# Docker-Compose Deployments

*Basic Technical Documentation*

## Introduction

There are a variety of tools within and around the Docker ecosystem for building and deploying multi-container solutions.

For this use case, a Dockerfile is used to create custom images for individual services. docker-compose is then referencing those images (or pre-built ones from registries like Docker Hub, ACR, etc.) to define how multiple containers are configured and run together.

Use docker-compose when you need to manage and run multiple containers as a cohesive application, especially in development or testing environments. However, docker-compose is not ideal for complex production orchestration (Kubernetes or Docker Swarm are better for that); it is primarily suited for development and simple deployments.

## Localhost/IDE-side Deployments

Use this method to deploy/execute containers on the local machine (e.g. a Laptop/Desktop PC).

Example workflow:

* Write a Dockerfile to build one or more custom images for the app.
* Use docker-compose.yml to define how the app containers interact with other services (e.g., API, database).
* Run ‘docker-compose up’ to launch the entire stack on your local machine.

## Cloud-side Deployments (IaC)

Use this method to deploy/execute Containers on public cloud hosts.

Infrastructure-as-Code (IaC) offers a more safe, secure, fast, robust, and repeatable way of deploying to cloud in comparison to ClickOps or shell scripts, due to its idempotency, and so has been included here.

Terraform has been selected for this example due to it being cloud provider agnostic, however provider-specific tools are also widely used (e.g. CloudFormation/CDK for AWS, ARM/Bicep for Azure). Azure has also been used in this example.

Terraform does not directly support deploying docker-compose files to Azure Container Apps, so the docker-compose services need to be translated into individual Azure Container Apps and configured accordingly. Azure Container Registry (ACR) is used here to store the Docker images and Azure Database for PostgreSQL for persistent storage. The volumes for frontend and backend are less critical in a containerized cloud setup, as application code is typically baked into the images, but this setup ensures database persistence.

Terraform deployment to cloud may execute from localhost/IDE or from automation pipeline worker node/agent (e.g. Jenkins, Azure DevOps, GitHub Actions, etc.).



Prerequisites

* An Azure subscription.
* Terraform CLI installed.
* Azure CLI installed and authenticated (az login).
* Docker images for the frontend and backend built and pushed to Azure Container Registry.
* A .env file or Terraform variables for sensitive data (e.g., database credentials).

If you prefer deploying the docker-compose file directly, you could use Azure App Service with multi-container support, but this has limitations and is less flexible than Container Apps.

Steps to Deploy:

* Prepare docker images
* Setup terraform backend
* Initialize terraform
* Set sensitive input variables
* Terraform fmt, validate, plan & apply
* Test access to frontend URL, API URL, and database CLI

Cleanup:

To remove the deployment once finished run:

terraform destroy

## Conclusion

The above localhost & IaC deployment examples should be fine for development and test scenarios. For production environments, there are many recommended security and performance enhancements/considerations, such as private networking, firewall/security groups, storing sensitive data in key vaults, encryption in-flight and at-rest, and autoscaling to name but a few; a managed Kubernetes service for container orchestration, such as Elastic Kubernetes Service (EKS) in AWS or Azure Kubernetes Service (AKS) in Azure, combined with Helm Chart for resource templates is a popular solution and should work well.