AZURE FUNCTION THEORETICAL Q&A

BY - SHUBHAM WADEKAR

Copyright © Shubham Wadekar. All rights reserved.

This material is for personal use only. No part may be copied, shared, resold, or published without written permission. Unauthorized distribution is strictly prohibited and may result in action.

For permissions or licensing, contact: [shubham.p.wadekar@gmail.com]

Disclaimer: This content is for educational purposes. Accuracy is attempted but not guaranteed. No job outcome is promised.

© Shubham Wa

1. What are Azure Functions, and what are their primary use cases?

Azure Functions is a serverless compute service that allows us to run small pieces of code in the cloud without setting up or managing any infrastructure. It helps us write event-driven code that runs only when it is needed. Azure automatically takes care of scaling, patching, and running the function on demand.

Some common use cases of Azure Functions are:

- Running background jobs like sending emails or notifications
- Processing data when files are uploaded to storage
- Creating serverless APIs that respond to HTTP requests
- Running code on a timer, like every 5 minutes or once a day
- Integrating with other Azure services like Event Hub or Service Bus
- Performing lightweight data transformation or logging

I find Azure Functions very helpful in scenarios where we want to respond quickly to events, automate repetitive tasks, or build a scalable system without worrying about the underlying infrastructure.

2. Explain the different types of triggers supported by Azure Functions.

Azure Functions are event-driven, which means they start when a specific trigger happens. Each function must have one trigger. There are many types of triggers available in Azure Functions, and each one is tied to a different event source.

Some of the main trigger types include:

- HTTP trigger: This runs the function when it receives an HTTP request. It is often used to build APIs or webhooks.
- Timer trigger: This runs the function on a schedule, like a cron job. It is useful for scheduled tasks like cleaning up data or sending daily reports.
- Blob storage trigger: This runs the function when a file is added or updated in Azure Blob Storage. It is useful for processing images, documents, or logs.
- Queue storage trigger: This runs the function when a new message is added to an Azure Storage Queue. It is used for background job processing.
- Event Hub trigger: This runs the function when it receives data from an Azure Event Hub, which is useful in event streaming scenarios like telemetry or IoT data.
- Service Bus trigger: This runs the function when it receives a message from a Service Bus queue or topic. It is used in enterprise messaging systems.
- Cosmos DB trigger: This runs the function when data changes in a Cosmos DB collection. It is useful for real-time processing of changes.

Each trigger is designed to handle specific scenarios, and we can choose the right one based on the event we want to respond to. The trigger also defines how the function is invoked and what data it receives.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

3. What are triggers and bindings in Azure Functions?

In Azure Functions, triggers and bindings help connect the function to other services or systems without writing extra code for integration. A trigger is what starts the function. Every function must have one trigger. For example, the function can run when it receives an HTTP request, a message is added to a queue, or a file is uploaded to blob storage.

Bindings, on the other hand, make it easy to read from or write to other services. They act like connectors. A binding can be input, output, or both. Input bindings provide data to the function, like reading a document from Cosmos DB. Output bindings let the function send data somewhere, like writing to a queue or a database. The best part is, we can use these without writing full connection code. Azure Functions handles that for us behind the scenes.

4. Explain the concept of bindings in Azure Functions and provide an example.

Bindings in Azure Functions are a way to connect to external services like storage, databases, or messaging systems. They make it easy to work with these services without writing detailed connection or authentication code. Azure automatically handles the interaction using configuration in the function code and in the function.json file.

There are two types of bindings: input bindings and output bindings. Input bindings bring data into the function, and output bindings send data out to another service.

For example, if I have a function that gets triggered by an HTTP request and I want to store the data in an Azure Table Storage, I can use an output binding to the table. I just pass the data to a specific output parameter, and Azure will automatically write it to the table storage.

This makes the function clean and simple because I don't have to manually write code to connect to Table Storage or handle authentication.

5. What is the difference between input binding and output binding in Azure Functions?

The main difference between input and output binding is the direction of data flow.

Input binding is used to bring data into the function. It allows the function to read data from an external source before the function runs. For example, the function can read a document from Cosmos DB or a blob from Azure Storage using input binding.

Output binding is used to send data from the function to an external service after the function runs. For example, the function can write a message to a Service Bus queue or store a file in blob storage using output binding.

Input binding provides data for processing. Output binding sends the result or processed data to another system. Both types of bindings help reduce boilerplate code and make it easier to connect Azure Functions with other services.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

6. How can you use bindings to connect Azure Functions with external systems or data sources?

Bindings make it very easy to connect Azure Functions to external systems like Azure Storage, Cosmos DB, Service Bus, or even external APIs. Instead of writing full connection logic in the function code, we define the bindings in the function's configuration, and Azure handles the rest.

To use bindings, we just need to declare them in the function code and also configure them in the function.json file. For example, if I want to read a file from Azure Blob Storage and process it in my function, I can use a blob input binding. Similarly, if I want to write a message to a queue after processing, I can use a queue output binding.

This way, bindings help the function receive input data and send output data to other services without manually writing code for connection, authentication, or serialization. It reduces the complexity and makes the code cleaner and more focused on business logic.

7. What is the purpose of the function.json file in an Azure Function?

The function.json file is a configuration file that describes the bindings and trigger for an Azure Function. It tells Azure how the function should be triggered and what inputs or outputs it will work with.

This file includes information like the type of trigger (HTTP, Timer, Blob, etc.), the direction of the binding (input or output), the name of the variable that will hold the data, and the details of the connection (like storage account name or path).

Azure reads this file to understand how to connect the function to other services. It is especially useful when the function is deployed, because Azure uses it to wire up the required bindings correctly.

In short, function.json acts like a bridge between the Azure platform and the function code, allowing serverless integration with other services.

8. What are the benefits of serverless computing in Azure?

Serverless computing in Azure, like Azure Functions, gives several benefits:

1. **No server management** – I don't have to worry about provisioning or maintaining servers. Azure automatically handles the infrastructure.
2. **Automatic scaling** – Azure Functions automatically scale up or down depending on the number of events or requests. If there are many requests, more instances are created. If there are none, it scales to zero.
3. **Cost efficiency** – In the Consumption Plan, I only pay for the time my function runs. If the function is idle, there is no cost.
4. **Faster development** – Since I focus only on writing code and not infrastructure, development is quicker. I can just write the logic, define triggers and bindings, and deploy.
5. **Easy integration** – It's easy to integrate with other Azure services like Blob Storage, Cosmos DB, Service Bus, and others using triggers and bindings.
6. **Event-driven architecture** – Serverless is great for responding to events, such as file uploads, database changes, or message arrivals.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

Overall, serverless in Azure allows me to build applications faster, with less operational overhead, and lower costs.

9. How does Azure Functions handle automatic scaling based on workload?

Azure Functions can scale automatically depending on the number of incoming events or requests. This means it can add more function instances when demand increases and reduce them when demand goes down. This scaling is handled by Azure without any manual setup.

For example, if I have a function that processes messages from a queue and suddenly the queue gets a thousand messages, Azure will automatically start multiple instances of the function in parallel to process those messages faster. Once the workload reduces, Azure will shut down the extra instances to save cost.

In the Consumption Plan, scaling happens dynamically and I don't need to configure anything. In the Premium and Dedicated plans, I have more control, such as setting minimum and maximum instances or using pre-warmed instances to avoid cold starts.

10. How does Azure Functions support parallel execution of multiple requests?

Azure Functions can run multiple instances at the same time to handle many requests or events in parallel. This parallelism is managed by Azure, based on the trigger type and the hosting plan.

For example, if many HTTP requests arrive at the same time or many messages are added to a queue, Azure creates multiple instances of the function and runs them independently to process each request or message. Each instance runs in isolation, so they don't interfere with each other.

In the Consumption Plan, Azure will keep increasing instances as needed, depending on the workload. In the Premium and Dedicated plans, the number of parallel executions depends on the number of workers and configuration settings.

Triggers like Event Hub and Queue Storage have built-in support for batch processing and partitioning, which further improves parallel processing. So, Azure Functions is well-suited for handling high-volume, parallel workloads efficiently.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

11. What are the different hosting plans available for Azure Functions, and how do they differ?

Azure Functions offers three main hosting plans, and each one is designed for different scenarios:

1. **Consumption Plan**
    - This is the default and most cost-effective plan.
    - I only pay when the function is running.
    - Azure automatically handles scaling.
    - It may have cold start delays when the function is idle for some time.
    - Suitable for small to medium workloads or occasional jobs.
2. **Premium Plan**
    - It includes all the features of the Consumption Plan, but with extra benefits.
    - Pre-warmed instances help avoid cold starts.
    - It supports longer execution time and higher performance.
    - Good for critical applications that need faster response and more control.
    - I pay based on the number of instances and memory used, even when idle.
3. **Dedicated (App Service) Plan**
    - The function runs on a regular App Service plan.
    - I can run other web apps and APIs on the same plan.
    - Good if I already have an App Service environment.
    - I have full control over scaling, but I pay for the reserved infrastructure whether the function runs or not.

Choosing the right plan depends on workload type, cost needs, performance, and whether I need advanced features like VNET support or long-running functions.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

12. What is the difference between the Consumption Plan and the Premium Plan in Azure Functions?

The main difference between the Consumption Plan and the Premium Plan in Azure Functions is how they handle scaling, performance, and pricing.

In the **Consumption Plan**, Azure creates function instances only when they are needed. I pay only for the time my function runs and the number of executions. This is cost-effective for light or unpredictable workloads. However, the first request after the function has been idle for a while might take longer to respond because of a cold start. Also, there are some limits like maximum execution time (usually 5 minutes by default, extendable to 10).

In the **Premium Plan**, Azure keeps pre-warmed instances ready at all times, so there is no cold start. It also supports longer execution times, more memory, and features like virtual network integration. I pay for the number of pre-warmed instances even if no requests come in, which makes it more suitable for applications that need high performance or constant availability.

So, I would choose the Consumption Plan for cost savings and simple tasks, and the Premium Plan for performance-critical or long-running applications.

13. What are Durable Functions, and how do they extend the capabilities of regular Azure Functions?

Durable Functions are an extension of Azure Functions that let me build workflows or long-running processes in a serverless environment. With regular Azure Functions, each function execution is short-lived and stateless. But Durable Functions can keep track of state and handle complex steps that happen over time.

They use something called an orchestrator function, which controls the flow and manages the state of other functions (called activity functions). Azure automatically saves the progress and state in storage, so I don't have to write any custom logic to manage it.

For example, if I want to send a welcome email, wait for the user to confirm their email address, then add them to a mailing list, I can do this using Durable Functions. The function can pause and resume without losing any data, even if hours or days pass.

This makes Durable Functions ideal for workflows, approvals, retries, and any scenario where I need to wait or coordinate multiple steps.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

14. What is the Function Chaining pattern in Durable Functions, and when would you use it?

The Function Chaining pattern in Durable Functions is when multiple functions need to run one after another in a specific order, and the output of one function is passed as input to the next.

For example, imagine I have three tasks:

1. Get a user's profile from the database
2. Generate a report based on that profile
3. Email the report to the user

Using function chaining, I can run all three tasks in order inside the orchestrator function. Each step waits for the previous one to finish and uses its result.

I would use function chaining when I have a strict sequence of steps that depend on each other. It helps keep the flow organized and makes sure each step only runs after the previous one is successful. Durable Functions automatically track the progress, so if the process is interrupted, it can resume from where it left off.

15. How do Azure Durable Functions handle state and long-running processes?

Azure Durable Functions manage state by using something called an orchestrator function. This orchestrator keeps track of the entire process flow and its state using Azure Storage behind the scenes. It automatically saves checkpoints after each step so that if the process is paused, failed, or restarted, it can continue from the last successful point without losing data.

For long-running processes, the orchestrator function doesn't keep running in memory. Instead, it pauses and saves its progress after calling each activity function. When the next step is ready, it wakes up, reads the state, and continues. This makes it possible to handle processes that last for hours, days, or even longer without consuming compute resources the whole time.

For example, if I'm waiting for user approval or data from an external system, the function can pause until the response is received. During that wait time, it costs nothing and uses no CPU. This is how Azure Durable Functions support long-running and stateful workflows in a very efficient way.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

16. How can Azure Functions be integrated with other Azure services like Event Hub or Cosmos DB?

Azure Functions can be easily integrated with other Azure services by using triggers and bindings. These help connect the function to services like Event Hub or Cosmos DB without writing full connection code.

For example:

- I can use an Event Hub trigger to start the function whenever a message is published to an Event Hub. This is useful in real-time data streaming or telemetry processing.
- I can use a Cosmos DB trigger to run the function whenever a new document is added or updated in a Cosmos DB collection. This helps in processing or syncing data changes.
- If I want to read from Cosmos DB, I can use an input binding to pass the data to my function.
- Similarly, if I want to write data to Cosmos DB, I can use an output binding to save the result.

This way, Azure Functions can both listen to and interact with various services in a clean and simple way. I only need to configure the right trigger or binding, and Azure manages the communication behind the scenes.

17. What are the ways to secure Azure Functions from unauthorized access?

There are several ways to secure Azure Functions and control who can access them:

1. **Function Keys –** For HTTP-triggered functions, Azure automatically creates a default key. Anyone calling the function must include the key in the request. This is useful for simple authentication.
2. **Authorization Levels –** Each HTTP function can be set to one of three levels: anonymous (no key required), function (key required), or admin (master key required). This controls how open the function is.
3. **Azure Active Directory (AAD) –** For enterprise-level security, I can integrate Azure Functions with AAD to require users or applications to authenticate using tokens. This is more secure and supports role-based access.
4. **API Management –** I can place Azure API Management in front of my functions to add security policies like rate limiting, IP restrictions, and JWT validation.
5. **VNET Integration –** In Premium or Dedicated plans, I can run the function inside a virtual network, so it is not exposed to the public internet.
6. **Access Restrictions –** I can configure IP restrictions to allow only certain IP ranges to call the function.

By using these methods, I can make sure my Azure Functions are protected from unauthorized access and meet security requirements.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

18. How can you secure an HTTP-triggered Azure Function using authentication and authorization?

To secure an HTTP-triggered Azure Function, I can use multiple layers of authentication and authorization. The most common and recommended way is to use Azure Active Directory (AAD) or built-in authentication provided by the Azure App Service platform.

Here are the main steps I usually follow:

1. **Set the authorization level –** When creating an HTTP trigger, I can define the access level as anonymous, function, or admin. This determines if a function key is needed to access the function.
2. **Enable App Service Authentication –** In the Azure portal, I can turn on authentication for the Function App. Azure supports identity providers like AAD, Microsoft accounts, Google, Facebook, etc.
3. **Use Azure Active Directory –** If I choose AAD, only users or applications with proper roles or tokens can access the function. I can define access based on groups or roles using managed identities or service principals.
4. **Validate tokens in the function code –** In some advanced scenarios, I may validate incoming JWT tokens manually inside the function to check claims or permissions.

This setup ensures that only authorized users or systems can call the HTTP endpoint, and it adds enterprise-level security without writing custom authentication logic.

19. What authentication options are available to protect Azure Functions?

Azure Functions can be protected using several authentication options depending on the scenario:

1. **Function Keys –** Each function has a default key that must be included in HTTP requests. This is a simple way to prevent anonymous access.
2. **Admin Keys –** The master key can be used for full access to all functions in the app. It's useful for managing the app or automating tasks securely.
3. **App Service Authentication (Easy Auth) –** Azure provides built-in authentication that integrates with identity providers like:
    - Azure Active Directory
    - Microsoft Account
    - Google
    - Facebook
    - GitHub

This option is configured in the Azure portal and automatically handles sign-in and access token validation.

4. **Azure Active Directory –** For enterprise applications, AAD provides secure access control. I can use AAD for both users and applications (via managed identity or service principal).
5. **API Management –** When using Azure API Management in front of Functions, I can add additional authentication layers like subscription keys, OAuth 2.0, and token validation.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

6. **IP Restrictions and VNET Integration –** These are network-based protections that prevent access from unknown locations or restrict access to internal systems.

By using these options, I can build a secure environment for my Azure Functions based on the sensitivity of the data and the access requirements.

20. What are some best practices for developing and deploying Azure Functions in a production environment?

Here are some best practices I follow when working with Azure Functions in production:

1. **Use application settings for configuration –** I avoid hardcoding values like connection strings and use environment variables in the Azure portal to manage settings securely.
2. **Use durable functions for long workflows –** When I need to manage state or run multi-step workflows, I prefer Durable Functions instead of writing custom logic.
3. **Enable authentication –** I always secure HTTP-triggered functions using function keys or Azure AD to avoid unauthorized access.
4. **Implement retries and error handling –** I use retry policies and logging to handle failures gracefully and identify issues quickly.
5. **Set up monitoring and alerts –** I connect my Function App to Application Insights to track performance, errors, and logs in real time.
6. **Use CI/CD pipelines –** I use tools like Azure DevOps or GitHub Actions to automate deployment and ensure consistent releases.
7. **Organize related functions into one Function App –** This helps with easier management, shared settings, and better cost control.
8. **Avoid long cold starts –** For critical workloads, I prefer Premium Plan to reduce delays and ensure availability.
9. **Test locally before deployment –** I use tools like Azure Functions Core Tools and Visual Studio Code to test my function thoroughly before moving to production.
10. **Use version control and infrastructure as code –** I keep my function code and infrastructure definitions in source control using ARM templates or Bicep to ensure repeatable deployments.

These practices help me ensure that the Azure Functions are secure, reliable, easy to manage, and perform well in a live environment.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

21. How can Azure Functions be deployed using CI/CD pipelines like Azure DevOps or GitHub Actions?

To deploy Azure Functions using CI/CD pipelines, I use either Azure DevOps or GitHub Actions to automate the process of building, testing, and deploying the function code.

In Azure DevOps, I create a pipeline using a YAML file or the classic UI. The pipeline usually has two main stages: build and release. In the build stage, I restore dependencies, build the function app, and publish the output to an artifact folder. In the release stage, I use the Azure Function App Deploy task to deploy that artifact to the target Azure Function App. I also store sensitive information like function keys or connection strings in Azure Key Vault or pipeline secrets.

In GitHub Actions, I create a workflow file in the .github/workflows folder. This workflow includes steps to check out the code, install any required packages, and deploy the code using the Azure Functions GitHub Action. I save Azure credentials in GitHub Secrets and use those secrets in the workflow to authenticate with Azure.

Both methods allow me to deploy code automatically whenever there is a code change. This ensures faster delivery, fewer manual errors, and better control over the deployment process.

22. How do you manage and install external dependencies in Azure Functions?

Managing external dependencies in Azure Functions depends on the programming language used. These dependencies are the libraries or packages that my function needs to work properly.

If I am using C# with the .NET language, I add dependencies through NuGet by editing the .csproj file. For example, if I need to use JSON features, I add the Newtonsoft.Json package. These packages are automatically restored when the project is built.

If I am using JavaScript or Node.js, I install packages using npm. I create a package.json file and run npm install to download all the dependencies. For example, if I need to call an API, I might install the axios package.

For Python, I use a requirements.txt file to list all the dependencies. Then, Azure installs them when the function runs or when I deploy the function.

These dependencies are stored along with the function code, and Azure automatically includes them when running the function. This allows me to use powerful external tools and libraries easily in my Azure Functions.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

23. How do you set up local development for Azure Functions using tools like Visual Studio Code?

To set up local development for Azure Functions using Visual Studio Code, I first install a few tools. I install the Azure Functions extension in Visual Studio Code. I also install the Azure Functions Core Tools and the runtime or SDK for the language I'm using, such as .NET, Node.js, or Python.

Then, I open Visual Studio Code and use the command palette to create a new Azure Functions project. I choose the language, the template like HTTP trigger, and the folder location. The extension generates the required files including host.json and local.settings.json for me.

I write the function code and use local.settings.json to store environment variables like connection strings. I can run the function locally by pressing F5 or using the terminal. The Azure Functions Core Tools will start a local server so I can test the function using a browser or Postman.

Once I am confident that the function works as expected, I can deploy it directly to Azure using the extension. This local setup helps me develop, debug, and test my function fully before deploying it to the cloud.

24. What is the difference between a Function App and an individual Function in Azure?

In Azure, a Function App is like a container or a group that holds one or more individual functions. It acts as the main unit for managing and deploying Azure Functions. All the functions inside a Function App share the same settings, runtime version, and hosting plan.

An individual function is a single piece of code that does one task, such as processing an HTTP request or handling a queue message. Each function has its own trigger and logic, but it runs within the Function App.

For example, I can have a Function App for order processing, and inside it, I can have one function to handle new orders, another to send confirmation emails, and another to update inventory. All of them live inside the same Function App, and they share the same configuration and resources.

So, the Function App is the top-level structure, and individual functions are the specific tasks inside it.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

25. What are the different runtime versions available for Azure Functions, and how do they affect compatibility?

Azure Functions supports different runtime versions to support different features and programming languages. The main runtime versions are version 1, version 2, version 3, and version 4.

- Version 1 supports only the .NET Framework and is mostly used for older applications.
- Version 2 introduced cross-platform support and allowed functions to run on .NET Core, JavaScript, Python, and other languages.
- Version 3 supports .NET Core 3.1 and is more stable and flexible for multi-language support.
- Version 4 supports .NET 6 and .NET 7 and is the latest version. It is designed for long-term support and works well with the latest features and tools.

The runtime version affects what language and framework I can use, and how I write and deploy my function. For example, if I want to use .NET 6 or isolated process models, I need to use runtime version 4. Choosing the right version is important to make sure the function works with the tools and libraries I plan to use.

26. What is the role of the host.json file in Azure Functions configuration?

The host.json file is a global configuration file used to set behavior and settings for all the functions inside a Function App. It controls things like logging, timeout settings, retry policies, and how triggers behave.

For example, I can use host.json to increase the maximum batch size for a queue trigger, or to change the logging level for all functions in the app. The settings in host.json apply to the Function App as a whole, not to individual functions.

This file is written in JSON format and is part of the function project. When I deploy the function app, Azure reads the host.json file to apply those settings during runtime. It helps in customizing the behavior of functions without changing the actual function code.

27. What is the cold start problem in Azure Functions, and how can it be minimized?

The cold start problem happens when an Azure Function is triggered after being idle for some time. Azure needs to allocate resources and load the function app before it can run the function. This delay is called a cold start. It usually takes a few seconds and can affect performance, especially for time-sensitive applications.

Cold starts mostly happen in the Consumption Plan, because Azure deallocates resources when the function is not in use to save cost.

To minimize cold start, I can use the Premium Plan, which keeps at least one instance always warm. This prevents startup delays. Another way is to use an external service or timer trigger to "ping" the function regularly, keeping it active. I also avoid using heavy dependencies in the startup code, which can increase cold start time.

So, while cold start can slow down the first request, using the right plan and keeping the app warm helps reduce or avoid it.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

28. What are the limitations of Azure Functions in terms of execution time, memory, and other resources?

Azure Functions do have some limits, especially in the Consumption Plan:

- **Execution time**: In the Consumption Plan, a function can run for a maximum of 5 minutes by default. This can be increased to 10 minutes using configuration. For longer executions, I use the Premium Plan or Durable Functions.
- **Memory**: The maximum memory available is 1.5 GB per instance in the Consumption Plan. In the Premium Plan, I can choose plans with higher memory, like 3.5 GB or more.
- **Storage**: Temporary storage (D:\ drive) is limited to about 500 MB. For large data, I use Azure Blob Storage or other services.
- **Concurrency**: Azure decides how many function instances to run based on workload. In the Consumption Plan, this is controlled by internal limits. For more control over scaling, I use the Premium or Dedicated Plan.
- **Startup time**: Cold start can add a few seconds of delay when functions are not frequently used.
- **No control over infrastructure**: I cannot control the virtual machines or containers running the functions in the Consumption Plan.

These limitations are usually acceptable for short tasks, but for high-performance needs, I use Premium Plan or adjust the design accordingly.

29. How is Azure Functions different from Azure App Services in terms of usage and billing?

Azure Functions and Azure App Services are both used to run code in the cloud, but they work differently.

Azure Functions is a serverless platform. It automatically runs code in response to events, like HTTP requests, queue messages, or timers. I don't need to manage infrastructure or servers. In the Consumption Plan, billing is based on the number of executions and the time the function runs. It is very cost-effective for low or irregular workloads.

Azure App Services, on the other hand, is used to host web applications, APIs, and background jobs. I need to choose and pay for a specific server plan, even if the app is idle. It gives me more control over the environment, such as custom domains, SSL, and advanced networking.

So, I use Azure Functions for event-driven, short-running tasks, and I use App Services for full web apps or APIs that need consistent availability and control.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar

30. What are some real-world scenarios where you have used Azure Functions effectively in a project?

In one of my projects, we used Azure Functions to automate the data processing workflow. Whenever a file was uploaded to Azure Blob Storage by the data team, a blob trigger function would run. It read the file, validated the data, and stored the results in Cosmos DB. This helped us process incoming data automatically without any manual steps.

In another project, we used Azure Functions with a Service Bus trigger to handle order processing in an e-commerce system. As soon as an order was placed, a message was added to the queue. The function picked up the message, updated the database, sent a confirmation email, and notified the shipping team.

We also used timer-triggered functions to clean up old logs and unused files from storage every night. This helped reduce storage cost and kept the environment clean.

Azure Functions worked very well in these cases because they are easy to scale, cost-effective, and quick to develop. They helped us respond to events in real time and reduced the need for running background services continuously.

© Shubham Wadekar

All rights reserved. Personal use only. Redistribution or resale is prohibited © Shubham Wadekar