ARCHITECTURAL CASE STUDY

Distributed Automation Architecture

From scheduled job to isolated execution: domain boundaries, event-driven coordination, tenant context, and failure handling.

Define | **Schedule** | **Coordinate** | **Execute** | **Trace**

Abdul Rehman

Solution Architect | Principal .NET Engineer

Architecture, engineering decisions, and delivery scope.

Clear ownership. Coordinated execution.

Reusable workflows, scheduled jobs and isolated workloads needed traceable lifecycles and tenant boundaries.

The engineering challenge

Support immediate, one-time and recurring work. Track a requested run separately from its execution. Account for retries, duplicate messages, restarts and temporary outages.

The selected approach

Separate business responsibilities into independently deployable services. Use APIs for direct interactions and events for asynchronous coordination.

Clients and security

Web app, admin portal, APIs and integrations; authentication, authorization and tenant context.

Execution domains

Flow, job, schedule, run and execution services.

Platform domains

Tenant, organization, project, user, access control, notification and environment services.

Events and data

Apache Kafka, MongoDB, PostgreSQL and Redis.

Infrastructure

Docker, Kubernetes, Argo CD delivery and isolated workloads.

Logical groups for readability; datastore names do not imply shared ownership by every service.

Follow one scheduled job

The run records the requested operation. The execution records the workload and its state.

01 Define the job

Save execution details and an optional schedule.

02 Publish the change

The job service publishes a creation or update event.

03 Register the schedule

The schedule service consumes the event and records timing.

04 Create a run

A trigger creates the record of the requested operation.

05 Create an execution

Record the execution and publish the workload request.

06 Launch the workload

The execution service creates an isolated Kubernetes workload.

07 Track the state

Explicit states: Created, Started, Succeeded or Failed.

08 Publish completion

Update the run, notify interested services and retain history.

Illustrative scheduled-execution path from the documented system. Immediate execution is also supported.

Design for the second delivery

The documented reliability approach includes safe retries, explicit state transitions and traceable events.

Idempotency and duplicate protection

Consumers and retry-safe commands account for repeated delivery. Engineering consideration: define what happens when a request has already changed state.

Run and execution state

Explicit status transitions and failure events describe progress. Engineering consideration: distinguish a failed workload from a delayed status update.

Correlation and visibility

Correlation IDs, structured logs and lifecycle history connect work across services. These support investigation of messaging and execution failures.

Versioned event contracts

Event versioning supports contract evolution. Tradeoff: independently released services still need compatible producer and consumer expectations.

Reliability depends on coordinated state changes, safe retries, and recovery procedures. These design patterns alone cannot guarantee exactly-once execution.

Tenant context travels with the work

The design validates context across requests, domain operations, persistence, events and workload execution.

Identity and authority

- Token-based authentication
- Policy-based authorization
- User and service identity
- Roles and permissions
- Protected environment values

Tenant boundaries

- Tenant, organization and project
- Commands, queries and entities
- Events and data access
- Workload execution
- Logs and audit context

Kubernetes operations

- Dynamic workload creation
- Health and readiness checks
- Declarative configuration
- Rolling updates
- Argo CD delivery

Boundary to review

A scheduled workload still needs authorized tenant and project context after the browser session ends. Isolation depends on application controls as well as workload boundaries.

Architecture with hands-on delivery

My role combined solution architecture, .NET engineering and distributed-system troubleshooting.

Architecture

Defined domain boundaries, APIs, event communication, tenant controls, datastore responsibilities, resilience patterns, and scheduling, run and execution lifecycles.

Engineering

Developed .NET services, commands, queries and validators. Designed Kafka producers and consumers. Implemented MongoDB persistence and revisions, scheduling integration and policy authorization.

Operational contribution

Diagnosed messaging and distributed-system failures. Supported production troubleshooting, architecture decisions and code reviews.

Documented capabilities

Independently deployable services, immediate and scheduled execution, traceable run and execution lifecycles, and isolated Kubernetes workloads.

Technology used

.NET 9, C#, ASP.NET Core, FastEndpoints, MediatR, FluentValidation, Kafka, MongoDB, PostgreSQL, Redis, Docker, Kubernetes and Argo CD.

Scope: engineering capabilities from the author's case study; no measured revenue, customer or performance results are claimed.