Cloud Team - Buildr

- YAML and GitHub Actions
- DevOps with Microsoft Azure
- Azure Services: A Tech Team Quick Reference Page
- Introduction to Azure Service Bus
- Smart Bank – AI Powered Banking Assistant
- OpenTelemetry Developer Handbook – Azure, GCP & AWS
- Zero-Downtime Deployment in Azure App Service: Deployment Slots, Health Checks and Rollbacks

YAML and GitHub Actions

YAML Fundamentals and GitHub Actions Workflows

Understanding Configuration as Code and Workflow Automation

By Presica Peter Pinto • Cloud Team

> Modern DevOps teams succeed when important work is repeatable, and repeatable work is automated. **YAML** gives teams a clear way to describe that automation in code. **GitHub Actions** then executes those definitions whenever repository events occur. Together, they connect engineering intent to reliable delivery.

YAML CONFIGURATION + GITHUB ACTIONS = CI/CD AUTOMATION

Readable Intent | **Event-Driven Execution** | **Reliable Software Delivery**

YAML and GitHub Actions together turn engineering intent into reliable delivery.

1. What Is YAML?

YAML (YAML Ain't Markup Language) is a human-readable format used to represent structured data, most often for configuration. Unlike XML or verbose JSON payloads, YAML keeps syntax light and relies on indentation to show hierarchy. That makes it easier for engineers to read quickly, review in pull requests, and maintain over time.

Files typically use the `.yml` or `.yaml` extension. In cloud and DevOps workflows, these files act as executable specifications stored in Git and interpreted by automation platforms.

Readable	Structured	Portable	Versionable	Universal
Reads close to natural language with minimal punctuation	Hierarchy defined by indentation, not brackets or braces	Platform-independent, works across all OS and cloud providers	Stored in Git, reviewable, traceable, and auditable	Used by Kubernetes, Docker, GitHub Actions, Azure Pipelines

Five characteristics that make YAML the standard language of cloud automation.

2. YAML Syntax Fundamentals

Most production YAML files are built from three patterns: **key-value pairs**, **lists**, and **nested objects**. Once these patterns are clear, engineers can work confidently across CI/CD pipelines, container tooling, and platform configuration files.

Key-Value Pairs

```
name:    AzureApp
version: 1.0
env:     production
```

Lists

```
services:
  - frontend
  - backend
  - database
```

Nested Objects

```
server:
  host: localhost
  port: 8080
```

Data Types

```
name:    "Sam"          # string
age:     52             # number
active:  true           # boolean
manager: null           # null
notes: |
  Multi-line string
  value supported
```

Rules in Action

```
# This is a comment
app:
  name: myapp         # spaces only, never ta
  version: 1.0        # indentation = hierarc
  services:
    - api             # hyphen = list item
    - web
```

YAML vs JSON: Choosing the Right Tool

Dimension	YAML	JSON
Readability	Clean and minimal, no brackets or commas	Structured, but visually heavier for human review
Comments	Supported with `#`	Not supported
Best Use	Human-authored configuration files	Machine-to-machine API payloads
Cloud Tooling	Kubernetes, GitHub Actions, Azure Pipelines	REST APIs, SDKs, programmatic output

Verdict: YAML for human-written configs | JSON for machine-to-machine APIs.

3. YAML in Modern Cloud Engineering

YAML is valuable not only because it is readable, but because the skill transfers across platforms. The same syntax appears in **Kubernetes** manifests, **Docker Compose** files, **GitHub Actions** workflows, and **Azure Pipelines** definitions—giving teams a practical *learn-once, apply-everywhere* advantage.

Platform	What YAML Defines	Key Benefit
Kubernetes	Deployments, Services, ConfigMaps, Secrets, Ingress	Declare desired cluster state as code
Docker Compose	Multi-container apps, networks, volumes	Reproducible local and CI environments
GitHub Actions	Workflow triggers, jobs, steps, release logic	CI/CD automation native to repository
Azure Pipelines	Build and release pipeline definitions	Enterprise-grade delivery on Azure DevOps

> **Key Message:** YAML is the common language of cloud automation. Engineers fluent in YAML can move smoothly between application configuration, infrastructure provisioning, and pipeline engineering.

4. Introduction to GitHub Actions

GitHub Actions is GitHub's built-in automation platform for CI/CD and repository-level operations. It reacts to events such as pushes, pull requests, schedules, and manual triggers, then runs workflows on managed runners.

Workflows are stored in `.github/workflows/` as YAML files, which keeps delivery logic version-controlled and visible alongside application code. This tight integration improves traceability and simplifies team collaboration.

Build	Test	Deploy	Security	Release
Compile and package code automatically on every commit	Run unit and integration tests before any merge	Push artifacts to Azure, AWS, or any cloud target	Scan dependencies and detect credential leaks	Automate versioned, documented, traceable releases

5. Anatomy of a GitHub Actions Workflow

A workflow combines **triggers**, **jobs**, **steps**, and **runners** into one automated sequence. A simple factory analogy helps: `on` is the entry sensor, jobs are departments, steps are tasks on each station, and runners are temporary workers assigned for one shift.

```
.github/workflows/build.yml

name: CI/CD Pipeline          # display name in Ac
                              # name UI
on:                           # WHEN to run
  push:                       # on
    branches: ["main"]
  pull_request:               # jobs
    branches: ["main"]        # runs-on
  workflow_dispatch:          # steps
                              # uses
jobs:                         # run
  build:                      # needs
    runs-on: ubuntu-latest    # ephemeral Ubuntu
    steps:                    # secrets
      - uses: actions/checkout@v4    # reusable ac
      - name: Build Application      # if
        run: npm run build           # shell com
```

Component	Role / Analogy
`name` UI	Factory recipe label
`on`	Motion sensor at gate
`jobs`	Departments in factory
`runs-on`	Temp worker per shift
`steps`	Assembly line tasks
`uses`	Pre-built supplier tool
`run`	Direct shell command
`needs`	Dependency between depts
`secrets`	Locked safe for credentials
`if`	Quality gate, stop/go condition

WORKFLOW LIFECYCLE: FROM CODE PUSH TO STATUS REPORT

Each stage acts as a quality gate — if one fails, downstream execution stops.

6. Hands-On Demo: Production CI/CD Pipeline Walkthrough

The live demonstration used a production-style pipeline for **GoCart**, a Next.js e-commerce application. Its structure reflects practical enterprise needs: controlled triggers, fail-fast quality checks, security visibility, containerization, and auditable releases.

FULL PIPELINE ARCHITECTURE: GOCART APPLICATION

Seven sequential quality gates — each job declares `needs` on the previous one.

Triggers and Concurrency Control

The workflow listens to `push` and `pull_request` events on main/master, and includes `workflow_dispatch` for manual runs. Concurrency settings prevent duplicate branch runs by canceling outdated executions. On protected branches, teams often keep in-progress runs intact to avoid partial deployment states.

Typical use of workflow_dispatch: hotfix redeployments, reruns for a specific commit, and operator-controlled release execution.

Global Environment Variables

```
env:
  NODE_VERSION: '20'                 # defined once and reused by all jobs
  DOCKER_IMAGE_NAME: gocart          # consistent image naming across stages
  NEXT_PUBLIC_CURRENCY_SYMBOL: '$'   # region-configurable app-level setting
```

Centralized variables reduce repetition and lower the risk of drift. For example, changing the Node runtime in one place updates every job that depends on it.

Build Stage: Reproducibility and Artifact Handoff

```
- uses: actions/checkout@v4
- uses: actions/setup-node@v4
  with:
    node-version: ${{ env.NODE_VERSION }}
    cache: npm                         # avoids re-downloading unchanged packages
- run: npm ci                          # exact lock-file install for reproducibility
- run: npm run build                   # compile Next.js and generate .next output
- uses: actions/upload-artifact@v4
  with:
    name: build-output
    path: .next/
    retention-days: 1                  # short-lived handoff between jobs
```

npm ci installs exactly what the lock file defines, which keeps builds deterministic across environments. Since jobs run on fresh runners, artifacts are used to hand off build output to later stages.

Lint and Test Stages: Quality Gates with Diagnostics

The lint stage enforces coding standards and catches static issues early. The test stage runs the automated suite with verbose reporting. Both stages publish logs using `if: always()`, so failure data is retained for troubleshooting and audit trails.

Security Stage: Visibility-First Governance

The security stage usually combines dependency auditing and secret-pattern detection. Dependency checks identify known CVEs; secret scanning looks for leaked credentials such as API keys and passwords. Together, these checks improve release confidence without relying on manual inspection.

In many enterprise teams, findings are reported and retained for compliance review, while remediation is prioritized based on severity and business impact.

Docker Build and Conditional Push: Governance in YAML

```
# Docker Build validates the image, but does not push on pull requests
- uses: docker/build-push-action@v6
  with:
    push: false                       # build-only on PR branches
    tags: gocart:test
    cache-from: type=gha              # layer caching shortens repeated builds
    cache-to: type=gha,mode=max

# Docker Push runs only for approved execution paths
if: github.event_name == 'push' || github.event_name == 'workflow_dispatch'
```

This condition enforces a critical policy: pull requests validate code, but do not publish release images. Credentials are injected through encrypted repository secrets and are never stored in plain text in workflow files.

Release Stage: Automated Versioning and Traceability

For successful main-branch runs, release automation can generate semantic tags and publish GitHub Releases with generated notes. This gives teams clean version history, commit-level traceability, and faster rollback capability.

7. Benefits, Best Practices, and Conclusion

Real-World Benefits

Benefit	What It Means in Practice
Speed	Manual hours compressed to automated minutes
Consistency	Every commit passes the same quality gates with no exceptions
Confidence	Failures caught before customers see them
Compliance	Artifact logs, scan reports, and release records built-in
Cost	Caching, timeouts, and concurrency minimize runner spend

DevOps high performers deploy significantly more often and recover faster — workflow automation is a major reason this performance gap exists.

Best Practices

- **Spaces only:** never use tabs in YAML files.
- **Centralize variables:** use `env` blocks for shared values.
- **Use secrets correctly:** keep credentials in repository secrets, never in YAML.
- **Capture failures:** use `if: always()` for logs and artifacts.
- **Gate deployments:** separate pull request validation from release jobs.
- **Pin action versions:** prevent surprise changes from upstream updates.
- **Fail fast:** use `needs` to stop early on quality failures.
- **Validate locally:** lint YAML before push to reduce failed runs.

Conclusion

> **Conclusion:** YAML provides the **structure**; GitHub Actions provides the **execution**. Together, they turn DevOps principles into repeatable daily practice. Teams gain faster feedback, clearer governance, and more reliable releases without increasing manual overhead.

YAML = Foundation | GitHub Actions = Automation Engine | DevOps = Culture of Continuous Delivery

DevOps with Microsoft Azure

DevOps with Microsoft Azure

Understanding Modern Software Delivery and Cloud Operations

By Dheeraj S Bhat

> Modern organizations must ship features faster, with higher quality and greater reliability than ever. The old model—developers writing code and “throwing it over the wall” to a separate operations team—cannot keep pace. **DevOps** answers this: a cultural and technical movement that unifies software **Development** and IT **Operations** through collaboration, automation, and continuous delivery.

DEVELOPMENT + OPERATIONS = DEVOPS

Faster Delivery | **Higher Quality** | **Greater Reliability**

DevOps unifies Development and Operations into one continuous, value-driven flow.

1. What Is DevOps?

DevOps is not a single tool or product. It is a combination of cultural philosophies, practices, and tools that increases an organization’s ability to deliver applications and services at high velocity, shortening the development life cycle while delivering fixes and updates frequently and reliably. The payoff is **faster delivery, higher quality, and greater reliability**.

- **Collaboration** — breaking down the silos between dev and ops teams.
- **Automation** — eliminating manual, repetitive, error-prone processes.
- **Continuous Delivery** — producing frequent, reliable software releases.
- **Feedback Loops** — enabling rapid learning and improvement cycles.
- **Shared Ownership** — collective responsibility for success in production.

DevOps directly attacks the pain points of traditional development—slow release cycles, manual deployments, communication gaps, frequent outages, and inability to adapt—through continuous delivery, automated testing, shared ownership, and rapid feedback.

Dimension	Traditional	DevOps

Team Structure	Siloed teams working independently	Cross-functional collaboration
Deployment Speed	Monthly or quarterly releases	Multiple deployments daily
Reliability	Unpredictable, high failure rate	Consistent, low failure rate
Automation	Mostly manual processes	Fully automated pipelines
Feedback Loop	Slow, delayed feedback	Real-time monitoring
Culture	Blame-oriented, finger-pointing	Shared ownership, trust

2. Core Principles: The CALMS Framework

DevOps maturity is measured against five interdependent pillars:

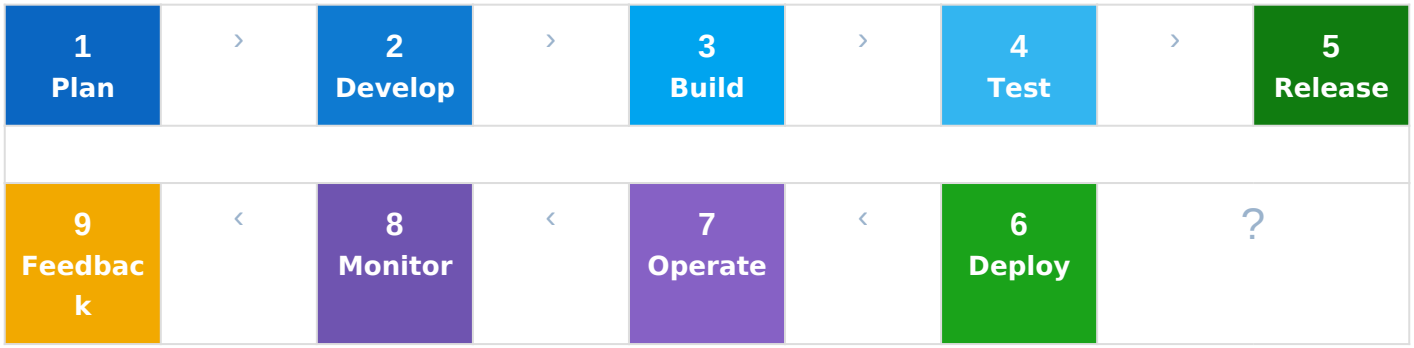
C	A	L	M	S
Culture	**Automation**	**Lean**	**Measurement**	**Sharing**
Break down silos, build trust, foster collaboration	Eliminate manual processes, cut errors, add speed	Remove waste, optimize flow, deliver value efficiently	Track metrics, drive data-informed decisions	Create feedback loops, spread knowledge widely

The five CALMS pillars of DevOps maturity.

These principles work together: culture is the foundation, automation enables speed, lean removes waste, measurement guides decisions, and sharing accelerates learning.

3. The DevOps Lifecycle

DevOps is best visualized as a continuous loop—an unending cycle of delivery and improvement, where feedback from the final stage drives the next planning cycle.



The nine-stage DevOps lifecycle — a continuous loop where feedback feeds the next cycle.

1. Plan

Grounded in **Agile**—iterative development with customer collaboration. Work is expressed as **user stories**, organized into time-boxed **sprints** (1–4 weeks), and refined through **backlog management**. *Azure Boards* provides Kanban boards, backlogs, and sprints.

2. Develop

Relies on **Git** distributed version control with branching strategies (feature branches, GitFlow, trunk-based) and quality enforced via **code reviews** and **pull requests**. *Azure Repos* offers unlimited private Git repos with branch policies.

3. Build

Transforms source into deployable software via compilation, packaging, and artifact generation. Tools vary by ecosystem: Maven/Gradle (Java), npm/webpack (JS), MSBuild/dotnet (.NET), Docker Build (containers).

4. Test

Unit tests validate components in isolation; **integration** tests verify interactions; **end-to-end** tests validate full workflows. These feed **quality gates**: coverage threshold, security scan, all tests passing, and a performance baseline.

5. Release

Prepares a tested artifact for production by versioning it and staging it behind **approval gates**. Strategies such as **blue-green** and **canary** releases reduce risk, while *Azure Pipelines* coordinates multi-stage, auditable releases.

6. Deploy

Pushes the release into the target environment—ideally automated and repeatable so every deployment is identical. **Rolling updates** and instant **rollback** keep deployments safe, with *Azure Pipelines* deploying to any cloud, on-premises host, or *AKS* cluster.

7. Operate

Keeps the live system healthy: managing infrastructure, scaling for demand, applying patches, and handling incidents. **Infrastructure as Code** (Terraform, Bicep) ensures consistent, drift-free environments.

8. Monitor

Observes the running application—collecting metrics, logs, and traces to surface bottlenecks and failures. *Azure Monitor*, *Application Insights*, and *Log Analytics* (KQL) provide visibility with proactive alerts and dashboards.

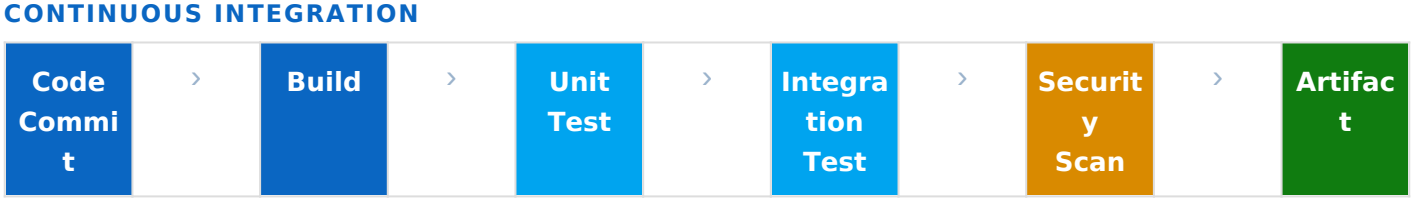
9. Feedback

Turns production insight into action: usage data, performance trends, and user input are analyzed to learn and improve. This feedback flows straight back into **Plan**, closing the loop and driving the next iteration of continuous improvement.

4. Continuous Integration & Continuous Delivery (CI/CD)

Continuous Integration (CI) automatically builds and tests code on every commit, delivering early bug detection, faster feedback, and consistent builds. *Azure Pipelines* provides cloud-hosted agents for any language on Windows, Linux, and macOS.

Although grouped as “CD,” two practices differ: **Continuous Delivery** keeps code always deployable with a **manual approval gate** before production, while **Continuous Deployment** automatically deploys every passing change with no human intervention.



Each stage is a quality gate — any failure stops the pipeline.

5. The Azure DevOps Platform

Azure DevOps is Microsoft’s comprehensive, end-to-end platform supporting any language and platform, cloud or on-premises deployment, deep Azure integration, and built-in enterprise security. It is organized into five core services:

Azure Boards

Agile planning, Kanban boards, backlogs, sprint planning, and work dashboards.

Azure Repos

Unlimited private Git repositories with branch policies, pull requests, and search.

Azure Pipelines

CI/CD for any platform with cloud-hosted agents, multi-stage deploys, and approvals.

Azure Test Plans

Manual testing, exploratory testing, and test case management for quality assurance.

Azure Artifacts

Package management for Maven, npm, NuGet, and Python with upstream sources and retention policies.

6. Infrastructure, Containers, and Orchestration

Infrastructure as Code (IaC)

IaC manages infrastructure through machine-readable code rather than manual processes, providing version control, consistent environments, repeatable deployments, and reduced drift. Azure tools include **Terraform** (multi-cloud, HCL), **Bicep** (Azure-native DSL), **ARM Templates** (JSON), and **Azure Blueprints** (governed environments).

Containerization with Docker

Containers package application code with all dependencies for portable, consistent deployment. Docker is lightweight versus VMs, starts quickly, and simplifies deployment: *Dockerfile* → *image* → *registry* → running *container*. *Azure Container Registry (ACR)* integrates natively with Azure DevOps and AKS.

Kubernetes & Azure Kubernetes Service (AKS)

Kubernetes automates deployment, scaling, and management of containers with auto-scaling, self-healing, load balancing, and zero-downtime rolling updates. *AKS* provides managed Kubernetes with Azure AD integration, Container Insights, auto-scaling, and Azure DevOps integration.

7. Monitoring, Observability, and Security

Observability

- **Azure Monitor** — unified metrics and logs from all Azure resources.
- **Application Insights** — APM with distributed tracing and diagnosis.
- **Log Analytics** — querying logs with KQL.
- **Alerts & Dashboards** — proactive notifications and health views.

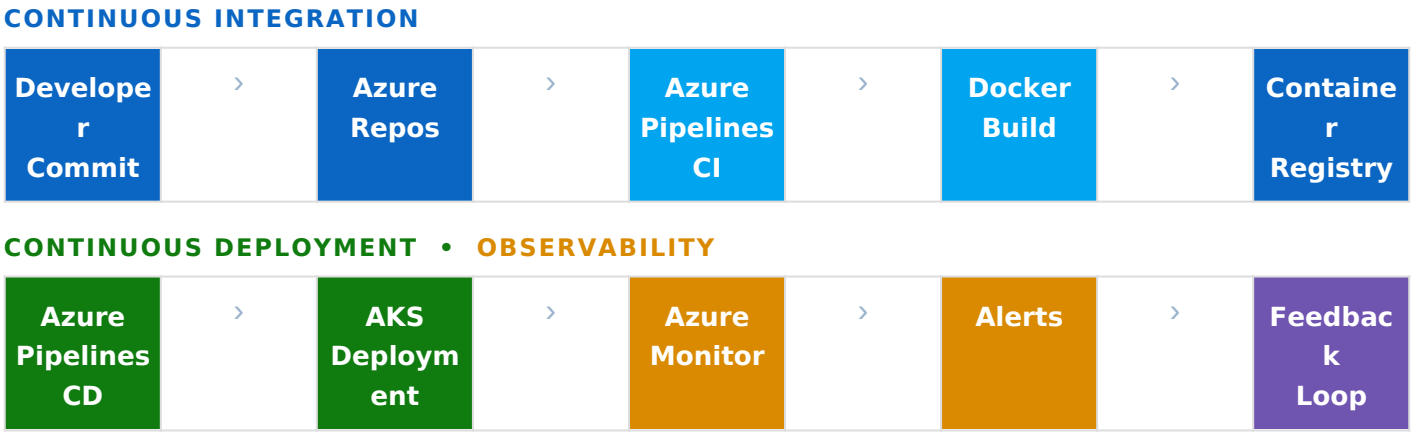
Observability is the feedback loop for continuous improvement—without it, teams fly blind.

DevSecOps

Shift-left security: integrating security from the start is roughly **10× cheaper** than fixing issues in production.

- **Secure coding** with training and guidelines.
- **Vulnerability scanning** in CI/CD pipelines.
- **Azure Key Vault** for secrets and keys.
- **RBAC** with least privilege; **Azure Policy** for compliance.

8. A Real-World Azure DevOps Architecture



A complete cloud-native pipeline — every component is a natively integrated Azure service.

Every component is an Azure service that integrates natively with the others, eliminating the friction of stitching together disparate tools.

9. Benefits, Challenges, and Best Practices

Benefits

- Faster time-to-market (months ? hours)
- Improved quality via automated testing
- Better collaboration and visibility
- Increased reliability and cost efficiency
- Higher customer satisfaction

*DevOps organizations deploy **200× more frequently** and recover **24× faster** than lower performers.*

Challenges & Best Practices

- **Challenges:** cultural transformation, learning curve, security integration, governance.
- **Best practices:** start small, invest in training, automate everything, measure DORA metrics.

*The four **DORA metrics**—Deployment Frequency, Lead Time for Changes, Change Failure Rate, and Time to Recovery—provide a research-backed way to measure performance.*

Conclusion & the Future of DevOps

DevOps is ultimately a **culture, not just a set of tools**. Microsoft Azure complements that culture with a complete platform: CI/CD pipelines automate delivery, IaC enables consistent environments, and monitoring and security are woven throughout.

Looking ahead, four trends shape the next chapter: **AI-assisted DevOps (AIOps)**, **GitOps** for Kubernetes-native deployments, **Platform Engineering**, and **FinOps** for cloud cost optimization. Organizations that embrace these practices—anchored by a collaborative culture and powered by Azure’s integrated toolset—position themselves to deliver software faster, more reliably, and more securely than ever.

Azure Services: A Tech Team Quick Reference Page

Azure Services: A Tech Team Quick Reference

A condensed, decision-focused guide to the major Azure service categories

By Swedel F Menezes - Cloud Team

> A condensed, decision-focused guide to the major Azure service categories. For each service: what it is, primary use cases, and a one-line “when to choose it” rule. Use this as a quick reference when designing or reviewing an Azure architecture.

Azure Reference Architecture — Layer Overview

Layer	Service(s)	Role
Edge / Ingress	Azure Front Door	Global HTTP load balancing, CDN, WAF & intelligent routing
Web / API Compute	App Service	Managed PaaS for web apps, REST APIs and mobile backends
Container Compute	AKS (Kubernetes)	Orchestrated containers, microservices, auto-scaling
Serverless Compute	Azure Functions	Event-driven, short-lived tasks triggered by HTTP/queue/blob
Relational Data	Azure SQL Database	Managed SQL Server — structured OLTP workloads

NoSQL / Global Data	Cosmos DB	Multi-model, globally distributed, <10 ms latency
Object Storage	Blob Storage	Unstructured data — media, backups, analytics staging
Caching	Azure Cache for Redis	Sub-millisecond in-memory caching & session state
Identity	Microsoft Entra ID	SSO, MFA, app registrations & managed identities
Secrets	Key Vault	Centralised store for secrets, keys & TLS certificates
Observability	Azure Monitor + App Insights	Full-stack metrics, logs, alerts & distributed tracing
Security Posture	Defender for Cloud	CSPM scoring, threat detection & compliance dashboards

1. Compute

Service	What it is & key use cases	When to choose it
Virtual Machines (VMs)	IaaS with full OS control. Lift-and-shift, legacy apps, custom OS, dev/test.	You need full OS control or have compliance/legacy needs that block PaaS.
App Service	Managed PaaS for web apps & REST APIs. ASP.NET/Node/Python/Java, auto-scale, CI/CD.	You want to focus on code, not infra, and don’t need container orchestration.
Azure Kubernetes Service (AKS)	Managed Kubernetes for containers. Microservices, auto-scaling, self-healing.	You run multiple containers needing orchestration & service discovery.
Container Instances (ACI)	Serverless single containers. Batch jobs, CI tasks, quick tests.	Simple isolated container tasks; use AKS if you need orchestration.
Azure Functions	Event-driven serverless compute. HTTP APIs, timers, queue/blob events.	Short-lived, event-triggered work. Avoid for long-running (>10 min) jobs.

2. Storage

Service	What it is & key use cases	When to choose it
Blob Storage	Object storage for unstructured data. Static sites, media, backups, analytics staging.	Any binary/unstructured data. Hot/Cool/Archive tiers by access frequency.
Azure Files	Managed SMB/NFS file shares. Replace on-prem file servers, shared config.	Apps that need a shared file system; use Blob for object storage.
Data Lake Storage Gen2	Blob + hierarchical namespace for big data. ML data, ETL, Synapse/Databricks.	Big-data workloads needing directory-level ACLs and hierarchy.

3. Networking

Service	What it is & key use cases	When to choose it
Virtual Network (VNet)	Isolated private network — foundation of secure deployments.	Always for production. Never expose resources without NSG rules.
Load Balancer	Layer 4 (TCP/UDP) balancing across VMs. HA, inbound NAT, internal LB.	Non-HTTP VM workloads needing HA; use App Gateway for HTTP.
Application Gateway	Layer 7 (HTTP/S) LB with WAF, SSL termination, URL routing.	HTTP/S apps needing WAF, SSL offload, or path-based routing (regional).
Front Door	Global HTTP LB with CDN, WAF, intelligent routing.	Global apps needing low latency worldwide + edge CDN/failover.
VPN Gateway	Site-to-site / point-to-site VPN to on-prem. Hybrid cloud, remote access.	Encrypted hybrid connectivity; use ExpressRoute for dedicated bandwidth.

4. Databases

Service	What it is & key use cases	When to choose it
Azure SQL Database	Managed relational PaaS (SQL Server engine). Web/enterprise OLTP, migrations.	Structured relational data. Elastic Pool for many DBs, MI for full compat.
Cosmos DB	Globally distributed multi-model NoSQL. IoT, catalogs, gaming, multi-region writes.	You need <10ms global latency, flexible schema, or active-active replication.
Azure Cache for Redis	Managed in-memory cache. Session state, query caching, leaderboards, pub/sub.	App has repetitive expensive queries or needs sub-millisecond responses.
Synapse Analytics	Unified data warehouse + big data analytics. ETL/ELT, BI, Power BI/ML.	Large-scale analytical workloads; use Azure SQL for operational OLTP.

5. AI & Machine Learning

Service	What it is & key use cases	When to choose it
Azure Machine Learning	End-to-end ML platform. Custom models, AutoML, MLOps, monitoring/retraining.	Building custom models; use AI Services for pre-built capabilities.
Azure AI Services	Pre-built AI APIs — Vision, Speech, Language, Decision. OCR, sentiment, STT/TTS.	You need AI features fast via REST without training custom models.
Azure OpenAI Service	OpenAI models (GPT, DALL-E, Whisper, embeddings) in Azure. Chatbots, summarization, code, semantic search.	You need enterprise security, private networking, compliance & data residency.

6. DevOps & Monitoring

Service	What it is & key use cases	When to choose it

Azure DevOps	Boards, Repos, Pipelines, Test Plans, Artifacts. CI/CD, agile, source control.	End-to-end DevOps lifecycle integrated with the Azure ecosystem.
Key Vault	Secure store for secrets, keys, certificates. Connection strings, TLS certs, CMK.	Always — never hardcode credentials. Access via managed identity.
Monitor & App Insights	Full-stack observability — metrics, logs, alerts, APM, distributed tracing.	Enable on every app/Function from day one; retro-fitting is harder.

7. Security & Identity

Service	What it is & key use cases	When to choose it
Microsoft Entra ID	Cloud identity & access (IAM). SSO, MFA, app registrations, managed identities.	Always for authentication; use managed identities for service-to-service auth.
Defender for Cloud	CSPM + workload protection. Posture scoring, threat detection, compliance.	Enable on all production subscriptions for a unified security view.
Microsoft Sentinel	Cloud-native SIEM + SOAR. Event aggregation, AI analytics, automated response.	Centralized security monitoring across Azure + on-prem + multi-cloud.

8. Integration & Messaging

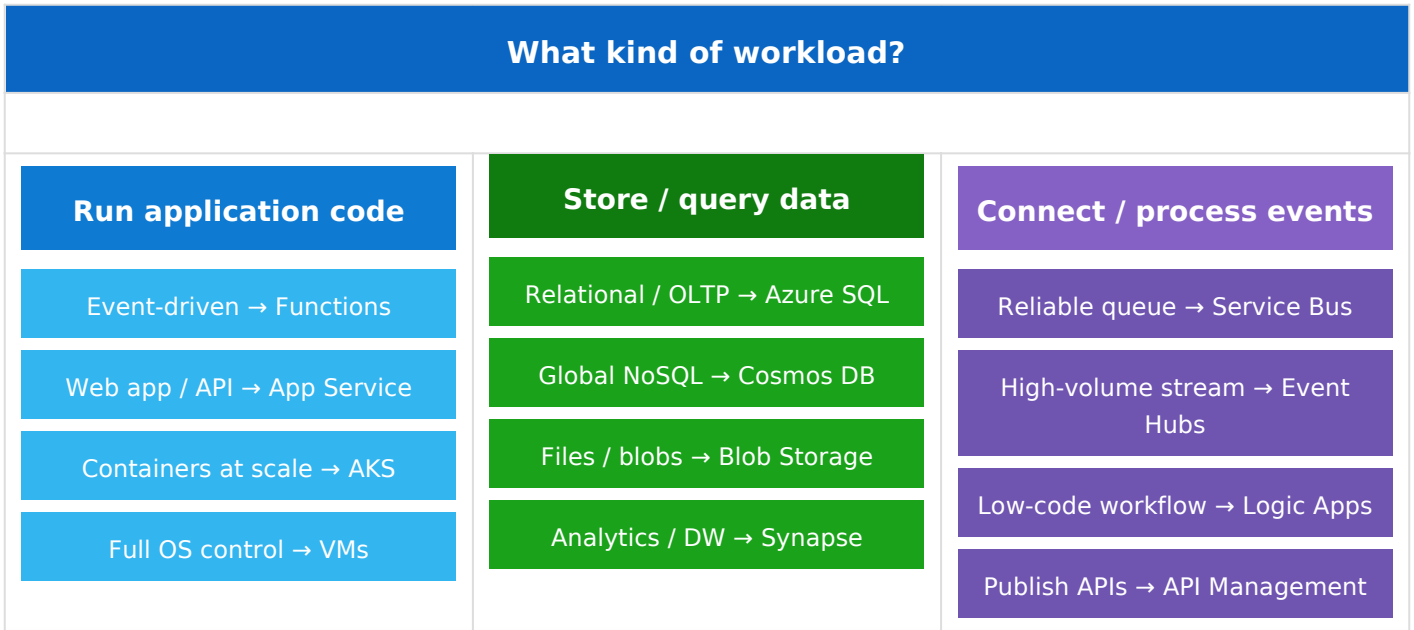
Service	What it is & key use cases	When to choose it
Service Bus	Enterprise broker — queues & topics (pub/sub). Decoupling, dead-lettering, ordering.	Reliable, ordered, transactional messaging; use Event Hubs for streaming.
Event Hubs	Big-data event streaming (millions/sec). IoT telemetry, logs, click-streams.	High-volume event ingestion feeding analytics pipelines.

Logic Apps	Low-code workflow automation, 400+ connectors. B2B, approvals, SaaS integration.	Integration workflows across SaaS/enterprise; use Functions for custom code.
API Management (APIM)	Full-lifecycle API gateway. Publish, secure, throttle, version, dev portal.	Exposing APIs externally or across teams with governance & observability.

9. Analytics

Service	What it is & key use cases	When to choose it
Azure Databricks	Spark-based analytics platform. Large ETL, ML at scale, streaming, lakehouse.	Complex big-data processing with Spark; integrates with ADLS Gen2 & Synapse.
Data Factory (ADF)	Cloud-scale ETL/ELT integration. Pipeline orchestration, 90+ connectors.	The orchestration layer of your data platform; moving data on-prem ↔ cloud.

Service Selection Decision Tree



Always add: Entra ID (identity) • Key Vault (secrets) • Monitor + App Insights (observability)

Figure 2 — Decision tree mapping a workload type to the recommended Azure service.

Service Selection Quick Reference

Need	Service
Host a web app	App Service
Run containers at scale	AKS
Simple container task	ACI
Serverless function	Azure Functions
Relational DB	Azure SQL Database
NoSQL / Global DB	Cosmos DB
Cache	Azure Cache for Redis
Data warehouse	Synapse Analytics
Big data processing	Databricks
ETL orchestration	Data Factory
Store files/blobs	Blob Storage
Shared file system	Azure Files
Pre-built AI APIs	Azure AI Services
Custom ML models	Azure ML
LLM / GPT	Azure OpenAI
Message queue	Service Bus
Event streaming	Event Hubs
API gateway	API Management
Secrets management	Key Vault
Identity / SSO	Microsoft Entra ID
Security posture	Defender for Cloud
SIEM	Microsoft Sentinel
Monitoring / APM	Azure Monitor + App Insights

Introduction to Azure Service Bus

Azure Service Bus: Reliable Messaging for Modern Cloud Applications

A Practical Guide to Decoupled, Resilient, and Scalable Cloud Communication

By Kenneth Gavin Dcosta • Cloud Team - Buildr

> Modern applications are rarely built as one large system anymore. Instead, they are made up of many smaller services: order services, payment services, inventory systems, notification engines, shipping workflows, analytics pipelines, and more. This makes applications easier to scale and maintain, but it also introduces a new challenge: **how do these services communicate reliably without becoming dependent on each other?**

AZURE SERVICE BUS + CLOUD ARCHITECTURE = RELIABLE COMMUNICATION

Decoupled Services | **Asynchronous Processing** | **Reliable Delivery**

Azure Service Bus turns fragile direct communication into reliable, scalable, production-ready messaging.

1. Why Direct Service Communication Becomes a Problem

At first, direct communication between services feels simple.

For example, in an e-commerce application, the order flow may look like this:

```
Order Service → Payment Service → Inventory Service → Shipping Service → Notification Service
```

This works well when everything is healthy. But in real-world systems, services fail, slow down, restart, or experience sudden traffic spikes. If one service in the chain goes down, the entire workflow can be affected.

Imagine the Shipping Service is unavailable. The Order Service may still be working, the Payment Service may still be working, and Inventory may still be available — but because the flow is tightly

connected, the overall order process may fail.

This is known as **tight coupling**.

Problem	What Happens in Practice
Service failure	One service failure can impact other services.
Peak traffic	Every service may need to scale at the same time.
Maintenance	Teams may need coordinated downtime.
New features	Adding a new service often requires modifying existing services.
Slow dependency	Slow services create delays across the entire workflow.

For small systems, this may be manageable. For modern cloud applications, it quickly becomes risky.

2. What Azure Service Bus Solves

Azure Service Bus solves this problem by introducing **asynchronous messaging**.

Instead of one service directly calling another, the sender places a message into Service Bus. The receiving service then picks up and processes that message independently.

```
Sender Application → Azure Service Bus → Receiver Application
```

The sender does not need to know whether the receiver is online. The receiver does not need to process the message immediately. Service Bus safely stores the message until it can be handled.

This gives applications breathing room.

- If the receiver is temporarily down, messages wait.
- If traffic increases suddenly, Service Bus absorbs the load.
- If one downstream service fails, other services can continue working.

> **Core value:** Azure Service Bus separates services so they can operate independently without losing messages.

3. A Simple Analogy: The Post Office

The easiest way to understand Azure Service Bus is to compare it to a post office.

When you send a letter, you do not personally deliver it to the recipient. You do not need to know the mail carrier, the route, or the exact delivery time. You simply drop the letter into the postal system.

The post office stores, sorts, and delivers the letter. The recipient collects it when available.

Post Office	Azure Service Bus
You drop a letter	Sender sends a message
Post office stores it	Service Bus stores it reliably
Mail carrier delivers it	Receiver processes it
Recipient collects later	Consumer processes when ready
Sender and receiver do not meet	Services remain decoupled

Service Bus acts as an intermediary that enables reliable, asynchronous communication between applications.

4. Core Components of Azure Service Bus

Azure Service Bus is built around a few key components. Understanding these makes the rest of the service much easier.

Component	Description	Simple Analogy
Namespace	Top-level container for messaging resources	Post office building
Queue	One-to-one message processing	Single bank line
Topic	One-to-many message publishing	Newspaper publisher
Subscription	Consumer-specific copy or filtered view of topic messages	Newspaper subscriber
Message	Payload, properties, and metadata	Letter with envelope

Namespace

A namespace is the top-level container for Service Bus resources. It holds queues, topics, subscriptions, and related configuration.

```
gocart-servicebus-namespace

orders-queue
payments-queue
neworders-topic
shipping-subscription
notification-subscription
```

Queue

A queue is used for one-to-one message processing. One or more senders place messages into a queue, and each message is processed by one receiver.

```
Order Service → Orders Queue → Order Processor
```

Queues are useful for background jobs, order processing, invoice generation, email sending, and other tasks where each message should be handled once. This is also known as the **Competing Consumers pattern**.

Topic

A topic is used for one-to-many communication. One service publishes a message to a topic, and multiple subscribers can receive their own copy of that message.

```
Order Service → NewOrders Topic
                    ├── Inventory Subscription
                    ├── Payment Subscription
                    ├── Shipping Subscription
                    └── Notification Subscription
```

Subscription

A subscription belongs to a topic. Each subscription receives a copy of messages from the topic. Subscriptions can also include filters, so different consumers receive only the messages relevant to them.

Message

A message is the unit of data sent through Service Bus. It usually contains a body, properties, metadata, message ID, timestamp, and other information needed by the receiver.

```
{
  "orderId": "ORD-10291",
  "customerId": "CUST-7781",
  "amount": 2499,
  "currency": "INR",
  "eventType": "OrderPlaced"
}
```

The message body carries the business data, while metadata helps with tracking, filtering, correlation, and troubleshooting.

5. Queues vs Topics: Choosing the Right Pattern

Queues and topics are both messaging entities, but they solve different problems.

Use a **queue** when one service should process each message. Use a **topic** when multiple services need to receive the same message.

Requirement	Queue	Topic

One receiver processes the message	Yes	No
Multiple services need the same event	No	Yes
Background job processing	Yes	Sometimes
Event broadcasting	No	Yes
Simple work distribution	Yes	No
Microservice fan-out	No	Yes

> **Simple rule:** Queue = one task, one processor. Topic = one event, many listeners.

6. How Messages Are Processed

Azure Service Bus follows a reliable message lifecycle.

MESSAGE LIFECYCLE: FROM SEND TO RETRY

- A producer sends a message to a queue or topic.
- Service Bus stores the message reliably.
- A consumer receives the message.
- Service Bus locks the message so other consumers cannot process it at the same time.
- If processing succeeds, the consumer completes the message.
- If processing fails or the consumer crashes, the lock expires and the message becomes available again for retry.

Peek-Lock ensures that a message is not lost if a receiver fails during processing. This is one of the most important reliability features of Service Bus.

7. Dead-Letter Queue: Handling Messages That Cannot Be Processed

In real systems, not every message can be processed successfully.

A message may fail because:

- Required data is missing.
- The format is invalid.

- A business rule fails.
- A downstream service is unavailable.
- The consumer has a bug.

If the same message keeps failing, it should not block the entire queue. Azure Service Bus handles this using a **Dead-Letter Queue**, commonly called a **DLQ**.

After the maximum retry count is reached, Service Bus moves the failed message to the DLQ. Developers or operations teams can then inspect it, understand why it failed, fix the issue, and decide whether to resubmit or discard the message.

> **Best practice:** Treat the DLQ as a problem mailbox. A growing DLQ usually indicates a code issue, schema mismatch, missing configuration, or dependency failure.

8. Enterprise Features That Make Service Bus Production-Ready

Azure Service Bus includes several features that are especially useful in enterprise systems.

Feature	Why It Matters
Duplicate Detection	Prevents the same message from being processed multiple times when senders retry.
Sessions	Groups related messages so they are processed in order by the same receiver instance.
Time-to-Live	Automatically expires messages that are no longer useful after a certain period.
Scheduled Messages	Allows an application to send a message now but deliver it later.
Transactions	Allows multiple Service Bus operations to succeed or fail together.
Auto-Forwarding	Moves messages automatically from one queue or subscription to another for advanced routing.

These features make Azure Service Bus more than a simple queue. It is designed for real production workloads where reliability, ordering, retries, and operational control matter.

9. Security and Monitoring

Security is a critical part of any messaging system because messages often carry business-sensitive data.

Authentication

- **Microsoft Entra ID**
- **Managed Identity**
- **Shared Access Signature tokens**

Managed Identity is often preferred because applications can authenticate without storing passwords or connection strings in code.

Monitoring

- Active message count
- Dead-letter message count
- Incoming messages
- Outgoing messages
- Queue depth
- Processing errors

Service Bus also supports encryption at rest and encryption in transit using TLS. Premium tier scenarios can also use customer-managed keys.

> **Operational rule:** If queue depth keeps increasing, consumers are not keeping up. That may mean you need more consumers, faster processing, better scaling, or investigation into downstream failures.

10. Azure Service Bus vs Other Azure Messaging Services

Azure provides multiple messaging and eventing services. Each has a different purpose.

Service	Best Use Case
Azure Service Bus	Enterprise messaging, reliable workflows, ordering, transactions
Azure Storage Queues	Simple task queues
Azure Event Grid	Event routing and reactive automation
Azure Event Hubs	High-volume telemetry and streaming

In real architectures, these services can also work together. For example, Event Grid may trigger a process, Event Hubs may ingest telemetry, and Service Bus may coordinate business workflows.

11. Real-World Example: E-Commerce Order Flow

Let us revisit the e-commerce example.

Instead of directly calling every service, the Order Service publishes one event to a topic:

```
Customer places order
        ↓
Order Service
        ↓
NewOrders Topic
        ↓
```

```
        ├── Inventory Subscription
        ├── Payment Subscription
        ├── Shipping Subscription
        └── Notification Subscription
```

Each service receives its own copy of the message and processes it independently.

- The Inventory Service reserves stock.
- The Payment Service processes payment.
- The Shipping Service prepares a label.
- The Notification Service sends an email.

If the Notification Service fails, payment and inventory can still continue. If the Payment Service is slow, shipping and notification are not necessarily blocked. Failed messages can go to the DLQ for review.

> **Key message:** One business event can safely trigger multiple independent workflows. If the business later wants fraud detection or analytics, a new subscription can be added without rewriting the Order Service.

12. Best Practices for Production Use

Azure Service Bus is powerful, but like any messaging technology, it should be used carefully.

Practice	Why It Matters
Start simple	Begin with queues, then move to topics when multiple services need the same event.
Prefer topics for business events	Topics reduce direct dependencies between microservices.
Monitor the DLQ	Failed messages should be inspected, fixed, resubmitted, or discarded intentionally.
Set lock duration carefully	A short lock duration can cause duplicate processing.
Design consumers to be idempotent	Processing the same message twice should not create incorrect results.
Use duplicate detection when needed	Useful when sender retries may produce duplicate messages.
Use sessions only when ordering is required	Sessions are powerful but add complexity.
Use Managed Identity	Avoid storing connection strings in code or configuration files.

Alert on queue depth	A growing queue usually means producers are sending faster than consumers can process.
Do not over-engineer early	Add sessions, transactions, filters, or forwarding only when the system actually needs them.

Conclusion

Azure Service Bus plays a vital role in modern cloud architecture. It helps applications communicate without being tightly connected to each other. By placing a reliable messaging layer between services, it improves resilience, scalability, and maintainability.

- Queues help distribute work to one processor.
- Topics allow one event to reach many independent subscribers.
- Dead-letter queues help isolate failed messages.
- Sessions, duplicate detection, TTL, scheduled delivery, and transactions support real enterprise scenarios.
- Security and monitoring features make the service suitable for production workloads.

A QUICK MENTAL MODEL

```
Azure Service Bus = Reliable Messaging Layer

Queues = One-to-One Work Processing
Topics = One-to-Many Event Distribution
DLQ = Failed Message Investigation
Managed Identity = Secure Authentication
Azure Monitor = Operational Visibility
```

> **The main lesson:** Azure Service Bus turns fragile direct communication into reliable, scalable, and production-ready messaging.

For teams building cloud-native applications, it is not just a messaging service. It is a foundation for building systems that can handle failure, scale with demand, and evolve without breaking everything around them.

Service Bus = Reliable Messaging | Queues = Work Distribution | Topics = Event Broadcasting

Smart Bank – AI Powered Banking Assistant

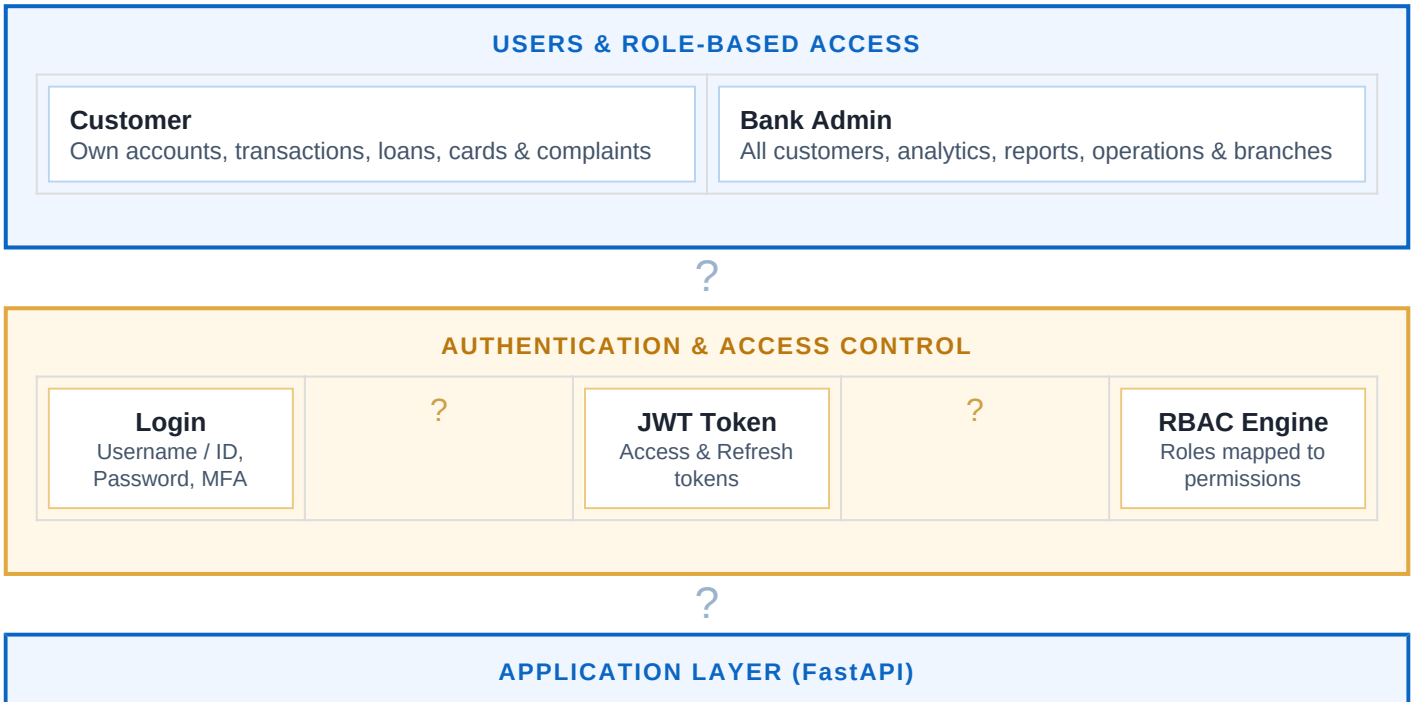
Smart Bank – AI Powered Banking Assistant

Role-Based Dashboards using Semantic Kernel, Azure OpenAI, MySQL & OpenTelemetry

A Reference Architecture for Intelligent, Secure, and Observable Digital Banking

> Banking customers now expect instant, conversational, and personalized service, while banks must keep every interaction secure, auditable, and compliant. **Smart Bank** answers both needs: an AI-powered banking assistant built on **role-based dashboards** for customers and administrators, orchestrated by **Semantic Kernel**, reasoning with **Azure OpenAI**, backed by a **MySQL** core data store, and observed end-to-end with **OpenTelemetry**.

Smart Bank: AI Powered Banking Assistant Architecture



Auth Service Login, MFA, tokens	**User Service** Profile, roles	**Account Service** Balances, summaries	**Transaction Service** Transactions, payments
Loan Service Loans, EMIs, dues	**Card Service** Cards, limits	**Complaint Service** Register, track, resolve	**Analytics Service** Reports, insights

Chat Assistant API: send / receive messages, maintain conversation session state

?

SEMANTIC KERNEL ORCHESTRATION LAYER

Intent Detection	**Prompt Management**	**Function Calling**
Context & Memory	**Plugin Invocation**	**Response Generation**

?

PLUGINS (BANKING CAPABILITIES)

Account, Transaction, Loan, Card, Complaint, Analytics, Customer & Payment. Typed operations mapped to application services.

KNOWLEDGE & SEARCH (RAG)

Azure AI Search (vector) over policy docs, FAQs, statements & guidelines. Grounds answers in the bank's own content.

AZURE OPENAI SERVICE

GPT-4o / GPT-4.1 with embeddings: chat completion, function calling & response generation.

?

MYSQL DATABASE (CORE DATA STORE)

users, roles, customers, accounts, transactions, loans, credit_cards, complaints, branches.

AZURE BLOB STORAGE

Statements, KYC files, loan agreements, policies & forms.

EXTERNAL INTEGRATIONS

Payment Gateway, SMS / Email, KYC / AML, Credit Bureau & Core Banking.

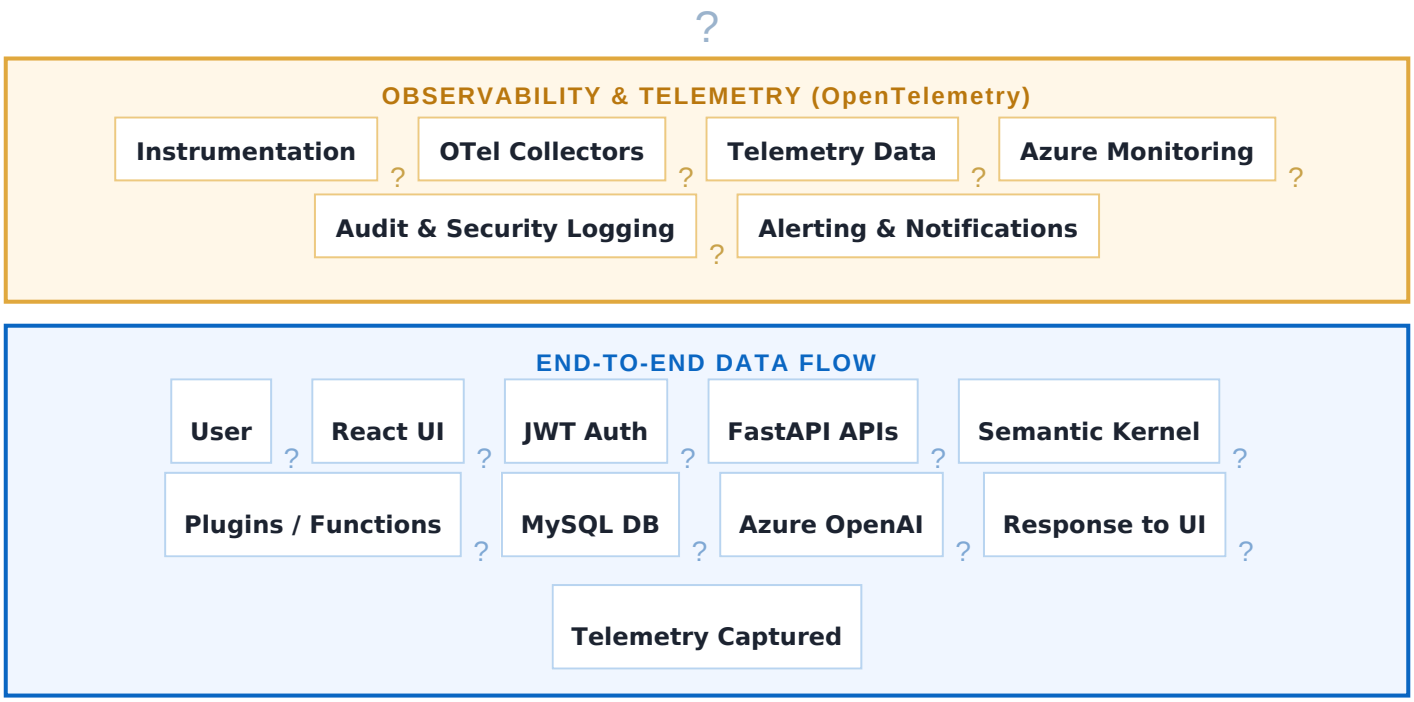


Figure 1 — The complete Smart Bank architecture, from role-based user access through the FastAPI application layer, Semantic Kernel orchestration, Azure OpenAI reasoning, MySQL persistence, and full-stack observability.

Smart Bank Architecture — Layer Overview

Layer	Component(s)	Role
Users / Access	Role-Based Dashboards	Separate Customer & Bank Admin experiences, enforced by RBAC
Authentication	JWT + MFA + RBAC Engine	Verify identity, issue tokens, map roles to permissions
Application Layer	FastAPI Services	Auth, User, Account, Transaction, Loan, Card, Complaint, Analytics, Chat APIs
Orchestration	Semantic Kernel	Intent, prompts, function calling, memory, response generation
AI Reasoning	Azure OpenAI (GPT-4o / 4.1)	Language understanding, function-calling decisions, replies

Knowledge	Azure AI Search (RAG)	Grounds answers in policy docs, FAQs, statements, guidelines
Plugins	Banking Capability Plugins	Typed banking operations mapped to application services
Core Data	MySQL Database	System of record for users, accounts, transactions, loans
Documents	Azure Blob Storage	Statements, KYC files, loan agreements, policies, forms
Integrations	External Services	Payment, SMS/Email, KYC/AML, Credit Bureau, Core Banking
Observability	OpenTelemetry + Azure Monitor	Traces, metrics, logs, alerts, audit & security logging

1. What Is Smart Bank?

Smart Bank is a reference architecture for an intelligent banking assistant that lets users converse naturally with their bank instead of navigating dozens of screens. It is not a single product but a composition of cloud-native services that turn natural-language requests like "show my last five transactions," "what is my EMI due date," or "raise a complaint" into safe, governed actions against real banking data.

Principle	What it means
Conversational	A chat assistant replaces complex navigation for everyday banking tasks.
Role-aware	Distinct experiences for Customers and Bank Admins, enforced by RBAC.
Grounded	Answers are based on the bank's own data and documents, not guesswork.
Secure	JWT authentication, MFA, and least-privilege permissions throughout.

Observable	OpenTelemetry traces, metrics, and logs feed Azure monitoring and alerting.

2. Users and Role-Based Access

Two primary roles drive the entire experience. The architecture deliberately keeps their capabilities separate so that a single platform can serve very different needs without compromising security.

Role	Scope of Access
Customer	Own accounts, transactions, loans, cards, and complaints (self-service only).
Bank Admin	All customers, analytics, reports, operations, and branch data (organization-wide).

3. Authentication & Access Control

Every session begins at the security boundary. Credentials are verified, a token is issued, and a role-based engine decides what the authenticated identity is allowed to do.

Stage	Purpose
Login	Username or ID, password, and multi-factor authentication (MFA) for identity assurance.
JWT Token	Issues a short-lived access token and a refresh token for stateless, scalable sessions.
RBAC Engine	Maps roles (Customer and Admin) to a granular set of least-privilege permissions.

4. Role-Based Dashboards

Once authenticated, each role lands on a tailored dashboard. Both dashboards embed the same AI Banking Assistant, but its scope and verbs differ by role.

Customer Dashboard	Bank Admin Dashboard
Account summary & balances	Customer & account management

Transactions history	Transactions & analytics
Loan details and EMIs	Loan & credit card management
Credit cards and limits	Complaint management
Complaints register & tracking	Branch performance
AI Banking Assistant: chat with the bank	Reports & operational analytics
Profile management	AI Banking Assistant: ask, analyze, act

5. The Application Layer (FastAPI)

A set of focused, independently scalable services, built with FastAPI, exposes the bank's capabilities as clean APIs. Each service owns a single domain, making the system easier to reason about, test, and evolve.

Service	Responsibility
Auth Service	Login, MFA, and token issuance & validation.
User Service	Profile, preferences, and role management.
Account Service	Accounts, balances, and summaries.
Transaction Service	Transactions and payments.
Loan Service	Loans, EMIs, and dues.
Card Service	Cards, limits, and payments.
Complaint Service	Register, track, and resolve complaints.
Analytics Service	Reports, insights, and dashboards.
Chat Assistant API	Send/receive messages and maintain conversation session state.

6. MySQL Database: The Core Data Store

A relational MySQL database is the system of record. A normalized schema links identities, roles, and financial entities through primary and foreign keys, keeping data consistent and queryable.

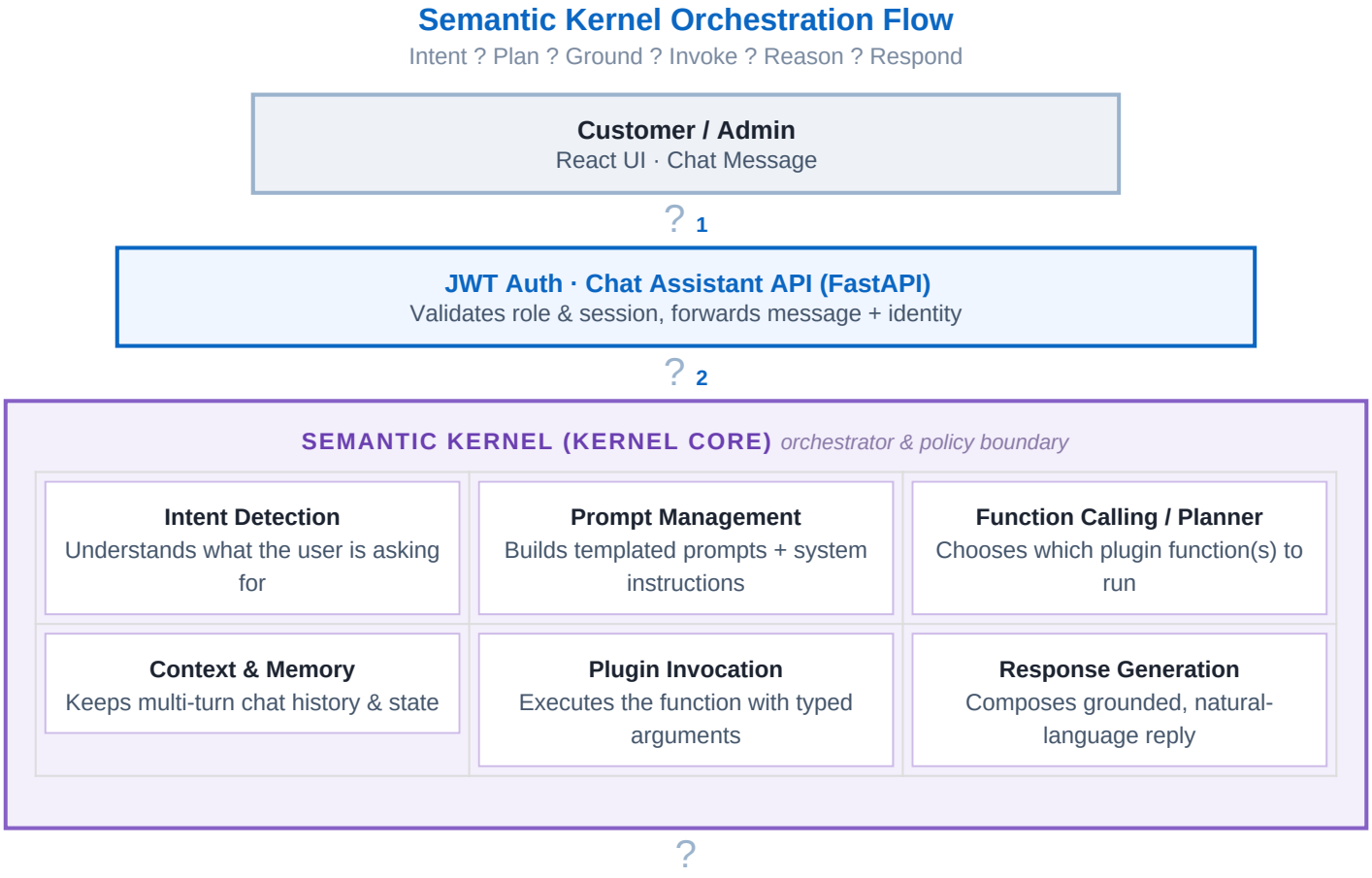
Table	Key Fields	Purpose
users	user_id (PK), username, password_hash, role_id (FK)	Authentication identities.
roles	role_id (PK), role_name, description	RBAC role definitions.
customers	customer_id (PK), name, email, phone, address	Customer master data.
accounts	account_id (PK), customer_id (FK), account_type, balance	Bank accounts & balances.
transactions	transaction_id (PK), account_id (FK), amount, status	Money movement records.
loans	loan_id (PK), customer_id (FK), loan_amount, emi_amount	Loan lifecycle & dues.
credit_cards	card_id (PK), customer_id (FK), credit_limit, available_limit	Card limits & usage.
complaints	complaint_id (PK), customer_id (FK), type, status	Complaint tracking.
branches	branch_id (PK), branch_name, location, manager_id	Branch & operations data.

7. Semantic Kernel Orchestration Layer

The intelligence of Smart Bank lives in the Semantic Kernel orchestration layer. It sits between the chat interface and the bank's capabilities, turning a free-form request into a precise, governed sequence of operations.

Kernel Stage	What it does
Intent Detection	Interprets what the user actually wants from natural language.

Prompt Management	Builds and templates the prompts that guide the model's reasoning.
Function Calling	Selects and invokes the right banking function for the intent.
Context & Memory	Maintains conversation context so multi-turn dialogue stays coherent.
Plugin Invocation	Routes the request to the correct banking capability plugin.
Response Generation	Composes a clear, grounded answer to return to the user.



PLUGINS (BANKING CAPABILITIES) 5

Account · Transaction · Loan · Card · Complaint · Analytics · Customer · Payment. Typed functions call FastAPI services, which query / update the **MySQL Database**.

KNOWLEDGE & SEARCH (RAG) 4

Azure AI Search · vector search over policy docs, FAQs, statements & guidelines. Grounds answers in the bank's own content.

AZURE OPENAI SERVICE 6

GPT-4o / GPT-4.1 · embeddings. Chat completion & function-calling decisions, reasoning over context + retrieved knowledge.

?

OBSERVABILITY · OpenTelemetry (every step is traced)
Traces · Metrics · Logs · AI Token Usage? OTel Collector ? Azure Monitor / Application Insights

Request path: steps 1 to 6. Response path returns through the Chat Assistant API to the user (steps 7 to 8). Telemetry is captured throughout.

Figure 2 — The Semantic Kernel orchestration flow, from chat message to grounded response, with OpenTelemetry tracing every step.

8. Plugins, Knowledge Search & Azure OpenAI

Three capabilities power the assistant's reasoning: a library of banking plugins for actions, a retrieval-augmented knowledge base for grounding, and Azure OpenAI for language understanding.

Capability	Role in the assistant
Plugins (Banking Capabilities)	Safe, typed operations the assistant can call: Account, Transaction, Loan, Card, Complaint, Analytics, Customer, and Payment. Each maps to an application-layer service.
Knowledge & Search (RAG)	Retrieval-Augmented Generation over Azure AI Search (vector search) across policy documents, FAQs, statements, and guidelines, grounding responses in the bank's own content.
Azure OpenAI Service	GPT-4o / GPT-4.1 with an embeddings model provide chat completion, function calling, and response generation: the linguistic engine behind every conversation.

9. External Integrations & Document Storage

Smart Bank does not operate in isolation. It connects to the broader banking ecosystem and stores documents durably in the cloud.

Component	Purpose
Payment Gateway	Processes payments and settlements.
SMS / Email Service	Delivers alerts, OTPs, and notifications.

KYC / AML Service	Identity verification and anti-money-laundering checks.
Credit Bureau API	Credit scores and history for lending decisions.
Core Banking System	Authoritative ledger and account operations.
Azure Blob Storage	Statements, documents, KYC files, loan agreements, policies, and forms.

10. Observability & Telemetry (OpenTelemetry)

Observability is the feedback loop that keeps the platform healthy. OpenTelemetry instruments the entire stack and pipes signals into Azure monitoring, audit logging, and alerting.

> **Pipeline:** Instrumentation (traces, metrics, logs, events) → OTel Collectors → Telemetry Data → Azure Monitoring → Audit & Security Logging → Alerting & Notifications

Stage	What it captures
Instrumentation	Request/response traces, DB query performance, API latency, and AI token usage.
Collectors	The OTel Collector gathers and forwards telemetry to backends.
Azure Monitoring	Application Insights dashboards, workbooks, alerts, and performance views.
Audit & Security Logging	Login attempts, RBAC changes, data-access logs, and compliance trails.
Alerting & Notifications	Email, Teams/Slack, SMS alerts, and incident escalation.

11. End-to-End Data Flow

Bringing every layer together, a single request travels a clear, traceable path from the user interface to the AI and back, while telemetry is captured at every hop.

> **Flow:** User → React UI (Dashboard / Chat) → JWT Auth → FastAPI APIs → Semantic Kernel (Orchestration) → Plugins / Functions → MySQL DB (Data Retrieval) → Azure OpenAI (Response

> Generation) → Response to UI → Telemetry Captured

Conclusion

Smart Bank shows how modern AI can be woven into banking without sacrificing security or control. Role-based dashboards keep experiences tailored and safe; Semantic Kernel and Azure OpenAI turn natural language into grounded action; MySQL provides a trustworthy system of record; and OpenTelemetry ensures the whole platform is observable and auditable. Every component is a managed, cloud-native service that integrates natively with the others, eliminating the friction of stitching together disparate tools and positioning the bank to deliver intelligent service that is **fast, secure, and reliable**.

A reference architecture for intelligent, secure, and observable digital banking.

OpenTelemetry Developer Handbook – Azure, GCP & AWS

CHAPTER 1

What is OpenTelemetry?

OpenTelemetry (OTel) is an open-source observability framework and toolkit that gives you a single, vendor-neutral way to generate, collect and export *telemetry data* — the signals your application produces to tell you what it is doing and how healthy it is.

It is a **CNCF Graduated project** (the highest maturity level), widely adopted by Google, Microsoft, AWS, Datadog, and hundreds of others.

?

WHY SHOULD YOU CARE AS A JUNIOR DEVELOPER?

When something breaks in production, you need to know *where* it broke, *why*, and *how long* it has been broken. OpenTelemetry gives you that information automatically, with one consistent standard.

The Three Pillars of Observability

?

Traces

A trace follows a single request across multiple services. Each step is a **span**.

?

Metrics

Numeric measurements over time: request count, error rate, memory usage, custom KPIs.

?

Logs

Timestamped event records. OTel correlates logs with the exact trace and span they belong to.

?	**OTEL COLLECTOR IS YOUR BEST FRIEND** The Collector receives data from your app, transforms or filters it, and forwards it to one or many backends. Switch cloud vendors without changing application code.

CHAPTER 2

Getting Started – Your First Instrumented App

We will use **Node.js** as the example. The same concepts apply to Python, Java, Go, .NET, etc.

Step 1 – Install the Core SDK Packages

```
npm install @opentelemetry/api @opentelemetry/sdk-node @opentelemetry/auto-instrumentations-node
```

Step 2 – Create instrumentation.js

```
// instrumentation.js — load BEFORE your app
const {{ NodeSDK }} = require('@opentelemetry/sdk-node');
const {{ getNodeAutoInstrumentations }} = require('@opentelemetry/auto-instrumentations-node');
const {{ Resource }} = require('@opentelemetry/resources');
const {{ SemanticResourceAttributes }} = require('@opentelemetry/semantic-conventions');
const {{ ConsoleSpanExporter }} = require('@opentelemetry/sdk-trace-node');

const sdk = new NodeSDK({{
  resource: new Resource({{
    [SemanticResourceAttributes.SERVICE_NAME]: 'my-first-service',
    [SemanticResourceAttributes.SERVICE_VERSION]: '1.0.0',
  }}),
  traceExporter: new ConsoleSpanExporter(),
  instrumentations: [getNodeAutoInstrumentations()],
}});
sdk.start();
process.on('SIGTERM', () => sdk.shutdown().finally(() => process.exit(0)));
```

Step 3 – Run your app

```
node -r ./instrumentation.js app.js
```

?	**TRACES WORKING!** ConsoleSpanExporter is only for development. The next chapters replace it with a real cloud exporter.

CHAPTER

3

Connecting to Microsoft Azure

Az

Azure Monitor + Application Insights

The Azure-native observability backend for OTel

1

Create an Application Insights Resource

Azure Portal ? "Application Insights" ? Create. Copy the **Connection String** from the resource overview.

2

Install the Azure Monitor exporter

```
npm install @azure/monitor-opentelemetry-expor
```

3

Update instrumentation.js

```
const {{ AzureMonitorTraceExporter }} = requir
const {{ AzureMonitorMetricExporter }} = requi
const {{ PeriodicExportingMetricReader }} = re
const connectionString = process.env.APPLICATI
const sdk = new NodeSDK({{
  traceExporter: new AzureMonitorTraceExporter
  metricReader: new PeriodicExportingMetricRea
    exporter: new AzureMonitorMetricExporter({
  }}),
  instrumentations: [getNodeAutoInstrumentatic
}});
sdk.start();
```

4

Set environment variable and run

```
# Linux / macOS
export APPLICATIONINSIGHTS_CONNECTION_STRING="
node -r ./instrumentation.js app.js

# Windows PowerShell
$env:APPLICATIONINSIGHTS_CONNECTION_STRING = "
node -r ./instrumentation.js app.js
```

5

Verify in Azure Portal

Azure Portal ? Application Insights ? Transaction Search, Application Map, Performance, Logs (KQL).

```
requests
| where timestamp > ago(1h) and duration > 500
| project timestamp, name, duration, resultCod
| order by duration desc | take 50
```

CHAPTER

4

Connecting to Google Cloud (GCP)

G

Cloud Trace + Cloud Monitoring

Google Cloud's distributed tracing and metrics platform

1

Enable APIs and create a Service Account

```
gcloud services enable cloudtrace.googleapis.c
gcloud iam service-accounts create otel-export
gcloud projects add-iam-policy-binding YOUR_PR
gcloud iam service-accounts keys create ./gcp-
```

2

Install and configure GCP exporters

```
npm install @google-cloud/opentelemetry-cloud-
```

```
const {{ TraceExporter }} = require('@google-c
const {{ MetricExporter }} = require('@google-
const projectId = process.env.GOOGLE_CLOUD_PRO
const sdk = new NodeSDK({{
  traceExporter: new TraceExporter({{ projectI
  metricReader: new PeriodicExportingMetricRea
    exporter: new MetricExporter({{ projectId
  }}),
  instrumentations: [getNodeAutoInstrumentatio
}});
sdk.start();
```

3

Set credentials and run

```
export GOOGLE_APPLICATION_CREDENTIALS="./gcp-c
export GOOGLE_CLOUD_PROJECT="your-gcp-project-
node -r ./instrumentation.js app.js
```

Google Cloud Console ? Cloud Trace ? Trace Explorer ? click any trace for the full span view.

CHAPTER

5

Connecting to Amazon Web Services (AWS)

AWS

AWS X-Ray + CloudWatch + ADOT

AWS Distro for OpenTelemetry — AWS's official OTel distribution

?

AWS X-RAY USES A SPECIAL TRACE FORMAT

X-Ray requires timestamp-based trace IDs. You must include AWSXRayIdGenerator or traces will not appear correctly.

1

Install ADOT packages

```
npm install @opentelemetry/id-generator-aws-xr
```

2

Update instrumentation.js

```
const {{ AWSXRayIdGenerator }} = require('@ope
const {{ AWSXRayPropagator }} = require('@oper
const {{ OTLPTraceExporter }} = require('@oper
const {{ propagation }} = require('@opentelem
propagation.setGlobalPropagator(new AWSXRayPr
const sdk = new NodeSDK({{
  idGenerator: new AWSXRayIdGenerator(),  // C
  traceExporter: new OTLPTraceExporter({{ url:
  instrumentations: [getNodeAutoInstrumentatio
}});
sdk.start();
```

3

Run and verify in AWS Console

```
export AWS_REGION=us-east-1
node -r ./instrumentation.js app.js
```

AWS Console ? CloudWatch ? X-Ray ? Traces. Check X-Ray ? Service Map for auto-generated topology.

CHAPTER

6

Best Practices & Quick Reference

Quick Comparison: Azure vs GCP vs AWS

FEATURE	AZURE MONITOR	GCP CLOUD TRACE	AWS X-RAY
Trace backend	Application Insights	Cloud Trace	AWS X-Ray
Metric backend	Azure Monitor Metrics	Cloud Monitoring	Amazon CloudWatch
Auth method	Connection String	Service Account JSON	IAM Role / Keys
Query language	KQL (Kusto)	Filter expressions	CloudWatch Insights
Free tier	5 GB/month	2.5M spans/month	100K traces/month
Special note	None	Enable APIs in console	X-Ray ID generator required

Common Errors and Fixes

ERROR	CAUSE	FIX
`No spans exported`	SDK not started before app	`node -r ./instrumentation.js app.js`
`401 Unauthorized (Azure)`	Wrong connection string	Check APPLICATIONINSIGHTS_CONNECTION_STRING
`PERMISSION_DENIED (GCP)`	Missing SA roles	`Add roles/cloudtrace.agent`
Traces missing in X-Ray	Missing ID generator	Add idGenerator: new AWSXRayIdGenerator()
`ECONNREFUSED :4317`	Collector not running	Start the Collector container first

?	**NEXT STEPS** 1. Add custom spans to your key business functions. 2. Add custom metrics (orders.processed, queue.depth). 3. Set up alerts on error rate and p99 latency. 4. Explore OTel Collector processors: filter, attributes, tail_sampling. 5. Read the official docs at opentelemetry.io.

Zero-Downtime Deployment in Azure App Service: Deployment Slots, Health Checks and Rollbacks

AZURE APP SERVICE DEPLOYMENT GUIDE

Zero-Downtime Deployment in Azure App Service

Deployment Slots, Health Checks, Database Migrations, GitHub Actions and Rollbacks

Deploying an application should not require displaying a maintenance page, restarting the production application in front of users, or hoping that the new release starts successfully. Azure App Service provides **deployment slots** that allow teams to deploy, initialize, validate and test a new application version before it receives production traffic.

A properly designed slot-based deployment process separates two activities that are often incorrectly treated as one operation:

- **Deploying the release** to an isolated staging environment.
- **Releasing the application** by directing production traffic to the validated version.

?	**THE CENTRAL IDEA** Do not build and initialize a new release while customers are using it. Build it, deploy it, warm it up and validate it in staging first. Only after it passes the release gates should production traffic be redirected to it.

CHAPTER 1

Production and Staging Slots

Every Azure App Service application has a default **production slot**. When the App Service plan supports deployment slots, you can create additional live environments such as staging, testing or pre-production.

Each slot has its own hostname, deployed application content and configurable settings. For example:

```
Production:
https://contoso-api.azurewebsites.net

Staging:
https://contoso-api-staging.azurewebsites.net
```

The responsibility of each slot

SLOT	PURPOSE	TRAFFIC
Production	Hosts the currently approved release.	Receives normal customer traffic.
Staging	Hosts the candidate release for validation.	Receives only deployment and test traffic.

Create a staging slot using Azure CLI

```
az webapp deployment slot create \
  --resource-group rg-production-app \
  --name contoso-api \
  --slot staging \
  --configuration-source contoso-api
```

??

APP SERVICE PLAN REQUIREMENT

Deployment slots are supported on Standard, Premium and Isolated App Service plans. The number of available slots depends on the plan tier. Confirm capacity and slot limits before designing the release workflow.

CHAPTER 2

Slot Swaps and Application Warm-Up

A slot swap is not the same as copying files from staging to production. Azure prepares the staging application with the target slot's applicable settings, restarts processes when required, sends warm-up requests and then redirects traffic.

What happens during a swap?

> **1. Apply target configuration**
> Azure applies the target slot's applicable configuration to the source slot.
>
> **2. Restart affected processes**
> Application instances restart when configuration changes require it.
>
> **3. Warm up the source slot**
> Azure sends requests to initialize the application on each instance.
>
> **4. Validate readiness**
> The platform waits for the configured warm-up process to succeed.
>
> **5. Redirect traffic**
> Production routing moves to the prepared application version.

Warm-up is essential for applications that perform initialization tasks such as loading configuration, establishing connection pools, compiling views, populating caches, loading machine-learning models or resolving external dependencies.

Configure a custom swap warm-up path

```
WEBSITE_SWAP_WARMUP_PING_PATH=/health/ready
WEBSITE_SWAP_WARMUP_PING_STATUSES=200
```

The warm-up endpoint should return success only when the application is actually ready to serve traffic. A shallow endpoint that always returns HTTP 200 can allow an incomplete or unusable application instance to enter production.

Preview and execute the swap

```
# Preview the swap and apply production configuration to staging
az webapp deployment slot swap \
  --resource-group rg-production-app \
  --name contoso-api \
  --slot staging \
  --target-slot production \
  --action preview

# Complete the swap after validation
az webapp deployment slot swap \
```

```
  --resource-group rg-production-app \
  --name contoso-api \
  --slot staging \
  --target-slot production \
  --action swap
```

?	**USE SWAP WITH PREVIEW FOR SENSITIVE APPLICATIONS** Swap with preview gives the team an additional validation window after production configuration is applied to staging but before production traffic is redirected.

CHAPTER 3

Sticky Application Settings

During a slot swap, some configuration values should move with the application, while environment-specific values should remain attached to their original slot. Azure calls environment-specific values **deployment slot settings**, commonly referred to as **sticky settings**.

SETTING	RECOMMENDED BEHAVIOUR	REASON
Database connection string	Sticky	Staging and production may use different databases.
API credentials	Sticky	Prevents staging from calling production integrations.
Application Insights connection	Usually sticky	Keeps staging telemetry separate from production.
Release version	Swappable	The value should move with the deployed release.
Feature flag	Depends on ownership	Decide whether the flag belongs to the release or environment.

Configure a sticky setting

```
az webapp config appsettings set \
  --resource-group rg-production-app \
  --name contoso-api \
  --slot staging \
  --slot-settings \
    ENVIRONMENT_NAME=staging \
```

```
    DATABASE_CONNECTION_STRING="staging-database-connection" \
    APPLICATIONINSIGHTS_CONNECTION_STRING="staging-insights-connection"
```

?	**A DANGEROUS CONFIGURATION MISTAKE** If the staging database connection is not configured correctly, the staging application may test against or modify production data. Treat slot configuration with the same level of control as application code.

CHAPTER 4

Database Migration Considerations

Deployment slots can make the web application deployment nearly seamless, but they do not automatically make database changes backward compatible. During a release, the old and new application versions can temporarily exist at the same time. Therefore, the database must support both versions throughout the transition.

Use the expand-and-contract pattern

Phase 1 — Expand
Add new tables, columns, indexes or stored procedures without removing structures used by the existing application.

Phase 2 — Deploy
Deploy an application version that can operate safely with both the old and new schema.

Phase 3 — Migrate
Backfill or transform existing data using a controlled and observable process.

Phase 4 — Contract
Remove obsolete columns or tables only after the previous application version can no longer receive traffic and rollback is no longer required.

Safe and unsafe database changes

CHANGE	RISK	RECOMMENDED APPROACH
Add a nullable column	Low	Add it before deploying the new application.
Create a new table	Low	Create it as an additive migration.

CHANGE	RISK	RECOMMENDED APPROACH
Rename a column	**High**	Add a new column, copy data and remove the old column later.
Drop a column	**Critical**	Delay until rollback to the old application is no longer required.
Add a required column	**Medium–High**	Add it as nullable, backfill it, and enforce the constraint later.

??	**DO NOT RUN DESTRUCTIVE MIGRATIONS DURING APPLICATION STARTUP** Multiple App Service instances may start simultaneously, resulting in migration conflicts or database locks. Run controlled migrations as a separate pipeline step and record exactly which migration version was applied.

CHAPTER

5

Health Checks and Release Validation

A deployment completing successfully only proves that files or a container image reached App Service. It does not prove that the application started correctly, can connect to its dependencies or can process business requests.

Configure an application endpoint such as **/health** or **/health/ready**. A meaningful readiness endpoint can validate:

- The application process is running.
- Required configuration is loaded.
- The database is reachable.
- Critical queues, caches or messaging services are reachable.
- The application has completed its startup sequence.

Enable App Service Health Check

```
az webapp config set \
  --resource-group rg-production-app \
  --name contoso-api \
  --generic-configurations '{"healthCheckPath": "/health/ready"}'
```

App Service Health Check regularly sends requests to the configured path on each instance. An endpoint response in the HTTP 200–299 range is treated as healthy. App Service can remove unhealthy instances from load balancing and continue checking them for recovery.

Run a staging smoke test

```
STAGING_URL="https://contoso-api-staging.azurewebsites.net"

curl --fail \
  --retry 12 \
  --retry-delay 10 \
  --retry-all-errors \
  "${STAGING_URL}/health/ready"

curl --fail "${STAGING_URL}/api/version"
curl --fail "${STAGING_URL}/api/smoke-test"
```

?	**LIVENESS AND READINESS ARE DIFFERENT** A liveness endpoint answers, “Is the process alive?” A readiness endpoint answers, “Can this application instance safely serve traffic?” Use readiness for deployment validation and swap warm-up.

CHAPTER

6

Rollback Strategy

After staging is swapped into production, the previous production version moves to the staging slot. This creates a fast rollback path because the earlier release remains deployed and can be swapped back.

Rollback command

```
az webapp deployment slot swap \
  --resource-group rg-production-app \
  --name contoso-api \
  --slot staging \
  --target-slot production
```

Rollback decision process

SIGNAL	SUGGESTED RESPONSE

Health endpoint fails	Rollback immediately.
Significant increase in HTTP 5xx errors	Rollback and investigate application logs.
Latency exceeds the release threshold	Pause, monitor briefly and rollback if sustained.
Non-critical UI defect	Evaluate business impact before rollback.
Destructive database migration already completed	Follow the database recovery plan; a slot swap alone may not be safe.

?	**A SWAP DOES NOT ROLL BACK THE DATABASE** Application rollback and database rollback are separate operations. This is why database changes must remain backward compatible for at least the duration of the rollback window.

CHAPTER

7

GitHub Actions Deployment Flow

A production-ready GitHub Actions workflow should deploy to staging first, validate the release, optionally run a controlled database migration, swap staging into production and then perform post-deployment verification.

Recommended pipeline

1. Checkout source code
2. Restore dependencies
3. Build and run automated tests
4. Authenticate to Azure using OpenID Connect
5. Deploy the artifact to staging
6. Wait for readiness and run smoke tests
7. Apply backward-compatible database migrations
8. Swap staging into production
9. Validate production health and monitor telemetry

Complete GitHub Actions example

```
name: Deploy Azure App Service
```

```yaml
on:
  push:
    branches:
      - main
  workflow_dispatch:

permissions:
  contents: read
  id-token: write

env:
  RESOURCE_GROUP: rg-production-app
  WEBAPP_NAME: contoso-api
  STAGING_SLOT: staging
  NODE_VERSION: 20
  STAGING_URL: https://contoso-api-staging.azurewebsites.net
  PRODUCTION_URL: https://contoso-api.azurewebsites.net

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    environment: production

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Configure Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: npm

      - name: Install dependencies
        run: npm ci

      - name: Run automated tests
        run: npm test

      - name: Build application
        run: npm run build --if-present

      - name: Create deployment package
        run: |
          zip -r release.zip . \
            -x ".git/*" \
            -x ".github/*" \
            -x "release.zip"

      - name: Sign in to Azure using OpenID Connect
        uses: azure/login@v2
        with:
          client-id: ${{ secrets.AZURE_CLIENT_ID }}
          tenant-id: ${{ secrets.AZURE_TENANT_ID }}
```

```
      subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}

  - name: Deploy release to staging
    uses: azure/webapps-deploy@v3
    with:
      app-name: ${{ env.WEBAPP_NAME }}
      slot-name: ${{ env.STAGING_SLOT }}
      package: release.zip

  - name: Wait for staging readiness
    shell: bash
    run: |
      for attempt in {1..20}; do
        echo "Staging readiness attempt ${attempt}"

        if curl \
          --silent \
          --show-error \
          --fail \
          "${STAGING_URL}/health/ready"; then
          echo "Staging is ready."
          exit 0
        fi

        sleep 15
      done

      echo "Staging did not become ready within the expected time."
      exit 1

  - name: Run staging smoke tests
    shell: bash
    run: |
      curl --fail --show-error "${STAGING_URL}/api/version"
      curl --fail --show-error "${STAGING_URL}/api/smoke-test"

  - name: Run backward-compatible database migrations
    shell: bash
    env:
      DATABASE_CONNECTION_STRING: ${{ secrets.DATABASE_CONNECTION_STRING }}
    run: npm run database:migrate

  - name: Swap staging into production
    shell: bash
    run: |
      az webapp deployment slot swap \
        --resource-group "${RESOURCE_GROUP}" \
        --name "${WEBAPP_NAME}" \
        --slot "${STAGING_SLOT}" \
        --target-slot production

  - name: Validate production
    shell: bash
    run: |
```

```
for attempt in {1..12}; do
  echo "Production validation attempt ${attempt}"

  if curl \
    --silent \
    --show-error \
    --fail \
    "${PRODUCTION_URL}/health/ready"; then
    echo "Production deployment is healthy."
    exit 0
  fi

  sleep 10
done

echo "Production validation failed."
exit 1
```

?	**PREFER OPENID CONNECT** OpenID Connect allows GitHub Actions to obtain short-lived Azure credentials instead of storing a long-lived client secret or App Service publishing profile in GitHub.

For high-risk production environments, configure the GitHub **production environment** with required reviewers. The workflow can deploy and test staging automatically, pause for approval and then perform the production swap.

CHAPTER 8

Common Deployment Mistakes

MISTAKE	IMPACT	PREVENTION
Deploying directly to production	Users experience startup failures or downtime.	Deploy to staging and swap after validation.
No health endpoint	A broken application can pass the deployment stage.	Implement liveness and readiness endpoints.
Shallow health check	The process appears healthy while dependencies are unavailable.	Check critical dependencies with strict timeouts.

MISTAKE	IMPACT	PREVENTION
Incorrect sticky settings	Staging connects to production resources or secrets move unexpectedly.	Document and audit slot-specific configuration.
Destructive schema migration	The previous application version can no longer run.	Use expand-and-contract migrations.
No post-swap validation	Production failures remain undetected.	Run smoke tests and monitor telemetry after every swap.
Overwriting the old release immediately	The fastest rollback option is lost.	Preserve the previous version in staging during the verification window.
Using long-lived deployment secrets	Credential leakage creates a security risk.	Use GitHub OpenID Connect and minimum Azure permissions.

CHAPTER 9

Production Release Checklist

Before deployment

- Confirm the staging slot exists and is correctly configured.
- Review sticky settings and connection strings.
- Confirm database migrations are backward compatible.
- Record the current release version and deployment artifact.
- Confirm alerts, dashboards and Application Insights are available.

Before the swap

- Verify the staging readiness endpoint.
- Run automated smoke tests.
- Verify the deployed application version.
- Check startup logs for warnings and errors.
- Confirm the rollback owner and rollback command.

After the swap

- Call the production health and version endpoints.

- Monitor HTTP 5xx responses and failed requests.
- Compare latency with the pre-deployment baseline.
- Check database and dependency failures.
- Preserve the previous version in staging until the release is stable.
- Record the deployment outcome and release timestamp.

?	**THE OPTIMAL PRODUCTION PATH** Build once ? test the artifact ? deploy to staging ? warm up ? validate health ? apply safe migrations ? obtain approval ? swap ? validate production ? monitor ? preserve the previous version for rollback.

Conclusion

Zero-downtime deployment is not achieved by a slot swap alone. It is the result of combining deployment slots, application warm-up, meaningful health checks, controlled configuration, backward-compatible database changes, automated validation and a tested rollback strategy.

When these practices are built into GitHub Actions, a production release becomes a controlled and repeatable operation rather than a high-risk manual event.