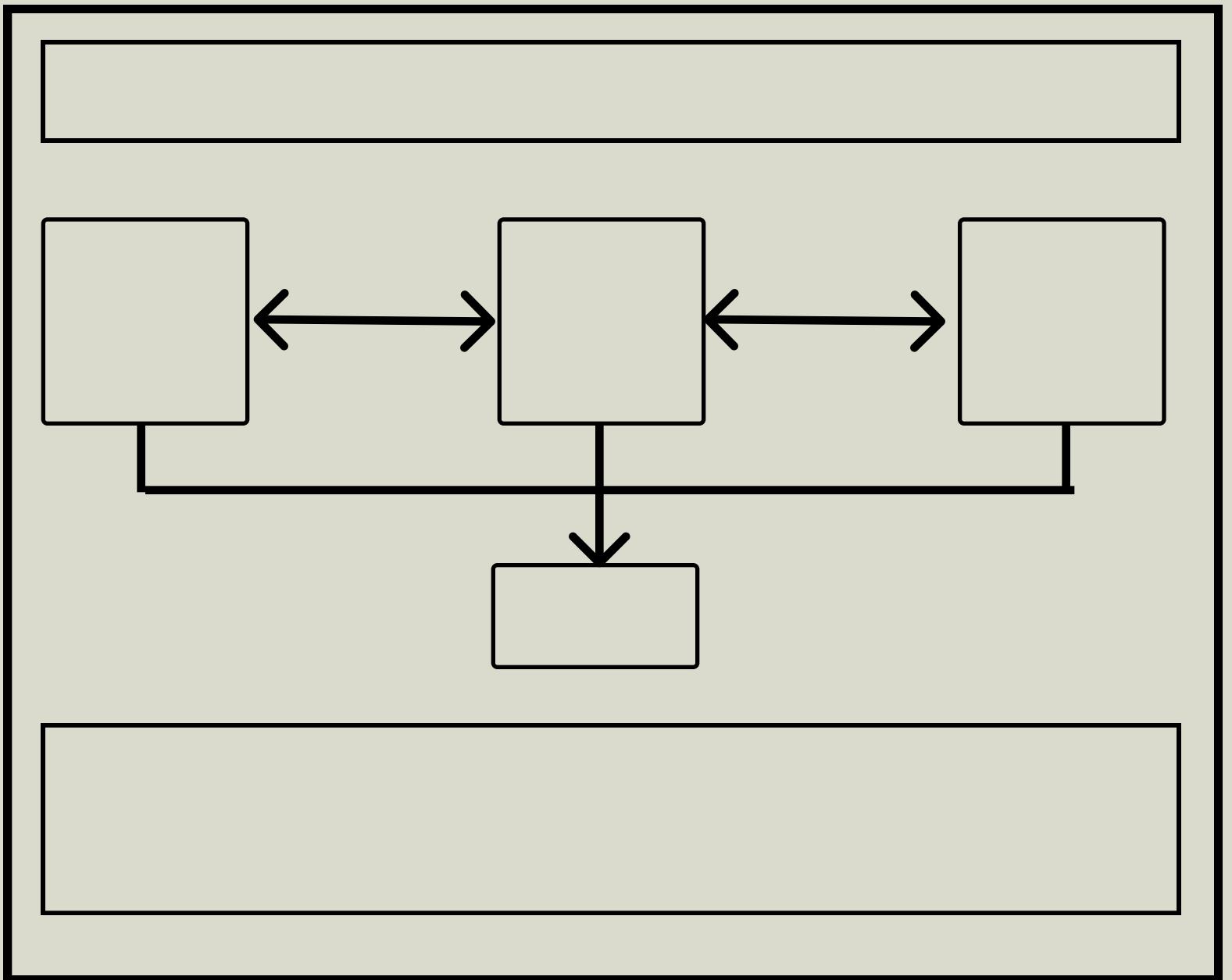


# Aufgabestellung

Bereitstellung einer Cloud-Umgebung für eine Single Page Application.

Für die Position DevOps Engineer



# Inhalt

- 1. ANFORDERUNGSANALYSE**
- 2. SOLUTION DESIGN**
- 3. IMPLEMENTIERUNG**

# 1. ANFORDERUNGSANALYSE

## Anforderungen:

- Bereitstellung einer Single Page Application (SPA) auf Microsoft Azure.
- Technologie Stack: Vue.js (Frontend), Node.js (Backend), MongoDB (NoSQL-Datenbank).
- Präferenz für Platform as a Service (PaaS) Komponenten.
- Vollständige Infrastruktur als Code (IaC/GitOps) Beschreibung.
- Lückenlose Automatisierung der Infrastrukturbereitstellung sowie des Application Build und Deployment Prozesses.
- Unterstützung schneller Entwicklungszyklen (Rapid Development) durch Cloud native Technologien.

## 2. SOLUTION DESIGN

Zwei Lösungen wurden für diese Aufgabe analysiert.

CONTAINERBASIERT MIT AZURE CONTAINER APPS.

AZURE STATIC WEB APPS MIT AZURE APP SERVICES.

## CONTAINERBASIERT MIT AZURE CONTAINER APPS.

### Vorteile

- Container-Flexibilität: Vollständige Kontrolle über die Laufzeitumgebung.
- Portabilität: Container können überall ausgeführt werden.
- Bereit für Microservices: Einfache spätere Aufteilung in mehrere Dienste.
- Serverlose Skalierung: Skalierung auf null bei Nichtnutzung.
- Revisionsmanagement: Einfache Rollbacks.

### Nachteile

- Höhere Komplexität: Manuelles Management von Dockerfiles und Container-Builds erforderlich.
- Etwas höhere Kosten: Kosten für Container Registry und Compute-Ressourcen.
- Lernkurve: Das Team benötigt Container-Kenntnisse.
- Mehr Komponenten: Zusätzliche Verwaltung von Container Registry, Builds und Deployments.

## AZURE STATIC WEB APPS (PaaS) MIT AZURE APP SERVICES.

### Vorteile

- Schnelle Entwicklungszyklen: Durch integrierte CI/CD-Pipelines und Preview-Umgebungen für Pull Requests.
- Kosteneffizienz: Nutzungsbasierte Bezahlung (Pay-as-you-go). No VMs.
- Hohe Verfügbarkeit und Skalierung: Plattformseitig integrierte Redundanz und automatische Skalierungsfunktionen.
- Sicherheit: Nutzung verwalteter Identitäten (Managed Identities) und integrierter Sicherheitsfeatures der Plattform.
- Optimierte Developer Experience: Vereinfachter Bereitstellungsprozess und gute Unterstützung für die Lokalentwicklung.

### Nachteile

- Geringere Kontrolle: Plattformbeschränkungen im Vergleich zu containerisierten Lösungen.
- Vendor Lock-in: Stärkere Bindung an spezifische Azure-Dienste und APIs.
- Begrenzte Anpassbarkeit: Einschränkungen bei Konfiguration und Anpassung der Laufzeitumgebung (Runtime).

Ausgewählte Lösung: **Azure Static Apps (PaaS) mit Azure App Services**

## BEGRÜNDUNG

**Anforderung:** Präferierter Einsatz von Platform as a Service Komponenten

**Argument:** Static Web Apps erreicht 100 % PaaS Nutzung ohne zusätzlichen operativen Aufwand

- Azure Static Web Apps: Vollständig verwaltet.
- Azure App Service: Verwaltete Laufzeitumgebung, kein VM.
- Azure Cosmos DB: Serverlose Datenbank mit automatischer Skalierung und Sicherung.

Ausgewählte Lösung

MANAGED PLATFORM AS A SERVICE (PAAS) MIT AZURE APP SERVICES.

BEGRÜNDUNG

**Anforderung:** Vollständige Automatisierung sowohl des Infrastruktur Deployments, als auch der Anwendungs-Build-und Deploy Jobs

**Argument:** Static Web Apps bietet native Automatisierung mit minimaler Pipeline-Komplexität:

- Terraform stellt alle PaaS-Dienste deklarativ bereit.
- Kein Setup einer Container Registry oder Image-Management nötig.
- Einfacheres State-Management.

## Ausgewählte Lösung

MANAGED PLATFORM AS A SERVICE (PAAS) MIT AZURE APP SERVICES.

### BEGRÜNDUNG

**Anforderung:** Unterstützung schneller Entwicklungszyklen durch Auswahl entsprechender cloud-nativer Technologien

**Argument:** Static Web Apps beschleunigt Entwicklungszyklen durch integrierte Entwickler-Produktivitätsfeatures

- PR Preview Environments: Every pull request automatically gets a live preview URL
- Keine Docker-Kenntnisse für Frontend-Entwickler nötig.
- Schnelleres Onboarding für neue Teammitglieder.
- Der Product Owner kann vor dem Merge prüfen.
- Staging-Slots in Azure App Service ermöglichen sofortiges Swappen.

## ARGUMENTE GEGEN DIE CONTAINERBASIERTE LÖSUNG UND ANERKENNUNG DER KOMPROMISSE.

Containerbasierte führt zusätzlichen operativen Aufwand ein:

- Azure Container Registry erfordert Wartung und Replikation.
- Container-Builds erhöhen die Komplexität der CI/CD-Pipelines.
- Kenntnisse in Container-Orchestrierung (z. B. Kubernetes) sind erforderlich.

Ich erkenne an, dass SWA Einschränkungen hat.

Szenarien, in denen ich stattdessen Option 2 wählen würde:

- Multi-Cloud-Anforderung – Wenn Portabilität entscheidend ist, bieten Container die notwendige Abstraktion.
- Geplante Microservices Evolution.
- Bestehende Container Expertise.
- Kostenoptimierung bei großer Skalierung – Container Apps können auf Null skalieren und so Kosten für dienste mit geringem Traffic sparen.

## 2. SOLUTION DESIGN

Architekturskizzen

CONTAINERBASIERT MIT AZURE CONTAINER APPS.

MANAGED PLATFORM AS A SERVICE (PAAS) MIT AZURE APP SERVICES.

## 3. IMPLEMENTIERUNG

### GIT INFRASTRUKTUR CODE PROJEKTSTRUKTUR

Für unsere gewählte Lösung verwenden wir Terraform als primäres IaC. Die Projektstruktur folgt GitOps-Prinzipien mit klarer Trennung der Zuständigkeiten.

GITHUB GIST: <https://gist.github.com/marcoschavez09/63ca9043b46c0732e8e086856e9110d6>

BEISPIEL REPO: [https://github.com/marcoschavez09/prodyna\\_task](https://github.com/marcoschavez09/prodyna_task)

Repos für Frontend und Backend:

## ZUSÄTZLICHE CD-TOOLS (PUSH/PULL)

Push-basiertes CD Tool: **Azure DevOps Pipelines (Native)**

Gründe für Push-basiertes:

- Centralized control – Alle Infrastrukturänderungen laufen über die CI/CD-Pipeline.
- Sicherheit – Dienstprinzipale mit Berechtigungen nach dem Prinzip der geringsten Rechte (Least-Privilege Access).
- Run pipeline history – Alle Änderungen werden in den Pipeline-Runs protokolliert.
- Approval Gates – Manuelle Freigaben für Produktions-Deployments.
- Konsistenz – Gleicher Bereitstellungsprozess für alle Umgebungen.

## WORKFLOW

```
Developer → Git Push → Pipeline Triggered → Terraform Plan →  
Approval (manual for Prod) → Terraform Apply → Infrastructure Updated
```

## BRANCHING STRATEGY.

### **GitHub Flow (für Infrastruktur + Anwendung empfohlen)**

Gründe für GitHub Flow bei Infrastructure as Code:

- Einfacher als Git Flow (“Weniger Merge Konflikte”, einfacher zu verwalten).
- Schnelle Integration ( Änderungen werden schnell in main gemerged).
- Infrastruktureränderungen sind meist klein und fokussiert (ideal für kurzlebige Branches).
- Environment separation wird durch Pipeline Gates gesteuert, nicht durch die Branch-Struktur.

```
main (production – source of truth)
  └ feature/feature-name (short-lived, < 3 days)
```

**Anmerkung:** Es gilt für Infra Repo und Application Repos (Frontend und Backend).

## Branches:

- main (Produktion)
  - Single Source of Truth für alle Umgebungen.
  - Geschützter Branch (keine direkten Pushes).
  - Alle Änderungen erfolgen über Pull Requests.
  - Wird nach Produktions-Deployments mit semantischen Versionen getaggt (z.B. v1.0.0, v1.1.0).
  - Wird in alle Umgebungen deployed (gesteuert durch Azure Pipelines).
- feature/ (Feature-Development)
  - Kurzlebige Branches für Infrastrukturänderungen (und Application).
  - Beispiel: feature/add-monitoring, feature/update-app-service-tier.
  - Wird via Pull Request in main gemerged.
  - Wird unmittelbar nach dem Merge gelöscht (auto remove).
  - Lebensdauer: < 3 Tage.

## BESPIEL WORKFLOW

### Feature branch creation

```
git checkout -b feature/add-monitoring
```

### Feature changes

```
# Add monitoring module  
git add terraform/modules/monitoring/  
git commit -m "Add Application Insights monitoring"  
git push origin feature/add-monitoring
```

### Create PR

- Automated checks: Terraform validate, fmt, plan
- Peer review (1-2 approvals required)
- Infrastructure plan review
- Merge to main

### Deployment

- Merge to main triggers pipeline
- Test Environment: Automatic deployment
- Production Environment: Manual approval gate required
- Tag release after production deployment: infra-v1.2.0

## PROTECTED BRANCHES + PULL REQUESTS + APPROVAL GATES

FEATURE BRANCH STRATEGIE FLOW.

TEAM (FEATURE DEVELOP) MERGE STRATEGIE FLOW UND BRANCH RULES.

PRODUCTION MERGE STRATEGIE FLOW UND BRANCH RULES.

**Diagram:** [https://www.figma.com/board/WUj1n6Ye4wJYxkqmuoHkxp/branching\\_team?node-id=0-1&t=jvMNVLAEMLQfrMGG-1](https://www.figma.com/board/WUj1n6Ye4wJYxkqmuoHkxp/branching_team?node-id=0-1&t=jvMNVLAEMLQfrMGG-1)

## VERSIONIERUNG UND NACHVERFOLGBARKEIT

**Problem:** Der Service Manager muss jederzeit sicherstellen können, dass die bereitgestellte Infrastrukturversion mit der Version im Code-Repository und in der Pipeline übereinstimmt

## LÖSUNGEN

### Git-Tags (Single Source of Truth):

- Semantische Versionierung:
- Tagging-Strategie:
  - Jeden Merge in den main-Branch taggen.
  - Tag-Format: infra-v<major>.<minor>.<patch>
  - Tags enthalten Metadaten: Commit-Hash, Datum, Autor.

## **State-File-Annotation (Terraform):**

- Versions-Tag im Terraform-State (terraform.tfstate) speichern.
- Terraform-Workspaces oder State-Metadaten nutzen.
- Abfrage des States, um die aktuell deployed Version zu erhalten.

## **Pipeline-Run-Informationen**

- **Azure DevOps / GitHub Actions:**

- Pipeline-Runs werden getaggt mit: Git-Commit-SHA, Git-Tag (falls vorhanden), Build-Nummer, Deployment-Zeitstempel.

- **Pipeline-Artefakte:**

- Terraform-Plan-Output speichern.
  - Angewendete State-Version speichern.
  - Pipeline-Run mit Git-Commit verknüpfen.

## **State-File-Annotation (Terraform):**

- Versions-Tag im Terraform-State (terraform.tfstate) speichern.
- Terraform-Workspaces oder State-Metadaten nutzen.
- Abfrage des States, um die aktuell deployed Version zu erhalten.

## **Pipeline-Run-Informationen**

- **Azure DevOps / GitHub Actions:**

- Pipeline-Runs werden getaggt mit: Git-Commit-SHA, Git-Tag (falls vorhanden), Build-Nummer, Deployment-Zeitstempel.

- **Pipeline-Artefakte:**

- Terraform-Plan-Output speichern.
  - Angewendete State-Version speichern.
  - Pipeline-Run mit Git-Commit verknüpfen.

## Verifizierung (Automatisierte) für den Service Manager:

- Automatisiertes Skript: Ein Skript nutzen, das automatisch Git-Tag, Terraform-State und Azure-Ressourcen prüft.
- Azure Dashboard.
- Kommandozeilen-Befehle: (entsprechende Befehle).

```
# 1. Check git tag for main branch
git describe --tags main

# 2. Check Terraform state version
terraform output -state=terraform/environments/prod/terraform.tfstate infrastructure_version

# 3. Check Azure resource tags
az resource list --tag Environment=prod --query "[0].tags.Version" -o tsv

# 4. Check pipeline run
az pipelines runs list --branch main --top 1 --query "[0].sourceVersion" -o tsv
```

## AUSGEWÄHLTE KOMPONENTEN FÜR DIE LÖSUNG

Basierend auf unserer gewählten Architektur (Azure Static Web Apps + App Service + Cosmos DB) sind dies die spezifischen PaaS- und SaaS-Komponenten:

### **Azure Static Web Apps**

- Typ: PaaS
- Zweck: Hosten der Vue.js SPA (Frontend)
- Warum: Speziell für SPAs entwickelt, kein Infrastrukturmanagement.

### **Azure App Service**

- Typ: PaaS
- Zweck: Hosten der Node.js API (Backend)
- Warum: Vollständig verwaltet, kein VM-Management, integrierte Skalierung.a

## Azure Cosmos DB (MongoDB API)

- Typ: PaaS (Database as a Service)
- Zweck: NoSQL-Datenbank für Anwendungsdaten
- Warum: Vollständig verwaltetes MongoDB, automatische Skalierung, globale Verteilung.

## SaaS-Komponenten (Software as a Service)

### Azure Key Vault

- Typ: SaaS
- Zweck: Secrets Management
- Warum: Zentrale Secrets-Verwaltung, sicher, compliant.

### Azure Application Insights

- Typ: SaaS
- Zweck: Application Performance Monitoring (APM)
- Warum: Integriertes Monitoring, kein Infrastrukturmanagement.

## Azure DevOps

- Typ: SaaS
- Zweck: Source Control und CI/CD
- Warum: Integriert mit Azure, unterstützt GitOps.

## Azure Resource Groups

- Typ: Verwaltungscontainer
- Zweck: Logische Gruppierung von Ressourcen
- Struktur: Eine Resource Group pro Umgebung.

## Azure Storage Account (Blob Storage)

- Typ: PaaS
- Zweck: Terraform State Storage
- Warum: Remote-State-Speicher, State-Locking, Versionierung.

## Azure Managed Identity

- Typ: PaaS (Identity Service)
- Zweck: Sichere Authentifizierung ohne Secrets
- Warum: Eliminiert Secret-Management, sicherer.

## TERRAFORM-PROJEKTSTRUKTUR (3 STUFEN)

- Ein einzelnes Terraform-Projekt mit Umgebungstrennung.
- Entscheidung: Ein Projekt mit umgebungsspezifischen Konfigurationen verwenden, keine separaten Projekte.

### **Warum ein Projekt?**

- Gemeinsame Module über alle Umgebungen
- Konsistenter Infrastruktur-Code
- Einfacher zu warten und aktualisieren
- Single Source of Truth
- Umgebungsunterschiede via Variablen, keine Code-Duplikation

## BRANCHING- UND MERGING-STRATEGIE

Ziel: Gleiche Version über alle Stufen + manuelle Produktionsfreigabe (bereits besprochen).

AZURE CI/CD PIPELINES BEISPIEL: [https://github.com/marcoschavez09/prodyna\\_task/tree/main/.infrastructure/pipelines/inrastructure](https://github.com/marcoschavez09/prodyna_task/tree/main/.infrastructure/pipelines/inrastructure)