Access Plus Ticketing Platform: US Market Design Document

This document outlines the design and architecture for the **Access Plus Ticketing Platform**, a highly specialized event management system built to serve the unique needs of the **US immigrant and diaspora communities**. It adapts the original technical framework to align with the US market, emphasizing compliance, cultural features, and scalability.

1. Introduction

The Access Plus Ticketing Platform is a web-based, mobile-first platform that allows users to discover and purchase event tickets online. The application is designed to be user-friendly and accessible to diverse communities across the United States. Users can search for events by location, category (now including cultural/diaspora tags), or keyword, view event details, and purchase tickets securely through integrated payment gateways.

2. System Summary (US Market Focus)

In the US, event organizers serving niche immigrant and diaspora communities struggle with generalist platforms that lack cultural focus, targeted marketing, and tools to simplify US regulatory compliance (like 1099-K reporting). The difficulty of efficiently marketing events to niche audiences and managing US financial requirements has proven cumbersome for both customers and event organizers.

The goal of the Access Plus Ticketing Platform is to solve these issues by providing a hyper-specialized, user-centric solution that makes purchasing tickets and organizing cultural events more seamless, accessible, and compliant online.

3. User Stories

The system must support three core user groups: Customer, Event Organizer, and Administrator.

Customer

- As a customer, I must have the ability to purchase tickets for any available diaspora events.

- As a customer, I must have the option to create a personal account or purchase tickets without an account.
- I must have the ability to purchase multiple tickets for a single event and for different events.
- I must have the ability to view all available events and filter them by category, date, location, and price.
- I must have the ability to contact customer support for any inquiries or issues related to my purchase.

Event Organizer (Enterprise Level)

- As an event organizer, I must have the ability to create and manage events through the site.
- I must have the ability to sell tickets to my events through the site.
- I must have the ability to view ticket sales and attendance for my events, **including generating reports that simplify US tax compliance (e.g., 1099-K data)**.
- I must have the ability to update event information and communicate with ticket purchasers regarding event updates or changes.

Administrator

- As an administrator, I must have the ability to manage user accounts and permissions.
- I must have the ability to manage event listings and ensure compliance with site policies.
- I must have the ability to manage payment and refund processes.

4. Architecture (Modular Monolith)

The system will employ a modular monolith architecture to ensure modularity, scalability, and fault tolerance. Each microservice will have its own database and can be independently scaled and deployed. The API Gateway serves as the entry point for all client requests.

Microservice	**Responsibility**
Authentication Service	User authentication and authorization, generating JWT tokens.
User Service	User profile management and account creation (Individual/Enterprise).

Event Service	Creating, updating, and deleting events, and managing event information.
Ticket Service	Creating, updating, and deleting tickets, managing ticket sales and availability.
Payment Service	Secure payment processing, communicating with the Ticket microservice to confirm availability.
Notification Service	Sending notifications to users (event reminders, payment confirmations).
Reporting Service	Generating reports on event attendance, ticket sales, and revenue (Critical for US tax compliance reporting).

5. System Quality Attributes & Constraints (US Specific)

Attribute	**US-Specific Requirement**
Performance	Response time for all user interactions should be **less than 2 seconds**, even under heavy load (e.g., high-demand event on-sale). Should handle at least **10,000 concurrent users**.
Availability	System should be highly available with minimal downtime to ensure tickets can be purchased at any time.
Security	Must use proper authentication and authorization. **Mandatory: Comply with Payment Card Industry Data Security Standards (PCI DSS) for handling payment data**.

Scalability	Must scale to accommodate increased traffic (e.g., expected **1000+ users on a given day**) using horizontal scaling and database sharding.
Regulatory Constraints	The application must adhere to all US legal and regulatory requirements, particularly those related to the ticketing industry, financial transactions, and **IRS Form 1099-K reporting**.

6. High Level Design

The high-level design uses a modular monolith architecture, with components communicating via APIs and a message broker (e.g., Kafka or RabbitMQ) for asynchronous communication.

- **Frontend:** Built using **React** to provide a user-friendly interface for customers and organizers.
- **Backend:** Handled by the microservices, implemented with **Java** and **Spring Boot**.
- **API Gateway:** Routes external client requests to the appropriate internal microservices.
- **Data Storage:** A combination of databases (e.g., **MySQL/PostgreSQL** or **Amazon RDS**) for structured data and caching (e.g., **Redis/Memcached**) for improved performance. Event images will use **Amazon S3**.

7. Implementation Stack & Future Improvements

Area	**Stacks and Tools**
Primary Language / Framework	Java with Spring Boot/Spring Cloud
Frontend	React
Database	PostgreSQL for relational data (considering sharding for scale)

Cloud Infrastructure	AWS, Google Cloud, or Microsoft Azure
Containerization / CI/CD	Docker for containerization, GitLab for Version Control/CI/CD
Future Improvements	Personalized event recommendations using AI, demand forecasting, and mobile app integration.

Differentiation Strategy: Targeting U.S. Immigrant Communities

To effectively differentiate a ticketing platform against major competitors like Eventbrite and Ticketmaster when targeting immigrant communities in the US, the strategy must move beyond transactional ticketing to address the specific **financial, cultural, and community-building needs** of the diaspora.

1. Financial & Cash Flow Advantage (Early Payouts & Flexible Payments)

The core competitor weakness is often delayed access to funds. Immigrant community organizers frequently run events with tight cash flow for artist deposits, venue rentals, and promotion.

- **Early/Instant Payouts:** Offer organizers **instant or accelerated payouts** (daily deposits of ticket sales) rather than the industry standard of waiting until the event is over. This provides crucial working capital and builds trust.
- **Non-Traditional Payment Gateways:** Integrate payment methods beyond standard US credit cards and PayPal. This includes support for:
    - **Mobile Money/Remittance Integration:** Allow attendees to pay using common cross-border payment or remittance apps, catering to a population that heavily utilizes these financial channels.
    - **Cash/Offline Payments:** Facilitate ticket purchases via designated, trusted community partners or a simple cash-at-door integration feature to serve the unbanked or underbanked segment.

2. Hyper-Local & Cultural Discovery

Mainstream platforms lack the necessary nuance and trust required to serve specific, often geographically dispersed, immigrant communities.

- **Multilingual Support:** Offer event creation and attendee-facing pages in key community languages (e.g., French, Spanish, various African languages) to lower barriers to entry for first-generation immigrants.
- **Curated Community Channels:** Instead of relying on a broad search, create **verified "Community Hubs"** or profiles for trusted local associations, cultural organizations, and community leaders. This leverages word-of-mouth and deep community trust for event promotion and discovery.
- **Cultural Taxonomy:** Implement highly specific event categorization and filtering (e.g., "Gala/Fundraiser," "National Day Celebration," "Afrobeat Concert," "Diaspora Film Screening") that standard platforms fail to capture

3. Diaspora Impact & Organizer Empowerment

Tap into the diaspora's strong desire to support their home country and local community.

- **Community Data Ownership:** Assure organizers that they retain **100% ownership and control** of their attendee data (names, emails). Unlike general platforms that may use this data for cross-promotion, this allows the organizer to build their own long-term, valuable community mailing list, empowering them as independent event entrepreneurs.
- **"Impact Fee" Feature:** Offer an optional feature to dedicate a small, transparent percentage (e.g., 1%) of the service fee to a **"Diaspora Development Fund"** verified by a community-led non-profit, allowing attendees to feel their ticket purchase contributes directly to a purpose beyond the event itself.
- **US Tax Support:** Provide streamlined reporting tools (e.g., 1099 form readiness for vendors/performers and sales tax breakdowns) to simplify the complex US tax and regulatory environment for community organizers who may be managing finances part-time.

Disadvantages & Risks of Early/Instant Payouts

Offering organizers access to their ticket revenue before the event takes place is the **highest-risk feature** you are proposing. It shifts the primary financial risk from the organizer to our platform.

Financial and Credit Risk

1. **High Chargeback/Refund Liability (The "Event Cancellation" Risk):**
    - **The Scenario:** An organizer is paid $10,000 instantly for tickets. The event is suddenly canceled, the venue is double-booked, or the organizer commits fraud and disappears.
    - **The Risk:** Attendees demand refunds. Since the money is already *outside* our platform (in the organizer's bank account), our platform is obligated to pay the attendees from its own reserves or operating funds to satisfy the credit card company/payment processor. This is a massive drain on cash flow and can quickly lead to insolvency if a large event fails.
2. **Fraud Exposure:** Instant payouts attract fraudulent actors (bad organizers) who create fake events, collect money rapidly, and vanish. Our platform becomes a target for "ticket flipping" scams where the event is never intended to be held.
3. **Working Capital Strain:** Even with a low fraud rate, a pool of money must be kept in reserve to cover the inevitable refunds and chargebacks. This *Reserve Fund* reduces the capital available for growth, marketing, and operations.

Operational and Regulatory Risk

1. **Need for Advanced Underwriting:** You cannot offer this to all organizers. You must build a sophisticated **Risk Score/Underwriting System** that analyzes:
    - The organizer's history (on our platform and others).
    - The event type and price (higher-risk events, like festivals, may require a holdback).
    - The organizer's KYC (Know Your Customer) documentation and financial stability.
    - Implementing and maintaining this system adds significant engineering and compliance costs.
2. **Increased Scrutiny from Payment Processors:** Payment gateways (like Stripe or PayPal) are sensitive to high chargeback rates. They may increase our transaction fees, mandate a higher rolling reserve (holding a percentage of our revenue), or even terminate our service if the refund risk is not contained.

Disadvantages & Risks of Community Data Ownership

Allowing organizers to retain 100% of attendee data (names, emails, etc.) removes a key asset that competing platforms use for their own growth and monetization.

Strategic and Business Risk

1. **Loss of Platform Network Effect (The Competitive Moat):**
    - **The Risk:** The data is our most valuable asset for customer retention. By giving it away, you lose the ability to cross-promote other events to the attendee. Attendees may find out about a community event through our platform once, but they will not return to our platform organically if they are directly marketed to by the organizer thereafter. This makes it easier for organizers to switch to a cheaper alternative once they've built their list on our platform.
2. **Reduced Lifetime Value (LTV) of the Customer:** The LTV of an attendee on our platform is drastically reduced because you cannot market repeat purchases to them across different events. This limits our profitability and makes customer acquisition more expensive.
3. **Diminished Platform Valuation:** Investors highly value the size and engagement of a platform's proprietary user database. A business that does not own its user data is generally viewed as having a lower valuation than one that does (like Eventbrite).

Compliance and Operational Risk

1. **Data Misuse and Reputational Harm:**
    - **The Scenario:** An organizer downloads the data and then uses it for non-event purposes, sells it, or spams the attendees.
    - **The Risk:** While the liability might legally fall on the organizer, the attendee purchased the ticket through our platform. A data breach or misuse event by an organizer will damage **our brand's reputation** within the community, especially a trust-sensitive immigrant community.
2. **Complicated Data Export/Privacy Compliance:** You must ensure the data export feature is robust, secure, and compliant with privacy regulations (CCPA, GDPR, etc.). If our Terms of Service grant the organizer ownership, our platform is still responsible for the secure *transfer* and must log the consent status of the attendee before transferring the data.

The customization feature, particularly when positioned as a **White-Label Solution**, is the most effective way to align our platform with the communal and trust-based nature of the US immigrant community market.

This approach transforms our platform from a transaction utility into a powerful, branded tool for community leaders, large organizations, and religious groups.

1. The Unique Proposition of Customization (White-Label)

The unique proposition of deep customization is that our platform becomes **"invisible"** to the end-user (the attendee), and the organizer's trusted community brand is put front-and-center. This addresses the deep cultural preference for direct engagement and trust within immigrant communities.

Feature Level	Unique Proposition	Community Value
Branding Control	The organizer can use their own **custom domain** (e.g., tickets.their-organization.org).	Attendees never feel they are buying from a third-party company; they are interacting directly with their trusted community group, preserving cultural authenticity.
Look & Feel	Full control over **fonts, colors, backgrounds, embedded videos, and page layout** (HTML/CSS control).	Allows the event page to match the organization's existing website and brand identity, offering a seamless, professional experience without requiring custom web development.
API/Integration	**Robust API services** allow deep system integrations.	Enables large organizations (e.g., a major cultural center or diaspora association) to integrate ticket sales data directly into their existing CRM or accounting systems, making the platform a core piece of their infrastructure.

2. Expansion Beyond Event Tickets

By offering a customizable platform, we can expand our value proposition far beyond simple ticket sales to become the central **"Mobile Box Office"** for the community.

Expansion Area	Feature Examples	Application for Diaspora Communities
Fundraising & Donations	Integrated donation options during checkout, fundraising thermometers, and tiered sponsorship packages.	Allows cultural centers and mosques/churches to run annual fundraising drives seamlessly alongside ticketed events, capturing additional revenue.
Merchandise & Upsells	In-checkout upsells for event merchandise, concessions, and add-ons.	Enables organizers to pre-sell traditional clothing, religious items, books, or community cookbooks alongside event admission, boosting their total revenue per attendee.
Virtual/Hybrid Access	Secured virtual event access, live streaming integration, and video-on-demand.	Critical for maintaining community connection for members living far away or those in the diaspora who want to attend events in their home country (Gambia/neighboring regions).
Subscription & Passes	Offering season passes, subscriptions, flex-passes, and group ticket bundles.	Ideal for small community theaters, annual cultural festivals, or religious institutions that charge annual membership fees.

3. Recommended Charging Structure (Fees)

A successful pricing strategy for customization uses a **Hybrid Model** to balance transaction volume with the high-value features of a white-label solution.

This structure allows you to earn revenue from all users while charging a premium for the organizers who receive the deep customization features.

A. Core Transaction Fee (For All Users)

This fee covers payment processing, customer service, and the core ticketing service.

- **Model:** Percentage of Ticket Price **or** Flat Fee per Ticket.
- **Recommendation:** Use a low-to-competitive **percentage fee (e.g., 2.5% to 5%)** or a **flat fee per ticket (e.g., $1.00 - $1.50)**.
- **Who Pays:** This fee should be transparently structured to be passed on to the ticket purchaser, keeping the platform **"free" to the organizer** and competitive with major ticketing platforms.

B. Customization Fee (Tiered SaaS Subscription)

This fee is charged to the organizer (our client) to unlock the customization and high-value features. This is where you monetize the **"white-label"** value.

Tier Name	Cost Structure	Features Unlocked (Value)
Basic	**$0/month** + Core Transaction Fee	Branded event page, basic color theme control, data export (as per Community Data Ownership promise).
Pro (The Custom Tier)	**Monthly/Annual Subscription Fee** (e.g., $99/month)	**Custom Domain Mapping** (No platform branding visible), full HTML/CSS control, merchandise/add-ons, priority support, and *premium* reporting/analytics.
Enterprise (The White-Label Tier)	**Custom Quote/Annual Contract** (e.g., $3,000+ per year)	Everything in Pro, plus: **API Access** for seamless integration, dedicated account manager, ability to host the software on a separate server, and custom feature development.

Key Benefit of the Hybrid Model:

- We capture all community organizers (from small to large) by offering a competitive free/basic tier.
- We earn a recurring, predictable revenue stream from the largest, most stable organizations (large associations) who are willing to pay a subscription for the powerful customization tools.

The strategy is to implement the core value (ticketing) while introducing the unique differentiators in a low-risk, scalable way.

Phase 0: Minimum Viable Product (MVP)

The MVP is designed to test the core value proposition: **"Can we sell tickets easily, and do organizers value our unique offer enough to switch?"**

MVP Core Feature Set (Ticketing)

Component	Features to Implement	Purpose & Link to File
Organizer Portal	**1. Simple Event Creation:** Title, Date/Time, Location, Description, Image Upload. **2. Basic Ticket Tiers:** Create 1-3 fixed-price ticket types (e.g., General, VIP).	Enable an organizer to go from sign-up to a live ticket page in under 5 minutes.
Customer Experience	**1. Event Discovery:** A simple list view of all events. **2. Purchase Flow:** Guest checkout (no account required) or simple account creation. **3. Confirmation:** Email with a scannable QR code ticket.	Ensure a frictionless, fast purchase experience, appealing to a potentially less-technical audience.
Event Operations	**Check-in App (Simple):** A basic mobile view for the organizer to scan and validate QR codes at the door.	Deliver on the core utility: successful event entry.

MVP Implementation of Differentiators (Low-Risk)

1. **Community Data Ownership:**
    - **Implementation:** A single, prominent button in the Organizer Portal that allows a
    - **full export of all attendee data** (Name, Email, Ticket Type) as a .CSV file.
    - **Risk Mitigation:** Define clear Terms of Service requiring organizers to comply with privacy laws regarding the exported data.

2. **Early/Instant Payouts:**
    - **Implementation: Standard T+2 Payout** (2 days *after* the event) for 95% of organizers.
    - **Limited Pilot:** Offer "Early Payouts" (e.g., 50% upfront, rest T+2) to a **hand-picked group of 3-5 trusted, established community organizers** as a pilot program. This tests the financial mechanics and risk control without exposing the entire platform to high chargeback risk.
3. **Customization (Basic Branding):**
    - **Implementation:** Allow organizers to **upload a logo** and select a **primary accent color** for their event page and ticket confirmation emails.
    - **Risk Mitigation:** The URL remains our platform's URL (e.g., ourplatform.com/event/XYZ).

The introduction of Artificial Intelligence (AI) features can significantly enhance our ticketing platform, providing a technological edge that supports our core differentiators: **Customization, Community Data Ownership, and Risk Management.**

1. AI for Organizer Success & Customization

These features leverage AI to make event creation more effective and increase ticket sales, justifying the premium tiers of your White-Label subscription.

A. Dynamic Pricing Engine

- **Feature:** An AI model analyzes historical sales data, ticket tier velocity, event capacity, day of the week, and local competitive events to automatically suggest optimal pricing changes (e.g., when to trigger the end of an "Early Bird" price or offer a flash discount).
- **Community Value:** Helps organizers, who may lack sophisticated business analysis skills, maximize revenue without manual effort. It turns data ownership into actionable, profitable insights.

B. Intelligent Event Description Generation

- **Feature:** Organizers input basic event details (date, time, performers). The AI generates compelling, SEO-optimized event descriptions and social media copy, tailored to the specific language or cultural nuances of the targeted immigrant community.
- **Community Value:** Lowers the barrier to creating professional, high-converting event pages for smaller, volunteer-run community groups.

C. Automated Marketing Segment Builder

- **Feature:** Based on the organizer's historical attendee data (which they own), the AI automatically segments the list into categories like "VIP Buyers," "Frequent Attendees," or "Last-Minute Purchasers."
- **Community Value:** Enables hyper-targeted communication (which the organizer controls) for personalized outreach, respecting the close-knit nature of diaspora groups while maximizing ticket sales.

2. AI for Attendee Experience & Discovery

These features improve user engagement and make it easier for community members to find relevant events, enhancing the overall platform utility.

A. Hyper-Personalized Event Recommendations

- **Feature:** Uses machine learning to analyze an attendee's past purchases, events they've viewed, and the events purchased by others in their **specific community segment** (e.g., based on language, neighborhood, or organization affiliation).
- **Community Value:** Moves beyond generic recommendations to suggest events that are genuinely relevant to the attendee's cultural identity or community group, fostering a sense of belonging and increasing repeat visits.

B. Natural Language Search & Filtering

- **Feature:** Allows attendees to search using conversational language (e.g., "Find comedy shows near me next weekend" or "Are there any Gambian community events next month?").
- **Community Value:** Improves accessibility for users who may be more comfortable with natural conversation than precise keyword searching, or for finding events tied to specific cultural names or holidays.

3. AI for Risk & Operations Management

These features are essential for safely scaling the **Early/Instant Payouts** differentiator, protecting your financial assets.

A. Organizer Risk Scoring and Fraud Detection (Crucial for Early Payouts)

- **Feature:** A model that calculates a real-time **Risk Score** for every event and organizer based on multiple factors:
    - **Historical Data:** Past chargeback rates, refund requests, and event success history.

    - **Behavioral Signals:** Speed of event setup, ticket price spikes, and discrepancies in event details.
    - **External Data:** Verification of organizer identity and social media presence (if permitted).
- **Operations Value:** This score determines the organizer's **Tiered Payout Level** (e.g., a score over 85 gets 100% instant payout; a score under 50 requires standard T+2 payout). This is the technological lynchpin for safely offering instant payouts at scale.

B. Check-in App Anomaly Detection

- **Feature:** Uses machine learning within the native Check-in App (or as a backend service) to flag unusual activity patterns, such as:
    - An unusually high volume of scans at a location outside the venue coordinates.
    - Multiple instances of the same "Already Scanned" ticket being attempted within a short time frame across different devices.
- **Operations Value:** Detects organized fraud or ticket scalping in real-time at the door, securing the event and protecting the organizer's revenue.

AI Feature Implementation Roadmap

These AI features should be phased in over the platform's lifecycle, starting with the most critical:

Iteration	Feature Focus	Status	Rationale
MVP/Iteration 1	**Organizer Risk Scoring (A. 3)**	**Pilot**	This is a non-negotiable feature for scaling our high-risk **Early/Instant Payouts** differentiator safely. Start simple and refine.
Iteration 2	**Personalized Recommendations (A. 2)**	**Build**	Focus on user engagement. Utilize the customer data you are collecting to deliver immediate value to attendees and drive repeat business.
Iteration 3	**Dynamic Pricing (A. 1)**	**Build**	Once you have enough sales data, this feature directly enhances organizer revenue, making the platform

			indispensable and justifying the premium subscription tier.

Iteration Roadmap

Iteration 1: Automated Risk & Payout Scaling

The focus shifts to scaling the high-risk payout feature and enhancing platform professionalism.

Focus Area	Features to Implement	Strategic Goal
Risk Management	**Automated Risk Scoring:** Develop an internal algorithm that scores organizers based on history, ticket price, event type, and documentation (KYC).	Allow the **Early/Instant Payouts** feature to be offered automatically to organizers with high trust scores, replacing the manual pilot program.
Payout Expansion	**Tiered Early Payouts:** Introduce 3 levels of payout speed tied to the risk score (e.g., 25% upfront, 50% upfront, 100% upfront).	Increase adoption of the key differentiator while actively managing chargeback liability.
Customization	**Basic Theming Upgrade:** Introduce pre-set templates and allow organizers to embed a promotional video/YouTube link on their event page.	Improve the quality and professionalism of event pages, increasing ticket sales conversion.
Data Utility	**Basic Organizer CRM:** Add a simple tool for organizers to email their ticket holders directly from our	Demonstrate the value of keeping the data within the community ecosystem, reducing reliance on raw data exports.

	platform (instead of just exporting the data).	

Iteration 2: White-Label & Vertical Expansion

The focus shifts to high-value, recurring revenue features that attract large community organizations.

Focus Area	Features to Implement	Strategic Goal
White-Label SaaS	**Custom Domain Mapping:** Allow organizers to use their own URL (e.g., tickets.churchname.org).	Launch the **Pro/Subscription Tier** to secure predictable, non-transactional revenue from premium organizers.
Vertical Expansion	**Add-Ons & Fundraising:** Integrate in-checkout upsells for merchandise, food, and specific donation/fundraising options.	Expand the platform's utility beyond tickets to capture a higher share of the community organization's total revenue (LTV).
Data Integration	**API Access (Beta):** Offer a limited API access for Enterprise clients to sync sales data into their own systems.	Attract the largest and most valuable community anchors by providing enterprise-level integration.

Iteration 3: Platform Maturity & Geo-Expansion

The focus is on optimizing the platform, increasing reach, and securing competitive advantage.

Focus Area	Features to Implement	Strategic Goal
Global Scaling	**Multi-Region Support:** Implement support for local payment gateways and multiple currencies, specifically for **neighboring regions** as per our business plan.	Execute on the long-term vision of serving the broader diaspora community, generating new revenue streams.
Advanced Tools	**Reserved Seating & Dynamic Pricing:** Implement tools for creating custom seating charts and automated tiered pricing (e.g., Early Bird discounts that expire).	Cater to complex events like weddings, galas, and conferences, increasing the platform's versatility and pricing power.
Ecosystem Building	**Loyalty/Subscription:** Introduce features for season passes, flex-passes, or recurring membership payments.	Create a deeper moat by locking in organizer loyalty through essential recurring revenue tools.

Technology Stack

Backend:
- Spring Boot 3.5.7 (Java 21) with Gradle multi-module setup
- PostgreSQL 16 database with Flyway migrations
- AWS services: Cognito (auth), S3 (storage), SQS (queuing), SES (email), SNS (SMS)
- Spring Security + JWT authentication

Frontend:
- React 19 + TypeScript + Vite
- Redux Toolkit for state management
- Tailwind CSS + shadcn/ui components
- AWS Cognito for authentication

Infrastructure:
- AWS ECS Fargate for containerized deployment
- Terraform for Infrastructure as Code
- Docker Compose + LocalStack for local development
- GitLab CI/CD for automated deployments

Architecture: Modular Monolith

The backend is organized into 6 modules that share a single database:

1. eventpro-core - Users, authentication, security, AWS Cognito integration ✅ Complete
2. eventpro-event - Event and ticket management ⏳ In progress
3. eventpro-order - Cart, orders, checkout ⏳ In progress
4. eventpro-payment - Stripe payment processing ⏳ In progress
5. eventpro-notification - Email, SMS, WebSocket notifications ⏳ In progress
6. eventpro-api - Main application with REST controllers and migrations ✅ Complete

Key Directory Structure

```
eventpro-site/
├── backend/modules/          # Spring Boot modules
├── frontend/src/             # React application
├── infrastructure/           # Terraform (VPC, RDS, ECS, CloudFront, etc.)
├── docker-compose.yml        # Local development setup
├── Makefile                  # Build automation
└── .gitlab-ci.yml            # CI/CD pipeline
```

Authentication Flow

1. User logs in via frontend → AWS Cognito
2. Cognito returns JWT token → stored in Redux
3. API requests include JWT in headers
4. Backend validates JWT → auto-syncs user to database on first login
5. Spring Security enforces role-based access (USER, ORGANIZER, ADMIN)

Database Design

A single PostgreSQL database with 12 tables:
- Users, Roles, UserRoles
- Events, Tickets, EventCategories
- Orders, OrderItems, Payments
- ShoppingCart, Notifications, NotificationPreferences

All managed through Flyway migrations with comprehensive indexes and constraints.

Notable Design Decisions

- Migrated from microservices to modular monolith → 52% cost reduction
- LocalStack for local development → work offline with AWS services
- Single database → simpler transactions, can split later if needed
- AWS Cognito → external identity provider, no custom user management
- Terraform modules → reusable infrastructure components

Local Development

```
make local-infra-only  # Start PostgreSQL + LocalStack
make local-infra       # Provision AWS resources via Terraform
make local-up          # Start backend + frontend with hot reload
```

The project is well-structured with solid foundations in place. The core authentication and user management are complete, with the remaining business logic modules (events, orders, payments,notifications) currently being implemented.

PostgreSQL (Relational)

PostgreSQL excels at structured data with complex relationships. It provides ACID transactions, which guarantee data consistency. For event ticketing, this means if you're processing an order, you can atomically update inventory, create the order record, and record the payment in one transaction—all or nothing.

DynamoDB (NoSQL)

DynamoDB is a fully managed key-value store optimized for high throughput and horizontal scaling. It trades some consistency guarantees for extreme scalability. It uses eventual consistency by default, though you can request strongly consistent reads at a performance cost.

For Event Ticketing—PostgreSQL is Better, Here's Why:

1. **Inventory Management** - You need strong consistency. If two people buy the last ticket simultaneously, exactly one should succeed. With DynamoDB's eventual consistency, both could initially think they succeeded, then one fails later. Nightmare scenario at scale.
2. **Complex Queries** - Ticketing requires searching: "Show me events in Seattle next week under $50" or "List all sold-out events." PostgreSQL handles these joins and filters elegantly. DynamoDB requires you to denormalize heavily and maintain multiple indexes.
3. **Transaction Safety** - When someone purchases tickets, you atomically: debit their payment, reserve seats, send confirmation, update inventory. PostgreSQL guarantees this works correctly. DynamoDB would require complex compensation logic if steps fail.
4. **Reporting** - Ticketing platforms need analytics: revenue by event, refund patterns, customer lifetime value. PostgreSQL's query flexibility makes this straightforward. DynamoDB requires pre-computing and storing aggregates.
5. **Payment Processing** - Financial transactions demand ACID guarantees. PostgreSQL is built for this; DynamoDB is not.

When DynamoDB Makes Sense:

If you had ultra-high-throughput requirements (like millions of real-time seat updates during a flash sale), DynamoDB could be valuable. But for most ticketing platforms, the consistency and query flexibility matter more than extreme scale.

Hybrid Approach (If Needed):

Some platforms use both: PostgreSQL for transactional data (orders, payments, inventory) and DynamoDB for high-velocity data (real-time seat availability cache, user activity logs). But for a modular monolith, start with PostgreSQL. It's simpler, safer, and handles ticketing workloads well.

My Recommendation:

Stick with PostgreSQL for a modular monolith. It covers all our needs for an event ticketing platform and keeps the architecture simpler. We can always add DynamoDB later for specific high-throughput use cases if they emerge.

ALB vs API Gateway

The key difference between an **Application Load Balancer (ALB)** and an **API Gateway** lies in their primary function and the layer of the application stack they manage.

- An **ALB** is a **traffic distribution and load balancing** tool. Its main job is to evenly distribute incoming HTTP/HTTPS traffic across multiple healthy targets (servers, containers, Lambdas).
- An **API Gateway** is an **API management and security** tool. Its main job is to act as the single entry point for all API calls, handling core API tasks like authentication, rate limiting, and request transformation *before* traffic reaches the backend services.

---

1. ⚖️ Core Functional Differences

Feature	**Application Load Balancer (ALB)**	**API Gateway**
Primary Goal	**Distribute** traffic evenly among a group of servers (load balancing).	**Manage, secure, and expose** APIs to clients (API governance).
Core Function	Load balancing, health checks, SSL termination, path-based routing.	Authentication/Authorization, Rate Limiting/Throttling, Request/Response Transformation, SDK Generation.
Pricing Model	**Per-hour fee** plus charges for Load Balancer Capacity Units (LCUs). You pay even when idle.	**Pay-per-request** and data transfer out. Serverless pricing model.
Timeout Limit	Very generous (up to **4,000 seconds**).	Quite restrictive (**29 seconds** default maximum for REST/HTTP APIs).

Target Integration	Integrates with **EC2, ECS, EKS, IP addresses, and Lambda**.	Integrates with **Lambda, HTTP Endpoints, and other AWS services** (like DynamoDB).
Protocol Support	Primarily **HTTP, HTTPS, and gRPC** (Layer 7).	**REST, HTTP, and WebSockets** APIs.

2. 🎯 When to Use One Over the Other

The choice depends on whether your main challenge is simple traffic distribution or complex API management.

Use the Application Load Balancer (ALB) When:

- **You have a traditional web application:** If you're running a monolithic application, standard web servers, or a simple microservices cluster where the front-end handles most logic.
- **You need high throughput and cost predictability:** ALBs are generally more cost-effective for large, sustained volumes of traffic that don't need complex API features.**You need long-running connections:** Critical when you have processes that require a long timeout (e.g., file uploads, complex processing jobs) due to its 4000-second limit.
- **You are already using it:** If your EC2 instances or containerized services are already sitting behind an ALB, it's simpler to extend it (e.g., adding a Lambda function as a new target path).

Use the API Gateway When:

- **You are building a Serverless Architecture:** It’s the native front-door for AWS Lambda functions, offering seamless, cost-effective integration.
- **You need rich API management features:** You require built-in features like **rate limiting**, **throttling**, **caching**, **API key management**, and native **IAM/Cognito authentication** before the request hits your code.
- **You need Request/Response Transformation:** You need to modify the request body, headers, or parameters to adapt incompatible clients and backend services.**You are exposing APIs to external developers:** It provides a professional layer for managing access, versioning, and documentation (via OpenAPI/Swagger).

3. 🤝 A Common Combination: ALB and API Gateway Together

In large, complex microservice architectures, they are often used in combination:

- **External Traffic:** An **ALB** sits at the edge, routing web traffic for the main application and distributing traffic to different versions or regions.
- **Internal Services:** The **API Gateway** sits *behind* the ALB (or sometimes directly at the edge) and acts as the front door for your internal microservices. It handles the API-specific logic (authentication, rate limiting) before sending the request to internal ALBs or individual Lambda functions.

For a deeper dive into the specific AWS services, check out the video below. Difference Between API Gateway and Load Balancer provides a visual explanation of the architectural differences between a load balancer and an API gateway.

Internal Event Bus - What It Is & How It Works

Simple Definition

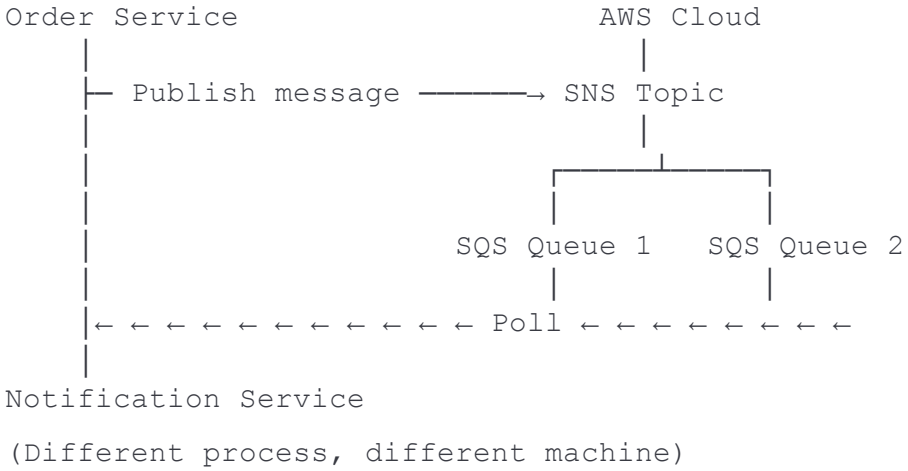
An internal event bus is a software component running inside your application that allows modules to communicate with each other through events, without sending messages over the network.

Think of it as a bulletin board inside your building:

- Module A posts a message on the bulletin board
- Module B and Module C read the message and react
- Everything happens inside the same building (same JVM)
- No need to send letters through the mail (network)

Internal Event Bus vs External Services

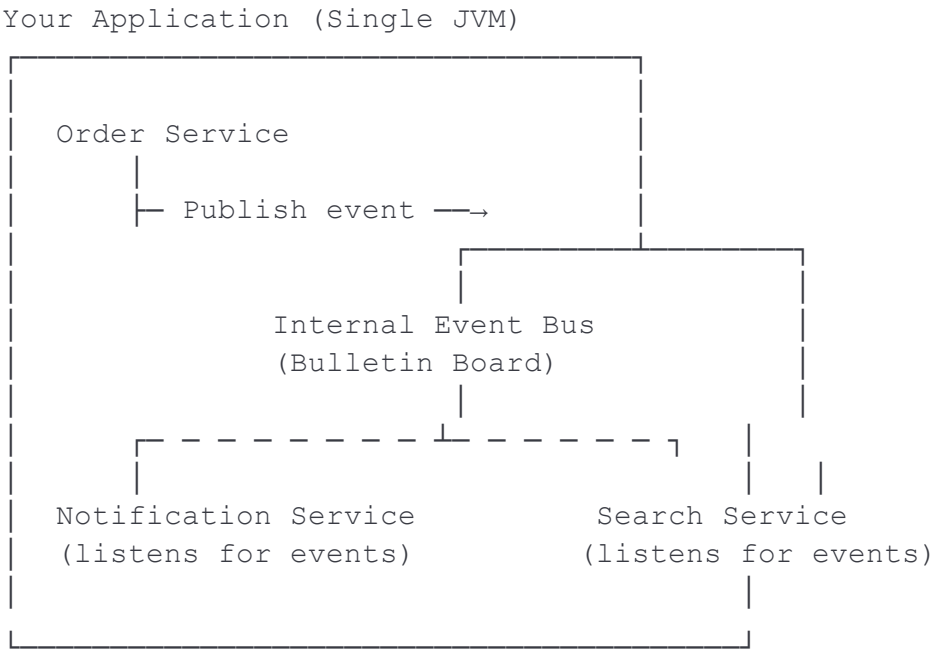
External Services (SNS/SQS) - Network-Based



Characteristics:

- Messages travel over network
- Services are completely separate processes
- Can run on different machines
- Network latency (milliseconds to seconds)
- Services can crash independently

Internal Event Bus - In-Process



Characteristics:

- Events stay inside your application
- Modules communicate via method calls
- Same JVM, same memory space
- Near-instant communication (microseconds)
- One crash affects everything

How It Actually Works (Code Level)

Step 1: Define an Event

java

```java
// An event is just a data object representing something that happened
public class OrderCreatedEvent extends DomainEvent {
    private String orderId;
    private String customerId;
    private LocalDateTime createdAt;

    public OrderCreatedEvent(String orderId, String customerId) {
        this.orderId = orderId;
        this.customerId = customerId;
        this.createdAt = LocalDateTime.now();
    }
```

```java
    // Getters
    public String getOrderId() { return orderId; }
    public String getCustomerId() { return customerId; }
}
```

Step 2: Publisher Posts Event to Bus

java

```java
// Order Service creates an order and publishes event
@Service
public class OrderService {
    @Autowired
    private EventPublisher eventPublisher;  // The internal bus

    @Autowired
    private OrderRepository orderRepository;

    public void createOrder(OrderRequest request) {
        // Create and save order
        Order order = new Order(request);
        orderRepository.save(order);

        // POST event to internal event bus
        OrderCreatedEvent event = new OrderCreatedEvent(
            order.getId(),
            order.getCustomerId()
        );
        eventPublisher.publish(event);
    }
}
```

Step 3: Subscribers Listen on Bus

java

```java
// Notification Service listens for OrderCreatedEvent
@Service
public class NotificationService {

    @EventListener  // This method is called when OrderCreatedEvent is
published
    public void onOrderCreated(OrderCreatedEvent event) {
        System.out.println("Got event from bus!");
        System.out.println("Order ID: " + event.getOrderId());
```

```
        // Send confirmation email
        sendConfirmationEmail(event.getCustomerId());
    }
}

// Search Service also listens for the same event
@Service
public class SearchService {

    @EventListener  // Also called when event is published
    public void onOrderCreated(OrderCreatedEvent event) {
        System.out.println("Got event from bus!");
        System.out.println("Order ID: " + event.getOrderId());

        // Index order for search
        indexOrder(event.getOrderId());
    }
}
```

Step 4: What Happens Behind the Scenes

```
Timeline of execution (all in same JVM):

1. orderService.createOrder() is called
   |
2. Order saved to database
   |
3. eventPublisher.publish(OrderCreatedEvent)
   |
4. Internal Event Bus receives event
   |
   ├─→ Calls: notificationService.onOrderCreated()
   |   └─→ Sends email
   |
   ├─→ Calls: searchService.onOrderCreated()
   |   └─→ Indexes order
   |
   ├─→ Calls: inventoryService.onOrderCreated()
   |   └─→ Updates stock
   |
5. All done, method returns to caller

Total time: milliseconds
All in same memory space
```

One transaction

The Internal Event Bus Implementation

Option 1: Spring's ApplicationEventPublisher (Built-in)

Most common for Java/Spring applications:

java

```java
// This is Spring's internal event bus
@Component
public class DomainEventPublisher {
    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    public void publish(DomainEvent event) {
        // This is the "bus" - distributes event to all listeners
        applicationEventPublisher.publishEvent(event);
    }
}
```

Spring does all the work for you:

- Finds all `@EventListener` methods
- Calls them when event is published
- Handles exceptions
- Optional async support

Option 2: Custom Event Bus (Advanced)

java

```java
// Manual implementation if you want more control
@Component
public class SimpleEventBus {
    private List<EventListener> listeners = new ArrayList<>();

    public void subscribe(EventListener listener) {
        listeners.add(listener);
    }

    public void publish(DomainEvent event) {
        // Call all listeners
        for (EventListener listener : listeners) {
            if (listener.canHandle(event)) {
                listener.handle(event);
```

```
            }
        }
    }
}

public interface EventListener {
    boolean canHandle(DomainEvent event);
    void handle(DomainEvent event);
}
```

But don't do this—use Spring! It's battle-tested and handles edge cases.

Comparison: SNS/SQS vs Internal Event Bus

Using SNS/SQS (Microservices)

java

```
// Order Service (Microservice 1)
public void createOrder(OrderRequest request) {
    Order order = new Order(request);
    orderRepository.save(order);

    // Send message over network to AWS
    snsClient.publish(
        new PublishRequest()

.withTopicArn("arn:aws:sns:us-east-1:123456:order-events")
            .withMessage(json(event))
    );
}

// Notification Service (Microservice 2, different machine)
public void handleOrderCreated(String message) {
    // Wait for message from SQS
    // Could take seconds
    // Network could fail
    // Message might not arrive
    sendEmail(extractCustomerId(message));
}
```

Problems:

- ❌ Networkcall adds latency (100ms+)
- ❌ Message might get lost

- ❌ Service might be down
- ❌ Complex error handling
- ❌ Eventual consistency (temporary data mismatch)
- ✅ Services independent (can restart separately)

Using Internal Event Bus (Modular Monolith)

java

```java
// Order Service (Module 1)
public void createOrder(OrderRequest request) {
    Order order = new Order(request);
    orderRepository.save(order);

    // Post to internal bulletin board
    eventBus.publish(new OrderCreatedEvent(order.getId()));
}

// Notification Service (Module 2, same application)
@EventListener
public void handleOrderCreated(OrderCreatedEvent event) {
    // Called immediately in same thread
    // Guaranteed to be called
    // Same transaction
    sendEmail(event.getCustomerId());
}
```

Benefits:

- ✅ Instant communication (microseconds)
- ✅ Guaranteed delivery
- ✅ Same transaction (all-or-nothing)
- ✅ Simple error handling
- ✅ Strong consistency
- ❌ Services crash together

Real-World Flow Example

Scenario: User clicks "Buy Concert Tickets"

Current Request (Using Internal Event Bus)

```
1. HTTP Request arrives at API
   POST /orders/create

2. OrderController receives it
```

```
3. OrderService.createOrder() starts
   └─ All these happen in ONE DATABASE TRANSACTION

4. Validate order

5. Call PaymentService directly (synchronous)
   └─ DebitCard() - if fails, everything rolls back

6. Save Order to database

7. Publish OrderCreatedEvent

8. Internal Event Bus immediately calls subscribers:

   ├─ NotificationService.onOrderCreated()
   │  └─ Send confirmation email
   │  └─ Still within same transaction
   │
   ├─ SearchService.onOrderCreated()
   │  └─ Index order for search
   │  └─ Still within same transaction
   │
   └─ InventoryService.onOrderCreated()
      └─ Decrement ticket count
      └─ Still within same transaction

9. Database transaction commits (everything succeeds or nothing does)

10. HTTP Response sent to user

Timeline: ~50 milliseconds total

Consistency: Strong (all changes committed or all rolled back)
```

If You Used SNS/SQS (Microservices)

```
1. HTTP Request arrives at API

2. OrderService.createOrder() starts

3. Save Order to database

4. Publish to SNS Topic
   └─ Returns immediately
```

```
5. HTTP Response sent to user

6. LATER... SNS pushes to SQS

7. LATER... Notification Worker polls SQS
   └─ Sends email (might take seconds)

8. LATER... Search Worker polls SQS
   └─ Indexes order (might take seconds)

9. LATER... Inventory Worker polls SQS
   └─ Updates count (might take seconds)

Timeline: User sees response in ~100ms, but email might arrive in 5 seconds

Consistency: Weak (data temporarily inconsistent between services)
```

When to Use Each

Situation	Use
Single deployment, monolith	Internal Event Bus
Services in different processes/machines	SNS/SQS
Need real-time broadcast	SNS
Need reliable queuing	SQS
Building microservices	SNS/SQS
Building modular monolith	Internal Event Bus

Key Takeaway

An internal event bus is just a mechanism to let modules in the same application talk to each other without network calls. It's like a method call, but more decoupled. Modules don't know about each other directly; they just publish and listen to events on a shared bus.
This keeps your monolith fast, consistent, and easy to debug, while SNS/SQS would make it distributed and complex.