**Hands-On Task Fragen**

1. **Warum wurde die vorhandene VM-Größe gewählt?**

Die VM-Größe wurde basierend auf den Anforderungen der Anwendung gewählt – z. B. CPU-, RAM- oder Netzwerkbedarf. Wenn die Anwendung z. B. statisches Hosting übernimmt, reicht eine kleinere Standard\_B1s-Instanz aus. Für produktionsnahe Umgebungen mit Last könnte Standard\_D2s\_v3 gewählt werden.

1. **Wurden nötige Ressourcen erstellt, damit die VM erreichbar ist?**

**1. Wie wird sichergestellt, dass die VM(s) aus dem Internet erreichbar sind?**

Damit eine VM im VM Scale Set über das Internet erreichbar ist, wurden mehrere Ressourcen korrekt konfiguriert:

* **Öffentliche IP-Adresse (azurerm\_public\_ip)**  
  → Bereitgestellt und an den Load Balancer gebunden.
* **Load Balancer (azurerm\_lb)**  
  → Verteilt HTTP-Traffic (Port 80) auf die Instanzen im VM Scale Set.
* **Frontend-IP-Konfiguration + Backend-Pool + Load Balancing Rule**  
  → Die frontend\_ip\_configuration mit der öffentlichen IP ermöglicht eingehenden Traffic.  
  → Die azurerm\_lb\_backend\_address\_pool verbindet die VMSS-NICs mit dem Load Balancer.  
  → Die azurerm\_lb\_rule leitet Port 80 an die Instanzen weiter.
* **Netzwerksicherheitsgruppe (azurerm\_network\_security\_group)**  
  → Erlaubt eingehenden HTTP-Traffic auf Port 80 über eine security\_rule.  
  → Die NSG ist mit dem Subnetz verbunden (azurerm\_subnet\_network\_security\_group\_association).

**Ergebnis:** Der Load Balancer ist unter einer öffentlichen IP erreichbar, leitet Anfragen an das VMSS weiter, das im Subnetz mit erlaubtem Port 80 liegt.

**2. Wie kann sich die VM nach außen verbinden (z. B. google.com erreichen)?**

* **Load Balancer mit SKU Standard**  
  → Standard-SKUs erfordern keine NAT-Gateway- oder Ausgangs-IP-Konfiguration für ausgehenden Traffic – sie ermöglichen automatisch ausgehenden Internetzugang über das Load Balancer Frontend.
* **Subnetzkonfiguration ohne ausgehende Einschränkungen**  
  → Keine NSG-Regeln verhindern ausgehenden Verkehr, daher kann die VM problemlos z. B. apt-get update ausführen oder google.com erreichen.

**Ergebnis:** Die VM kann ins Internet kommunizieren, z. B. um Pakete herunterzuladen, Telemetrie zu senden oder Websites zu erreichen.

**Fazit**

* **Inbound erreichbar** über Public IP → Load Balancer → NSG-Regel → Port 80
* **Outbound möglich**, weil Standard Load Balancer mit keiner blockierenden NSG kombiniert ist.

1. **Ist der Terraform Code parametrisiert, damit der Code in unterschiedlichen Environments wiederverwendet wurde?**  
   Ja, durch die Verwendung von Variablen (variables.tf) und Umgebungsabhängigen terraform.tfvars-Dateien kann derselbe Code z. B. für test und prod wiederverwendet werden
2. **Wie kann ich sicherstellen, dass die VM über Terraform verwaltet wird?**  
   Durch den terraform state-Befehl kann überprüft werden, ob die VM im State-File enthalten ist. Änderungen sollten ausschließlich über Terraform erfolgen – nicht manuell im Azure Portal.
3. **Wie muss der Code angepasst werden, damit er in einem Team-Setup funktioniert?**
   * Verwendung von Remote State Storage mit Locking
   * Modulstruktur mit klarer Trennung von Komponenten.
   * Eingabevariablen für Flexibilität.
   * Git-basierter Workflow (PRs, Reviews).
4. **Wie kann sichergestellt werden, dass Ressourcen, die voneinander abhängig sind, in der richtigen Reihenfolge erstellt werden?**

**Gute Praxis: Implizite Abhängigkeiten über Referenzen**

Terraform erstellt automatisch einen **Abhängigkeitsgraphen**, wenn Ressourcen auf andere Ressourcen verweisen. Das ist deutlich übersichtlicher und leichter zu pflegen.

**Beispiel:**

hcl

CopyEdit

resource "azurerm\_network\_interface" "example" {

name = "example-nic"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

ip\_configuration {

name = "internal"

subnet\_id = azurerm\_subnet.example.id

private\_ip\_address\_allocation = "Dynamic"

}

}

In diesem Fall erkennt Terraform automatisch, dass die Netzwerkschnittstelle von der Ressourcengruppe und dem Subnetz abhängt. depends\_on ist hier **nicht notwendig**.

**Wann ist depends\_on sinnvoll? (selten, aber manchmal notwendig)**

depends\_on sollte **nur dann** verwendet werden, wenn:

* eine Ressource **indirekt** referenziert wird (z. B. durch Variablen oder Module)
* Terraform die Abhängigkeit **nicht automatisch erkennen** kann, zum Beispiel bei Provisionern oder null\_resource

**Beispiel für einen sinnvollen Einsatz:**

hcl

CopyEdit

resource "null\_resource" "wait\_for\_policy" {

depends\_on = [azurerm\_policy\_assignment.example]

}

**Anti-Pattern: Übermäßige Nutzung von depends\_on**

Wenn depends\_on zu oft und unnötig verwendet wird, hat das mehrere Nachteile:

* die automatische Erkennung von Abhängigkeiten wird untergraben
* Terraform kann Ressourcen nicht mehr parallel erstellen, was zu längeren Laufzeiten führt
* der Code wird fehleranfälliger und schwerer zu warten

**Zusammenfassung**

| **Aspekt** | **Empfehlung** |
| --- | --- |
| Ressourcenreferenzen verwenden | Ja |
| depends\_on überall einsetzen | Nein |
| depends\_on gezielt bei Bedarf nutzen | In Ausnahmefällen sinnvoll |

1. **Wie kann dieser Code automatisiert ausgeführt werden? Welche Terraform-Commands können wann Sinn machen?**

Ich habe ein einfaches GitHub Actions Setup erstellt, das bei jedem Push auf den main-Branch automatisch den Terraform-Code ausführt. Zuerst wird das **Azure-Login über OIDC (OpenID Connect)** durchgeführt, um eine sichere, passwortlose Authentifizierung ohne gespeicherte Zugangsdaten zu ermöglichen.

Anschließend folgen die typischen Terraform-Befehle:

* terraform init initialisiert das Backend und lädt die benötigten Provider.
* terraform validate prüft, ob der Code syntaktisch korrekt ist und keine offensichtlichen Fehler enthält – dadurch werden Probleme früh erkannt, noch bevor ein Plan erstellt wird.
* terraform plan zeigt die geplanten Änderungen.
* terraform apply -auto-approve wendet diese Änderungen automatisiert an.

Alle sensiblen Informationen wie Zugangsdaten und Variablen (z. B. SSH-Keys, Benutzernamen, Storage-Konfiguration) werden sicher über GitHub Secrets verwaltet. Damit ist eine automatisierte, sichere und reproduzierbare Bereitstellung der Infrastruktur gewährleistet.

1. **Welche Vor- und Nachteile hat die Verwendung von Terraform?**  
   **Vorteile:**
   * Infrastructure as Code, versionierbar
   * Wiederverwendbarkeit über Module
   * Anbieterunabhängigkeit durch Plugins (AWS, Azure, GCP etc.)
   * Übersicht über geplante Änderungen (plan)

**Nachteile:**

* + Lernkurve
  + Fehleranfälligkeit bei manuellem Eingriff in Infrastruktur
  + Zustandssynchronisation bei Teams kann komplex sein ohne Remote State

**DevOps Verständnis Fragen**

1. **Was bedeutet DevOps für dich?**

DevOps ist die enge Zusammenarbeit von Entwicklung und Betrieb mit dem Ziel, Software schneller, zuverlässiger und sicherer zu liefern. Es basiert auf Automatisierung, kontinuierlichem Feedback und einer Kultur des gemeinsamen Verantwortens.

1. **Welches DevOps Tooling hast du bisher verwendet? Falls ja, wie hast du damit gearbeitet?**

Ich habe mit einer Vielzahl an DevOps-Werkzeugen gearbeitet, insbesondere in folgenden Bereichen:

**CI/CD (Continuous Integration / Continuous Deployment)**

* **GitHub Actions**, **GitLab CI**  
  → Automatisierte Builds, Tests und Deployments für Terraform, Container und Applikationen.  
  → Beispielsweise habe ich Pipelines für Infrastruktur-Deployments nach Azure und GCP gebaut.

**Infrastructure as Code (IaC)**

* **Terraform**  
  → Modulbasierte Infrastruktur für Azure, GCP und teilweise AWS.

**Container & Orchestrierung**

* **Docker**  
  → Containerisierung von Backends und Microservices.
* **Kubernetes (k8s)**  
  → Lokale Entwicklung mit Minikube & k3s, sowie produktive Deployments auf AKS/GKE.
* **Ansible**  
  → Konfigurationsmanagement und Provisionierung von VMs und Servern.

**Monitoring, Logging, Observability**

* **Prometheus + Grafana**  
  → Metrikbasiertes Monitoring und Dashboarding.
* **GCP Monitor**, **Log Analytics**, **Application Insights**
* **ELK Stack (Elasticsearch, Logstash, Kibana)**

**Security & Secrets Management**

* **HashiCorp Vault**
* **SOPS (Secrets OPerationS)** + GitOps

**Automatisierung & Provisionierung**

* **Bash**, **Python**, (Skripting für Glue-Code und CI/CD)

**Beispiel aus der Praxis**

Mit GitHub Actions habe ich automatisierte Deployments nach GCP und Azure umgesetzt – z. B.:

* Infrastruktur-Provisionierung mit Terraform
* Rollout von Docker-basierten Services in Kubernetes
* Automatisches Monitoring-Setup mit Azure Monitor & Prometheus
* Alerts bei Ausfällen oder hoher Auslastung

1. **Welche Tools hast du verwendet, um Applikationen und Infrastruktur zu monitoren und loggen?**
   * **Monitoring:** Prometheus, Grafana, Azure Monitor
   * **Logging:** Loki, ELK Stack (Elasticsearch, Logstash, Kibana), Azure Log Analytics
   * **Alerting:** Grafana Alerting, Azure Alerts, Prometheus Alertmanager
2. **Was ist der Unterschied zwischen Monitoring, Alerting und Logging**
   * **Monitoring:** Erfasst systematische Metriken (z. B. CPU, RAM, Latenzen).
   * **Alerting:** Setzt Regeln auf Monitoring-Daten, um Benachrichtigungen bei Schwellenwertüberschreitungen zu erzeugen.
   * **Logging:** Sammelt strukturierte/unstrukturierte Protokolle von Anwendungen oder Systemen zur späteren Analyse oder Fehlersuche.
3. **Welche Rolle spielt Kommunikation im Alltag eines DevOps Teams?**

Kommunikation spielt eine **zentrale Rolle** in DevOps – sie ist nicht nur ein „Soft Skill“, sondern ein **Schlüsselelement für technische Effektivität und Team-Performance**.

Einige Punkte basierend auf dem **Google SRE/DevOps-Buch** (*Site Reliability Engineering*, *The DevOps Handbook*, *Accelerate*):

**Warum ist Kommunikation so wichtig?**

* **Transparenz & Vertrauen:**  
  Wie im *Google SRE Book* betont wird, brauchen Teams **Vertrauen**, um Verantwortung für Deployments, Monitoring und Fehlerbehandlung zu übernehmen. Vertrauen entsteht durch offene Kommunikation.
* **Schnelle Fehlerbehebung (Incident Response):**  
  Bei Incidents ist effektive Kommunikation entscheidend – z. B. bei der Rollenverteilung (Commander, Scribe), klaren Eskalationswegen und Postmortems.
* **Feedbackkultur (Blameless Postmortems):**  
  Laut Google ist es essenziell, **blame-free retrospectives** durchzuführen, in denen man Probleme offen analysiert – ohne Schuldzuweisungen, aber mit Verbesserungspotenzial.
* **Abstimmung zwischen Dev und Ops:**  
  DevOps lebt von enger Zusammenarbeit. Kommunikation sorgt dafür, dass:
  + Anforderungen früh verstanden werden
  + Betriebsanforderungen in die Entwicklung einfließen (z. B. Observability, Scalability)
* **Releases & Change Management:**  
  Wie in *Accelerate* beschrieben, sind schnelle, stabile Releases nur möglich, wenn Dev, QA und Ops **gemeinsam planen und kommunizieren** – inkl. Feature Toggles, Rollbacks, Maintenance Windows etc.

**Zusammengefasst ermöglicht Kommunikation:**

* Schnelle Fehlerbehebung (MTTR reduzieren)
* Gemeinsames Verständnis von Anforderungen und Risiken
* Koordination bei Releases und Incidents
* Etablierung einer Feedback- und Lernkultur
* Cross-Funktionale Zusammenarbeit zwischen Dev, Ops, Security, QA