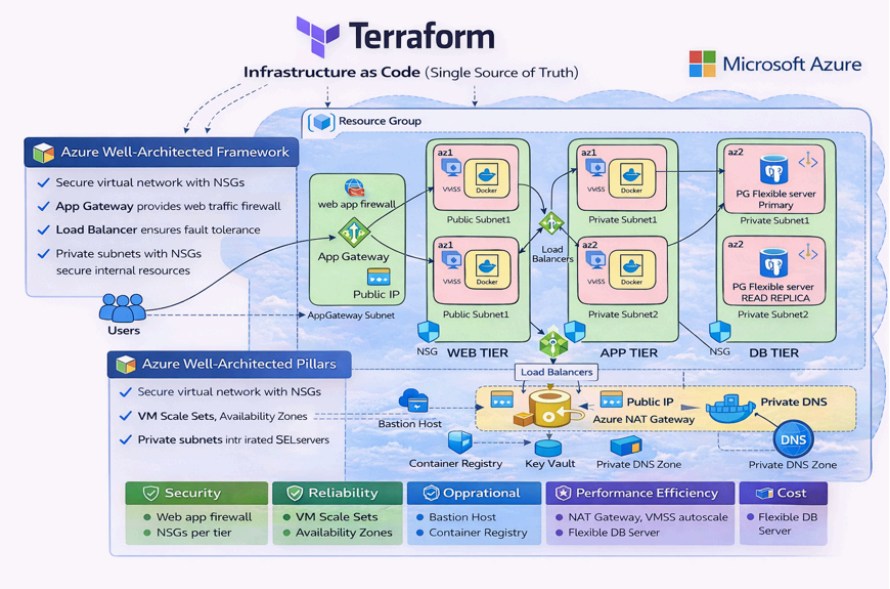
Deploying a Secure, Scalable Multi-Tier Application on Azure using Terraform



This project demonstrates the design, deployment, and operation of a production-grade multi-tier application on Microsoft Azure, aligned with the Azure Well-Architected Framework.

The focus is not just on making it work, but on making deliberate architectural trade-offs across security, reliability, cost, and operational excellence — the same principles used in large-scale, real-world systems.

This project implements a production-style, three-tier web application on Azure, designed to demonstrate real-world cloud architecture, infrastructure decision-making, and operational readiness.

The system consists of:

- A public ingress layer using Azure Application Gateway for Layer 7 routing and health checks
- A stateless application tier running containerized Node.js services on Azure Virtual Machine Scale Sets
- A data access layer secured via Managed Identity and Azure Key Vault, with no secrets embedded in code or configuration

All infrastructure is provisioned using Terraform, emphasizing reproducibility, least-privilege security, and operational clarity over platform complexity.

Rather than optimizing for feature richness, this project intentionally focuses on:

- Clear architectural boundaries
- Explicit trade-offs and non-goals
- Failure-mode awareness
- Leadership-level system design thinking

In short: this is not a demo app — it's a production-aligned cloud system built to reflect how senior engineers, engineering managers, and technical leaders design, reason about, and operate infrastructure at scale.

This architecture aligns with Microsoft's **Azure Well-Architected Framework**, emphasizing intentional design decisions over default configurations.

emphasizing intentional design decisions over default configurations.

Pillar	How It's Addressed
Operational Excellence	Infrastructure defined using Terraform for reproducible, auditable deployments. Clear health endpoints, structured logging, and explicit dependency management across Application Gateway, VMSS, and database layers.
Security	Defense-in-depth approach using Application Gateway with WAF, private subnets, strict NSGs, Azure Bastion for secure access, and Azure Key Vault for secret management. No direct public access to application or database tiers.
Reliability	Virtual Machine Scale Sets provide self-healing and horizontal scaling. Azure Load Balancers and Application Gateway health probes ensure traffic is routed only to healthy backend instances. PostgreSQL Flexible Server includes read replica support.
Performance Efficiency	Tiered architecture reduces contention and improves latency. Internal load balancing enables efficient east-west traffic. Stateless application containers allow rapid scaling based on demand.
Cost Optimization	Right-sized VM SKUs, autoscaling to match traffic patterns, managed database services to reduce operational overhead, and centralized NAT Gateway to control outbound traffic costs.

The solution implements a **three-tier architecture** with clear separation of concerns:

· **Web Tier**

Internet-facing entry point protected by Azure Application Gateway (WAF-enabled).

· **Application Tier**

Horizontally scalable backend services running in Docker on Virtual Machine Scale Sets (VMSS).

· **Database Tier**

Azure PostgreSQL Flexible Server deployed in private subnets with read replica support.

All tiers are isolated using private networking, strict NSGs, and controlled ingress/egress, following zero-trust networking principles.

All tiers are isolated using private networking, strict NSGs, and controlled ingress/egress, following zero-trust networking principles.

🔐 **Security**

- Azure Application Gateway with Web Application Firewall (WAF) protects against OWASP Top 10 threats.
- Strict Network Security Groups (NSGs) enforce least-privilege network access across tiers.
- Web, application, and database tiers are isolated using private subnets to minimize blast radius.
- Azure Bastion enables secure administrative access without exposing VMs to the public internet.
- Azure Key Vault securely stores secrets such as database credentials and sensitive configuration.

🛡️ **Reliability**

- Virtual Machine Scale Sets (VMSS) provide high availability, self-healing, and horizontal scaling.
- Azure Load Balancers distribute traffic across healthy instances, removing single points of failure.
- PostgreSQL Flexible Server includes a read replica to support high availability and read scalability.
- Health probes ensure traffic is routed only to healthy backend instances.

⚡ Performance Efficiency

- Horizontal autoscaling enables the platform to handle variable traffic without manual intervention.
- Tier separation reduces resource contention and improves request latency.
- Internal load balancing enables low-latency communication between application components.

💰 Cost Optimization

- Autoscaling ensures infrastructure scales with demand, avoiding over-provisioning.
- VM sizes are deliberately right-sized to balance performance and cost.
- Managed database services reduce operational overhead and long-term maintenance costs.
- Centralized NAT Gateway controls outbound traffic, avoiding unnecessary public IP sprawl.

🔧 Operational Excellence

- Infrastructure is fully provisioned using Terraform, enabling repeatable and auditable deployments.
- Infrastructure-as-Code (IaC) improves change management, consistency, and rollback capability.
- Centralized logging and metrics support faster incident detection and resolution.
- Explicit health endpoints enable proactive monitoring and automated recovery.

Prerequisites

- Azure CLI installed and configured
- Terraform v1.5.0 or later
- Azure subscription and permissions to create resources
- Docker installed locally for building and pushing container images

Clone Repository

Step 1: Clone the repository using the URL GitHub - sandeep-ssh/three-tier-az

Logging in and Authentication to Azure CLI

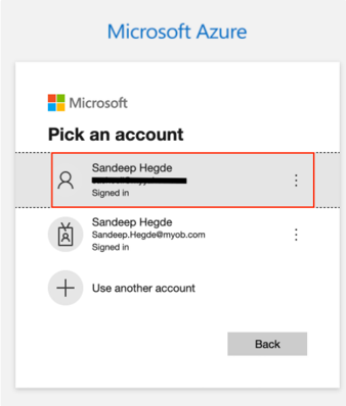
Service Principal Setup for Terraform

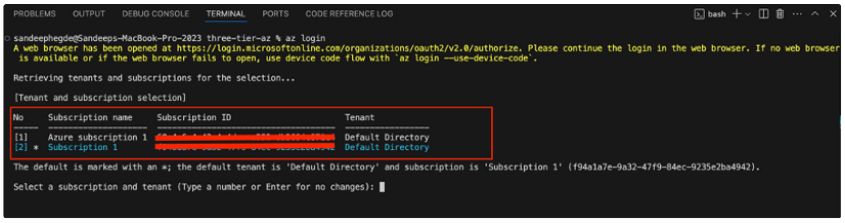
Before deploying infrastructure with Terraform, you need to create a service principal with contributor permissions. This allows Terraform to authenticate with Azure and create resources on your behalf.

Create Service Principal

Login to Azure CLI:

```
az login
```

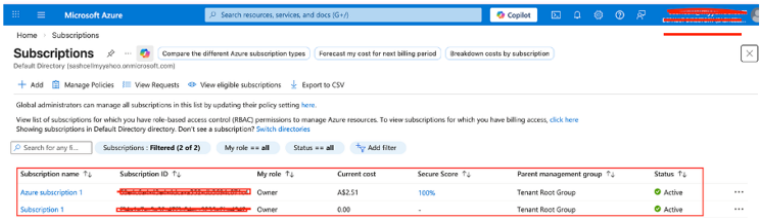




Create a service principal with Contributor role

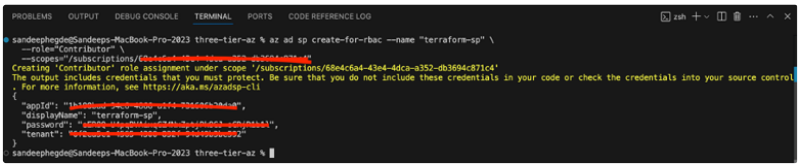
What this command does: It creates a service principal with Contributor role access to your Azure subscription, which is commonly used for Terraform to authenticate and manage Azure resources.

First step is to log into the console and retrieve the subscription ID




```
az ad sp create-for-rbac --name "terraform-sp" \
  --role="Contributor" \
  --scopes="/subscriptions/<YOUR_SUBSCRIPTION_ID>"

```

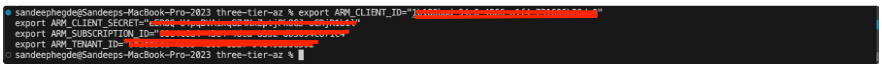
Once this has been set up – it will show like this



> ***Tip:*** *Copy and paste this securely on a notepad. We will need to use this in the next steps.*

Set environment variables for Terraform authentication:

```
export ARM_CLIENT_ID="<appId>"
export ARM_CLIENT_SECRET="<password>"
export ARM_SUBSCRIPTION_ID="<YOUR_SUBSCRIPTION_ID>"
export ARM_TENANT_ID="<tenant>"

```



Environment variables have been set

Important security notes:

- These variables are now in your shell's memory. They'll be lost when you close the terminal session.
- **Never commit these credentials to version control** (Git, GitHub, etc.)
- For persistent credentials across sessions, consider using a `.env` file (which you should `.gitignore`) or your system's credential manager.
- In production CI/CD pipelines, use secure secret management (GitHub Secrets, Azure Key Vault, etc.) instead of exporting in plain text.

Alternative: Create a `.env` file (recommended for persistent use if account is used in production):

```
cat << EOF > .env
export ARM_CLIENT_ID="<appId>"
export ARM_CLIENT_SECRET="<password>"
export ARM_SUBSCRIPTION_ID="<YOUR_SUBSCRIPTION_ID>"
export ARM_TENANT_ID="<tenant>"
EOF

# Source the environment variables
source .env

```

Verify the service principal can authenticate:

```
az login --service-principal \
  --username $ARM_CLIENT_ID \
  --password $ARM_CLIENT_SECRET \
  --tenant $ARM_TENANT_ID

```

Security Best Practices

- **Use least privilege**: The Contributor role provides broad permissions. For production, consider creating custom roles with minimal required permissions
- **Rotate credentials regularly**: Service principal secrets should be rotated periodically
- **Use Azure Key Vault**: For production deployments, consider storing service principal credentials in Azure Key Vault
- **Store credentials securely**: Never commit the `.env` file or credentials to version control. Add `.env` to your `.gitignore` file

```
echo ".env" >> .gitignore
```

Steps to Deploy infrastructure using Terraform

1. **Set up Terraform Backend**

Create an Azure Storage Account for storing Terraform state:

```
# Login to Azure
az login

```

Create Resource Group for Terraform state

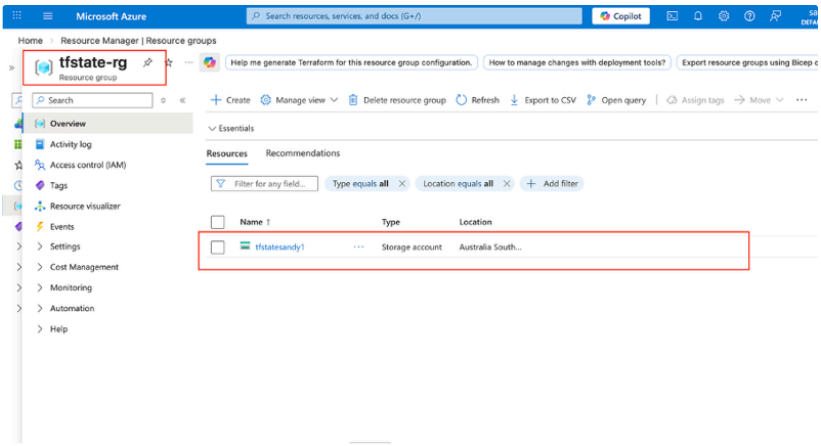
```
az group create --name tfstate-rg --location australiasoutheast
```

Create Storage Account

> ***Tip:*** *Storage account must be unique across all accounts*

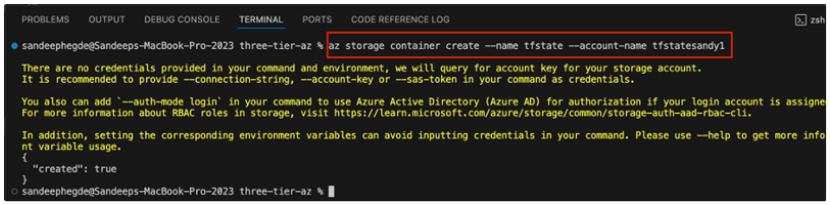
```
az storage account create --name tfstate<unique_suffix> --resource-
group tfstate-rg --sku Standard_LRS --encryption-services blob
```

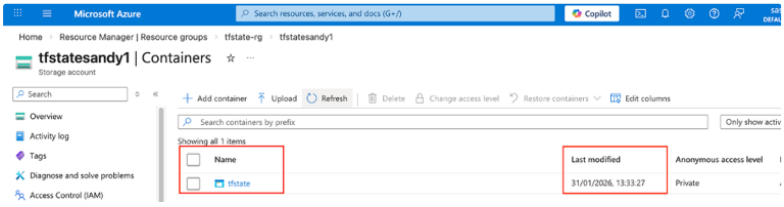
To validate on Azure – just navigate to Resource Group to view this



Create Storage Container

```
az storage container create --name tfstate --account-name
tfstate<unique_suffix>
```



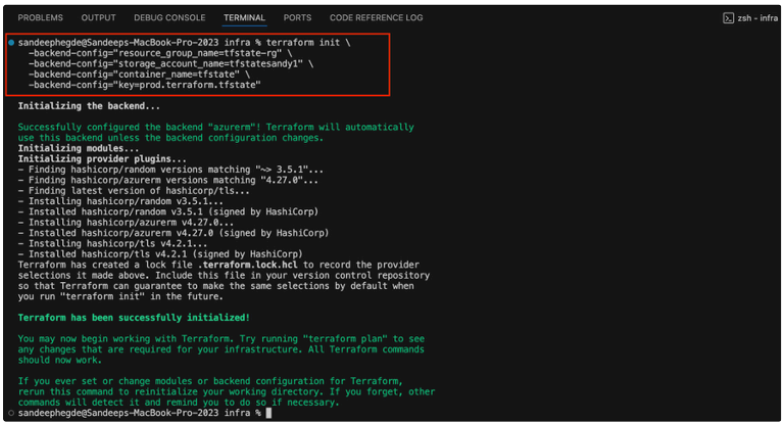


Next step is to initialise terraform to

Initialize Terraform

```
cd infra
terraform init \
  -backend-config="resource_group_name=tfstate-rg" \
  -backend-config="storage_account_name=tfstate<unique_suffix>" \
  -backend-config="container_name=tfstate" \
```

```
  -backend-config="key=prod.terraform.tfstate"

```

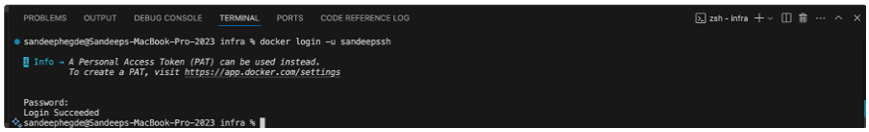


One-Step Deployment Approach

Since we're now using Docker Hub instead of Azure Container Registry, we can deploy the entire infrastructure in one step. First, make sure your Docker images are pushed to Docker Hub:

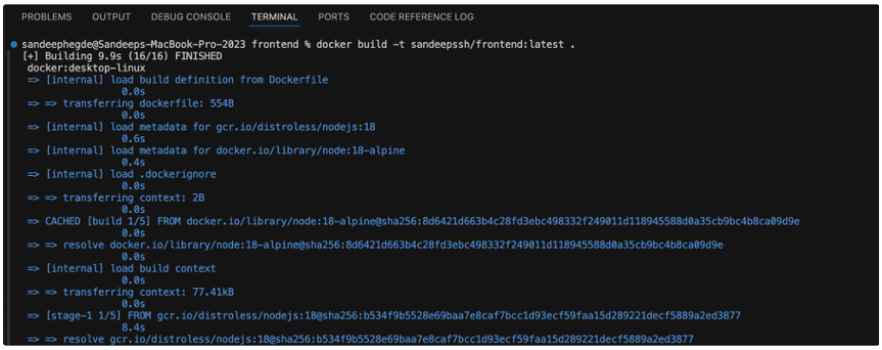
Log in to Docker Hub

```
docker login -u YOUR_DOCKERHUB_USERNAME
```

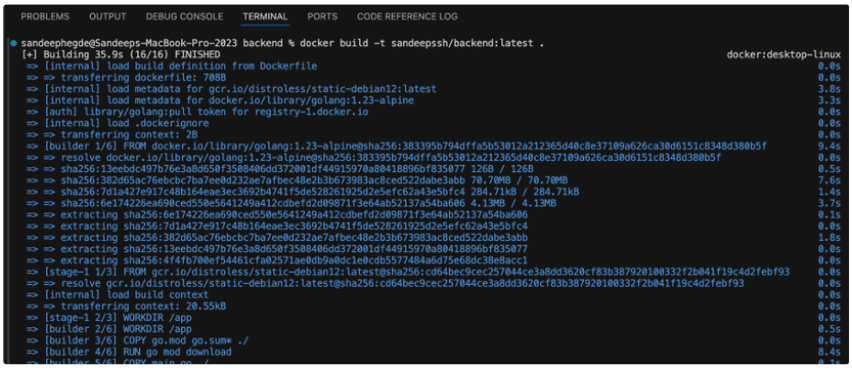


Build and tag your images (Run from the root of the project)

```
docker build -t YOUR_DOCKERHUB_USERNAME/frontend:latest
```




```
docker build -t YOUR_DOCKERHUB_USERNAME/backend:latest .
```



Push to Docker Hub

```
docker push YOUR_DOCKERHUB_USERNAME/frontend:latest
```

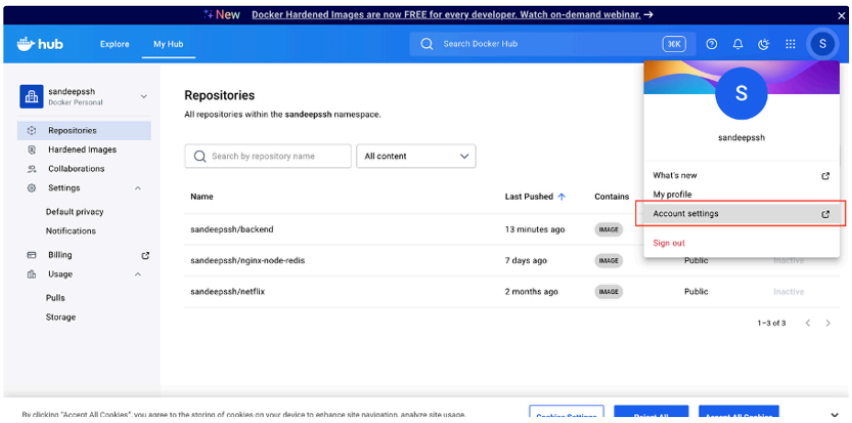
```
sandeephegde@Sandeeps-MacBook-Pro-2023 frontend % docker push sandeepssh/frontend:latest
The push refers to repository [docker.io/sandeepssh/frontend]
b3960cc2fb30: Pushed
ac4c8a89e5c0: Pushed
887f0ce528cf: Pushed
4e07a52ef8d8: Pushed
08088cdaeb91: Pushed
6b0b1527d75b: Pushed
26e3e4b0848c: Pushed
7f27fa967f88f: Pushed
5e50d1da3318: Pushed
latest: digest: sha256:abfbc77d4060d12416b7cc3f40ae5f4be90457e693af522b1c340e5bb1ac1d41 size: 856
sandeephegde@Sandeeps-MacBook-Pro-2023 frontend %
```

```
docker push YOUR_DOCKERHUB_USERNAME/backend:latest
```

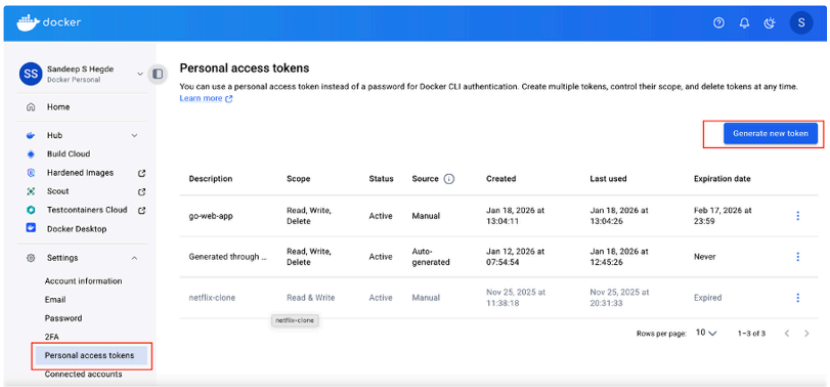
```
sandeephegde@Sandeeps-MacBook-Pro-2023 backend % docker push sandeepssh/backend:latest
The push refers to repository [docker.io/sandeepssh/backend]
52630fc75a18: Layer already exists
dcaa5a89b0cc: Layer already exists
62de241dac5f: Layer already exists
dd64bf2dd177: Layer already exists
49eb3d1d8630: Layer already exists
4aa0ea1413d3: Layer already exists
7c12895b777b: Layer already exists
d1c559a043f5: Layer already exists
bfb59b82a9b6: Layer already exists
069d1e267530: Layer already exists
c3c024fdda6b: Already exists
017886f7e176: Layer already exists
3214acf345c0: Layer already exists
ae270ebef036: Layer already exists
2780920e5dbf: Layer already exists
latest: digest: sha256:df6bef9b19e126a6a4d6cf8726a2f734bef9d1e80d32e3fdc5cc69293824cd67 size: 856
```

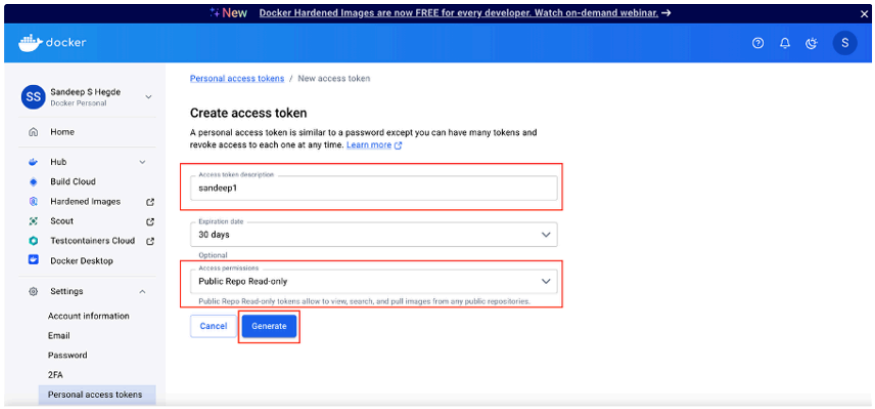
After pushing your images to Docker Hub, we will now need to deploy the infrastructure.

We also need to generate a Personal Access Token so lets login to Dockerhub to create this.



After we select Account Settings now we need to





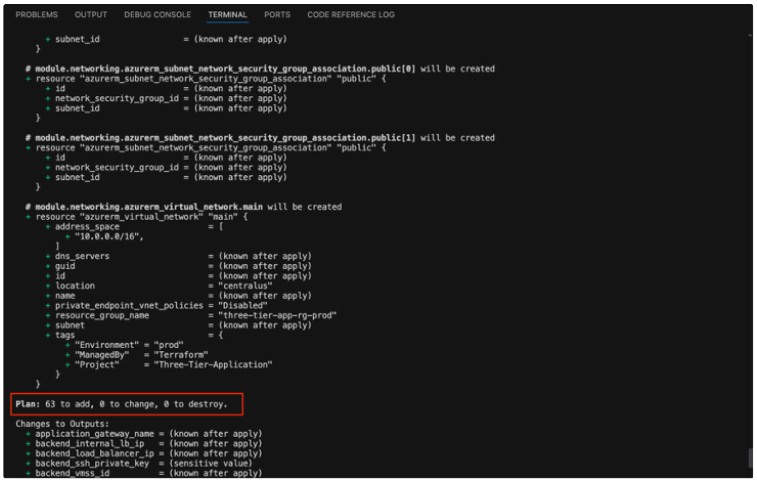
with your Docker Hub credentials:

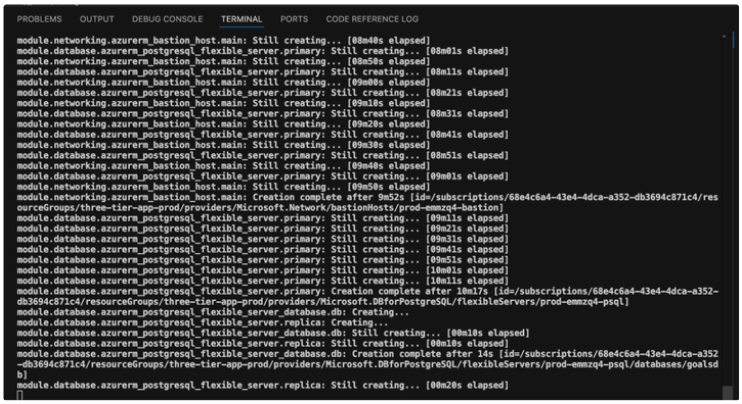
```
cd infra
terraform apply \
  -var-file="environments/prod/terraform.tfvars" \
  -var="dockerhub_username=YOUR_DOCKERHUB_USERNAME" \
  -var="dockerhub_password=YOUR_DOCKERHUB_PAT" \
  -var="frontend_image=YOUR_DOCKERHUB_USERNAME/frontend:latest" \
  -var="backend_image=YOUR_DOCKERHUB_USERNAME/backend:latest"

```

This command will:

- Deploy all infrastructure components including compute resources
- Store your Docker Hub Personal Access Token securely in Azure Key Vault
- Configure the VM Scale Sets to pull images from Docker Hub
- Use the specified Docker images for frontend and backend deployments

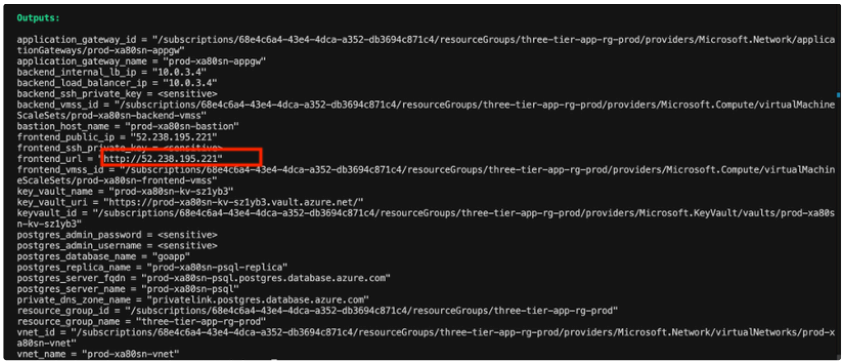
> ***Tip:*** This may take over 20-30 mins to be fully implemented





Access the Application

It will now create the required Outputs as defined in the `outputs.tf` file



After deployment completes, access your application:

Frontend: Use the Application Gateway public IP address:

```
echo "Frontend URL: http://$(terraform output -raw frontend_public_ip)"
```

Backend: Access via internal load balancer (from within the VNet):

```
echo "Backend internal endpoint: http://$(terraform output -raw backend_internal_lb_ip):8080"
```

Database: Access via private endpoints from the backend tier:

```
echo "PostgreSQL Server: $(terraform output -raw postgres_server_fqdn)"
echo "PostgreSQL Replica: $(terraform output -raw postgres_replica_name)"

```

SSH into the Bastion host to access the backend and frontend:

```
terraform output -raw frontend_ssh_private_key > frontend_key.pem
terraform output -raw backend_ssh_private_key > backend_key.pem

```

Cleanup

To destroy the infrastructure when no longer needed:

```
terraform destroy -auto-approve
```

> ***Tip :*** If You get a error in the destrucion process rerun the above command again

Challenges Faced

- Debugging **Application Gateway backend health** when VMSS instances were healthy but not correctly registered.
- Aligning **Docker container ports**, load balancer probes, and Application Gateway listener configurations.
- Handling **private DNS resolution** for PostgreSQL Flexible Server within isolated VNets.
- Managing **ACR image availability** and ensuring VMSS instances pulled the correct container versions.
- Identifying silent failures caused by mismatched health endpoints (`/health` vs `/`).

Each challenge required systematic troubleshooting across multiple Azure services — reinforcing the importance of **systems thinking over isolated fixes**.

⚖️ Architectural Trade-offs & Non-Goals

Area	Decision	Rationale

Container Orchestration	Did not use AKS	VM Scale Sets were intentionally chosen to reduce platform complexity and operational overhead, keeping focus on core architecture principles rather than cluster management. This mirrors early-stage or cost-conscious production environments.
High Availability Scope	Regional HA instead of multi-region	Architecture prioritizes availability zones within a single region to balance resilience, latency, and cost. Multi-region failover was considered but deemed unnecessary for the current reliability objectives.
CI/CD Complexity	No full GitOps pipeline	Deployment automation was kept intentionally minimal to emphasize infrastructure correctness, repeatability, and observability before introducing advanced delivery workflows.
Service Mesh	No service mesh introduced	Traffic patterns are simple and well-understood. Introducing a service mesh would add operational complexity without proportional reliability or security benefits at this scale.
Zero Trust Maturity	Partial Zero Trust implementation	Network isolation, identity-based access, and secret management are implemented. Advanced conditional access, workload identity federation, and continuous posture assessment are future enhancements.
Autoscaling Strategy	Reactive scaling over predictive scaling	Autoscaling is driven by platform signals (VMSS health and load) rather than predictive or ML-based models, favoring reliability and explainability over optimization.
Observability Depth	Metrics-focused over full tracing	Core metrics and health checks are implemented first. Distributed tracing and deep request-level correlation were deferred to avoid premature complexity.
Cost vs. Resilience	Balanced, not maximized	The design intentionally avoids over-engineering (e.g., always-on replicas, cross-region traffic routing) to maintain a strong cost-to-value ratio.
Security Hardening	No customer-managed HSM	Azure-managed Key Vault encryption was selected for operational simplicity. Customer-managed keys and HSM-backed keys remain a future hardening option.

Dimension	**Reflection**
Engineering Philosophy	Prefer explicit trade-offs over maximal feature sets
Leadership Lens	Architecture should reduce team cognitive load, not increase it
System Design	Simplicity enables faster diagnosis, iteration, and scaling
Growth Path	This system is intentionally extensible without being overbuilt

Why This Project Matters

This project reflects how I approach engineering problems:

- ✅ Think in **systems**, not just services
- ✅ Design for **failure, scale, and security from day one**
- ✅ Balance **technical depth with business impact**

- ✅ Treat infrastructure as a **product**, not a one-time setup

It mirrors the kind of decision-making, ownership, and technical leadership expected in **senior engineering, engineering management, and platform roles**.

🚀 Core Skills Used

- **Cloud**: Azure (VMSS, App Gateway, PostgreSQL, Key Vault)
- **IaC**: Terraform (modular, tested)
- **Languages**: Go, Python, Bash
- **Containers**: Docker, image optimization
- **Observability**: Metrics, logging, alerting concepts
- **Security**: Defense-in-depth, compliance frameworks

📜 Architectural Decision Records (ADRs)

ADR ID	Decision	Context	Rationale (Why)	Trade-offs / Consequences
ADR-001	VM Scale Sets instead of AKS	Needed horizontal scalability and health-based recovery without excessive platform complexity	VMSS keeps operational focus on core infrastructure fundamentals and reduces cognitive overhead	Manual container lifecycle management, less orchestration flexibility than Kubernetes
ADR-002	Azure Application Gateway as L7 ingress	Required TLS termination, health probes, and intelligent routing	Native Azure integration, production-grade Layer 7 routing, simpler than third-party ingress	Higher cost than L4 LB, sensitive to probe/backend misconfiguration
ADR-003	Managed Identity + Azure Key Vault	Secure handling of DB credentials and secrets	Eliminates hardcoded secrets, enforces least privilege, aligns with Zero Trust	Initial IAM setup complexity, steeper troubleshooting curve
ADR-004	Single-region, zone-resilient deployment	High availability required without global scale	Realistic cost/resilience balance, mirrors many production systems	No cross-region DR, regional outage remains a risk

ADR-005	Metrics-first observability	Needed visibility with minimal tooling overhead	Faster signal-to-noise ratio, easier operational ownership	Limited request tracing, debugging deep flows is harder

---

📌 Next Improvements

- Introduce Azure Monitor dashboards and alerts
- Add CI/CD for container builds and Terraform plans
- Enable blue/green or canary deployments
- Expand database backup and disaster recovery strategy

🏁 Final Reflection

Dimension	Reflection
Engineering Philosophy	Prefer explicit trade-offs over maximal feature sets
Leadership Lens	Architecture should reduce team cognitive load, not increase it
System Design	Simplicity enables faster diagnosis, iteration, and scaling
Growth Path	This system is intentionally extensible without being overbuilt

Key takeaway:

> *This project reflects how I design systems in real organizations — start simple, document decisions, acknowledge trade-offs, and evolve architecture with intent rather than novelty.*