# Background Task Queue — Design Summary

## 1. Project Type Chosen

**Project type:** ASP.NET Core Web API (Web SDK / minimal hosting model).
**Why:**

- Web API project hosts controllers, health checks and also supports hosted services (BackgroundService) in the same process using the generic host.
- Minimal startup and configuration model in .NET 8 (WebApplication) simplifies wiring of DI, EF Core, hosted services and health checks.
- Same deployable artifact runs on Kestrel, IIS (in-process or out-of-process), or Azure App Service with small configuration changes.
- Hosted services (producer/consumer/warmup) run as background workers within the Web API process, avoiding separate worker process complexity for a simple, cost-efficient prototype.

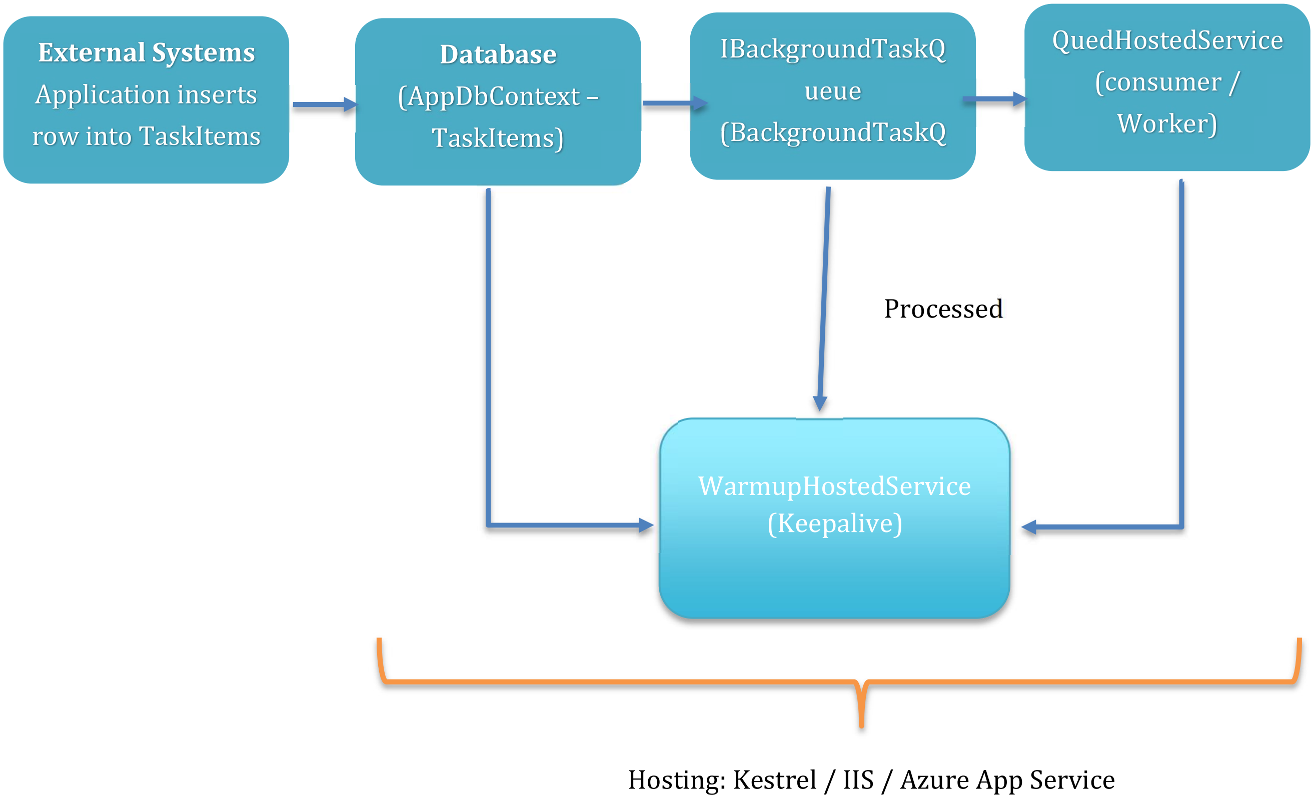
## 2. Architecture (Producer / Consumer / Task Flow / Hosting)

**Components:**

- AppDbContext (EF Core) — persistent storage of TaskItem.
- DbPollingProducerService — polls DB for new rows inserted by external systems, marks Enqueued = true, and enqueues them to IBackgroundTaskQueue.
- BackgroundTaskQueue (in-memory) — thread-safe queue (ConcurrentQueue + SemaphoreSlim); demo-only; replaceable with Azure Storage Queue or Service Bus for durability & multi-instance scale.
- QueuedHostedService — consumer that rehydrates pending DB items at startup, dequeues items and processes them with bounded concurrency and retry handling.
- TaskProcessorService — thin application service abstraction that delegates to the queue (keeps domain logic separate).
- WarmupHostedService — periodically queries queue count to keep the runtime active (reduces cold start).
- PI controllers — allow simulating external DB inserts and expose status/metrics endpoints.
- Health checks endpoint (/health) — used by hosting environment to determine health.

**Task Flow:**

1. External application inserts row into TaskItems.
2. DbPollingProducerService finds new rows, marks Enqueued, enqueues them to IBackgroundTaskQueue.
3. QueuedHostedService dequeues, processes the item, updates DB (Processed = true or Failed = true).
4. WarmupHostedService keeps runtime active by regularly calling GetApproximateCountAsync.



**Hosting Environments:**

- Local Kestrel (dotnet run / docker).
- IIS (Windows) in-process/out-of-process.
- Azure App Service (Windows/Linux).

## 3. Configuration Details to Prevent Shutdowns

**IIS (on Windows):**

- Install and enable the Application Initialization module.
- App Pool:
  - Set Start Mode = AlwaysRunning.
  - Set Idle Time-out (minutes) = 0 to prevent idle shutdown.
- Website:
  - Set Preload Enabled = True.
  - Configure Application Initialization to call a warmup URL (e.g., /health or /api/health/warmup).

**Azure App Service:**

- Enable Always On (under Configuration → General settings).
- Set the Health Check path to /health.
- Use Application settings to store connection strings and secrets.
- Optionally enable ARR affinity only if required.

**Kubernetes:**

- Use readiness and liveness probes (point them to /health).
- Configure PodDisruptionBudget and set minReplicas >= 1.

## 4. Strategy for Minimizing Azure Costs

- Decouple producer and consumer with durable queues for independent scaling.
- Choose the right compute size and enable Always On with WarmupHostedService.
- Use autoscaling with sensible min/max instance counts.
- Store secrets in Key Vault and use Managed Identity.
- Optimize database and polling frequency for cost and performance balance.
- Use burstable or spot compute for non-critical workloads.
- Monitor and alert on queue length, latency, and costs.

## 5. Production Readiness Checklist

- Replace in-memory queue with a durable queue (Azure Storage Queue or Service Bus).
- Store secrets in Azure Key Vault; use Managed Identity.
- Add CI/CD pipeline with GitHub Actions for App Service deployment.
- Add unit/integration tests covering end-to-end flow.
- Enable App Service Always On and configure Health Check path.