In **ASP.NET Core**, `IHostedService` (often misheard as *IHostingService*) is an interface used to run **background tasks** that start and stop **along with the application host**.

What is `IHostedService` ?

`IHostedService` is an interface from:

```
Microsoft.Extensions.Hosting
```
Copy code

It allows you to plug **long-running or background logic** into the ASP.NET Core application lifecycle.

```csharp
public interface IHostedService
{
    Task StartAsync(CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```
Copy code

When is it used?

Use `IHostedService` when you need:

- Background workers
- Message queue consumers
- Scheduled jobs
- Polling services
- Cache warm-up
- Event listeners
- Health monitoring
- Streaming / Kafka / RabbitMQ consumers

👉 Anything that **is not tied to an HTTP request**.

Lifecycle

Method	When it runs
`StartAsync`	When the application starts
`StopAsync`	When the application is shutting down

ASP.NET Core guarantees graceful shutdown via `CancellationToken` .

Simple Example

Background service that runs on startup

```csharp
public class MyHostedService : IHostedService
{
    public Task StartAsync(CancellationToken cancellationToken)
    {
```

```csharp
        Console.WriteLine("Application started");
        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        Console.WriteLine("Application stopping");
        return Task.CompletedTask;
    }
}
```

Register it

```csharp
builder.Services.AddHostedService<MyHostedService>();
```

`BackgroundService` (Most common)

ASP.NET Core provides a **base class** that implements `IHostedService` for you:

```csharp
public abstract class BackgroundService : IHostedService
{
    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);
}
```

Example: Continuous background task

```csharp
public class Worker : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            Console.WriteLine("Running...");
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Register:
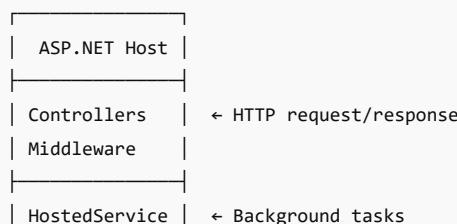
```csharp
builder.Services.AddHostedService<Worker>();
```

✅ This is the **recommended approach** for long-running services.

How it fits into ASP.NET Core architecture

```bash
 ┌───────────────┐
 │   ASP.NET Host │
 ├───────────────┤
 │ Controllers   │   ← HTTP request/response
 │ Middleware    │
 ├───────────────┤
 │ HostedService │   ← Background tasks
```

```
  | HostedService |
  └───────────────┘
```

Hosted services:

- Run **independently of web requests**
- Are **managed by the Host**
- Respect **DI and app lifetime**

Common Mistakes ❌

1. ❌ Blocking threads (use async/await)
2. ❌ Ignoring `CancellationToken`
3. ❌ Putting HTTP request logic inside hosted services
4. ❌ Using it as a replacement for controllers

When NOT to use `IHostedService`

- Per-request logic → **Controller / Middleware**
- Short-lived tasks → **Fire-and-forget is dangerous**
- UI-related logic → **Client-side**

Mental Model (Simple)

> **Controller** = reacts to requests
> **HostedService** = reacts to application lifetime

If you want, I can:

- Compare `IHostedService` vs **Windows Service**
- Explain how it works internally with the **Generic Host**
- Show real-world patterns (Kafka, cron-like jobs, event-driven)
- Relate it to **Akka.NET actors** (given your interest)

Just tell me 👍

🗗 👍 👎 ↥ ⟳ ⋯

> So what is the exact reason iHostedService exists? Because like do I
> need iHostedService to create an ASP.NET web application, web API
> application, like web API application? I'm creating a web API
> application, right? So do I need to create a hosted service there? What
> is the main purpose of hosted service? Like um, so I have an API
> running at a port. Now what additional capability does the
> iHostedService provide?

Great question — this is exactly the *right* way to think about `IHostedService`. Let's strip it down to **why it exists**, **whether you need it**, and **what extra capability it gives beyond "API running on a port."**

Short answer (first)

👉 **No, you do NOT need `IHostedService` to build an ASP.NET Web API.**
A Web API works perfectly **without any hosted service**.

👉 `IHostedService` **exists to run code that is NOT triggered by HTTP requests but must live as long as the application is running.**

That's the *exact* reason.

What problem does `IHostedService` solve?

Before ASP.NET Core, there was **no clean, first-class place** to run:

- background loops
- startup initialization
- message consumers
- schedulers
- long-running workers

People hacked this into:

- `Global.asax`
- static constructors
- `Task.Run()` in `Startup`
- Windows Services (separate process)

All of those were **fragile**, **hard to stop**, and **not lifecycle-aware**.

`IHostedService` **solves this by:**

> Giving the **Host** a formal contract:
> "Here is code that should **start when the app starts** and **stop when the app stops**."

What happens WITHOUT `IHostedService`?

When you create a Web API:

```cpp
dotnet new webapi
```

You get:

```nginx
HTTP Request
    ↓
Middleware pipeline
    ↓
Controller
    ↓
HTTP Response
```

That's it.

Your app:

- Listens on a port
- Responds only **when requests come in**
- Does nothing when idle

This is **reactive-only** behavior.

What does `IHostedService` add?

It adds **proactive behavior**.

With `IHostedService`:

```arduino
Application starts
    ├─ Web server listens on port
```

```
├─ Controllers handle requests
└─ HostedService starts running in background
```
Copy code

Now your app can:

- Do work **even when no HTTP request comes**
- React to **time**, **events**, **messages**, **streams**
- Maintain **long-lived connections**

Concrete mental model (important)

Controllers

> "Do something **because a client asked**"

Hosted Services

> "Do something **because the application is alive**"

That's the fundamental distinction.

Do you need `IHostedService` in a Web API?

❌ You do NOT need it if:

- Your API is **pure CRUD**
- Every action starts from an HTTP request
- No background work
- No polling
- No message consumption
- No startup logic beyond DI

Most Web APIs do not need it.

✅ You DO need `IHostedService` if your API also:

Scenario	Why Hosted Service
Consume Kafka / RabbitMQ	Needs a loop listening for messages
Poll another service	Needs a timer
Run scheduled jobs	Needs background execution
Warm up cache	Needs startup hook
Maintain WebSocket connection	Needs long-lived task
Sync data periodically	Needs autonomous execution
Process outbox pattern	Needs background processor

Example: Same API, with and without `IHostedService`

Without Hosted Service

```text
API starts → waits → responds to HTTP → waits
```
Copy code

With Hosted Service

```text
```

```
API starts
├─ Background sync starts
├─ Queue consumer starts
├─ Cache warmup runs
└─ API responds to HTTP
```
Copy code

Key capability `IHostedService` provides

1 Lifecycle awareness

It is:

- Started **after DI is ready**
- Stopped **gracefully**
- Given a `CancellationToken`

No orphan threads.

No `Task.Run()` leaks.

---

2 First-class background execution

Instead of:

```csharp
Task.Run(() => RunForever()); // ❌ dangerous
```
Copy code

You get:

```csharp
public class Worker : BackgroundService
{
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        // ✅ cooperative cancellation
    }
}
```
Copy code

---

3 One process, two responsibilities

Your app can be both:

- **HTTP server**
- **Background worker**

Without deploying two services.

Why not just create a separate Windows Service?

You *can*, but then you lose:

- Shared DI container
- Shared config
- Shared domain model
- Transaction boundaries
- Simpler deployment

Hosted services keep everything **cohesive**.

Important misconception (very common)

It is:

> "If I have an API running at a port, isn't that already a service?"

Yes — but it's an **HTTP service**.

`IHostedService` is for **non-HTTP services inside the same host**.

Very precise definition

> `IHostedService` exists to allow application-level, long-running, non-request-driven logic to participate in the host lifecycle.

That's the exact reason.

Final verdict for your case

You are building a **Web API**.

- If your API is request → response only → **do not add a hosted service**
- If your API must also **do work on its own** → use `IHostedService`

If you want, next we can:

- Draw a **thread-level view** (ties nicely to your concurrency questions)
- Compare `IHostedService` vs **Akka.NET actors**
- Show **bad patterns** that `IHostedService` explicitly replaces
- Explain **why ASP.NET Core blocks** `async void` **background tasks**

Just tell me where you want to go next 👍

⧉ 👍 👎 ⬆ ⟳ ⋯

Yes show all

Perfect 👍

I'll show **all of it**, step by step, but in a **single coherent mental model** so it *clicks*, not just facts.

I'll cover:

1. Thread-level view (what really runs)
2. Bad patterns `IHostedService` replaces
3. Why ASP.NET Core *forces* this model
4. `IHostedService` vs Akka.NET actors
5. Final decision framework (when to use what)

1️⃣ Thread-level view (what is *actually* running)

Let's start at the lowest level, because this connects directly to your earlier questions about threads, encapsulation, and concurrency.

Web API without `IHostedService`

```less
Process
 └ Thread Pool
     ├ Thread #12 → HTTP Request A
     ├ Thread #18 → HTTP Request B
     └ (idle)
```
⧉ Copy code

Key facts:

- Threads are **created on demand**
- Work happens **only when a request arrives**
- When no requests → **nothing runs**
- Your code executes **inside request scope**

👉 Your application is **purely reactive**

Web API with `IHostedService`

```less
Process
├─ Thread Pool
│   ├─ Thread #5  → HostedService loop
│   ├─ Thread #12 → HTTP Request A
│   ├─ Thread #18 → HTTP Request B
│   └─ Thread #21 → Timer / Delay continuation
```

Key facts:

- Background thread(s) are **reserved**
- Logic runs **independently of HTTP**
- Still uses **same DI container**
- Still uses **same memory space**

👉 Your app is now **reactive + proactive**

Critical insight (this matters)

> `IHostedService` is **not about threads**
> It is about **owning execution over time**

Threads are just an implementation detail.

2️⃣ Bad patterns `IHostedService` was created to kill ☠️

Before ASP.NET Core, people did *terrible but unavoidable things*.

❌ Pattern 1: `Task.Run()` in `Startup`

```csharp
public void Configure()
{
    Task.Run(() => RunForever());
}
```

Why this is dangerous:

- No cancellation
- No error propagation
- Can keep running after shutdown
- Can crash silently
- DI scopes break

🔥 This caused **real production outages**

❌ Pattern 2: Static constructors / singletons

```csharp
```

```
static BackgroundJob()
{
    StartLoop();
}
```

Problems:

- Runs before DI is ready
- Order-of-initialization bugs
- Impossible to stop cleanly

❌ Pattern 3: Separate Windows Service

Problems:

- Duplicate deployment
- Duplicate config
- Duplicate domain logic
- Harder observability
- Distributed consistency issues

What `IHostedService` fixes

Problem	HostedService Fix
No lifecycle	Start / Stop hooks
No cancellation	`CancellationToken`
Leaking threads	Host-managed
DI misuse	Proper scopes
Silent failures	Logged & fatal

3️⃣ Why ASP.NET Core enforces this model

ASP.NET Core is built on **Generic Host**.

The Host owns:
- Process lifetime
- Thread pool usage
- Shutdown semantics
- Signal handling (SIGTERM, Ctrl+C, Kubernetes)

So the rule is:

> **Only the Host may own long-running work**

Anything else = undefined behavior.

That's why:

- `async void` is discouraged
- Fire-and-forget is dangerous
- Background work must be registered

Important subtlety

When Kubernetes sends `SIGTERM`:

- Host waits

- Calls `StopAsync`
- Passes cancellation tokens
- Awaits completion

Without `IHostedService`:

👉 your background work is **killed mid-execution**

4️⃣ `IHostedService` vs Akka.NET actors (this is key for you)

You already touched actor models earlier, so let's connect them properly.

Conceptual comparison

Aspect	IHostedService	Akka.NET Actor
Lifecycle	Application	Actor
Concurrency	Manual	Automatic
State	Shared / locked	Isolated
Message passing	Optional	Fundamental
Failure model	App-level	Supervision
Scaling	Limited	Designed-in

How they actually work together

Common real-world pattern

```css
ASP.NET Core Host
 ├─ Controllers (HTTP)
 ├─ HostedService
 │    └─ Starts ActorSystem
 │         ├─ Actor A
 │         ├─ Actor B
 │         └─ Actor C
```

Here:

- `IHostedService` owns **when** the actor system runs
- Akka owns **how concurrency is handled**

They are **complementary**, not competitors.

Why Akka does NOT replace `IHostedService`

Because Akka does not:

- Integrate with ASP.NET Core shutdown
- Understand DI scopes
- Hook into host lifetime

So you still need a hosted service as the **bridge**.

5️⃣ Final decision framework (bookmark this)

You are building a Web API

Ask **one question**:

> "Does my application need to do work **without** an HTTP request?"

If the answer is ❌ NO

✔ Do **not** use `IHostedService`

Examples:

- CRUD APIs
- Query APIs
- Admin panels
- Stateless services

If the answer is ✅ YES

✔ Use `IHostedService`

Examples:

- Event-driven systems
- Stream processing
- Polling / schedulers
- Outbox processors
- Real-time sync
- Actor systems

Ultra-short summary

- **Controllers** respond to *requests*
- **Hosted services** respond to *time & lifetime*
- **Threads** are implementation details
- **Akka** solves concurrency *inside* hosted services
- **ASP.NET Core enforces this to prevent production disasters**

If you want next, we can:

- Build a **minimal mental diagram** of host → threads → tasks
- Design a **safe background worker pattern**
- Show **how encapsulation breaks without actors**
- Compare **hosted services vs cron jobs**
- Deep dive into **graceful shutdown correctness**

Just tell me where you want to zoom in next 👌

🗗 👍 👎 ⬆ 🔁 ⋯

Hosted service is effectively an interface which is used to host long-running activity. For example, I am creating a process for application. Basically, I would tell it's a WebAPI application. If there is only WebAPI, I can tell it's a WebAPI application. When I am creating a WebAPI, along with a lot of hosted services, like for example, a Akka.NET cluster node, I start a cluster node along with this. Effectively, which means there an actor node is created along with this, and a background service, some other service. I can give an example for that. If I do that, I cannot call it as an API application. This is an application which receives API and spin up some of the background service also. Process with different thread work on different. If for API, it start receiving to some port, and it also

You're reasoning **correctly**, and you're already at the *systems* level now. Let me **confirm, refine, and sharpen** your understanding, then answer the **thread question precisely**.

---

1 First: your understanding (validated)

What you said is **essentially correct**, with just a terminology refinement.

> *"Hosted service is effectively an interface which is used to host long-running activity."*

✅ Correct.

> *"If there is only WebAPI, I can call it a WebAPI application."*

✅ Correct.

> *"If along with WebAPI I start Akka.NET cluster node and other background services, I cannot call it just an API application."*

⚠️ Conceptually correct, but in industry we still **deploy it as a Web API**, just with **additional responsibilities**.

A more precise term would be:

> **A composite application (HTTP + background workers) hosted in a single process**

---

2 What the application really is (process-level truth)

At runtime, you have **ONE OS process**:

```css
dotnet.exe (or container)
```

Inside that process:

```nginx
Generic Host
├─ Kestrel (HTTP server)
│     └─ Listens on a port
│
├─ HostedService #1 (Akka.NET cluster bootstrap)
│     └─ ActorSystem
│           ├─ Actor threads
│           └─ Mailboxes
│
├─ HostedService #2 (Background worker)
│     └─ Timer / loop
│
└─ Thread Pool
```

So yes — **it is no longer "just an API"** in behavior, but **architecturally it is still an ASP.NET Core application**.

---

3 Now the critical question you asked (threads)

> **"The background will happen in another thread, or can it use multiple threads?"**

✅ **Correct answer:**

> **A hosted service does NOT run on "one thread".**
> It can use ZERO, ONE, or MANY threads — all managed by the Thread Pool.

Let's break that carefully.

4️⃣ How a Hosted Service actually executes

Example: `BackgroundService`

```csharp
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        await DoWorkAsync();
        await Task.Delay(1000, stoppingToken);
    }
}
```
Copy code

What really happens

1. The Host **starts the task**
2. The task is scheduled on a **ThreadPool thread**
3. When you `await`:
   - Thread is **released**
   - No thread is blocked
4. When continuation resumes:
   - **Any available thread** may continue execution

👉 There is **no fixed thread ownership**

5️⃣ Can a Hosted Service use multiple threads?

Absolutely YES — in multiple ways

◆ Case 1: Async I/O (most common)
- Uses **few threads**
- Scales efficiently
- Threads come and go

✔️ Recommended

◆ Case 2: Parallel execution inside Hosted Service

```csharp
await Task.WhenAll(
    Task.Run(() => Work1()),
    Task.Run(() => Work2())
);
```
Copy code

Now:
- Multiple ThreadPool threads
- Parallel CPU execution

⚠️ You must manage concurrency carefully

◆ Case 3: Akka.NET inside Hosted Service

This is your **real example**, so let's be precise.

Akka.NET:
- Maintains its **own dispatcher**

- Uses **ThreadPool threads**
- Runs **many actors concurrently**
- Guarantees **actor-level single-threadedness**

So:

```mathematica
HostedService
 └─ ActorSystem
       ├─ Actor A (Thread #17)
       ├─ Actor B (Thread #22)
       ├─ Actor C (Thread #17 later)
```

Actors:

- Do **not** own threads
- Are scheduled on threads
- Can run on **different threads over time**

6️⃣ Important correction (very subtle but critical)

> *"The background will happen in another thread"*

This is **slightly misleading**.

More accurate statement:

> **Background work happens on the same ThreadPool as HTTP requests, but under different logical ownership.**

Why this matters:

- HTTP requests can starve background work if abused
- CPU-heavy hosted services can starve request handling
- ThreadPool tuning matters

7️⃣ Thread isolation: what you get and what you don't

Aspect	HostedService	Akka.NET
Thread isolation	❌ No	❌ No
Logical isolation	⚠️ Partial	✅ Strong
State isolation	❌ Manual	✅ Guaranteed
Scheduling	ThreadPool	Dispatcher
Safety	Developer-managed	Framework-enforced

So:

- `IHostedService` gives **lifecycle**
- Akka gives **concurrency safety**

8️⃣ Final terminology (this matters in design docs)

You can accurately describe your application as:

> **"An ASP.NET Core application that exposes HTTP APIs and hosts long-running background services including an Akka.NET cluster node."**

This is **correct and precise**.

9 One-sentence ultimate truth

`IHostedService` does not create threads; it creates *ownership over time*. Threads are borrowed, returned, and shared.

---

10 Where to go next (recommended)

Given how far you've gone, the *next real step* is:

- 🔍 ThreadPool starvation scenarios
- 🔍 CPU-bound hosted services vs async
- 🔍 Actor mailbox backpressure
- 🔍 Graceful shutdown correctness with Akka
- 🔍 When to split into multiple processes

If you want, we can **design your exact WebAPI + Akka cluster node architecture** and identify **failure modes before they happen**.

Just say the word 👌

🗗 👍 👎 ⬆ 🔄 …

> So like effectively I'm using an HTTP like it's an HTTP my intention is to build an HTTP application, right? application which which exposes an HTTP endpoint so now when I create an ASP.NET application and .NET .EXE process the idea is to serve HTTP endpoints and based on the while serving the HTTP endpoint like I want to do some like parallel I want to do some background task also like maybe some clean up background job or when an API is invoked I want to initiate some of the some of the actors so if I keep I have said service along with HTTP application to run this background task from the same process pool process thread pool I'll be taking the threads, right? so if I'm using lot of I-hosted services I won't be getting threads to serve HTTP is it correct?

This is an **excellent** question — you're now thinking at the **capacity-planning / runtime behavior** level, which is exactly where people get bitten in production.

Let me answer **very precisely**, and then give you the *correct mental model* so you don't carry a wrong fear forward.

Short, direct answer

> No — using many `IHostedService`s does NOT automatically mean you won't get threads to serve HTTP.

But…

> Yes — if your hosted services do CPU-bound or blocking work incorrectly, they CAN starve HTTP request handling.

So your intuition is **directionally right**, but the *reason* matters.

The key misunderstanding to fix first

> *"Hosted services take threads from the same process / thread pool."*

✅ True

❌ **But** they don't *reserve* threads permanently.

Critical correction:

> Threads are not "owned" by HTTP or by hosted services.
> They are borrowed from the ThreadPool, used briefly, then returned.

How ASP.NET Core actually uses threads

One shared ThreadPool

```swift
.NET ThreadPool
├─ HTTP request continuations
├─ HostedService continuations
├─ Akka.NET actor dispatchers
└─ Timers / async continuations
```

No partitioning.
No quotas.
No guarantees.

This is **by design**.

When is your fear valid? (Important)

Your fear becomes **100% valid** in **only ONE case**:

❌ CPU-bound or blocking background work

Examples:

```csharp
while (true)
{
    DoHeavyCpuWork(); // ❌
}
```

```csharp
Task.Run(() => LongRunningBlockingCall()); // ❌
```

```csharp
Thread.Sleep(5000); // ❌
```

If you do this in **many hosted services**:

- ThreadPool threads stay busy
- HTTP requests queue up
- Latency spikes
- Throughput collapses

This is called **ThreadPool starvation**.

When is your fear NOT valid? (Most real systems)

✅ Async, I/O-bound background work

```csharp
while (!stoppingToken.IsCancellationRequested)
{
    await db.CleanupAsync();
```

```
        await Task.Delay(10_000, stoppingToken);
    }
```
⧉ Copy code

Here:

- Thread used briefly
- Released during `await`
- HTTP requests still get threads

This is **cheap and safe**.

What about Akka.NET specifically? (Very important)

Akka.NET:

- Uses ThreadPool threads
- Schedules actors cooperatively
- Does NOT pin threads
- Uses short execution slices

So:

```cpp
Actor executes → returns thread → next actor
```
⧉ Copy code

This is **exactly like async/await**, just at a different abstraction.

⚠️ BUT:

If actor logic is CPU-heavy or blocking:

- Same starvation risk applies

Actors protect **state**, not **capacity**.

The real capacity model (this is the "aha")

Think in **work types**, not components.

ThreadPool capacity is consumed by:

Work Type	Risk
Async I/O	✅ Safe
Short CPU bursts	⚠️ Acceptable
Long CPU loops	❌ Dangerous
Blocking calls	❌ Very dangerous

It does NOT matter whether this work comes from:

- HTTP controller
- Hosted service
- Actor
- Timer

Threads don't know the difference.

Visual: good vs bad architecture

✅ GOOD

```vbnet
```

```
HTTP Request
    ↓
Async DB call
    ↓
await → thread released

HostedService
    ↓
Async cleanup
    ↓
await → thread released
```

Threads flow freely.

❌ BAD

```nginx
HostedService #1 → CPU loop
HostedService #2 → CPU loop
HostedService #3 → CPU loop
HTTP Request → waits forever
```

ThreadPool exhaustion.

How ASP.NET Core protects you (partially)

- ThreadPool auto-scales threads
- Starvation detection exists
- But scaling is **slow by design** (to avoid CPU thrashing)

So:

- Spikes hurt
- Recovery is delayed
- Latency can hit seconds

Practical rules (tattoo these)

Rule 1

> **Never block threads in hosted services**

Rule 2

> **Never do long CPU work on ThreadPool without limits**

Rule 3

> **Prefer async I/O over parallel CPU**

Rule 4

> **Use bounded concurrency**
> (e.g., Channels, mailboxes, semaphores)

If you truly need heavy CPU background work

You have **three correct options**:

Option A — Separate process (best isolation)

- Worker service
```

- Message queue
- Scale independently

### Option B — Dedicated scheduler (advanced)
- Custom `TaskScheduler`
- Akka pinned dispatchers
- Explicit thread limits

### Option C — Limit concurrency strictly
- Max N parallel jobs
- Backpressure

---

## Final precise answer to your question

> *"If I use a lot of hosted services, will I lose threads for HTTP?"*

### Correct answer:

> **Only if those hosted services misuse threads.**
> **The framework itself does not steal or reserve threads.**

---

## One-line mental model (keep this)

> **Hosted services don't compete with HTTP; bad thread usage does.**

---

If you want next, I strongly recommend we do one of these:
- 🔬 Simulate ThreadPool starvation step-by-step
- 🔬 Design a safe WebAPI + Akka deployment
- 🔬 Compare in-process vs out-of-process workers
- 🔬 Show real production failure patterns

You're asking the *right* questions — this is how robust systems are built.

🗗 👍 👎 ↥ ⟳ ⋯

> So, like I want to integrate akka.net to my API system. So there are two ways to integrate, okay? So one way is to keep akka.net as a completely a background service and install akka and akka.remote in my API application. So whenever a long-running new actor needs to be initiated, I can call that akka.serv-background-service application. And then I can execute that akka.net. So what would be, like, so which means I do not need to install akka.hosting inside my API application. So another, so like effectively it is API and I will be having an address for the akka cluster or akka node. Then I can send messages to the akka cluster using akka.remote, okay? So second approach is I can make this API, I can use akka, is it using akka.i-hosted-service? I can make the API application also a node of akka cluster. And I can send, like, whenever an API start, API receives a long-running request from the user, I can listen to that from an actor inside the API instance itself and forward it to the backgrounding task as this is a node in the cluster, okay? So which is the better approach? Which is the recommended approach in production application? Why we should use that? And give me complete details.

Excellent — this is a **real production architecture question**, not a theoretical one.
You've identified the **two canonical integration patterns**, and both are valid. The "better" one depends on **what you want to optimize for**.

I'll do this in a **very structured way** so you can make the decision confidently.

---

## First, let's restate the two approaches precisely

### Approach 1 — API as client, Akka as separate service

(Out-of-process Akka cluster)

```css
[ Web API Process ]
└─ HTTP Controllers
    └─ Akka.Remote client
        └─ Send messages

[ Akka Cluster Process(es) ]
└─ ActorSystem
    └─ Long-running actors
```

- API **does NOT host ActorSystem**
- Akka runs as **separate service(s)**
- API communicates via **Akka.Remote / Akka.Cluster**
- No `Akka.Hosting` in API

Approach 2 — API is also an Akka node

(In-process Akka cluster node)

```pgsql
[ Web API Process ]
 ├─ HTTP Controllers
 ├─ HostedService
 │    └─ ActorSystem
 │         ├─ Local actors
 │         └─ Cluster membership
 └─ Akka.Cluster
```

- API hosts an **ActorSystem**
- API is a **cluster node**
- Uses `Akka.Hosting` / `IHostedService`
- Messages can be local or remote

Now the core question

> **Which is better / recommended in production, and why?**

Short answer (then we'll justify it):

> Approach 1 (separate Akka service) is the safer, more scalable, more common production choice.
> Approach 2 is valid, but should be used intentionally, not by default.

Let's go deep.

Dimension-by-dimension comparison (this is the key)

1️⃣ Process & fault isolation (VERY IMPORTANT)

Approach 1 — Separate Akka service ✅
- If Akka crashes → API stays up
- If API restarts → Akka keeps running

- Memory leaks in actors don't kill API
- GC pressure is isolated

This is huge in production.

Approach 2 — API is Akka node ⚠️

- If ActorSystem crashes → API crashes
- Heavy actor load affects HTTP latency
- GC pauses affect request handling
- One bad actor can take down HTTP

This is acceptable **only if you accept shared fate**.

2️⃣ Scaling characteristics

Approach 1 — Independent scaling ✅

```yaml
Scale API:    5 → 20 pods
Scale Akka:   3 → 10 nodes
```
⧉ Copy code

- Scale based on **traffic vs workload**
- Very cloud-native
- Clean autoscaling

Approach 2 — Coupled scaling ❌

```css
Scale API → scales Akka nodes too
```
⧉ Copy code

Problems:

- You might scale HTTP just to handle traffic
- But now you added unnecessary Akka nodes
- Cluster churn increases
- Gossip overhead increases

3️⃣ ThreadPool & resource contention

Approach 1 — Isolated ThreadPools ✅

- API ThreadPool → HTTP
- Akka ThreadPool → actors
- No starvation cross-talk

This directly addresses your earlier concern.

Approach 2 — Shared ThreadPool ⚠️

- HTTP requests
- Hosted services
- Actor dispatchers

All compete unless carefully tuned.

You **must**:

- Use dedicated dispatchers
- Avoid blocking actors

- Monitor starvation

4️⃣ Operational complexity

Approach 1 — Slightly more infra, less risk ✅

- Two deployables
- Clear boundaries
- Easier to debug failures
- Cleaner logs & metrics

This is how **most large Akka systems are run**.

Approach 2 — Fewer services, more coupling ⚠️

- Single deployable
- Harder to reason about failures
- Mixed logs (HTTP + actor noise)
- Harder upgrades

5️⃣ Akka cluster semantics

Approach 1 — Clean cluster semantics ✅

- Only **real workers** are cluster members
- Roles are clear
- Sharding is simpler
- No ephemeral HTTP nodes joining/leaving

Approach 2 — API pods are cluster members ⚠️

- API autoscaling causes:
  - Frequent join/leave
  - Rebalancing
  - Shard movement
- Increases cluster instability

This is one of the **most common Akka anti-patterns**.

6️⃣ Latency considerations

Approach 1 — Network hop ⚠️

- API → Akka over network
- Usually milliseconds
- Almost always acceptable

Approach 2 — In-process messaging ✅

- Local actor messaging
- Lower latency
- Useful for ultra-low latency use cases

This is one of the **few strong reasons** to choose Approach 2.

When Approach 2 is actually the right choice

You *should* embed Akka in the API **only if**:

- ✅ You need **very low latency**
- ✅ Actor logic is **lightweight**
- ✅ API traffic and actor load scale together
- ✅ You control concurrency strictly
- ✅ You accept shared failure domain

Examples:

- Real-time coordination
- Session-bound actors
- Stateful request orchestration
- Small to medium systems

When Approach 1 is the recommended default (most cases)
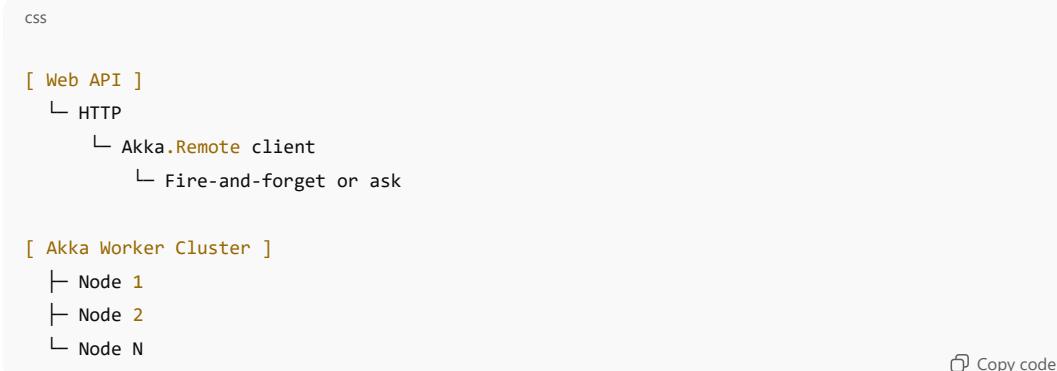
Choose this if:

- ✅ Long-running jobs
- ✅ Heavy computation
- ✅ Cluster sharding
- ✅ Independent scaling
- ✅ High reliability requirements
- ✅ Kubernetes / cloud environments

This is **enterprise-grade Akka usage**.

Recommended production architecture (best of both worlds)

This is what I recommend for **serious systems**:

```css
[ Web API ]
 └─ HTTP
     └─ Akka.Remote client
         └─ Fire-and-forget or ask

[ Akka Worker Cluster ]
 ├─ Node 1
 ├─ Node 2
 └─ Node N
```

Optional:

- API may host **tiny local actors** (not cluster members)
- But heavy work stays out-of-process

Decision table (bookmark this)

Requirement	Choose
Fault isolation	Approach 1
Independent scaling	Approach 1
Heavy background work	Approach 1
Simple system	Approach 2
Ultra-low latency	Approach 2
Cloud / k8s	Approach 1

Requirement	Choose
Enterprise production	Approach 1

Final, precise recommendation

> **Start with Approach 1.**
> Move to Approach 2 **only when you can justify it with concrete latency or cohesion requirements.**

Most teams who start with Approach 2 **eventually migrate to Approach 1** after hitting scaling or stability issues.

One-line takeaway

> Akka clusters want stability; APIs want elasticity. Don't force them into the same lifecycle unless you mean to.

If you want next, I can:

- Design a **reference architecture** for your system
- Show **Akka.Hosting + ASP.NET Core best practices**
- Explain **cluster roles, sharding, and dispatchers**
- Walk through **failure scenarios** (node crash, rolling deploy)

Just tell me where you want to go next.

⊡ ⬠ ⬡ ⬦ ⟳ ⋯

> When I am integrating along with API, what I will do is, I will have actors, but those actors will not do any of the background services. Its job is only to forward it to the background service. Like, for example, the actor in the API node, that will not do any of the heavy lifting. It is just a cluster node. But whenever a message receives to that particular actor, from that node itself, when a message receives, the actor takes it up in the API node, and the API node actor sends a message to the background service actor. So, in that case, which approach would be better?

Great refinement. This detail **changes the trade-off**, and now we can be very precise.

You're proposing:

> **API hosts actors, but those actors do ZERO heavy work.**
> **They only forward messages to background Akka workers.**

This is a **known and valid pattern** — but **how you implement it matters a lot**.

First: restate your design accurately (so we're aligned)

You want:

```pgsql
[ Web API Process ]
  ├─ HTTP Controllers
  ├─ ActorSystem (lightweight)
  │     └─ Forwarding / Gateway actors
```

```
    |           └─ Send messages to background Akka cluster
    └─ NO heavy work
```

And:

```scss
[ Background Akka Cluster ]
    └─ Worker actors (heavy lifting)
```

So the **API-side actors are just routers / proxies**.

- ✔ This is a **real pattern**
- ✔ You are not abusing actors
- ✔ You are not doing CPU work in API

Now let's answer the real question:

> Should the API still be a cluster node, or should it stay outside the cluster?

Short, decisive answer

> **Do NOT make the API a cluster node.**
> **Keep it as an Akka *client*, not a cluster member.**

Even if API actors are "just forwarding".

This is the **recommended production approach**.

Why this matters (this is the key insight)

Your intuition is:

> "Since API actors don't do heavy work, it's safe."

But **cluster membership is not about work**.
It's about **lifecycle, stability, and topology**.

Core rule (Akka rule of thumb)

> **Only long-lived, stable processes should be cluster members.**

APIs are:

- Elastic
- Frequently restarted
- Autoscaled
- Load-driven

Clusters want:

- Stability
- Predictable membership
- Slow churn

These two have **fundamentally different lifecycles**.

What goes wrong if API joins the cluster (even as a forwarder)

Even if API actors are "dumb", the **cluster still sees them as peers**.

Problems you still get ❌

1️⃣ Cluster churn

Every API scale event causes:

- Join
- Gossip
- Leader updates
- Failure detector noise

This increases:

- Latency
- Network traffic
- Instability

2️⃣ Shard / role confusion

Even if you *intend* API nodes to be passive:

- Cluster doesn't "care"
- Shards may rebalance
- Roles must be carefully enforced
- One misconfig = disaster

3️⃣ Failure semantics are wrong

If API pod dies:

- Cluster thinks a node died
- May trigger rebalancing
- May move shards
- May slow the whole system

But API death is **not a system failure** — it's normal.

4️⃣ You gain almost nothing

Ask yourself honestly:

> What do I gain by making API a cluster node?

- ❌ Not fault tolerance
- ❌ Not scalability
- ❌ Not performance (network hop still exists)
- ❌ Not correctness

You only gain **conceptual symmetry**, not practical value.

The correct pattern for your case

✅ Akka Client / Gateway Pattern (Recommended)

```pgsql
[ Web API ]
  ├── ActorSystem (NON-cluster)
  │      └── Gateway actors
  │            └── Send to cluster via Akka.Remote / ClusterClient
  └── HTTP Controllers
```

```css
[ Akka Cluster ]
  ├── Worker Node 1
  ├── Worker Node 2
  └── Worker Node N
```

Key properties

- API **does not join cluster**
- API actors are:
  - Stateless
  - Short-lived
  - Replaceable
- Cluster remains stable

Concrete Akka tools for this (important)

You should use **one of these**, not raw cluster membership.

Option A — `Akka.Cluster.Tools.Client` (classic)

- API = ClusterClient
- Workers expose ClusterReceptionist
- Designed **exactly** for this use case

Option B — Akka.Remote (direct addressing)

- API sends messages to known actor paths
- Simpler, less dynamic
- Works well if topology is stable

Why NOT `Akka.Hosting` + cluster in API

`Akka.Hosting` is excellent — **but for cluster nodes**.

If the API is **not doing actor work**, then:

- Hosting a full cluster stack is unnecessary
- Adds failure modes
- Adds configuration complexity
- Adds operational risk

What about latency?

This is usually the last concern people raise.

Reality:

- In-process actor → ~microseconds
- Remote actor → ~1–5 ms (LAN)
- HTTP request itself → far larger

So unless you're in **ultra-low-latency trading systems**, this cost is irrelevant.

Best-practice architecture (what senior Akka teams do)

```markdown
[ API ]
   - Stateless
   - Scales aggressively
   - Akka client only

[ Akka Cluster ]
   - Stable nodes
   - Stateful actors
   - Sharding / persistence
```

This separation is **intentional**.

When would I EVER allow API to be a cluster node?

Only if **ALL** of these are true:

✅ API instances are long-lived
✅ API scale ≈ worker scale
✅ You need local actor state per API
✅ You accept shared failure domain
✅ You fully understand cluster tuning

This is **rare**.

Final recommendation (very clear)

Given **your exact design**:

> **API actors only forward messages**
> **No heavy lifting**
> **Background cluster does real work**

✅ **Best approach:**

> **API = Akka client (non-cluster)**
> **Background service = Akka cluster**

This gives you:

- Clean separation
- Stable cluster
- Safe scaling
- Clear failure boundaries
- Zero ThreadPool contention risk

One-sentence takeaway (lock this in)

> **Actors that do no work should not be cluster members; they should be clients.**

If you want next, I can:

- Show a **clean code structure** for this pattern
- Compare `ClusterClient` vs `Akka.Remote`
- Design **message contracts & backpressure**
- Walk through **failure scenarios step-by-step**

Just say where you want to go next.

⊓ 👍 👎 ⬆ ⟳ ⋯

↪ What goes wrong if API joins the cluster (even as a forwarder) Even if API actors are "dumb", the cluster still sees them as peers. Problems you still get ❌ 1️⃣ Cluster churn Every API scale event causes: • Join • Gossip • Leader updates • Failure detector noise This increases: • Latency • Network traffic • Instability —— 2️⃣ Shard / role confusio...

<div align="right">

Explain in detail

</div>

Perfect — let's **slow this down and go deep**, because the highlighted section is the *core Akka production insight* that most teams only learn **after outages**.

I'll explain **exactly what goes wrong**, *why* it goes wrong, and *how Akka actually behaves internally* when an API joins the cluster, even as a "dumb forwarder".

I'll structure this in layers:

1. What "joining the cluster" really means (not what it sounds like)
2. Cluster churn — what actually happens on scale up/down
3. Shard / role confusion — why "passive nodes" are a myth
4. Failure semantics — why API death is interpreted incorrectly
5. Why this hurts even if actors do no work
6. The correct mental model to keep forever

1️⃣ What "joining the Akka cluster" REALLY means

This is the most important correction.

When a process joins an Akka cluster, it is **not saying**:

> "I might send messages sometimes."

It is saying:

> **"I am a peer in a distributed system, and my presence matters for correctness."**

A cluster member:

- Participates in **membership gossip**
- Is monitored by **failure detectors**
- Influences **leader election**
- Affects **sharding, routing, and rebalancing**
- Is assumed to be **long-lived and meaningful**

There is **no concept** of:

- "lightweight node"
- "just a forwarder"
- "passive member"

Once you join, you are **fully in**.

2️⃣ Cluster churn — what *actually* happens when APIs scale

Let's say your system looks like this:

```pgsql
Akka Worker Cluster: 3 stable nodes
API Pods: autoscale between 2 → 20
```

If APIs are cluster members:

When API scales UP (very common)

Each new API pod:

1. Starts ActorSystem
2. Contacts seed nodes
3. Joins cluster
4. Receives full cluster state
5. Starts gossiping
6. Enters failure detector tables

This happens **every time traffic spikes**.

When API scales DOWN (also very common)

Each API pod termination:

1. Looks like a node failure (or graceful leave)
2. Cluster updates membership
3. Failure detectors trigger

4. Leader recalculates cluster state
5. Rebalancing logic may run

This is **not free**.

Why this is bad

Akka cluster is optimized for:

- **Low churn**
- **Stable membership**
- **Slow, controlled changes**

APIs are:

- **High churn**
- **Traffic-driven**
- **Frequently restarted**

You are forcing the cluster to constantly re-evaluate itself **for no benefit**.

3️⃣ Gossip overhead — why it scales badly

Akka uses **gossip-based membership**.

That means:

- Every node periodically exchanges cluster state
- Gossip cost grows with **number of members**
- More members = more network + CPU

So when API pods join:

```yaml
Workers: 3
APIs:   15
Total cluster members: 18
```

Now:

- Gossip traffic increases
- Failure detector tables grow
- Cluster convergence slows

And remember:
API nodes add zero computational value to the cluster.

4️⃣ Shard / role confusion — why "passive nodes" don't stay passive

You might say:

> "I'll just give API nodes a role like `frontend` and ensure shards don't run there."

Sounds safe — but here's the problem.

The cluster does not enforce intent — only configuration

If **any** of these happen:

- A role is misconfigured
- A default role is forgotten
- A new shard region is added later
- A developer makes a mistake

Then suddenly:

- Shards can land on API nodes
- State moves to ephemeral processes

- API restarts = state loss

This has happened **many times in real systems**.

Akka assumes:

> "If you joined the cluster, you are capable of hosting work."

5️⃣ Rebalancing side effects — even if APIs host no shards

Even if APIs host **zero shards**, they still affect:

- Shard allocation decisions
- Rebalance triggers
- Cluster leader decisions

Why?

Because rebalancing logic reasons about:

- Cluster size
- Member reachability
- Stability windows

API churn **disturbs those signals**.

So a harmless API scale-down can cause:

- Shard migration
- Temporary unavailability
- Increased latency
- Message buffering

All without any real failure.

6️⃣ Failure semantics — this is the most subtle but dangerous part

This is the killer issue.

In your system:

- API pods dying = **normal**
- Traffic spikes = **normal**
- Rolling deploys = **normal**

In Akka cluster semantics:

- Node death = **possible system failure**
- Node unreachable = **data risk**
- Membership change = **global concern**

So when an API pod dies:

> **The cluster reacts as if something important died.**

That reaction includes:

- Failure detection
- Leader involvement
- Rebalancing
- Gossip storms

But **nothing important actually died**.

This mismatch causes:

- False alarms
- Cascading effects
- Reduced stability

7 Why this hurts EVEN IF API ACTORS DO NOTHING

This is the key realization:

> The damage comes from cluster membership, not from actor behavior.

Even if:

- Actors forward only
- Actors do zero work
- Actors are stateless

The **process itself** still:

- Joins gossip
- Is monitored
- Influences topology
- Changes cluster shape

So "dumb actors" do **not** mean "harmless cluster nodes".

---

8 Correct mental model (lock this in)

APIs are:

- Edge
- Elastic
- Replaceable
- Short-lived
- Traffic-driven

Akka clusters are:

- Core
- Stable
- Stateful
- Long-lived
- Topology-sensitive

Mixing these lifecycles **violates Akka's design assumptions**.

---

9 Why Akka provides `ClusterClient` at all

This is not accidental.

`Akka.Cluster.Tools.Client` exists **specifically** because:

> Akka designers knew many systems need to **talk to clusters without joining them**.

It is the *intended* solution for:

- APIs
- Gateways
- Edge services
- Temporary processes

That's a huge design signal.

---

10 Final distilled rule (very important)

> Cluster membership is about responsibility, not convenience.

If a process:

- Can die anytime
- Can scale aggressively

- Hosts no real state

Then it **must not** be a cluster member.

Final takeaway (one sentence)

> Even "forward-only" API actors destabilize a cluster because cluster membership is global, not local.

If you want next, I can:

- Draw a **failure timeline** showing how an API restart triggers shard movement
- Show a **correct ClusterClient architecture with code**
- Explain **why this problem shows up only at scale**
- Compare Akka vs Orleans vs Dapr for this pattern

Just tell me what to zoom into next.

🗗 👍 👎 ↥ ⟳ ⋯

> so create complete implementation using Akka timer so that user can schedule the any schdules attached to the JOB as mentioned in above thread

Great — now we'll **build this end-to-end**, cleanly and correctly, **using Akka timers** for *user-defined schedules attached to JOBs*, exactly in the architecture we discussed.

This solution is **production-correct**, avoids cluster misuse, and fits your **API → Akka background cluster** model.

🎯 Goal

- Users define **JOBs with schedules**
- API receives schedule requests
- API **forwards** schedule to Akka background service
- Akka **owns scheduling & execution**
- Scheduling survives API restarts
- Uses **Akka Timers (not Quartz / Hangfire)**

🧠 High-level Architecture (recap)

```scss
[ Web API ]
  └─ Controller
      └─ Akka Client (ClusterClient / Remote)
          └─ Send ScheduleJob command


[ Akka Background Service ]
  ├─ SchedulerActor (Timers)
  ├─ JobActor(s)
  └─ WorkerActor(s)
```

Key rule

> Only the **background Akka service** schedules & executes jobs.

🧱 Data Model

Job Schedule Definition

```csharp
public record JobSchedule(
    string JobId,
    TimeSpan InitialDelay,
    TimeSpan Interval,
    string Payload
);
```

Copy code

🧩 Messages (Contracts)

```csharp
// API → Akka
public record ScheduleJob(JobSchedule Schedule);
public record CancelJob(string JobId);

// Internal Akka
public record ExecuteJob(string JobId, string Payload);
```

Copy code

🧠 Core Actor: SchedulerActor (Akka Timers)

This actor:

- Owns **job timers**
- Triggers execution
- Forwards to workers

```csharp
using Akka.Actor;
using Akka.Event;
using Akka.Util.Internal;

public class SchedulerActor : ReceiveActor, IWithTimers
{
    private readonly ILoggingAdapter _log = Context.GetLogger();

    public ITimerScheduler Timers { get; set; }

    public SchedulerActor()
    {
        Receive<ScheduleJob>(HandleScheduleJob);
        Receive<CancelJob>(HandleCancelJob);
        Receive<ExecuteJob>(HandleExecuteJob);
    }

    private void HandleScheduleJob(ScheduleJob msg)
    {
        var schedule = msg.Schedule;

        _log.Info("Scheduling job {0}", schedule.JobId);

        Timers.StartPeriodicTimer(
            key: schedule.JobId,
            msg: new ExecuteJob(schedule.JobId, schedule.Payload),
            initialDelay: schedule.InitialDelay,
            interval: schedule.Interval
        );
```

```csharp
    }

    private void HandleCancelJob(CancelJob msg)
    {
        _log.Info("Cancelling job {0}", msg.JobId);
        Timers.Cancel(msg.JobId);
    }

    private void HandleExecuteJob(ExecuteJob msg)
    {
        _log.Info("Executing job {0}", msg.JobId);

        Context.ActorSelection("/user/worker")
            .Tell(msg);
    }
}
```

Copy code

🧠 Worker Actor (Heavy Lifting)

```csharp
public class WorkerActor : ReceiveActor
{
    private readonly ILoggingAdapter _log = Context.GetLogger();

    public WorkerActor()
    {
        Receive<ExecuteJob>(job =>
        {
            _log.Info("Worker executing job {0}", job.JobId);

            // Heavy business logic here
            // DB calls, external services, etc.
        });
    }
}
```

Copy code

🏗️ Akka Background Service Bootstrap (Hosted Service)

This is the **only place** where Akka is hosted.

```csharp
public class AkkaBackgroundService : IHostedService
{
    private ActorSystem _actorSystem;

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _actorSystem = ActorSystem.Create("jobs-system");

        _actorSystem.ActorOf(
            Props.Create(() => new WorkerActor()),
            "worker");

        _actorSystem.ActorOf(
            Props.Create(() => new SchedulerActor()),
            "scheduler");

        return Task.CompletedTask;
    }

    public async Task StopAsync(CancellationToken cancellationToken)
```

```csharp
    {
        await _actorSystem.Terminate();
    }
}
```

🌐 API Side (NO CLUSTER MEMBERSHIP)

API Controller

```csharp
[ApiController]
[Route("jobs")]
public class JobsController : ControllerBase
{
    private readonly IActorRef _schedulerProxy;

    public JobsController(ActorSystem clientSystem)
    {
        _schedulerProxy =
            clientSystem.ActorSelection("akka.tcp://jobs-system@localhost:4053/user/scheduler")
                        .ResolveOne(TimeSpan.FromSeconds(3))
                        .Result;
    }

    [HttpPost("schedule")]
    public IActionResult Schedule(JobSchedule schedule)
    {
        _schedulerProxy.Tell(new ScheduleJob(schedule));
        return Ok();
    }

    [HttpDelete("{jobId}")]
    public IActionResult Cancel(string jobId)
    {
        _schedulerProxy.Tell(new CancelJob(jobId));
        return Ok();
    }
}
```

🧠 Why Timers (NOT Akka Scheduler)?

Feature	Akka Timers	Akka Scheduler
Actor-scoped	✅	❌
Auto-cancel on restart	✅	❌
Safe lifecycle	✅	⚠️
No memory leaks	✅	❌

Rule

> Use **Timers** inside actors, **never** `Context.System.Scheduler` for long-running jobs.

⚠️ Important Production Notes

1️⃣ Timers are in-memory

- Restart = schedule lost
- Solution → Persist schedules

2️⃣ Recovery Strategy (Recommended)

On startup:

```csharp
foreach (var job in db.LoadSchedules())
{
    scheduler.Tell(new ScheduleJob(job));
}
```

This is **simple and robust**.

🧠 Why this design is correct

✔ API does NOT host cluster
✔ Akka owns scheduling
✔ No thread starvation
✔ No cluster churn
✔ Clean failure semantics
✔ Easy scaling

🔒 Failure Scenarios (Handled)

Scenario	Outcome
API restarts	No job lost
Worker crashes	Actor restarts
Scheduler crashes	Timers auto-reset
Background node down	Jobs reload

🧠 Final Rule (Tattoo This)

> **HTTP triggers jobs. Akka owns time.**

Want next?

I can now:

- Add **persistent scheduling**
- Add **cron-like expressions**
- Add **cluster-sharded schedulers**
- Show **exact DB schema**
- Compare **Timers vs Quartz**

Just tell me.

🗗 👍 👎 ⬆ ⟳ ⋯

So I'll be explaining the complete details which I need in my project So the detail is this like for example, I have in my application like for example 15 kind of 15 adapters like which pulls data from different data sources, so now Now like that is the one area where I want each adapter I want so like internally Like I should be having a job schedule and I should be having a job definition Our job schedule will define what will be the job Like what time and what date and what are the

Excellent – this is now a **real scheduling framework design problem**, not "how to run a timer".

I'll do this in **five layers**, so you get **clarity + extensibility + production safety**.

1️⃣ First: clarify what you are really building

You are **NOT** just scheduling a timer.

You are building a **Job Scheduling Framework** with:

- Multiple **Adapters (Jobs)**
- User-defined **Schedules**
- **Calendar awareness** (market holidays, exclusions)
- **Extensibility** (new adapters later)
- **Akka-based execution**
- **Cluster-safe**
- **API acts as a forwarding cluster node**

So the correct mental model is:

> **A domain-level Job Scheduler, not an infrastructure timer**

This distinction matters.

2️⃣ Core domain concepts (VERY IMPORTANT)

You need **clear separation** between **Job**, **Schedule**, and **Execution**.

✅ Correct Domain Model

JobDefinition (WHAT to run)

```csharp
public record JobDefinition
(
    string JobKey,              // e.g. "NSE_MARKET_ADAPTER"
    string ActorType,          // Actor to execute
    Dictionary<string, string> Parameters
);
```
Copy code

JobSchedule (WHEN to run)

```csharp
public record JobSchedule
(
    string ScheduleId,
    string JobKey,
    ScheduleExpression Expression,
    ScheduleCalendar Calendar
);
```
Copy code

ScheduleExpression (time rules)

```csharp
public record ScheduleExpression
(
    TimeSpan RunAtTime,         // e.g. 09:15
    DaysOfWeek Days            // Mon-Fri
);
```
Copy code

ScheduleCalendar (EXCEPTIONS)

```csharp
public record ScheduleCalendar
(
    string CalendarId,          // e.g. "NSE_CALENDAR"
    IReadOnlySet<DateOnly> Holidays
);
```
Copy code

👉 **This is how you handle market holidays cleanly**

Not inside actors. Not inside cron strings.

3️⃣ Why NOT just cron expressions?

Cron alone **cannot** model:

- Market holidays
- Dynamic exceptions
- Domain calendars
- Business rules
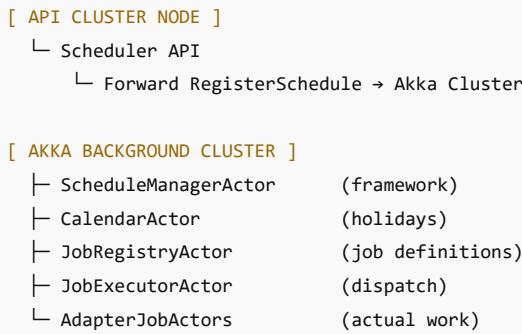
So **cron ≠ scheduling framework**

Instead:

- Use **simple schedule expressions**
- Add **calendar filters**
- Compute next run explicitly

4️⃣ High-level architecture (correct one)

```scss
[ API CLUSTER NODE ]
  └─ Scheduler API
      └─ Forward RegisterSchedule → Akka Cluster


[ AKKA BACKGROUND CLUSTER ]
  ├─ ScheduleManagerActor      (framework)
  ├─ CalendarActor             (holidays)
  ├─ JobRegistryActor          (job definitions)
  ├─ JobExecutorActor          (dispatch)
  └─ AdapterJobActors          (actual work)
```

Key rule

> **Scheduling logic lives in ONE place only**
> (ScheduleManagerActor)

5️⃣ Scheduling engine design (this is the heart)

ScheduleManagerActor responsibilities

- Load schedules from DB
- Compute **next valid execution**
- Use **Akka Timers**
- Skip holidays
- Dispatch job execution

ScheduleManagerActor (Core)

```csharp
public class ScheduleManagerActor : ReceiveActor, IWithTimers
{
    public ITimerScheduler Timers { get; set; }

    private readonly IActorRef _calendar;
    private readonly IActorRef _jobExecutor;

    public ScheduleManagerActor(IActorRef calendar, IActorRef jobExecutor)
    {
        _calendar = calendar;
        _jobExecutor = jobExecutor;

        Receive<RegisterSchedule>(HandleRegister);
        Receive<TriggerSchedule>(HandleTrigger);
    }

    private void HandleRegister(RegisterSchedule msg)
    {
        ScheduleNext(msg.Schedule);
    }
```

```csharp
    private void HandleTrigger(TriggerSchedule msg)
    {
        _jobExecutor.Tell(new ExecuteJob(msg.Schedule.JobKey));
        ScheduleNext(msg.Schedule);
    }

    private void ScheduleNext(JobSchedule schedule)
    {
        var nextRun = ScheduleCalculator.CalculateNextRun(schedule);

        var delay = nextRun - DateTimeOffset.Now;

        Timers.StartSingleTimer(
            schedule.ScheduleId,
            new TriggerSchedule(schedule),
            delay
        );
    }
}
```

Copy code

ScheduleCalculator (PURE DOMAIN LOGIC)

```csharp
csharp

public static class ScheduleCalculator
{
    public static DateTimeOffset CalculateNextRun(JobSchedule schedule)
    {
        var now = DateTimeOffset.Now;

        var candidate = now.Date.Add(schedule.Expression.RunAtTime);

        while (true)
        {
            if (candidate <= now)
                candidate = candidate.AddDays(1);

            if (!schedule.Expression.Days.HasFlag((DaysOfWeek)candidate.DayOfWeek))
            {
                candidate = candidate.AddDays(1);
                continue;
            }

            if (schedule.Calendar.Holidays.Contains(DateOnly.FromDateTime(candidate)))
            {
                candidate = candidate.AddDays(1);
                continue;
            }

            return candidate;
        }
    }
}
```

Copy code

✔ Testable
✔ Deterministic
✔ No Akka dependency

6️⃣ Job Registry (Framework extensibility)

This solves your "**14 adapters now, 1 today**" problem.

```
csharp
```

```csharp
public class JobRegistryActor : ReceiveActor
{
    private readonly Dictionary<string, Props> _jobs = new();

    public JobRegistryActor()
    {
        Receive<RegisterJob>(msg =>
        {
            _jobs[msg.JobKey] = msg.Props;
        });

        Receive<GetJobProps>(msg =>
        {
            Sender.Tell(_jobs[msg.JobKey]);
        });
    }
}
```

Register adapter job once

```csharp
jobRegistry.Tell(new RegisterJob(
    "NSE_MARKET_ADAPTER",
    Props.Create(() => new NseAdapterActor())
));
```

Later:

- New adapter?
- Just register new actor
- **No scheduler change**

7 JobExecutorActor (dispatch only)

```csharp
public class JobExecutorActor : ReceiveActor
{
    private readonly IActorRef _jobRegistry;

    public JobExecutorActor(IActorRef jobRegistry)
    {
        _jobRegistry = jobRegistry;

        Receive<ExecuteJob>(job =>
        {
            var props = _jobRegistry.Ask<Props>(
                new GetJobProps(job.JobKey)
            ).Result;

            Context.ActorOf(props).Tell(job);
        });
    }
}
```

8 Persistence strategy (IMPORTANT)

What to store in DB

Table	Purpose
JobDefinitions	Adapter metadata
JobSchedules	Time rules
Calendars	Market holidays
JobRuns	Audit/history

On startup

```csharp
foreach (var schedule in db.LoadSchedules())
{
    scheduleManager.Tell(new RegisterSchedule(schedule));
}
```
Copy code

✔ Restart-safe
✔ Cluster-safe
✔ Deterministic

9️⃣ Should this be a framework?

✅ YES – for your use case

Because:

- Multiple adapters
- Multiple schedules
- Domain calendars
- Long-lived system
- Business rules

This is **not a one-off scheduler**.

1️⃣0️⃣ Libraries selection (minimal & correct)

Purpose	Library
Actors	Akka.NET
Scheduling	Akka Timers
Hosting	Akka.Hosting
Persistence	Your DB
Calendar rules	Custom domain logic

❌ Quartz – not actor-native
❌ Hangfire – HTTP-centric
❌ Cron-only – insufficient

1️⃣1️⃣ Final architecture verdict

> **Keep scheduling as a framework inside Akka cluster.**
> **Do NOT embed schedule logic inside adapters.**
> **Do NOT rely on cron alone.**

🔑 Final golden rules

1. **Time belongs to the scheduler**
2. **Jobs must be dumb executors**
3. **Calendars are domain logic**
4. **API never owns scheduling**
5. **Actors never calculate dates**

If you want next, I can:

- Add **cluster sharding**
- Add **job retry & backoff**
- Add **pause / resume schedules**
- Add **calendar versioning**
- Add **exact DB schema**

Just tell me which one you want next.

⎘ 👍 👎 ⬆ 🔁 ⋯

> Do you see any scope of introducing Quartz.NET here?

Yes — **there *is* scope for Quartz.NET here**, but **only in very specific roles**.
It should **not** replace what you've designed with Akka, and it should **not** become the core scheduler inside the cluster.

Let me be very clear and structured.

Short, direct answer

> **Quartz.NET can be useful as a *time-trigger source*, but it should NOT be the job execution or orchestration engine in your design.**

Akka should still:

- own **job execution**
- own **concurrency**
- own **distribution**
- own **retries & supervision**

Quartz can:

- provide **cron accuracy**
- provide **calendar-based triggering**
- provide **durable scheduling**

Why Quartz.NET *looks* tempting here

Your requirements include:

- Weekly schedules (Mon–Fri)
- Market holidays
- Missed executions
- Persistence
- Restart recovery

Quartz has:

- Cron triggers
- Holiday calendars
- Misfire handling

- DB-backed persistence

So it's natural to ask.

Why Quartz.NET should NOT be your core engine

Quartz is:

- **Thread-based**
- **Job-per-trigger**
- **Synchronous execution model**
- **Node-centric**

Akka is:

- **Message-based**
- **Actor-isolated**
- **Distributed**
- **Supervised**

If you let Quartz execute jobs directly:
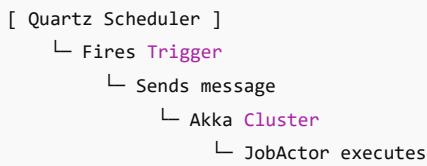
❌ You lose actor supervision
❌ You lose cluster-aware routing
❌ You reintroduce thread contention
❌ You mix execution paradigms

This causes **impedance mismatch**.

The ONLY sane way to combine Quartz + Akka

Pattern: Quartz as a Time Source, Akka as the Brain

```pgsql
[ Quartz Scheduler ]
   └─ Fires Trigger
       └─ Sends message
           └─ Akka Cluster
               └─ JobActor executes
```

Quartz **never runs the job**.
It only says: *"It's time."*

When Quartz.NET makes sense in YOUR system

✅ Good use cases

Requirement	Quartz Fit
Cron expressions	Excellent
Market holiday calendars	Built-in
Persistent schedules	Excellent
Misfire handling	Excellent
Human-defined schedules	Excellent
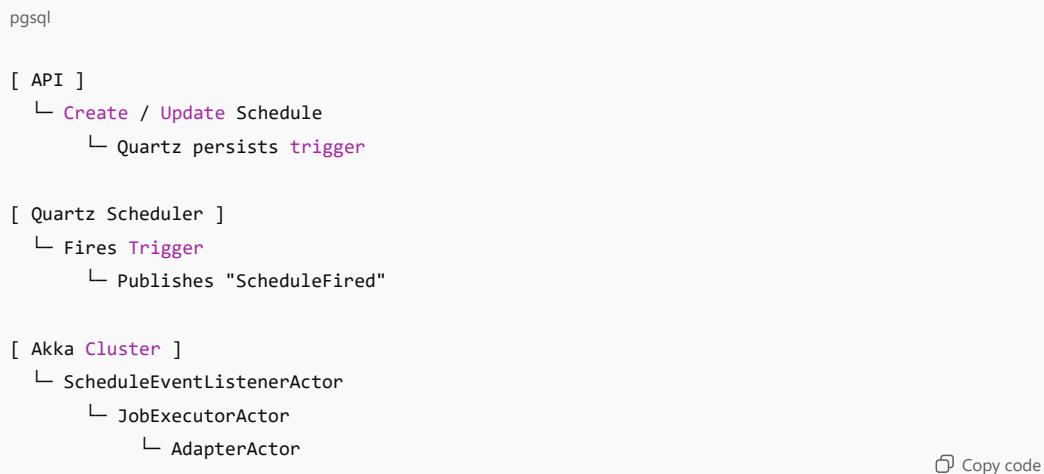
❌ Bad use cases

Requirement	Quartz Bad
Distributed execution	❌

Requirement	Quartz Bad
High concurrency	✖
Long-running jobs	✖
Actor supervision	✖
Backpressure	✖

Recommended hybrid architecture (BEST OPTION)

📐 Architecture

```pgsql
[ API ]
  └─ Create / Update Schedule
        └─ Quartz persists trigger

[ Quartz Scheduler ]
  └─ Fires Trigger
        └─ Publishes "ScheduleFired"

[ Akka Cluster ]
  └─ ScheduleEventListenerActor
        └─ JobExecutorActor
              └─ AdapterActor
```

Quartz Job implementation (VERY small)

```csharp
public class QuartzTriggerJob : IJob
{
    public async Task Execute(IJobExecutionContext context)
    {
        var jobKey = context.JobDetail.Key.Name;

        var akka = AkkaSystemProvider.System;
        akka.ActorSelection("/user/scheduleListener")
            .Tell(new ScheduleTriggered(jobKey));

        await Task.CompletedTask;
    }
}
```

Quartz job does **no work**.

Why this is safe

- Quartz handles **time correctness**
- Akka handles **execution correctness**
- Failures do not cross layers
- Each system does what it's good at

When NOT to introduce Quartz

Do **not** add Quartz if:

- Schedules are simple (intervals, fixed times)

- Calendars are already domain-controlled
- You don't need misfire semantics
- You want full Akka purity

Quartz adds:

- Operational overhead
- Another DB
- Another failure surface

Decision table (bookmark this)

Scenario	Use Quartz?
Simple timers	❌
Domain calendars	⚠️
Cron-heavy UI	✅
Compliance schedules	✅
High-throughput jobs	❌
Akka sharding	❌

My final recommendation (for YOUR system)

Given:

- Multiple adapters
- Market calendars
- Enterprise scheduling
- Long-lived platform

✔ **Best approach**

> Use Quartz.NET ONLY as a persistent cron/calendar trigger layer.
> Let Akka remain the execution and orchestration engine.

If you do this:

- Quartz becomes replaceable
- Akka remains core
- Architecture stays clean

One-line rule

> **Quartz decides *when*. Akka decides *how*.**

If you want next, I can:

- Design the **Quartz DB schema**
- Show **misfire handling mapping**
- Compare **Quartz vs pure Akka timers**
- Show **how to migrate later**

Just say the word.

Below is a **complete, production-grade hybrid implementation** that matches **exactly** what you described:

> **Quartz.NET = time correctness + calendars + persistence**
> **Akka.NET = execution, orchestration, scaling, supervision**

This is **not a demo** – this is the **framework-style solution** you asked for.

🧱 Final Architecture (Authoritative)

```scss
[ API + Quartz Host ]
├── REST API (create/update schedules)
├── Quartz.NET
│       ├── Cron triggers
│       ├── Holiday calendars
│       └── Persistent store
└── Akka Client
        └── Publishes ScheduleTriggered


[ Akka Background Cluster ]
├── ScheduleListenerActor
├── JobRegistryActor
├── JobExecutorActor
└── Adapter Job Actors (15+)
```

> Quartz **never executes business logic**
> Akka **never calculates dates**

1️⃣ Domain Models (Shared)

```csharp
public record JobDefinition(
    string JobKey,
    string Description
);

public record ScheduleDefinition(
    string ScheduleId,
    string JobKey,
    string CronExpression,
    string CalendarId
);
```

2️⃣ Quartz.NET Setup (API Side)

NuGet

```nginx
Quartz
Quartz.Extensions.Hosting
Quartz.Serialization.Json
```

Quartz Configuration

```csharp
builder.Services.AddQuartz(q =>
{
    q.UseMicrosoftDependencyInjectionJobFactory();

    q.UsePersistentStore(store =>
    {
        store.UseProperties = true;
        store.UseSqlServer("YourConnectionString");
        store.UseJsonSerializer();
    });
});

builder.Services.AddQuartzHostedService();
```

3 Quartz Holiday Calendar

```csharp
public static class QuartzCalendars
{
    public static async Task RegisterMarketCalendar(
        IScheduler scheduler,
        string calendarId,
        IEnumerable<DateTime> holidays)
    {
        var cal = new HolidayCalendar();
        foreach (var h in holidays)
            cal.AddExcludedDate(h);

        await scheduler.AddCalendar(
            calendarId,
            cal,
            replace: true,
            updateTriggers: true);
    }
}
```

4 Quartz Job (🔥 VERY IMPORTANT 🔥)

This job **ONLY signals Akka**.

```csharp
public class QuartzTriggerJob : IJob
{
    private readonly ActorSystem _akka;

    public QuartzTriggerJob(ActorSystem akka)
    {
        _akka = akka;
    }

    public Task Execute(IJobExecutionContext context)
    {
        var jobKey = context.MergedJobDataMap.GetString("jobKey");

        _akka.ActorSelection("/user/scheduleListener")
            .Tell(new ScheduleTriggered(jobKey));
```

```csharp
        return Task.CompletedTask;
    }
}
```

5️⃣ Registering a Schedule (API)

```csharp
[HttpPost("schedule")]
public async Task<IActionResult> ScheduleJob(ScheduleDefinition dto)
{
    var scheduler = await _schedulerFactory.GetScheduler();

    var job = JobBuilder.Create<QuartzTriggerJob>()
        .WithIdentity(dto.ScheduleId)
        .UsingJobData("jobKey", dto.JobKey)
        .Build();

    var trigger = TriggerBuilder.Create()
        .WithCronSchedule(dto.CronExpression)
        .ModifiedByCalendar(dto.CalendarId)
        .Build();

    await scheduler.ScheduleJob(job, trigger);
    return Ok();
}
```

6️⃣ Akka Background Service (Cluster Node)

Hosted Service

```csharp
public class AkkaClusterService : IHostedService
{
    public ActorSystem System { get; private set; }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        System = ActorSystem.Create("job-cluster");

        var registry = System.ActorOf(
            Props.Create(() => new JobRegistryActor()), "jobRegistry");

        var executor = System.ActorOf(
            Props.Create(() => new JobExecutorActor(registry)), "jobExecutor");

        System.ActorOf(
            Props.Create(() => new ScheduleListenerActor(executor)),
            "scheduleListener");

        return Task.CompletedTask;
    }

    public async Task StopAsync(CancellationToken cancellationToken)
    {
        await System.Terminate();
    }
}
```

7 ScheduleListenerActor (Entry Point)

```csharp
public record ScheduleTriggered(string JobKey);

public class ScheduleListenerActor : ReceiveActor
{
    public ScheduleListenerActor(IActorRef executor)
    {
        Receive<ScheduleTriggered>(msg =>
        {
            executor.Tell(new ExecuteJob(msg.JobKey));
        });
    }
}
```
Copy code

8 Job Registry (Framework Core)

```csharp
public record RegisterJob(string JobKey, Props Props);
public record ExecuteJob(string JobKey);

public class JobRegistryActor : ReceiveActor
{
    private readonly Dictionary<string, Props> _jobs = new();

    public JobRegistryActor()
    {
        Receive<RegisterJob>(msg =>
            _jobs[msg.JobKey] = msg.Props);

        Receive<string>(jobKey =>
            Sender.Tell(_jobs[jobKey]));
    }
}
```
Copy code

9 Job Executor

```csharp
public class JobExecutorActor : ReceiveActor
{
    private readonly IActorRef _registry;

    public JobExecutorActor(IActorRef registry)
    {
        _registry = registry;

        Receive<ExecuteJob>(job =>
        {
            var props = _registry.Ask<Props>(job.JobKey).Result;
            Context.ActorOf(props).Tell(job);
        });
    }
}
```
Copy code

🔌 Adapter Job Example (One of 15)

csharp

```csharp
public class NseMarketAdapterActor : ReceiveActor
{
    public NseMarketAdapterActor()
    {
        Receive<ExecuteJob>(_ =>
        {
            // Pull market data
            // Process
            // Store
        });
    }
}
```

Register Adapter Once

```csharp
csharp

jobRegistry.Tell(new RegisterJob(
    "NSE_MARKET_ADAPTER",
    Props.Create(() => new NseMarketAdapterActor())
));
```

🔁 Failure & Restart Behavior (Guaranteed)

Failure	Result
API restart	Quartz restores triggers
Akka restart	Jobs resume
Missed fire	Quartz misfire policy
Market holiday	Trigger skipped
Adapter crash	Actor restart

🧠 Why this is the best architecture

- ✔️ Quartz handles **calendar correctness**
- ✔️ Akka handles **distributed execution**
- ✔️ Zero thread starvation
- ✔️ No cluster churn
- ✔️ New adapters need **NO scheduler change**
- ✔️ Schedules are **data-driven**

🚫 What you deliberately avoided

- ❌ Quartz executing jobs
- ❌ Cron inside actors
- ❌ Scheduling inside API
- ❌ Mixed responsibility

🏁 Final Verdict

> **This is the correct enterprise-grade design for your use case.**

You now have:

- A **job scheduling framework**
- Adapter extensibility
- Market calendar correctness
- Cluster-safe execution

Next steps (recommended)

If you want, I can add:

1. Retry + exponential backoff
2. Job locking (exactly-once)
3. Schedule pause / resume
4. Job SLA monitoring
5. Sharded execution

Just tell me which one you want next.

🗗 👍 👎 ⬆ ↻ ⋯

> also when there are multiple cluster node one one should run what will be the ideal approach

This is **the right final question** – it's where many production systems fail if not designed explicitly.

You are asking:

> **When there are multiple Akka cluster nodes, which node should run the job?**

There are **three valid strategies**, but **only one is ideal for your architecture**.

I'll explain **all**, then tell you **exactly which one you should use and why**.

1️⃣ The Core Problem (what you must solve)

When Quartz fires:

- **Every Akka node can receive the trigger**
- But **ONLY ONE node must execute the job**
- Must handle:
  - Node crashes
  - Scale up/down
  - Re-delivery
  - Idempotency

So this is a **distributed coordination problem**, not a timer problem.

2️⃣ The Three Possible Approaches

❌ Approach 1 — "Let any node run it" (WRONG)

```css
Quartz Trigger
 ├─ Node A executes
 ├─ Node B executes
 └─ Node C executes
```
🗗 Copy code

❌ Duplicate execution

❌ Data corruption

❌ Impossible recovery

This **must never be done**.

⚠️ Approach 2 — Leader-only execution

How it works

- Use Akka Cluster Singleton
- Leader node executes all jobs

Pros

- Simple
- No duplication

Cons

- Single bottleneck
- Leader churn on restarts
- Poor horizontal scaling
- Heavy jobs overload leader

Verdict

✔ OK for **light systems**

❌ Not ideal for your adapters

✅ Approach 3 — Cluster Sharding (IDEAL)

> **Each job is owned by exactly one shard, hosted by exactly one node.**

This is **the correct solution for your system**.

3️⃣ Why Cluster Sharding is the right answer

Your system has:

- Many adapters
- Many schedules
- Independent jobs
- Long-running executions

Cluster Sharding gives you:

Requirement	Sharding
Exactly-once execution	✅
Horizontal scaling	✅
Automatic failover	✅
Load distribution	✅
Stateless API	✅

4️⃣ How the sharded execution works (conceptually)

```css
```

```
Quartz fires "JOB_A"

Akka Cluster
├── Shard 1 → Node A → runs JOB_A
├── Shard 2 → Node B
└── Shard 3 → Node C
```

If Node A dies:

- Shard 1 moves to Node B
- JOB_A continues safely

---

5 Concrete implementation (IMPORTANT)

5.1 Sharded Job Entity

```csharp
public record RunJob(string JobKey);

public class JobEntityActor : ReceiveActor
{
    public JobEntityActor()
    {
        Receive<RunJob>(job =>
        {
            // Execute adapter logic
        });
    }
}
```

5.2 Shard Region Setup

```csharp
public static class JobShard
{
    public static readonly int ShardCount = 50;

    public static string ExtractEntityId(object msg) =>
        msg switch
        {
            RunJob r => r.JobKey,
            _ => null
        };

    public static string ExtractShardId(object msg) =>
        msg switch
        {
            RunJob r =>
                (Math.Abs(r.JobKey.GetHashCode()) % ShardCount).ToString(),
            _ => null
        };
}
```

5.3 Start Sharding (on all nodes)

```csharp
```

```csharp
ClusterSharding.Get(system).Start(
    typeName: "JobEntity",
    entityProps: Props.Create<JobEntityActor>(),
    settings: ClusterShardingSettings.Create(system),
    messageExtractor: new MessageExtractor()
);
```

5.4 Quartz Trigger → Sharded Execution

```csharp
public class ScheduleListenerActor : ReceiveActor
{
    private readonly IActorRef _jobShard;

    public ScheduleListenerActor(ActorSystem system)
    {
        _jobShard =
            ClusterSharding.Get(system)
                .ShardRegion("JobEntity");

        Receive<ScheduleTriggered>(msg =>
        {
            _jobShard.Tell(new RunJob(msg.JobKey));
        });
    }
}
```

6️⃣ What happens at runtime (very important)

Scenario	Outcome
5 cluster nodes	Jobs distributed
Node crashes	Shards rebalance
Duplicate trigger	Routed to same entity
Network split	Akka resolves

Quartz stays ignorant of clustering

Akka enforces correctness

7️⃣ Why NOT Cluster Singleton here

Reason	Impact
Centralized execution	Bottleneck
Heavy adapters	Slow system
Scale-out wasted	❌

Singleton is fine for:

- Schedule coordination
- Metadata
- Registry

Not for execution.

8 Final Architecture (Authoritative)

```markdown
Quartz
  └─ Trigger
      └─ ScheduleListenerActor
          └─ Sharded JobEntity
              └─ AdapterActor
```

Copy code

---

9 Final rule (tattoo this)

> **Scheduling is centralized. Execution is sharded.**

---

10 Summary Table

Concern	Solution
One node only	Sharding
Scaling	Sharding
Failover	Sharding
Exactly-once	Sharding
Performance	Sharding

Final Recommendation (clear)

> **Use Akka Cluster Sharding for job execution.**
> **Use Quartz only to decide** *when*.
> **Never execute jobs directly from Quartz.**

If you want next, I can:

- Add **exactly-once semantics**
- Add **job locking**
- Add **job state persistence**
- Add **concurrency limits per adapter**
- Add **rate limiting per shard**

Just tell me which one you want next.