OpenGlob POS Engine v2: Universal Offline-First Synchronization Engine

OpenGlob Architecture Board

February 7, 2026

Abstract

This document defines the *OpenGlob POS Engine v2* (Persistent Offline Synchronization) protocol as the mandatory technical standard for all future OpenGlob web applications, including data-heavy, stateful systems that exceed simple task-list complexity. The protocol formalizes an architectural shift from synchronous request–response workflows to an *Asynchronous Persist–Sync* model: all user actions are first committed to a local, durable queue and applied to local state immediately; network synchronization occurs in a background drain process with strict idempotency guarantees.

The primary objectives are (i) *zero UI blocking* (0ms perceived latency under typical rendering constraints), (ii) *universal action schema* across all OpenGlob products, and (iii) *duplicate-free, retry-safe synchronization* under network faults and backend timeouts. This standard explicitly corrects prior architectural flaws in universal state management and idempotency.

1 Scope and Non-Negotiable Requirements

Scope. This standard applies to all OpenGlob web applications that mutate user-visible state (Create/Update/Delete) and synchronize with any remote backend (Flask, Google Apps Script, or equivalent).

Hard requirements.

- Every user action that mutates state *MUST* (a) update local application state and repaint the UI immediately, and (b) be serialized into a durable local persistence queue.
- All sync requests *MUST* be idempotent. Retries *MUST NOT* create duplicates.
- The client-generated identifier (`tempId`) *MUST* be transmitted to the server and honored as the canonical identifier for the event (and, where applicable, the resource primary key).
- The protocol *MUST* be universal: the queue schema cannot be task-specific.

2 Abstract Core Philosophy: From Synchronous Request–Response to Asynchronous Persist–Sync

2.1 Problem Statement

Traditional web applications couple user interaction to server execution: the UI blocks (spinner/disabled controls) while the network round-trip and server-side write completes. This coupling is fragile under high latency and catastrophic under burst concurrency.

In Google Apps Script (GAS) backends, burst traffic increases simultaneous executions and increases time-to-response. Operationally, this pattern collapses around ˜30 simultaneous writers when requests arrive at the same moment and executions elongate, exhausting concurrency limits.

2.2 Solution Statement

POS Engine v2 decouples *User Interaction* from *Server Execution*. Every mutation is (1) applied optimistically to local state and rendered immediately, and (2) written to a durable local FIFO queue (`localStorage`) that is drained by a background worker.

2.3 Key Metrics (Operational Targets)

- **0ms perceived latency** independent of network speed: user-visible updates occur on the local event loop and paint cycle (typical first paint ˜16ms at 60Hz).
- **10x backend concurrency capacity**: shifting from direct synchronous writes (˜30 concurrent users) to queued background sync (˜300+ sustained users) by smoothing burst traffic into sequential/background drains.

3 Architecture: Dual-Branch Execution Model (Mandatory)

Every Create/Update/Delete action triggers two branches in the same user event.

Branch A (Optimistic UI). Mutate local state immediately and re-render the DOM.

Branch B (Persistence Queue). Serialize the action into a durable FIFO queue in `local Storage`using the universal schema defined in Section 4.1.

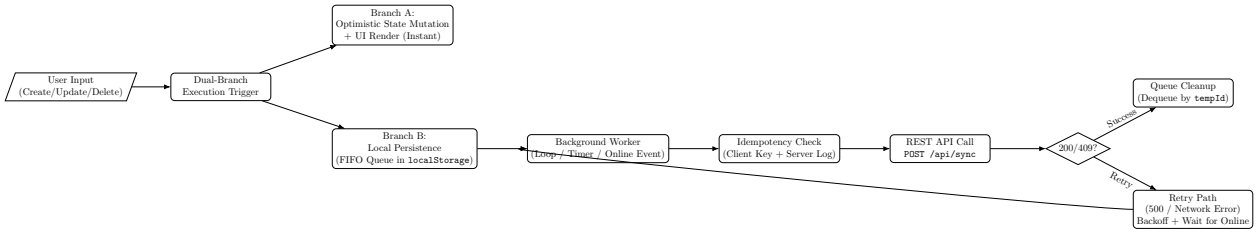

Figure 1: POS Engine v2 Dual-Branch Execution Model with strict idempotency and retry-safe queue drain.

3.1 Protocol Flowchart (TikZ)

4 Technical Specifications (Corrected Logic)

4.1 Universal Queue Schema

The queue item is an *event envelope* that must support *any* OpenGlob product (tasks, inventory, users, orders, etc.). It must be stable across versions and suitable for backend replay.

Listing 1: Universal POS queue schema (event envelope).

```
{
  "tempId": "uuid_v4_client_generated",
  "action": "CREATE" | "UPDATE_STATUS" | "SOFT_DELETE",
  "resource": "Task" | "InventoryItem" | "User",
  "payload": { "...": "..." },
  "timestamp": 1707300000000
}
```

Normative semantics.

- `tempId` is a client-generated UUIDv4 and is the idempotency key for the event. It is stable across retries and across page reloads.
- `timestamp` is client epoch-millis at time of enqueue.
- `resource` and `action` define the meaning of `payload`. The protocol is agnostic to domain-specific fields.

4.2 Corrected Sync Engine: `processQueue` (Strict Idempotency)

Crucial correction: the client *MUST* send `tempId` to the server and the server *MUST* use it for idempotency. A retry is not a new write.

Listing 2: Corrected POS v2 queue drain with strict idempotency.

```
async function processQueue() {
  const item = TaskQueue.peek();
```

```
  if (!item) return;

  try {
    // Idempotency: the server uses 'tempId' as the permanent primary key
    // or checks it against a processed-events log.
    const res = await fetch('/api/sync', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'X-OpenGlob-Idempotency-Key': item.tempId
      },
      body: JSON.stringify(item)
    });

    // 409 Conflict means "Already Processed" -> Treat as Success
    if (res.ok || res.status === 409) {
      TaskQueue.remove(item.tempId);
      processQueue(); // Recursive drain
    } else if (res.status >= 500) {
      setTimeout(processQueue, 5000); // Server busy, backoff
    }
  } catch (networkError) {
    // Wait for 'online' event
  }
}
```

4.3 Correcting the Historical Duplicate Bug (Root Cause)

Prior architecture produced duplicates under timeouts due to a two-part failure mode.

Flaw A: Amnesiac backend. A retry was treated as a brand new request. A typical failure pattern is server-side UUID generation on each request (e.g., `uuid.uuid4()`) which guarantees duplicates on retry.

Flaw B: Blind persistence layer. A write target (e.g., Google Sheets `appendRow`) accepted every request without checking for prior processing.

Ghost-write scenario. If the backend write completes but the response times out (e.g., GAS takes 11s, caller times out at 10s), the client retries; the backend generates a new ID and persists again. POS Engine v2 eliminates this class of defects by requiring that the client-supplied `tempId` is stable and is checked server-side before any write.

Metric	Current App (Synchronous)	POS v2 (Persist–Sync)
Max concurrent users (burst writes)	˜30 users	˜300+ users
Daily active users (DAU) capacity	˜500 DAU	˜5,000+ DAU
Primary bottleneck	Simultaneous executions; the 31st burst request fails	Daily quotas (e.g., total fetches/day); bursts are smoothed
Duplicate risk under retry	High (new IDs on retry)	Eliminated (idempotency key enforced)

Table 1: Operational comparison: synchronous coupling vs. POS Engine v2 queue smoothing.

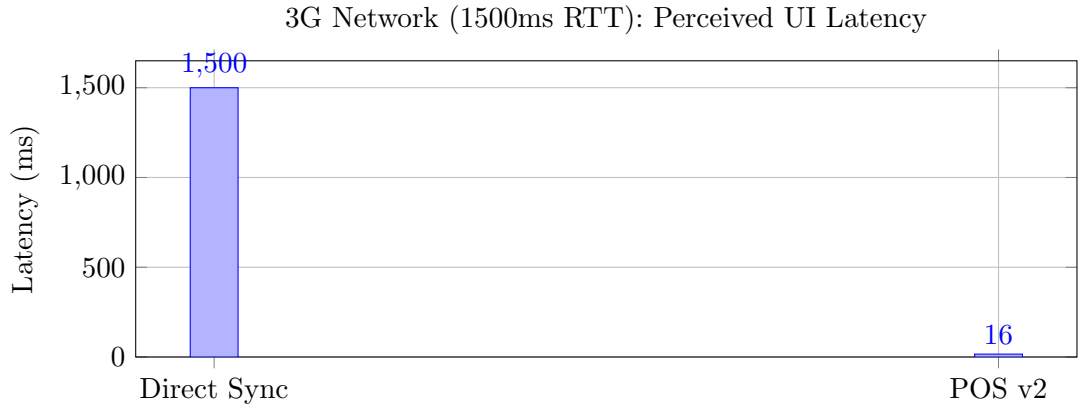

Figure 2: Direct sync blocks UI on network RTT; POS v2 commits locally and paints immediately (˜16ms frame).

5 Performance Comparison (Hard Data, `pgfplots`)

5.1 Capacity and Reliability Summary Table

5.2 Chart 1: User Perceived Latency (Lower is Better)

5.3 Chart 2: Backend Concurrency Capacity (Higher is Better)

5.4 Chart 3: Data Reliability (Higher is Better)

6 Integration Contract (Backend Requirements)

This section is normative. Backends that do not meet these requirements are *non-compliant* with OpenGlob POS Engine v2.

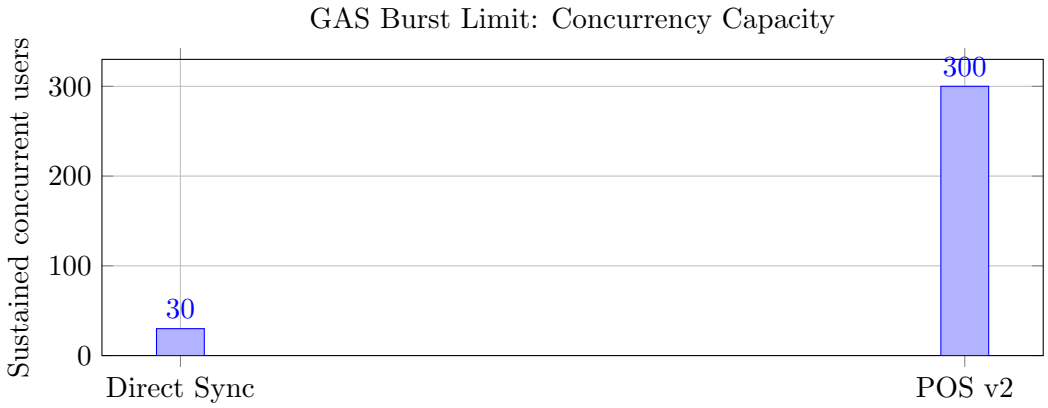

Figure 3: Queue smoothing reduces peak simultaneous executions by shifting writes to background drains.

Figure 4: POS v2 persists events locally before network attempts; actions survive tab closure and sync on next visit.

6.1 Canonical Client ID Rule

The server MUST NOT generate a new UUID for an event that already has a `tempId`. The server must accept the client-provided `tempId` as the canonical identifier.

6.2 Idempotency Rule

Before writing any mutation, the backend MUST perform an idempotency check:

- If `tempId` has not been processed: apply the mutation and mark `tempId` as processed.
- If `tempId` has already been processed: return **200 OK** or **409 Conflict**. Both are interpreted by the client as *success* and trigger dequeue.

6.3 Minimal Backend Interface

The POS v2 client requires a single endpoint:

- `POST /api/sync`: accepts the universal event envelope and returns one of:
  - `200 OK`: processed successfully,
  - `409 Conflict`: already processed (idempotent replay),
  - `5xx`: retry with backoff.

7 Standardized State Management (Universal, Application-Agnostic)

7.1 Why "Universal State" Is Required

A queue without a universal state model results in inconsistent UI, divergent caches, and untraceable edge cases. POS Engine v2 requires that every UI mutation is derived from the same action envelope that is persisted and later synced.

7.2 Normative Rule: Actions Are the Source of Truth

- UI state transitions MUST be applied by reducing an action into local state (i.e., a deterministic state transition function).
- The same action envelope MUST be the unit of persistence and the unit of synchronization.

7.3 Recommended Determinism Constraints

- Action handlers should be pure with respect to state (no hidden network reads).
- Any nondeterminism (e.g., timestamps) must be captured in the action envelope at enqueue time.

8 Operational Notes and Failure Modes

8.1 Backoff and Online Recovery

When network calls fail, the client must wait for the browser `online` event or a scheduled backoff before re-attempting `processQueue`. Repeated failures must not block UI.

8.2 Crash/Reload Safety

The queue MUST be loaded from `localStorage`on application startup and drained automatically. This ensures that actions taken immediately before a crash or tab close are eventually synchronized.

9 Compliance Checklist

An OpenGlob application is POS v2-compliant only if all of the following are true:

1. Every user action writes an action envelope to a durable FIFO queue before any network attempt.
2. Every user action updates local state and re-renders UI without awaiting the server.
3. Every sync request includes `tempId` as an idempotency key.
4. The backend enforces idempotency based on `tempId` and returns `200` or `409` for replays.