Technology Decision Log

Project: Forecast Service Microservice

Date: January 18, 2026
Prepared for: Volue SmartPulse Technical Interview

---

1. Framework & Runtime

Decision: .NET 10 with ASP.NET Core

Rationale:

- **Modern Performance:** .NET 10 offers excellent performance with minimal allocations and high throughput
- **Cross-Platform:** Runs on Windows, Linux, and macOS, perfect for containerized deployments
- **Strong Typing:** C# provides compile-time safety with latest language features
- **Async-First:** Native async/await support for I/O-bound operations like database queries
- **Modern C# Features (Pragmatic Usage):**
  - `field` **keyword:** Used in BaseEntity only for UTC timestamp normalization
  - Applied to CreatedAt and UpdatedAt properties to eliminate explicit backing fields
  - Limited scope - not used for validation (Data Annotations preferred)
- **C# 14 Features Evaluated but Not Used:**
  - **Extension Members:** Only replaced `Sum()` which is already concise (overengineering)
  - `Span<T>` **optimizations:** This is an I/O-bound microservice where network/database latency dominates
  - `field` **keyword for validation:** Data Annotations provide better separation of concerns and ASP.NET Core integration
  - Event publishing happens occasionally, not in tight loops requiring micro-optimizations
  - **Principle:** Prefer readability, maintainability, and framework conventions over premature optimization
- **Enterprise Ready:** Widely adopted in the energy and trading sectors
- **Long-term Support:** Microsoft provides excellent support and regular updates

Alternatives Considered:

- **Node.js/TypeScript:** Good for rapid development but less type-safe at compile time
- **Java/Spring Boot:** Excellent choice but .NET 10 offers better performance metrics
- **Go:** Great performance but smaller ecosystem for enterprise features

---

2. Architecture Pattern

Decision: Clean Architecture (Layered Architecture)

Rationale:

- **Separation of Concerns:** Clear boundaries between business logic, data access, and presentation
- **Testability:** Each layer can be tested independently
- **Maintainability:** Changes in one layer don't cascade to others
- **Dependency Inversion:** Core business logic depends on abstractions, not implementations
- **Industry Standard:** Well-understood pattern for microservices
- **Interview Requirement:** Project specification explicitly requested layered structure

Layer Structure:

1. **Domain Layer:** Core business entities and interfaces (no dependencies)
2. **Application Layer:** Business logic, services, and DTOs
3. **Infrastructure Layer:** Data access, repositories, external integrations
4. **API Layer:** Controllers, request/response handling, middleware

Alternatives Considered:

- **CQRS (Command Query Responsibility Segregation):** Overkill for this scope
- **Event Sourcing:** Too complex for the current requirements
- **Simple Three-Tier:** Less flexible for future expansion

---

3. Database

Decision: PostgreSQL 16

Rationale:

- **Open Source:** No licensing costs, widely supported
- **ACID Compliance:** Ensures data consistency for financial/trading data
- **JSON Support:** Can handle semi-structured data if needed in the future
- **Performance:** Excellent query optimizer and indexing capabilities
- **Time-Series Extensions:** TimescaleDB available for historical forecast analysis
- **Enterprise Adoption:** Widely used in trading and energy sectors
- **Docker Support:** Official Docker images with excellent documentation

Azure Cloud Considerations:

- **Azure Database for PostgreSQL:** Fully managed service with automatic backups, patching, and HA
- **Cost Efficiency:** Pay-per-use pricing with Burstable tier for development, General Purpose for production
- **Scalability:** Vertical scaling (vCores/memory) and horizontal read replicas without code changes
- **Flexible Server:** Best balance of features, performance, and cost in Azure
- **vs Azure SQL:** PostgreSQL offers lower licensing costs and open-source flexibility

Alternatives Considered:

- **SQL Server:** Excellent but expensive licensing for microservices
- **MySQL:** Good but PostgreSQL has better feature set for complex queries

- **MongoDB:** Not ideal for transactional financial data requiring strong consistency
- **SQLite:** Not suitable for production microservices

---

4. ORM (Object-Relational Mapping)

Decision: Entity Framework Core 10.0

Rationale:

- **First-Class Support:** Native .NET integration with excellent tooling
- **Code-First Migrations:** Easy database version control and deployment
- **LINQ Support:** Type-safe queries with IntelliSense
- **Performance:** Compiled queries and efficient change tracking
- **Provider Support:** Excellent Npgsql provider for PostgreSQL
- **Async Operations:** Full async/await support for non-blocking I/O

Alternatives Considered:

- **Dapper:** Faster but more manual work, less suited for rapid development
- **NHibernate:** Mature but EF Core is now performance-competitive
- **Raw ADO.NET:** Too low-level for this project scope

---

5. Repository Pattern

Decision: Generic Repository Pattern with Specific Implementations

Rationale:

- **Abstraction:** Decouples business logic from data access technology
- **Testability:** Easy to mock repositories for unit testing
- **Flexibility:** Can switch data access strategies without changing business logic
- **Interview Alignment:** Matches the requested layered architecture
- **Single Responsibility:** Each repository handles one aggregate root

Implementation:

- Interface definitions in Domain layer
- Concrete implementations in Infrastructure layer
- Scoped lifetime for thread safety

Alternatives Considered:

- **Direct DbContext Usage:** Less flexible, harder to test
- **Unit of Work Pattern:** Added complexity not needed for this scope

---

6. Containerization

Decision: Docker with Docker Compose

Rationale:

- **Environment Consistency:** Same runtime everywhere (dev, test, prod)
- **Easy Setup:** Single command to start entire application stack
- **Isolation:** Database and API run in separate containers
- **Industry Standard:** Docker is widely used for microservices
- **Orchestration Ready:** Easy migration to Kubernetes if needed
- **Multi-Stage Builds:** Optimized image sizes (SDK for build, Runtime for deployment)

Docker Compose Benefits:

- PostgreSQL and API defined together
- Health checks ensure database is ready before API starts
- Persistent volumes for database data
- Easy networking between containers

Azure Cloud Deployment Options:

- **Azure Container Apps:** Serverless containers with auto-scaling, pay-per-use model
  - Cost-effective for variable workloads, scales to zero when idle
  - Built-in ingress, HTTPS, and microservices patterns
- **Azure Kubernetes Service (AKS):** Full Kubernetes orchestration for complex workloads
  - More control but higher operational overhead
  - Cluster-based pricing with per-node charges
- **Azure Container Instances (ACI):** Simple container hosting
  - Lower cost for simple scenarios but limited features
- **Recommendation:** Start with Container Apps for cost efficiency, migrate to AKS if complex orchestration needed

Alternatives Considered:

- **Virtual Machines:** Too heavy, slower startup
- **No Containerization:** Difficult environment management
- **Kubernetes Directly:** Overkill for development/demo purposes

---

7. Event Publishing

Decision: RabbitMQ Message Broker

Rationale:

- **Reliability:** Message persistence, delivery guarantees, and automatic reconnection
- **Standards-Based:** AMQP protocol with wide client support
- **Management UI:** Built-in monitoring at http://localhost:15672 for message inspection
- **Docker Integration:** Official Docker image with easy compose configuration

- **Battle-Tested:** Proven in high-throughput environments
- **Decoupling:** Asynchronous event processing for downstream systems
- **Scalability:** Supports clustering and message routing patterns

Azure Cloud Alternatives & Cost Comparison:

- **Azure Service Bus (Recommended for Azure):**
  - Fully managed, no infrastructure management
  - Pricing: Standard tier with base monthly fee + per-operation charges
  - Enterprise features: Dead letter queues, duplicate detection, sessions
  - Better integration with Azure services (Managed Identity, Key Vault)
  - Lower operational overhead compared to self-hosted RabbitMQ
- **Self-Hosted RabbitMQ on Azure Container Apps:**
  - More control but requires monitoring and maintenance
  - Pricing: Container hosting fees + storage costs
  - Good for AMQP compatibility requirements
- **Azure Event Grid:**
  - Serverless event routing, pay-per-event model
  - Best for event-driven architectures with multiple subscribers
  - Limited to push-based delivery (no pull/polling)

Trade-offs:

- RabbitMQ chosen for demo: portable, self-contained, no cloud dependency
- Azure Service Bus recommended for production: managed service, lower TCO, enterprise features

Implementation Details:

- **Exchange:** `forecast.events` (topic exchange)
- **Routing Key:** `position.changed`
- **Connection:** Automatic reconnection with retry logic
- **Message Format:** JSON serialized events
- **Publisher:** `RabbitMqEventPublisher` in Infrastructure layer

Interface Design:

```
public interface IEventPublisher
{
    Task PublishPositionChangedEventAsync(PositionChangedEvent eventData);
}
```

Alternative Options Considered:

- **In-Memory Publisher:** Simple but no inter-service communication
- **Apache Kafka:** High throughput but complex setup for this scope
- **AWS EventBridge:** Cloud-native but vendor lock-in

---

8. API Design

Decision: RESTful API with Scalar (OpenAPI Documentation)

Rationale:

- **Industry Standard:** REST is universally understood and supported
- **HTTP Semantics:** Proper use of status codes (200, 201, 400, 404, 500)
- **Modern Documentation:** Scalar provides a superior UI experience compared to Swagger
- **OpenAPI 3.0:** Standards-compliant API specification
- **Tooling:** Excellent client generation tools (TypeScript, C#, etc.)
- **Idempotency:** POST operations handle upsert logic gracefully
- **.NET 10 Native:** Built-in OpenAPI support without Swashbuckle dependency

Endpoint Design Philosophy:

- Resource-based URLs (`/api/forecasts`, `/api/companyposition`)
- Proper HTTP verbs (POST for create/update, GET for retrieval)
- Query parameters for filtering (startDate, endDate)
- **ApiResult Pattern:** Consistent `success`, `data`, `error` response structure
- Proper error handling with meaningful messages and error codes

Alternatives Considered:

- **GraphQL:** Flexible but overkill for this simple API
- **gRPC:** Better performance but REST more accessible for demo
- **Swagger UI:** Standard but Scalar offers better UX

---

9. Concurrency & Thread Safety

Decision: Scoped DbContext with Async/Await Pattern

Rationale:

- **Thread Safety:** Each HTTP request gets its own DbContext instance
- **No Shared State:** Repositories are scoped, preventing race conditions
- **Async I/O:** Non-blocking operations for better scalability
- **Connection Pooling:** Npgsql manages connection pooling automatically
- **Scalability:** Stateless design allows horizontal scaling

Implementation Details:

- All repository methods are async

- DbContext registered with Scoped lifetime
- No static shared state
- Each request is isolated

---

10. Data Validation & Error Handling

Decision: Result Pattern with Multi-Layer Validation

Rationale:

- **Type Safety:** Explicit success/failure handling without exceptions
- **Performance:** No exception overhead for business rule violations
- **Functional Style:** Clear separation between success and error paths
- **Controller Level:** Model validation using data annotations
- **Service Level:** Business rule validation (e.g., negative production values)
- **Database Level:** Constraints ensure data integrity
- **Consistent Errors:** Domain errors centralized in `DomainErrors` class

Result Pattern Implementation:

```
public record Result<T>
{
    public bool IsSuccess { get; init; }
    public T? Value { get; init; }
    public Error? Error { get; init; }
}

public record Error(string Code, string Message);
```

Error Response Strategy:

- **400 Bad Request:** Invalid input or business rule violations (domain errors)
- **404 Not Found:** Resource doesn't exist
- **409 Conflict:** Concurrency or constraint violations
- **500 Internal Server Error:** Unexpected errors (logged)
- **Meaningful Messages:** Error codes (e.g., `PowerPlant.NotFound`) and descriptions

Benefits:

- No try-catch blocks in controllers (global exception handler middleware)
- Automatic HTTP status code mapping via Result Pattern
- Clear error propagation through layers
- Easy to test success and error paths
- Centralized exception logging and formatting

---

11. Logging & Monitoring

Decision: Serilog for Structured Logging

Rationale:

- **Structured Logging:** Rich context with property-based logging
- **Multiple Sinks:** Console (development) + File (persistent logs)
- **Performance:** Efficient with minimal overhead
- **Configuration:** Flexible via code and appsettings.json
- **Request Logging:** HTTP request/response details with timing
- **Log Enrichment:** Automatic context injection (timestamp, level, source)

Current Implementation:

- **Console Sink:** Real-time output during development with color coding
- **File Sink:** Daily rolling logs in `logs/forecast-service-YYYYMMDD.log`
- **Retention:** 30-day automatic cleanup
- **Log Levels:** Debug (app), Information (Microsoft), Information (EF Core)
- **Format:** Structured JSON-like output with context

Example Log Output:

```
[2026-01-17 10:30:45 INF] ForecastService.Application.Services.ForecastService
Creating forecast for PowerPlant: 22222222..., DateTime: 2026-01-18T10:00:00Z

[2026-01-17 10:30:45 INF]
ForecastService.Infrastructure.Events.RabbitMqEventPublisher
Published Position Changed Event to RabbitMQ: CompanyId=11111111...,
TotalPosition=850.5 MWh
```

Future Enhancements:

- Correlation IDs for distributed request tracing
- Application Insights or ELK stack for centralized logging
- Structured JSON sink for log aggregation
- Performance metrics and custom event tracking

---

12. Input Validation Strategy

Decision: ASP.NET Core Data Annotations with Nullable Types

Rationale:

- **Framework Integration:** Automatic validation before controller action execution
- **Declarative Approach:** Clear, readable validation rules on properties

- **Consistent Error Format:** Custom `InvalidModelStateResponseFactory` for uniform responses
- **Type Safety:** Nullable value types (`Guid?`, `DateTime?`, `decimal?`) with `[Required]` ensure proper validation
- **Separation of Concerns:** Validation logic separate from business logic and property setters
- **Standard Pattern:** Industry-standard ASP.NET Core approach

Implementation:

```
// DTOs with nullable properties and validation attributes
public class CreateOrUpdateForecastRequest
{
    [Required(ErrorMessage = "Power plant ID is required")]
    public Guid? PowerPlantId { get; set; }

    [Required(ErrorMessage = "Forecast date and time is required")]
    public DateTime? ForecastDateTime { get; set; }

    [Required(ErrorMessage = "Production value is required")]
    [Range(0, double.MaxValue, ErrorMessage = "Production value must be non-
negative")]
    public decimal? ProductionMWh { get; set; }
}

// Controller parameters also validated
public async Task<ActionResult> GetForecast(
    [Required(ErrorMessage = "Forecast ID is required")] Guid? id)
```

Custom Error Response Configuration:

```
builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            var errors = context.ModelState
                .Where(e => e.Value?.Errors.Count > 0)
                .SelectMany(e => e.Value!.Errors)
                .Select(e => e.ErrorMessage);

            return new BadRequestObjectResult(new {
                success = false,
                data = null,
                error = new {
                    title = "Validation Error",
                    detail = string.Join("; ", errors),
                    status = 400,
                    instance = context.HttpContext.Request.Path,
```

```
                    timestamp = DateTime.UtcNow
                };
            });
        };
    });
```

Evolution from Initial Approach:

- **Initial:** Used C# 14 `field` keyword with `ArgumentException` in property setters
- **Problem:** Exceptions thrown during model binding before middleware could catch them
- **Solution:** Nullable types with Data Annotations - validated automatically by ASP.NET Core
- **Benefits:**
  - No custom middleware needed for validation errors
  - Consistent with ASP.NET Core conventions
  - Clear separation: validation (annotations) vs domain logic (services)
  - Better error messages returned as 400 Bad Request, not 500 Internal Server Error

Error Response Format:

```
{
  "success": false,
  "data": null,
  "error": {
    "title": "Validation Error",
    "detail": "Power plant ID is required; Production value must be non-negative",
    "status": 400,
    "instance": "/api/forecasts",
    "timestamp": "2026-01-18T10:30:45Z"
  }
}
```

Alternatives Considered:

- **FluentValidation:** More powerful but unnecessary for simple validation rules
- **Custom Middleware:** Catches exceptions but validation should happen earlier in pipeline
- **Field Keyword Validation:** Elegant but throws during model binding, inconsistent HTTP status codes

---

13. Development & Deployment

Decision: Visual Studio Code-Friendly with .NET CLI

Rationale:

- **Cross-Platform:** Works on any OS
- **Lightweight:** Fast development experience

- **CLI First:** All operations scriptable
- **CI/CD Ready:** Easy integration with GitHub Actions, Azure DevOps

Build & Run:

```
dotnet build
dotnet run
docker-compose up
```

Summary

The technology choices prioritize:

1. **Enterprise Patterns:** RabbitMQ, Serilog, Result Pattern, Docker
2. **Maintainability:** Clean architecture with clear separation
3. **Scalability:** Stateless design, containerization, async operations
4. **Developer Experience:** Modern tooling, Scalar API docs, structured logging
5. **Interview Alignment:** Exceeds all specified requirements
6. **Industry Best Practices:** Patterns used in real trading platforms
7. **Error Handling:** Type-safe Result Pattern without exception overhead
8. **Observability:** Serilog with multiple sinks, RabbitMQ management UI

These decisions balance **pragmatism** (efficient development) with **professionalism** (enterprise architecture), demonstrating both technical competence and architectural maturity suitable for an energy trading platform microservice. The implementation goes beyond basic requirements to showcase enterprise-grade patterns including event-driven architecture, structured logging, and functional error handling.