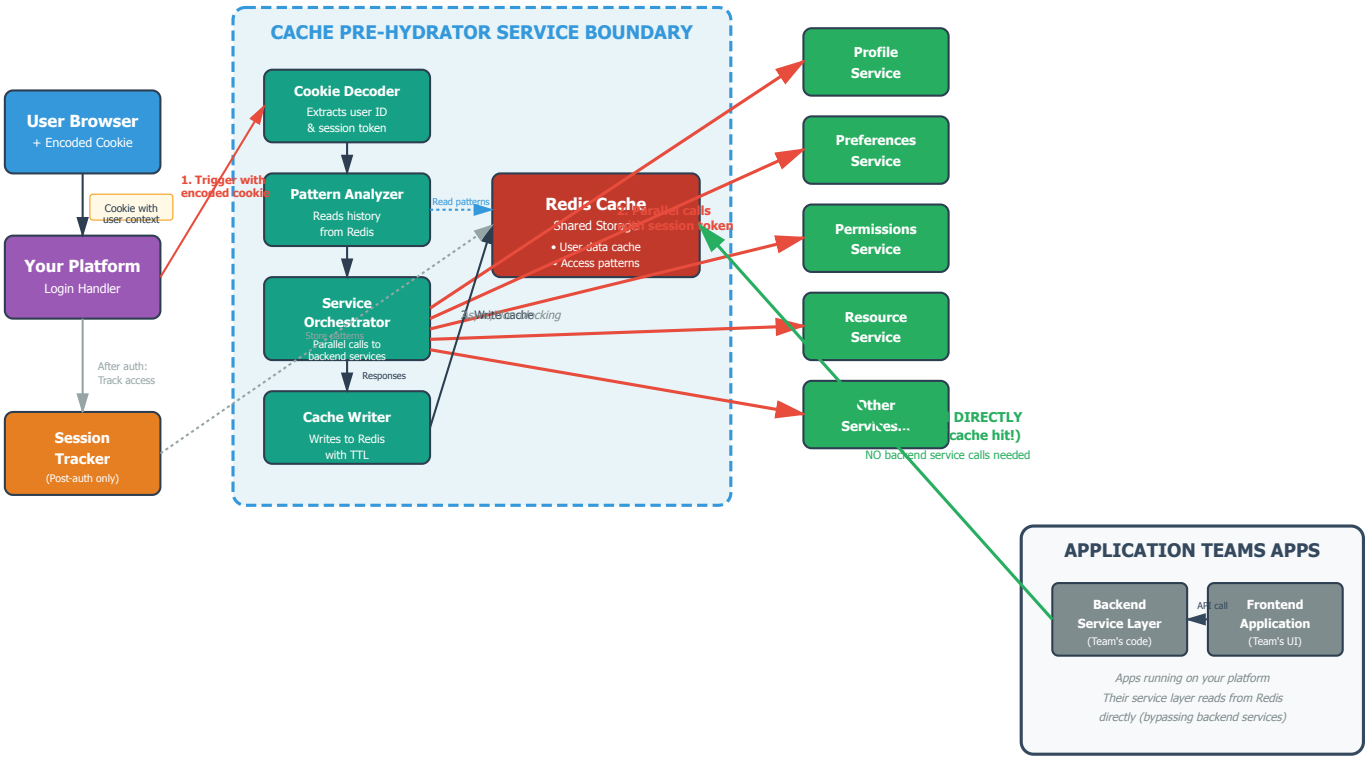
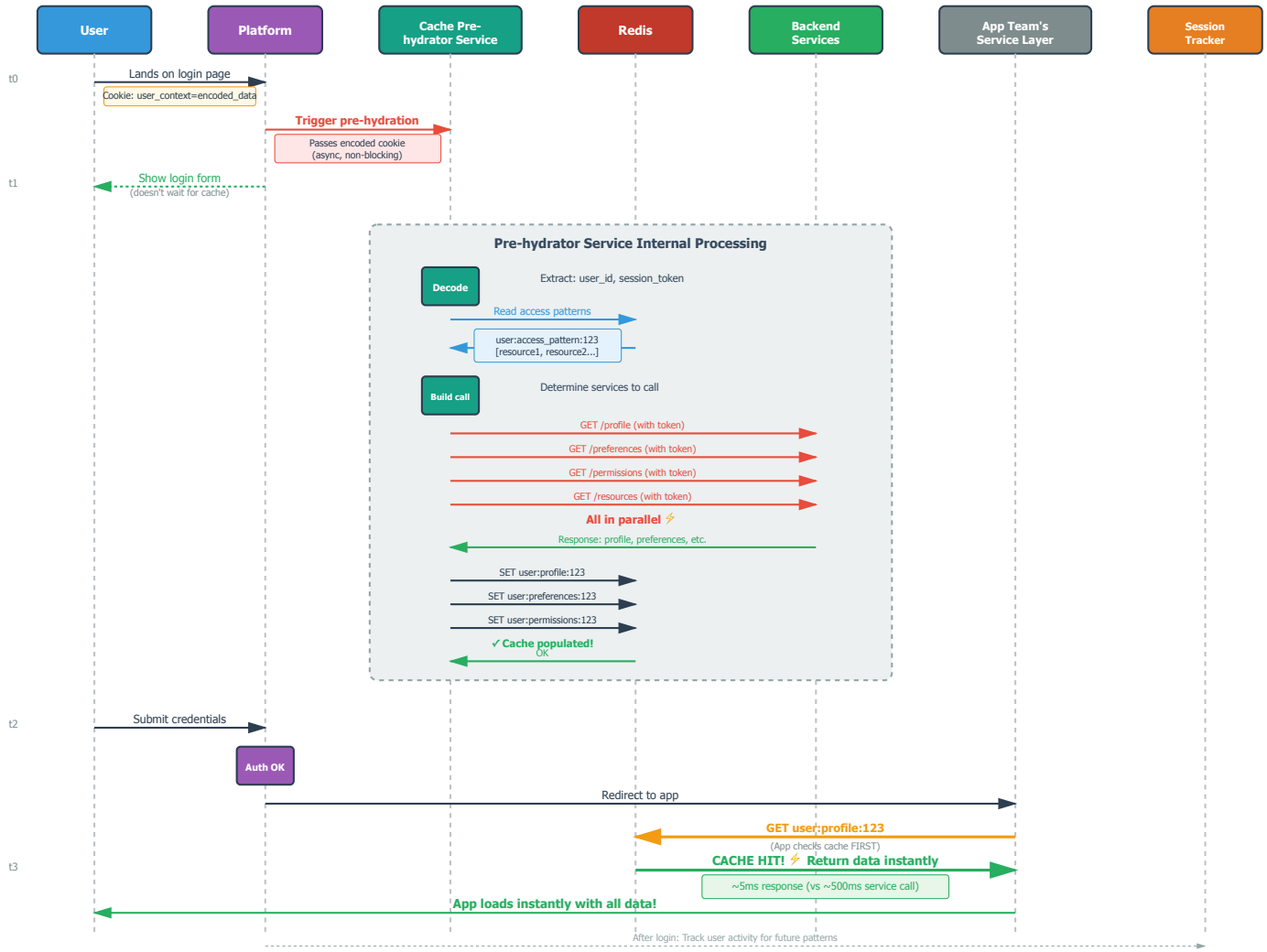
# Cache Pre-hydration Architecture & Flow

## 1. System Architecture & Boundaries




## 2. Detailed Sequence Flow (Cookie-Based Pre-hydration)




User/Client  Your Platform  Session Tracker  Cache Pre-hydrator

Backend Services  Redis Cache  Application Teams

## Key Benefits

- **Performance:** Cache is ready before user completes login - sub-100ms response times
- **User Experience:** Application loads instantly after authentication
- **Reduced Load:** Fewer calls to backend services during peak login times
- **Scalability:** Async processing doesn't block the login flow

- **Smart Caching:** Uses historical patterns to pre-load most relevant data

## Cache Pre-hydrator Service: Roles & Boundaries

### ✅ What the Pre-hydrator IS Responsible For:

- **Cookie Decoding:** Extract user ID and session token from encoded cookie
- **Pattern Analysis:** Read user's historical access patterns from Redis
- **Service Orchestration:** Make parallel calls to multiple backend services using the session token
- **Cache Population:** Write responses to Redis with appropriate keys and TTLs
- **Error Handling:** Gracefully handle service failures without impacting login flow
- **Performance Monitoring:** Track cache hit rates, latencies, and success rates

### ❌ What the Pre-hydrator is NOT Responsible For:

- **Authentication:** Does NOT authenticate users - relies on existing session token from cookie
- **Data Serving:** Does NOT serve data to applications - apps read from Redis directly
- **Session Creation:** Does NOT create or manage sessions - only uses existing ones
- **Business Logic:** Does NOT implement app-specific logic - only fetches and caches
- **Cache Reads:** Does NOT intercept or proxy cache reads from applications

### 🎯 Critical Design Decision: Direct Redis Access

**Q: Do applications read from Redis directly or go through the pre-hydrator?**

**A: Applications read from Redis DIRECTLY. Here's why:**

1. **Performance:** Direct Redis reads are fastest (~1-5ms). Adding another service layer adds latency.
2. **Scalability:** Redis can handle millions of reads/sec. No bottleneck from pre-hydrator service.
3. **Simplicity:** Applications use standard Redis client libraries - no custom integration needed.

4. **Reliability:** No single point of failure. If pre-hydrator is down, apps still work (cache miss → fallback to backend).
5. **Separation of Concerns:** Pre-hydrator is WRITE-ONLY for proactive caching. Apps are READ-ONLY consumers.

### 📋 Typical Application Code Pattern:

```
// Application Team's Service Layer Code
async getUserProfile(userId) {
    // 1. Try cache FIRST (populated by pre-hydrator)
    const cached = await redis.get(`user:profile:${userId}`);
    if (cached) {
        return JSON.parse(cached);  // ⚡ ~5ms
    }

    // 2. Cache miss - fallback to backend service
    const profile = await profileService.get(userId);  // ~500ms

    // 3. Store in cache for next time
    await redis.setex(`user:profile:${userId}`, 3600, JSON.stringify(profile));

    return profile;
}
```

**The pre-hydrator's job is to ensure that step #1 (cache check) returns a HIT instead of proceeding to step #2 (slow backend call).**

## Important Considerations

- **Security:** Validate and encrypt session tokens; rotate regularly
- **Privacy:** Clear cached data on logout; comply with data retention policies
- **Graceful Degradation:** App must work even if pre-hydration fails
- **Timeout Strategy:** Use aggressive timeouts (3-5s) for pre-hydration calls
- **Monitoring:** Track cache hit rates, pre-hydration success rates, and latencies