Firebase and NoSQL Database Architecture Report: Priority-Driven Ticketing System

1. Technology Stack Justification

Parameter	Technology	Justification
Database	**Firebase Firestore (NoSQL)**	Chosen for its **real-time synchronization capabilities** and **scalable, flexible data model**. This is ideal for a live ticketing system where instantaneous updates between users and staff are critical.
Authentication	**Firebase Authentication**	Provides secure, industry-standard authentication mechanisms and simplified user management for both customer and staff roles.
Client-Side Logic	**JavaScript (ES Modules)**	Used to directly interface with the Firebase SDKs, enabling front-end CRUD (Create, Read, Update, Delete) operations and real-time listeners without the need for a separate backend server (serverless architecture).

2. Implementation Accuracy and Real-Time Capabilities

Real-Time Synchronization (`onSnapshot`)

The system leverages Firestore's `onSnapshot()` listeners to establish a live connection between the database and the client dashboards.

- **Staff Dashboard:** Listens to the entire `tickets` collection. Any new ticket submission or status update instantly triggers a re-fetch and re-sort of the Priority Queue on the staff side.

- **Customer Dashboard:** Listens to a filtered query (tickets belonging only to the logged-in user). When a staff member updates a ticket status, the customer's dashboard reflects the change within milliseconds.

Core CRUD Operations

Action	Function	Location	Purpose
Create (Submission)	`addDoc(collection(db, 'tickets'), data)`	Customer Dashboard	Saves a new support request with default status "Open" and auto-generated `created_at` timestamp.
Read (Tracking)	`query(ticketsRef, where('customerId', '==', user.uid))`	Customer Dashboard	Filters data based on the authenticated user's ID to ensure they only see their own tickets.
Update (Management)	`updateDoc(doc(db, 'tickets', ticketId), { status: newStatus })`	Staff Dashboard	Used by staff to change the ticket lifecycle state (e.g., to "In Progress" or "Resolved").
Delete (Management)	`deleteDoc(doc(db, 'tickets', ticketId))`	Staff Dashboard	Used by staff to permanently remove a ticket document from the database.

3. Data Modeling (Firestore - NoSQL)

Firestore uses a flexible document model within collections. This design is optimized for fast, real-time read and write operations required by the ticketing system.

Collection: `tickets`

Field	Data Type	Purpose in the System
`name`, `email`	String	Customer contact details.
`issue`	String	The detailed support request description.
`priority`	String	The ticket urgency (Critical, High, Medium, Low)—crucial for Priority Queue sorting.
`status`	String	The current stage in the ticket lifecycle (Open, In Progress, Resolved).
`customerId`	String	**The unique ID (UID) used to filter tickets for the Customer Dashboard.**
`created_at`	Timestamp	**Firestore Timestamp**—essential for the FIFO tie-breaker rule in the Priority Queue.

Collection: `users` (For Auth/Roles)

This supplementary collection stores role information necessary to grant Staff access.

- **Fields:** `displayName`, `email`, `role` (e.g., "customer" or "staff").
- **Access Control:** Used in `staff_login.js` to verify the user's role after successful login before redirecting to the Staff Dashboard.

4. Security, Modularity, and Scalability

Security and Access Control

While Firestore rules are outside the scope of this client-side code, the architectural intent includes:

- **Role-Based Access:** Ensuring that only authenticated users with the `role: "staff"` can perform `update` and `delete operations` on the `tickets` collection.
- **Data Isolation:** Enforcing rules that ensure a customer can only `read` documents where the `customerId` field matches their authenticated UID (`request.auth.uid`).

Code Quality and Modularity (JavaScript)

- **Centralized Firebase Config:** All Firebase initializations and import statements are centralized in a single `firebase.js module`, adhering to best practices for code reusability.
- **Modular Imports:** Other application files (`dashboard_user.html`, `staff_dashboard.html`, etc.) use JavaScript ES Modules (`import/export`) to selectively load only the necessary Firebase functions (e.g., `getAuth`, `onSnapshot`, `addDoc`), promoting clean, modular code.

Scalability

Firestore's architecture provides built-in horizontal scaling. The use of unique document IDs and a well-defined `ticket collection` ensures that the system can handle a large volume of concurrent tickets without impacting performance, as NoSQL databases excel at high-speed reading and writing of individual documents.