Azure Container Apps (ACA) provides **built-in load balancing** at the environment level, eliminating the need to manually create or manage an Azure Load Balancer resource for basic HTTP/HTTPS traffic distribution. This is handled automatically by an integrated **Envoy proxy,** which routes incoming traffic across app replicas within the environment. The proxy terminates TLS, supports HTTP/1.1 and HTTP/2, and exposes a single static IP address for the entire environment.

* **Key Features**:
  + **Automatic Scaling and Routing**: Traffic is load-balanced across replicas based on concurrency, CPU/memory, or HTTP requests. No additional configuration is needed for intra-environment balancing.
  + **Ingress Control**: Each app can enable ingress for external (public), internal (VNet-only), or environment-internal access.
  + **When to Use Built-in**: For simple setups where apps are in the same environment and you need basic round-robin or concurrency-based distribution.

For advanced scenarios (e.g., global distribution, WAF integration, or cross-environment balancing), integrate with external services like Azure Application Gateway, Azure Front Door, or Azure Load Balancer.

**Prerequisites**

* An Azure subscription.
* Azure CLI installed (version 2.50+ recommended) and logged in (az login).
* An existing Container Apps environment (create one if needed; see [Microsoft Docs](https://learn.microsoft.com/en-us/azure/container-apps/environment)).
* Docker image for your app (pushed to a registry like Azure Container Registry).
* For VNet-integrated setups: A dedicated subnet (minimum /24 for production) in an existing VNet.

**Step-by-Step: Setting Up Built-in Load Balancing (Basic HTTP Ingress)**

1. **Create or Update a Container App with Ingress Enabled**: Use Azure CLI to deploy an app. Ingress is enabled by default for external access unless specified otherwise.

Bash

az containerapp create \

--name myapp \

--resource-group myResourceGroup \

--environment myEnvironment \

--image nginx:latest \

--target-port 80 \

--ingress external \

--cpu 0.5 \

--memory 1.0Gi

* + --ingress external: Exposes the app publicly via the environment's load-balanced IP.
  + --target-port: The port your container listens on (e.g., 80 for HTTP).
  + For internal-only: Use --ingress internal.
  + For environment-internal (no public/VNet exposure): Use --ingress external with environment public access disabled (see Step 2).

This command automatically configures the Envoy proxy to route traffic to replicas.

1. **Configure Environment Networking (Optional for VNet-Internal Load Balancing)**: If you need private access (e.g., via VNet), integrate the environment with a VNet and disable public access.

bash

az containerapp env vnet add \

--name myEnvironment \

--resource-group myResourceGroup \

--vnet-name myVNet \

--subnet-name mySubnet

az containerapp env update \

--name myEnvironment \

--resource-group myResourceGroup \

--disable-public-network-access true

* + This creates an internal load balancer IP for the environment.
  + Traffic routes within the VNet; use private DNS for resolution.

1. **Scale the App for Load Balancing**: Define replicas or autoscale rules to distribute load.

bash

az containerapp update \

--name myapp \

--resource-group myResourceGroup \

--min-replicas 2 \

--max-replicas 10

*# Or add HTTP-based autoscale*

az containerapp revision set \

--name myapp \

--resource-group myResourceGroup \

--yaml scale.yml

Example scale.yml:

yaml

scale:

minReplicaCount: 2

maxReplicaCount: 10

rules:

- name: http-rule

http:

average: 100 *# Concurrent requests per replica*

The built-in balancer evenly distributes requests across active replicas.

1. **Verify Load Balancing**:
   * Get the FQDN: az containerapp show --name myapp --resource-group myResourceGroup --query properties.configuration.ingress.fqdn.
   * Test with curl <fqdn> and check logs: az monitor app-insights query --app myAppInsights --analytics-query "requests | take 10".
   * Monitor via Azure Portal > Container App > Metrics (e.g., Request Count, Replica Count).

**Advanced: Integrating External Load Balancer (e.g., Application Gateway for WAF)**

For Layer 7 routing, path-based rules, or WAF, use Azure Application Gateway as a frontend to the ACA internal load balancer.

1. **Create Application Gateway**:

bash

az network application-gateway create \

--resource-group myResourceGroup \

--name myAppGateway \

--location eastus \

--sku WAF\_v2 \

--public-ip-address myPublicIP \

--vnet-name myVNet \

--subnet myGatewaySubnet \

--priority 100 \

--capacity 2

1. **Configure Backend Pool**:
   * Target the ACA environment's internal IP (found in Portal > Environment > Overview > Internal IP).
   * Add a health probe: Path /healthz, Port 32172 (ACA default).
   * Use VNet peering if ACA and Gateway are in different VNets.
2. **Set Routing Rules**:

bash

az network application-gateway http-settings update \

--gateway-name myAppGateway \

--resource-group myResourceGroup \

--name appGatewayBackendHttpSettings \

--connection-timeout 30

az network application-gateway url-path-map create \

--gateway-name myAppGateway \

--resource-group myResourceGroup \

--name pathMap \

--paths /api/\* \

--backend-address-pool myBackendPool \

--backend-http-settings appGatewayBackendHttpSettings

1. **Test and Secure**:
   * Enable HTTPS with managed certificates.
   * Monitor health in Portal > Application Gateway > Backend Health.

For global setups, use Azure Front Door in front of regional Gateways. Refer to [Protect Azure Container Apps with Application Gateway](https://learn.microsoft.com/en-us/azure/container-apps/application-gateway) for full details.

**Best Practices**

* **Security**: Disable public access for production; use managed identities and NSGs.
* **Monitoring**: Integrate Azure Monitor and Application Insights for replica health and traffic patterns.
* **Cost Optimization**: Use consumption plans for variable loads; autoscale based on actual metrics.
* **Limitations**: Built-in balancer doesn't support TCP/UDP directly—use external Load Balancer for that.

References:

<https://learn.microsoft.com/en-us/azure/architecture/example-scenario/infrastructure/wordpress-container>

<https://rajanieshkaushikk.com/2024/10/03/azure-container-apps-secrets-storage-and-connectivity/>

<https://docs.snowflake.com/en/user-guide/privatelink-azure>

<https://azure.github.io/aca-dotnet-workshop/aca/00-workshop-intro/2-scenario-architecture/>

CNCF cloud native computing foundation envisions cloud-native applications to be loosely coupled , resilient , manageable and observable. Cloud-Native Apps should minimize the engineering effort to achieve the business benefits.



Scaling

**Overview of KEDA Integration in Azure Container Apps**

Azure Container Apps natively integrates Kubernetes Event-Driven Autoscaling (KEDA), so there's no need to install KEDA separately—it's enabled by default when you create a Container App environment. KEDA enables event-driven scaling based on external metrics or events (e.g., queue length in Azure Service Bus or Azure Queue Storage) rather than just CPU/memory usage. This allows your apps to scale from zero replicas during idle periods and react dynamically to workload spikes.

Scaling is configured via **limits** (min/max replicas), **rules** (HTTP, TCP, or custom/KEDA-based), and **behavior** (e.g., polling intervals, cool-down periods). The default polling interval is 30 seconds, and the cool-down period is 300 seconds.

Key supported KEDA scalers for Azure Container Apps include:

* Azure Service Bus (queues/topics)
* Azure Event Hubs
* Azure Queue Storage
* Apache Kafka
* Redis
* CPU/Memory (for resource-based scaling)
* And others via KEDA's full catalog (e.g., Azure Monitor metrics, cron schedules), as long as they align with Azure services.

For a full list of KEDA scalers, refer to the [KEDA documentation](https://keda.sh/docs/2.14/scalers/).

To configure scaling for Azure Container Apps based on CPU or memory usage**, you can use KEDA's built-in CPU and memory scalers, which are natively supported in Azure Container Apps.** This allows your app to scale dynamically based on resource utilization (e.g., CPU percentage or memory usage) rather than event-driven triggers like queues. Below is a step-by-step guide to set this up.

**Prerequisites**

* An Azure Container App environment created (e.g., via az containerapp env create --name <env-name> --resource-group <rg> --location <location>).
* Permissions to manage Container Apps in your Azure subscription.
* Azure CLI, Portal, or Bicep/ARM template tools for configuration.

**Step-by-Step Configuration**

**1. Understand CPU/Memory Scaling**

* **CPU Scaler**: Scales based on CPU usage percentage (e.g., 80% of allocated CPU).
* **Memory Scaler**: Scales based on memory usage percentage (e.g., 70% of allocated memory).
* You can set **minReplicas** (default: 0 for scale-to-zero) and **maxReplicas** (default: 10, up to 1,000).
* Scaling is triggered when the specified threshold is exceeded, checked at a default polling interval of 30 seconds.

**2. Configure Scaling Using Azure CLI**

You can define CPU/memory scaling rules when creating or updating a Container App.

**Example: Scale on 80% CPU Usage**

bash

az containerapp create \

--name myapp \

--resource-group myrg \

--environment myenv \

--image mcr.microsoft.com/azuredocs/containerapps-helloworld:latest \

--cpu 0.5 \

--memory 1.0Gi \

--min-replicas 1 \

--max-replicas 10 \

--scale-rule-name cpu-rule \

--scale-rule-type cpu \

--scale-rule-metadata "usage=80"

--cpu 0.5: Allocates 0.5 vCPU per container.

--memory 1.0Gi: Allocates 1 GiB of memory per container.

--scale-rule-type cpu: Specifies CPU-based scaling.

--scale-rule-metadata "usage=80": Triggers scaling when CPU usage exceeds 80%.

**Example: Scale on 70% Memory Usage**

bash

az containerapp update \

--name myapp \

--resource-group myrg \

--cpu 0.5 \

--memory 1.0Gi \

--min-replicas 1 \

--max-replicas 10 \

--scale-rule-name memory-rule \

--scale-rule-type memory \

--scale-rule-metadata "usage=70"

--scale-rule-type memory: Specifies memory-based scaling.

--scale-rule-metadata "usage=70": Triggers scaling when memory usage exceeds 70%.

**Combining CPU and Memory Rules** To scale based on either CPU or memory (whichever threshold is hit first):

bash

az containerapp update \

--name myapp \

--resource-group myrg \

--cpu 0.5 \

--memory 1.0Gi \

--min-replicas 1 \

--max-replicas 10 \

--scale-rule-name cpu-rule \

--scale-rule-type cpu \

--scale-rule-metadata "usage=80" \

--scale-rule-name memory-rule \

--scale-rule-type memory \

--scale-rule-metadata "usage=70"

Multiple rules are evaluated, and the highest replica count is applied.

**3. Configure Using Bicep (Alternative)**

For infrastructure-as-code, use Bicep to define CPU/memory scaling.

**Example: CPU Scaling (80% Threshold)**

bicep

resource containerApp 'Microsoft.App/containerApps@2024-10-01-preview' = {

name: 'myapp'

location: resourceGroup().location

properties: {

managedEnvironmentId: '/subscriptions/<sub-id>/resourceGroups/<rg>/providers/Microsoft.App/managedEnvironments/myenv'

template: {

containers: [

{

name: 'mycontainer'

image: 'mcr.microsoft.com/azuredocs/containerapps-helloworld:latest'

resources: {

cpu: 0.5

memory: '1.0Gi'

}

}

]

scale: {

minReplicas: 1

maxReplicas: 10

rules: [

{

name: 'cpu-rule'

custom: {

type: 'cpu'

metadata: {

usage: '80'

}

}

}

]

}

}

}

}

Deploy with: az deployment group create --resource-group <rg> --template-file app.bicep.

**Example: Memory Scaling (70% Threshold)**

bicep

resource containerApp 'Microsoft.App/containerApps@2024-10-01-preview' = {

name: 'myapp'

location: resourceGroup().location

properties: {

managedEnvironmentId: '/subscriptions/<sub-id>/resourceGroups/<rg>/providers/Microsoft.App/managedEnvironments/myenv'

template: {

containers: [

{

name: 'mycontainer'

image: 'mcr.microsoft.com/azuredocs/containerapps-helloworld:latest'

resources: {

cpu: 0.5

memory: '1.0Gi'

}

}

]

scale: {

minReplicas: 1

maxReplicas: 10

rules: [

{

name: 'memory-rule'

custom: {

type: 'memory'

metadata: {

usage: '70'

}

}

}

]

}

}

}

}

**4. Configure Using Azure Portal**

1. Go to your Container App in the Azure Portal.
2. Navigate to **Scale** > **Edit and deploy**.
3. Set **Min replicas** (e.g., 1) and **Max replicas** (e.g., 10).
4. Under **Scaling rules**, click **Add** > **Custom**.
5. Configure:
   * **Rule name**: e.g., cpu-rule.
   * **Type**: Select cpu or memory.
   * **Metadata**: Set usage to the desired percentage (e.g., 80 for CPU, 70 for memory).
6. Save and deploy.

**5. Validate and Monitor**

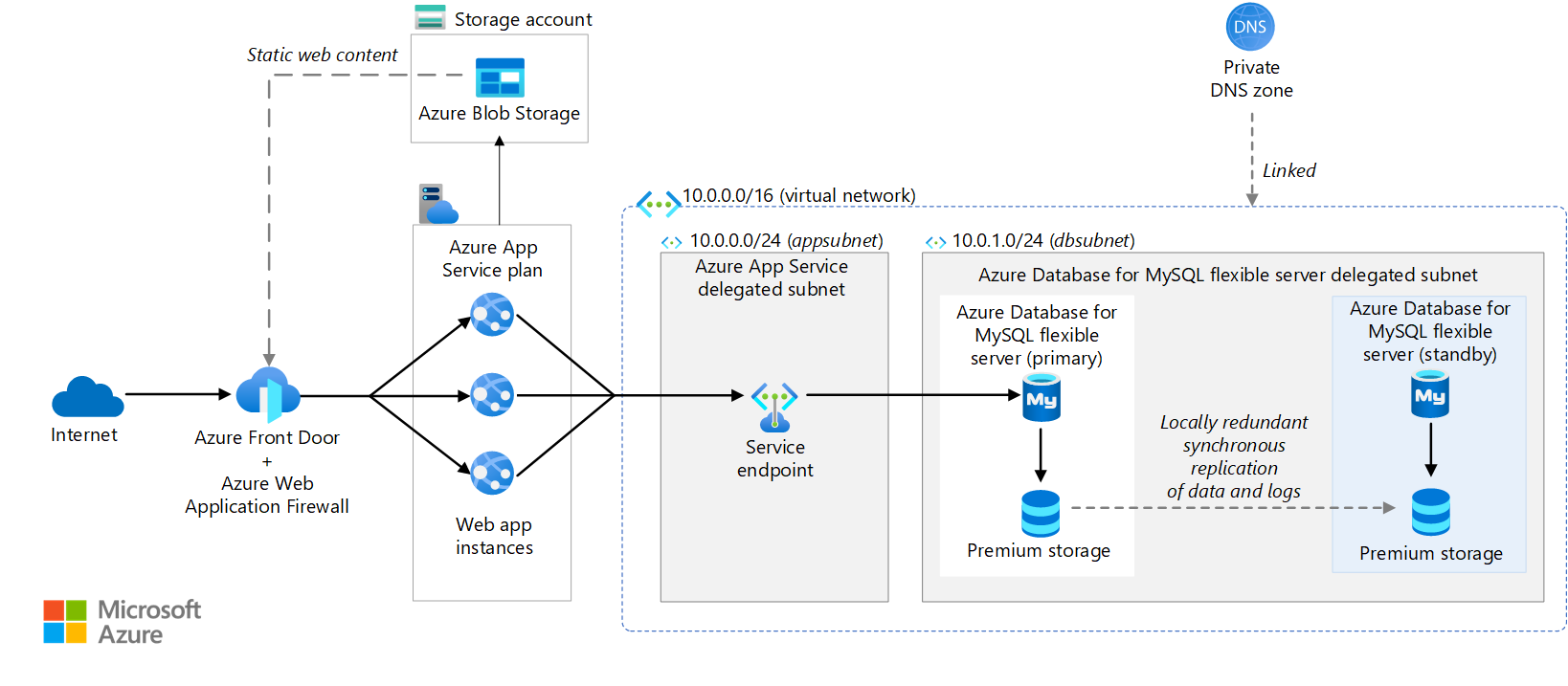
* **Test Scaling**: Simulate load (e.g., via a stress-testing tool like wrk or a custom script) to push CPU/memory usage above thresholds.
* **Monitor Replicas**: Check the **Scale** tab in the Portal or use az containerapp show --name myapp --resource-group myrg to verify replica count.
* **Logs and Metrics**: Use Azure Monitor or Container App logs (az containerapp logs show --name myapp --resource-group myrg) to track scaling events and resource usage.
* **Troubleshoot**: Ensure container resource limits (--cpu and --memory) are set appropriately, as scaling is based on these allocations.

**6. Advanced Considerations**

* **Scale-to-Zero**: Set minReplicas: 0 for cost savings during idle periods (supported for CPU/memory scalers).
* **Multiple Rules**: Combine CPU, memory, or other scalers (e.g., HTTP) for flexible scaling. The rule yielding the highest replica count takes precedence.
* **Tuning Behavior**: Adjust polling intervals or cool-down periods via scale.behavior (e.g., scaleDown: { policies: [{ type: 'Percent', value: 10, duration: 60 }] }) for smoother scaling.
* **Resource Allocation**: Ensure --cpu and --memory are sufficient for your workload to avoid premature scaling or throttling.

**Notes**

* CPU/memory scaling doesn't require managed identities or secrets, unlike event-driven scalers (e.g., Azure Service Bus).
* For very low-latency scaling, consider HTTP-based scaling if your app handles web traffic.
* Refer to the [KEDA documentation](https://keda.sh/docs/2.14/scalers/cpu/) for advanced CPU/memory scaler options.



**Azure Container Apps supports built-in authentication** and authorization (often called "Easy Auth") to secure your applications without writing custom code. This integrates with identity providers like Microsoft Entra ID (formerly Azure AD) to handle user sign-ins and access control. Whether you need to register an app in Microsoft Entra ID depends on your specific use case:

* **If you're enabling authentication for users** (e.g., restricting access to your Container App based on Entra ID sign-ins): Yes, you must register an app in Microsoft Entra ID. This establishes a trust relationship for the identity provider to issue tokens to your app.
* **If you're only using the Container App's managed identity** (e.g., for outbound calls to other Azure services like Key Vault or Azure Container Registry without user interaction): No app registration is required in Entra ID for the app itself. Managed identities handle authentication automatically.

Below, I'll explain the details for the common scenario (user authentication), including when it's needed and how to set it up.

**When Is App Registration Required?**

|  |  |  |
| --- | --- | --- |
| Scenario | App Registration in Entra ID Required? | Why? |
| User authentication/authorization (e.g., login to access the app) | Yes | Entra ID needs to know about your Container App to validate users and issue tokens. This is done via an app registration, which acts as the app's identity in Entra ID. |
| Daemon/app-to-service auth (e.g., your app calling APIs on behalf of itself) | Yes (for the app acting as a client) | Register the app as a client to use OAuth 2.0 client credentials flow. No redirect URI needed. |
| Native client access (e.g., mobile/desktop app calling your Container App's APIs) | No (for basic user auth setup) | You can skip registering the server-side app if only authenticating users, but register native clients separately for API access. |
| Managed identity for resource access (e.g., pulling images from ACR) | No | Managed identities (system- or user-assigned) are built into Azure and don't require manual Entra ID registration. |

In most cases involving Entra ID-based security for the Container App itself, registration is essential to avoid errors like invalid redirect URIs or failed token validation.

**How to Register and Configure for Azure Container Apps**

Follow these steps in the Azure portal to enable Entra ID auth (assumes you have an existing Container App):

1. **Navigate to Your Container App**:
   * Go to the Azure portal > Container Apps > Select your app.
   * In the left menu, under **Settings**, select **Authentication**.
2. **Add Identity Provider**:
   * Click **Add identity provider**.
   * For **Identity provider**, select **Microsoft**.
   * Choose **Tenant type** (e.g., "Current tenant" for internal users).
   * For **App registration type**, select **Create new app registration** (this auto-creates the Entra ID app for you).
     + Enter a **Name** (e.g., "MyContainerApp-Auth").
     + Under **Supported account types**, choose based on your needs (e.g., "Accounts in this organizational directory only" for single-tenant).
   * A **Client secret** is auto-generated—note it down if needed for custom configs.
   * Configure **Authentication requirements** (e.g., "Require authentication" to block unauthenticated access).
3. **Manual Registration (If Not Using Auto-Create)**:
   * If you prefer manual setup (e.g., for advanced roles or multi-tenant):
     + Go to Microsoft Entra ID > **App registrations** > **New registration**.
     + Enter a name and select **Supported account types**.
     + Under **Redirect URI**, add your Container App's URL (e.g., https://<app-name>.azurecontainerapps.io/.auth/login/aad/callback).
     + Note the **Application (client) ID** and **Directory (tenant) ID**.
     + In your Container App's Authentication settings, select **Provide the details of an existing app registration** and input the IDs.
4. **Expose an API (Optional for Authorization)**:
   * In the app registration > **Expose an API**, add a scope (e.g., access\_as\_user).
   * Grant admin consent if needed.
   * For client apps calling your APIs, add permissions under **API permissions** > **My APIs**.
5. **Test and Secure**:
   * Save changes. Your app now has endpoints like /.auth/login/aad for sign-in.
   * Update **Ingress** settings to block unsecured connections.
   * Test by accessing your app URL—users should be redirected to Entra ID login.

**Additional Notes**

* **Custom Domains/Front Door**: If using Azure Front Door, ensure the redirect URI in the app registration matches your custom domain to avoid callback errors.
* **Terraform/CLI**: As of now, Terraform doesn't fully support Container Apps auth blocks natively—use Azure CLI or portal for setup.
* **Costs/Security**: App registrations are free. Always use least-privilege scopes and rotate secrets.
* **Alternatives**: For non-Entra providers (e.g., Google, Facebook), registration is in those services, not Entra ID.

If this doesn't match your exact setup (e.g., managed identities only), provide more details for tailored advice!

**Specific Impacts for Azure Cache for Redis**

Since you’re using Azure Cache for Redis, the losses are particularly relevant in the following areas:

* **Local Testing**: Without Aspire, you might need to connect to an actual Azure Cache for Redis instance during development (incurring costs) or set up a local Redis instance manually. Aspire’s AddRedis provides a free, local Redis container that mimics Azure Cache for Redis’s behavior.
* **Configuration Management**: You’ll need to manually manage connection strings for local Redis and Azure Cache for Redis, increasing the risk of errors when switching environments. Aspire automates this with WithReference.
* **Azure Deployment**: Without Aspire’s azd integration, provisioning Azure Cache for Redis and connecting it to your app requires manual IaC or Azure Portal setup, which is more time-consuming and error-prone.
* **Observability**: Debugging caching issues (e.g., cache misses, connection failures) is harder without Aspire’s dashboard, which provides unified telemetry for Redis and your services.

**What You Lose Without .NET Aspire**

1. **Simplified Local Development Orchestration**
   * **Loss**: Without Aspire, you’ll need to manually set up and manage your local development environment, including the Redis instance (to simulate Azure Cache for Redis), your .NET services, and their dependencies. This involves writing and maintaining Docker Compose files, shell scripts, or manual commands to start services in the correct order.
     + Example: You’d need to run docker run -d -p 6379:6379 redis to start a local Redis container, manually configure connection strings, and ensure your .NET app connects to it correctly.
   * **Impact**: Increased setup time (potentially 20-40% more effort) and risk of configuration errors, especially for complex apps with multiple services or dependencies.
   * **Aspire Benefit**: Aspire automates this with a single builder.AddRedis("cache") in the AppHost project, spinning up a Redis container and injecting connection strings automatically. Running dotnet run launches everything in the correct order.
2. **Unified Application Model**
   * **Loss**: You’ll miss Aspire’s code-first application model, which defines your entire app topology (services, Redis, databases, etc.) in a single Program.cs file. Without Aspire, you’d use disparate tools (e.g., Docker Compose, scripts, or IDE settings) to define and manage your app’s components, leading to fragmented configurations.
     + Example: You might maintain a docker-compose.yml for Redis and separate launch settings for your .NET services, requiring manual synchronization.
   * **Impact**: Inconsistent environments across team members, increasing “works on my machine” issues and onboarding time for new developers.
   * **Aspire Benefit**: The AppHost project centralizes configuration (e.g., builder.AddProject<Projects.MyApi>("api").WithReference(cache)), ensuring all developers use the same setup and reducing configuration drift.
3. **Automatic Dependency Injection and Service Discovery**
   * **Loss**: Without Aspire, you’ll need to manually manage connection strings, environment variables, and service discovery for your .NET services to communicate with Redis or other components. This often involves hardcoding endpoints or using configuration files that are error-prone to maintain.
     + Example: You’d manually set a Redis connection string like localhost:6379 in your appsettings.json or environment variables and update it for production (e.g., myredis.redis.cache.windows.net:6380,ssl=true).
   * **Impact**: More boilerplate code and risk of misconfiguration when switching between local and production environments (e.g., forgetting to update the connection string).
   * **Aspire Benefit**: Aspire injects connection strings (e.g., for Redis) via WithReference, automatically resolving endpoints and ensuring services start only after dependencies (like Redis) are healthy.
4. **Real-Time Dashboard for Observability**
   * **Loss**: Aspire’s built-in dashboard provides real-time visibility into your app’s resources, including Redis health, logs, metrics, and traces (via OpenTelemetry). Without it, you’d need to set up separate monitoring tools (e.g., Prometheus, Grafana, or Redis CLI) to debug caching issues.
     + Example: To check Redis performance locally, you might need to run redis-cli MONITOR or set up a third-party tool, which requires additional configuration.
   * **Impact**: Slower debugging and less visibility into how your services interact with Redis, potentially prolonging issue resolution.
   * **Aspire Benefit**: The dashboard (accessible at http://localhost:18888) shows Redis metrics, logs, and app telemetry in one place, speeding up diagnostics.
5. **Streamlined Azure Deployment**
   * **Loss**: Aspire integrates with the Azure Developer CLI (azd) to generate Bicep files that provision Azure Cache for Redis and configure your app for production. Without Aspire, you’ll need to manually write Infrastructure-as-Code (IaC) templates (e.g., Bicep, ARM, or Terraform) or use the Azure Portal to set up Azure Cache for Redis and connect it to your app.
     + Example: You’d write a Bicep file like this manually:

bicep

resource redis 'Microsoft.Cache/Redis@2023-08-01' = {

name: 'myredis'

location: resourceGroup().location

properties: {

sku: { name: 'Standard', family: 'C', capacity: 0 }

enableNonSslPort: false

}

}

And then manually configure your app’s connection string to use the Azure Cache for Redis endpoint.

* + **Impact**: Increased deployment complexity and time, especially for teams unfamiliar with Azure IaC tools. Risk of misaligned configurations between local and production environments.
  + **Aspire Benefit**: Aspire generates Azure-compatible manifests automatically and integrates with azd up to provision Azure Cache for Redis, reducing manual effort and ensuring dev-prod parity.

1. **Team Productivity and Onboarding**
   * **Loss**: Without Aspire, new developers must learn and configure the local environment (e.g., installing Docker, setting up Redis, configuring connection strings) manually, which can take hours or days. This leads to slower onboarding and potential inconsistencies.
     + Example: A new developer might struggle to replicate your local Redis setup or miss a dependency, causing delays.
   * **Impact**: Reduced team velocity and higher setup costs, especially in larger teams or projects with frequent onboarding.
   * **Aspire Benefit**: A single dotnet run in the AppHost project sets up the entire environment (Redis, services, etc.), making onboarding as simple as cloning the repo and running one command.
2. **Built-In Integrations and Extensibility**
   * **Loss**: Aspire provides NuGet packages (e.g., Aspire.StackExchange.Redis) that simplify integration with Redis and Azure Cache for Redis, including health checks and client configuration. Without Aspire, you’d need to manually integrate and configure Redis clients like StackExchange.Redis, potentially writing more boilerplate code.
     + Example: You’d need to manually add health checks for Redis or handle connection failures in your code, which Aspire automates.
   * **Impact**: More development effort to achieve the same level of reliability and integration, especially for Azure-specific features.
   * **Aspire Benefit**: Pre-built integrations streamline Redis usage, and extensibility supports other Azure services (e.g., Azure SQL, Key Vault) if needed.