

LEITFADEN FÜR ENTWICKLER

Containerisierung in Azure

Von der Entwicklung zur Bereitstellung: Azure als Fundament für containerisierte Anwendungen.

Wer Anwendungen in Azure containerisiert betreiben möchte, steht vor der Wahl: App Service für schnellen Einstieg, Container Apps für flexible Skalierung oder Azure Kubernetes Service für maximale Kontrolle. Im Folgenden wird gezeigt, wie sich Container-Images für den produktiven Einsatz erstellen lassen, welche Best Practices beim Betrieb in Azure gelten und anhand welcher Kriterien sich die passende Plattform auswählen lässt.

Wie Container den Betrieb in Azure verbessern

Container sind längst keine neue Technologie mehr, doch viele Teams setzen sie noch nicht umfassend ein oder meiden sie ganz, weil die Vorteile nicht klar sind oder die Einstiegshürden abschreckend wirken. Dabei bilden Container heute die Grundlage für zahlreiche moderne Cloud-Dienste und ermöglichen eine standardisierte, portable und reproduzierbare Art und Weise, Anwendungen bereitzustellen. Wer Container versteht und nutzt, profitiert von konsistenter Laufzeitkonfiguration, vereinfachten Deployments und einer klaren Trennung zwischen Anwendung und Infrastruktur.

Ein Container bündelt die Anwendung zusammen mit allen Abhängigkeiten in einem isolierten Paket. Frameworks, Bibliotheken und eine definierte Runtime sind darin enthalten [1]. Dieses Paket kann auf jeder Plattform ausgeführt werden, die Container unterstützt. Ob lokal, in einer privaten Cloud oder in Azure, spielt keine Rolle. Entwickler müssen nicht mehr sicherstellen, dass die Zielumgebung exakt die richtige Runtime oder zusätzliche Tools installiert hat. Alles, was die Anwendung benötigt, steckt im Image. Dadurch entsteht echte Portabilität. Ein einmal gebautes Image kann lo-

kal für Entwicklung und Tests genutzt, an andere Teams weitergegeben oder in der Cloud skaliert werden, ohne dass sich die Umgebung unterscheidet.

Für .NET-Teams bedeutet das konkret: Keine Abhängigkeit von installierten .NET-Versionen auf dem Host. Das Image enthält die benötigte .NET-Runtime und die kompilierte Anwendung. Der Host benötigt nur eine Container-Run-time. Das vereinfacht Deployments erheblich und reduziert Fehlerquellen durch inkonsistente Umgebungen.

Container bieten auch Isolation. Jeder Container läuft in seiner eigenen Umgebung, verwaltet seine Dateien und Netzwerkeinstellungen und vermeidet Konflikte mit anderen Anwendungen. Das macht den Betrieb stabiler und reduziert Seiteneffekte. Gleichzeitig sind Container leichtgewichtig und starten schnell, weil sie keine vollständige virtuelle Maschine benötigen, sondern nur die definierte Schicht aus Betriebssystem und Anwendung. Für die Entwicklung bedeutet das: Abhängigkeiten wie Datenbanken oder Messaging-Systeme lassen sich als Container hochziehen, was lokale Tests und CI/CD-Pipelines erheblich vereinfacht. Eine Anwendung kann inklusive aller benötigten Services in einer Pipeline gestartet und getestet werden, ohne komplexe Setup-Skripte.

Der Bau eines Container-Images erfolgt typischerweise über ein Dockerfile [2], das als Installationsanleitung dient (siehe Listing 1). Es definiert zunächst ein Basis-Image. Ein Basis-Image ist ein vorgefertigtes Image, das eine bestimmte Laufzeitumgebung bereitstellt oder grundlegende Betriebssystemfunktionen enthält.

Für .NET gibt es verschiedene Optionen: Standard-Images mit vorinstallierter Runtime, Alpine-basierte Images für redu-

● Listing 1: Dockerfile für eine einfache ASP.NET-Core-Anwendung

```
# ASP.NET Core 10 runtime
# Noble = Ubuntu 24.04
# Chiseled = Minimales Image mit nur den notwendigsten
# Komponenten
FROM mcr.microsoft.com/dotnet/aspnet:10.0-noble-chiseled

# Arbeitsverzeichnis im Container festlegen
WORKDIR /app

# Kopiere die Binaries aus dem lokalen Verzeichnis in
# den Container
# Anwendung muss vorher lokal gebaut worden sein
COPY ./publish .

# Metadaten für das Image, dass die Anwendung auf Port
# 8080 läuft
EXPOSE 8080

# Starte die Anwendung über die .NET Runtime
ENTRYPOINT ["dotnet", "WebApi.dll"]
```

● Listing 2: Image erstellen und nach ACR pushen mit Azure CLI

```
# Docker Image lokal bauen
docker build -t webapi:1.0 ./src/WebAPI

# Image mit dem ACR-Registry-Namen taggen,
# damit es in das richtige Repository gepusht wird
docker tag webapi:1.0
    $registryName.azurecr.io/webapi:1.0

# Bei ACR anmelden mit dem aktuellen Azure CLI Account
az acr login --name $registryName

# Image in ACR pushen
docker push $registryName.azurecr.io/webapi:1.0

# Images in ACR listen und löschen für Data Retention
az acr repository show-tags --name $registry
    --orderby time_desc
az acr repository delete --name $registry
    --image webapi:1.0 --yes
```

zierte Größe oder Chiseled Images [3], die nur die absolut notwendigen Abhängigkeiten enthalten. Kleinere Images haben mehrere Vorteile: Sie reduzieren die Angriffsfläche für potenzielle Sicherheitslücken und verkürzen die Zeit, die Azure-Dienste benötigen, um das Image zu laden. Das wirkt sich direkt auf Startzeiten und Skalierungsgeschwindigkeit aus.

Nach dem Basis-Image ergänzt das Dockerfile die Anwendung sowie optionale Abhängigkeiten. Das fertige Image enthält alles, was für den Betrieb notwendig ist, und kann beliebig oft gestartet werden.

Ein Image muss nach dem Erstellen abgelegt werden, damit es von Plattformdiensten genutzt werden kann. Hierfür dient eine Registry als zentrale Quelle. Diese kann entweder selbst gehostet oder als Managed Service in Azure genutzt werden.

Von lokal in die Cloud

Public Registries wie Docker Hub sind gut für Open Source, aber für Anwendungen ungeeignet, die nicht für alle offen im Netz stehen sollen. Für den Betrieb in Azure bietet sich die Azure Container Registry (ACR) an [4]. Diese ist eine private Registry, die als Managed Service bereitsteht und sich nahtlos in die Azure-Plattform integriert.

ACR gibt es in drei Varianten: „Basic“ ist für die meisten Szenarien ausreichend, solange nicht mehr als zwei Webhooks benötigt werden. Mit Webhooks lassen sich bestimmte Ereignisse erkennen und darauf reagieren. Beispielsweise kann die Registry automatisch eine Pipeline starten, sobald ein neues Image hochgeladen wird. „Standard“ erweitert die Anzahl auf zehn Webhooks und bietet höhere Durchsatzlimits für Lese- und Schreiboperationen. „Premium“ stellt zusätzliche Features bereit: Geo-Replikation für globale Verteilung, Private Endpoints für vollständige Netzwerkisolation und erweiterte Sicherheitsfeatures für komplexe Umgebungen. Für die meisten Teams ist die Basic-Variante der richtige Einstieg. Premium lohnt sich für Enterprise-Szenarien mit strengen Compliance-Anforderungen oder globaler Infrastruktur.

Ein zentraler Vorteil von ACR ist die Integration mit Azure RBAC. Dienste wie App Service, Container Apps oder AKS können über Managed Identity direkt auf die Registry zugreifen. Statische Zugangsdaten in Pipelines oder Konfigurationen sind nicht erforderlich. Das reduziert Sicherheitsrisiken

und vereinfacht den Betrieb erheblich. Für den Zugriff wird dem Dienst die Rolle *AcrPull* zugewiesen. Diese Zuweisung kann über das Azure Portal, das Azure CLI oder Infrastructure as Code erfolgen.

Tags sind das zentrale Werkzeug für Ordnung und Versionierung in der Registry. Ein Tag ist eine Bezeichnung, die einem bestimmten Image zugeordnet wird. Derselbe Image-Name kann mehrere Tags haben, um unterschiedliche Versionen oder Deployment-Kanäle zu kennzeichnen. Typische Tagging-Strategien sind: Versionsnummern wie *1.4.2* für spezifische Releases, Major-Minor-Tags wie *1.4*, die immer auf die neueste Patch-Version zeigen, Kanäle wie *stable*, *canary* oder *dev* für Deployment-Stufen und *latest* für die aktuellste Version [5]. Wichtig zu verstehen: Viele Azure-Dienste erkennen neue Images nicht automatisch, wenn das Tag unverändert bleibt. Selbst bei einem erneuten Deployment wird ein Image mit dem gleichen Tag oft nicht neu gezogen. Deshalb sollten produktive Deployments immer eine eindeutige Versionsnummer verwenden. Das stellt sicher, dass Deployments reproduzierbar sind und keine alten Artefakte weiterlaufen.

Images verbrauchen Speicher und verursachen Kosten. ACR bietet in der Premium-SKU eine automatische Data Retention. Diese Funktion entfernt automatisch Images, die kein Tag mehr haben, nach einer konfigurierbaren Aufbewahrungsfrist. Damit diese Retention sinnvoll genutzt werden kann, muss jedoch zunächst ein Mechanismus existieren, der nicht mehr benötigten Images ihre Versions-Tags entzieht. Für Teams ohne Premium-SKU oder mit individuellen Anforderungen bleibt die automatische Bereinigung über eine Pipeline die flexiblere Lösung (siehe Listing 2). Das hält die Registry schlank, und die Kosten bleiben kalkulierbar.

Der Push in die Registry erfolgt typischerweise automatisiert über CI/CD-Pipelines. Azure DevOps und GitHub Actions bieten fertige Tasks für den Image-Build und nutzen anschließend einen normalen *docker push*-Befehl, wie er auch für Docker Hub gilt (siehe Listing 2). Es ist kein spezieller Task für ACR erforderlich, da die Authentifizierung über Azure CLI oder Service Connections erfolgt.

Für das Deployment zwischen Umgebungen gibt es zwei gängige Ansätze. Variante eins nutzt eine Registry pro Umgebung wie *Dev*, *Test* und *Prod*. Bei einem Deployment in ►

die nächste Umgebung wird das Image aus der vorherigen Registry gepullt und in die nächste Registry gepusht, sodass das Image nicht neu gebaut werden muss.

Variante zwei verwendet eine zentrale Registry, die alle Versionen enthält. Deployments zeigen nur noch auf diese zentrale Registry und nutzen das dort hinterlegte Image für die jeweilige Umgebung. Dieser Ansatz ist einfacher, vermeidet redundante Pushes und wird für die meisten Teams empfohlen. Eine Registry mit klarer Tagging-Strategie reicht in der Regel aus.

Wer alles in eine zentrale Registry schreibt, hat unter Umständen den Bedarf, dass nicht alle Anwendungen auch alle Images sehen dürfen. Über sogenannte Scope Maps und Tokens kann hier nochmal feingranular berechtigt werden, wer auf welches Repository Zugriff hat [6]. Ein Repository ist in diesem Fall eine logische Gruppierung von Images innerhalb der Registry, typischerweise pro Anwendung. Wer Managed Identity verwendet, kann in seiner Container Registry auch ABAC (Attribute-Based Access Control) nutzen, um Zugriffe noch feiner zu steuern [7]. Hier kann selbiges wie bei den Scope Maps gesteuert werden, aber mit konkreten Entra-Identitäten wie zum Beispiel einer Managed Identity auf konkrete Repositories. Mit ABAC stehen in Azure andere Rollen zur Auswahl als die bisherige *AcrPull*-Rolle.

Wer seine Images einmal in ACR hat, sollte trotz allem regelmäßig auf Sicherheit prüfen. Dafür muss Microsoft Defender for Containers in der Azure Container Registry aktiviert werden [8]. Nach der Aktivierung wird bei jedem Push automatisch ein Scan ausgelöst. Zusätzlich werden alle Images, die in den letzten 30 Tagen gepullt wurden, wöchentlich erneut geprüft. Das Pricing ist einfach: Pro Image wird einmal abgerechnet, unabhängig davon, wie oft es gescannt wird. Ein wöchentlicher Rescan verursacht keine zusätzlichen Kosten. Die Ergebnisse der Scans können im Azure Portal eingesehen oder direkt in der Pipeline abgefragt werden. Damit lässt sich ein Release Gate einrichten, das die Pipeline abbricht, wenn im gepushten Image eine Schwachstelle gefunden wurde. So wird verhindert, dass unsichere Images in produktive Umgebungen gelangen.

ACR bietet auch die Möglichkeit, Images direkt zu bauen [9]. Über ACR-Tasks kann ein Build ohne externe Pipeline erfolgen. Das ist praktisch für Teams ohne bestehende CI/CD-Infrastruktur. Allerdings verlängert sich die Build-Zeit, und die Nutzung verursacht zusätzliche Kosten. Für produktive Szenarien empfiehlt sich eine dedizierte CI/CD-Pipeline.

Typische Stolpersteine sind fehlende Role Assignments, die zu Pull-Fehlern führen, eine unklare Tagging-Strategie, Deployment zwischen verschiedenen Stages oder eine falsche SKU-Wahl, die unnötige Kosten erzeugt oder Features ver-

Azure App Service Container Deployment (Bild 1)

missen lässt. Sobald die Registry steht und die Images dort liegen, können wir diese in Azure-Diensten nutzen.

Als Nächstes sehen wir uns an, wie wir ein Image aus ACR in einen App Service deployen und damit klassische ZIP-Deployments ablösen.

Container in Azure App Service

Azure App Service ist für viele Teams der erste Schritt, um Container in der Cloud zu betreiben, ohne mit der Komplexität eines Orchestrators wie Kubernetes konfrontiert zu sein. Der Dienst ist vollständig gemanagt und bietet eine vertraute Umgebung für Web-Apps und APIs, die bisher über ZIP-Deployments bereitgestellt wurden. Der entscheidende Unterschied besteht darin, dass App Service statt Dateien ein Container-Image referenziert, das in einer Registry wie ACR liegt.

Der Zugriff auf ACR erfolgt über Managed Identity oder über den Admin Login, wobei Ersteres die präferierte und sicherere Variante ist. Die Image-Referenz wird in den App Settings beziehungsweise beim App Service Deployment hinterlegt, typischerweise als vollständiger Pfad inklusive Tag (siehe Bild 1) [10]. Wichtig zu verstehen ist, dass Dienste ein neues Image nur ziehen, wenn sich das Tag ändert. Wer *latest* verwendet, riskiert, dass ein Deployment weiterhin das alte Image nutzt. Auch ein Rollback ist dann nicht mehr einfach möglich, da hier nicht auf eine generische letzte Version referenziert werden kann, sondern die konkrete Versionsnummer des Vorgängers genutzt werden muss.

Der typische CI/CD-Flow ist einfach: Build des Images, Push nach ACR und Deployment durch Aktualisierung der Image-Referenz in den App Settings. Azure DevOps und GitHub Actions bieten Tasks, die diese Schritte automatisieren. Ein Deployment löst keinen erneuten Build aus, sondern referenziert nur das neue Image aus ACR.

Die Konfiguration unterscheidet sich deutlich von klassischen Deployments. Token-Ersetzungen in *appsettings*-Dateien ent-

fallen, weil das Image unveränderlich ist. Einstellungen werden über Umgebungsvariablen bereitgestellt, die App Service in die Container-Umgebung injiziert. Für Secrets empfiehlt sich die Integration mit Azure Key Vault über Managed Identity.

Eine Besonderheit ist die Port-Konfiguration. App Service muss wissen, auf welchem Port die Anwendung im Container lauscht. Dieser Wert wird über die App Setting `WEBSITES_PORT` gesetzt. Ohne korrekte Port-Angabe bleibt der Container erreichbar, aber der Ingress funktioniert nicht, und es kommen keine Anfragen von außen an.

App Service unterstützt auch Multi-Container-Deployments über Docker Compose. Damit lassen sich Szenarien wie Sidecars für Caching oder Logging realisieren. Sidecars sind Container, die neben dem eigentlichen Hauptcontainer mitlaufen und den Hauptcontainer unterstützen. Ein typisches Beispiel ist ein API, das zusammen mit einem Redis-Container läuft, oder ein Logging-Sidecar, das alle Anwendungs-Logs zentral sammelt. Dabei kann nur ein einziger Container externen Traffic empfangen. Alle anderen Container sind intern erreichbar und dienen als Hilfsdienste. Die Skalierung erfolgt horizontal über den App Service Plan. Jede Instanz startet alle definierten Container, was bedeutet, dass Skalierung die gesamte Container-Gruppe multipliziert. Startzeiten wirken sich direkt auf Autoscaling aus, weil eine neue Instanz erst dann Traffic übernimmt, wenn alle Container bereit sind.

Das Monitoring ähnelt dem des klassischen App Service, aber mit Container-Spezifika. Konsolen-Ausgaben können über den Live-Log-Stream oder Diagnoseeinstellungen eingesehen werden. Für die Trennung zwischen Container-Logs und Applikations-Logs ist wichtig zu verstehen, dass Konsolen-Ausgaben die Container-Runtime betreffen, während Application Insights Telemetrie auf Anwendungsebene sammelt. Für tiefere Analysen empfiehlt sich die Integration mit Application Insights, die auch in Container-Szenarien Metriken wie Request-Zeiten, Fehlerquoten und Abhängigkeiten liefert.

Networking ist einfach, aber mit Einschränkungen. App Service stellt einen Ingress für genau einen Container bereit. Alle anderen Container sind nur intern erreichbar. Wer mehrere Container mit externem Zugriff benötigt, stößt hier an Grenzen und braucht unterschiedliche App Services dafür.

Die Kosten richten sich nach dem gewählten App Service Plan. Ein Plan verursacht Kosten, solange er existiert, unabhängig von der tatsächlichen Nutzung. Für Kostenoptimierung empfiehlt sich die Wahl eines angemessenen Plans, der Einsatz von Autoscaling, um Über- oder Unterdimensionierung zu vermeiden, die Nutzung von Dev/Test-Plänen für nicht produktive Umgebungen sowie Saving Plans und Reserved Instances.

Stolpersteine sind falsche, den Ingress blockierende Port-Konfigurationen, lange Start-up-Zeiten bei falscher Planwahl und die Annahme, Multi-Container sei beliebig skalierbar.

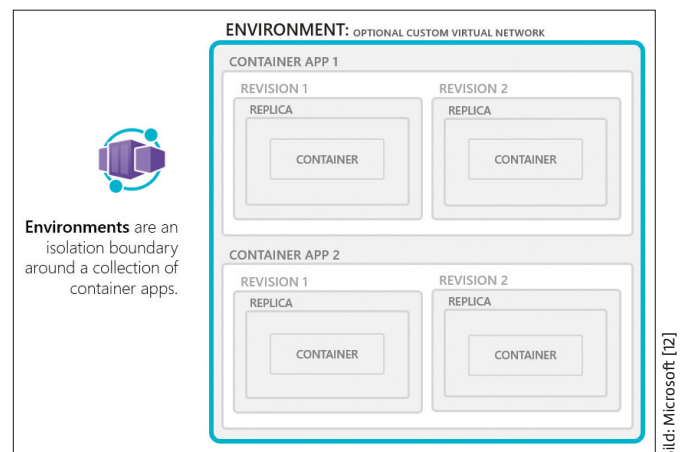
App Service ist ein solider Einstieg in Container-Hosting, bleibt aber auf einfache Szenarien beschränkt. Wer dynamische Skalierung, Event-getriebene Workloads oder komplexe Microservice-Architekturen benötigt, stößt bald an Grenzen.

Die Weiterentwicklung ist Azure Container Apps, eine Plattform, die Container-Workloads mehr Flexibilität für moderne Anwendungen bietet.

Azure Container Apps für Scale-to-Zero-Szenarien

Azure Container Apps sind die Weiterentwicklung für Container-Workloads in Azure. Sie bauen technisch auf Azure Kubernetes Service (AKS) auf, abstrahieren aber die Komplexität von Clusterverwaltung, Netzwerk und Add-ons [11]. Damit bieten sie die Flexibilität einer Kubernetes-basierten Plattform, ohne dass Teams sich um Nodes, Upgrades oder Ingress-Controller kümmern müssen. Wer maximale Kontrolle über Netzwerk, Policies und Cluster-Add-ons benötigt, wird mit AKS mehr Flexibilität haben. Container Apps sind der einfachere Einstieg.

Ein zentrales Konzept bei Container Apps sind Container App Environments. Ein Environment ist die logische Hülle für eine oder mehrere Container Apps und bildet die Grundlage für gemeinsame Features wie VNET-Integration, interne Kommunikation, Logging und Zertifikate (siehe Bild 2) [12]. Innerhalb eines Environments können Apps miteinander sprechen, ohne externen Ingress zu nutzen. Zu beachten ist auch, dass im Environment ein Maintenance Window gesetzt werden kann. Dabei kann vorgegeben werden, zu welchem Zeitpunkt Azure im Hintergrund ein Update von zum Beispiel Kubernetes durchführen kann. Wer hier eine Container App mit nur einer Instanz hat, kann von Downtime betroffen sein. In der Pra-



Aufbau und Struktur von Azure Container App (Bild 2)

xis reicht für viele Teams ein einzelnes Environment pro Stage (Dev, Test, Prod), in dem alle Apps einer Umgebung laufen.

Container Apps unterscheiden sich deutlich von App Service. Während App Service feste Pläne und horizontale Skalierung pro Instanz nutzt, skaliert Container Apps dynamisch auf Basis von Last und Ereignissen und im Zweifel auch bis auf null. Dieses Zero-Scaling ist ein entscheidender Vorteil für Workloads mit variabler Nutzung, weil keine Kosten entstehen, wenn keine Anfragen verarbeitet werden. Auch das Deployment ist flexibler. Statt Slots gibt es ein Revisionsmodell. Jede neue Version erzeugt eine Revision, die entweder allein aktiv ist (Single Revision, wie bei App Service) oder parallel zur alten Revision läuft. Bei parallelen Revisionen lässt sich der Traffic prozentual verteilen. Läuft alles stabil, wird der Traffic schrittweise erhöht. Bei Problemen kann ►

ein sofortiger Rollback erfolgen, ohne dass Nutzer flächendeckend betroffen sind.

Der Zugriff auf Container-Images aus ACR erfolgt wie bei App Service über Managed Identity oder Admin Login. Auch der Zugriff auf andere Azure-Ressourcen wie Key Vault oder Storage erfolgt über dieselbe Identity. Mehrere Container pro Container App sind ebenfalls wie beim App Service möglich, etwa für Sidecars oder anderen Szenarien.

Die CI/CD-Integration ist einfach. Die Images liegen bereits in ACR. Ein Deployment auf Container Apps erzeugt lediglich eine neue Revision mit der referenzierten Image-Version [13]. Auch hier sollte darauf geachtet werden, direkt die Versionsnummer und kein allgemeines Tag wie *latest* zu nutzen, da auch Container App nicht einfach mitbekommt, ob sich das Image geändert hat. Die Pipeline muss keinen neuen Build ausführen, sondern nur die Revision aktualisieren. Das macht Blue-Green- oder Canary-Deployments effizient und risikoarm. Sowohl in Azure DevOps Pipelines als auch GitHub Actions gibt es vorgefertigte Tasks und Actions für das Deployment eines Images nach Container Apps. Wer hier aber einen Flow abbilden will, der nach und nach basierend auf Health Checks und Ähnlichem den Traffic auf die neueste Version erhöht, muss selbst über zum Beispiel das Azure CLI Hand anlegen.

Wer seine Container App über Infrastructure as Code (IaC) bereitstellen möchte, sollte beachten, dass beim Erstellen bereits das Container Image angegeben werden und es auch existieren muss. Das ist ein Henne-Ei-Problem, da die Container Registry meist auch mit der Infrastruktur bereitgestellt wird. Eine Lösung ist, in IaC ein Dummy-Image anzugeben, zum Beispiel das Microsoft Image für eine ASP.NET-Core-Beispielanwendung. Dieses wird dann später in der CI/CD-Pipeline mit dem tatsächlichen Image überschrieben. Somit laufen die Infrastruktur und die Container App, bis die eigentliche Applikation gebaut und deployt wird.

Die Skalierung in Container Apps erfolgt über KEDA (Kubernetes Event-Driven Autoscaler). Container Apps können auf HTTP-Requests, Queue-Nachrichten wie zum Beispiel von einer Azure Service Bus Queue oder benutzerdefinierte Metriken reagieren (siehe Bild 3) [14]. Min- und Max-Replicas sowie Concurrency-Einstellungen steuern die Grenzen. Für Queue-basierte Workloads lässt sich KEDA so konfigurieren, dass pro N Nachrichten ein Replica gestartet wird. Das erinnert an den Consumption Plan von Azure Functions, bei dem auch je nach Trigger-Last die Instanzen automatisch skaliert werden können. Für die Skalierung lassen sich zusätzlich das Polling Interval und die Cooldown Period definieren, um das Skalierungsverhalten zu steuern. Das Polling Interval bestimmt, wie häufig KEDA die Event-Quellen prüft, um aktuelle Metriken zu ermitteln. Die Cooldown Period legt fest, wie lange nach dem letzten beobachteten Event gewartet wird, be-

vor die Anwendung auf die minimale Anzahl von Replicas zurückskaliert. Das verhindert unnötiges Hoch- und Herunterskalieren bei kurzfristigen Lastschwankungen. Die Wahl dieser Werte sollte sich am Lastprofil der Anwendung orientieren.

Abgerechnet wird nach Nutzung. Das Pricing basiert auf der Anzahl vCores, der Größe des Speichers und darauf, wie lange die Instanzen laufen und wie viele Anfragen gestellt werden. Zusätzlich gibt es ein Freikontingent: Zwei Millionen Requests und 400 000 GB-Sekunden Memory sowie 180 000 vCPU-Sekunden sind kostenlos. Das entspricht zwei vCores und 4 GB RAM im Dauerbetrieb für 24 Stunden. Für Workloads mit geringer Nutzung bedeutet das oft null Kosten. Ein App Service Basic Plan kostet dabei schon deutlich mehr. Der Freikontingent wird zwischen allen Container Apps im selben Azure-Abonnement geteilt.

Ein weiteres Alleinstellungsmerkmal ist die Dapr-Integration. Dapr abstrahiert Cloud-Komponenten wie Pub/Sub, Secret Management oder State Stores. Die Anwendung muss nicht wissen, ob Pub/Sub über Azure Service Bus oder ein anderes Backend läuft. Sie spricht nur das Dapr-API an. Das erhöht die Portabilität und unterstreicht den Container-Vorteil. Die App ist unabhängig von der konkreten Cloud-Implementierung. Dapr wird als Sidecar aktiviert und läuft neben der Anwendung im selben Replica. Die Konfiguration erfolgt über Component-Definitionen, die Pub/Sub, Bindings oder State Stores deklarativ beschreiben.

Die Konfiguration erfolgt wie gewohnt über Umgebungsvariablen und Secrets. Key Vault lässt sich direkt integrieren, oder Secrets können anders als beim App Service auch sicher direkt in der Container App abgelegt werden.

Networking ist flexibel. Ingress kann extern oder intern sein. Externer Ingress macht die App über das Internet erreichbar, interner Ingress nur innerhalb des Environments oder über ein Azure VNET. Die VNET-Integration ermöglicht private Kommunikation mit anderen Azure-Ressourcen wie SQL-Datenbanken oder Storage-Accounts, sofern diese nicht öffentlich erreichbar sind. Alle Apps im selben Environment können über ihre App-Namen miteinander kommunizieren, ohne DNS oder Load Balancer konfigurieren zu müssen.

Monitoring und Logging sind umfassend. Container-Logs und Diagnose-Logs landen automatisch in Log Analytics, wenn das Environment mit einem Log Analytics Workspace verbunden ist.

Typische Stolpersteine sind Cold Starts bei Min-Replica, die etwas dauern können. Die erste Anfrage nach einer Idle-Phase kann mehrere Sekunden in Anspruch nehmen, abhängig von Image-Größe und Startzeit der Anwendung. Für latenzempfindliche Workloads sollte die Mindestanzahl an Replicas auf eins gesetzt werden. Das verursacht konstante Kosten, verhindert aber Cold Starts. Die Komplexität bei Revisions- und Ingress-Konfiguration ist ein weiteres Thema. Parallele Revisionen erfordern ein klares Verständnis von Traffic-Splitting und Rollback-Strategien. Falsch konfigurierte Ingress-Regeln können dazu führen, dass Apps nicht erreichbar sind oder ungewollt öffentlich exponiert werden.

Container Apps sind ideal für dynamische Workloads, APIs mit variabler Last und Event-getriebene Szenarien. Wer jedoch volle Kontrolle über Netzwerk, Policies und Cluster-Add-ons benötigt oder komplexe Microservice-Landschaften betreibt, sollte den Schritt zu AKS erwägen. Dort lassen sich alle Kubernetes-Funktionen ohne Einschränkungen nutzen, allerdings zum Preis höherer Betriebsaufwände.

Maximale Kontrolle mit AKS

Azure Kubernetes Service (AKS) ist die Plattform für Teams, die maximale Kontrolle über ihre Container-Workloads benötigen. Im Gegensatz zu App Service und Container Apps, die viele Aspekte abstrahieren, bietet AKS die volle Kubernetes-Funktionalität inklusive eigener Ingress-Controller, Policies, Netzwerk-Design und Add-ons [15]. Das bedeutet einerseits mehr Flexibilität, auf der anderen Seite aber auch mehr Verantwortung. Cluster-Upgrades, Node-Pool-Management und RBAC müssen aktiv verwaltet werden. Wer die Vorteile von Container Apps schätzt, aber mehr Kontrolle über Infrastruktur, Netzwerk, Konfiguration und anderes benötigt, findet in AKS die nächste Stufe.

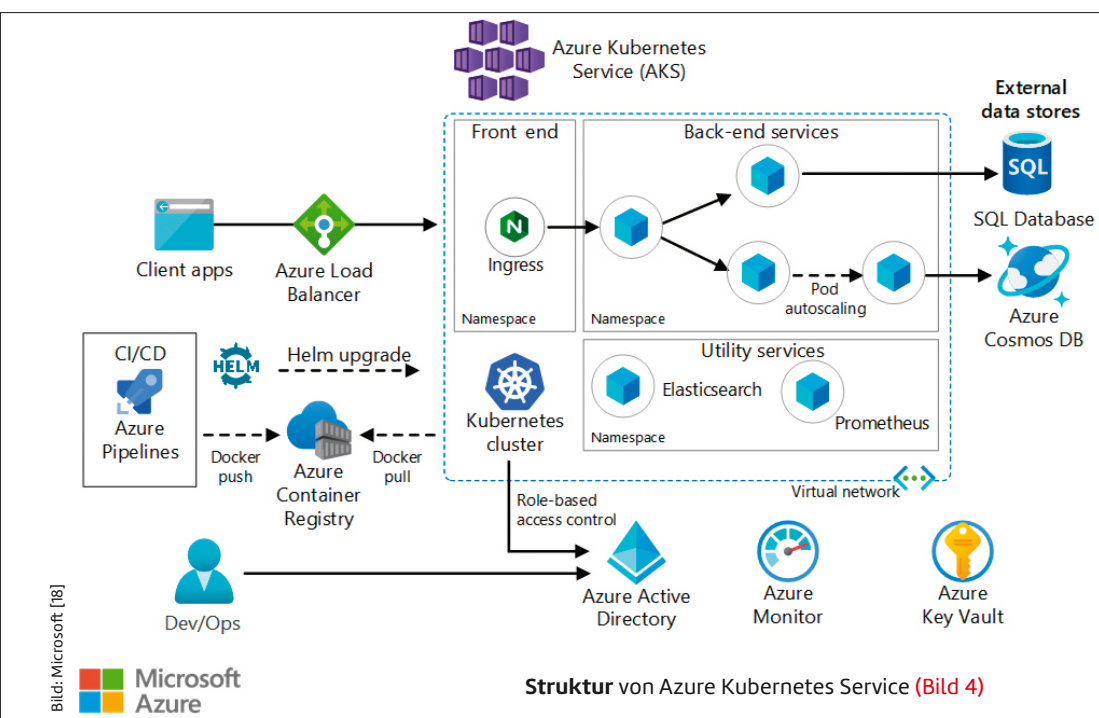
Der Einstieg beginnt mit der Bereitstellung eines AKS-Clusters, das Images aus ACR bezieht. Für die Berechtigung, Images aus ACR zu pullen, wird die Managed Identity des Node Pools genutzt, in dem die Container später laufen. Diese Variante ist einfach einzurichten und funktioniert für die meisten Szenarien. Für granulare Berechtigungen auf Pod-Ebene kommt Workload Identity zum Einsatz. Hierbei wird jeder Pod mit einer eigenen Workload Identity ausgestattet, die nur auf die Ressourcen zugreifen kann, die explizit zugewiesen wurden. Das ist besonders relevant, wenn verschiedene Container unterschiedliche Azure-Ressourcen wie Storage Accounts oder Key Vaults ansprechen müssen. Workload Identity ist der moderne, empfohlene Ansatz für neue Deployments, da er dem Prinzip der geringsten Rechte folgt und sich nahtlos in Kubernetes Service Accounts integriert.

Deployments erfolgen deklarativ über Helm-Charts [16] oder Kustomize [17]. Ein Deployment nutzt entweder das kubectl CLI Tool, um Manifeste anzuwenden, oder Helm installiert das Chart mit den passenden Werten für die Zielumgebung (siehe Bild 4) [18]. Anders als bei Container Apps erzeugt ein neues Deployment keine Revision, sondern aktualisiert die Kubernetes-Ressourcen direkt. Das gibt maximale Flexibilität, erfordert aber ein klares Versions- und Rollback-Konzept. Mit Helm lässt sich direkt ein Rollback durchführen; bei direkten kubectl-Deployments muss das Team entweder auf frühere Manifest-Versionen zurückgreifen oder Blue-Green-Deployments über den Ingress-Controller steuern.

GitOps bringt hier einen entscheidenden Vorteil. Tools wie ArgoCD [19] oder Flux [20] überwachen ein Git-Repository, in dem die gewünschten Kubernetes-Manifeste oder Helm-Charts liegen. Jede Änderung im Repository wird automatisch auf den Cluster angewendet. Das Repository wird zur Single Source of Truth. Entwickler pushen keine Manifeste

mehr direkt auf den Cluster, sondern committen ihre Änderungen ins Git-Repository. ArgoCD synchronisiert den Cluster-Zustand kontinuierlich mit dem Repository. Rollbacks funktionieren über einfache Git-Reverts. Dieser Ansatz trennt Build und Deployment sauber. Das reduziert die Komplexität in den Pipelines und macht Deployments nachvollziehbar und audittierbar.

Skalierung ist ein zentrales Thema. AKS unterstützt den Horizontal Pod Autoscaler (HPA) für CPU, Memory oder benutzerdefinierte Metriken sowie den Cluster Autoscaler für Node-Pools [21]. Aber auch KEDA, das wir schon von Container Apps kennen, kann hier verwendet werden, ►



um dynamisch auf Metriken zu skalieren. Diese Mechanismen sind mächtiger als die Autoscaling-Optionen in App Service oder Container Apps, erfordern aber ein gutes Verständnis von Ressourcenlimits und Requests. Wer hier sauber arbeitet, kann dynamische Skalierung mit granularer Kontrolle kombinieren. Der HPA oder KEDA beobachtet Metriken wie durchschnittliche CPU-Auslastung über alle Pods eines Deployments und skaliert die Anzahl der Replicas entsprechend. Der Cluster Autoscaler fügt bei Bedarf neue Nodes hinzu oder entfernt ungenutzte Nodes. Beide Mechanismen greifen ineinander, sodass sowohl Anwendungs- als auch Infrastruktur-Ebene dynamisch skalieren.

Automatisierung ist essenziell, um die Komplexität zu beherrschen. Bicep eignet sich für die Bereitstellung von Cluster-Ressourcen wie den AKS-Cluster selbst, Node-Pools, RBAC-Rollen und die Anbindung an ACR.

Für App-Deployments sind Helm-Charts der Standard. Unterschiedliche Environments wie Dev, Staging und Production nutzen separate Dateien oder überschreiben Werte zur Deployment-Zeit. Helm verwaltet auch Dependencies zwischen Charts, sodass komplexe Anwendungen mit mehreren Services als Paket deployt werden können. In Kombination mit GitOps wird Helm zum deklarativen Deployment-Tool, das den Cluster-Zustand jederzeit reproduzierbar macht.

Monitoring und Logging erfolgen über Azure Monitor und Container Insights, ergänzt durch Application Insights für Anwendungstelemetrie.

Die Netzwerkconfiguration ist deutlich flexibler. Ingress-Controller wie NGINX oder Azure Application Gateway übernehmen externen Traffic und leiten ihn an die passenden Services weiter (siehe [Bild 4](#)) [22]. Zusätzliche Services steuern die interne Kommunikation zwischen Pods oder die externe Erreichbarkeit und auch das Load Balancing von Pods. VNET-Integration und Private Endpoints sind vollständig konfigurierbar. Das eröffnet Möglichkeiten für komplexe Routing-Szenarien, die in App Service oder Container Apps nicht verfügbar sind. Teams können Traffic-Splitting für Canary-Deployments konfigurieren, Ingress-Regeln für mehrere Domains verwalten und Verschlüsselung zentral am Ingress-Controller durchführen.

Kosten entstehen durch Node-Kapazitäten, Load Balancer, Storage und Add-ons. Spartipps sind Spot-Nodes für nichtkritische Workloads, Autoscaling zur Vermeidung ungenutzter Kapazitäten und schlanke Images für schnellere Pull-Zeiten und geringeren Storage-Verbrauch. Im Gegensatz zu Container Apps gibt es kein verbrauchsorientiertes Modell. AKS verursacht Infrastrukturkosten, auch wenn keine Workloads laufen. Das muss bei der Planung berücksichtigt werden. Die Cluster-Control-Plane ist in vielen SKUs kostenlos, aber die Nodes, Load Balancer und Disks verursachen kontinuierliche Kosten. Node-Pools in der richtigen Größe sowohl horizontal als auch vertikal zu skalieren und das Herunterfahren von Dev-Clustern außerhalb der Arbeitszeiten sind praktische Maßnahmen zur Kostenoptimierung.

Typische Stolpersteine sind komplexe RBAC-Konfigurationen, Cluster-Upgrades und die Verwaltung von Ressourcenlimits. Die Lernkurve ist höher als bei App Service oder Con-

tainer Apps. Wer Kubernetes nicht kennt, sollte mit einem Proof of Concept starten und die Betriebsprozesse frühzeitig etablieren. Besonders kritisch sind fehlende Resource-Requests und Limits in den Deployments. Ohne diese Angaben kann der Scheduler die Pods nicht optimal verteilen, und einzelne Pods können Nodes überlasten. RBAC-Fehler führen oft dazu, dass Pods keine Secrets lesen oder nicht auf andere Services zugreifen können. Cluster-Upgrades erfordern Planung, da sowohl Control-Plane als auch Node-Pools aktualisiert werden müssen. Microsoft bietet automatisierte Upgrade-Kanäle, aber Teams sollten regelmäßig testen, ob ihre Workloads nach einem Upgrade noch funktionieren.

AKS ist die richtige Wahl für Teams, die volle Kontrolle und Flexibilität benötigen. Typische Szenarien sind komplexe Microservice-Landschaften mit unterschiedlichen Skalierungs- und Netzwerkanforderungen, hybride Architekturen, die On-Premises und Cloud verbinden, oder besondere Anforderungen an Netzwerk- und Compliance, die granulare Policies erfordern. Wer hingegen einfache Deployments und Konfiguration sucht, bleibt bei App Service oder Container Apps. AKS ist kein Kubernetes light, sondern Kubernetes mit Azure-Integration. Teams sollten bereit sein, in Wissen und Betriebsprozesse zu investieren, um die Plattform produktiv zu nutzen.

Fazit

Container sind längst mehr als ein Trend. Sie bilden einen zentralen Baustein moderner Anwendungsarchitekturen. In diesem Artikel haben wir den Weg von der Containerisierung einer .NET-Anwendung über den Aufbau einer Registry bis hin zum Betrieb auf drei Azure-Plattformen betrachtet. App Service, Container Apps und AKS verfolgen alle das gleiche Ziel: Anwendungen reproduzierbar und skalierbar bereitzustellen. Die Unterschiede liegen in den Ansätzen, Verantwortlichkeiten und Freiheitsgraden. Die Wahl der richtigen Plattform ist deshalb keine rein technische Entscheidung, sondern hängt von der Erfahrung im Team und den Anforderungen an Skalierung, Flexibilität, Kosten und Betriebsaufwand ab.

Für den schnellen Einstieg und minimalen Verwaltungsaufwand ist App Service mit Containern die naheliegende Wahl. Der Dienst eignet sich hervorragend für klassische Web-Apps und APIs, die bisher Binary Deployments genutzt haben und nun von den Vorteilen der Container profitieren sollen. Wer hingegen dynamische Workloads betreibt, variable Lasten bewältigen muss oder ereignisgetriebene Szenarien umsetzt, findet in Azure Container Apps die passende Lösung. Dieser Dienst kombiniert die Einfachheit von App Service mit der Flexibilität von Kubernetes, bietet Zero-Scaling für Kostenoptimierung und ermöglicht Canary-Deployments sowie Dapr-Integration für Cloud-Abstraktion. Für Teams, die maximale Kontrolle benötigen, führt kein Weg an AKS vorbei. Dies gilt besonders bei komplexen Microservice-Landschaften, optimaler Nutzung der Ressourcen, hybriden Szenarien oder besonderen Anforderungen an Netzwerk und Compliance. Hier stehen alle Kubernetes-Funktionen zur Verfügung, von Ingress-Controllern über Policies bis hin zu GitOps-Ansätzen. Der Preis dafür ist eine höhere Komplexität und ein größerer Betriebsaufwand.

Die Entscheidung ist selten trivial. Faktoren wie Teamgröße, Betriebsreife, Budget und regulatorische Anforderungen spielen eine Rolle. Eine einfache Faustregel lautet: Beginne dort, wo die Einstiegshürde am geringsten ist, und skaliere die Plattformwahl mit deinen Anforderungen. Wer heute eine monolithische App betreibt, startet mit App Service. Wer morgen flexible Skalierung und Event-Integration benötigt, evaluiert Container Apps. Und wer langfristig eine komplexe, hochgradig vernetzte Architektur plant, bereitet sich auf AKS vor. Wichtig ist, die Entscheidung nicht als endgültig zu betrachten, sondern als Teil einer Roadmap, die sich an den Bedürfnissen des Unternehmens orientiert.

Ein entscheidender Vorteil containerisierter Anwendungen ist die Flexibilität beim Plattformwechsel. Wer heute mit App Service startet, kann morgen zu Container Apps wechseln und später zu AKS migrieren, ohne die Anwendung selbst grundlegend verändern zu müssen. Das Container-Image bleibt plattformunabhängig. Eine Entscheidung für eine bestimmte Azure-Plattform ist damit nicht endgültig, sondern kann sich mit wachsenden Anforderungen weiterentwickeln. Teams, die von Anfang an auf saubere Containerisierung und automatisierte CI/CD-Pipelines setzen, schaffen die Grundlage für einen reibungslosen Wechsel zwischen den Plattformen. Das Image wird einmal gebaut und kann überall deployt werden. Nur die Deployment-Konfiguration ändert sich.

Der Einstieg beginnt mit den Grundlagen. Erstelle zuerst ein produktionsreifes Container-Image. Etabliere eine zentrale Registry wie ACR mit klarer Tagging-Strategie. Baue eine erste CI/CD-Pipeline für automatisierte Builds und Deployments auf. Danach folgt die Pilotphase auf der Plattform, die dem aktuellen Szenario am nächsten kommt. Richte parallel dazu Monitoring und Kostenkontrolle ein, um frühzeitig Transparenz über Skalierungsverhalten und Betriebskosten zu gewinnen.

Container in Azure bieten echten Mehrwert, wenn die Grundlagen stimmen und die Plattform zur Aufgabe passt. Wer diese Prinzipien konsequent umsetzt, baut eine Architektur, die heute funktioniert und morgen skaliert. Der Weg von der ersten containerisierten Anwendung zur ausgereiften Container-Plattform ist kein Sprint, sondern ein iterativer Prozess. Starte klein, lerne aus der Praxis und wachse mit den Anforderungen. ■

- [1] Microsoft Azure, What is a container?, <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-container>
- [2] Dockerfile reference, <https://docs.docker.com/reference/dockerfile/>
- [3] Microsoft Dev Blogs, Announcing .NET Chiseled Containers, <https://devblogs.microsoft.com/dotnet/announcing-dotnet-chiseled-containers/>
- [4] Microsoft Learn, Azure Container Registry documentation, <https://learn.microsoft.com/en-us/azure/container-registry/>
- [5] Microsoft Learn, Recommendations for tagging and versioning container images, <https://learn.microsoft.com/en-us/azure/container-registry/container-registry-image-tag-version>

- [6] Microsoft Learn, Non-Microsoft Entra token-based repository permissions in Azure Container Registry, <https://learn.microsoft.com/en-us/azure/container-registry/container-registry-token-based-repository-permissions>
- [7] Microsoft Learn, Azure ABAC repository permissions in Azure Container Registry, <https://learn.microsoft.com/en-us/azure/container-registry/container-registry-rbac-abac-repository-permissions>
- [8] Microsoft Learn, Introduction to Microsoft Defender for container registries (deprecated), <https://learn.microsoft.com/en-us/azure/defender-for-cloud/defender-for-container-registries-introduction>
- [9] Microsoft Learn, Automate container image builds and maintenance with Azure Container Registry tasks, <https://learn.microsoft.com/en-us/azure/container-registry/container-registry-tasks-overview>
- [10] Microsoft Learn, Quickstart: Run a custom container in Azure, <https://learn.microsoft.com/en-us/azure/app-service/quickstart-custom-container>
- [11] Microsoft Learn, Azure Container Apps overview, <https://learn.microsoft.com/en-us/azure/container-apps/overview>
- [12] Microsoft Learn, Azure Container Apps environments, <https://learn.microsoft.com/en-us/azure/container-apps/environment>
- [13] Microsoft Learn, Update and deploy changes in Azure Container Apps, <https://learn.microsoft.com/en-us/azure/container-apps/revisions>
- [14] Microsoft Learn, Set scaling rules in Azure Container Apps, <https://learn.microsoft.com/en-us/azure/container-apps/scale-app>
- [15] Microsoft Learn, What is Azure Kubernetes Service (AKS)?, <https://learn.microsoft.com/en-us/azure/aks/what-is-aks>
- [16] Helm Charts, <https://helm.sh/docs/topics/charts/>
- [17] Kustomize, <https://kustomize.io/>
- [18] Microsoft Learn, Compute services on Azure and AWS, <https://learn.microsoft.com/en-us/azure/architecture/aws-professional/compute>
- [19] Argo CD, <https://argo-cd.readthedocs.io/>
- [20] Flux, <https://fluxcd.io>
- [21] Scaling options for applications in Azure Kubernetes Service (AKS), Microsoft Learn, <https://learn.microsoft.com/en-us/azure/aks/concepts-scale>
- [22] Microsoft Learn, Ingress in Azure Kubernetes Service (AKS), <https://learn.microsoft.com/en-us/azure/aks/concepts-network-ingress>



Florian Bader

ist Solution Architect bei Lunarix Digital Solutions und Microsoft MVP für Azure IoT. Mit langjähriger Erfahrung im Bereich DevOps, Cloud und Digitalisierung unterstützt er Unternehmen bei ihren digitalen Projekten.
florian.bader@lunarix.digital