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.

CHANGE	RISK	RECOMMENDED APPROACH
Create a new table	Low	Create it as an additive migration.
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.

Revision #1
Created 2026-07-16 05:37:44 UTC by Adarsh Dhananjaya K
Updated 2026-07-16 05:41:29 UTC by Adarsh Dhananjaya K