    - **GKE Managed Environment**: Fully automatic scaling with HPA, efficiently handling high and variable traffic loads without manual intervention.
- **Resource Utilization**:
    - **Single VM Instance**: High resource contention and inefficient use of resources, leading to bottlenecks.
    - **3-Node Cluster Setup**: Better resource distribution and redundancy but still required manual resource management.
    - **GKE Managed Environment**: Optimal resource usage with automatic scaling, reducing costs and ensuring efficient operation.
- **Performance**:
    - **Single VM Instance**: High latency and reduced reliability under load.
    - **3-Node Cluster Setup**: Improved latency and reliability, but performance gains were limited by manual management.
    - **GKE Managed Environment**: Consistent low latency, high performance, and high availability, with automatic issue detection and resolution.
- **Ease of Management**:
    - **Single VM Instance**: Simple to set up but challenging to manage under load.

- **3-Node Cluster Setup**: More complex setup with better performance but required ongoing manual management.
- **GKE Managed Environment**: Simplified management with automated scaling, monitoring, and maintenance, allowing focus on application development.

In conclusion, testing and comparing these deployment configurations highlighted the superior capabilities of GKE for running our Tinode chat application. GKE's managed environment, coupled with HPA and load testing with Locust, provided a scalable, reliable, and efficient solution that met our project requirements.