# Auto healing

Auto-healing for applications is a concept where the system automatically detects and recovers from failures to ensure high availability and reliability.

In the context of Kubernetes, auto-healing is primarily achieved through features like health checks (probes), pod replication, and self-healing mechanisms.

### Key Components of Auto-Healing in Kubernetes

***Health Checks (Probes):***

* **Liveness Probes:**

**Purpose:** To determine if a container is still running. If a liveness probe fails, Kubernetes will restart the container.

**Use Case:** This helps when a container might be running but stuck in an unrecoverable state. Restarting the container can often resolve such issues.

* **Readiness Probes:**

**Purpose:** To determine if a container is ready to serve requests. If a readiness probe fails, Kubernetes will stop sending traffic to the container.

**Use Case:** This is useful when a container needs some initialization before it can start accepting traffic or if it temporarily cannot handle traffic (e.g., during a heavy load or when dependencies are unavailable).

* **Startup Probes:**

**Purpose:** To determine if an application within a container has started. If a startup probe fails, Kubernetes will restart the container.

**Use Case:** This is particularly useful for applications that have a long startup time. The startup probe can give the application more time to initialize without interfering with liveness or readiness checks.2.

**Benefits of Using Probes**

* **Self-Healing**
* **Efficient Load Balancing**
* **Graceful Startup and Shutdown**
* **Monitoring and Observability**

# ACA

What ?

Serverless container service that supports microservice applications and robust autoscaling capabilities without the overhead of managing complex infrastructure.

Azure Container Apps enables you to run microservices and containerized applications on a serverless platform that runs on top of Azure Kubernetes Service.

Common Uses.

* Deploying API endpoints
* Hosting background processing applications
* Handling event-driven processing
* Running microservices

With Azure Container Apps, you can:

* Use the Azure CLI extension, Azure portal or Azure Resource Manager templates to manage your applications.
* Enable HTTPS or TCP ingress without having to manage other Azure infrastructure.
* Build microservices with Dapr and access its rich set of APIs.
* Run jobs on-demand, on a schedule, or based on events.
* Add Azure Functions and Azure Spring Apps to your Azure Container Apps environment.
* Use specialized hardware for access to increased compute resources.
* Run multiple container revisions and manage the container app's application lifecycle.
* Autoscale your apps based on any KEDA-supported scale trigger. Most applications can scale to zero.
* Split traffic across multiple versions of an application for Blue/Green deployments and A/B testing scenarios.
* Use internal ingress and service discovery for secure internal-only endpoints with built-in DNS-based service discovery.
* Run containers from any registry, public or private, including Docker Hub and Azure Container Registry (ACR).
* Provide an existing virtual network when creating an environment for your container apps.
* Securely manage secrets directly in your application.
* Monitor logs using Azure Log Analytics.
* Override (already generous) quotas to increase limits on a per-account basis.

We can

* Any Linux-based x86-64 (linux/amd64) container image with no required base image
* Containers from any public or private container registry
* Sidecar and init containers

Multiple Containers

You can define multiple containers in a single container app to implement the [sidecar pattern](https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar).

Examples of sidecar containers include:

* An agent that reads logs from the primary app container on a shared volume and forwards them to a logging service.
* A background process that refreshes a cache used by the primary app container in a shared volume.

## Limitations

Azure Container Apps has the following limitations:

* **Privileged containers**: Azure Container Apps can't run privileged containers. If your program attempts to run a process that requires root access, the application inside the container experiences a runtime error.
* **Operating system**: Linux-based (linux/amd64) container images are required.

## Revisions

Revisions are a way to deploy multiple versions of an app where you have the option to send the traffic to a certain revision.

Can have up to 100 revisions.

2 types

Single revision mode

Multiple Mode - good for A/B testing . Canary deployment

## Pod

Composed of the application container or any required side car container

# Microservices – Challenges

But more often than not we face the same challenges:

* Recovering state after failures
* Services discovery and calling other microservices.
* Integration with external resources
* Asynchronous communications between different services
* Distributed tracing
* Measuring message calls and performance across components and networked services

# DAPR

Distributed application Runtime.

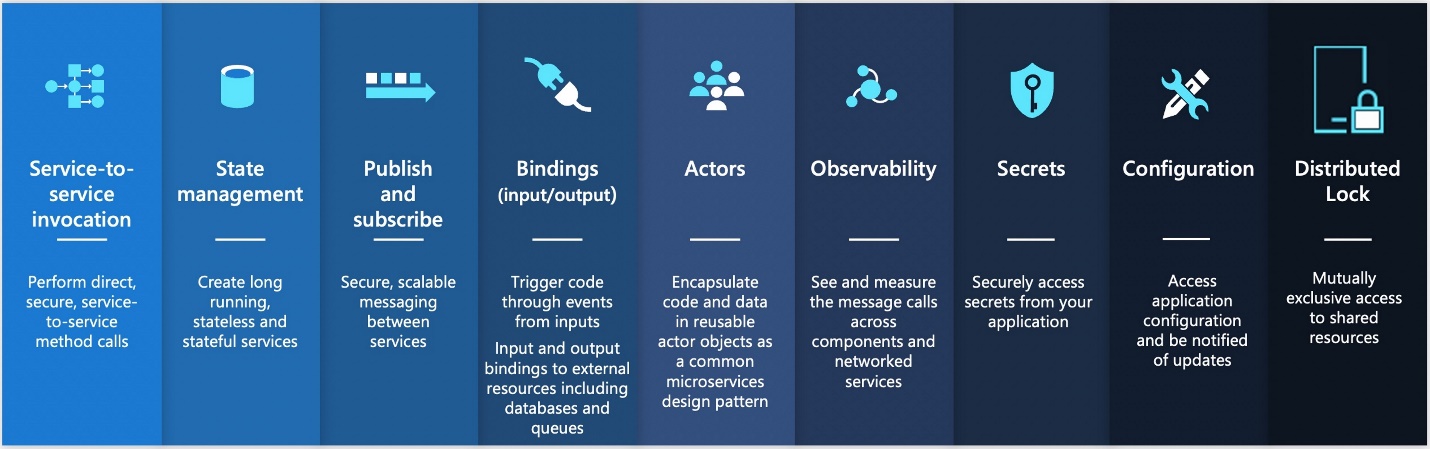
Offers a solution for the common challenges that are faced in any distributed microservice application.

Can be used with any language (Go, .NET python, Node, Java, C++) and run anywhere.

(On-premise, Kubernetes, and any public cloud (e.g. Azure)

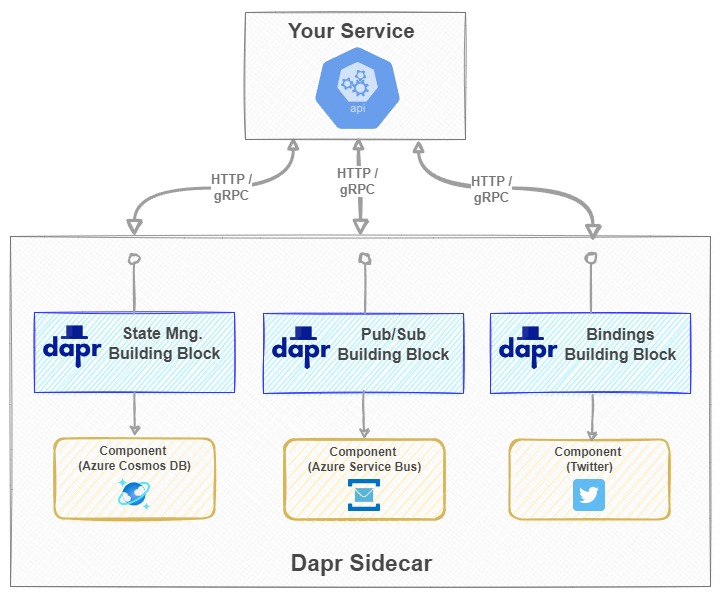
Dapr’s core component is its building blocks . So far it supports 9 building blocks

Simply put, a Building Block is a modular component which encapsulates best practices and can be accessed over standard HTTP or gRPC APIs.



## Dapr & Microservices

Dapr exposes its Building Blocks and components through a **sidecar architecture**. A sidecar enables Dapr to run in a separate memory process or separate container alongside your service. Sidecars provide isolation and encapsulation as they aren't part of the service but connected to it.



While Azure Container Apps features the building blocks for running microservices, using the Distributed Application Runtime (Dapr) provides an even richer microservices programming model.

<https://github.com/dotnet-architecture/eBooks/blob/1ed30275281b9060964fcb2a4c363fe7797fe3f3/current/dapr-for-net-developers/Dapr-for-NET-Developers.pdf>

Because the calls will flow through container sidecars, Dapr can inject some useful cross-cutting behaviors that are meaningfully abstracted from our application containers.

Some features include

* Automatically retry calls upon failure.
* Make calls between services secured with mutual authentication (mTLS), including automatic certificate rollover.
* Control what operations clients can perform using access control policies.
* Capture traces and metrics for all calls between services to provide insights and diagnostics.

# Demo

## Application Overview

1. Frontend  web app that accepts requests from public users to manage their tasks. It invokes the component " -Backend" endpoints via HTTP or gRPC.
2. Backend is a backend Web API which contains the business logic of tasks management service, data storage, and publishing messages to Azure Service Bus Topic.
3. Processor-Backend is an event-driven backend processor which is responsible for sending emails to task owners based on messages coming from Azure Service Bus Topic. Here there is a continuously running background processor, which is based on Dapr Cron timer configuration, to flag overdue tasks.