# Architecture Decision Records

This document logs the key architectural decisions made for the PineCone Pro Supplies ERP/IMS project. Each record captures the context, the decision taken, alternatives considered, the rationale and the consequences. The log is maintained to provide traceability for future team members and to avoid revisiting the same questions.

## ADR‑001: Adopt a Microservices Architecture

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

The ERP/IMS platform must support a wide range of functions—product information management, inventory, order management, purchasing, shipping, tax and accounting—and must evolve quickly as the business grows. A monolithic application would be complex to develop, deploy and scale across these diverse domains. We therefore need to decide how to structure the system.

### Decision

The solution will be built as a suite of domain‑focused microservices. Each domain (e.g., Product Information, Inventory, Orders, Purchasing, Shipping) will be implemented as an independent service running in its own container and owning its own data store. Services communicate through RESTful APIs for synchronous requests and through asynchronous events for workflow orchestration.

### Alternatives

* **Monolithic application**: Build a single codebase with all modules deployed together.
* **Modular monolith**: Use modules within a single deployment but without network boundaries.
* **Service‑oriented architecture (SOA)**: Implement shared services with an enterprise service bus.

### Rationale

* Microservices allow **independent scaling** of each domain. If one service experiences heavy seasonal demand, it can be scaled up without affecting others. This improves resource utilisation and reduces cost.
* The architecture improves **fault isolation**; a failure in one service is less likely to cascade across the system. Developers can deploy new modules without redesigning the entire application, enabling **faster time‑to‑market**.
* Microservices are **language‑agnostic**, allowing each service to use the most suitable technology stack. In our case, we have chosen ASP.NET Core for all services to maintain consistency but future extensions could use other languages.
* Each service manages its own database, improving **data security** and allowing compliance boundaries to be clearly defined.

### Consequences

* Microservices introduce operational complexity (container orchestration, service discovery, network policies).
* Testing and debugging distributed systems is more complex than with a monolith. Integration testing requires orchestrating multiple services.
* Managing multiple deployment pipelines and APIs demands robust DevOps practices and governance.

## ADR‑002: Choose Angular for the Front‑End

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

The system requires a responsive, feature‑rich web user interface that can serve both B2B and B2C customers, as well as internal staff. The front‑end must be modular, maintainable and capable of rapid development by multiple developers.

### Decision

Angular (version 14+) will be the primary front‑end framework for the ERP/IMS. The application will be a single‑page application (SPA) that consumes RESTful APIs and uses Angular’s CLI for build and deployment. The UI will use Angular Material and the component‑based architecture to deliver a consistent experience across web, tablet and mobile.

### Alternatives

* **React**: A popular component library with more flexibility but a less opinionated structure.
* **Vue.js**: Lightweight framework suitable for smaller applications.
* **Razor pages/Blazor**: Use server‑side rendering or WebAssembly with .NET.

### Rationale

* Angular’s **component‑based architecture** and **modular design** help break down complex UIs into reusable pieces, which suits our multi‑module ERP.
* The framework provides **high‑performing, adaptable single‑page applications** and includes TypeScript, which adds static typing and improves maintainability.
* Angular CLI and its opinionated structure make it easier for larger teams to collaborate; modules can be shared as NPM libraries, which streamlines development across teams.
* Angular Material supplies ready‑made UI components and supports responsive design, reducing the time to build a polished interface.

### Consequences

* Angular has a steeper learning curve than some alternatives and requires developers to be comfortable with TypeScript and RxJS.
* The framework is opinionated and may be less flexible for unconventional UX patterns.
* Major upgrades (e.g., Angular 15, 16) might require refactoring parts of the codebase.

## ADR‑003: Use ASP.NET Core for Microservices

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

We need a reliable, high‑performance backend technology for building microservices that can handle thousands of requests per second and run cross‑platform on Linux containers. The framework should have strong tooling, robust security and a large ecosystem.

### Decision

All backend services will be implemented using ASP.NET Core 6 or later. Services will expose RESTful APIs, implement dependency injection, and use Entity Framework Core for database access. Each service will be packaged into a Docker container for deployment on Kubernetes.

### Alternatives

* **Node.js with Express**: Lightweight, event‑driven JavaScript runtime.
* **Java Spring Boot**: A mature, feature‑rich framework for building microservices.
* **Python Django/Flask**: Rapid development frameworks but less performant for high‑traffic APIs.

### Rationale

* ASP.NET Core is **cross‑platform**; it runs on Windows, Linux and macOS. Its Docker integration allows deployment in any environment and fits well with Kubernetes.
* The framework delivers **high performance** through a lightweight request pipeline and the Kestrel web server, outperforming many alternatives in benchmarks.
* It includes **built‑in support for dependency injection**, configuration management and environment targeting, which simplifies testability and clean architecture.
* ASP.NET Core integrates with **Entity Framework Core** for data access, supports multiple databases (SQL Server, PostgreSQL, MySQL) and can also connect to NoSQL stores.
* Security is a first‑class citizen: the framework provides authentication and authorization support for JWT, OAuth2 and OpenID Connect and includes middleware for enforcing HTTPS, CSRF and data protection.

### Consequences

* The development team must maintain proficiency in C# and .NET.
* While ASP.NET Core is open source, it is primarily stewarded by Microsoft; strategic changes may be influenced by vendor priorities.
* Some libraries and tools may be less mature than in other ecosystems (e.g., Node.js or Java), although the .NET ecosystem has grown substantially.

## ADR‑004: Adopt Domain‑Driven Design & Bounded Contexts

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

To define microservice boundaries, we need a systematic approach to decompose the business domain. The ERP/IMS spans diverse functions (product management, orders, purchasing, warehousing). Without proper boundaries, services could end up too coarse or too chatty.

### Decision

We will apply **Domain‑Driven Design (DDD)** principles to define bounded contexts. Each microservice corresponds to a bounded context that aligns with a domain area (e.g., Product, Inventory, Orders). Within each context we will model aggregates, entities and value objects that encapsulate business rules and invariants.

### Alternatives

* **Layered architecture without DDD**: Decompose by technical layers (e.g., presentation, business, data) but without explicit domain boundaries.
* **Anemic domain model**: Use data transfer objects and services without rich domain models.
* **Event storming only**: Use event discovery to identify bounded contexts without fully adopting DDD.

### Rationale

* DDD advocates modeling based on the reality of business domains and uses **bounded contexts** to define clear boundaries where a model applies. Each context correlates to a microservice and emphasises a common business language.
* Using DDD helps align the software with business processes and reduces coupling between domains; the ubiquity of the model improves communication between developers and domain experts.
* DDD encourages rich domain models (aggregates, entities, value objects) and encapsulates business rules behind aggregate roots, which supports clean and maintainable code.
* Bounded contexts help identify when microservices should remain independent and when a function should not be split (to avoid chatty calls and poor autonomy).

### Consequences

* DDD has a learning curve and requires collaboration with domain experts.
* It may introduce complexity where simple CRUD services could suffice; thus we will apply DDD primarily to services with significant business rules (e.g., Orders, Purchasing) and use simpler patterns for CRUD‑heavy services.
* The process of defining bounded contexts and ubiquitous language will require workshops and ongoing refinement as the business evolves.

## ADR‑005: Containerization & Azure Container Registry

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

We need a consistent runtime environment across development, testing and production and a way to package microservices along with their dependencies. We also need a secure repository to store and manage container images.

### Decision

All services and the front‑end application will be containerized using Docker. We will use **multi‑stage Docker builds**: for backend services, the first stage will restore and publish the .NET code, and the final stage will copy the output into a lightweight runtime image; for the front‑end, the first stage will build the Angular app using a Node image and the second stage will host it with an Nginx server. Images will be stored in a private **Azure Container Registry (ACR)**.

### Alternatives

* **Run code directly on VMs**: Deploy .NET services to virtual machines without containers.
* **Use Azure App Service**: Deploy services as App Service apps.
* **Publish Docker images to Docker Hub**: Use a public registry.

### Rationale

* Containers ensure that services run consistently on any environment (developer workstation, CI server, AKS cluster). Multi‑stage builds reduce image size by separating build and runtime stages, improving security and performance.
* A private ACR secures images and integrates with Azure AD for authentication; AKS can pull images using its managed identity without exposing credentials.
* Containerization makes it easy to replicate and scale services on Kubernetes, enabling rapid deployments and rollbacks.

### Consequences

* Developers must learn Docker basics and manage Dockerfiles for each service.
* Image storage and scanning must be managed (ACR tasks, vulnerability scanning).
* ACR introduces additional cost compared with public registries but improves security.

## ADR‑006: Deployment Platform – Azure Kubernetes Service

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

Our containerized services require an orchestration platform that can handle high availability, scaling, rolling upgrades, service discovery and secure networking. Azure offers multiple compute options including AKS, App Service and Container Apps. We must choose the deployment platform.

### Decision

The microservices and front‑end container will be deployed on **Azure Kubernetes Service (AKS)**. A **managed NGINX ingress controller** will handle incoming HTTP(S) traffic and route it to the appropriate service. AKS will run across multiple availability zones for high availability and will integrate with Azure Load Balancer for external connectivity.

### Alternatives

* **Azure App Service**: Deploy each service as a separate web app.
* **Azure Container Apps**: Use a serverless container platform.
* **Self‑managed Kubernetes**: Operate Kubernetes clusters on virtual machines.

### Rationale

* AKS is a **managed Kubernetes** platform, offloading cluster management tasks while providing full Kubernetes APIs and features.
* A **managed NGINX ingress** implements the API gateway pattern; it terminates TLS, validates JWT tokens and routes traffic based on URL paths. This simplifies client interaction and reduces duplication of cross‑cutting concerns.
* AKS integrates with Azure AD for cluster authentication and with ACR for pulling container images via managed identities.
* AKS supports autoscaling, rolling upgrades, node pools and multi‑zone deployments, which are critical for a production ERP system.

### Consequences

* Kubernetes adds operational complexity; the team must manage manifests, Helm charts and cluster policies.
* AKS incurs management costs; cost optimization is required.
* There is a learning curve for Kubernetes, but it provides powerful primitives for scaling and resilience.

## ADR‑007: Persistence & Data Stores

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

Each microservice must persist data and maintain its own state. The ERP/IMS system deals with transactional data (orders, inventory levels), reference data (products, vendors) and potentially large volumes of semi‑structured data. We need to choose appropriate data stores and determine whether services share databases.

### Decision

Each microservice will have its own **Azure SQL Database** instance to store its transactional data. Where appropriate (e.g., product attributes with varying schemas), **Azure Cosmos DB** may be used. **Azure Cache for Redis** will be used for caching frequently accessed data. Files and attachments will be stored in **Azure Blob Storage**.

### Alternatives

* **Shared database**: All services share a single database instance.
* **NoSQL only**: Use a document database such as Cosmos DB for all services.
* **Self‑hosted SQL**: Run a central SQL Server in a VM.

### Rationale

* The microservices reference architecture suggests using **external data stores** such as Azure SQL DB, Cosmos DB and Redis for stateful microservices. This ensures data is persisted outside of the cluster and can scale independently.
* Having a **database per service** avoids tight coupling and allows each service to evolve its schema independently. It supports independent scaling and deployment.
* Azure SQL Database provides managed, highly available relational storage; Cosmos DB provides schema flexibility for semi‑structured data; Redis provides low‑latency caching for read‑heavy workloads.
* Storing binary objects in Blob Storage decouples large file storage from database operations.

### Consequences

* Managing multiple databases increases administrative overhead; automated provisioning and migration scripts are required.
* Data must be synchronised across services via events or explicit APIs, leading to eventual consistency.
* Developer tooling must support multiple connection strings and database contexts.

## ADR‑008: Asynchronous Messaging & Service Bus

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

Synchronous API calls can result in tight coupling between services and can degrade performance when a downstream service is unavailable or slow. The ERP/IMS requires asynchronous workflows (e.g., order processing, inventory updates) and event‑driven patterns to decouple services and improve resilience.

### Decision

We will use **Azure Service Bus** topics and queues for asynchronous messaging. Microservices will publish events (e.g., OrderPlaced, InventoryReserved, ShipmentCreated) to topics. Other services will subscribe to these events using the **publisher‑subscriber** and **competing consumer** patterns. Service Bus will also be used for command queues when appropriate.

### Alternatives

* **Synchronous REST for all interactions**: Services call each other directly via HTTP.
* **Kafka**: Use Apache Kafka for messaging.
* **RabbitMQ**: Self‑hosted message broker.

### Rationale

* The reference architecture uses Service Bus to queue delivery requests and illustrates the **publisher‑subscriber** and **competing consumers** patterns for decoupled workflows. Using Service Bus ensures at‑least‑once delivery and durable messaging.
* Events allow services to react to changes without tight coupling; this improves scalability and fault tolerance.
* Azure Service Bus is fully managed and integrates with Azure AD; it scales horizontally and supports topics, subscriptions, dead‑letter queues and message sessions.

### Consequences

* Additional complexity: message contracts and versioning must be managed carefully.
* Services become eventually consistent; consumers must handle idempotency and duplication.
* Operational monitoring of queues and handling of poison messages is required.

## ADR‑009: Authentication & Authorization

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

The ERP/IMS is accessed by multiple user roles (e.g., customers, warehouse associates, purchasing leads) and services. We need to secure APIs and implement role‑based access control across the system.

### Decision

We will use **Microsoft Entra ID** (formerly Azure Active Directory) for user authentication, single sign‑on and role management. Each microservice will validate JSON Web Tokens (JWTs) issued by Entra ID. AKS and service pods will use **managed identities** to access Azure resources (e.g., SQL databases, Service Bus). Role‑based access control (RBAC) will be enforced at the API gateway and within services.

### Alternatives

* **Custom identity provider**: Implement our own token issuing and management.
* **Auth0/Okta**: Use a third‑party identity service.
* **Basic authentication**: Use simple username/password for each service.

### Rationale

* The microservices reference architecture requires a **managed identity** for the cluster to access Azure Container Registry and other resources. Using Entra ID provides unified identity management and integrates with RBAC.
* Entra ID supports OpenID Connect, OAuth2 and multi‑factor authentication; it provides enterprise security features such as conditional access and identity protection.
* Managed identities simplify service‑to‑service authentication and allow secure access to Azure services without storing secrets.

### Consequences

* Requires configuration of Azure AD tenants, app registrations and service principals.
* External partners may need guest accounts or B2B integration.
* If the organisation uses another identity provider, integration with Entra ID must be established.

## ADR‑010: Continuous Integration & Deployment (CI/CD)

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

To ensure reliable releases and reduce manual errors, the ERP/IMS project needs automated processes for building, testing and deploying microservices and the front‑end. Multiple teams will contribute to different services, so the pipeline must support independent deployments.

### Decision

We will adopt **Azure Pipelines** for CI/CD. Each microservice and the front‑end application will have its own pipeline that performs linting, unit tests, integration tests, builds Docker images and publishes them to Azure Container Registry. We will use **Helm** charts to package Kubernetes manifests and deploy to AKS. Pipelines will promote releases through dev, staging and production environments with approvals and automated smoke tests.

### Alternatives

* **GitHub Actions**: Use GitHub‑hosted runners for CI/CD.
* **Jenkins**: Self‑hosted CI server.
* **Manual deployment**: Build and deploy images manually.

### Rationale

* Azure Pipelines provides a cloud‑hosted CI/CD service tightly integrated with Azure DevOps and AKS; it can run automated builds, tests and deployments and supports multiple repositories and stages.
* Using Helm standardises the deployment process and allows versioned releases; charts bundle all Kubernetes objects into a single unit that can be deployed and rolled back.
* Pipelines allow each microservice team to own its pipeline and release cadence, supporting parallel development.

### Consequences

* Requires configuration and maintenance of YAML pipeline definitions and Helm charts.
* Additional compute cost for pipeline execution.
* Teams must monitor and respond to pipeline failures and maintain secrets for deployment (handled via Azure Key Vault).

## ADR‑011: Observability & Monitoring

**Date**: 2025‑09‑28  
**Status**: Accepted

### Context

Operating a distributed microservices platform requires comprehensive observability to detect and diagnose issues, monitor performance and ensure reliability. The ERP/IMS must track system health as well as business‑level metrics (e.g., order processing times, inventory accuracy).

### Decision

We will use **Azure Monitor** and **Application Insights** to collect metrics, logs and traces from all microservices and the AKS infrastructure. Each service will emit structured logs and telemetry that can be correlated via a common operation identifier. Dashboards will be set up to visualize key indicators and alerts will notify the on‑call team of anomalies.

### Alternatives

* **Prometheus + Grafana**: Deploy open‑source monitoring stack.
* **ELK Stack**: Use Elasticsearch, Logstash and Kibana for logging.
* **Third‑party APM**: Use tools such as Datadog or New Relic.

### Rationale

* Azure Monitor integrates with AKS to collect metrics from controllers, nodes and containers; it stores logs centrally and supports querying via Kusto.
* **Application Insights** monitors microservices and provides distributed tracing, dependency tracking and availability tests, enabling a single application map and end‑to‑end visibility of request flows.
* Using native Azure tools reduces operational overhead and seamlessly integrates with Azure RBAC and identity management.

### Consequences

* Azure Monitor and Application Insights incur usage‑based costs; careful retention and sampling policies are necessary.
* Additional instrumentation code must be added to services to capture custom metrics and traces.
* Development teams must learn Kusto Query Language (KQL) for querying logs and metrics.

These Architecture Decision Records capture the key decisions and their justifications for the PineCone Pro ERP/IMS project. This log will be maintained as the system evolves. New decisions should follow this template, and existing records should not be altered once accepted; instead, superseding ADRs should be created if a decision changes.