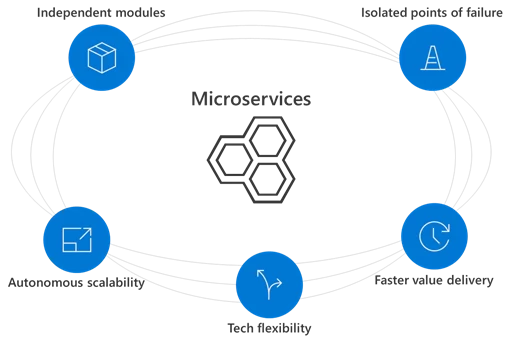
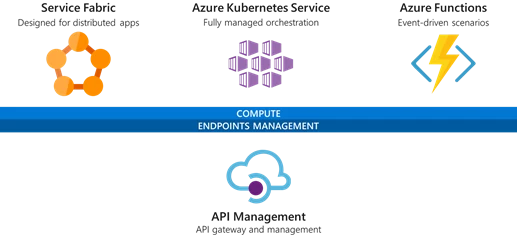
**Building serverless microservices in Azure – sample architecture**

Distributed applications in the cloud leverage their global presence, evade bottlenecks, and ensure constant accessibility for users worldwide. Achieving these capabilities necessitates suitable infrastructure deployment and support for a decoupled architecture, a departure from the traditional monolithic approach. As a result, the majority of cloud-native applications adopt a microservices architecture to attain global scalability. By breaking down the application into smaller, loosely connected services, it becomes easier to distribute tasks, scale specific components independently, and enhance overall resilience and availability for users across the globe.



When building applications in the cloud, utilizing a microservices architecture yields significant advantages, particularly when complemented by a diverse array of managed services. These services facilitate the realization of the microservices' potential by handling infrastructure management, scaling, and optimizing crucial processes like deployment and monitoring. As a result, developers can focus more on delivering value through their applications, as the burden of infrastructure management is alleviated. This approach maximizes the efficiency and productivity of development cycles, enabling organizations to deliver more value to their users in less time.



* **Building serverless, event-driven microservices:**

By using an event-driven approach for constructing microservices-based applications, we can effectively tackle various issues that are commonly encountered in a more conventional development approach.

1. **Scaling compute resources**: In traditional deployments, scaling applications to meet varying demands can be a complex and time-consuming task. However, with serverless platforms, this process becomes automatic and flexible. The serverless architecture automatically handles scaling based on the actual demand your application receives. This means that as the number of users or requests increases, the platform will automatically allocate the necessary resources to handle the load. Likewise, during periods of low activity, it will scale down to save costs. As a developer, you don't need to worry about manually configuring or managing the scaling process in your code. This not only simplifies development but also ensures that your application can handle fluctuations in traffic seamlessly.

2. **Operations dependency**: When deploying a microservices-based solution traditionally, the operations team is often responsible for provisioning and managing infrastructure resources. This can create dependencies and bottlenecks, as any changes to the application, such as updates or additional services, require coordination with the operations team. However, adopting a serverless approach eliminates this dependency. Serverless platforms, like AWS Lambda or Azure Functions, manage all the underlying infrastructure for you. This means you can deploy and run your microservices without worrying about provisioning servers or managing virtual machines. The platform takes care of resource allocation and optimization, enabling developers to focus more on building and deploying their applications.

3. **Costs for hosting**: In conventional hosting models, you typically pay for each hosting node, which can lead to overallocation of resources. This means you might end up paying for idle resources during periods of low demand. With serverless architectures, you only pay for the actual resources used during the execution of functions or handling requests. The pricing is consumption-based, meaning you're charged based on the number of requests or operations your application performs, and the time it takes to execute them. As a result, serverless architectures often offer a more cost-effective solution, especially for applications with variable workloads. The pricing is directly aligned with the real usage of the solution, leading to potential cost savings.

1. **Services discovery**: In distributed applications using microservices, service integration and communication are critical aspects. Each service has a specific responsibility, and often, one service needs to communicate with others to achieve its goals. Managing these connections can be challenging, as services should remain loosely coupled to ensure flexibility and maintainability. With an event-driven approach, services can interact with each other using events. Events act as messages that trigger specific actions in other services. This approach simplifies services discovery, as services only need to listen for relevant events and act accordingly. The decoupled nature of event-driven architectures allows you to modify or add services without disrupting the overall system, making it easier to maintain and scale the application.

With an event-driven approach, you can take advantage of both of the following:

* **Centralized Communication via Pub-Sub Model with Azure Event Grid:**

Imagine you have multiple services in your application that need to communicate with each other when specific events occur. Instead of setting up complex direct connections between these services, a centralized approach called a "Pub-Sub model" is used. In this model, services communicate by publishing events to a central hub (Pub) and subscribing (Sub) to events they are interested in.

Azure Event Grid is a fully managed service provided by Microsoft Azure that facilitates this centralized communication. It acts as the hub for events, allowing services to publish events when something important happens, and other services can subscribe to those events to receive the information they need. This decouples the services from one another, making the system more flexible and scalable.

For example, let's say you have a payment service, a notification service, and an order processing service in your application. When a successful payment occurs, the payment service can publish an "OrderPaid" event to Azure Event Grid. The notification service and order processing service can then subscribe to the "OrderPaid" event. When the event is published, both services will receive it, allowing the notification service to send an email to the customer and the order processing service to update the order status.

* **Integrated Programming Model with Azure Functions and Logic Apps:**

To automatically respond to the events published through Azure Event Grid and seamlessly integrate different services, Azure offers an integrated programming model. This model is based on "triggers" and "bindings," which simplify the development process.

- Triggers: Triggers are event-based hooks that start the execution of a function or logic app when a specific event occurs. For example, when the "OrderPaid" event is published to Azure Event Grid, it can trigger an Azure Function or Logic App to run automatically.

- Bindings: Bindings are connections that link the input and output of your Azure Function or Logic App with other Azure services or external resources. They allow you to easily access and process data from different sources and interact with other services without writing complex code.

For instance, using Azure Functions, you can create a function that gets triggered by the "OrderPaid" event. With bindings, you can connect the function to your database to store the payment information, and at the same time, send a message to a message queue for further processing. This integrated approach significantly simplifies the development and integration of services, as you don't need to manage the event handling or service connections manually. Azure Functions and Logic Apps take care of the plumbing, allowing you to focus on building the business logic of your application.

* **Sample architecture for serverless microservices:**

In the [sample architecture](https://aka.ms/serverless-microservices) for a rideshare application for a fictitious company named Relecloud, you can learn more about the architectural design of a microservices-based application. The sample uses fully managed services from the Azure Serverless platform to build the main building blocks of microservices solutions such as:

**1. API Gateway:**

Think of the API Gateway as a central entry point for your application. It uses API Management to expose the endpoints (or URLs) of the backend services securely. This means the client application (e.g., a website or mobile app) can interact with the backend services through the API Gateway without directly connecting to each individual backend service.

The API Gateway provides a layer of abstraction, which decouples the client application from the backend services. It allows you to manage changes on where the services are hosted without affecting the client application. For example, if you need to move a service to a different server or provider, you can do so behind the API Gateway without the client application being aware of the change.

**2. Entry Points:**

Entry points are the public facing APIs that your client application uses to communicate with the backend services. These APIs are powered by Azure Functions, which respond to HTTP requests. Azure Functions are like small, individual pieces of code that get triggered when a specific request comes in. They allow you to handle different types of requests and perform specific actions accordingly.

**3. Workflow Orchestrator:**

The Workflow Orchestrator is like the middle-tier service that sits between the public facing APIs and the actual backend services. It's responsible for coordinating and interconnecting these services based on actions initiated by the client application. This orchestrator acts as a conductor, directing the flow of tasks and data between various backend services, making sure they work together smoothly to fulfill the client's request.

**4. Async Queue:**

The Async Queue is a messaging service that facilitates communication between the different services in your application. In this case, it's represented by Azure Event Grid. An event-driven approach is used, where services exchange information using events. When one service completes a task or produces important information, it pushes an event to the Async Queue. Other services subscribe to these events, and when an event is received, it triggers a handler to process the information. This "fire-and-forget" approach allows services to communicate without being tightly coupled, ensuring a more flexible and scalable architecture.

"Fire and forget" is an asynchronous communication pattern where a sender sends a message to a receiver without waiting for a response. It's used when loose coupling is needed, and the sender doesn't require immediate feedback. Common in event-driven architectures. Caution needed for critical actions; consider request-response pattern for reliability.

* The emitter fires and forgets. No need to wait until a response arrives.
* Events can be delivered to multiple listeners that can process the event data.
* Events have data and meta data such as subject that can be used to determine processing. For example, the `Power BI Trip Processor filters out events based on subject.

**5. Backend Services:**

Backend services are the heart of your application. They directly interact with the data layer and other critical components of the solution. These services are isolated from the rest of the application, meaning they operate independently. If you need to make changes, like switching to a different type of database, you can do so without affecting the overall application or its interactions. This modularity and isolation make the application more maintainable and adaptable.

In summary, this architecture separates concerns, providing a flexible and scalable system. The API Gateway exposes the backend services to clients, while the Workflow Orchestrator coordinates tasks between backend services. The Async Queue enables asynchronous communication, and the Backend Services operate independently with the data and other components. Together, this architecture promotes decoupling, scalability, and ease of maintenance.