Waitwize Technical White Paper

Giving Customers Their Time Back with a Scalable Virtual Queue & Notifications Platform

Architecture, Scalability, Multi-Tenancy, and Multi-Platform Delivery

Developed, implemented, and operated by WizardLabz LLC
Website: https://wizardlabz.com • Product: https://waitwize.com • Contact: info@wizardlabz.com

Document version: 1.0
First built: 2023
Distribution: Public PDF

Table of Contents

Table of Contents

Waitwize Technical White Paper 1

Giving Customers Their Time Back with a Scalable Virtual Queue & Notifications Platform 1

Architecture, Scalability, Multi-Tenancy, and Multi-Platform Delivery 1

Table of Contents 2

1. Executive Summary 4

2. The Problem: Waiting Is a Broken Customer Experience 4

3. What Waitwize Is and Where It Fits 5

4. Design Goals That Shaped the System 5

5. System Overview 5

6. Architecture Overview 6

7. Multi-Tenancy: One Platform Serving Many Businesses 6

8. Scalability: Designing for Bursts, Not Averages 7

9. Real-Time Operations: Why the Manager Dashboard Must Be Live 7

10. Notifications: The Feature That Customers Actually Feel 8

11. Reliability: Predictable Behavior Under Failure 8

12. Security and Tenant Isolation 9

13. Observability: Running the Platform Like a Product 9

14. Deployment Model: Repeatable, Scalable Delivery on AWS 9

15. Technology Stack 10

16. Technical Decisions and Rationale 10

Why Each Choice Supports Scalability and Multi-Platform 10

17. Roadmap: Where This Platform Can Go Next 11

18. Outcomes and Lessons Learned 11

19. About WizardLabz LLC 12

1. Executive Summary

Waitwize is a virtual queue and notifications platform created to solve a simple, costly problem: **customers hate waiting**—especially when waiting means standing in one place with no clear timing and no confidence that progress is happening.

WizardLabz began building Waitwize in 2023 as a SaaS product with an enterprise tier for branding and customization. The platform is designed for real operational environments, not ideal conditions—places like car service workflows, admissions offices, and "buzzer style" businesses where traffic arrives in bursts and staff must manage many moving parts in real time.

This white paper explains how Waitwize was engineered to be scalable and dependable as a multi-tenant platform, while offering a multi-platform customer experience. We focus on the "why" behind the design: why the platform needs real-time updates, why caching is necessary, why multi-tenancy is implemented the way it is, and how the stack choices connect directly to operational reliability.

2. The Problem: Waiting Is a Broken Customer Experience

The traditional waiting model is built around physical presence. Customers show up and then spend time "hovering" standing in a line, waiting near a counter, or sitting close enough to hear their name called. In buzzer systems, customers are still trapped inside the location. In many service workflows, waiting becomes a source of frustration, and staff become the human glue holding everything together.

The business impact is not subtle. Uncertainty makes customers impatient. Crowding makes the environment stressful. Interruptions make staff slower. Eventually, the line itself becomes the problem: people leave, or they stay and resent it. Either outcome harms trust.

Waitwize was built around a single concept: **customers should be free to reclaim their waiting time**, and businesses should be able to move customers through service without chaos.

3. What Waitwize Is and Where It Fits

Waitwize is not limited to one type of queue. It exists to support the real range of queue-heavy operations that businesses deal with every day, including:

- **Walk-in queues** where customers arrive unpredictably and expect quick progress
- **Pickup and "order ready" workflows** where the main friction is "when should I come back?"
- **Phone-based joining** for customers who do not want to install an app or use QR codes

The platform supports SaaS deployment at scale and includes an enterprise tier that allows branding customization. That matters because many organizations want the queue experience to feel like an extension of their brand, not a generic third-party tool.

4. Design Goals That Shaped the System

Waitwize's design goals came directly from the nature of queues:

First, queue demand is not smooth. It is spiky. A school admissions office can go from calm to overloaded within minutes. A car service location can receive a rush after a shift ends. If a system is designed for "average" traffic, it will fail during the exact moments it is most needed.

Second, operational tools cannot be slow. When staff are actively calling the next customer, marking an order ready, or dealing with a crowded reception area, they need the system to reflect reality immediately. A dashboard that is delayed or stale causes mistakes and confusion.

Third, adoption must be easy. Not every customer will install a mobile app. Many environments require phone-first joining, while still giving customers confidence about when to return.

Finally, the platform must scale as SaaS. It must support multiple independent businesses without building separate deployments for each one.

5. System Overview

At its core, Waitwize is composed of two experiences:

- The **customer experience**, centered on joining a queue and receiving notifications
- The **manager experience**, centered on controlling queue flow in real time

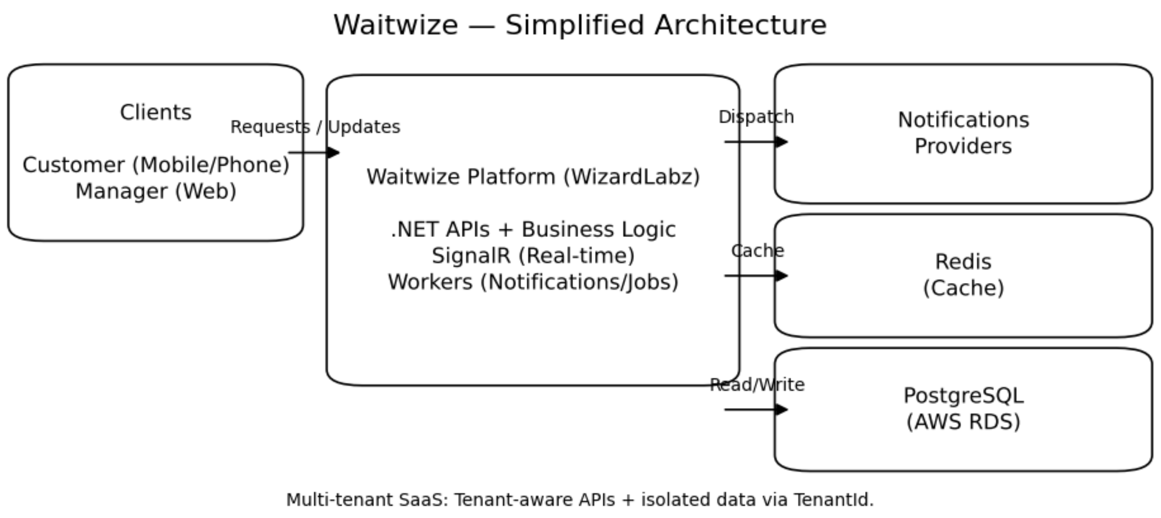
Behind these experiences is a tenant-aware backend that provides:

- APIs for mobile and web clients
- durable persistence for tickets, queue state, and configuration
- a notification pipeline that can send status updates asynchronously
- caching mechanisms that protect the database under load
- infrastructure capable of scaling horizontally as tenants grow

6. Architecture Overview

Waitwize uses a modern cloud-native structure where services are packaged in containers and deployed in a way that allows scaling by replication.

The backend is built on .NET Core with a mix of MVC and APIs. The durable source of truth is PostgreSQL, accessed through Entity Framework. Redis is used to reduce repeated reads and stabilize burst traffic. SignalR provides real-time updates to the manager dashboard so staff aren't forced into refresh loops or polling patterns. On the customer side, Flutter enables a consistent mobile experience across iOS and Android while keeping development velocity high.


Waitwize — Simplified Architecture
Clients
Customer (Mobile/Phone)
Manager (Web)
Requests / Updates
Waitwize Platform (WizardLabz)
.NET APIs + Business Logic
SignalR (Real-time)
Workers (Notifications/Jobs)
Dispatch
Notifications Providers
Cache
Redis (Cache)
Read/Write
PostgreSQL (AWS RDS)
Multi-tenant SaaS: Tenant-aware APIs + isolated data via TenantId.

The system runs in Docker containers deployed through AWS infrastructure. PostgreSQL is hosted on AWS RDS for operational stability, and services run on ECS to scale horizontally.

7. Multi-Tenancy: One Platform Serving Many Businesses

Waitwize is a multi-tenant SaaS system. That means the same platform can serve many businesses, with each tenant seeing only their own data and configuration.

WizardLabz chose a one-database, multi-tenant model because it offers the most practical foundation for SaaS growth. It reduces operational overhead, speeds onboarding, and keeps the platform easier to manage. The key is not the database count—it's the discipline of data isolation.

Tenant isolation is enforced by design. Tenant-owned tables are anchored by a `TenantId`, and every query is tenant-filtered. Indexing strategies are built around tenant-first access patterns so that even as the data grows, queries remain fast and predictable.

This model also preserves future flexibility. If a tenant becomes large enough to justify isolated infrastructure, the platform can evolve toward partitioning or dedicated databases for that tenant as part of an enterprise tier. The important decision is that the system is built with that option in mind rather than locking itself into complexity from day one.

---

8. Scalability: Designing for Bursts, Not Averages

Queue systems are defined by burst patterns. Designing for the average arrival rate is a common mistake. When the system is under stress, it's usually because many people join within a small time window, or staff are making rapid operational updates.

Waitwize is designed to absorb bursts without collapsing the database.

The first protection is architectural: the API layer is stateless and can scale horizontally. Adding capacity is operationally simple—ECS can run more service instances.

The second protection is behavioral: the platform avoids designs that force constant polling. Polling creates a steady baseline load even when nothing changes. In queue systems, polling tends to explode database traffic as more manager sessions and more customers are active.

Instead, Waitwize relies on event-oriented updates and caching. Redis reduces repeated lookups, and SignalR pushes changes to managers when they occur. This results in a platform where the system is active when there is activity, rather than constantly churning the database.

---

9. Real-Time Operations: Why the Manager Dashboard Must Be Live

For staff, the dashboard is the system. It is where the queue moves forward. If the dashboard is stale, the queue becomes chaotic.

SignalR is used because it eliminates the need for managers to refresh constantly. When a ticket moves, when a status changes, the dashboard updates immediately. This reduces operational mistakes and speeds service.

Scaling real-time systems must be done thoughtfully. The platform's real-time strategy focuses on keeping messages small and event-based. Instead of broadcasting the entire queue state repeatedly, the system emits "delta style" updates: what changed and why. This keeps message volume manageable and prevents update storms.

10. Notifications: The Feature That Customers Actually Feel

Customers don't care about architecture. They care about one thing: "Will I know when to come back?"

That is why notifications are central to Waitwize's customer promise. The system is designed so that the customer receives meaningful updates: confirmation that they joined, changes in status, and a clear signal when it's time to return or when an order is ready.

From an engineering perspective, notification systems must be asynchronous. If notifications are dispatched directly during a request, the customer experience becomes vulnerable to third-party latency and temporary failures.

Waitwize records the state transition durably and then dispatches notifications in the background. This approach makes the system faster, more reliable, and easier to retry safely.

11. Reliability: Predictable Behavior Under Failure

Systems at scale don't fail in clean ways. You get partial outages: a temporary Redis issue, a notification provider slowdown, a burst of traffic, a dropped real-time connection.

Waitwize is built to degrade gracefully. If real-time updates are interrupted, the manager dashboard can still function through standard API reads. If notification delivery is delayed, the queue state remains correct because the database is the source of truth.

Reliability is also about correctness. Queue systems must avoid double-updates and inconsistent state. Waitwize is designed around idempotent actions and durable state transitions, so repeating an operation does not corrupt data.

12. Security and Tenant Isolation

Multi-tenant platforms are only trustworthy when isolation is deliberate. Waitwize's security posture focuses on strict tenant scoping and role-based access. Tenant boundaries are enforced in the application layer and supported at the data layer through consistent tenant identifiers and query patterns.

Customer identifiers such as phone numbers require careful handling, both for privacy and compliance. The platform is designed to keep sensitive configuration and integration secrets out of code and managed through environment-based configuration.

13. Observability: Running the Platform Like a Product

Scalability is not only about handling load; it's about being able to see and respond to operational issues.

Waitwize is designed to be observable. The platform benefits from monitoring:

- join rates per tenant
- queue time trends
- notification success and failure rates
- API performance and error rates
- real-time connection volume
- database health indicators
- cache hit rate and effectiveness

Logs must be tenant-aware so that support and operational troubleshooting can be precise without searching blindly.

14. Deployment Model: Repeatable, Scalable Delivery on AWS

Waitwize runs in Docker containers to maintain environment parity and simplify deployment. AWS ECS is used to orchestrate and scale services by increasing task counts. PostgreSQL runs on AWS RDS to reduce operational burden and improve reliability through managed backups and monitoring.

This deployment model allows the platform to evolve and scale without re-architecting. It also supports the reality of SaaS operations: fast iteration, safe rollouts, and predictable rollbacks.

15. Technology Stack

Waitwize is built on:

- **.NET Core** for backend services and APIs
- **MVC + Web APIs** for operational dashboards and client contracts
- **Entity Framework** for persistence
- **PostgreSQL** for durable state
- **Redis** for caching and burst stabilization
- **SignalR** for real-time operational updates
- **Flutter** for iOS/Android mobile delivery
- **Docker** for packaging and environment parity
- **AWS RDS** for managed PostgreSQL
- **AWS ECS** for scalable service deployment

---

16. Technical Decisions and Rationale

Why Each Choice Supports Scalability and Multi-Platform

Waitwize's stack isn't a "bag of tools." Each choice was made to support the real constraints of queue-driven environments: burst load, real-time operations, multi-tenant SaaS, and multi-platform adoption.

.NET Core was chosen because it provides strong performance for API-heavy workloads and predictable behavior under scale. It supports async patterns cleanly, which is essential for notification flows and concurrent queue updates. It also runs seamlessly in containerized Linux environments, making deployment consistent.

MVC + APIs provide a stable backbone for both the manager dashboard and the mobile experience. The APIs allow future clients and integrations without changing core logic. MVC supports operational web views where structured workflows matter.

Entity Framework was selected because state transitions in queue systems must be correct, not just fast. EF enables transactional consistency and evolves well with multi-tenant patterns. Where needed, hot paths can be optimized, but the default posture remains maintainable and safe.

PostgreSQL was chosen as the durable store due to its reliability under concurrency and its strong indexing and query planning. Multi-tenant systems rely heavily on predictable query patterns, and PostgreSQL is well suited to those demands.

The **one database, multi-tenant** model was chosen because it is the most operationally efficient SaaS starting point. It reduces onboarding overhead and simplifies backups, monitoring, and environment management. The design is built so that enterprise isolation remains possible in the future for larger tenants.

Redis exists primarily to protect the database under burst traffic. The system avoids constant reads by caching frequently requested state and serving it quickly. Redis also supports patterns for scaling real-time systems across multiple service instances.

SignalR was selected because real-time operations eliminate polling, and polling is one of the fastest ways to destroy a database under load. SignalR enables event-driven updates that occur only when something changes.

Flutter was chosen because adoption depends on availability across iOS and Android without doubling development cost. A single codebase supports multi-platform delivery and allows enterprise branding to be applied consistently.

Docker exists to make deployments repeatable. Scaling requires predictability. Containers ensure that what runs locally matches what runs in production.

AWS RDS reduces operational risk by handling backups and managed operations for PostgreSQL. SaaS growth is hard enough; self-managed database failures are avoidable pain.

AWS ECS makes horizontal scaling straightforward and allows service roles to be separated. API services and background workers can scale independently.

This combination creates a platform that can scale by replication, remain operationally predictable, and deliver multi-platform experiences consistently.

---

17. Roadmap: Where This Platform Can Go Next

Waitwize's architecture supports growth into:

- smarter ETA estimation and forecasting
- deeper reporting and operational analytics
- integrations (webhooks and API-driven events)
- multi-location enterprise dashboards
- improved notification analytics and delivery optimization

---

18. Outcomes and Lessons Learned

Waitwize was built to make waiting less painful and operations calmer. The major lessons are:

- queues should be event-driven, not polling-driven
- multi-tenant SaaS requires discipline in isolation and indexing
- caching protects databases from burst patterns
- real-time operations must favor small updates over full state broadcasts
- multi-platform delivery is essential for adoption in real environments

---

19. About WizardLabz LLC

WizardLabz LLC builds reliable systems that businesses depend on. Waitwize is a WizardLabz product initiative started in 2022, delivered as SaaS with enterprise branding customization and consultancy to support rollout and adoption.

Website: https://wizardlabz.com
Product: https://waitwize.com
Contact: info@wizardlabz.com