Zava Retail Architecture Guidelines

**Zava** (a regional B2C retailer serving suburban communities) is modernising its customer-facing applications while keeping a local, trusted feel. This document presents a comprehensive architecture for Zava’s new system, using a structured methodology and guided by the provided architecture diagram. We incorporate a **C4 model** approach – outlining Context, Containers, Components, and Code – to ensure clarity and completeness.

Build vs Buy Decisions

### Decision 1: AI-Powered Product Recommendation Engine

Requirement: Zava wants to offer highly personalised product recommendations to shoppers, leveraging AI to analyse preferences and behaviour.

Build Option:

Develop a custom recommendation engine using Azure OpenAI and in-house data science expertise.

Integrate tightly with Zava’s product catalogue and loyalty data for maximum relevance.

Buy Option:

Purchase a SaaS recommendation platform (e.g., Microsoft Personalizer or a retail-focused AI vendor).

Integrate via APIs, accepting some limitations in customisation.

Decision Process:

Strategic Fit: Personalisation is a key differentiator for Zava, so building allows for unique features and brand alignment.

Cost & Time: Building is more expensive and slower, but offers long-term flexibility. Buying is faster to market.

Risk: Buying reduces technical risk but may limit innovation.

Outcome: Zava chooses to build, using Azure AI services for core models but developing the orchestration and business logic in-house.

### Decision 2: Loyalty Programme Management System

Requirement: Zava needs a robust system to manage customer loyalty points, rewards, and promotions.

Build Option:

Custom-build a loyalty management module as part of the backend API, using Azure PostgreSQL and Cosmos DB for data storage.

Buy Option:

Acquire a commercial loyalty management solution with pre-built features and integrations.

Decision Process:

Requirements: Zava’s needs are standard (points, tiers, offers) and not a major differentiator.

Integration: Buying a solution that supports Azure and Entra ID simplifies deployment.

Cost: Buying is more cost-effective and reduces maintenance burden.

Outcome: Zava opts to buy, selecting a vendor with proven Azure integration and customisation options for branding.

### Decision 3: In-Store Layout Guidance (AI Agent)

Requirement: Zava wants to help customers and associates optimise store layouts using AI-driven insights.

Build Option:

Develop a bespoke AI agent using Azure Cognitive Services and custom logic for Zava’s unique store formats.

Buy Option:

Purchase a retail analytics tool with layout optimisation features.

Decision Process:

Innovation: Zava’s stores have unique layouts and customer flows, so off-the-shelf tools may not fit well.

Data Sensitivity: Building in-house ensures customer and store data remain within Zava’s secure environment.

Time-to-Value: Buying could provide a quick pilot, but may require significant adaptation.

Outcome: Zava pilots a commercial tool but ultimately builds a tailored solution for full rollout.

New Frontend Design

A screenshot of a computer

AI-generated content may be incorrect.

Architecture Methodology and Views

We follow a **well-known architecture methodology** to produce the necessary documents and artefacts. In particular, we apply the **C4 model** for software architecture:

* **Context (Level 1):** High-level system scope and user/system environment.
* **Containers (Level 2):** Major technical building blocks (applications, services, databases) and how they interact.
* **Components (Level 3):** Internal components within each container and their relationships.
* **(Code (Level 4):** [Optional for detailed design] Specific implementation details, if needed.)

This approach is widely used to communicate architecture clearly to both technical and non-technical stakeholders. It ensures we document the system from different perspectives with consistent detail. Accordingly, the architecture documentation will include C4 diagrams (context, container, component) and supporting text for each view. We also adopt elements of the “4+1” view model by discussing runtime scenarios and deployment in addition to static structure.

The remainder of this document is organised as follows: an **Architecture Overview** (context and key design decisions), a **Solution Architecture** breakdown (containers and components of the system), and specific considerations for **Authentication** and **Deployment** (including CI/CD). Tables and diagrams are provided for clarity.

Use Cases

**Personalised Product Recommendation**

* Actors: Shopper ↔ Shopper Agent
* Description: AI agent provides personalized product suggestions based on customer preferences and behavior

**In-Store Layout Guidance**

* Actors: Shopper, Store Associate ↔ Interior Design Agent
* Description: AI assists with optimizing store layout and guiding customers to products

**Loyalty Programme Management**

* Actors: Loyalty Programme Manager, Shopper ↔ Customer Loyalty Agent
* Description: Manages customer loyalty programs, points, and rewards

**Real-Time Inventory Check**

* Actors: Inventory Manager, Store Associate, Shopper ↔ Inventory Agent
* Description: Provides real-time inventory status and availability information

**Customer Query Handling**

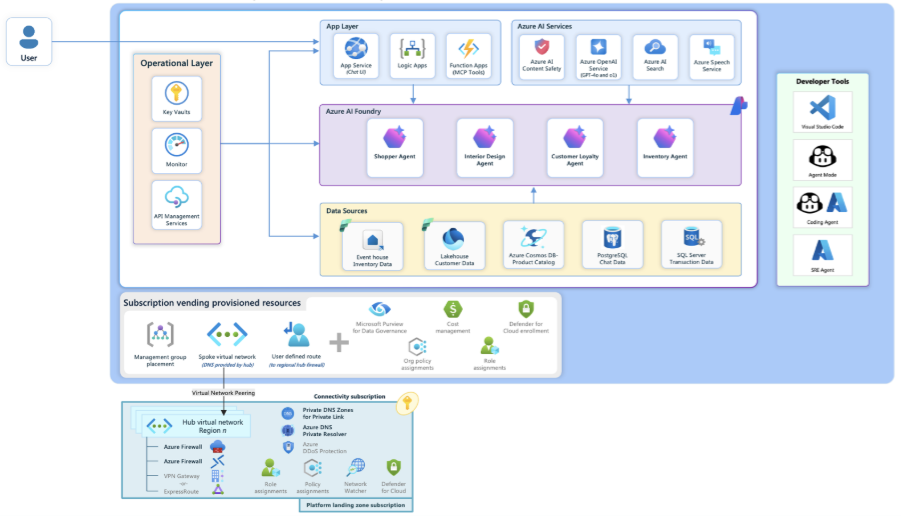
* Actors: Shopper ↔ Shopper Agent
* Description: AI-powered customer service for handling inquiries and support

**Offer Triggering at Checkout**

* Actors: Shopper, Store Associate ↔ Customer Loyalty Agent
* Description: Automatically triggers relevant offers and promotions during checkout process



Architecture Overview



Zava’s envisioned solution is a **cloud-enabled, multi-channel retail system** that connects customers, web/mobile frontends, backend services, and supporting cloud services. The attached architecture diagram illustrates the end-to-end customer experience powered by cloud and AI. In broad strokes:

* **Users (Customers)** interact via a React-based web application (and optionally a mobile app), browsing products, receiving recommendations, and making purchases. They are served personalised and seamless experiences, whether online or in-store.
* **Frontend Applications** (the web app, possibly store kiosks or mobile apps) connect to **backend services** running on Azure. The backend is built with C# (.NET), exposing APIs that the frontends call. This backend encapsulates business logic for product catalogue, shopping cart, orders, etc.
* An **AI/Agent layer** complements the core application. Zava leverages Azure AI services to enhance user interactions – for example, the “Shopper Agent” can converse with customers to guide product selection. These agents use advanced GPT-4-based models to interpret user inputs (including images or free text) and query data (like product inventory or customer history) in real time. This is inspired by Zava’s demo scenario of a proactive AI-powered shopping assistant.
* **Data and Integration**: The system’s data is stored in cloud databases. The primary data store is a relational database (Azure PostgreSQL) holding structured data like product info, inventory, orders, and customer profiles. PostgreSQL was chosen as Zava’s backbone for its openness and flexibility. Additionally, Azure Cosmos DB is used for certain scenarios requiring ultra-fast, scalable access (for instance, the loyalty/rewards chat application which handles spikes in traffic and rapidly evolving data models). Legacy systems (like an ERP or inventory management system) are integrated via connectors or periodic data syncs, ensuring the new platform can retrieve up-to-date inventory or financial data as needed.
* **Security and Identity**: The entire ecosystem is secured by Microsoft Entra ID (formerly Azure AD) for authentication and authorisation. Customers and employees sign in through Entra ID, which provides tokens used to access the app’s APIs. Entra ID enforces identity policies and can integrate with social logins or external identities if needed for a B2C context. In addition, a governance layer using Microsoft Purview provides data classification and auditing to maintain compliance (useful for handling customer personal data), and Azure Defender for Cloud monitors and secures the cloud resources3. Network security groups, encryption at rest, and other Azure security best practices are in place to protect data and services.
* **User Journey Example (Scenario + Agents):** A typical scenario can illustrate the interplay of components. *For example:* A customer named Bruno uses the Zava web app to get paint advice. He uploads a photo of his living room and asks which finish (eggshell vs semi-gloss) is better given the lighting. The **Shopper Agent** in the backend (powered by a multimodal AI model) analyses the photo and question, then queries the **product catalogue** for paint products. It might also call an **inventory service** to check availability at Bruno’s local store. Within seconds, Bruno receives a tailored recommendation via the React frontend: a suggestion of a specific paint product (with explanation), confident that it’s in stock nearby. Behind the scenes, the agent’s logic is orchestrated on Azure AI services, but to Bruno it feels like a natural conversation with an expert. If Bruno decides to purchase, he proceeds through a secure checkout, with Entra ID ensuring his identity (or guest login) and the backend processing the order and updating inventory.

This overview shows how **various subsystems integrate**: a responsive frontend, a robust API backend, intelligent services, and strong security. Next, we break down the architecture into its constituent components and layers.

Solution Architecture

Using the C4 **Container view**, the system comprises several main deployable components (containers). The table below summarises the key components, the technologies chosen, and their roles in the Zava solution:

|  |  |  |
| --- | --- | --- |
| Component / Layer | Technology Stack | Purpose and Responsibilities |
| Web Frontend (SPA) | React (TypeScript), Tailwind CSS, shadcn/ui library | Presents the user interface in browsers. Provides dynamic, responsive UX for product browsing, search, recommendations, and checkout. Tailwind and shadcn/ui ensure a modern, consistent design. The SPA communicates with backend APIs and handles user authentication via Entra ID. |
| Mobile App (future) | React Native or Xamarin (C#) | *(Optional)* Mobile channel for customers. Would reuse backend APIs. Not detailed in this phase, but architecture supports adding mobile frontends later. |
| Backend API | C# .NET (ASP.NET Core Web API) | Implements business logic and exposes RESTful endpoints to frontends. Handles product catalog queries, shopping cart, orders, etc. Also orchestrates calls to sub-services (e.g. the AI agents or external APIs). Utilises Entra ID JWT validation for secure access. |
| AI Agents & Services | Azure OpenAI (GPT-4 models), Azure AI Foundry, Custom Python/C# services | Enhances user experience with AI. E.g., “Shopper Agent” for product Q&A, “Interior Design Agent” for style suggestions, “Loyalty Agent” for rewards info. These run as microservices or workflows that the backend API can invoke. They use Azure OpenAI for natural language and may access data via the backend or direct DB queries for relevant info4. |
| Database – Product & Orders | Azure Database for PostgreSQL | Central relational database for core retail data: product catalogue, inventory levels, customer info, orders, etc. Chosen for compatibility with open-source tools and to let Zava leverage SQL skills. Supports ACID transactions (e.g. order placement) and can scale read replicas for heavy read scenarios. |
| Database – NoSQL | Azure Cosmos DB (with MongoDB API or Table API) | High-scale datastore for specific needs – e.g., session data, user activity logs, or the Customer Rewards chat app data. Cosmos provides low-latency access globally and flexible schema, allowing rapid iteration of features6. Used where schemaless or massive scale is required alongside the structured Postgres store. |
| Authentication/Identity | **Entra ID** (Azure AD B2C tenant) | Securely manages user identities (customers can register/login). Issues OAuth2/OpenID Connect tokens used by the React app and API. Simplified integration via MSAL in frontend and JWT validation middleware in backend. Supports social logins and MFA if enabled. |
| Authorisation & Roles | Entra ID, .NET AuthZ | Enforces access control. The API uses Entra ID roles or app roles to authorise admin vs customer actions. For B2C, could use Entra ID External Identities. The design keeps authorisation checks centralised in the backend. |
| Storage & CDN | Azure Blob Storage and Azure CDN | Stores static content (images, product photos) and serves them efficiently. Product images uploaded via an admin interface go to Blob Storage; CDN ensures fast delivery to users globally. |
| Monitoring & Analytics | Azure Application Insights, Log Analytics | Collects telemetry (performance metrics, errors) from the backend and frontend (via App Insights JavaScript SDK). Helps track usage patterns and diagnose issues. |
| Security & Governance | Microsoft Purview, Defender for Cloud, Key Vault | Ensures compliance and security. Purview tracks sensitive data usage and classifications (e.g. GDPR-related personal data)7. Defender for Cloud monitors for vulnerabilities and threats across Azure resources. Key Vault stores secrets/keys (like DB connection strings, API keys) securely for the application. |

Each of these components interacts through well-defined interfaces. The **React SPA** communicates with the **.NET API** via HTTPS (JSON/REST). The **API** in turn queries the **PostgreSQL** database (using an ORM or SQL queries) and uses the **Cosmos DB** for specific data. It also calls out to **AI services** when needed – either by invoking Azure OpenAI endpoints or by delegating tasks to specialised agent services. All network calls are secured (HTTPS/TLS) and require valid credentials/tokens.

**Internal component decomposition (C4 Component view):** Within the **Backend API**, there are likely modules such as *ProductService*, *OrderService*, *UserService*, etc., corresponding to different domains of functionality. For example, ProductService might handle searches and recommendations (possibly integrating with an AI component or Azure Cognitive Search), OrderService handles checkout, payment (which might involve calling an external payment gateway API, not shown above), etc. We also anticipate a *Recommendation Engine* component that works with the Shopper Agent – e.g., calling Azure AI to get recommended items and possibly storing recent recommendation results in cache.

The **AI Agent layer** might be composed of multiple microservices or Azure Functions, each corresponding to a specific agent:

* *Shopper Agent Service*: Receives a query (text or image) from a customer via the API, calls an OpenAI model (e.g. Azure OpenAI GPT-4) with appropriate prompt and context (product data) and returns a response (product suggestion with reasoning).
* *Interior Design Agent*: Similar pattern, focusing on style advice (maybe uses Azure Cognitive Services for image recognition).
* *Loyalty/Rewards Agent*: Interfaces with loyalty data (maybe stored in Cosmos DB) to answer questions about points or offers.  
  These services could be implemented in Python or C# and deployed as Azure Functions or containerised microservices, depending on complexity. They use **Model Context Protocol (MCP)** or similar techniques to securely access backend data without exposing secrets to the model.

**Note:** The architecture allows evolving these AI components independently. They are integrated but loosely coupled – the backend API calls them via internal REST calls or Azure Queue messages so that failures or updates in AI services do not break core functionality.

Authentication Strategy

Authentication is designed to be simple for users yet secure and standard-based. We utilise **Microsoft Entra ID** as the identity provider across the app. For the B2C scenario, a dedicated Entra ID tenant (formerly Azure AD B2C) can handle customer identities. Key points of the auth strategy:

* **OpenID Connect Flow:** The React application uses MSAL (Microsoft Authentication Library) to sign users in via Entra ID. On first use, a customer is redirected to the **Entra ID hosted login page**, where they can sign up or sign in (using email/password or social accounts if configured). Entra ID then issues an **ID token** and **access token** for the application.
* **Token-Based API Security:** The React app attaches the acquired access token (a JWT) on every API request to the .NET backend (in the HTTP Authorization header). The backend API is configured to **validate JWT tokens** using Entra ID’s public keys and ensure the token is intended for this API (using the audience and issuer claims).
* **Simple roles:** As this is primarily customer-facing, we might not have complex role differentiation initially (all customers have similar rights in the app). However, Entra ID allows defining roles (e.g., “Admin” for Zava employees to manage catalogue). Role claims from the token can be checked in the API to protect admin-only endpoints.
* **No passwords stored in app:** All authentication is delegated to Entra ID; the app’s database does not hold user passwords, which reduces security risk. Entra ID also handles things like password resets, MFA (if we enable it), and account verification out-of-the-box.

The authentication integration can be summarised in the following flow:

|  |  |
| --- | --- |
| Step | Authentication Flow (Entra ID integration) |
| 1. | **User Initiates Login:** A customer clicking “Sign In” on the React app triggers MSAL to redirect the user to the Microsoft Entra ID authentication endpoint. The app specifies the scopes it needs (e.g. a custom API scope like *api://zavaretail/backend.access*). Entra ID’s hosted login page is presented. |
| 2. | **Entra ID Authentication:** The user enters credentials or uses a social login. Entra ID verifies the identity (and executes any policies like email signup or OAuth for Google/Facebook if using External Identities). Upon success, Entra ID generates an **ID Token** (with user info) and an **Access Token** for the Zava API, and redirects back to the React frontend’s redirect URI with these tokens. |
| 3. | **Token Storage:** MSAL in the React app receives the tokens and stores them (usually in browser local storage or memory) securely. The user is now considered authenticated in the SPA; the ID token can be used for display (e.g., show user’s name). |
| 4. | **API Call with Token:** When the React app needs to call the backend (for example, to get the user’s loyalty points or place an order), it attaches the **Access Token** in the request header (*Authorization: Bearer <token>*). |
| 5. | **Token Validation (Backend):** The .NET API uses Entra ID’s **JWKS (JSON Web Key Set)** to validate the incoming JWT. Azure’s *.NET JwtBearer* middleware automatically checks signature, expiration, audience, and issuer. It ensures the token was issued by our Entra ID tenant and is valid. If valid, the user’s identity (claims principal) is established in the API context. |
| 6. | **Authorisation (Backend):** For each API endpoint, the code can enforce any authorisation rules. For most customer endpoints, a valid login is enough. For any admin or privileged operation, the API would check for a role claim like *Admin* in the token. If the token lacks required privileges or is absent/invalid, the API returns an HTTP 401/403 error. |
| 7. | **Session Persistence:** The application is stateless in terms of auth sessions. The token represents the session. Tokens typically have an hour lifetime; MSAL will automatically use a refresh token or prompt login again as needed. This keeps the user logged in seamlessly without the backend storing any sessions. |
| 8. | **Sign-Out:** If the user clicks “Sign out”, the app clears the tokens and optionally triggers a Entra ID global logout (so Entra ID also ends the session). Next time the user or another person uses the app, they must log in again to get a new token. |

This simple flow covers authentication. By using Entra ID, we get a robust and standard solution with minimal custom code. In summary, **Entra ID provides a secure Identity-as-a-Service**, handling the heavy lifting of user authentication and allowing the Zava app to focus on its core functionality.

Deployment Architecture and CI/CD

The deployment of the Zava solution is designed with Azure cloud services for scalability and reliability. We also set up a **Continuous Integration/Continuous Deployment (CI/CD)** pipeline to automate builds, testing, and releases. This ensures that new features and fixes can be delivered to users rapidly and repeatedly with high confidence.

Hosting and Infrastructure

All major components are hosted in Azure:

* The **React frontend** is a static single-page app. We will build it and then deploy the static assets to **Azure Storage** (or an Azure Static Web Apps service, which automates a lot of this). Azure CDN is put in front of the static files for low-latency delivery worldwide. Alternatively, Azure Static Web Apps (which has integrated Entra ID support) can directly host the React app with a global endpoint.
* The **.NET API backend** is deployed to **Azure App Service** (Web App) or **Azure Container Apps**. For simplicity, an App Service (in Linux or Windows) is ideal for a monolithic API. It provides easy scaling (scale-out to multiple instances) and integrates with deployment slots (for staging). The App Service runs the C# application and is configured with managed identity or service principals to access Azure resources (like Key Vault or databases). We will enable autoscale rules – e.g., if CPU usage or HTTP queue length grows, additional instances spin up – to handle higher loads.
* **Databases**: Azure Database for PostgreSQL is provisioned (likely a flexible server or managed instance) in the same region. We enable performance features like read replicas or scaling vCores as needed for load. Azure Cosmos DB is a fully managed service; it will automatically scale throughput (we can use autoscale throughput settings) to handle spikes for the usage it gets.
* **AI Services**: If using Azure OpenAI, this is provided as a service (the model inference happens in Azure’s managed environment). Our agent services, if any custom code, could run as Azure Functions or container instances. For instance, a Python-based AI agent might be deployed as an Azure Function with a consumption plan (scales out on demand).
* **Networking**: We use Azure’s virtual network integration for App Service to allow secure connections to the PostgreSQL (if using private endpoints) and to ensure all traffic to databases stays within Azure. Public endpoints (for frontend and API) are protected by HTTPS and Azure Front Door could be introduced if we needed global routing or web application firewall features. But initially, a simpler setup is acceptable (HTTPS endpoints with App Service’s built-in domain or a custom domain for Zava).
* **Scalability**: Both the web app and API can scale independently. The React app on CDN can handle virtually unlimited traffic. The App Service can scale up (more CPU/RAM per instance) or out (more instances) depending on load. Azure Monitor auto-scaling rules will keep response times smooth during peak (for example, evening shopping hours).
* **High Availability**: We deploy multi-zone (if available in the region) or have at least multiple instances so that if one server or zone goes down, the service stays up. Azure App Service takes care of distributing instances across fault domains. The database services have high availability features (PostgreSQL can be configured with geo-redundant backup or failover, Cosmos DB is multi-region by design if enabled).
* **CI/CD Pipeline**: We use GitHub or Azure DevOps for our code repository. Every commit triggers a CI build:
  + For **frontend**, run *npm build* to produce optimised static files. Then, publish the build artifacts (the *build* folder).
  + For **backend**, compile the .NET solution, run unit tests, and publish the build (the compiled DLLs or a Docker image if containerised).
* After CI succeeds, a CD pipeline (using GitHub Actions or Azure Pipelines) deploys the artifacts:
  1. Deploy frontend to Storage/Static Web App. For Static Web Apps, there is an action that does this directly. For Storage+CDN, we use Azure CLI/PowerShell to upload files to Blob storage and purge the CDN cache.
  2. Deploy backend API to App Service. If using Azure Web Apps, the pipeline can use the Azure WebApp Deploy action to push the new version (or swap slots if using a staging slot). If using containers, push the image to Azure Container Registry, then update the Container App or App Service to pull the new image.
  3. Run integration tests or smoke tests. After deployment to a staging environment, automated tests hit a few endpoints to ensure everything is working (for example, a ping to the health check endpoint, or a test login).
  4. Approval and production release. Assuming tests pass, the release can automatically swap the staging slot to production (zero-downtime deployment), or a manager can approve a promotion if required.
* **Observability**: Application Insights is connected; after deployment, we monitor the live metrics. If errors are detected post-deployment, we have the option to quickly rollback (App Service deployment slots allow easy rollback to previous). Logging from App Service and Functions goes to Azure Monitor Logs.

The table below outlines a simplified deployment pipeline flow:

|  |  |
| --- | --- |
| Stage / Environment | Deployment Actions & Tools |
| Source | Developers push code to the main branch in GitHub/Azure DevOps. Each push triggers the CI pipeline. Branch policies ensure code review and testing before merge. |
| CI – Build & Test | **GitHub Actions / Azure Pipelines:** • **Build Frontend:** Install npm dependencies and run *npm run build*. Artifact: static web assets. • **Build Backend:** Restore NuGet packages, compile the .NET solution, run unit tests (*dotnet test*). Artifact: app bundle or Docker image. • **Quality Gates:** If any test fails or build fails, pipeline stops and notifies the team. Successful artifacts are versioned. |
| CD – Staging Deploy | **Automatic Deployment to Staging:** • **Frontend:** Deploy React app artifacts to an **Azure Storage account (static website)**. This can be done with Azure CLI (using *az storage blob upload-batch*). Or, if using **Azure Static Web Apps**, a dedicated action builds and deploys directly. • **Backend API:** Deploy to **Azure App Service (Staging slot)**. Using the Azure Web App Deploy action with the web package or by updating the container image tag. Configuration settings (like DB connection strings, which are stored in Azure App Service settings or Key Vault) are already present. • **Database:** (Usually the database schema changes are rare; if needed, a migration step would run here, e.g., using Entity Framework migrations to update the PostgreSQL schema). |
| Verification | **Smoke Tests:** After staging deploy, run basic tests: e.g., call endpoint, attempt a test login using a dummy account, fetch a sample product. Use Postman/Newman or integration test scripts. **Visual Check:** The team or automated UI tests ensure the staging front-end is functional (perhaps using Cypress or Playwright to load the site). |
| Production Release | **Approval Gate:** A lead or release manager approves promotion to production (this can be automatic if all tests green, or scheduled during a low-traffic period). **Swap/Deploy:** For App Service, use slot swap – the new code in staging slot swaps into production slot with no downtime.8. The React app – promote the same static content to the production endpoint (if Static Web Apps, it’s already live; if using Storage+CDN, we might just point CDN at the new content or it’s same storage if we deployed directly to prod). **Post-Deployment:** Monitor App Service for any error spikes via Application Insights Live Metrics. Keep the previous deployment slot available in case rollback is needed. |
| Monitoring & Feedback | **Monitoring:** Azure Monitor alerts are set (e.g., alert on high error rate or CPU > 80% for 10 minutes). If an alert triggers, on-call engineers are notified (email/SMS/Teams). **Scaling:** If load increases, Azure will auto-scale the API App Service (based on our rules). Cosmos DB and other services auto-scale as configured. **Continuous Improvement:** The DevOps pipeline logs and application metrics are reviewed sprint by sprint to improve build/deploy times and reliability. |

Azure’s cloud-native services ensure that the application can **scale** to meet demand and remain **resilient**. For example, if the site faces a surge of traffic during a sale, App Service will scale out, and Cosmos DB’s throughput will scale up automatically to handle the spike, then scale down to control cost.

We also plan for **disaster recovery**: regular backups of the PostgreSQL database are enabled, and in a multi-region setup, we could deploy a secondary instance of the app in another Azure region for failover. At minimum, data is backed up, and the infrastructure is defined (we use Infrastructure-as-Code scripts for Azure resources) so it can be redeployed in a new region if needed.

Finally, our CI/CD process embraces the DevOps best practice of infrastructure as code and continuous testing. This reduces the chance of human error in deployments and allows the team to deliver improvements to the Zava platform quickly. With monitoring and automated alerts, we get rapid feedback on any issues in production, which closes the loop for continuous improvement.

## Architectural Decisions

|  |  |  |  |
| --- | --- | --- | --- |
| Decision Area | Context | Decision | Consequences |
| Multi-Agent Framework | Need for modular, scalable, intelligent platform | Adopt Azure AI Foundry with dedicated agents for shopper, design, loyalty, and inventory | Enables independent scaling and rapid evolution; increases orchestration complexity |
| Data Storage Technologies | Real-time analytics, flexible models, high availability | Use Fabric OneLake, Azure SQL, and Cosmos DB for different data needs | Low-latency, global scale; requires robust integration and consistency management |
| Security & Compliance | Sensitive customer data, GDPR and regulatory requirements | Enforce Entra ID, Microsoft Purview, Defender for Cloud | Strong security and compliance; adds operational overhead for policy management |
| Cloud-Native Deployment | Need for resilience, scalability, and frequent updates | Deploy all components as Azure cloud-native services; implement automated CI/CD pipelines | Rapid, reliable deployments; requires investment in DevOps tooling and practices |
| Event-Driven Integration | Agents must communicate efficiently and respond to real-time events | Use secure APIs and event-driven messaging (e.g., Event Grid, Service Bus) | Loose coupling and scalability; adds complexity in event management and error handling |
| Governance & Change Mgmt | Maintain alignment with business goals and technical standards | Establish Architecture Review Board; document exceptions and maintain a change log | Ensures consistency and traceability; may slow urgent changes due to review processes |

## Non-Functional Requirements

|  |  |
| --- | --- |
| NFR | Architectural Considerations |
| Availability | Deploy all critical services (API, database, AI agents) across multiple Azure availability zones to ensure uptime. |
| Backup & Recovery | Schedule automated daily backups for PostgreSQL and Cosmos DB; test restores quarterly to validate recovery process. |
| Capacity Estimates | Use Azure Monitor to track usage trends and auto-scale App Service and Cosmos DB throughput based on forecasted peaks. |
| Configuration Management | Store all configuration in Azure App Configuration and Key Vault, versioned and promoted through CI/CD pipelines. |
| Disaster Recovery | Implement geo-redundant backups and define a failover region in Azure for rapid recovery in case of regional outage. |
| Environmental Factors | Select Azure regions with renewable energy sources and ensure data centre compliance with EU environmental standards. |
| Extensibility/Flexibility | Design APIs using OpenAPI/Swagger and modularise backend services to allow easy addition of new retail features. |
| Failure Management | Integrate Application Insights and Azure Monitor alerts to detect failures and trigger automated remediation scripts. |
| Maintainability | Enforce code reviews, automated testing, and maintain up-to-date architecture diagrams in the project wiki. |
| Performance | Use Azure CDN for static assets and implement caching for product catalogue queries to minimise latency. |
| Quality of Service | Define SLAs for API response times (<300ms) and error rates (<0.1%), monitored via Azure Application Insights. |
| Reliability | Use health probes and auto-healing for App Service; enable database replication for consistent operation. |
| Scalability | Architect all stateless services to support horizontal scaling; use Cosmos DB’s auto-scale for unpredictable loads. |
| Security | Apply defence-in-depth: Entra ID for identity, Key Vault for secrets, and Defender for Cloud for threat detection. |
| Service Level Agreements | Monitor and report on uptime and support response times using Azure Service Health and custom dashboards. |
| Standards | Adhere to ISO 27001 for security and GDPR for data privacy; follow Microsoft’s Cloud Adoption Framework. |
| Systems Management | Centralise logging and monitoring in Azure Monitor; automate patching and updates for all managed services. |