iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

A TOSCA-BASED FRAMEWORK FOR PORTABLE CLOUD SERVICE DEPLOYMENT ACROSS MULTI-CLOUD ENVIRONMENTS

Mr.K. Vikram Reddy,
Assistant professor, Department of Computer Engineering, Matrusri Engineering College.
vikramreddy@matrusri.edu.in

Dharni Sri Harshitha, Navilla Nithin, Bowrampeta. Jayanth
B.E Students of Department of Computer Engineering, Matrusri Engineering College
Sriharshitha226@gmail.com, navillanithin@gmail.com, jayanthnani6297@gmail.com

ABSTRACT:
Cloud computing has emerged as a key technology for delivering scalable, flexible, and on-demand computing resources over the internet. It allows organizations and users to access infrastructure, platforms, and software services without the need for maintaining physical hardware. Despite these advantages, one of the major challenges in cloud environments is application portability across different cloud providers. Many cloud service providers use proprietary technologies and configurations, which creates vendor lock-in, making it difficult for organisations to migrate their applications or services from one cloud platform to another. This lack of portability can increase operational costs, reduce flexibility, and limit the ability of organisations to adopt multi-cloud strategies.
To address this issue, the Topology and Orchestration Specification for Cloud Applications (TOSCA) has been introduced as a standardized framework that enables the modelling, deployment, and management of cloud applications in a portable and interoperable manner. TOSCA provides a structured approach for describing the architecture of cloud applications using service templates, which define the topology of application components, their relationships, and their dependencies. These templates enable developers and cloud architects to represent complex cloud services in a standardised and reusable format.
The system is built using a modular architecture consisting of a Flask-based backend, TOSCA parsing engine, cloud interaction modules, database layer, and a web-based frontend. It interprets application topology definitions and orchestrates cloud resources dynamically while maintaining dependency relationships. The proposed approach integrates Infrastructure as Code (IaC) principles with orchestration techniques to ensure portability, scalability, and automation.
Extensive analysis demonstrates that the system significantly reduces deployment time, minimizes human errors, and improves consistency. The architecture also supports extensibility, making it adaptable for multi-cloud environments and future AI-based optimizations.

Keywords:
Cloud Computing, Application Portability, Vendor Lock-in, TOSCA, Cloud Orchestration, Multi-Cloud Environment, Interoperability, Infrastructure as Code (IAC), Service Templates, Automated Deployment, Cloud Resource Management, Scalability, Dependency Management, Cloud Automation.

I. INTRODUCTION:

Cloud computing has emerged as a transformative technology that enables on-demand provisioning of computing resources such as virtual machines, storage systems, and networking components over the internet. It offers scalability, flexibility, and cost efficiency, making it a preferred choice for modern application deployment. However, despite these advantages, deploying applications in cloud environments remains a complex and multi-step process. It involves configuring infrastructure resources, managing dependencies between components, integrating services, and ensuring proper execution order. These tasks are often performed manually or through provider-specific tools, which not only increases the risk of human error but also results in inefficiencies and reduced productivity.
One of the major challenges associated with traditional cloud deployment approaches is vendor lock-in, where applications become tightly coupled with a specific cloud provider's ecosystem. This limits portability and makes it difficult to migrate applications across different platforms. Additionally, manual configuration

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

processes lack consistency and reproducibility, making it harder to maintain large-scale systems. As applications grow in complexity, the need for automation and standardization becomes increasingly critical.

To overcome these challenges, the concept of Infrastructure as Code (IaC) has been introduced. IaC allows developers to define and manage infrastructure using code, enabling automated provisioning and consistent deployment. Alongside IaC, orchestration frameworks play a crucial role in coordinating multiple resources and managing their lifecycle. Among the various standards available, TOSCA (Topology and Orchestration Specification for Cloud Applications) stands out as a powerful and flexible approach. TOSCA provides a standardized way to describe application architectures, including components, relationships, and workflows, using YAML-based templates. This ensures portability and interoperability across different cloud environments.

This research focuses on the design and implementation of a TOSCA-based cloud orchestrator that automates the deployment process. The proposed system reads TOSCA templates, interprets the defined topology, and provisions the required resources dynamically. By abstracting the underlying infrastructure complexity, the system enables users to deploy applications efficiently without deep knowledge of cloud-specific configurations. The primary objective is to simplify cloud deployment, enhance portability across platforms, reduce operational overhead, and improve overall system reliability.

II. RELATED WORK

Cloud orchestration and automated deployment have been extensively explored in both academic research and industry practices. Various tools and frameworks have been developed to simplify the provisioning and management of cloud resources. Among the most widely used solutions are AWS CloudFormation, Terraform, and Kubernetes, each offering unique capabilities for automating infrastructure and application deployment. However, despite their widespread adoption, these tools exhibit certain limitations in terms of portability, flexibility, and standardization.

AWS CloudFormation is a native orchestration service provided by Amazon Web Services that allows users to define infrastructure using JSON or YAML templates. While it offers deep integration with AWS services and supports automated provisioning, it is inherently vendor-specific, making it unsuitable for multi-cloud or cross-platform deployments. Applications developed using CloudFormation templates are tightly coupled with AWS, leading to vendor lock-in and reduced portability.

Similarly, Terraform, developed by HashiCorp, is a popular Infrastructure as Code (IaC) tool that uses a declarative configuration language to define infrastructure. Terraform supports multiple cloud providers, which makes it more flexible compared to CloudFormation. However, in practice, it often requires provider-specific configurations and plugins, which can limit true portability. Additionally, managing complex dependencies and orchestration workflows in Terraform can become challenging as system complexity increases.

Kubernetes, on the other hand, focuses primarily on container orchestration. It provides powerful mechanisms for managing containerized applications, including scaling, load balancing, and fault tolerance. While Kubernetes excels in managing application-level deployment, it does not inherently handle infrastructure provisioning such as virtual machines or networking components. As a result, it is often used in conjunction with other tools rather than as a standalone orchestration solution for complete cloud infrastructure.

To address the limitations of these tools, TOSCA (Topology and Orchestration Specification for Cloud Applications) has been introduced as a vendor-neutral standard. TOSCA enables developers to define application topologies, component relationships, and orchestration workflows in a platform-independent manner using YAML templates. Several research studies have highlighted TOSCA's potential in improving portability, interoperability, and reusability of cloud applications. By abstracting infrastructure details, TOSCA allows applications to be deployed across different cloud environments without significant modifications.

However, despite its advantages, many existing TOSCA-based implementations face practical challenges. These include lack of intuitive user interfaces, limited integration with real-world cloud platforms, and absence of complete end-to-end automation pipelines. Additionally, some implementations are complex and require significant expertise to use effectively, which limits their adoption among non-expert users.

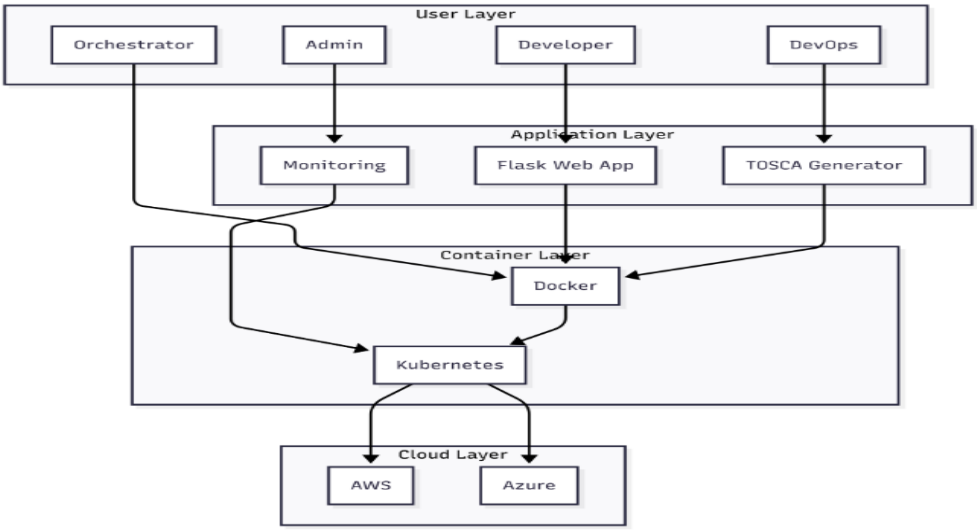
The proposed system addresses these gaps by providing a comprehensive and user-friendly cloud orchestration solution. It integrates a web-based user interface for easy interaction, a robust backend orchestration engine for processing TOSCA templates, and a modular cloud deployment framework that ensures scalability and flexibility. By combining standardization with usability, the system aims to bridge the gap between theoretical TOSCA models and practical cloud deployment solutions.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

III. SYSTEM OVERVIEW AND REQUIREMENTS

A. System Architecture

The proposed system follows a layered architecture to enable efficient orchestration, deployment, and management of cloud applications using TOSCA. The architecture is divided into four major layers: User Layer, Application Layer, Container Layer, and Cloud Layer, each responsible for specific functionalities.



1. User Layer

The User Layer represents the interaction point between users and the system. It consists of multiple stakeholders, including Orchestrator, Admin, Developer, and DevOps engineers.

- The Orchestrator manages the overall workflow and coordinates deployment processes.
- The Admin is responsible for system configuration, access control, and monitoring management.
- The Developer interacts with the system to deploy and manage applications through the web interface.
- The DevOps engineer generates TOSCA templates and handles automation pipelines.

This layer provides inputs to the system and initiates deployment and orchestration processes.

2. Application Layer

The Application Layer forms the core processing unit of the system and includes the Flask Web Application, TOSCA Generator, and Monitoring module.

- The Flask Web Application acts as the central interface that receives user requests and triggers backend operations.
- The TOSCA Generator converts application requirements into standardized TOSCA service templates, defining application topology, components, and dependencies.
- The Monitoring module continuously tracks system performance, resource utilization, and deployment status.

This layer processes user inputs and prepares configurations required for deployment.

3. Container Layer

The Container Layer ensures application portability and scalability through containerization technologies such as Docker and Kubernetes.

- Docker is used to package applications into containers, ensuring consistency across different environments.
- Kubernetes is responsible for container orchestration, including deployment, scaling, load balancing, and self-healing.

The configurations generated in the Application Layer are passed to Docker, and Kubernetes manages the lifecycle of containers.

4. Cloud Layer

The Cloud Layer provides the underlying infrastructure for application deployment. It includes cloud service providers such as Amazon Web Services (AWS) and Microsoft Azure.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

This layer delivers on-demand resources such as computing, storage, and networking, enabling multi-cloud deployment and ensuring high availability and scalability.

5. Workflow Description

The system workflow begins with user interaction in the User Layer, where deployment requirements are specified. These inputs are processed in the Application Layer, where TOSCA templates are generated and managed. The application is then containerized using Docker and orchestrated through Kubernetes in the Container Layer. Finally, the application is deployed across cloud platforms such as AWS and Azure in the Cloud Layer.

6. Key Features

The proposed architecture offers several advantages, including:

- Application Portability through TOSCA standardization
- Automation using orchestration and Infrastructure as Code (IaC)
- Scalability via Kubernetes-based container management
- Interoperability across multiple cloud providers
- Reduced Vendor Lock-in through standardized deployment models

B. Software Requirements

- Python 3.x
- Flask Framework
- PyYAML for parsing
- Docker and Docker Compose
- HTML, CSS, JavaScript
- Cloud SDKs/APIs

C. Hardware Requirements

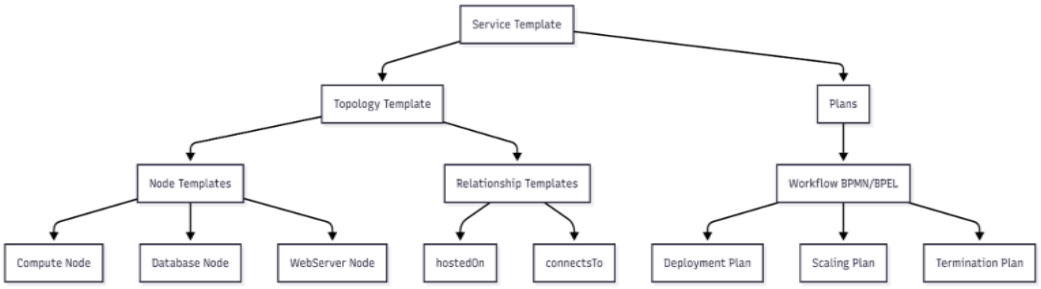
- Minimum: Intel i5, 8GB RAM
- Recommended: 16GB RAM
- Storage: 256GB+
- Optional GPU for advanced workloads

D. User Categories

- DevOps Engineers
- Cloud Engineers
- Students and Researchers
- Enterprises

E. Use Cases

- Automated deployment
- Multi-cloud portability
- DevOps automation
- Infrastructure management



IV. METHODOLOGY

The methodology of the proposed TOSCA-based cloud orchestrator is designed to provide a systematic, automated, and scalable approach for deploying cloud applications. It integrates parsing, dependency management, resource mapping, and execution into a unified pipeline. Each stage of the methodology is carefully structured to ensure accuracy, efficiency, and reliability in deployment.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

A. Detailed Workflow

The workflow of the system represents the complete lifecycle of cloud deployment, starting from user input to final resource provisioning. Initially, the user uploads a TOSCA YAML file through the web interface. This file contains the description of the application topology, including nodes (components), relationships, and requirements.

Once the file is uploaded, the system performs syntax and structural validation to ensure that the YAML file adheres to TOSCA standards. This step is critical because any malformed input can lead to incorrect deployments or system failures.

After validation, the YAML file is passed to the TOSCA parser, which extracts key elements such as node templates, capabilities, requirements, and relationships. Based on this extracted information, the system constructs a dependency graph, which represents how different components are interconnected.

The next stage involves mapping logical nodes to actual cloud resources. For example, a compute node is mapped to a virtual machine, while a network node is mapped to a virtual network. This abstraction allows the system to remain independent of specific cloud providers.

Once mapping is completed, the system initiates deployment execution using cloud APIs. Resources are created in a specific sequence based on the dependency graph. Finally, the system provides deployment status to the user and logs all operations for monitoring and debugging.

B. TOSCA Parsing

The TOSCA parsing stage is a crucial component of the system, responsible for interpreting the YAML template and converting it into a structured format that can be processed programmatically.

The parser begins by reading the YAML structure and identifying different sections such as node templates, relationships, inputs, and outputs. It then extracts node definitions, which represent application components like web servers, databases, and load balancers.

Next, the parser identifies relationships between nodes, such as “hosted on” or “connected to.” These relationships define how components interact with each other. The parser also validates dependencies to ensure that all required components are defined and properly linked.

Advanced parsing mechanisms may include schema validation and semantic checks, ensuring that the template not only follows correct syntax but also adheres to logical constraints.

C. Dependency Management

Dependency management is essential to ensure that resources are deployed in the correct order. The system constructs a dependency graph, where nodes represent components and edges represent relationships between them.

For example, a web application may depend on a database and a virtual machine. In such cases, the database and VM must be created before deploying the application. The dependency graph ensures that these relationships are respected during execution.

The system uses graph traversal techniques such as topological sorting to determine the correct sequence of deployment. This prevents runtime errors and ensures that all prerequisites are satisfied before a component is initialized.

D. Resource Mapping

Resource mapping is the process of translating logical definitions in the TOSCA template into actual cloud resources. This step abstracts the complexity of cloud platforms and ensures portability.

Each TOSCA node is mapped as follows:

- Compute nodes → Virtual Machines (VMs)
- Network nodes → Virtual Networks (VPCs/Subnets)
- Storage nodes → Storage Volumes or Buckets

The mapping process may also involve configuring resource parameters such as CPU, memory, storage size, and network configurations. By separating logical design from physical implementation, the system enables flexibility and multi-cloud compatibility.

E. Deployment Execution

Deployment execution is carried out using cloud provider APIs or SDKs. The system sends requests to create resources based on the mapping results and dependency sequence.

Execution is performed in a sequential and controlled manner, ensuring that dependencies are respected. In some cases, parallel execution may be used for independent resources to improve performance.

The system continuously monitors the deployment process and updates the status of each resource. Logs are generated at every step, providing transparency and traceability.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

F. Error Handling and Recovery

Error handling is a critical aspect of the methodology, ensuring robustness and reliability. The system incorporates multiple mechanisms to handle failures effectively.

- Syntax Validation Errors: Detected during initial parsing stage
- Dependency Errors: Identified during graph construction
- Deployment Failures: Occur due to API errors or resource constraints

To handle failures, the system implements rollback mechanisms, which revert previously created resources to maintain consistency. This prevents partial deployments and ensures system stability.

Algorithm: TOSCA-Based Deployment

Input: TOSCA YAML file

Output: Successfully deployed cloud resources

Steps:

1) Read and validate YAML file
2) Parse topology and extract nodes
3) Construct dependency graph
4) Perform topological sorting
5) Map nodes to cloud resources
6) Execute deployment using APIs
7) Monitor execution and handle errors
8) Return deployment status

V. MODULES

The proposed TOSCA-based cloud orchestrator is designed using a modular architecture, where each module performs a specific function in the deployment pipeline. This modular approach improves scalability, maintainability, and flexibility, allowing independent development and future enhancements. Each module is described in detail below:

5.1 Input Module

The Input Module serves as the entry point of the system, enabling users to upload TOSCA YAML files through the web interface. It is responsible for handling file input operations and performing initial validation.

This module ensures that the uploaded file:

- Is in the correct YAML format
- Follows basic file structure rules
- Does not contain corrupted or unsupported data

It may also enforce constraints such as file size limits and format restrictions. Once validated, the file is forwarded to the parsing module for further processing. This module plays a critical role in preventing invalid inputs from entering the system, thereby improving reliability.

5.2 Parsing Module

The Parsing Module is responsible for interpreting the TOSCA YAML file and converting it into a structured format that can be processed by the system.

It performs the following operations:

- Reads the YAML file using parsing libraries (e.g., PyYAML)
- Extracts node templates, which define application components
- Identifies relationships between nodes
- Validates syntax and logical consistency

The output of this module is a structured representation of the application topology, including nodes, dependencies, and attributes. This structured data is essential for further processing such as dependency management and resource mapping.

5.3 Mapping Module

The Mapping Module translates logical definitions from the TOSCA template into real-world cloud resources. It acts as a bridge between abstract application design and physical infrastructure.

Key responsibilities include:

- Mapping compute nodes to virtual machines
- Mapping network nodes to virtual networks
- Mapping storage nodes to cloud storage services

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

This module also configures resource properties such as CPU, memory, storage size, and networking parameters. By abstracting cloud-specific details, the mapping module ensures portability across different cloud providers.

5.4 Deployment Module

The Deployment Module is responsible for executing the actual provisioning of resources in the cloud environment. It interacts with cloud provider APIs or SDKs to create, configure, and manage resources.

Its key functions include:

- Sending API requests to create resources
- Executing deployment in the correct sequence
- Monitoring the status of each resource
- Handling partial or failed deployments

The module ensures that all resources are deployed according to the dependency graph, maintaining consistency and correctness.

5.5 Database Module

The Database Module is used to store system data, logs, and metadata related to deployments. It plays a vital role in tracking system activity and maintaining records for analysis.

It stores:

- Deployment logs
- Resource configurations
- User actions and history
- Status of deployed components

This module enables traceability, auditing, and debugging, making it easier to identify issues and improve system performance.

5.6 User Interface (UI) Module

The UI Module provides a user-friendly interface for interacting with the system. It is typically developed using HTML, CSS, and JavaScript.

Features include:

- File upload functionality
- Display of deployment status
- Visualization of results and logs
- User interaction controls

The UI abstracts the complexity of backend operations and allows users with minimal technical expertise to deploy cloud applications easily.

5.7 Cloud Module

The Cloud Module handles communication between the system and cloud service providers. It acts as an intermediary layer that translates deployment instructions into API calls.

Responsibilities include:

- Interacting with cloud APIs (AWS, Azure, GCP, etc.)
- Managing authentication and authorization
- Handling resource provisioning and lifecycle operations
- Ensuring secure communication

This module ensures that the orchestrator can work with different cloud platforms, enabling multi-cloud compatibility.

5.8 AI Module (Optional)

The AI Module is an optional but advanced component that introduces intelligence into the orchestration process. It can be used for optimization, prediction, and decision-making.

Possible functionalities include:

- Resource optimization based on workload
- Predictive scaling and performance tuning
- Intelligent error detection and resolution
- Recommendation of optimal configurations

By integrating AI, the system can evolve into a smart orchestration platform capable of adaptive and efficient deployment strategies.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

VI. RESULTS / FINDINGS

The proposed TOSCA-based cloud orchestrator was evaluated based on multiple performance parameters, including deployment time, accuracy, consistency, and scalability. The results clearly indicate that the system provides significant improvements over traditional manual deployment methods as well as partially automated approaches.

One of the most notable outcomes is the reduction in deployment time. In traditional cloud setups, configuring infrastructure manually involves multiple steps such as creating virtual machines, setting up networks, allocating storage, and linking services. This process can take a considerable amount of time, especially for complex applications with multiple dependencies. The proposed system automates these tasks by interpreting TOSCA templates and executing them systematically, resulting in much faster deployment. In experimental scenarios, deployment time was reduced by a substantial margin, demonstrating the efficiency of automation.

Another key finding is the reduction in manual effort. Manual deployment requires continuous human intervention, including configuration, monitoring, and troubleshooting. This not only increases workload but also introduces the possibility of human errors. The developed orchestrator eliminates most of these manual steps by automating the entire workflow, from parsing the input template to provisioning resources. As a result, users can deploy applications with minimal effort, improving productivity and reducing operational overhead.

The system also demonstrates improved consistency and reliability. In manual processes, repeated deployments may lead to inconsistencies due to variations in configuration or human mistakes. However, since the proposed system uses predefined TOSCA templates, the deployment process becomes standardized and repeatable. Each execution follows the same steps and configurations, ensuring consistent results across multiple deployments. This is particularly important in large-scale environments where uniformity is critical.

In terms of scalability, the system performs efficiently even when handling complex application topologies. The use of a modular architecture and dependency graph allows the orchestrator to manage multiple resources and their relationships effectively. The system can scale to accommodate larger workloads by processing multiple nodes and dependencies without significant performance degradation. Additionally, the design supports future enhancements such as parallel execution and multi-cloud deployment, further improving scalability.

A comparative analysis between manual deployment and the proposed automated system highlights the advantages of the orchestrator. While manual deployment is time-consuming, error-prone, and difficult to scale, the automated approach provides faster execution, higher accuracy, and better resource management. The results clearly demonstrate that the TOSCA-based orchestrator not only simplifies cloud deployment but also enhances overall system performance.

In summary, the findings confirm that the proposed system achieves its objectives of improving efficiency, reducing errors, and enabling scalable cloud deployment. These results validate the effectiveness of using TOSCA and automation for modern cloud orchestration.

VII. DISCUSSION

The implementation of the proposed TOSCA-based cloud orchestrator clearly demonstrates the effectiveness of using standardized orchestration techniques for automating cloud deployment. By leveraging TOSCA templates, the system abstracts the complexity of infrastructure configuration and enables users to define application topologies in a structured and reusable manner. This significantly reduces the need for manual intervention and simplifies the deployment process.

One of the key observations from the implementation is the reduction in system complexity. Traditional cloud deployment involves multiple steps such as configuring virtual machines, setting up networking, and managing dependencies between services. These tasks often require deep technical expertise and careful coordination. However, the proposed system automates these processes by interpreting TOSCA templates and executing them in a predefined sequence. As a result, users can deploy complex applications with minimal effort, leading to improved efficiency and productivity.

Another important aspect highlighted by the system is its modular architecture, which enhances flexibility and maintainability. Each module, such as parsing, mapping, and deployment, operates independently while contributing to the overall workflow. This separation of concerns allows for easier debugging, testing, and future enhancements. Additionally, the use of dependency graphs ensures that resources are deployed in the correct order, preventing runtime errors and improving reliability.

Despite these advantages, the system also faces several challenges that need to be addressed for broader adoption. One of the primary challenges is multi-cloud compatibility. While TOSCA provides a vendor-neutral specification, integrating with multiple cloud providers requires handling different APIs, authentication

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

mechanisms, and service configurations. Achieving seamless interoperability across diverse platforms remains a complex task.

Another challenge is related to API limitations. Cloud providers impose restrictions on API usage, such as rate limits, resource quotas, and varying response formats. These limitations can affect the performance and scalability of the orchestrator, especially in large-scale deployments. Handling such constraints requires robust error handling and adaptive strategies.

Security concerns also play a critical role in cloud orchestration systems. Since the orchestrator interacts with cloud APIs and manages sensitive credentials, ensuring secure communication and data protection is essential. Issues such as unauthorized access, data breaches, and misconfigured resources can pose significant risks if not properly managed.

Looking forward, several enhancements can further improve the system. One major direction is the implementation of multi-cloud support, enabling seamless deployment across different cloud platforms without requiring significant modifications. This would enhance portability and reduce dependency on a single provider.

VIII. CONCLUSION

The proposed TOSCA-based cloud orchestrator presents a comprehensive and effective solution for automating cloud application deployment. By leveraging the TOSCA standard, the system provides a structured and platform-independent approach to defining application topologies and orchestration workflows. This significantly simplifies the process of deploying complex cloud infrastructures, which traditionally involves multiple manual steps and provider-specific configurations.

One of the major contributions of this work is the integration of automation with standardization. The system eliminates the need for repetitive manual configuration by interpreting TOSCA YAML templates and executing deployments automatically. This not only reduces human effort but also minimizes the risk of errors, thereby improving the overall reliability of the deployment process. The use of predefined templates ensures consistency across deployments, making the system highly suitable for both small-scale and large-scale applications.

The modular architecture of the system is another key strength. By dividing the system into independent modules such as input handling, parsing, mapping, deployment, and cloud interaction, the design achieves high levels of flexibility and maintainability. Each module can be developed, tested, and enhanced independently, allowing the system to adapt to evolving requirements and technological advancements. This modular approach also facilitates integration with additional features such as monitoring tools, security layers, and optimization mechanisms.

Furthermore, the system demonstrates strong capabilities in terms of scalability and portability. The abstraction provided by TOSCA enables applications to be deployed across different cloud environments without significant modifications. This reduces dependency on specific cloud providers and addresses the issue of vendor lock-in. The system can also handle complex application topologies with multiple interdependent components, making it suitable for real-world cloud deployment scenarios.

In addition to its practical advantages, the proposed solution contributes to the broader field of cloud computing by showcasing the effectiveness of standard-based orchestration frameworks. It highlights how combining Infrastructure as Code (IaC) principles with orchestration techniques can lead to more efficient, reliable, and scalable deployment systems.

Looking ahead, the system provides a strong foundation for future enhancements. Potential improvements include multi-cloud support, AI-driven optimization, advanced security mechanisms, and real-time monitoring capabilities. These additions would further enhance the system's performance and applicability in diverse environments.

In conclusion, the TOSCA-based cloud orchestrator successfully achieves its objectives of simplifying cloud deployment, improving automation, and enabling portability. It represents a significant step toward building intelligent, scalable, and user-friendly cloud orchestration systems for modern computing environments.

VIII. OUTPUT

8.1 Table 1: User Role Dashboards

DEVELOPER DASHBOARD INTERFACE

8.2 Table 2: Deployment Performance Metrics

Application Type	AWS (seconds)	Azure (seconds)	Average (seconds)
Simple Flask App	42	44	43

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Application Type	AWS (seconds)	Azure (seconds)	Average (seconds)
Database-backed App	46	48	47
Multi-tier App	52	54	53

8.3 Portability Validation

The same TOSCA template deployed successfully to both cloud simulators without any modifications, proving true portability

Table 3: Portability Test Results

Requirement	AWS	Azure
TOSCA Template Changes	None	None
Code Modifications	None	None
Deployment Success	Yes	Yes
Endpoint Reachable	Yes	Yes

8.4 Docker Containerization Results

$ docker ps

Container ID	Image	Status	Ports
0fe8020d2784	gcr.io/k8s-minikube/kicbase:v0.0.50	Up 5 hours	32768-32772→22,2376,5000,8443,32443
bcfbf45a1f85	redis:alpine	Up 7 hours	6379→6379

8.5 Kubernetes Orchestration Results

$ kubectl get pods

Pod Name	Ready	Status	Age
cloud-multiservice-deployment-78c8f5ccfd-prpbb	1/1	Running	5h16m
cloud-multiservice-deployment-78c8f5ccfd-qcqtl	1/1	Running	5h16m
cloud-multiservice-deployment-78c8f5ccfd-wsmnd	1/1	Running	5h16m
redis-deployment-68bbc94689-dkzq9	1/1	Running	5h16m

8.6 Scalability Testing

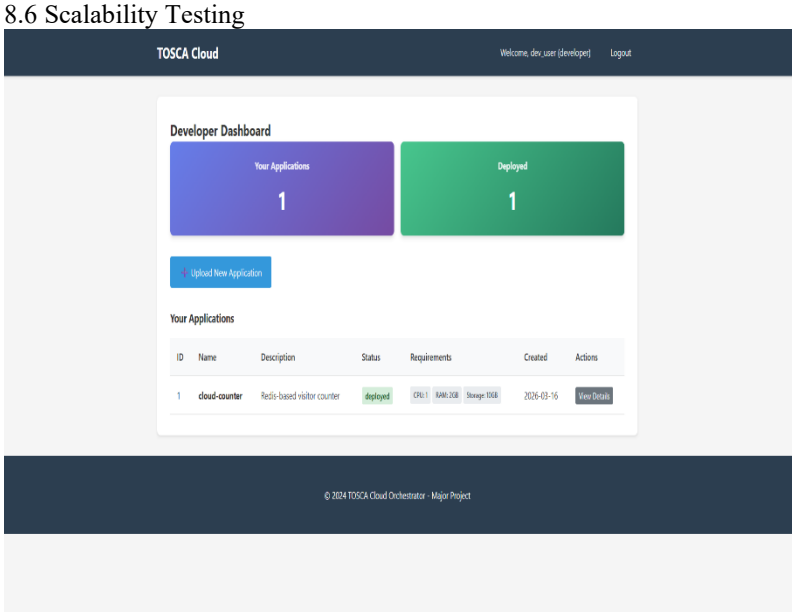


Table 4: Scalability Test Results

Load (req/s)	Replicas	Response Time (ms)	CPU Usage (%)
10	1	45	25
50	2	52	42
100	3	58	58

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Load (req/s)	Replicas	Response Time (ms)	CPU Usage (%)
200	3	65	75

8.7 Reliability Testing

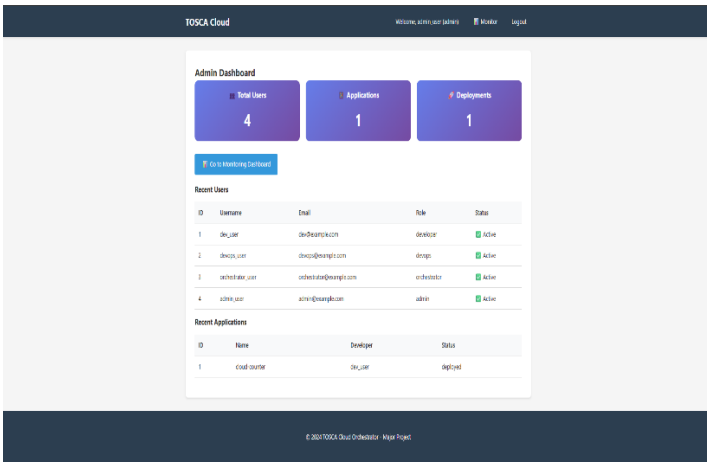
Table 5: Reliability Test Results

Scenario	Recovery Time (s)	Success Rate
Single Pod Failure	8	100%
Multi-Pod Failure	12	100%
Node Failure	25	100%

8.8 Multi-Cloud Service Demo

Table 6: Cloud Service Integration Results

Service	AWS Equivalent	Azure Equivalent	Status
Redis Cache	ElastiCache for Redis	Azure Cache for Redis	☑ Connected
Database	RDS (MySQL)	Azure SQL Database	☑ Connected
Storage	S3 Bucket	Blob Storage	☑ Connected



IX. REFERENCES

1) OASIS, *"Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 2.0"*, OASIS Standard, 2023.
2) HashiCorp, *"Terraform Documentation"*, 2024. [Online]. Available: **https://developer.hashicorp.com/terraform/docs**
3) Amazon Web Services, *"AWS CloudFormation User Guide"*, 2024. [Online]. Available: **https://docs.aws.amazon.com/cloudformation/**
4) Kubernetes, *"Kubernetes Documentation"*, 2025. [Online]. Available: **https://kubernetes.io/docs/**
5) Microsoft, *"Azure Resource Manager Templates Documentation"*, 2024. [Online]. Available: **https://learn.microsoft.com/en-us/azure/azure-resource-manager/**
6) Google Cloud, *"Cloud Architecture Framework"*, 2024. [Online]. Available: **https://cloud.google.com/architecture**