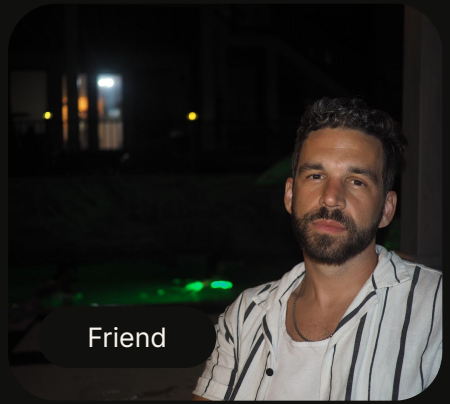
# Redis

## The Key to Delivering Value

*Unlocking Advanced Patterns for Scalable Healthcare*

Brandon Stiles
Talkiatry
07-11-2025

About Me






# What is Redis?

## The 30-Second Primer

*What it is:*

# REmote DIctionary Server.

*An open-source, in-memory data store.*

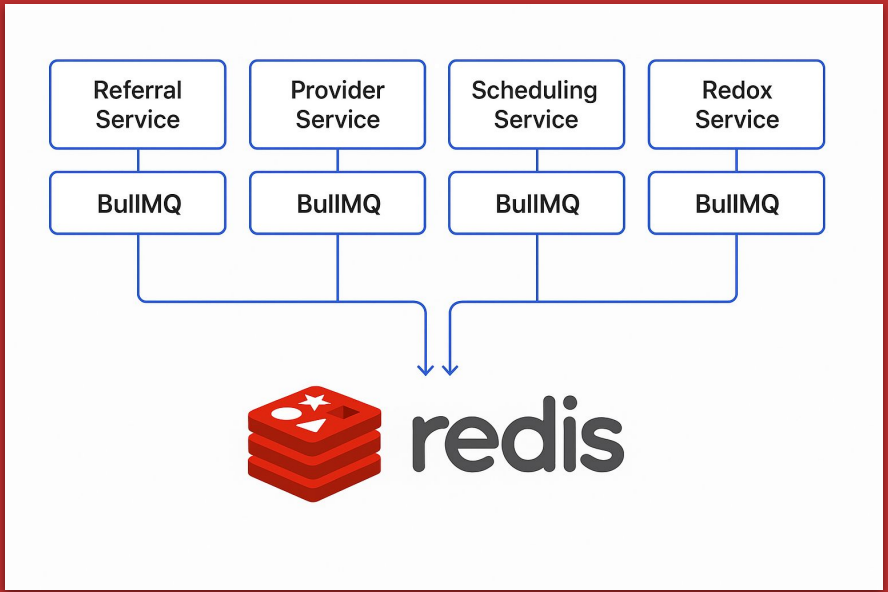
## The Key Idea: It's a Data Structure Server, not just a Key-Value Store

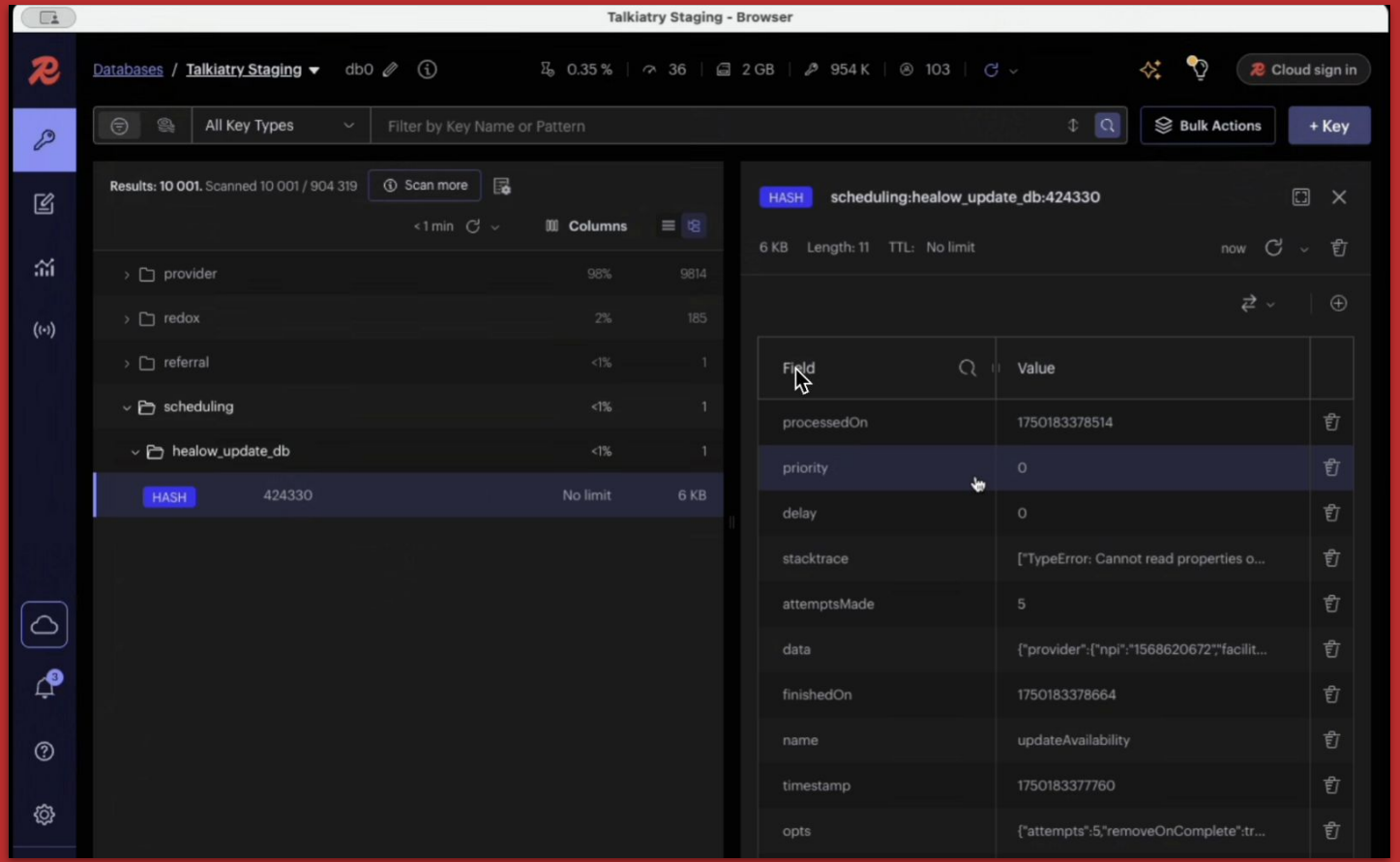
Caching

Session Store

Job Queue

# Talkiatry's Current Redis Architecture


Referral Service
Provider Service
Scheduling Service
Redox Service
BullMQ
BullMQ
BullMQ
BullMQ
redis


Talkiatry Staging - Browser
Databases / Talkiatry Staging
db0
0.35 %
36
2 GB
954 K
103
Cloud sign in
All Key Types
Filter by Key Name or Pattern
Bulk Actions
+ Key
Results: 10 001. Scanned 10 001 / 904 319
Scan more
< 1 min
Columns
provider 98% 9814
redox 2% 185
referral <1% 1
scheduling <1% 1
healow_update_db <1% 1
HASH 424330 No limit 6 KB
HASH scheduling:healow_update_db:424330
6 KB Length: 11 TTL: No limit
now
Field
Value
processedOn 1750183378514
priority 0
delay 0
stacktrace ["TypeError: Cannot read properties o...
attemptsMade 5
data {"provider":{"npi":"1568620672","facilit...
finishedOn 1750183378664
name updateAvailability
timestamp 1750183377760
opts {"attempts":5,"removeOnComplete":tr...

# Use Case:
## Caching Provider Availability in the Scheduling Service

**The Problem:**
- Repetitive, identical API calls for provider availability due to non-debounced clicks in the front-end (TalkApp).

**The Solution:**
- Implemented a simple read-through cache using Redis.
- The first request is processed, and its response is cached with a 60-second TTL (Time-To-Live).
- Subsequent identical requests are served instantly from the cache.

**The Impact:**
- A huge reduction in redundant processing.
- Result: 58% of all availability requests are now served directly from the cache.

# A Few Primers / Reminders Before Diving In...

Caching Strategies

# How we write data

**Read-Through**
App reads from cache. On miss, cache fetches from DB. (Simplifies app logic).

**Write-Through:**
App writes to cache and DB simultaneously. (Ensures consistency).

**Write-Back (Write-Behind):**
App writes to cache, cache writes to DB later. (Fastest writes).

**Write-Around:**
App writes directly to DB, bypassing the cache. (Avoids caching cold data).

Eviction Policies

# What data do we remove?

**Least Recently Used:**
Removes the key not accessed for the longest time. ("Use it or lose it").

**Least Frequently Used:**
Removes keys that aren’t often accessed

**TTL :**
Removes keys closest to their expiration time.

**noeviction (Default):**
Returns an error on writes when memory is full. (This is the policy that can cause cascading failures if not managed).

# Creative & Advanced Redis Patterns

# An Evolution in Auth

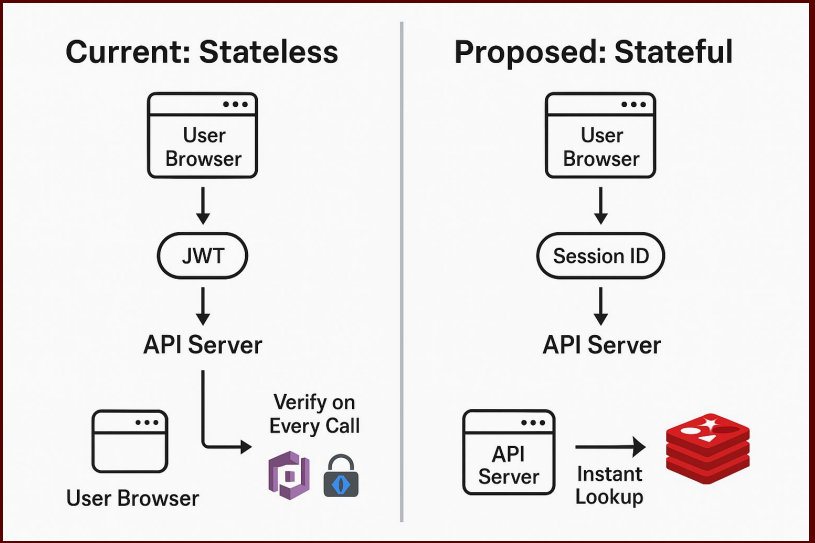
## From Stateless to Stateful Sessions

**Current (Stateless):**
Uses standard JWTs, requiring cryptographic verification on every API call.

**Proposed (Stateful):**
Use Redis to store sessions. After login, the client gets a simple session ID, turning future authentication into a fast Redis lookup.

**Key Advantages:**

- **Performance:** Swaps slow crypto verification for a microsecond Redis lookup.
- **Security:** Allows for instant session revocation (e.g., "log out everywhere"), which is not possible with JWTs.



# Holding Appointments

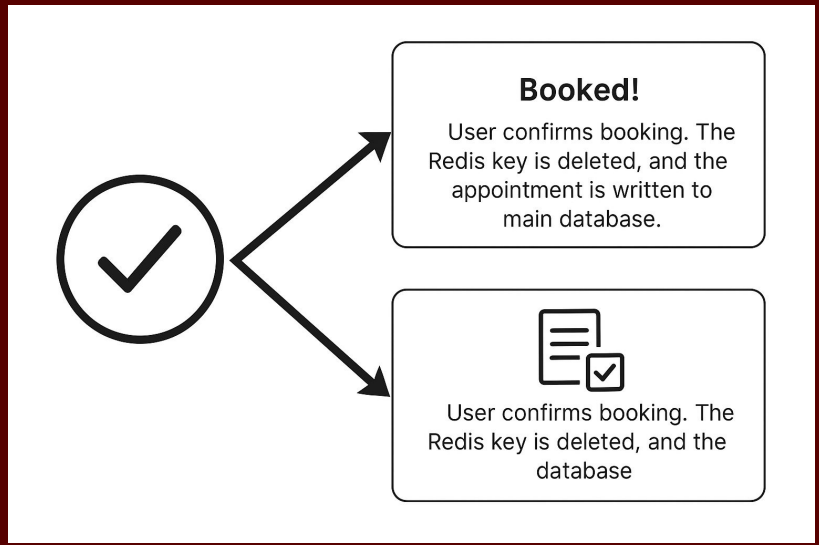
## The Ticketmaster Model

**The Challenge:**
Our current method of holding appointment slots in the main database is slow and requires complex cleanup for abandoned holds.

**The Redis Solution:**
Use SETNX to atomically create a key for the slot with a 5-minute EXPIRE time, leveraging Redis's built-in TTL feature.

**The Benefits:**

- *Prevents Double-Booking*: The atomic operation ensures only one user can hold a slot.
- *Reduces Database Load:* Moves frequent, temporary writes out of the main database.
- *Self-Cleaning:* Redis automatically releases abandoned holds, eliminating the need for cleanup jobs.



# The Smart Caching Wrapper

**The Challenge:**
Implementing caching for frequently-read but rarely-changed data is repetitive work for developers.

**The Proposed Solution:** A lightweight database model wrapper that enables read-through caching automatically with a single line of code (e.g., `useCache: true`).

**High-Impact Use Cases:**

- **Provider Data & Medallion Models:** Caching this rarely updated data would greatly reduce database load.
- **Insurance Mapping Table:** Caching its 37,000+ records would fix a significant server performance bottleneck.

```
// Before: Standard data fetch
// Every call queries the database directly.

const providers = await ProviderModel.findAll();
```

```
// After: With the Caching Wrapper

// 1. A developer opts-in by extending the base model
//    and adding a single property.
class ProviderModel extends CachingBaseModel {
  static useCache = true;
}

// 2. The data fetch code remains identical, but now
it automatically checks Redis first!
const providers = await ProviderModel.findAll();
```

# Real-Time Geolocation

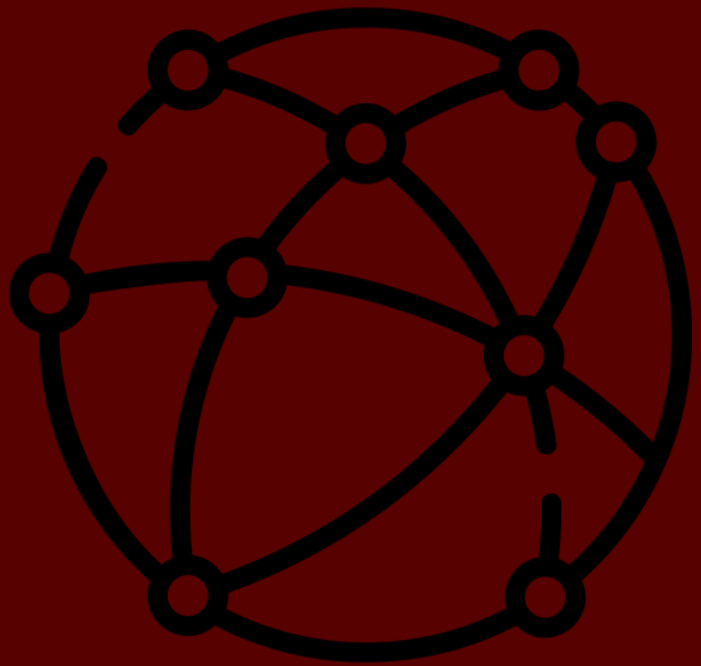
## "Find a Pharmacy Near Me"

**The Challenge:**
Finding nearby pharmacies with traditional database queries is slow and puts a heavy load on our primary database, especially with many concurrent users.

**The Redis Solution:**
Leverage Redis's built-in Geospatial data structure, which is optimized for this exact purpose.

**The Benefits:**

- **Extremely Fast:** Turns a complex database query into a simple, microsecond operation.
- **Reduces Database Load:** Offloads all geospatial calculations from our primary database.
- **Versatile:** Can be used for pharmacies, labs, provider offices, and more.



```
// --- 1. The Background Job (Runs Nightly) ---
/**
 * Fetches all pharmacies from the primary database and populates
 * the Redis geospatial cache.
 */
async function populatePharmacyCache() {
  const pharmacies = await mockDbClient.getAllPharmacies();

  // Format the data for the GEOADD command: [lon, lat, member, lon, lat, member, ...]
  const locations = pharmacies.flatMap(p => [p.lon, p.lat, p.name]);

  if (locations.length > 0) {
    await mockRedisClient.geoadd('pharmacies:locations', locations);
  }
  console.log('Nightly job complete.');
}

// A mock Redis client for demonstration.
const mockRedisClient = {
  // In a real client, this would be a single network call.
  geoadd: async (key, locations) => {
    console.log(`[Redis] Added ${locations.length / 3} locations to '${key}'.`);
    return locations.length / 3;
  },
  georadius: async (key, lon, lat, radius, unit, withdist) => {
    console.log(`[Redis] Searching '${key}' within ${radius}${unit} of (${lon},
${lat})`); // would return real data from the Redis server.
    return [['CVS - Downtown Atlanta', '0.15'], ['Walgreens - Midtown', '3.5']];
  },
};


// --- 2. The User-Facing Function (Runs on API Request) ---
/**
 * Finds nearby pharmacies using the Redis cache.
 * @param {object} userCoordinates - The user's { lon, lat }.
 * @returns {Promise<Array>} A list of nearby pharmacies.
 */
async function findNearbyPharmacies(userCoordinates) {
  const { lon, lat } = userCoordinates;
  const searchRadiusKm = 5;

  const results = await mockRedisClient.georadius(
    'pharmacies:locations',
    lon,
    lat,
    searchRadiusKm,
    'km',
    'WITHDIST'
  );

  // Format the raw Redis response into a clean object array.
  return results.map(([name, distance]) => ({
    name,
    distance: `${parseFloat(distance).toFixed(2)} km`,
  }));
}
```

# Distributed Locks for Critical Operations

**The Challenge:**
Finding nearby pharmacies with traditional database queries is slow and puts a heavy load on our primary database, especially with many concurrent users.

**The Redis Solution:**
Leverage Redis's built-in Geospatial data structure, which is optimized for this exact purpose.

**The Benefits:**

- **Extremely Fast:** Turns a complex database query into a simple, microsecond operation.
- **Reduces Database Load:** Offloads all geospatial calculations from our primary database.
- **Versatile:** Can be used for pharmacies, labs, provider offices, and more.

```
/**
 * A mock Redis client for demonstration. In a real app,
 * this would be an instance of a library like 'ioredis'.
 */
const redisClient = {
  // A map to simulate Redis keys
  _keys: new Map(),
  set: async function(key, value, options) {
    if (options?.NX && this._keys.has(key)) {
      return null; // 'NX' fails if key exists
    }
    this._keys.set(key, value);
    if (options?.PX) {
      setTimeout(() => this._keys.delete(key), options.PX);
    }
    return 'OK'; // 'OK' indicates success
  },
  del: async function(key) {
    this._keys.delete(key);
    return 1;
  }
};

/**
 * Simulates sending a refill request to an e-prescribing service.
 */
async function sendPrescriptionToEcw(patientMrn, medicationId) {
  console.log(`Sending refill for ${medicationId} for patient ${patientMrn}...`);
  // Simulate network delay
  await new Promise(resolve => setTimeout(resolve, 1000));
  console.log('Prescription sent successfully.');
}


/**
 * Handles a patient's refill request using a distributed lock
 * to prevent duplicate submissions.
 */
async function requestRefillWithLock(patientMrn, medicationId) {
  const resourceKey = `refill:${patientMrn}:${medicationId}`;
  const lockKey = `lock:${resourceKey}`;
  const lockValue = 'in-progress'; // A simple value for the lock
  const lockTtlMs = 30000; // 30-second lock timeout

  // Try to acquire the lock atomically.
  const lockAcquired = await redisClient.set(lockKey, lockValue, {
    NX: true, // Set only if the key does not exist
    PX: lockTtlMs, // Set expiration in milliseconds
  });

  if (!lockAcquired) {
    console.log(`Request for ${resourceKey} is already in progress. Ignoring
duplicate.`);
    return;
  }

  // If we acquired the lock, perform the critical action.
  try {
    await sendPrescriptionToEcw(patientMrn, medicationId);
  } finally {
    // ALWAYS release the lock when done.
    await redisClient.del(lockKey);
    console.log(`Lock for ${resourceKey} released.`);
  }
}

// --- Simulate a double-click ---
console.log('Patient double-clicks "Request Refill"...');
requestRefillWithLock('mrn-123', 'prozac-20mg');
requestRefillWithLock('mrn-123', 'prozac-20mg');
```

# In closing...

Thank you!

# Powering a Real-Time Patient Feed (5 mins)

Mockup of a patient's home screen feed: "Upcoming Appointment," "New Message from Dr. Smith," "Refill Ready."

Show `ZADD`, `ZREVRANGEBYSCORE`.

# Conclusion & Vision (3 minutes)

List the 4 patterns discussed: Geospatial, Sorted Sets (Feeds), Distributed Locks, and Atomic Counters (Analytics).

Reiterate the theme: moving from a utility to a strategic tool.

## Use Case: Caching the Matching Engine

- **The Problem:** The patient-provider matching process is computationally expensive. Users often refresh the screen without changing search criteria, triggering the same intensive calculation repeatedly
- **The Solution:** Cache the results of each unique matching request in Redis.
    - A unique cache key is created based on the specific search parameters (e.g., date, time window, provider preferences).
    - When a user refreshes with the same criteria, the results are served instantly from the cache instead of re-running the match.
- **The Impact:**
    - Drastically reduces server load on the matching service.
    - Creates a faster, more responsive experience for patients looking for a provider.

# Driving Adherence with Personalized Streaks

A mockup of a private view within the patient portal. It shows a simple, encouraging message like: "You're on a 4-week streak for completing your check-ins!" or "You've attended 3 appointments in a row. Great job staying on track!"

No scores, no comparisons, just positive reinforcement.

**B. Building a Smarter, Faster Patient Feed (5 mins)**

Mockup of a patient's home screen feed: "Upcoming Appointment," "New Message from Dr. Smith," "Refill Ready."

Show commands like `ZADD`, `ZREVRANGEBYSCORE`.

**Preventing Double-Bookings with Distributed Locks (5 mins)**

UI showing two users trying to book the same appointment slot simultaneously. One succeeds, one gets an "already booked" message.

Show a pseudo-code implementation of the Redlock algorithm.

# Rate Limiting (5 mins)

A simple dashboard showing "Messages Sent per Minute" or "Failed Logins per User."

Show commands like `INCR`, `EXPIRE`.

# State Machine for Patient Intake & Onboarding

- A diagram showing the flow of the 10-minute patient assessment: `InsuranceCheck` -> `CoverageVerified` -> `InitialQuestions` -> `SchedulingOffered` -> `Completed`.
- Show Redis Hash commands: `HSET`, `HGETALL`.