**Dummies Guide to Setup GHES on Azure with Terraform and Ansible for Automation and Management**

Contents

[**Dummies Guide to Setup GHES on Azure with Terraform and Ansible for Automation and Management** 1](#_Toc178082124)

[**1.** **Objective:** 2](#_Toc178082125)

[**2.** **Prerequisites** 2](#_Toc178082126)

[**2.1.** **Knowledge and Certifications** 2](#_Toc178082127)

[**2.2.** **Tools & Technologies** 2](#_Toc178082128)

[**3.** **System Requirements** 3](#_Toc178082129)

[**3.1.** **GHES Server Requirements (Minimal Setup for 2 Users)** 3](#_Toc178082130)

[**3.2.** **Azure CLI and Ansible Server Requirements (DevOps Local Machine)** 3](#_Toc178082131)

[**4.** **Azure Cost Estimation (Daily Basis)** 3](#_Toc178082132)

[**4.1.** **VM Instance (Standard DC4ds v3) for each server** 4](#_Toc178082133)

[**4.2.** **Storage (100 GB Attached) for each server** 4](#_Toc178082134)

[**4.3.** **Total Daily Cost Estimate:** 4](#_Toc178082135)

[**5.** **Workflow: Tools and Best Practices for Infrastructure Code Management** 4](#_Toc178082136)

[**5.1.** **Overview of the Workflow** 5](#_Toc178082137)

[**5.2.** **Best Practices for Infrastructure Code Management** 6](#_Toc178082138)

[**5.3.** **Final Recommendation for Storing Infrastructure Code** 7](#_Toc178082139)

[**6.** **Installation and Setup of Tools** 8](#_Toc178082140)

[**Included Tools:** 8](#_Toc178082141)

[**6.1.** **Set Up GitHub.com for Infrastructure Code Management** 8](#_Toc178082142)

[**6.2.** **Microsoft Azure Setup** 10](#_Toc178082143)

[**6.3.** **Set Up Microsoft Azure CLI** 11](#_Toc178082144)

[**6.4.** **Automating VM Shutdown and Startup with Azure Automation** 12](#_Toc178082145)

[**6.5.** **Install and Configure Terraform** 14](#_Toc178082146)

[**6.6.** **Install and Configure Ansible** 15](#_Toc178082147)

[**6.7.** **Set Up GitHub Enterprise Server (GHES)** 16](#_Toc178082148)

[**7. Step-by-Step Guide: Automating GHES Setup, High Availability, and Data Recovery** 16](#_Toc178082149)

[**7.1.** **Task 1: Primary and Replica Management** 17](#_Toc178082150)

[**7.2.** **Task 2: User Alerting System for Compliance** 22](#_Toc178082151)

[**7.3.** **Task 3: Test Environment Setup for GHES Upgrades** 23](#_Toc178082152)

[**7.4.** **Task 4: Georeplication Setup** 25](#_Toc178082153)

[**7.5.** **Task 5: Repository Restore** 27](#_Toc178082154)

[**8.** **Future Improvement and Expansion possibilities** 28](#_Toc178082155)

[**8.1.** **Improvement Possibilities** 28](#_Toc178082156)

[**8.2.** **Future Scope and Expansion:** 32](#_Toc178082157)

1. **Objective:**

To implement and automate various aspects of GitHub Enterprise Server (GHES) management on Azure using Terraform and Ansible, ensuring high availability, compliance, and data recovery.

1. **Prerequisites**
   1. **Knowledge and Certifications**

Before starting, ensure familiarity with the following:

* **Basic Linux knowledge** (commands, working with CLI)
  + [Linux Tutorials for Beginners](https://ubuntu.com/tutorials/command-line-for-beginners)
* **Basic GitHub Enterprise Server (GHES) knowledge**
  + [GitHub Enterprise Server Documentation (v3.14)](https://docs.github.com/en/enterprise-server@3.14/admin/overview/about-github-enterprise-server)
* **Azure Administration and Networking**
  + [Microsoft Azure Fundamentals](https://learn.microsoft.com/en-us/training/azure/)
* **Terraform basics**
  + [Terraform Intro](https://developer.hashicorp.com/terraform/intro)
* **Ansible basics**
  + [Ansible Documentation](https://docs.ansible.com/)

**Certification** recommendations (optional):

* **Azure Fundamentals (AZ-900)**
  + [Microsoft Learn: AZ-900](https://learn.microsoft.com/en-us/certifications/azure-fundamentals/)
  1. **Tools & Technologies**

Ensure that the following are installed on your system:

* [**GitHub Enterprise Server Trial Version**](https://docs.github.com/en/enterprise-server@3.9/admin/overview/setting-up-a-trial-of-github-enterprise-server)
* **Microsoft Azure Account**
  + [Azure Free Account Setup](https://azure.microsoft.com/en-us/free/)
  + [Azure Pricing Calculator](https://azure.microsoft.com/en-us/pricing/calculator/)
* **Terraform** (version >= 1.0.0)
  + [Download Terraform](https://developer.hashicorp.com/terraform/install)
* **Ansible** (version >= 2.9)
  + [Ansible Installation](https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html)
* **Azure CLI** (version >= 2.0.80)
  + [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli)
* **Visual Studio Code**
  + [Download VS Code](https://code.visualstudio.com/)
  + **Extensions**:
    - [HashiCorp Terraform](https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform)
    - [YAML support](https://marketplace.visualstudio.com/items?itemName=redhat.vscode-yaml)
    - [Ansible support](https://marketplace.visualstudio.com/items?itemName=redhat.ansible)

1. **System Requirements**
   1. **GHES Server Requirements (Minimal Setup for 2 Users)**

For the **Primary** and **Replica** GHES servers, the following minimal configuration is required for trial or demo purposes:

* **vCPUs**: 4 x86-64 (for each of primary and replica servers)
* **Memory**: 32 GB (for each of primary and replica servers)
* **Root Storage**: 200 GB (for each of primary and replica servers)
* **Attached Data Storage**: 150 GB (for each of primary and replica servers)
* **Operating System**: Ubuntu 20.04 LTS (or later)

For more detailed system requirements and further reference, you can review the official [GHES System Requirements](https://docs.github.com/en/enterprise-server@3.13/admin/installing-your-enterprise-server/setting-up-a-github-enterprise-server-instance/installing-github-enterprise-server-on-azure).

* 1. **Azure CLI and Ansible Server Requirements (DevOps Local Machine)**
* **For Developer Machine**:
  + **Memory**: Minimum 2GB RAM
  + **CPU**: 2 cores
  + **Storage**: 20GB
  + **Operating System**: Ubuntu, CentOS, macOS, or Windows

For more details:

* [Azure CLI System Requirements](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli)
* Terraform System Requirements
* Ansible System Requirements

1. **Azure Cost Estimation (Daily Basis)**

For cost estimation, we'll calculate for:

* **Two VM Instances**: Standard **DC4ds v3** (4 vCPUs, 32GB RAM) for **both the primary and replica servers**.
* **Storage**: 200 GB (root) + 150 GB (attached data storage) for **each server**.
* **Regions**: Germany West Central (chosen based on the location in Stuttgart and Munich).

**You can use** [**Azure VM Selector**](https://azure.microsoft.com/en-us/pricing/vm-selector/) **to find right VM series for the project. I have selected a General Purpose VM instead of Memory optimised VM which is recommended by GitHub.**

A black and white rectangular object

Description automatically generated

Using the [Azure Pricing Calculator](https://azure.microsoft.com/en-us/pricing/calculator/), here's a rough estimate for the cost on a daily basis:

* 1. **VM Instance (Standard DC4ds v3) for each server**
* Approximate cost per hour per server: €0.24
* Approximate daily cost per server: **€5.76** (24 hours of operation)

For two servers (Primary + Replica):

* Total daily VM cost: **€11.52/day**
  1. **Storage (100 GB Attached) for each server**
* Managed Disk Premium SSD v2: **€0.000117**
* Total Storage Cost per server: **0.002808 €/day**

For two servers (Primary + Replica):

* Total daily storage cost: **0.005616 €/day**
  1. **Total Daily Cost Estimate:**

**€11.52 (VM for two servers) + €0.005616 (Storage for two servers) = 11.525616€/day**

**This estimate assumes that both GHES servers will be running continuously. We could minimize costs by scheduling the VM to stop outside of working hours if needed. We can configure to turn off the VMs between 8 PM to 8 AM every day, and on weekends.**

Here is the link for the saved estimate at azure portal: <https://azure.com/e/d57374b792b14f579c56347a33e361e5> for 120hrs.

1. **Workflow: Tools and Best Practices for Infrastructure Code Management**

This section explains how the tools used in this setup (Terraform, Ansible, Azure CLI, and GHES) interact, and what the best practices are for storing infrastructure code.

* 1. **Overview of the Workflow**

The main goal of this workflow is to automate the deployment, management, and monitoring of **GitHub Enterprise Server (GHES)** instances on Azure, ensuring high availability and disaster recovery through the use of **Terraform** and **Ansible**. Here’s how each tool fits into the overall workflow:

1. **Terraform**:

* **Purpose**: Terraform is used for **Infrastructure as Code (IaC)**, meaning it defines and automates the provisioning of cloud resources such as VMs, networking, and storage in Azure.
* **Role in the Workflow**: Terraform provisions both the **primary** and **replica** GHES instances using the **Azure Marketplace** GHES image. It also configures related infrastructure, such as virtual networks and network interfaces.
* **Workflow**: You write your infrastructure configuration as Terraform code in the main.tf file. After verifying the configuration, you apply it to Azure using terraform apply, which creates all resources as specified.

1. **Ansible**:

* **Purpose**: Ansible is used for **configuration management** and **automation**, including tasks like health monitoring, failover automation, and user compliance monitoring on GHES.
* **Role in the Workflow**: Ansible is used post-deployment to automate failover between the GHES instances, conduct health checks, and ensure compliance by monitoring user profiles and repository usage.
* **Workflow**: After Terraform provisions the infrastructure, Ansible is used to configure and maintain the GHES instances. You create Ansible playbooks that are executed on the primary and replica GHES servers to ensure their health and automate tasks like user alerts and failovers.

1. **Azure CLI**:

* **Purpose**: Azure CLI provides a way to interface with Azure from the command line, allowing you to manage and monitor resources directly.
* **Role in the Workflow**: Azure CLI is used during setup to configure Azure resources such as the service principal and for any manual resource management that may be required.
* **Workflow**: You use the Azure CLI to interact with your Azure account, manage resources, and perform manual operations that may complement the automated Terraform processes.

1. **GHES**:

* **Purpose**: GitHub Enterprise Server is the core version control platform being deployed and managed in this setup. It ensures that repositories, users, and other version control-related services are available and reliable.
* **Role in the Workflow**: GHES instances are deployed on Azure, with the **primary** and **replica** configurations for high availability and disaster recovery.
* **Workflow**: After the infrastructure is set up using Terraform, the GHES instances are configured and managed through Ansible to ensure high availability, data replication, and compliance.
  1. **Best Practices for Infrastructure Code Management**

When managing Infrastructure as Code (IaC), storing your code, configurations, and automation scripts in a secure, version-controlled environment is critical. Here are the key considerations for this project:

1. **Separate Version Control System (VCS) vs. Using GHES for Infrastructure Code Storage**

* **Option 1: Use GitHub.com (Separate VCS)**
  + - **Pros**:
      * **Centralized and Global Access**: GitHub.com is a cloud-hosted VCS that is globally accessible and can act as a central repository for your infrastructure code.
      * **Availability and Redundancy**: Since GitHub.com is highly available and managed, it ensures that your infrastructure code is always available, even if your GHES servers are down.
      * **Integration with CI/CD Tools**: GitHub.com offers seamless integration with other automation and CI/CD tools like GitHub Actions.
    - **Cons**:
      * Requires external hosting if there are strict on-premise requirements for all code and data.

**Recommendation**: **GitHub.com** is generally the best option for storing your infrastructure code. It provides robust features for collaboration, security (two-factor authentication, access controls), and version control. It also avoids the risk of hosting your infrastructure code on the very GHES servers that manage your other repositories, which could cause issues if those servers become unavailable.

* **Option 2: Use GHES (Same Server)**
  + - **Pros**:
      * **On-Premise Control**: Storing the infrastructure code directly on GHES keeps everything on-premise, which might be a requirement for some organizations with strict compliance and security policies.
    - **Cons**:
      * **Availability Risk**: If the GHES server goes down (either primary or replica), you may not be able to access the very infrastructure code you need to restore or fix the system.
      * **Redundancy Issues**: GHES may not provide the same level of redundancy and global access that GitHub.com can offer.

**Recommendation**: While you can store the infrastructure code on the GHES servers you are creating, this setup is not recommended due to potential availability issues. If your GHES infrastructure has any downtime, your infrastructure code could become inaccessible, complicating recovery and troubleshooting.

1. **Best Practices for Storing Infrastructure Code**

* **Use a Separate Repository**: Your Terraform, Ansible playbooks, and automation scripts should be stored in a dedicated repository, separate from your application code. This provides clarity, security, and ease of management.
* **Version Control**: Ensure that all infrastructure code is version-controlled to track changes, collaborate, and revert to previous states if needed.
* **Git Ignore**: Use a .gitignore file to exclude sensitive information like Terraform state files and secrets from being versioned. Secrets should be stored securely in a tool like Azure Key Vault or HashiCorp Vault.

**Example .gitignore File**:

# Ignore Terraform state files

\*.tfstate

\*.tfstate.backup

# Ignore Ansible cache and logs

.ansible/

# Ignore secrets or sensitive files

secrets.yml

* 1. **Final Recommendation for Storing Infrastructure Code**

Based on the above analysis, **GitHub.com** is the most robust and secure option for storing your infrastructure code. It ensures high availability, global access, and integrates well with CI/CD workflows. Using GHES for hosting infrastructure code is not recommended, as it creates a dependency on the very servers you are provisioning and managing.

1. **Installation and Setup of Tools**

This section covers the installation and setup of all the tools necessary for deploying and managing GitHub Enterprise Server (GHES) on Azure. It includes setting up **GitHub.com** for Infrastructure as Code (IaC), along with Azure CLI, Terraform, Ansible, and GHES itself.

**Included Tools:**

* **GitHub.com for Infrastructure as Code (IaC)**
* **Microsoft Azure Setup**
* **Microsoft Azure CLI**
* **Terraform**
* **Ansible**
* **GitHub Enterprise Server (GHES)**
  1. **Set Up GitHub.com for Infrastructure Code Management**

Since **GitHub.com** is the recommended version control system for managing Infrastructure as Code (IaC), here's how you can set it up:

**6.1.1. Create a GitHub Account (if you don’t have one already)**

* Go to [GitHub.com](https://github.com) and sign up for an account if you don't already have one.

**6.1.2. Create a New Repository for Infrastructure Code**

* Navigate to **Repositories** and click on **New**.
* Create a repository specifically for your infrastructure code (e.g., ghes-infra).
* Set the repository to **private** if you want to restrict access.

**6.1.3. Clone the Repository**

* Clone your newly created GitHub repository to your local machine, where you’ll store your Terraform and Ansible configurations:

git clone https://github.com/yourusername/ghes-infra.git

cd ghes-infra

**6.1.4. Folder Structure for the Infrastructure Code**

Here’s the recommended folder structure for organizing your Infrastructure as Code (IaC), including Terraform configurations, Ansible playbooks, and any scripts related to GHES automation.

/ghes-infra # Root directory of the infrastructure project

├── /terraform # Contains all Terraform-related configurations

│ ├── main.tf # Main Terraform configuration file

│ ├── variables.tf # File defining variables used in Terraform

│ ├── output.tf # Output file for defining outputs from Terraform

│ └── terraform.tfstate # Terraform state file (optional, should be gitignored)

│

├── /ansible # Contains all Ansible playbooks and related configurations

│ ├── inventory.yml # Inventory file defining hosts (primary, replica)

│ ├── failover\_ghes.yml # Playbook for automating failover between primary and replica

│ ├── user\_alerts.yml # Playbook for user compliance monitoring (e.g., email alerts)

│ └── ansible.cfg # Configuration file for Ansible behavior

│

├── /scripts # Any scripts for automating GHES-related tasks (optional)

│ ├── backup\_restore.sh # Script for backing up and restoring GHES repositories

│

├── .gitignore # File to exclude sensitive or unnecessary files from Git versioning

└── README.md # Documentation for the project (can be added later)

**6.1.5. Explanation of Each Folder and File**

1. **Root Directory /ghes-infra**:

* This is the main directory for your infrastructure project, containing all Terraform, Ansible, and script files necessary to deploy and manage GHES on Azure.

1. **Terraform Directory /terraform**:

* Contains all Terraform configuration files that define the Azure infrastructure, including virtual machines, networking, and storage for the GHES primary and replica servers.
* **Key Files**:
  + - **main.tf**: The main Terraform file that defines the infrastructure.
    - **variables.tf**: This file defines the variables used in the Terraform configuration (e.g., resource names, VM sizes).
    - **output.tf**: Defines any outputs from the Terraform process (e.g., public IPs of VMs).
    - **terraform.tfstate**: Stores the current state of the infrastructure. This file should be added to .gitignore to prevent it from being versioned.

1. **Ansible Directory /ansible**:

* This directory contains all the Ansible playbooks used to automate tasks on the GHES instances.
* **Key Files**:
  + - **inventory.yml**: Defines the GHES hosts (primary and replica) where the playbooks will be run.
    - **failover\_ghes.yml**: Playbook that automates failover between the primary and replica GHES instances.
    - **user\_alerts.yml**: Playbook for monitoring user compliance and sending alerts for private email usage.
    - **ansible.cfg**: Optional configuration file for customizing Ansible's behavior.

1. **Scripts Directory /scripts** (optional):

* This directory can contain any additional bash or Python scripts used to manage or automate GHES tasks, such as backups or data restoration.
* **Key Files**:
  + - **backup\_restore.sh**: Script for backing up and restoring GHES repositories using ghe-backup and ghe-restore commands.

1. **.gitignore**:

* The .gitignore file is essential to exclude files that shouldn't be versioned (e.g., Terraform state files, Ansible logs, secrets).
* **Key Entries**:

# Ignore Terraform state files

\*.tfstate

\*.tfstate.backup

# Ignore Ansible cache and logs

.ansible/

# Ignore secrets or sensitive files

secrets.yml

**6.1.6. Push the Infrastructure Code to GitHub**

* Once the directory structure and code are ready, push the changes to GitHub:

git add .

git commit -m "Initial infrastructure setup with Terraform and Ansible"

git push origin main

This setup allows you to manage your infrastructure code (Terraform, Ansible, and scripts) in a version-controlled manner using <GitHub.com> .

* 1. **Microsoft Azure Setup**

Before configuring Azure resources with **Terraform**, you need to perform some minimal setup in Azure:

**6.2.1: Create an Azure Account**

If you don’t already have an Azure account, you can sign up for a free account or use a paid one:

* [Azure Free Account Setup](https://azure.microsoft.com/en-us/free/)
* [Azure Pricing Information](https://azure.microsoft.com/en-us/pricing/)

**6.2.2. Create an Azure Resource Group (Optional)**

A **Resource Group** is a container for resources like VMs, storage accounts, and networks. While Terraform can create this resource group, you can also create one manually if you want better control.

1. **Create a Resource Group Using Azure CLI**: You can create a resource group using the Azure CLI if you prefer doing this step manually:

az group create --name ghes-resource-group --location "West Europe"

1. **Create Resource Group via Azure Portal**: Alternatively, use the Azure Portal:

* Go to the **Azure Portal**: [portal.azure.com](https://portal.azure.com)
* Navigate to **Resource Groups** and click **+ Add**.
* Enter the details and click **Create**.

**6.2.3. Create a Service Principal for Terraform Authentication**

A **Service Principal** is required to authenticate Terraform with Azure. The service principal provides programmatic access to your Azure resources.

1. **Create a Service Principal**: Use the Azure CLI to create a new service principal for Terraform:

az ad sp create-for-rbac --name "terraform-ghes" --role="Contributor" --scopes="/subscriptions/<your-subscription-id>"

This command returns a JSON object containing the following details that you need for Terraform authentication:

* **AppId** (Client ID)
* **Password** (Client Secret)
* **Tenant**
* **Subscription ID**

Store these values securely, as you will use them later in your Terraform configuration.

1. **Set Role-Based Access Control (RBAC)**: Make sure the service principal has the right role (e.g., Contributor) to manage your Azure resources.
2. **Verify the Service Principal**: After creating the service principal, you can verify it using the following command:

az ad sp show --id <app-id>

**6.2.4. Ensure Azure Subscriptions are Configured Correctly**

If you have multiple Azure subscriptions, ensure that the one you intend to use is set as the default:

az account list --output table

az account set --subscription "your-subscription-id"

Once these minimal Azure setup steps are completed, you can move forward with the **Terraform setup** to automate the deployment of GHES.

* 1. **Set Up Microsoft Azure CLI**

The Azure CLI will allow you to manage Azure resources from the command line.

**6.3.1. Install Azure CLI**: Follow the installation instructions based on your operating system:

* [Install Azure CLI](https://learn.microsoft.com/en-us/cli/azure/install-azure-cli)

**For Ubuntu**:

curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

**For macOS**:

brew update && brew install azure-cli

**For Windows**: Download and install the Azure CLI by following these instructions.

* + 1. **Log in to Azure** using the Azure CLI: After installation, log in to Azure using the following command:

az login

This will open a browser window for you to authenticate. Once logged in, the Azure CLI will link to your account.

* + 1. **Set the Active Subscription**: If you have multiple Azure subscriptions, select the one you wish to use:

az account set --subscription "your-subscription-id"

* + 1. **Validate Installation**: Confirm that Azure CLI is correctly installed by checking its version:

az –version

* 1. **Automating VM Shutdown and Startup with Azure Automation**

In this section, we will set up automated VM shutdown and startup schedules using **Azure Automation**. The goal is to achieve:

* **Monday to Friday**: Shut down VMs from **8 PM to 8 AM**.
* **Weekend**: Complete shutdown from **Friday 8 PM** to **Monday 8 AM**.

This will help reduce costs by ensuring that VMs are only running during working hours and shut down during non-working hours and weekends.

* + 1. **Prerequisites:**

**Azure Automation Account:** You will need an Azure Automation account to create and manage Runbooks for automating VM operations.

[Azure Automation Documentation](https://docs.microsoft.com/en-us/azure/automation/automation-intro)

* + 1. **Step 1: Create an Azure Automation Account**
* **Go to the Azure Portal** and search for "Automation Accounts."
* Click **Create** and fill out the necessary information:

Name: ghes-auto-scheduler

* **Resource Group**: Use the same resource group as your VMs.
  + - **Location**: West Europe (or your preferred region).
    - Click **Review + Create** and then **Create**.
    1. **Step 2: Create a Runbook for VM Shutdown**
    - In your **Automation Account**, go to **Runbooks**.
    - Click **+ Create a Runbook**.
    - Fill out the following:
      * **Name**: shutdown-vms
      * **Runbook Type**: PowerShell
      * **Description**: Script to shut down all VMs in the resource group.
* Paste the following PowerShell script:

# PowerShell script for stopping VMs

$ResourceGroupName = "ghes-resource-group"

# Authenticate to Azure

Connect-AzAccount

# Get all VMs in the resource group

$vms = Get-AzVM -ResourceGroupName $ResourceGroupName

# Stop each VM

foreach ($vm in $vms) {

Stop-AzVM -ResourceGroupName $vm.ResourceGroupName -Name $vm.Name -Force

}

* Click **Save** and **Publish** the Runbook.
  + 1. **Step 3: Create a Runbook for VM Startup**
* In the **Automation Account**, go to **Runbooks**.
* Click **+ Create a Runbook**.
* Fill out the following:
* **Name**: start-vms
* **Runbook Type**: PowerShell
* **Description**: Script to start all VMs in the resource group.
* Paste the following PowerShell script:

# PowerShell script for starting VMs

$ResourceGroupName = "ghes-resource-group"

# Authenticate to Azure

Connect-AzAccount

# Get all VMs in the resource group

$vms = Get-AzVM -ResourceGroupName $ResourceGroupName

# Start each VM

foreach ($vm in $vms) {

Start-AzVM -ResourceGroupName $vm.ResourceGroupName -Name $vm.Name

}

* Click **Save** and **Publish** the Runbook.
  + 1. **Step 4: Create Schedules for Weekdays and Weekends**

You will now create schedules that handle the different shutdown and startup times for weekdays and weekends.

* + - 1. **Create a Schedule for Weekdays**:

Go to the **Automation Account** > **Schedules** > **+ Add a Schedule**.

* **Name**: weekday-shutdown-schedule
* **Recurrence**: Daily (Monday through Friday).
* **Start Time**: Set the time to **8 PM** CET (or your local time).
* **Time Zone**: Central European Time (or your preferred timezone).
  + - 1. **Create a corresponding schedule for starting VMs**:
* **Name**: weekday-start-schedule
* **Recurrence**: Daily (Monday through Friday).
* **Start Time**: Set the time to **8 AM** CET.
  + - 1. **Create a Schedule for Weekends**:
* Create a shutdown schedule for **Friday at 8 PM** (same process as above).
* Create a startup schedule for **Monday at 8 AM**.
  1. **Install and Configure Terraform**

Terraform is used to automate the deployment of Azure resources, including the GitHub Enterprise Server.

* + 1. **Install Terraform**: Follow the official Terraform installation instructions for your operating system.
* [Terraform Installation Guide](https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli)

**For Ubuntu**:

sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl

curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -

sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb\_release -cs) main"

sudo apt-get update && sudo apt-get install terraform

**For macOS**:

brew tap hashicorp/tap

brew install hashicorp/tap/terraform

**For Windows**: Download the Terraform binary from Terraform Download. Extract the binary and add it to your system's PATH.

* + 1. **Set Terraform Environment Variables**: After installing Terraform, set up the necessary environment variables for authentication with Azure:

**For Ubuntu/macOS**: Add the following to your .bashrc or .zshrc file:

export ARM\_CLIENT\_ID="your-client-id"

export ARM\_CLIENT\_SECRET="your-client-secret"

export ARM\_SUBSCRIPTION\_ID="your-subscription-id"

export ARM\_TENANT\_ID="your-tenant-id"

**For Windows**: You can set environment variables via the command line or through the system settings.

Command line example:

setx ARM\_CLIENT\_ID "your-client-id"

setx ARM\_CLIENT\_SECRET "your-client-secret"

setx ARM\_SUBSCRIPTION\_ID "your-subscription-id"

setx ARM\_TENANT\_ID "your-tenant-id"

* + 1. **Initialize Terraform**: In your project directory, initialize Terraform to download the necessary providers:

terraform init

* + 1. **Create Terraform Configuration**: Create a directory for your Terraform files and define the infrastructure in main.tf. As covered earlier, you’ll configure Azure resources like Virtual Machines (using the **GitHub Enterprise Server image** from the Azure Marketplace).
    2. **Plan and Apply Terraform Configuration**:
       1. **Plan the infrastructure changes:**

terraform plan

* + - 1. **Apply the configuration to provision resources:**

terraform apply

* 1. **Install and Configure Ansible**

Ansible is used to manage configuration and automation tasks on the GitHub Enterprise Server instances.

* + 1. **Install Ansible: Follow the official Ansible installation guide based on your OS:**
* [Ansible Installation Guide](https://docs.ansible.com/ansible/latest/installation_guide/index.html)

**For Ubuntu**:

sudo apt update

sudo apt install ansible -y

**For macOS**:

brew install ansible

**For Windows**: Use **WSL (Windows Subsystem for Linux)** and install Ansible within the WSL environment:

sudo apt update

sudo apt install ansible -y

* + 1. **Configure Ansible for GHES Automation:** Create an inventory file for the GHES primary and replica servers, and set up your playbooks for tasks like health checks and failovers.

**Inventory file (inventory.yml)**:

all:

hosts:

primary:

ansible\_host: <primary-ghes-ip>

ansible\_user: azureuser

replica:

ansible\_host: <replica-ghes-ip>

ansible\_user: azureuser

* + 1. **Run Ansible Playbooks**: Run your playbooks using the following command:

ansible-playbook -i inventory.yml failover\_ghes.yml

* 1. **Set Up GitHub Enterprise Server (GHES)**

Once Terraform provisions the VMs, connect to the VMs and follow the setup process for GitHub Enterprise Server.

* + 1. **Connect to the VMs**: SSH into the VMs:

ssh azureuser@<vm-public-ip>

* + 1. **Complete the GHES Setup**: Follow the steps in the official guide to complete the GitHub Enterprise Server setup:
* [Setting up GitHub Enterprise Server on Azure](https://docs.github.com/en/enterprise-server@3.10/admin/installing-your-enterprise-server/setting-up-a-github-enterprise-server-instance/installing-github-enterprise-server-on-azure)

**7. Step-by-Step Guide: Automating GHES Setup, High Availability, and Data Recovery**

This section provides detailed steps for each task to automate the GitHub Enterprise Server (GHES) setup, ensure high availability, monitor user compliance, and manage data recovery using Terraform and Ansible.

* 1. **Task 1: Primary and Replica Management**

**Objective: Ensure high availability of GHES instances by setting up a primary and replica server with automated failover.**

**Step 1: Provision GHES Primary and Replica Servers Using Terraform**

1. **Create the Terraform Configuration**:

* The main.tf file should define the infrastructure for both the primary and replica GHES instances. It will use the **GitHub Enterprise Server** image from the **Azure Marketplace**.

**Example main.tf**:

provider "azurerm" {

  features {}

  # Configures the Azure Resource Manager (azurerm) provider, required to interact with Azure resources.

}

resource "azurerm\_resource\_group" "ghes\_rg" {

  name     = "ghes-resource-group"    # The name of the resource group that will contain all your Azure resources.

  location = "West Europe"            # Azure region where the resource group will be created (e.g., West Europe).

}

resource "azurerm\_virtual\_network" "vnet" {

  name                = "ghes-vnet"                      # Name of the virtual network (VNet) where your VMs will be placed.

  address\_space       = ["10.0.0.0/16"]                  # IP range (CIDR block) for the VNet, defining the internal network space.

  location            = azurerm\_resource\_group.ghes\_rg.location  # Location for the VNet (same as resource group location).

  resource\_group\_name = azurerm\_resource\_group.ghes\_rg.name      # Associates the VNet with the resource group created above.

}

resource "azurerm\_subnet" "subnet" {

  name                 = "ghes-subnet"                   # Name of the subnet within the VNet where the VMs will be located.

  resource\_group\_name  = azurerm\_resource\_group.ghes\_rg.name   # Associates the subnet with the resource group.

  virtual\_network\_name = azurerm\_virtual\_network.vnet.name      # Associates the subnet with the virtual network created above.

  address\_prefixes     = ["10.0.1.0/24"]                 # CIDR block defining the address range for the subnet.

}

resource "azurerm\_virtual\_machine" "primary\_vm" {

  name                  = "ghes-primary-vm"               # Name of the primary virtual machine (VM) that will run GHES.

  location              = azurerm\_resource\_group.ghes\_rg.location  # Specifies the region where the VM will be deployed.

  resource\_group\_name   = azurerm\_resource\_group.ghes\_rg.name      # Associates the VM with the resource group created earlier.

  network\_interface\_ids = [azurerm\_network\_interface.primary\_nic.id]  # References the network interface (NIC) for the VM.

  vm\_size               = "Standard\_DC4ds\_v3"              # Specifies the VM size (4 vCPUs, 32GB RAM).

  storage\_image\_reference {                               # Specifies the Azure Marketplace image to use for the VM.

    publisher = "GitHub"                                  # The publisher of the GHES image.

    offer     = "GitHub-Enterprise"                       # The offer name (in this case, GitHub Enterprise).

    sku       = "GitHub-Enterprise-Server"                # The SKU for GHES.

    version   = "latest"                                  # The latest version of the GHES image will be used.

  }

  storage\_os\_disk {                                       # Configuration for the operating system disk of the VM.

    name              = "primary\_os\_disk"                 # Name of the OS disk.

    caching           = "ReadWrite"                       # Disk caching mode.

    create\_option     = "FromImage"                       # The disk is created from the specified image.

    managed\_disk\_type = "Standard\_LRS"                    # Managed disk type (Standard locally redundant storage).

  }

  os\_profile {                                            # Defines the operating system settings for the VM.

    computer\_name  = "ghes-primary"                       # Hostname for the VM.

    admin\_username = "azureuser"                          # The admin username for the VM.

    admin\_password = "Password123!"                       # Password for the admin user.

  }

  os\_profile\_linux\_config {                               # Additional Linux OS configuration.

    disable\_password\_authentication = false               # Password authentication is enabled.

  }

  tags = {                                                # Tags can help with resource management and billing.

    environment = "production"                            # Tag for identifying the environment.

  }

}

resource "azurerm\_virtual\_machine" "replica\_vm" {

  name                  = "ghes-replica-vm"               # Name of the replica virtual machine for GHES.

  location              = "Germany West Central"          # Region where the replica VM will be deployed.

  resource\_group\_name   = azurerm\_resource\_group.ghes\_rg.name  # Associates the replica VM with the same resource group.

  network\_interface\_ids = [azurerm\_network\_interface.replica\_nic.id]  # References the network interface for the replica VM.

  vm\_size               = "Standard\_DC4ds\_v3"             # Specifies the VM size for the replica (same as primary).

  storage\_image\_reference {                               # Specifies the GHES image from the Azure Marketplace.

    publisher = "GitHub"                                  # The publisher of the GHES image.

    offer     = "GitHub-Enterprise"                       # The offer name (in this case, GitHub Enterprise).

    sku       = "GitHub-Enterprise-Server"                # The SKU for GHES.

    version   = "latest"                                  # The latest version of the GHES image will be used.

  }

  storage\_os\_disk {                                       # Configuration for the replica VM's OS disk.

    name              = "replica\_os\_disk"                 # Name of the OS disk for the replica VM.

    caching           = "ReadWrite"                       # Disk caching mode.

    create\_option     = "FromImage"                       # The disk is created from the specified image.

    managed\_disk\_type = "Standard\_LRS"                    # Managed disk type (Standard locally redundant storage).

  }

  os\_profile {                                            # OS profile for the replica VM.

    computer\_name  = "ghes-replica"                       # Hostname for the replica VM.

    admin\_username = "azureuser"                          # Admin username.

    admin\_password = "Password123!"                       # Password for the admin user.

  }

  os\_profile\_linux\_config {                               # Additional Linux OS configuration.

    disable\_password\_authentication = false               # Password authentication is enabled.

  }

  tags = {                                                # Tags for the replica VM.

    environment = "replica"                               # Identifies this VM as the replica in the environment.

  }

}

# DevTest Lab resource to manage VM auto-shutdown and auto-start

resource "azurerm\_dev\_test\_lab" "ghes\_lab" {

  name                = "ghes-auto-lab"                   # Name of the DevTest Lab, used for setting up schedules for auto-shutdown/start.

  location            = azurerm\_resource\_group.ghes\_rg.location   # Location for the DevTest Lab (same as resource group location).

  resource\_group\_name = azurerm\_resource\_group.ghes\_rg.name        # Associates the DevTest Lab with the resource group.

}

# Schedule for auto-shutdown at 8 PM CET every day

resource "azurerm\_dev\_test\_schedule" "ghes\_shutdown\_schedule" {

  name                = "auto-shutdown"                   # Name of the auto-shutdown schedule.

  location            = azurerm\_resource\_group.ghes\_rg.location    # Location for the schedule.

  resource\_group\_name = azurerm\_resource\_group.ghes\_rg.name         # Associates the schedule with the resource group.

  lab\_name            = azurerm\_dev\_test\_lab.ghes\_lab.name          # Associates the schedule with the DevTest Lab.

  task\_type           = "Shutdown"                        # Type of task - in this case, shutting down the VM.

  status              = "Enabled"                         # Enables the schedule.

  daily\_recurrence {

    time               = "2000"                            # Shutdown time at 8 PM CET (24-hour format, no colon).

  }

  time\_zone\_id        = "Central European Standard Time"  # Time zone set to CET.

  notification\_settings {

    status = "Disabled"                                   # Disables notifications (you can enable if needed).

  }

}

# Schedule for auto-start at 8 AM CET every day

resource "azurerm\_dev\_test\_schedule" "ghes\_start\_schedule" {

  name                = "auto-start"                      # Name of the auto-start schedule.

  location            = azurerm\_resource\_group.ghes\_rg.location    # Location for the schedule.

  resource\_group\_name = azurerm\_resource\_group.ghes\_rg.name         # Associates the schedule with the resource group.

  lab\_name            = azurerm\_dev\_test\_lab.ghes\_lab.name          # Associates the schedule with the DevTest Lab.

  task\_type           = "Start"                           # Type of task - starting the VM.

  status              = "Enabled"                         # Enables the schedule.

  daily\_recurrence {

    time               = "0800"                            # Start time at 8 AM CET (24-hour format, no colon).

  }

  time\_zone\_id        = "Central European Standard Time"  # Time zone set to CET.

  notification\_settings {

    status = "Disabled"                                   # Disables notifications (you can enable if needed).

  }

}

**Initialize Terraform**: Navigate to your Terraform project directory and run:

terraform init

1. **Validate and Plan the Terraform Configuration**: Ensure there are no errors and preview the infrastructure that Terraform will create:

terraform validate

terraform plan

1. **Apply the Configuration**: Provision the infrastructure:

terraform apply

**Step 2: Configure Ansible for Health Checks and Failover**

1. **Create Ansible Playbook for Health Monitoring and Failover**: After the infrastructure is deployed, use Ansible to automate health checks and failover.

**Example Ansible Playbook (failover\_ghes.yml)**:

- name: GHES Failover Automation

hosts: primary

tasks:

- name: Check GHES status on Primary

shell: ghe-check

register: ghes\_status

ignore\_errors: yes

- name: Promote Replica if Primary is Down

shell: ghe-repl-promote

when: ghes\_status.rc != 0

- name: Send Email Notification for Failover

mail:

host: smtp.example.com

port: 587

username: "alert@example.com"

password: "password"

to: "admin@example.com"

subject: "GHES Failover Triggered"

body: "The primary GHES instance is down. The replica has been promoted."

1. **Run the Ansible Playbook**: Use the following command to run the playbook:

ansible-playbook -i inventory.yml failover\_ghes.yml

**Step 3: Validate the High Availability Setup**

1. **Test Failover**:

* Manually stop the GHES service on the primary server to trigger the failover:

sudo systemctl stop ghes

* Verify that the replica has been promoted by checking its status using the command:

ghe-repl-status

1. **Review Logs**:

* Check logs to verify failover has been executed correctly:

sudo journalctl -u ghes

* 1. **Task 2: User Alerting System for Compliance**

**Objective: Monitor user profiles for compliance with company policies (e.g., no private email addresses).**

**Step 1: Implement User Compliance Alerting**

1. **Create Ansible Playbook to Detect Non-Compliant Emails**: Use the **GHES API** to detect non-compliant user profiles and log results.

**Example Playbook (user\_alerts.yml)**:

- hosts: ghes

tasks:

- name: Fetch GitHub Enterprise users

uri:

url: "https://ghes.example.com/api/v3/users"

method: GET

headers:

Authorization: "token {{ github\_token }}"

register: users

- name: Check for non-compliant emails

debug:

msg: "User {{ item.login }} has a private email address."

loop: "{{ users.json }}"

when: "'@privateemail.com' in item.email"

- name: Log non-compliant users

lineinfile:

path: /var/log/non\_compliant\_users.log

line: "User {{ item.login }} has private email {{ item.email }}"

loop: "{{ users.json }}"

when: "'@privateemail.com' in item.email"

1. **Run the Ansible Playbook**:

ansible-playbook -i inventory.yml user\_alerts.yml

* 1. **Task 3: Test Environment Setup for GHES Upgrades**

**Objective: Validate the functionality of GitHub Enterprise Server (GHES) after upgrades by creating a separate test environment on Azure and performing key tests, such as git operations and API calls.**

**Step 1: Create a Test Environment Using Terraform**

1. **Duplicate the Terraform Configuration**:

* To create a test environment without affecting production, duplicate your existing Terraform configuration files (e.g., main.tf) and modify them to deploy resources in a different region or resource group.

**Example of Changes to the main.tf for Test Environment**:

* Change the **resource group** and **VM names** for the test environment:

resource "azurerm\_resource\_group" "ghes\_rg\_test" {

name = "ghes-resource-group-test"

location = "East US"

}

resource "azurerm\_virtual\_machine" "primary\_vm\_test" {

name = "ghes-primary-vm-test"

resource\_group\_name = azurerm\_resource\_group.ghes\_rg\_test.name

location = azurerm\_resource\_group.ghes\_rg\_test.location

...

}

resource "azurerm\_virtual\_machine" "replica\_vm\_test" {

name = "ghes-replica-vm-test"

resource\_group\_name = azurerm\_resource\_group.ghes\_rg\_test.name

location = "West US"

...

}

1. **Deploy the Test Environment**:

* Navigate to the directory containing the Terraform files for the test environment.
* Initialize and apply the Terraform configuration:

terraform init

terraform apply

1. **Set Up GHES in the Test Environment**:

* Once the VMs are created, follow the same GHES setup process for both the primary and replica servers as you would for production (as described in **Step 6: GHES Setup**). This will create a fully functional test environment.

**Step 2: Perform Tests After GHES Upgrades**

1. **Upgrade GHES in the Test Environment**:

* If testing an upgrade, follow the [GHES upgrade documentation](https://docs.github.com/en/enterprise-server@3.10/admin/upgrade/upgrading-github-enterprise-server) to upgrade the test instance to the desired version.

1. **Test Core Git Operations**:

* Once the upgrade is complete, validate basic git operations on the test GHES instance.
* **Test Cloning a Repository**:

git clone https://<test-ghes-url>/<user>/<repository>.git

* **Test Pushing Changes to the Repository**:

git add .

git commit -m "Test commit after GHES upgrade"

git push origin main

1. **Test API Calls**:

* Test if the GHES API is working as expected after the upgrade.
* For example, list the users using the API:

curl -H "Authorization: token <your-access-token>" https://<test-ghes-url>/api/v3/users

1. **Monitor Logs for Errors**:

* After the upgrade, monitor GHES logs to ensure that the system is functioning properly and there are no errors:

sudo journalctl -u ghes

**Step 3: Document the Test Cases and Results**

1. **Create a List of Test Cases**:

* Document the test cases performed in the test environment, including git operations (clone, push, pull), API calls, and other important functionalities like user management and repository operations.

1. **Capture the Results**:

* Record the results of each test, noting any issues, errors, or differences from expected behavior.
* Use this documentation as a reference when upgrading the production GHES instance.
  1. **Task 4: Georeplication Setup**

**Objective: Ensure data redundancy and availability across different geographic locations by setting up geo-replication between GHES instances in two or more regions.**

**Step 1: Configure Geo-Replication for GHES**

1. **Set Up Two Geo-Replicated GHES Instances**:

* You will need two GHES instances in different geographic regions to enable geo-replication.
* If you haven't already done so, you can provision these instances using Terraform. Ensure that they are deployed in different Azure regions (e.g., **West Europe** and **Germany West Central**).

1. **Prepare the Primary GHES for Geo-Replication**:

* On the **primary** GHES instance, you need to configure geo-replication using the ghe-repl-setup command.
* **SSH into the Primary Server**:

ssh azureuser@<primary-ghes-ip>

* **Run the Geo-Replication Setup Command**:

ghe-repl-setup --primary

1. **Configure the Replica GHES Instance**:

* On the **replica** GHES instance, run the following command to configure it as a replica and start receiving replicated data from the primary:

ghe-repl-setup --replica --primary-host <primary-ghes-ip>

1. **Verify Geo-Replication Configuration**:

* Use the following command to verify that replication has been set up correctly and that the replica server is receiving data from the primary:

ghe-repl-status

**Step 2: Test Repository Replication Between Primary and Replica**

1. **Create a Test Repository on the Primary GHES**:

* Create a repository on the **primary** GHES instance and push some changes to it:

git init test-repo

git add .

git commit -m "Initial commit"

git push origin main

1. **Verify the Repository Replication**:

* Check the **replica** GHES instance to ensure that the newly created repository has been replicated.
* Use the following command on the replica server:

ghe-repl-status

1. **Perform a Test Failover** (Optional):

* You can test the failover between primary and replica by manually stopping the GHES services on the primary instance:

sudo systemctl stop ghes

* Then promote the replica to primary:

ghe-repl-promote

1. **Switch Back to the Original Primary**:

* After verifying replication and failover, switch back to the original primary by demoting the replica:

ghe-repl-demote

**Step 3: Document the Replication Setup and Testing Process**

1. **Create a Replication Setup Document**:

* Document the steps taken to set up geo-replication, including the commands used and any specific configurations.

1. **Record the Test Results**:

* Document the results of the repository replication and failover testing. Ensure that there were no data discrepancies and that the replication process is working as expected.
  1. **Task 5: Repository Restore**

**Objective: Ensure data recovery capabilities by automating repository backups and restoring them from backup snapshots.**

**Step 1: Use the Backup and Restore Script**

To automate repository backup and restoration processes, we’ll use the **backup\_restore.sh** script, which you can find in the /scripts folder.

1. **Backup GitHub Enterprise Repositories**:

* To back up all GHES repositories, run the script with the backup option:

./scripts/backup\_restore.sh backup

* This will execute the **ghe-backup** command, which saves a snapshot of all repositories to the defined BACKUP\_PATH in the script.

1. **Restore a Specific Repository**:

* To restore a specific repository from a backup snapshot, use the restore option followed by the repository name:

./scripts/backup\_restore.sh restore <repository\_name>

* This command restores the selected repository using the **ghe-restore** command from the backup directory defined in the script (RESTORE\_PATH).

**Step 2: Validate the Restore**

1. **Check the Repository Availability**:

* After the restoration process is complete, ensure that the restored repository is accessible via the GitHub Enterprise Server web interface and through git operations (e.g., cloning, pulling, pushing).

1. **Review Logs**:

* You can review the GHES logs to confirm that the backup and restoration processes were completed without errors:

sudo journalctl -u ghes

By integrating the **backup and restore script**, you automate the process of both backing up and restoring GHES repositories, making it easier to manage data recovery in the event of any issues with the GitHub Enterprise Server.

1. **Future Improvement and Expansion possibilities**

To make the project more efficient and fully automated, there are several key improvements and expansions you can focus on. These will streamline the GitHub Enterprise Server (GHES) management, optimize infrastructure costs, and allow for future scalability and enhancements.

* 1. **Improvement Possibilities**

Here’s a detailed approach to making the project more efficient and scalable:

* + 1. **Full CI/CD Pipeline for Infrastructure**
* **Automation using Terraform & GitHub Actions**: You can set up a Continuous Integration/Continuous Deployment (CI/CD) pipeline to automate infrastructure provisioning and management using **GitHub Actions**. This will allow you to:
  + Automatically apply infrastructure changes when Terraform configurations are updated in the repository.
  + Automatically validate infrastructure as code (e.g., running terraform plan and terraform apply) whenever changes are pushed to your repository.
  + Implement **GitOps** best practices by deploying infrastructure changes through code stored in your version control system (GitHub.com or GHES).
* **Automation Workflow**:
  + **Push changes to the repository** (e.g., a new feature in main.tf or variables.tf).
  + **GitHub Actions triggers** a pipeline to validate and apply the infrastructure changes via Terraform.
  + **Ansible** playbooks automatically configure the environment and services on VMs once provisioned.
  + Optionally, use GitHub Actions to manage **VM Start/Stop** using the Azure CLI in the pipeline itself.
* **Future Scope**: You could expand this to include **dynamic scaling** and **auto-healing** of infrastructure. If GHES usage increases, Terraform could automatically spin up additional resources (e.g., additional replicas) or scale up/down based on load.
  + 1. **Cost Optimization using Automation**
* **Dynamic Scheduling**: Use Azure Automation or a serverless platform (like **Azure Functions**) to dynamically start and stop VMs based on real-time demand instead of fixed schedules.
  + Use metrics from **Azure Monitor** to automatically scale the infrastructure up or down based on CPU, memory, or network usage.
  + Implement **spot instances** or use cheaper VM types during off-peak hours to reduce costs.
* **Future Scope**: Explore **autoscaling** for VMs. For instance, based on CPU utilization, VMs could automatically scale vertically (increase/decrease VM size) or horizontally (add/remove instances).
  + 1. **Integrate Monitoring and Alerts**
* Set up comprehensive **monitoring and alerting** for your GHES instances, infrastructure, and automation workflows.
  + Use **Azure Monitor** or **Prometheus** to track the performance of GHES servers, network traffic, and other resources.
  + Integrate **Grafana** for visualization dashboards that provide real-time insights into your system's health and performance.
  + Configure **alerts** for high CPU usage, disk space issues, or errors in automation workflows. For example, alerts can notify you if a VM fails to shut down properly.
* **Future Scope**: Add **automated remediation**. For example, if CPU usage on the GHES primary node exceeds 80%, an alert could trigger an automation script to start a new replica instance.
  + 1. **Infrastructure as Code Best Practices**
* Implement advanced **Infrastructure as Code (IaC) patterns**:
  + **Modularize Terraform**: Break your Terraform code into reusable modules (e.g., one module for VMs, one for networking, etc.). This increases maintainability and reusability.
  + **State Management**: Use **Terraform Cloud** or **Azure Blob Storage** to securely store your Terraform state files. This allows multiple users to safely collaborate on infrastructure without overwriting each other's changes.
* **Version Control Best Practices**: Implement **branching strategies** (e.g., GitFlow or trunk-based development) for managing infrastructure changes. You can also enforce **code reviews** for any changes to Terraform or Ansible playbooks.
* **Future Scope**: You can integrate with other IaC tools like **Pulumi** or **Crossplane** to create hybrid and multi-cloud deployments.
  + 1. **Security and Compliance Automation**
* **Automated Security Audits**: Integrate tools like **HashiCorp Sentinel** or **Azure Policy** to enforce security policies on the infrastructure. For example, enforce network security rules or ensure that only authorized SSH keys are used to access VMs.
* **Compliance Checks**: Use **Ansible** or **Terraform** to automatically check for compliance with organizational policies (e.g., ensure that all users use company-approved email domains).
* **Encryption & Access Control**: Ensure **encryption at rest** and **in transit** for all GHES data. Set up **role-based access control (RBAC)** and integrate with **Azure Active Directory (AAD)** for seamless authentication and authorization.
* **Future Scope**: Introduce **self-healing** security mechanisms, where compliance or security issues detected by monitoring systems automatically trigger a remediation script.
  + 1. **Serverless Components for Automation**
* **Azure Functions** or **AWS Lambda**: Offload certain tasks to serverless components, such as automated backups, monitoring scripts, or lightweight compute tasks that don’t require full VMs. Serverless solutions are highly scalable, cost-efficient, and allow you to pay only for the execution time of your code.
* **Future Scope**: You can expand serverless functions to handle complex workflows or integrate with third-party systems (e.g., sending notifications via Slack or integrating with a ticketing system like Jira).
  + 1. **Auto-Healing and High Availability**
* Implement **auto-healing** for your GHES servers:
  + If the primary GHES server goes down, automate the failover process to promote the replica to primary. This can be fully automated via **Ansible** playbooks or integrated into your monitoring stack.
  + Ensure **data replication** between primary and replica GHES instances for high availability across regions.
* **Future Scope**: You can expand this by enabling **cross-region replication** for disaster recovery, ensuring that even if an entire Azure region goes down, your data remains safe and accessible.
  + 1. **Containerization and Kubernetes**
* Explore **containerizing** some parts of your infrastructure, such as running GHES in a **Kubernetes** environment (though this is complex and requires additional consideration for persistent storage). You could use **Azure Kubernetes Service (AKS)** to run GHES replicas or supporting services (e.g., monitoring, backup services) as containers.
* **Future Scope**: Leverage **Kubernetes** for deploying and scaling microservices, supporting services, or GHES replicas. Kubernetes' native features like auto-scaling and self-healing can be beneficial.
  + 1. **Data Backups and Disaster Recovery**
* Automate your backup processes using **Azure Backup** or **ghe-backup**. Schedule **automatic backups** and ensure **off-site replication** for disaster recovery.
* **Future Scope**: Implement a **disaster recovery plan** that includes replicating critical data to a different Azure region. Test recovery processes periodically to ensure business continuity.
  + 1. **Expand to Multi-Cloud or Hybrid Cloud**
* Once the infrastructure is stable, you can explore integrating with other cloud providers (e.g., AWS or Google Cloud) for a **multi-cloud strategy**. This can help you avoid vendor lock-in and ensure redundancy across cloud providers.
* **Future Scope**: Use tools like **Terraform** or **Pulumi** to deploy to multiple cloud platforms or integrate with on-premise infrastructure, making your GHES deployment more robust and adaptable.
  1. **Future Scope and Expansion:**
     1. **Scaling:**

As the number of users increases, scale infrastructure dynamically with **autoscaling** of VMs or GHES replicas.

* + 1. **Disaster Recovery:**

Implement **cross-region replication** and full disaster recovery (DR) solutions, ensuring the GHES infrastructure is resilient even if an entire Azure region fails.

* + 1. **Machine Learning for Predictive Monitoring:**

Integrate **AI/ML models** to predict performance bottlenecks, identify failure patterns, and trigger preventive maintenance.

* + 1. **Multi-Cloud and Hybrid Deployments:**

Expand infrastructure to support deployments on other clouds like AWS or Google Cloud, while maintaining a unified management layer.

* + 1. **Containerization with Kubernetes:**

Shift parts of your infrastructure (or supporting services) to containers and use **Kubernetes** for orchestration, providing auto-scaling, load balancing, and high availability for supporting services.

* + 1. **DevSecOps Integration:**

Introduce **DevSecOps** practices by automating security checks, vulnerability scans, and compliance audits during your CI/CD workflows.

By implementing these optimizations and expansions, your project will become more cost-efficient, scalable, secure, and ready for future growth. Let me know if you'd like further details on any of these suggestions or if you'd like to dive deeper into specific areas!