International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 11-No.3 (2025) pp. 6083-6090
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Seamless Low-Cost, Low-Maintenance DR Orchestration with Azure

Swati Karni*

Department of Information Technology, University of the Cumberlands, KY, USA
*** Corresponding Author Email:** skarni70360@ucumberlands.edu - **ORCID:** 0009-0005-6180-7603

Article Info:

DOI: 10.22399/ijcesen.3777
Received : 24 June 2025
Accepted : 12 August 2025

Keywords

Disaster Recovery (DR),
Low-cost solution,
Microsoft Azure,
Azure Site Recovery,
Terraform,
Automated replication

Abstract:

As more organizations move their systems to the cloud, keeping important applications running during unexpected disruptions has become very important. Traditional disaster recovery (DR) methods can protect these systems, but they are often expensive and hard to manage. This study introduces a simpler, more affordable, and low-maintenance DR solution using built-in tools from Microsoft Azure. By using Azure Site Recovery, Azure Automation, and Terraform, the solution was designed to automatically take care of replication, failover, and failback. It was tested in real-world scenarios to check how well it works during outages. Results showed faster recovery times, minimal manual effort, and significant cost savings—over 60% compared to older DR methods. With smart use of automation and cost controls like resource scaling and storage optimization, the solution proved easy to maintain and effective in keeping services running. Overall, the Azure-based DR model offers a simple yet powerful way for organizations to protect their operations without expensive tools or adding extra workload to IT teams. In addition to streamlining recovery operations, the solution enables seamless, automated DR testing without impacting production environments—ensuring ongoing readiness, compliance, and confidence in recovery processes.

1. Introduction

The cloud is now being used by many companies to run their important systems. When unexpected events—such as power outages, cyberattacks, or system crashes—occur, it is expected that those systems be restored quickly. Disaster recovery (DR) is used for this purpose, but older DR methods are often seen as expensive and difficult to manage. Traditional disaster recovery technology is a reactive remediation model, which is inevitably affected by the timeliness and integrity of data backup and the backup method and volume of backup. Frequent testing of DR plans is also avoided by many businesses, as the process is viewed as time-consuming and likely to cause issues in daily operations and have the potential to disrupt production data. In this study, a simple, low-cost, and easy-to-maintain method for disaster recovery is presented using built-in tools from Microsoft Azure. Azure Site Recovery, Azure Automation, and Terraform are used to manage key DR tasks automatically—such as data replication, failover to backup systems, and failback once normal operations are restored. This approach is designed to reduce manual work, lower costs, and allow DR testing to be carried out without disturbing live systems. In the past, disaster recovery was done using physical equipment and manual work, which made it slow and costly. Now, with cloud technology, the process is faster and more efficient because it can be automated. However, testing DR systems regularly is still difficult for many organizations. DR drills are often avoided because they are costly and can lead to disruptions if they affect the production environment. In some cases, when the DR setup is changed over time (a problem known as “drift”), it may no longer reflect the production environment, which can result in failed recovery attempts. Although the benefits of cloud-based DR have been shown in earlier research, limited information has been provided on fully automated setups that use both Azure and Terraform. Clear methods for safe, effective DR testing—without returning test data to the production environment—have also not been well-documented. This study has been guided by the following research questions: In what ways can Azure and Terraform be used to build a low-cost, low-maintenance disaster recovery solution?

How can DR testing be carried out easily without causing harm to live systems?
What issues—such as configuration and code drift—still exist, and how might they be addressed?
This research has been focused solely on Microsoft Azure and does not consider other cloud providers or third-party recovery tools. Attention has been placed on routine DR testing, not full system failovers during actual outages. One known limitation is that Terraform code drift is not fully addressed in this study, though it is suggested that future research can explore this issue further.
In conclusion, it is shown through this study that a cloud-based, automated disaster recovery system using Azure and Terraform can be created in a way that is cost-effective, reliable, and simple to maintain. Additionally, regular DR testing can be made easier and safer without affecting production environments.

2. Material and Methods

This section explains the tools and methods used to build and test an automated disaster recovery system in the cloud. Infrastructure as Code, automation, and cloud-based recovery services were used to improve performance and protect data.

HashiCorp Configuration Language (HCL)

HCL is the language used to write Terraform scripts. It allows users to define cloud infrastructure—like servers and storage—in code. This makes it easier to manage and repeat deployments. HCL helps create clear and organized files that tell Terraform what to build and how to do it [1].

Visual Studio Code

Visual Studio Code is the main tool used to write and edit HCL files. It is a code editor with helpful features like syntax checking and plug-ins that work with Terraform. It also connects to GitHub so developers can track changes to their code [2].

GitHub

GitHub was used to store all the infrastructure code and manage versions. Every change made to the code was saved in GitHub, allowing for team collaboration and history tracking. GitHub was also set up to automatically trigger Terraform actions when new code was added [3].

Terraform Enterprise (TFE)

Terraform, an infrastructure-as-code software tool developed by HashiCorp, utilizes the HashiCorp Configuration Language (HCL) as its primary configuration language. Terraform does the provisioning and management of the infrastructure by using single configuration file and also using versioning for easy save and rollbacks [4].
Terraform Enterprise is a paid version of Terraform that provides advanced features for managing cloud infrastructure. When HCL code is submitted, TFE checks what resources already exist and what needs to be changed. It compares the current environment with the desired setup and applies only the changes needed. It also keeps a “state file,” which records all the resources that have been created. This file helps Terraform know what’s been done and what still needs to be done [1].

HashiCorp Vault

HashiCorp Vault was used to keep sensitive information—like passwords and access keys—safe. Instead of hardcoding secrets into the code, Vault gave Terraform temporary credentials to log in to Microsoft Azure. This improves security by avoiding long-lived secrets and limiting who can access them [5].

Microsoft Azure

Microsoft Azure was the cloud provider where all virtual machines, storage, recovery service vaults and disaster recovery services were set up. Azure allowed the team to automatically create and manage cloud resources using scripts. It also supports integration with other tools like Terraform and Ansible [6].

Ansible

Ansible was used to configure virtual machines after they were created. For example, it installed necessary software and updated system settings. Ansible works without needing agents on the machines, which made it fast and easy to use. It helped keep all machines consistent [7].

Disaster Recovery

Disaster recovery (DR) means bringing back critical IT systems after something goes wrong, such as a power cut, hardware failure, or cyberattack. In the past, DR relied on physical machines and a lot of manual work, which made the process slow and costly. Today, cloud-based DR tools help organizations back up and restore systems much faster and with less effort [8].
Modern DR uses tools that automatically detect issues, failover to backup systems, and restore data with minimal human intervention. Key goals include reducing Recovery Time Objective (RTO)—how quickly systems come back online—and Recovery Point Objective (RPO)—how much recent data is recoverable [9].

Azure Site Recovery (ASR)

Azure Site Recovery is a service from Microsoft that helps keep systems running during a disaster. It copies virtual machines and data to another location in Azure or on-premises. If a failure happens, organizations can “failover” to the backup copy and keep working. ASR can also test recovery plans without affecting the live system [6]. ASR supports both planned and emergency recovery and can be customized to meet business needs. It helps meet compliance requirements and reduces downtime by automating the recovery process [10].

Code Development and Versioning

Infrastructure code is written in HCL using Visual Studio Code. Code is committed and pushed to a GitHub repository, typically to the master branch.

CI/CD Integration

GitHub is configured to trigger Terraform Enterprise build plans upon code commits. TFE workspaces are set up to either auto-apply plans or require manual approval for change review.

Authentication and Provisioning

TFE uses HashiCorp Vault to securely authenticate with Azure. Upon successful authentication, TFE initiates the provisioning of defined resources.

3. Virtual Machine Deployment

Azure Virtual Machines are created based on the HCL definitions. Post-deployment, TFE triggers Ansible playbooks for additional configuration and software installation. The Terraform code shown in *Listing 1* is used to provision a basic infrastructure environment in Microsoft Azure using the azurerm provider. It begins by specifying the Azure provider configuration with the required features block. A **resource group** is created first, which acts as a logical container for all other resources.

Next, the code provisions a **virtual network (VNet)** and a **subnet** within that VNet, defining address spaces for internal communication. A **network interface card (NIC)** is then created and linked to the subnet. This NIC will later be attached to a virtual machine.

Finally, the configuration defines a **Windows virtual machine (VM)** using the azurerm_windows_virtual_machine resource. It specifies VM properties such as size, admin credentials, operating system image, and attaches the previously created NIC. The OS disk configuration and source image reference point to a Microsoft-published Windows Server 2019 image.

This infrastructure is defined using HashiCorp Configuration Language (HCL) and can be deployed in a fully automated way via Terraform. It provides a foundational setup commonly used in cloud automation pipelines and disaster recovery solutions (HashiCorp, n.d.).

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "East US"
}

resource "azurerm_virtual_network" "example" {
  name                = "example-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.example.location
                  resource_group_name                =
azurerm_resource_group.example.name
}

resource "azurerm_subnet" "example" {
  name                 = "example-subnet"
                  resource_group_name                =
azurerm_resource_group.example.name
                  virtual_network_name               =
azurerm_virtual_network.example.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_network_interface" "example" {
  name                = "example-nic"
  location            = azurerm_resource_group.example.location
                  resource_group_name                =
azurerm_resource_group.example.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.example.id
    private_ip_address_allocation = "Dynamic"
  }
}

resource "azurerm_windows_virtual_machine" "example" {
  name                = "example-vm"
                  resource_group_name                =
azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  size                = "Standard_DS1_v2"
  admin_username      = "azureuser"
  admin_password      = "P@ssword1234!"
  network_interface_ids = [
    azurerm_network_interface.example.id,
  ]

  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  source_image_reference {
    publisher = "MicrosoftWindowsServer"
    offer     = "WindowsServer"
    sku       = "2019-Datacenter"
    version   = "latest"
  }
}
```

Listing 1 : Terraform Code

Azure Site Recovery (ASR)

According to Microsoft (2024), Azure Site Recovery is a built-in cloud service that automatically copies, switches over, and restores virtual machines, physical servers, and on-premises systems to Azure when a disaster happens.

- **Automated DR setup:** ASR enables automated deployment of replication and disaster recovery directly from the Azure portal using Recovery Services vaults, without needing manual scripting or configuration [2].
- **One-click failover and failback:** Planned or unplanned failovers can be initiated with a single click, and automated recovery plans can be

created to orchestrate the order in which VMs come online in the secondary region [2].

- **Minimal downtime:** Microsoft states that ASR can achieve recovery time objectives (RTOs) of minutes and recovery point objectives (RPOs) as low as seconds, depending on the replication frequency and workload configuration.
- **Non-disruptive testing:** DR drills can be carried out anytime using test failover, allowing validation of recovery plans without impacting production environments.

Terraform code in Listing 2 defines the azurerm_site_recovery_replicated_vm resource, which automates the setup of Azure Site Recovery (ASR) for replicating a virtual machine from a primary region to a secondary region. This configuration includes references to the source virtual machine, the associated Recovery Services Vault, and the replication policy that governs how often data is synchronized and retained. It also specifies the source and target recovery fabrics and protection containers, which represent logical groupings of infrastructure in both the primary and secondary locations. The replicated virtual machine is configured to use Premium_LRS managed disks in the secondary region, with data temporarily staged in a specified storage account. The network interface settings map the source NIC to a target subnet and assign a public IP address in the secondary region, ensuring the VM can be accessed after failover. Additionally, the `depends_on` block ensures that protection container mappings and network mappings are established before replication begins. This resource simplifies and automates the disaster recovery configuration process, allowing for a reliable failover strategy using Azure Site Recovery and Infrastructure as Code (HashiCorp, n.d.).

```
    resource "azurerm_site_recovery_replicated_vm" "vm-replication" {
    name                                = "vm-replication"
    resource_group_name                 = azurerm_resource_group.secondary.name
    recovery_vault_name                 = azurerm_recovery_services_vault.vault.name
    source_recovery_fabric_name         = azurerm_site_recovery_fabric.primary.name
    source_vm_id                        = azurerm_virtual_machine.vm.id
    recovery_replication_policy_id      = azurerm_site_recovery_replication_policy.policy.id
    source_recovery_protection_container_name = azurerm_site_recovery_protection_container.primary.name

    target_resource_group_id            = azurerm_resource_group.secondary.id
    target_recovery_fabric_id           = azurerm_site_recovery_fabric.secondary.id
    target_recovery_protection_container_id = azurerm_site_recovery_protection_container.secondary.id

    managed_disk {
    disk_id                  = azurerm_virtual_machine.vm.storage_os_disk[0].managed_disk_id
    staging_storage_account_id = azurerm_storage_account.primary.id
    target_resource_group_id   = azurerm_resource_group.secondary.id
    target_disk_type           = "Premium_LRS"
    target_replica_disk_type   = "Premium_LRS"
    }

    network_interface {
    source_network_interface_id   = azurerm_network_interface.vm.id
    target_subnet_name            = azurerm_subnet.secondary.name
    recovery_public_ip_address_id = azurerm_public_ip.secondary.id
    }

    depends_on = [
    azurerm_site_recovery_protection_container_mapping.container-mapping,
    azurerm_site_recovery_network_mapping.network-mapping,
      ]
}
```

4. Infrastructure Automation Workflow

To develop the CI/CD framework and achieve automated disaster recovery infrastructure, a systematic and repeatable approach was followed. The objective was to ensure that the framework could be quickly replicated by others. The following step-by-step process was used during the investigation:

Step 1: Terraform code was written in Visual Studio Code to define all required Azure resources such as resource groups, virtual networks, subnets, availability sets, cache storage accounts, Recovery Services Vaults, replication policies and site recovery resource for the secondary region.

Step 2: The Terraform files were then committed to a GitHub repository so that changes are version-controlled and collaboration is managed. GitHub was connected to Terraform Enterprise (TFE) so that any update to the main branch triggers an automated Terraform *plan*.

Step 3: After the plan was manually reviewed and applied in Terraform Enterprise or permissions were granted to auto apply the plan, TFE securely retrieved Azure credentials from HashiCorp Vault and authenticated with Azure.

Step 4: Upon authentication, Terraform Enterprise provisioned the infrastructure automatically in the correct sequence — beginning with network and storage resources, followed by virtual machines, and finally, the Azure Site Recovery resources once the VMs have been deployed.
Step 5: After the virtual machines were created, an Ansible workflow was run to set them up by installing necessary software, agents, and security settings.
Step 6: Finally, the disaster recovery setup was validated by monitoring replication health and performing test failover and failback procedures to ensure a one-click, automated recovery workflow is functional.

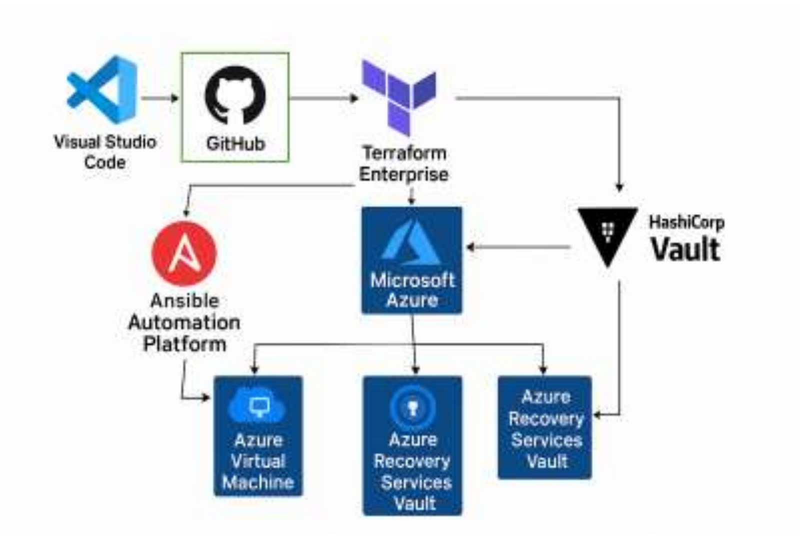

Figure 1: *Infrastructure Deployment Architecture*

Figure 1 shows infrastructure deployment architecture. For Azure Site Recovery configuration on Virtual Machines, additional resources—including cache storage accounts, Recovery Services Vaults, and replication policies—must be deployed. These resources can be defined within the same TFE workspace and deployed alongside the Virtual Machines. Terraform Enterprise intelligently manages resource dependencies, ensuring that Site Recovery components are provisioned only after the Virtual Machine is successfully built.

5. Failover and Failback Options

This section outlines three distinct approaches for performing failover and failback, each tailored to achieve optimal results based on specific scenarios or requirements.

5.1 Failover Process for a real Disaster or Outage

Step 1: The Azure portal is opened.
Step 2: The **Recovery Services Vault** is accessed.
Step 3: Under *Site Recovery* > *Replicated Items*, the virtual machine (VM) or workload to fail over is selected.
Step 4: "Failover" is clicked.

- A recovery point is selected (usually the latest one).
- The desired **target network** is chosen for the failover operation.

Step 5: The failover process is started by Azure.

- The VM is brought online in the secondary (disaster recovery) region.
- This process needs shutting down the primary region servers, which is also automatically handled by the resource manager unless opted out.
- Services and connectivity are checked in the secondary region.

Step 6: Commit is clicked.

- This confirms that the failover was successful.
- Recovery points are cleaned up, and rollback is no longer possible.

Step 7: Re-protect is initiated.

- Replication is set up in reverse—from the new (secondary) region back to the original (primary) region.
- This prepares the system for a future failback.

Step 8: Business operations continue from the secondary site until the primary site is ready.

Failback Process (Returning to the Primary Site)

Step 1: The primary site is restored and verified to be ready for failback.
Step 2: In the Azure portal, the **Recovery Services Vault** is accessed again.
Step 3: The failed-over VM is selected.
Step 4: "Failback" is clicked.

- A recovery point is chosen to capture the latest data from the secondary site.

Step 5: Azure starts the failback process.

- A VM is created in the original primary site using the latest replicated data.

Step 6: Once the VM is running in the primary site, **Commit** is clicked.

- This finalizes the failback and confirms the return to normal operations.

Step 7: Re-protect is clicked (optional but recommended).

- Replication is resumed from the primary site to the secondary site, restoring full disaster recovery readiness.

Proposed Failover Process for end to end DR drills/tests

Step 1: Repeat Steps 1 through 5 as outlined in Section 5.1.
Step 2: Perform validation and testing from the secondary site.
Step 3: Upon completion of testing, decommission the temporary resources provisioned in the secondary region.
Step 4: Power on the primary virtual machines (VMs).
Step 5: Reconfigure and reestablish Azure Site Recovery in the primary region using the automated CI/CD pipeline.

This disaster recovery approach provides several important benefits. It helps reduce time taken and expense to perform DR drills, by quickly switching back to the primary systems using Azure Site Recovery and automated CI/CD tools. Testing can be done safely in a separate environment, so the main systems are never affected. Using automation and scripts makes the test process easy to repeat and less likely to have mistakes

5.3 Test Failover Process for high level tests/drills

A Test Failover is used to check if the disaster recovery setup is working properly, without affecting the live (production) systems. It is considered a safe way to practice the recovery plan.

Step 1: Azure Portal is opened.
Step 2: The Recovery Services Vault is located and accessed in the Azure portal.
Step 3: Under *Site Recovery* > *Replicated Items*, the VMs that need to be tested are selected.
Step 4: "Test Failover" is clicked.

- A recovery point is chosen (usually the latest one).
- A test network is selected—this must be a separate network so that the production environment is not affected.

Step 5: The test VM is started by Azure.
Step 6: A temporary copy of the VM is created and started in the test network.

- The production VM remains unchanged.
- The test VM is used for validation.

Step 7: The test VM is logged into.

- It is confirmed that the VM boots up successfully.
- Applications and services are checked.
- Network and storage connections are verified.

Step 8: Notes are taken.

- Any issues or unexpected behavior are written down.
- Suggestions for improvement are recorded.

Step 9: The test environment is cleaned up.

- In the portal, the "Cleanup Test Failover" button is clicked.
- The test VM and related resources are deleted.

During a test failover, the production environment remains unaffected, as the testing is always carried out in a separate, isolated network. While temporary costs may be incurred for compute and storage, the process can be repeated easily using automation tools such as PowerShell. The purpose of conducting a test failover is to validate the disaster recovery plan in a safe environment, identify any weaknesses before an actual disaster occurs, and build confidence in the organization's ability to recover systems effectively.

Role of OpenAI in the Proposed Project

According to S. N. et al., OpenAI contributes to the automation of Infrastructure as Code (IaC) even though it has not developed a dedicated IaC generation tool, and its technologies help developers and engineers more easily build and enhance such systems [11]. First, OpenAI's language models—such as GPT-3.5—can understand everyday language from users and turn it into useful code examples or templates. This means people can describe what they need in plain English, and the system can help write the IaC code for them. Second, OpenAI's research in artificial intelligence provides techniques that can be used to automatically learn from existing infrastructure setups and create code that fits different needs or environments. Third, these models have access to a large amount of information about cloud services, IaC tools, and best practices. As a result, they can suggest improvements, identify mistakes, and guide users through the process of writing IaC code. OpenAI's technology can also be built into collaboration tools that developers use, offering helpful features like auto-completion, smart code suggestions, and context-based help. Even though OpenAI doesn't offer its own IaC generator, companies can use OpenAI's API to build custom tools that automate parts of the IaC process to save time and reduce errors [11].

6. Results and Discussion

In today's business environment, the data services operated by Cloud Providers encounter many challenges in ensuring a high level of reliability of data services before and after disasters [12]. Data services must ensure reliability and flexibility through an effective and practical DR plan [13]. The main issue concerning disaster recovery in the cloud computing context is how to provide an effective plan for data backup and recovery that guarantees high data reliability at a reasonable cost prior to a disaster [13].

With the proposed architecture in this study, significant improvements were demonstrated by the proposed automated model in deploying and managing automated disaster recovery. First, the provisioning and deployment of Azure Site Recovery along with all required cloud components were successfully automated through Infrastructure as Code (IaC) using Terraform. Manual efforts were reduced to writing the Terraform code once—a task that can be further streamlined by leveraging AI services such as OpenAI. Subsequent deployments using the proposed CI/CD pipeline takes only a few minutes for a standard 4 vcpu 16 GB RAM VM, with 128 GB OS and Data disk attached, which could be higher or lower, depending on the scale and number of virtual machines involved.

After ASR was configured, one-click failover for all protected virtual machines was available using recovery plans in Azure Recovery Services vaults. The applications were successfully brought online in the secondary region, confirming that the failover process was effective. For failback, three efficient methods were developed.

1. In the first method, made for actual DR situations, the state of VMs in the secondary region was reprotected and committed, then failback to the primary region was performed using simple recommit, reprotect, and failback options in the ASR dashboard.
2. The second method, made for disaster recovery tests, involved shutting down the backup region's resources, restarting the virtual machines in the main region, and turning site recovery back on. This way, disaster recovery testing could be done without affecting live data.
3. High level DR drills can be also carried out anytime using test failover, allowing validation of recovery plans without impacting production environments. Shutting down the primary environment is not required in this approach, however a full application restoration in the secondary region is not achievable with test failover approach.

Recovery point objective (RPO) as low as five minutes for crash-consistent snapshots and one hour for application-consistent snapshots, were achieved, reflecting high data protection standards.

Recovery Time Objective (RTO) as low as 15-20 minutes was achieved. Traditional disaster recovery service systems no longer meet user demands for fast deployment, shared cloud resources, and cross-cloud failover in multi-cloud environments. Traditional manual disaster recovery setups usually take 5 to 10 days to set up and several hours to switch over or switch back. This automated model greatly cuts down the time and work needed. Failover and failback are now done with just one click, which lowers the chance of mistakes and makes the process more reliable.

Azure Site Recovery further reduces disaster recovery costs by eliminating the need to maintain a full standby infrastructure. Instead of paying for duplicate servers and storage in a secondary site, organizations only pay based on the number of protected machines — with the first 31 days free per instance. After that, protection is charged at \$16/month per instance for customer-owned site-to-site replication and \$25/month per instance for replication to Azure, with pricing varying by region. Since Azure only builds the recovery infrastructure during an actual failover, and the replicated virtual machines stay powered off under normal conditions, expensive compute charges are avoided—making ASR a highly cost-efficient alternative to traditional disaster recovery setups.

Additionally, cost, data drifts and downtime was minimized during DR drills by avoiding data movement from secondary to primary regions. The full failover option does come with code in the terraform code due to the disk resource ids being different after the failover. This limitation is yet to be addressed and remains an area for future research and enhancement.

Overall, disaster recovery readiness was improved, human error was reduced, and failover and failback processes were accelerated through the automation of Azure Site Recovery deployment and management with Infrastructure as Code and CI/CD pipelines.

7. Future Scope

In the future, this model can be improved by adding smart, AI-powered model that can plan, get approvals, communicate and perform disaster recovery test/drills automatically. Another improvement would be to create tools that can automatically find and fix any changes or "drift" in the Infrastructure as Code (IaC) setup after failback, so the recovery system always matches the production environment. The model could also be tested in hybrid and multi-cloud setups, where systems run across different cloud providers. Since Terraform already works with many cloud platforms, it could support cross-cloud recovery, cost optimization, and backup locations in different regions. Future work could also include adding better monitoring tools, automatic security and compliance checks, and stronger security measures. Testing this solution in industries like healthcare, finance, or critical infrastructure would also help prove its reliability, security, and ability to handle large-scale operations.

8. Conclusion

The goal of this study was to propose architecture, composition, and functional ideas of an automated disaster recovery system using Azure Site Recovery, Infrastructure as Code, and CI/CD pipelines to reduce cost, downtime and manual work.
The results showed that the automated model made the process much faster, cutting it from several days to just minutes or hours. It also allowed failover and failback to be done with just one click. Recovery Point Objectives (RPOs) of a few minutes and fast recovery times were achieved. Testing could also be done without affecting live systems and cost could also be reduced significantly due to not having to maintain standby recovery infrastructure.
These findings show that automated disaster recovery greatly improves speed, reliability, expense and readiness, and can be used by many companies that still rely on traditional manual methods.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1]HashiCorp. (2023). Terraform Enterprise documentation. https://developer.hashicorp.com/terraform/enterprise

[2]Microsoft. (2024). Visual Studio Code documentation. https://code.visualstudio.com/docs

[3]Michael Kaufmann; Thomas Dohmke; Donovan Brown, Accelerate DevOps with GitHub: Enhance software delivery performance with GitHub Issues, Projects, Actions, and Advanced Security , Packt Publishing, 2022.

[4]Hashicorp Terraform, “Multi-Cloud Provisioning with HashiCorp Terraform”, https://www.hashicorp.com/resources/enabling-multi-cloud-with-hashicorpterraform, 2018

[5]Akinbolaji, T. J., Nzeako, G., Akokodaripon, D., & Aderoju, A. V. (2024). Proactive monitoring and security in cloud infrastructure: Leveraging tools like Prometheus, Grafana, and HashiCorp Vault for robust DevOps practices. World Journal of Advanced Engineering Technology and Sciences, 13(2), 90–104.

[6]Microsoft. (2023). Azure Site Recovery documentation. https://learn.microsoft.com/en-us/azure/site-recovery/

[7]Elradi, M. D. (2023). Ansible: A reliable tool for automation. Electrical and Computer Engineering Studies, 2(1)

[8]Chan, P. (2022). Information technology disaster recovery planning (Order No. 29992753). ProQuest Dissertations & Theses Global. (2763544495). https://www.proquest.com/dissertations-theses/information-technology-disaster-recovery-planning/docview/2763544495/se-2

[9]Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (NIST SP 800-145). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-145

[10]Kavis, M. J. (2014). Architecting the cloud: Design decisions for cloud computing service models (SaaS, PaaS, and IaaS). Wiley.

[11]S. N., J. M., & H. V. (2025). Automatic cloud formation using LLM. 2025 International Conference on Intelligent and Cloud Computing (ICoICC) (pp. 1–6). IEEE. https://doi.org/10.1109/ICoICC64033.2025.11052114

[12]Saquib Z, Tyagi V, Bokare S, Dongawe S, Dwivedi M, Dwivedi J (2013). A new approach to disaster recovery as a service over cloud for database system. 2013 15th International Conference on Advanced Computing Technologies (ICACT), Rajampet, India.

[13]Abedallah Zaid Abualkishik, Ali A. Alwan and Yonis Gulzar, “Disaster Recovery in Cloud Computing Systems: An Overview” International Journal of Advanced Computer Science and Applications(IJACSA), 11(9), 2020. http://dx.doi.org/10.14569/IJACSA.2020.0110984