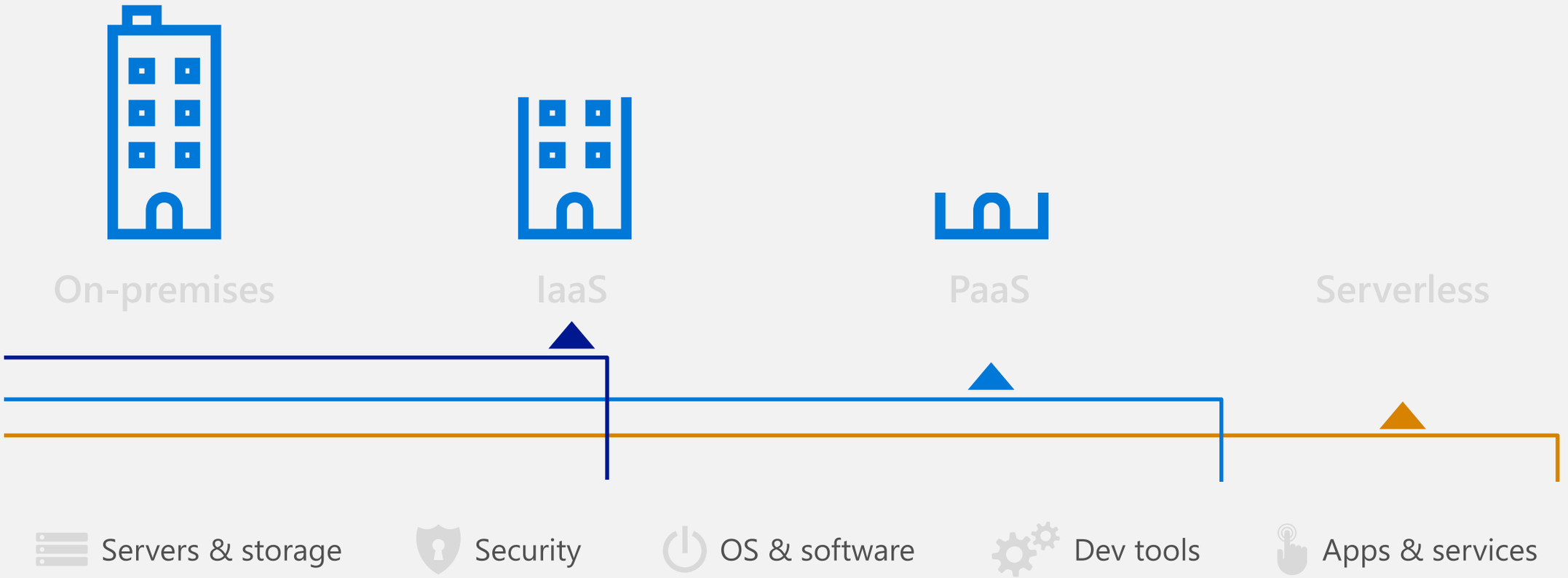
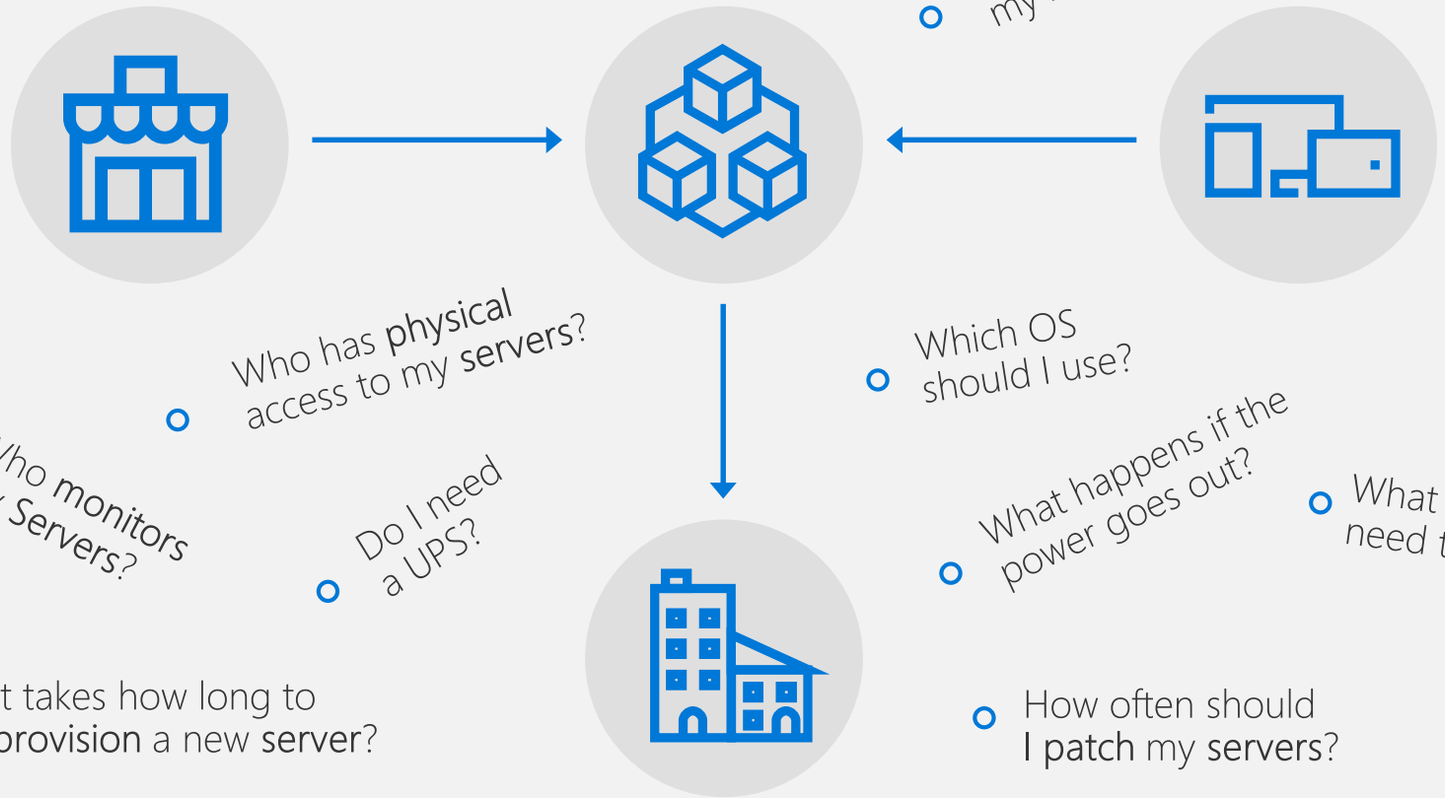
Azure Functions

History of cloud development

Increasingly advanced cloud technologies have led companies to entrust more and more of their IT activities to service providers



Before cloud…



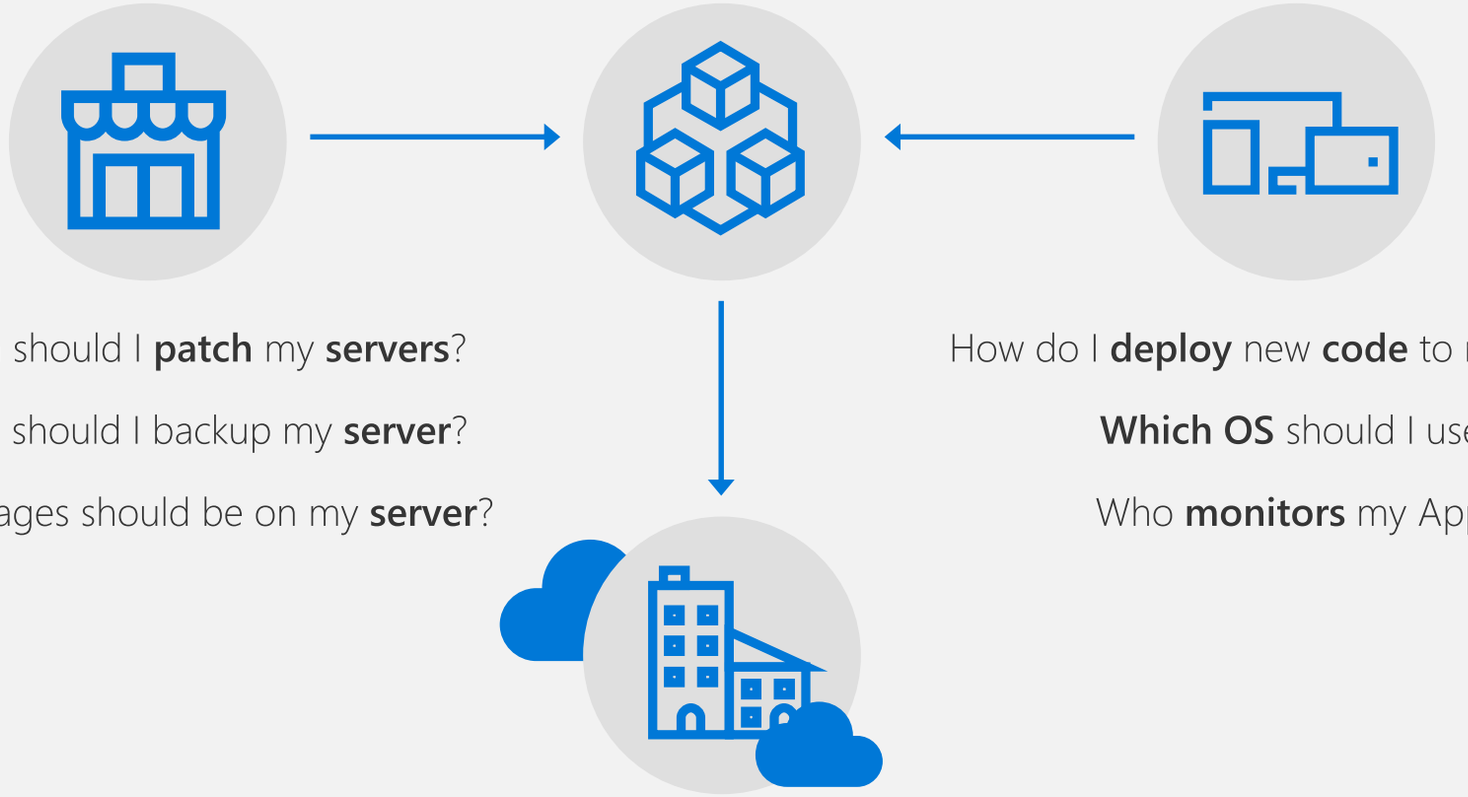
…then IaaS set the table stakes for digital business…

What is the right **size** of **servers** for my business needs?

How can I increase **server** utilization?

How many **servers** do I need?

How can I **scale** my app?



How often should I **patch** my **servers**?

How often should I backup my **server**?

Which packages should be on my **server**?

How do I **deploy** new **code** to my **server**?

Which OS should I use?

Who **monitors** my App?

…then PaaS, critical for digital transformation

What is the right **size** of **“servers”** for my business needs?

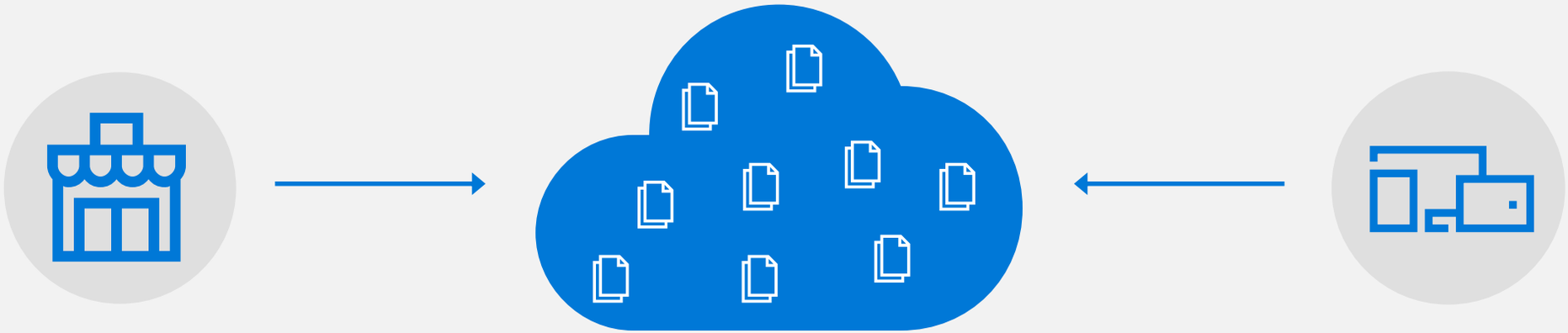
How can I increase **“server”** utilization?

How many **“servers”** do I need?

How can I **scale** my app?

Serverless: the platform for next gen apps, today

How do I **architect** my app to become Serverless?



What is serverless?

Full abstraction of servers

Developers can just focus on their code—there are no distractions around server management, capacity planning, or availability.

Instant, event-driven scalability

Application components react to events and triggers in near real-time with virtually unlimited scalability; compute resources are used as needed.

Pay-per-use

Only pay for what you use: billing is typically calculated on the number of function calls, code execution time, and memory used.*

*Supporting services, like storage and networking, may be charged separately.

What are the benefits?

Focus

Solve business problems—not technology problems related to undifferentiated heavy lifting

Efficiency

Shorter time to market
Fixed costs converted to variable costs
Better service stability
Better development and testing management
Less waste

Flexibility

Simplified starting experience
Easier pivoting means more flexibility
Easier experimentation
Scale at your pace—don't bet the farm on Day 1
Natural fit for microservices

FaaS is at the center of serverless

Functions-as-a-Service programming model use functions to achieve true serverless compute

Single responsibility

Functions are single-purposed, reusable pieces of code that process an input and return a result

Short lived

Functions don't stick around when finished executing, freeing up resources for further executions

Stateless

Functions don't hold any persistent state and don't rely on the state of any other processes

Event driven & scalable

Functions respond to predefined events, and are instantly replicated as many times as needed

What is Azure Functions?

An event-based, serverless compute experience that accelerates app development

Azure Functions = FaaS++

Integrated programming model

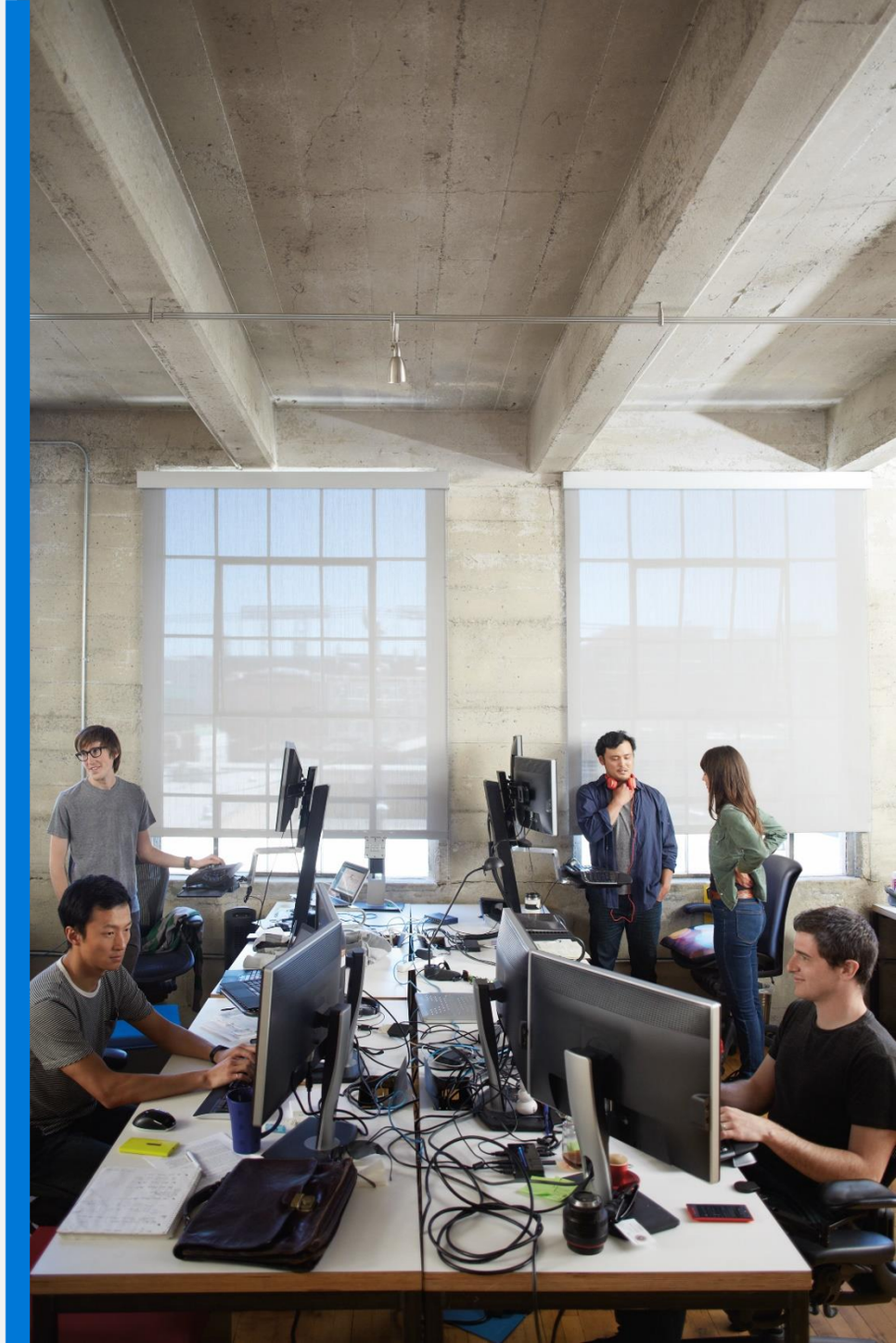
Use built-in triggers and bindings to define when a function is invoked and to what data it connects

Enhanced development experience

Code, test and debug locally using your preferred editor or the easy-to-use web based interface including monitoring

Hosting options flexibility

Choose the deployment model that better fits your business needs without compromising development experience

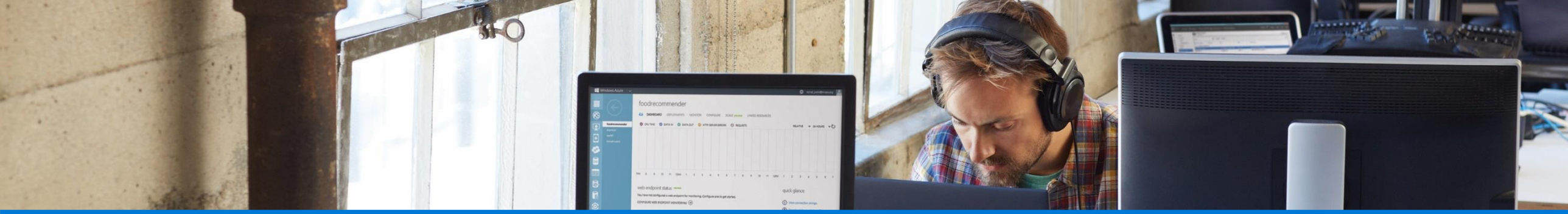


Focus on code, not plumbing

No infrastructure management

Auto-scale based on your workload

No wasted resources, pay only for what you use



Boost development efficiency

Triggers

Use triggers to define how functions are invoked
Avoid hardcoding with preconfigured JSON files
Build serverless APIs using HTTP triggers

Bindings

Connect to data with input and output bindings
Bind to Azure solutions and third-party services
Use HTTP bindings in tandem with HTTP triggers

Proxies

Define one API surface for multiple function apps
Create endpoints as reverse proxies to other APIs
Condition proxies to use variables

Local debugging

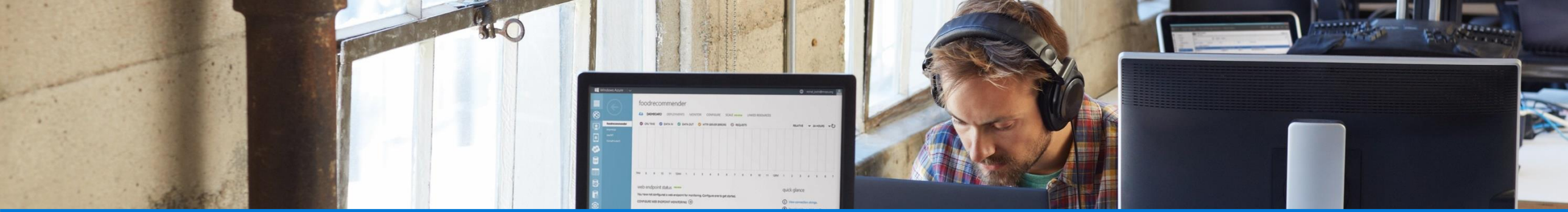
Debug C# and JavaScript functions locally
Use debugging tools in Azure portal, VS, and VS Code

CI/CD

Save time with built-in DevOps
Deploy functions using App Service for CI
Leverage Microsoft, partner services for CD

Monitoring

Integrate with Azure Application Insights
Get near real-time details about function apps
See metrics around failures, executions, etc.



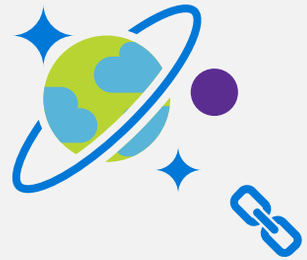
Boost development efficiency

Trigger object

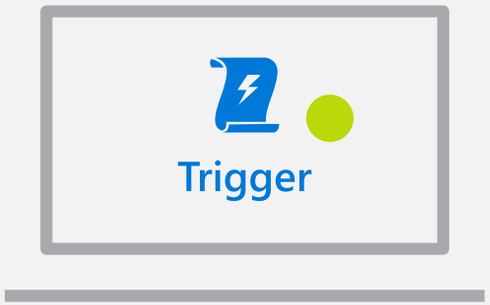
Your code

Input object

Output object



Input Binding



Output Binding

Gain flexibility and develop your way

Multiple languages

Write code in C#, JavaScript, F#, and Java
Continuous investment in new, experimental languages

Durable Functions

Write stateful functions in a serverless environment
Simplify complex, stateful coordination problems
Add the extension to enable advanced scenarios

Hosting options

Choose from six consumption plans to run Functions
Run your first million function executions for free

Dev options

Simplify coding for new users with native Azure portal
Select from popular editors, like VS, VS Code, CLI, Maven*

*VS and VS Code only support C#; Maven only supports Java

Gain flexibility and develop your way

Hosting options

Consumption
Serverless

Only pay for what you use; charges apply per execution and per GB second

AS Plan
Free, Basic, Standard, Premium

Gain all the advantages of Functions along with Microsoft's financially-backed SLA and the always-on features of an App Service Plan

AS Environment
Network isolation

Use a dedicated App Service cloud environment (ASE) that comes with network isolation for apps, greater scale, and secure connectivity to local vNets

Azure Stack
On-premises

Bring the power of the entire Azure stack to your own data centers

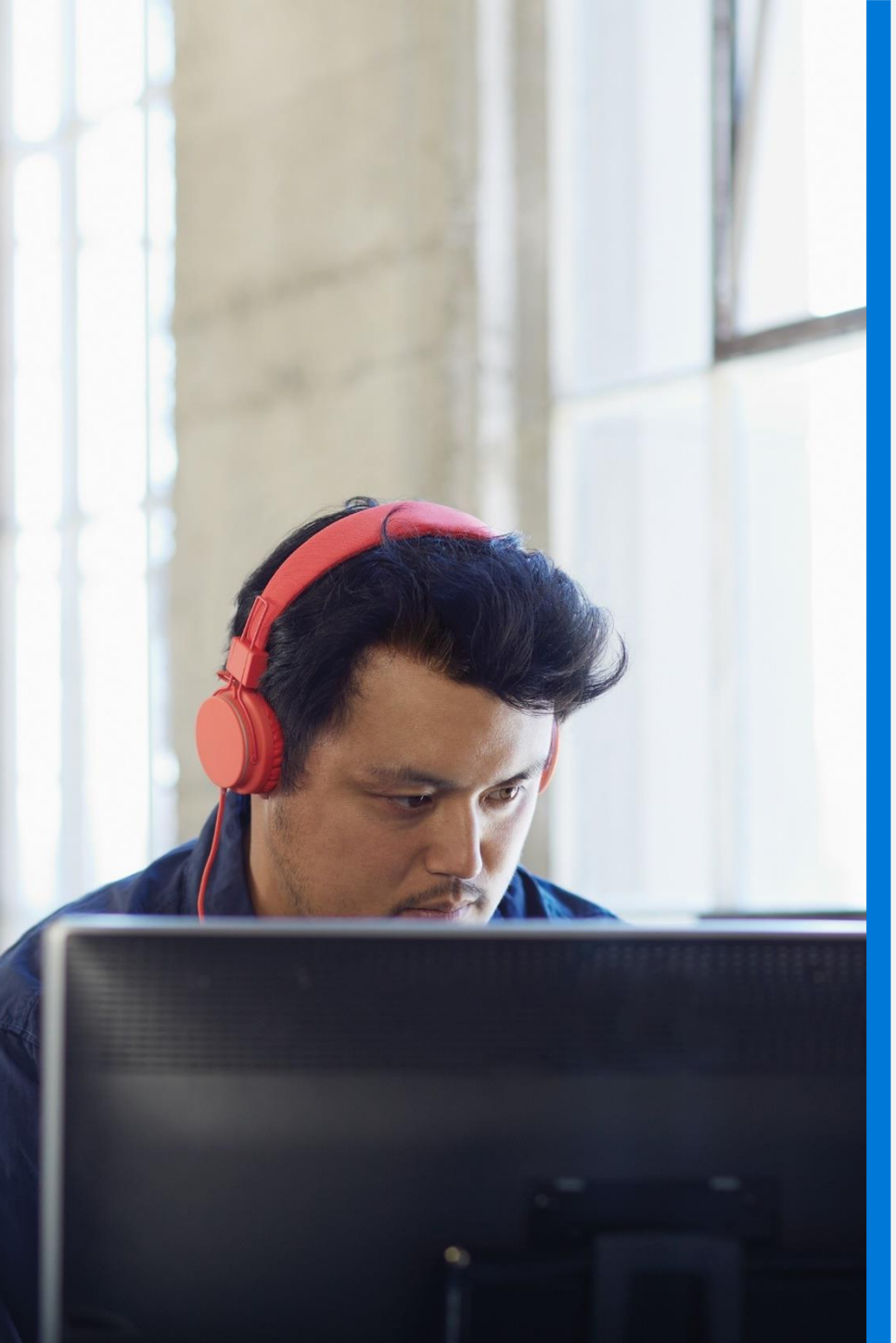
Runtime
Functions on your server

Run Functions on your local server; does not include the entire Azure stack

IoT Edge*
On devices

Deploy custom Azure modules on IoT devices

*Currently in preview



Sample scenarios for Functions

Web application backends

Mobile application backends

IoT-connected backends

Conversational bot processing

Real-time file processing

Real-time stream processing

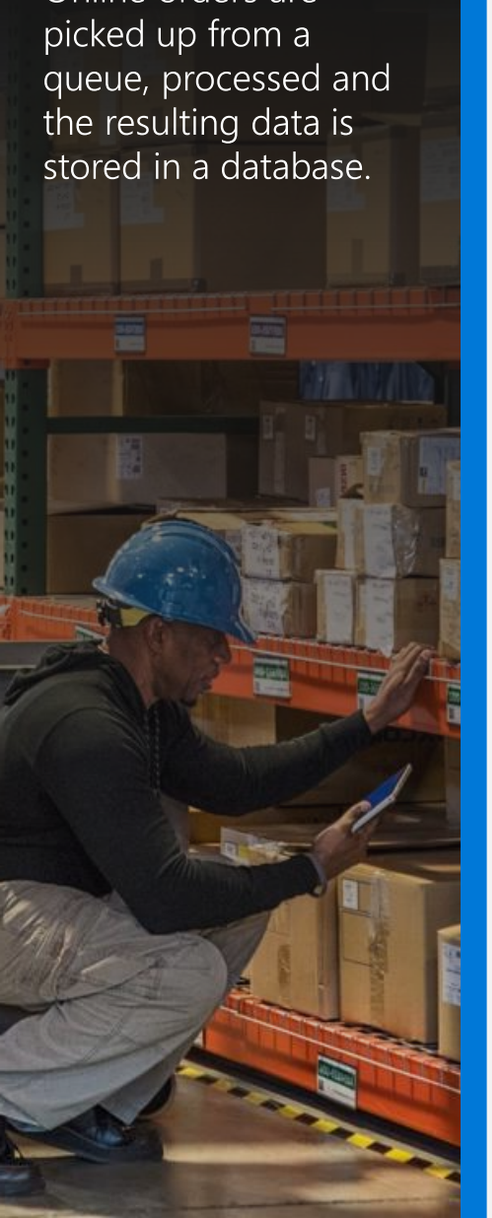
Automation of scheduled tasks

Extending SaaS Applications

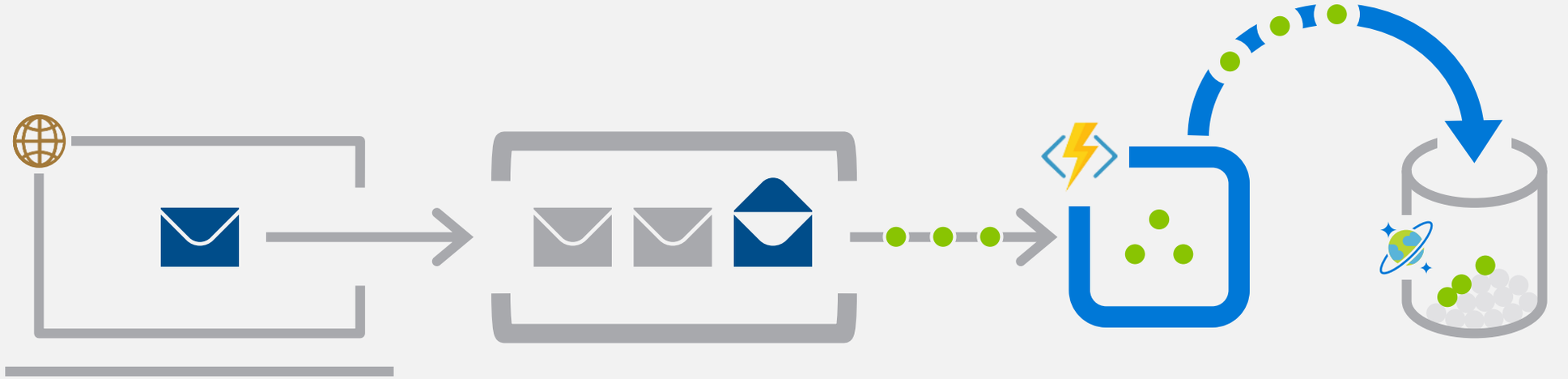
Scenario Example

Retail

Online orders are picked up from a queue, processed and the resulting data is stored in a database.



Web application backends



Request made in a web app

Request queued in Service Bus

A function processes the request...

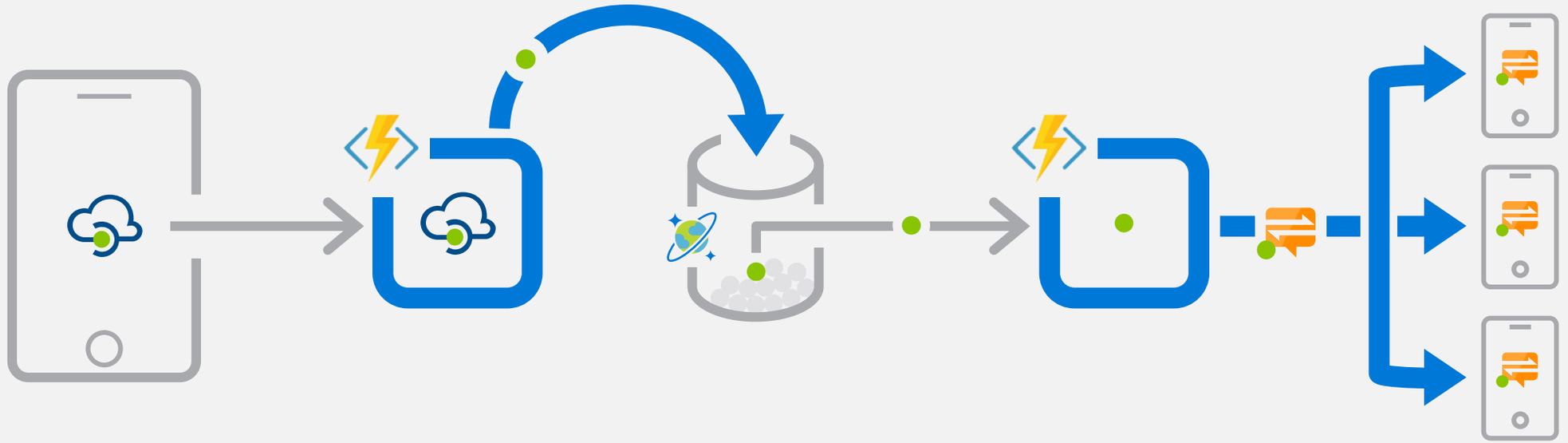
...sends output to Cosmos DB

Mobile application backends

Scenario Example

— Financial Services —

Colleagues use mobile banking to reimburse each other for lunch: the person who paid for lunch requests payment through his mobile app, triggering a notification on his colleagues' phones.



HTTP API call from a mobile app

Call processed by a function

Output data stored in Cosmos DB

Data transfer triggers second function…

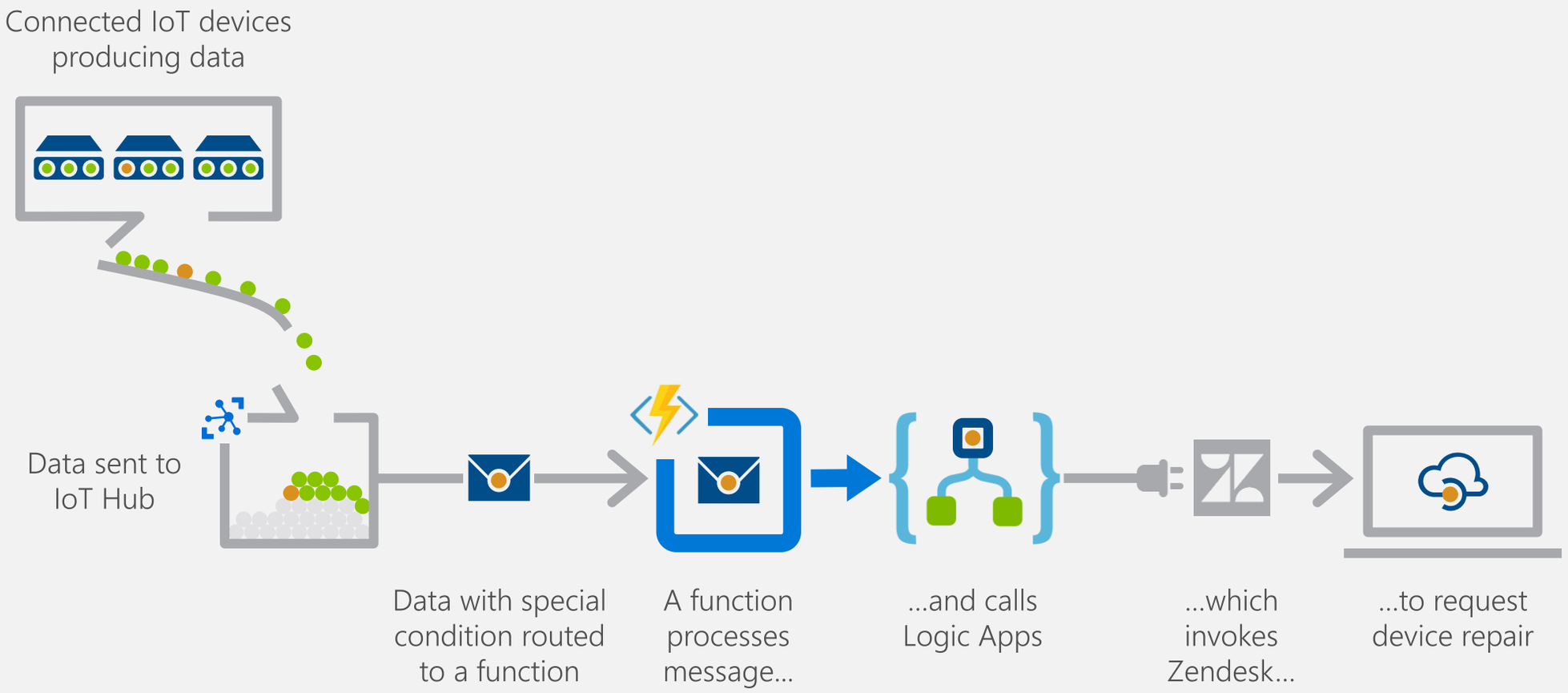
…which sends notifications using Notifications Hub

IoT-connected backends

Scenario Example

— Manufacturing —

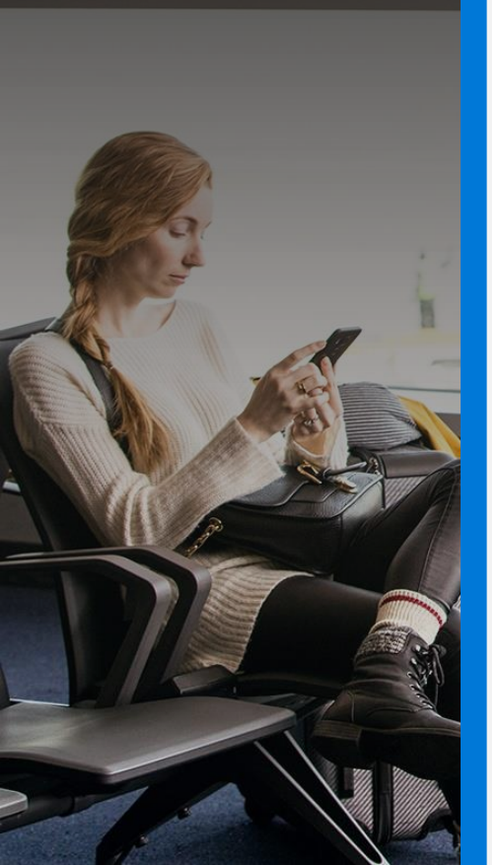
A manufacturing company uses IoT to monitor its machines. Functions detects anomalous data and triggers a message to Service department when repair is required.



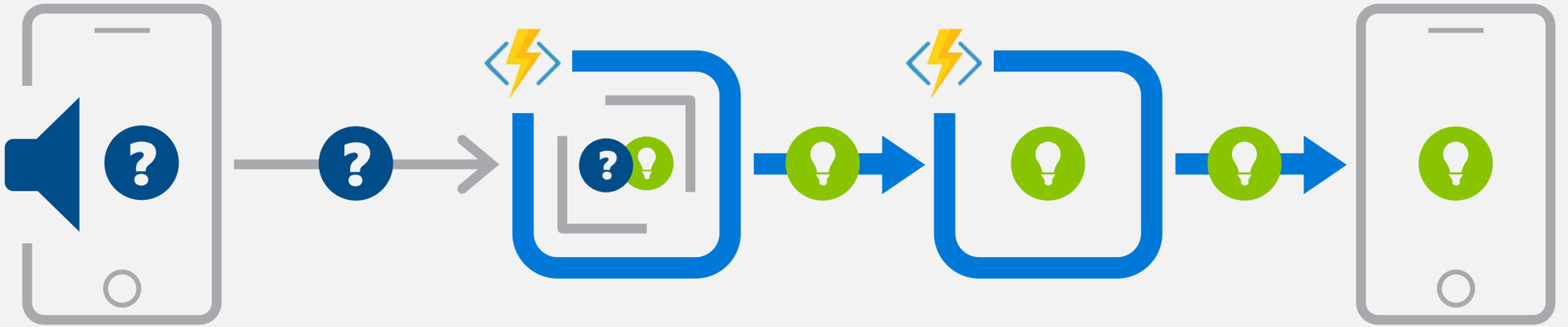
Scenario Example

Hospitality

Customer asks for available vacation accommodations on her smartphone. A serverless bot deciphers the request and returns vacation options.



Conversational bot processing



User request through conversational interface

Bot running in a function deciphers request using language understanding

Another function processes the request

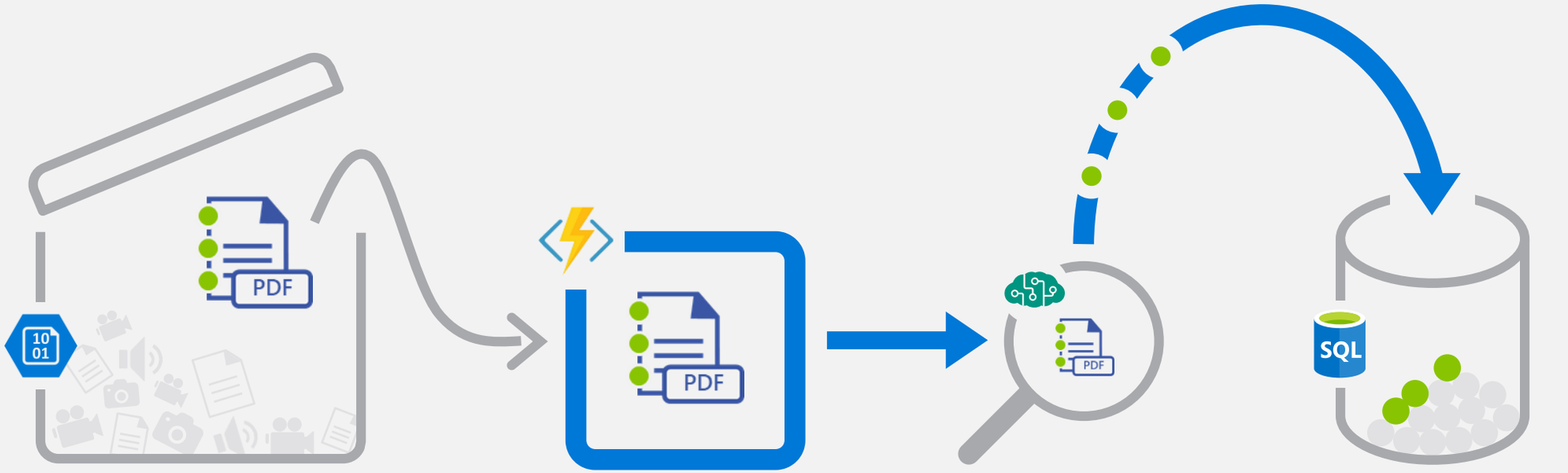
...and sends response to original requester

Scenario Example

Healthcare

Patient records are securely uploaded as PDF files. That data is then decomposed, processed using OCR detection, and added to a database for easy queries.

Real-time file processing

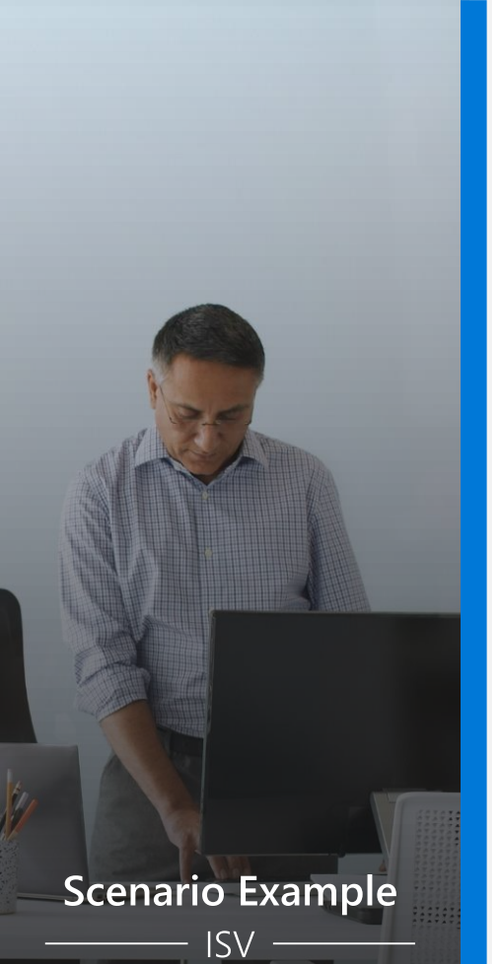


PDF file added to Blob Storage

A function decomposes PDF file…

…and sends it to Cognitive Services for OCR detection

Structured data from file sent to SQL DB

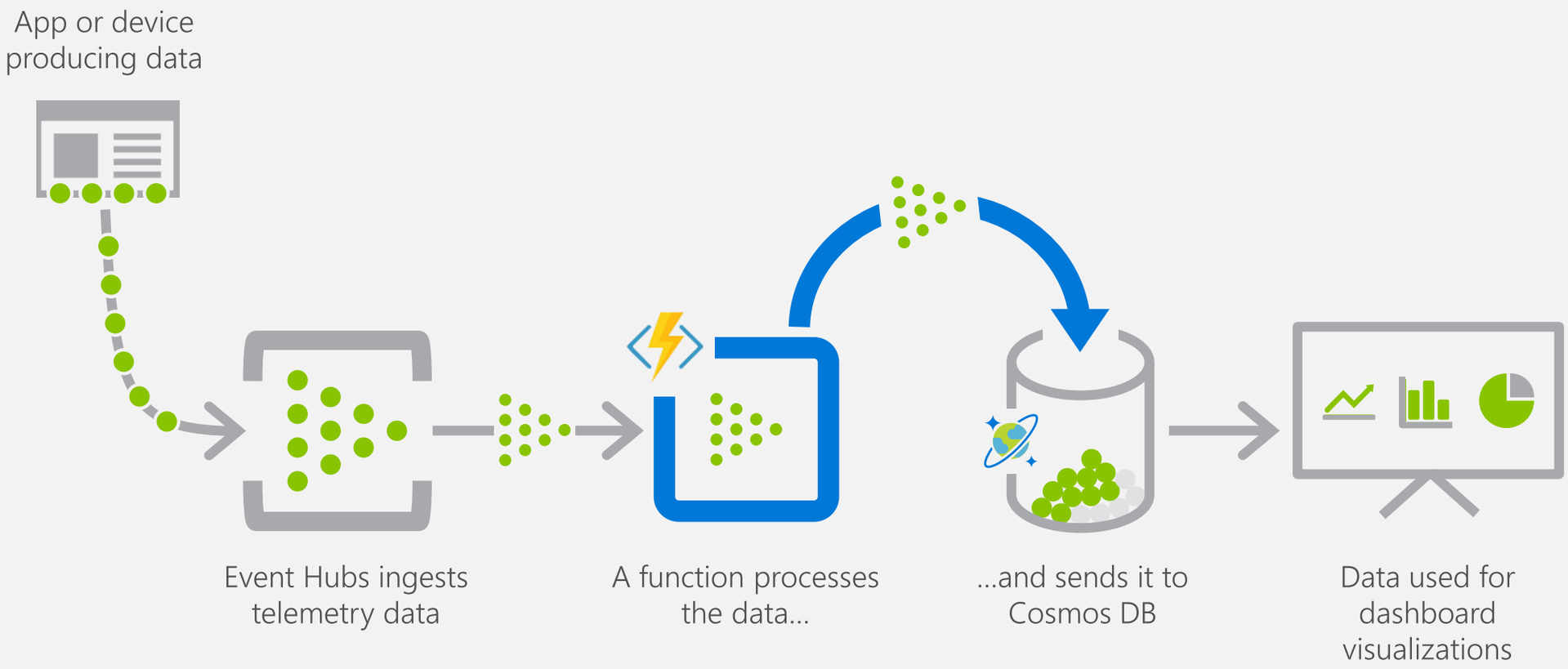


Scenario Example

ISV

Huge amounts of telemetry data is collected from a massive cloud app. That data is processed in near real-time and stored in a DB for use in an analytics dashboard.

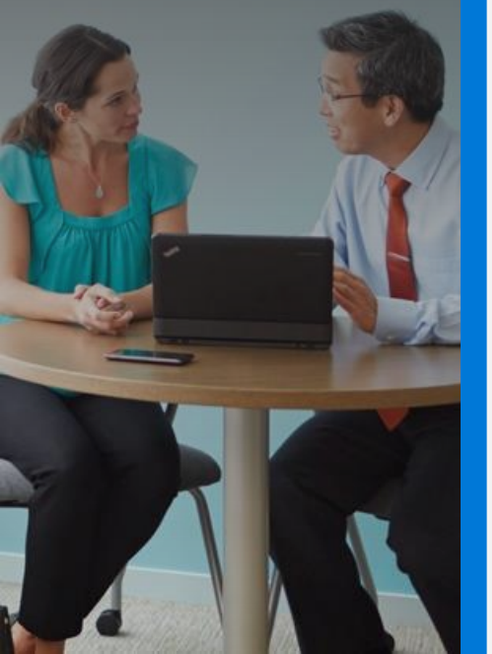
Real-time stream processing



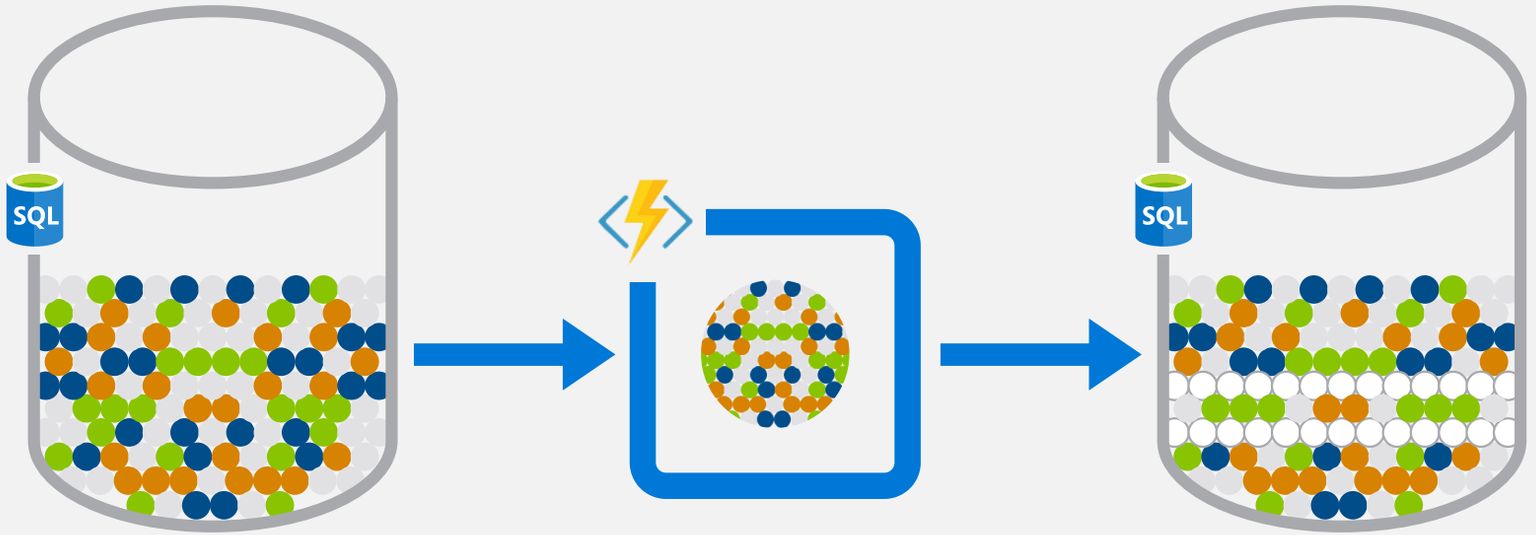
Scenario Example

— Financial Services —

A customer database is analyzed for duplicate entries every 15 minutes, to avoid multiple communications being sent out to same customers.



Automation of scheduled tasks



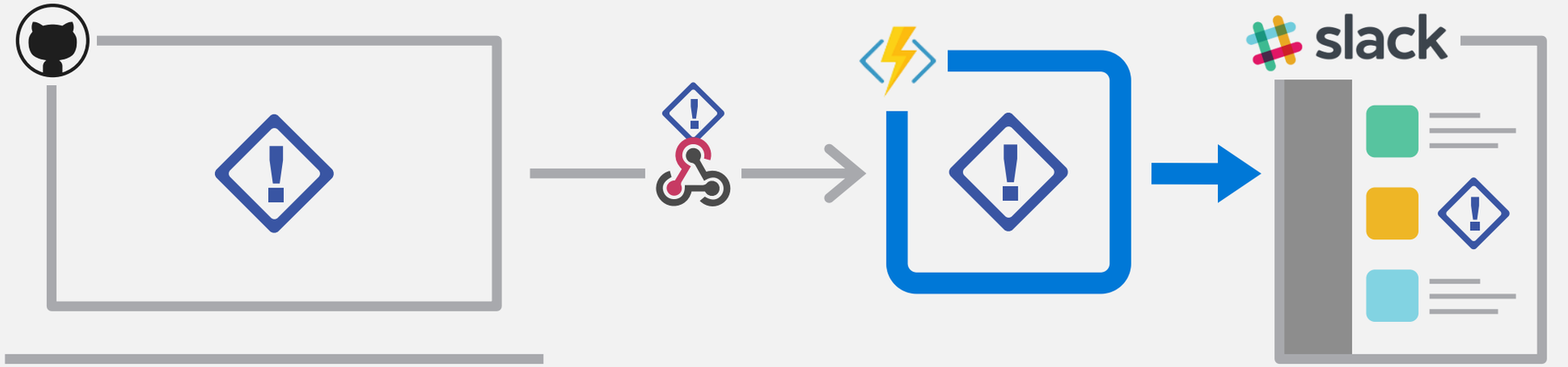
A function cleans a database every 15 minutes…

…deduplicating entries based on business logic

Scenario Example
—Professional Services—

A SaaS solution provides extensibility through webhooks, which can be implemented through Functions, to automate certain workflows.

Extending SaaS applications



Issue created in GitHub...

...which triggers a webhook call

...which is processed by a function...

...by posting the issue details to Slack

What makes Functions unique?

Intuitive experience
Easy-to-use, familiar tools like the Azure portal and Visual Studio help you get running on serverless fast

Flexible run options
With so many plan options, you can choose the level of access that's best for your company and customers

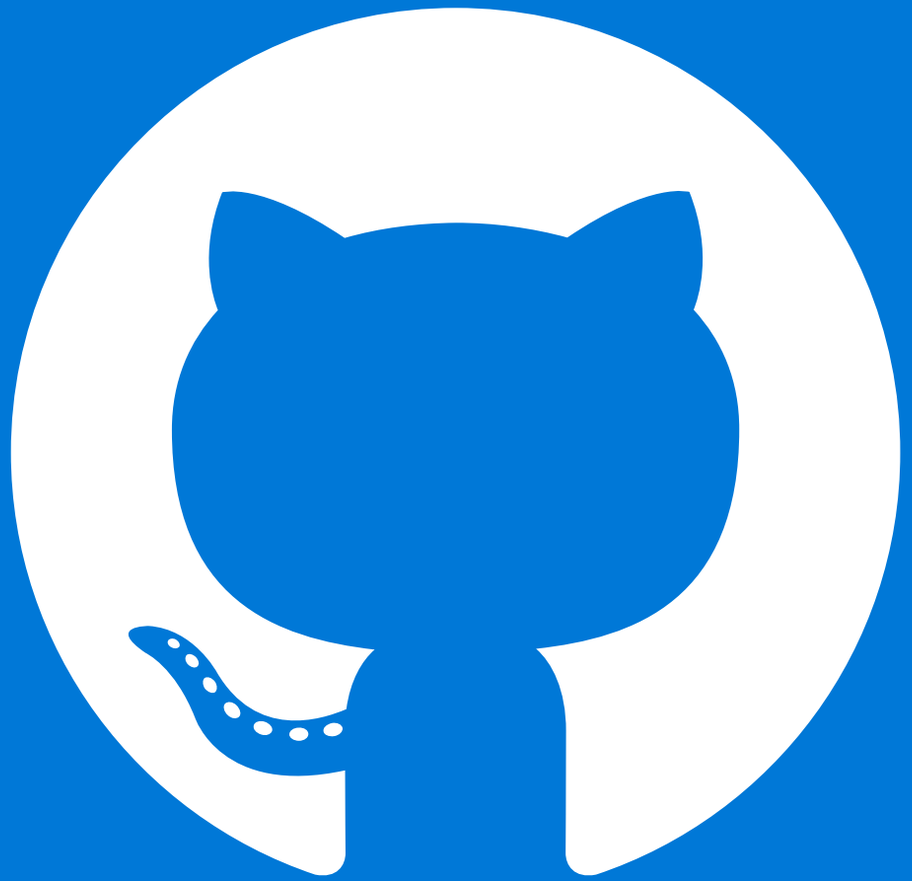
Feature variety
A host of features, from bindings to code editors, gives you the tools needed to quickly create the best apps

Durable Functions
With this extension Azure Functions help you simplify complex, stateful orchestration problems

Consolidated tools
Thanks to so many built-in solutions, you can do more in Functions than AWS, which often requires external tools

Azure Functions is an **open-source** project

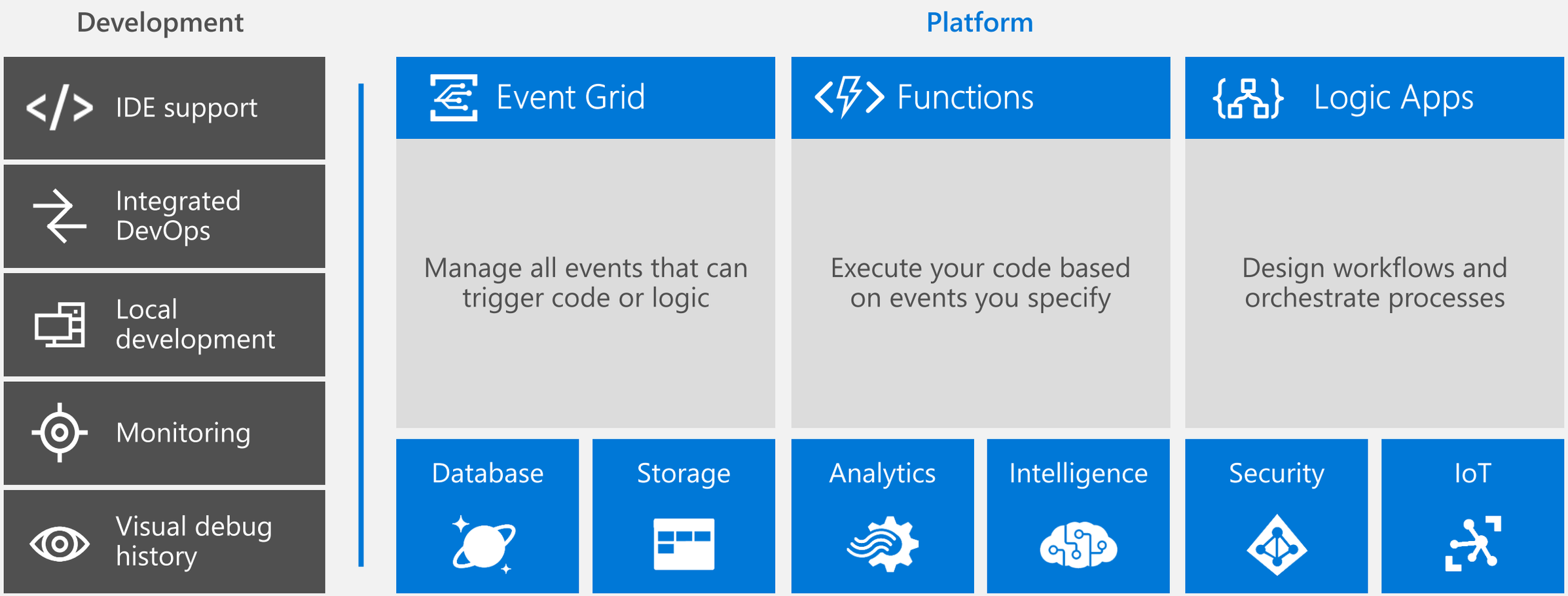
Functions runtime and all extensions are fully open source



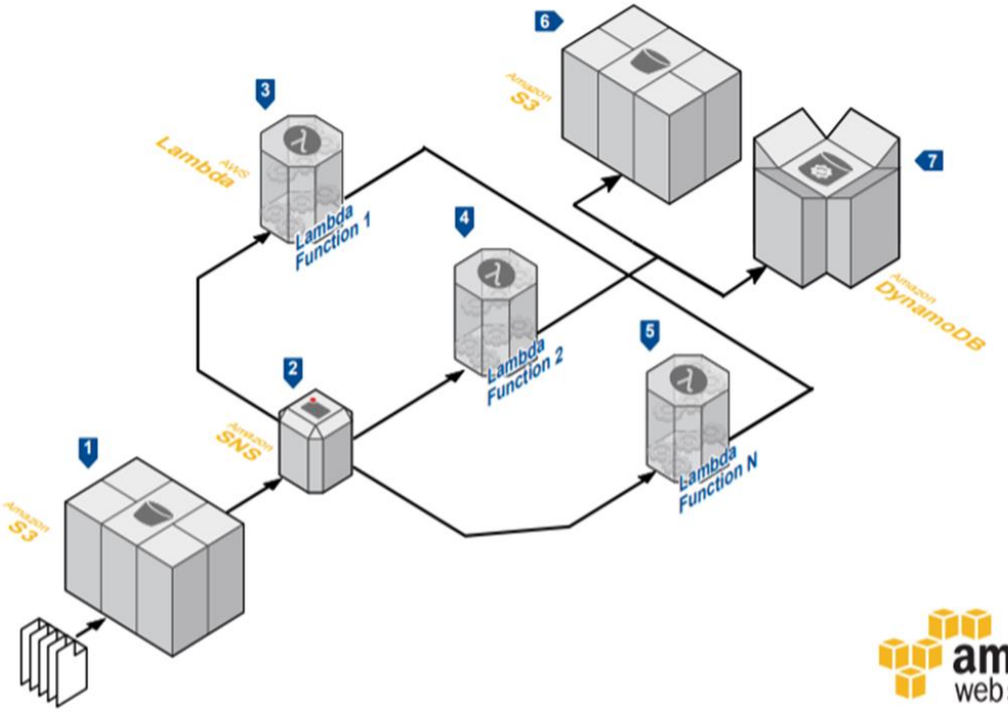
https://github.com/Azure/Azure-Functions

Full integration with Azure ecosystem

Functions is the center piece of the Serverless platform



AWS Serverless Stack



amazon web services

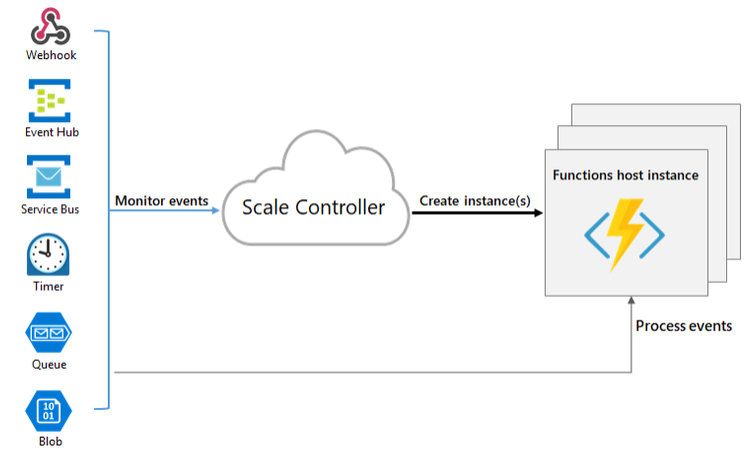
1. Objects are uploaded into **Amazon Simple Storage Service (S3)**, a highly available and persistent data store. The S3 bucket publishes an event notification to an **Amazon Simple Notification Service (Amazon SNS)** topic.
2. **Amazon SNS** is a fully managed push messaging system that can fan-out messages to multiple subscribers.
3. **AWS Lambda** is used to create a layer of processing for a wide variety of data sets and subsequently sends the results to a post-processing storage layer. For example, Lambda Function 1 can process objects put into the S3 bucket and create a data derivative.
4. Lambda Function 2 can, in parallel, process and create another data derivative from the same objects in S3.
5. N number of Lamba functions can be created to process data, all without the need to provision or manage servers.
6. A post-processing storage layer such as S3 can be added to store the results in a cost effective and durable fashion.
7. Alternatively, results can be sent to **Amazon DynamoDB** for further processing and/or querying with low-latency.

Built-in integration with Azure Services

- Azure Cosmos DB
- Azure Event Hubs
- Azure Event Grid
- Azure Mobile Apps (tables)
- Azure Notification Hubs
- Azure Service Bus (queues and topics)
- Azure Storage (blob, queues, and tables)
- GitHub (webhooks)
- On-premises (using Service Bus)
- Twilio (SMS messages)

Consumption Plan

- Automatically scales CPU and memory based on the number of events
- Scale unit is Function host (1.5 GB memory)
- Each instance of host is the Function App
- All functions within a function app shares scale together
- Watch for “slow starts”



Trigger

- Trigger defines how functions are invoked
- Bindings provide a declarative way to connect

Triggers, Input and Output Binding

Example scenario	Trigger	Input binding	Output binding
A new queue message arrives which runs a function to write to another queue.	Queue*	*None*	Queue*
A scheduled job reads Blob Storage contents and creates a new Cosmos DB document.	Timer	Blob Storage	Cosmos DB
The Event Grid is used to read an image from Blob Storage and a document from Cosmos DB to send an email.	Event Grid	Blob Storage and Cosmos DB	SendGrid
A webhook that uses Microsoft Graph to update an Excel sheet.	HTTP	*None*	Microsoft Graph

Bindings

Trigger Bindings

- **Orchestration Trigger Binding**
  - Triggers orchestrator functions
  - Polls control-queue
  - Partition-aware
  - Handles return values
- **Activity Trigger Binding**
  - Triggers activity functions
  - Polls work-item queue
  - Stateless
  - Handles return values

Non-Trigger Bindings

- **Orchestrator Client**
  - Output binding
  - Start new orchestrator instances
  - Terminate instances
  - Send event notifications
  - Fetch instance status*

Triggers and Bindings

Type	1.x	2.x	Trigger	Input	Output
Blob Storage	✔	✔[1]	✔	✔	✔
Cosmos DB	✔	✔	✔	✔	✔
Event Grid	✔	✔	✔		
Event Hubs	✔	✔	✔		✔
External File[2]	✔			✔	✔
External Table[2]	✔			✔	✔
HTTP	✔	✔[1]	✔		✔
Microsoft Graph Excel tables		✔		✔	✔
Microsoft Graph OneDrive files		✔		✔	✔
Microsoft Graph Outlook email		✔			✔

Bindings Example

```
{
  "bindings": [
    {
      "name": "order",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "%input-queue-name%",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    }
  ]
```

Bindings Example (Class Library)

```
[FunctionName("QueueTrigger")]
public static void Run(
    [QueueTrigger("%input-queue-name%")]string myQueueItem,
    ILogger log)
{
    log.LogInformation($"C# Queue trigger function processed: {myQueueItem}"
}
```

Function Code

```
#r "Newtonsoft.Json"

using Newtonsoft.Json.Linq;

// From an incoming queue message that is a JSON object, add fields and write to
// The method return value creates a new row in Table Storage
public static Person Run(JObject order, TraceWriter log)
{
    return new Person() {
            PartitionKey = "Orders",
            RowKey = Guid.NewGuid().ToString(),
            Name = order["Name"].ToString(),
            MobileNumber = order["MobileNumber"].ToString() };
}

public class Person
{
    public string PartitionKey { get; set; }
    public string RowKey { get; set; }
    public string Name { get; set; }
    public string MobileNumber { get; set; }
}
```

Register Azure Functions binding

- In Azure Functions version 2.x, bindings are available as separate packages

Azure Functions Binding Expression

```
    ...
    {
      "name": "imageSmall",
      "type": "blob",
      "path": "sample-images-sm/{filename}",
      "direction": "out",
      "connection": "MyStorageConnection"
    }
  ],
}
```

```
[FunctionName("ResizeImage")]
public static void Run(
    [BlobTrigger("sample-images/{filename}")] Stream image,
    [Blob("sample-images-sm/{filename}", FileAccess.Write)] Stream imageSmall,
    string filename,
    ILogger log)
{
    log.LogInformation($"Blob trigger processing: {filename}");
    // ...
}
```

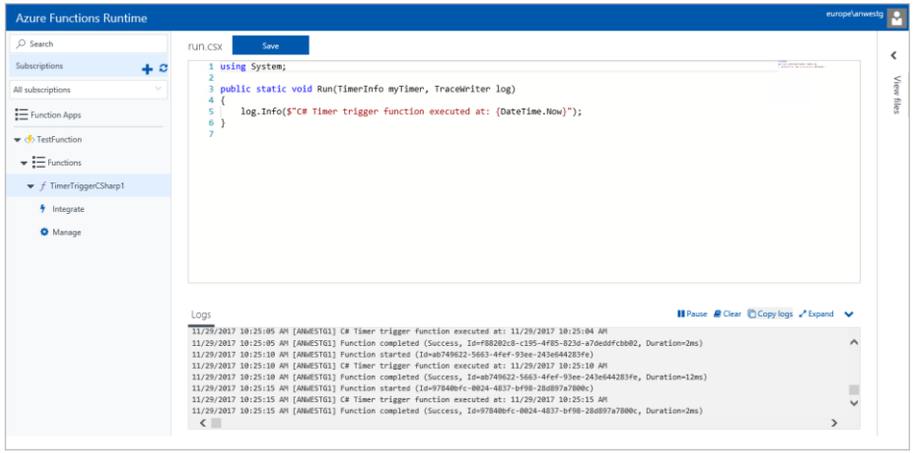
Binding expression DateTime

```
{
  "type": "blob",
  "name": "blobOutput",
  "direction": "out",
  "path": "my-output-container/{DateTime}.txt"
}
```

On-premises functions

- Run functions on-premises (no Azure subscription needed)
- Tap into unused on-premises compute resources

- Consists of
  - Management Roles
    - Portal
    - Publishing Endpoint
  - Worker Roles
    - Deployed inside Windows Containers
    - This is where functions execute
    - Deploy the Worker Roles across the organization



Function Proxies

```
{
    "$schema": "http://json.schemastore.org/proxies",
    "proxies": {
        "proxy1": {
            "matchCondition": {
                "methods": [ "GET" ],
                "route": "/api/{test}"
            },
            "backendUri": "https://<AnotherApp>.azurewebsites.net/api/<Funct
        }
    }
}
```

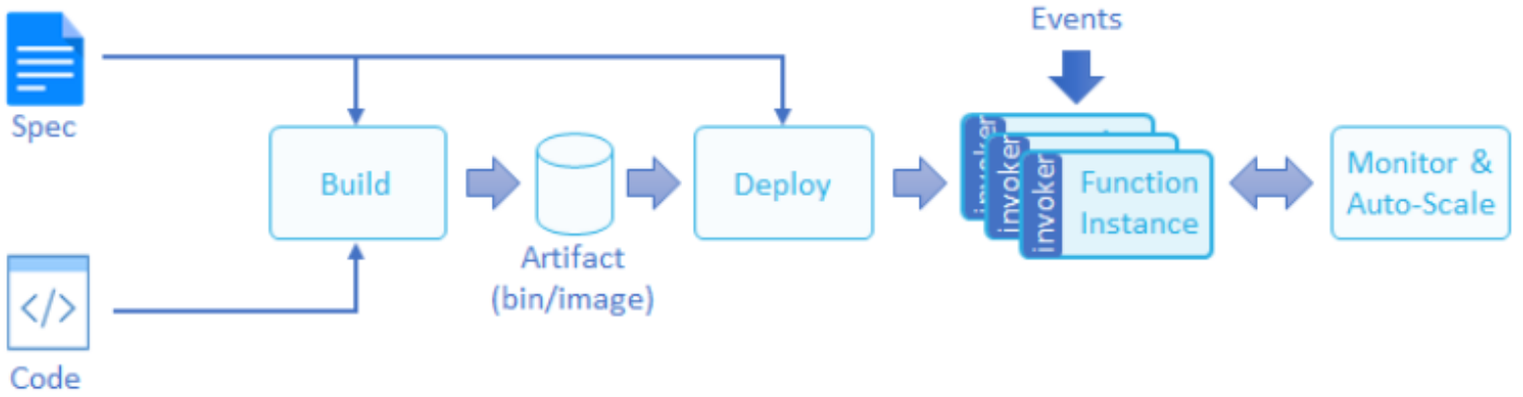
Function Proxies

```
{
    "$schema": "http://json.schemastore.org/proxies",
    "proxies": {
        "proxy1": {
            "matchCondition": {
                "methods": [ "GET" ],
                "route": "/api/{test}"
            },
            "backendUri": "https://<AnotherApp>.azurewebsites.net/api/<FunctionName>",
            "requestOverrides": {
                "backend.request.headers.Accept": "application/xml",
                "backend.request.headers.x-functions-key": "%ANOTHERAPP_API_KEY%"
            }
        }
    }
}
```

Function Proxies

```
{
    "$schema": "http://json.schemastore.org/proxies",
    "proxies": {
        "proxy1": {
            "matchCondition": {
                "methods": [ "GET" ],
                "route": "/api/{test}"
            },
            "responseOverrides": {
                "response.body": "Hello, {test}",
                "response.headers.Content-Type": "text/plain"
            }
        }
    }
}
```

Azure Functions SDLC



Additional Considerations

- Consider End to End solution design complexity
- Be cognizant of multi-tenant nature
- Cold Start
- A single function app will only scale to a maximum of 200 instances.
- A single instance may process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions.
- New instances will only be allocated at most once every 10 seconds

Pricing Plans

- **Consumption plan**: Azure provides all of the necessary computational resources.
- **Premium plan**: You specify a number of pre-warmed instances that are always online and ready to immediately respond. When your function runs, Azure provides any additional computational resources that are needed.
- **App Service plan**: Run your functions just like your web apps. If you use App Service for your other applications, your functions can run on the same plan at no additional cost.

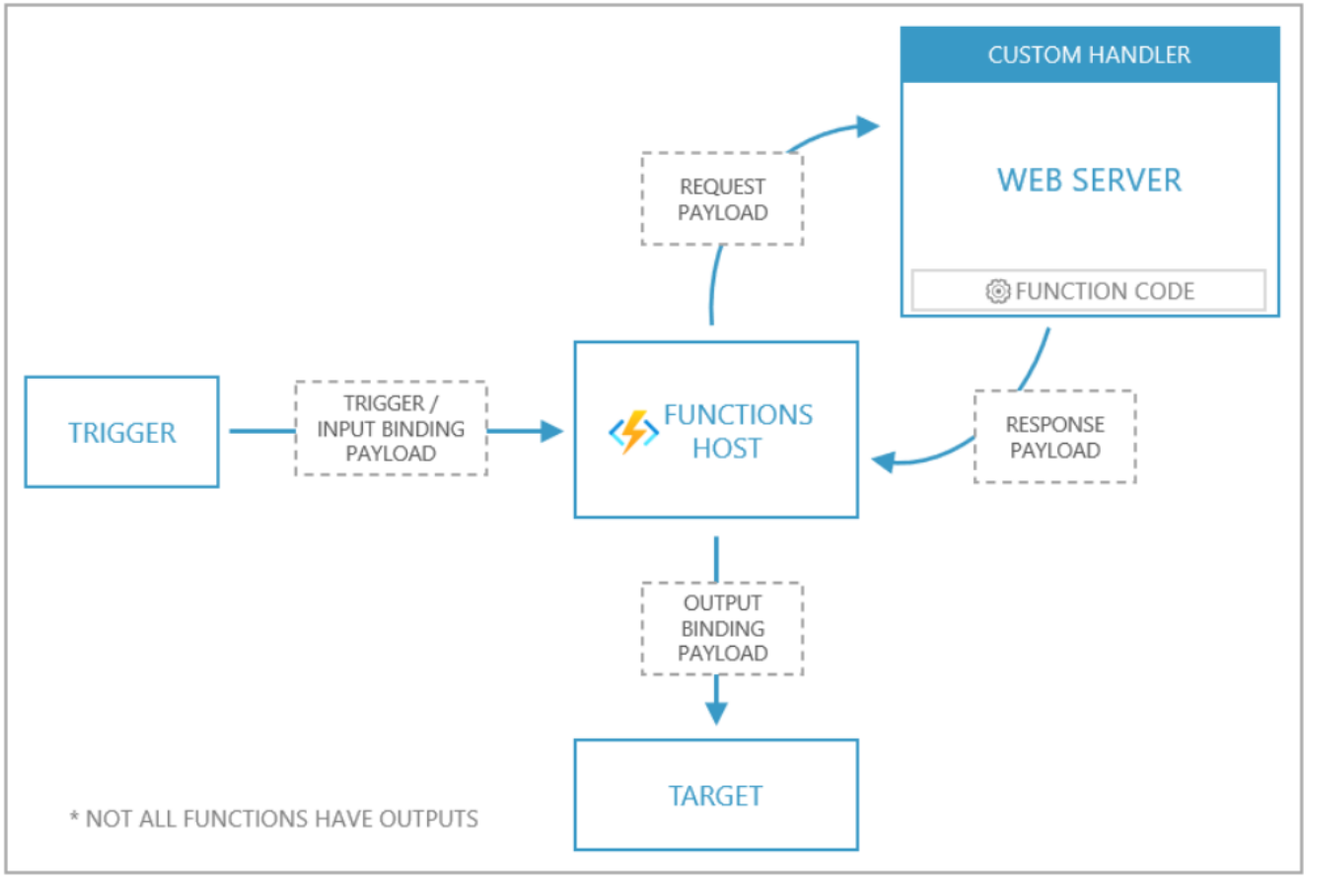
Premium Plan

- Azure Functions host are added and removed based on the number of incoming events
- Perpetually warm instances to avoid any cold start
- VNet connectivity
- Unlimited execution duration (60 minutes guaranteed)
- Premium instance sizes (one core, two core, and four core instances)
- More predictable pricing
- High-density app allocation for plans with multiple function apps

Matrix of networking features

	Consumption plan	Premium plan	App Service plan	App Service Environment
Inbound IP restrictions and private site access	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Virtual network integration	❌ No	✅ Yes (Regional)	✅ Yes (Regional and Gateway)	✅ Yes
Virtual network triggers (non-HTTP)	❌ No	✅ Yes	✅ Yes	✅ Yes
Hybrid connections (Windows only)	❌ No	✅ Yes	✅ Yes	✅ Yes
Outbound IP restrictions	❌ No	✅ Yes	✅ Yes	✅ Yes

Azure Functions custom handlers (preview)



Configure the HTTP-triggered function

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": ["get", "post"]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    }
  ]
}
```

Configure to run server.js file

```
{
    "version": "2.0",
    "httpWorker": {
        "description": {
            "defaultExecutablePath": "node",
            "defaultWorkerPath": "server.js"
        }
    }
}
```

server.js

```
const express = require("express");
const app = express();

app.use(express.json());

const PORT = process.env.FUNCTIONS_HTTPWORKER_PORT;

const server = app.listen(PORT, "localhost", () => {
  console.log(`Your port is ${PORT}`);
  const { address: host, port } = server.address();
  console.log(`Example app listening at http://${host}:${port}`);
});

app.get("/hello", (req, res) => {
  res.json("Hello World!");
});

app.post("/hello", (req, res) => {
  res.json({ value: req.body });
});
```

Durable Functions and Logic Apps

	Durable Functions	Logic Apps
Development	Code-first (imperative)	Designer-first (declarative)
Connectivity	About a dozen built-in binding types, write code for custom bindings	Large collection of connectors, Enterprise Integration Pack for B2B scenarios, build custom connectors
Actions	Each activity is an Azure function; write code for activity functions	Large collection of ready-made actions
Monitoring	Azure Application Insights	Azure portal, Operations Management Suite, Log Analytics
Management	REST API, Visual Studio	Azure portal, REST API, PowerShell, Visual Studio
Execution context	Can run locally or in the cloud.	Runs only in the cloud.

Flow vs Logic Apps

	Flow	Logic Apps
Users	Office workers, business users, SharePoint administrators	Pro integrators and developers, IT pros
Scenarios	Self-service	Advanced integrations
Design Tool	In-browser and mobile app, UI only	In-browser and Visual Studio, Code view available
Application Lifecycle Management (ALM)	Design and test in non-production environments, promote to production when ready.	DevOps: source control, testing, support, automation, and manageability in Azure Resource Management
Admin Experience	Manage Flow Environments and Data Loss Prevention (DLP) policies, track licensing https://admin.flow.microsoft.com	Manage Resource Groups, Connections, Access Management, and Logging https://portal.azure.com
Security	Office 365 Security and Compliance audit logs, Data Loss Prevention (DLP), encryption at rest for sensitive data, etc.	Security assurance of Azure: Azure Security, Security Center, audit logs, and more.

Deployment Options

Deployment technology	Windows Consumption	Windows Premium	Windows Dedicated	Linux Consumption
External package URL[1]	✓	✓	✓	✓
Zip deploy	✓	✓	✓	✓
Docker container				
Web Deploy	✓	✓	✓	
Source control	✓	✓	✓	
Local Git[1]	✓	✓	✓	
Cloud sync[1]	✓	✓	✓	

Performance Considerations

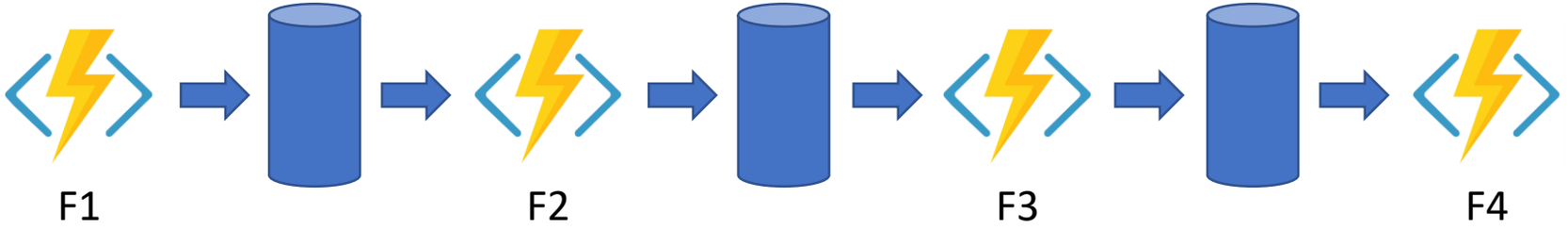
- Avoid long running functions
- Cross function communication
  - Use storage queues
  - Durable functions or Logic Apps
  - Service Bus topics are useful if you require message filtering before processing
  - Event hubs are useful to support high volume communications.
- Write functions to be stateless

maxConcurrentRequests

Scalability Considerations

- Reuse connections to external resources whenever possible
- To maximize performance, use a separate storage account for each function app
- Avoid referencing the Result property or calling Wait method on a Task instance
- Multiple worker processes
  - FUNCTIONS_WORKER_PROCESS_COUNT
- Receive messages in batch whenever possible
- Configure host behaviors to better handle concurrency
  - maxConcurrentRequests

Pattern #1: Function chaining - Today



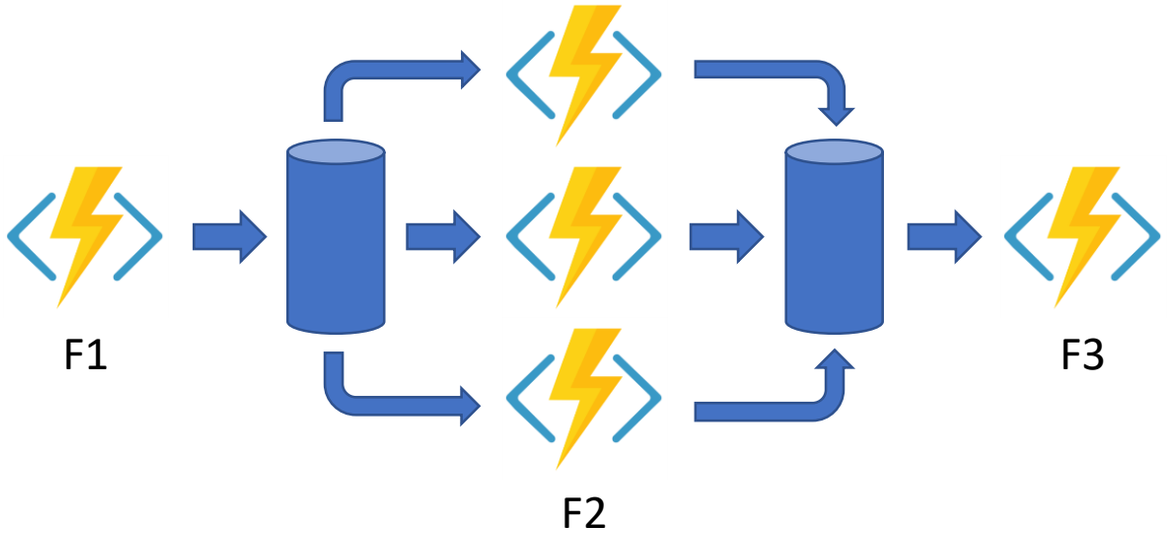
Problems:

- No visualization to show relationship between functions and queues.
- Middle queues are an implementation detail – conceptual overhead.
- Error handling adds a lot more complexity.

Pattern #1: Function chaining - Better

```
// calls functions in sequence
public static async Task<object> Run(DurableOrchestrationContext ctx)
{
    try
    {
        var x = await ctx.CallFunctionAsync("F1");
        var y = await ctx.CallFunctionAsync("F2", x);
        var z = await ctx.CallFunctionAsync("F3", y);
        return  await ctx.CallFunctionAsync("F4", z);
    }
    catch (Exception)
    {
        // global error handling/compensation goes here
    }
}
```

Pattern #2: Fan-out/Fan-in - Today



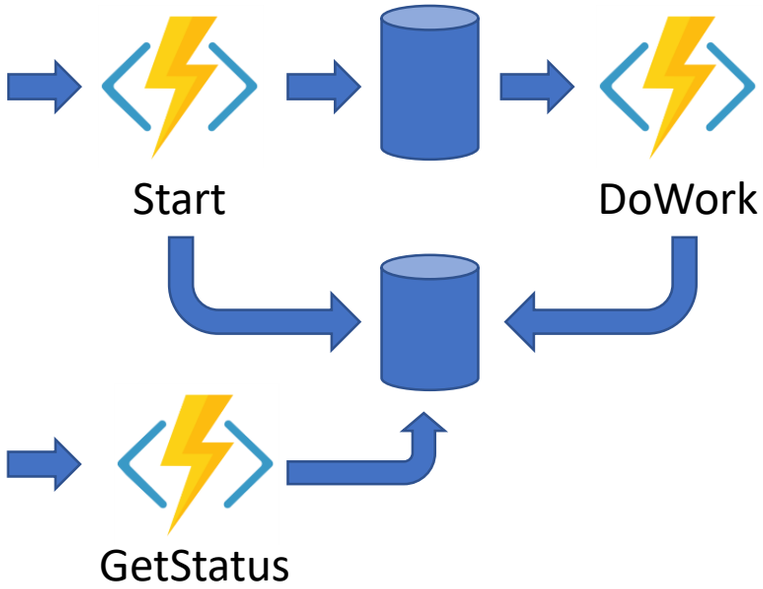
Problems:

- Fanning-out is easy, but fanning-in is significantly more complicated
- Functions offers no help with this scenario today
- All the same problems of the previous pattern

Pattern #2: Fan-out/Fan-in - Easy

```
public static async Task Run(DurableOrchestrationContext ctx)
{
    var parallelTasks = new List<Task<int>>();

    // get a list of N work items to process in parallel
    object[] workBatch = await ctx.CallFunctionAsync<object[]>("F1");
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = ctx.CallFunctionAsync<int>("F2", workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // aggregate all N outputs and send result to F3
    int sum = parallelTasks.Sum(t => t.Result);
    await ctx.CallFunctionAsync("F3", sum);
}
```

Pattern #3: HTTP Async Response



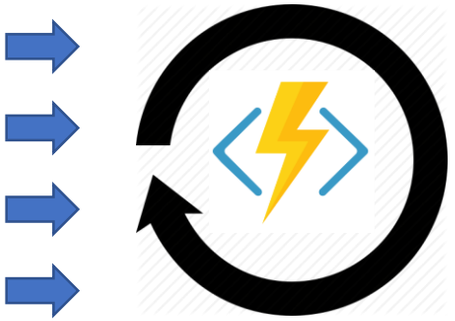
Problems:

- Execution state needs to be explicitly stored and managed.
- Execution state and trigger state must be kept in sync manually.
- *Start* and *GetStatus* is often boilerplate code that is not related to the business problem.

Pattern #3: HTTP Async Response – Built in!

```
> curl -X POST https://myfunc.azurewebsites.net/orchestrators/DoWork -H "Content-Length: 0" -i

HTTP/1.1 202 Accepted
Location: https://myfunc.azurewebsites.net/orchestrators/b79baf67f717453ca9e86c5da21e03ec
Server: Microsoft-IIS/8.0


> curl https://myfunc.azurewebsites.net/orchestrators/b79baf67f717453ca9e86c5da21e03ec -i

HTTP/1.1 202 Accepted
Content-Length: 173
Content-Type: application/json
Location: https://myfunc.azurewebsites.net/orchestrators/b79baf67f717453ca9e86c5da21e03ec
Server: Microsoft-IIS/8.0

{"runtimeStatus":"Running","createdTime":"2017-03-16T21:20:36Z","lastUpdatedTime":"2017-03-16T21:20:47Z"}


> curl https://myfunc.azurewebsites.net/orchestrators/b79baf67f717453ca9e86c5da21e03ec -i

HTTP/1.1 200 OK
Content-Length: 175
Content-Type: application/json
Server: Microsoft-IIS/8.0

{"runtimeStatus":"Completed","createdTime":"2017-03-16T21:20:36Z","lastUpdatedTime":"2017-03-16T21:20:57Z"}
```

Pattern #4: Actors



Problems:

- Functions are stateless and short-lived.
- Read/write access to external state needs to be carefully synchronized.
- Functions do not support the singleton pattern today.

Pattern #4: Actors (Eternal Orchestrations)

```
public static async Task Run(DurableOrchestrationContext ctx)
{
    int counterState = ctx.GetInput<int>();

    string operation = await ctx.WaitForExternalEvent<string>("operation");
    if (operation == "incr")
    {
        counterState++;
    }
    else if (operation == "decr")
    {
        counterState--;
    }

    ctx.ContinueAsNew(counterState);
}
```

Pattern #5: Human Interaction w/Timeout

Problems:

- Can’t easily coordinate a timeout with an approval request notification.
- Need a mechanism to reliably cancel either the approval handling or the timeout depending on the outcome.

Pattern #5: Human Interaction w/Timeout

```
public static async Task Run(DurableOrchestrationContext ctx)
{
    await ctx.CallFunctionAsync<object[]>("RequestApproval");
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = ctx.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = ctx.CreateTimer(dueTime, 0, cts.Token);

        Task<bool> approvalEvent = ctx.WaitForExternalEvent<bool>("ApprovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTimeout))
        {
            timeoutCts.Cancel();
            await ctx.CallFunctionAsync("HandleApproval", approvalEvent.Result);
        }
        else
        {
            await ctx.CallFunctionAsync("Escalate");
        }
    }
}
```

Important Orchestrator Limitations

- Orchestrator code is **replayed on every rehydration** to restore all local state (local variables, etc).
  - Function calls are never replayed – the outputs are remembered.

- This requires the orchestrator code to be deterministic.
  - Rule #1: Never write logic that depends on random numbers, DateTime.Now, Guid.NewGuid(), etc.
  - Rule #2: Never do I/O directly in the orchestrator function.
  - Rule #3: Do not write infinite loops
  - Rule #4: Use the built-in workarounds for rules #1, #2, and #3

Durable Functions

- Stateful functions
- Completely define stateful workflow in code
- Call other functions synchronously and asynchronously
- Automatically checkpoint state (“durably” i.e. survives process recycles and VM reboots)
- Built on Durable Task Framework

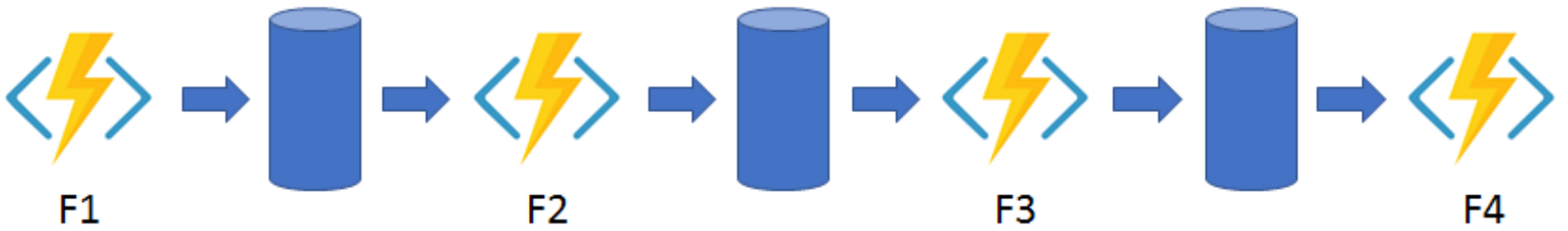
Orchestrator and Activity Functions

- Orchestrator Functions
  - Stateful
  - Orchestrates activity functions
  - Sub-orchestrations are also supported
  - Avoid compute intensive tasks or IO bound tasks
  - Deterministic

- Activity Functions
  - Stateless

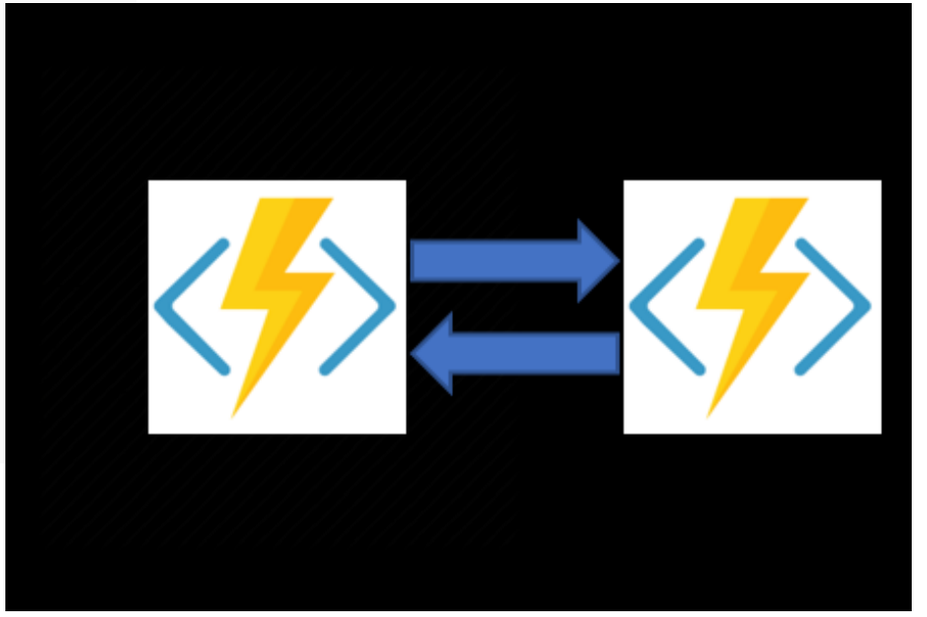
Durable Functions Example 1 of 2

```
public static async Task<object> Run(DurableOrchestrationContext ctx)
{
    try
    {
        var x = await ctx.CallActivityAsync<object>("F1");
        var y = await ctx.CallActivityAsync<object>("F2", x);
        var z = await ctx.CallActivityAsync<object>("F3", y);
        return  await ctx.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // error handling/compensation goes here
    }
}
```



Durable Functions Example 2 of 2

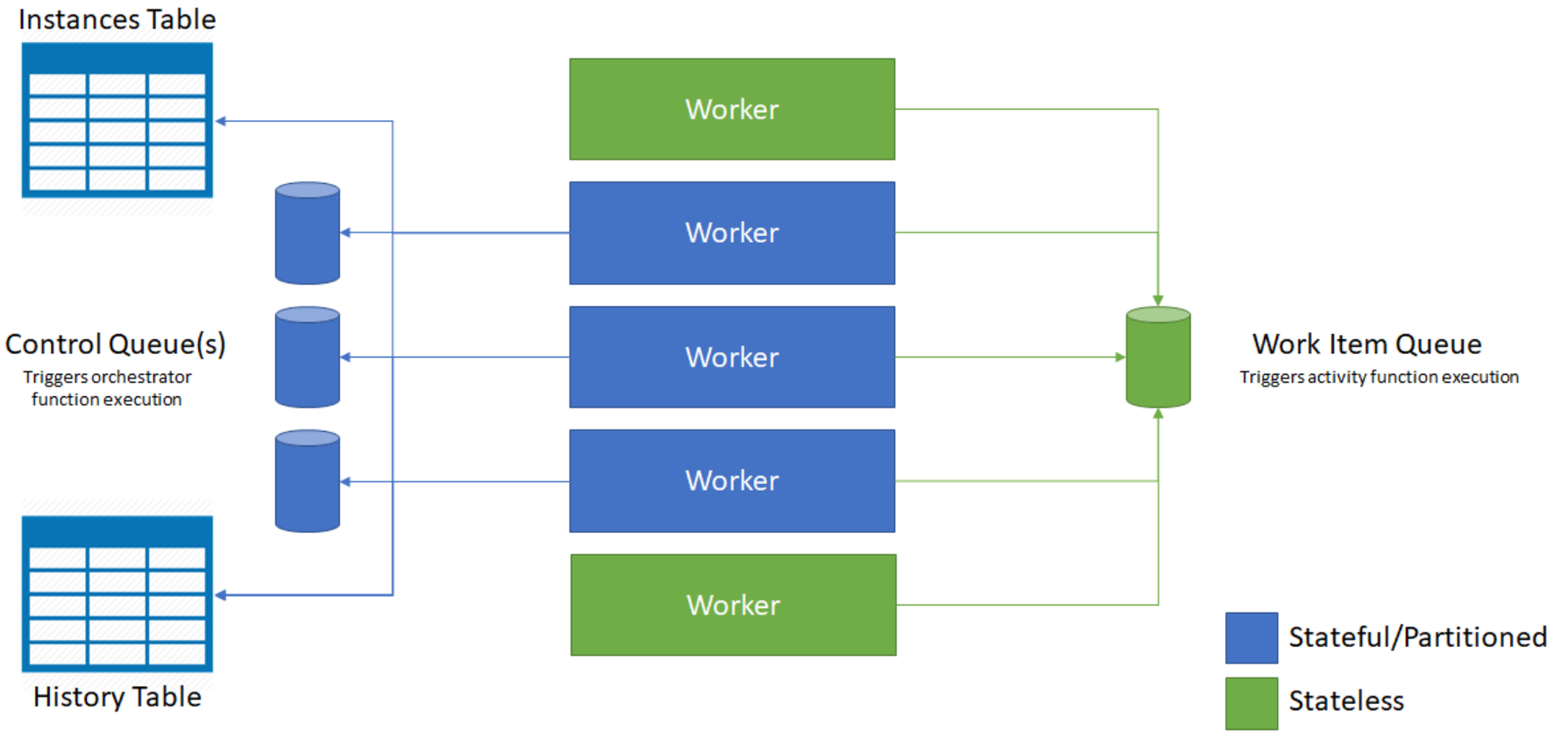
```
public static async Task Run(DurableOrchestrationContext ctx)
{
    int jobId = ctx.GetInput<int>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

    while (ctx.CurrentUtcDateTime < expiryTime)
    {
        var jobStatus = await ctx.CallActivityAsync<string>("GetJobStatus", jobId);
        if (jobStatus == "Completed")
        {
            // Perform action when condition met
            await ctx.CallActivityAsync("SendAlert", machineId);
            break;
        }

        // Orchestration will sleep until this time
        var nextCheck = ctx.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await ctx.CreateTimer(nextCheck, CancellationToken.None);
    }

    // Perform further work here, or let the orchestration end
}
```



Durable Functions

- Use Azure Storage queues, tables and blobs to persist state (and execution history)
- Storage account created under the covers
- No storage access logic is needed
- Partitioning considerations

Durable Functions State Management



Durable Functions Essentials

Why do we need “Durable Functions”

- Enable “long running” functions while maintaining local state.
- Simplify *complex* Function coordination (chaining, etc.)
- Easily call a Function from another Function
- All of the above using code-only

What is Durable Functions?

- Advanced feature for writing long-running orchestrations as a single C# function. No JSON schemas. No designer.
- New **orchestrator functions** can synchronously or asynchronously call other functions.
- Automatic **checkpointing**, enabling “long running” functions.
- Solves a variety of complex, transactional coding problems in serverless apps.
- Built on the open source Durable Task Framework.

Key Takeaways

- Orchestrator functions are:
  - **Code**: code-oriented way to implement business logic.
  - **Stateful**: local variables and execution progress is preserved.
  - **Durable**: no state is lost when the process recycles or the VM reboots.
  - **Long-running:** they can theoretically run forever:
    - **Can go to sleep** (dehydrate) and **billing can be stopped** while waiting for results.
    - **Automatically wake up** (hydrate) when there is more work to do.
  - **Scalable**: queue-based messaging allows for massive scale-out.

Functions and Logic Apps

- Functions and Logic Apps work great together and that integration will only get better.
- You can orchestrate Functions using Logic Apps today.

- Durable Functions overlaps with Logic Apps, one function calling another, but:
  - Offers advance ‘code only’ orchestration ‘like’ capabilities to function
  - Enables "long running" functions
  - Enables "advance" scenarios/patterns which are difficult otherwise: async HTTP, map reduce, actors, etc.