Customer Support Ticket Prioritization System

1. Project Overview and Executive Summary

This project is a high-efficiency **web-based ticketing system** designed to modernize customer support management. It establishes a secure, two-sided platform that allows end-users to submit and track support issues, while utilizing a **Priority Queue (PQ)** data structure on the staff side to automatically sort and manage the workload. This mechanism ensures that high-priority, time-sensitive issues are always addressed first, directly enhancing service delivery and operational effectiveness.

Key System Features:

- **Dual Dashboards:** Separate user experiences for Customers (tracking their submissions) and Staff (managing the entire queue).
- **Priority-Driven Workflow:** All incoming tickets are sorted by a strict Priority Queue logic (Critical/High over Medium/Low, with FIFO within the same priority level).
- **Real-Time Synchronization:** All status updates and new submissions are reflected instantly across all relevant user interfaces.
- **Complete Ticket Lifecycle:** Supports the required transition states: **Open** → **In Progress** → **Closed/Resolved**.

2. Core Subject Area Analysis

A. Data Structures & Algorithms (DSA) Focus: Priority Queue

The Priority Queue (PQ) is the critical algorithmic component responsible for efficient staff workflow management.

Aspect	Details
Data Structure	Priority Queue, implemented in **JavaScript** for in-memory sorting on the Staff Dashboard.

Priority Order	**High (**3**) > Medium (**2**) > Low (**1**).** The priority values are mapped numerically for reliable comparison.
Tie-Breaker	**FIFO** principle: If two tickets share the same priority level, the ticket with the **older `created_at` timestamp** is prioritized.
Sorting Logic	Tickets are fetched from the database, and the entire set is processed by the PQ implementation to produce the single, optimally sorted queue displayed to staff.
Time Complexity	**Enqueue:** O(nlogn) (due to array insertion followed by sort); **Dequeue:** O(1) (retrieving the top element).

B. Node/Firebase Technology Stack

The project relies on a modern, serverless stack for robust, real-time data handling.

Component	Role in the System
Frontend	**HTML5, CSS3 (Tailwind CSS), and JavaScript.** Provides structure, responsive styling, and client-side logic for the Priority Queue implementation and Firebase integration.
Backend/Database	**Firebase Firestore (Cloud NoSQL Database).** Used for persistent storage of all tickets and user data, offering real-time synchronization capabilities.
Authentication	**Firebase Authentication** manages secure user (customer) sign-up/login and role-based access control for staff users.

Real-Time Sync	Firestore's `onSnapshot()` listeners are used to ensure that the User and Staff Dashboards automatically and instantly reflect any updates (status changes, new submissions) without manual refresh.

C. Business Perspective: Operational Efficiency

The system is engineered to solve key operational challenges in customer service environments.

- **Critical Task Guarantee:** The Priority Queue sorting logic directly ensures that the highest-impact issues (e.g., "Critical" severity) are pushed to the top of the staff queue, minimizing downtime and mitigating business risk.
- **Staff Productivity Boost:** Staff members are presented with a pre-sorted, actionable list, removing the manual effort required for prioritization and allowing them to immediately focus on the most urgent ticket.
- **Customer Transparency:** The dedicated User Dashboard provides customers with self-service tracking, dramatically reducing the volume of inbound status-check calls or emails, thereby freeing up staff time for resolution work.

3. System Architecture and Functional Modules

3.1 Ticket Submission Workflow (Customer)

Field	Description / Action
Inputs	Name, Email, Issue Description, Priority (High/Medium/Low).
Action	Ticket data is sent to **Firebase Firestore**.
Auto-Fields	Status set to **"Open"**. Auto-generated unique ID and `created_at` timestamp.

3.2 Staff Dashboard Management Workflow

1. **Data Fetch:** Staff dashboard fetches **all** tickets from the dedicated Firestore collection (`tickets`).
2. **Prioritization:** The fetched data is immediately passed into the **Priority Queue** implementation.
3. **Display:** The dashboard renders the queue in its strict, prioritized order.
4. **Staff Action:** Staff update the ticket's status (**Open** → **In Progress** → **Closed**) or delete the ticket entirely.
5. **Synchronization:** The Staff action is written back to Firestore, triggering an instant, real-time update on the Customer Dashboard.

3.3 Database Design (Firestore)

Collection	Document Field	Data Type	Purpose
`tickets`	`name`, `email`	String	Customer contact information.
	`issue`	String	Detailed description of the problem.
	`priority`	String (High/Med/Low)	Priority level, used for PQ sorting.
	`status`	String (Open/In Progress/Closed)	Current state of the ticket lifecycle.
	`created_at`	Timestamp	Crucial for FIFO sorting logic.
	`unique ID`	Auto-Generated String	Primary key for ticket identification.

4. Deployment and Security Considerations

Deployment

Since the project utilizes Firebase and client-side JavaScript, deployment is streamlined via static hosting services such as **GitHub Pages, Netlify, Firebase Hosting, or Vercel**.

Security & Best Practices

Security measures are focused on data isolation and input validation:

- **Access Control:** Access to the Staff Dashboard must be restricted via **Firebase Auth** and appropriate **Firestore Security Rules**.
- **Data Filtration:** Firestore rules are essential to enforce that a Customer can **only** view tickets associated with their specific ID/email, while Staff can view all tickets.
- **Data Integrity:** Storing `created_at` as a secure Firestore Timestamp ensures consistent, non-mutable data for accurate FIFO ordering.