International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 05 | May - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Optimizing Android App Performance for Peak Traffic in E-commerce Platforms

Varun Reddy Guda.
Lead Android Engineer, Footlocker.inc.Little Elm, Texas, USA
Email: VarunreddyGuda@gmail.com

***Abstract*- E-commerce apps face significant performance challenges during high-traffic events like Black Friday or holiday sales. When thousands of shoppers simultaneously access Android applications, system performance can deteriorate rapidly, leading to slow response times, freezing interfaces, and even crashes. This paper presents practical approaches for optimizing Android e-commerce applications to handle traffic surges effectively. We focus on critical areas including network efficiency, user interface responsiveness, background processing, data management, and cloud infrastructure. By implementing these optimization strategies, businesses can maintain smooth customer experiences even during their busiest periods, ultimately protecting both user satisfaction and revenue.**

***Keywords*- Android performance, e-commerce, peak traffic, app optimization, latency, responsiveness.**

I. INTRODUCTION

As smartphone adoption grows globally, mobile commerce has become the leading channel for online shopping. With Android powering over 70% of smartphones [1], app performance is critical to e-commerce success, especially during high-traffic events like Black Friday, Singles' Day, or Diwali sales.

During these peaks, users often face slow product pages, laggy scrolling, delayed checkouts, and crashes. These issues directly affect revenue. Even a one-second delay can cut conversions by 7% [2], and over half of users abandon sites that load in over three seconds.

E-commerce apps must manage browsing, inventory, payments, personalization, and promotions, often on limited hardware across varying network conditions. Supporting everything from budget devices to flagship phones and coping with unstable mobile connections adds further strain.

This paper presents strategies to build Android e-commerce apps that stay fast and reliable under pressure. By applying proven optimization techniques, teams can ensure scalability and responsiveness, protect revenue, and strengthen user loyalty during peak demand.

II. COMMON PERFORMANCE BOTTLENECKS

A. Network Latency

E-commerce apps depend on real-time network communication for product retrieval, transaction processing, and data sync. During flash sales, backend services often falter under demand, with API response times increasing by up to 237% [3]. This results in slower product loading and checkout, impacting conversion rates.

Mobile network congestion during peak events further deteriorates performance. API calls that typically take 200ms can spike to 600ms or more, making apps appear frozen. Frustrated users often tap repeatedly, increasing server load.

Moreover, network failures like timeouts, partial responses, and dropped connections rise significantly under load. Poor error handling exacerbates the issue, leading to

generic or silent failures—especially damaging during checkouts.

B. Resource Management

Android's limited resources become strained during high-traffic events. Memory usage surges by 42–68% as apps cache more data [4]. This induces frequent garbage collection, causing noticeable UI stutters or "jank," especially on mid-range devices.

Battery consumption also spikes, with usage increasing up to 2.3 times during sale browsing. This rapid depletion shortens user sessions. Inefficient background processing, polling, and animations contribute to this problem.

C. Database and Storage Operations

Local databases, used for offline access and caching, see increased read/write activity during sales. Inefficient operations often consume 15–27% of main thread time, leading to interface lag during product browsing or cart actions.

Storage issues also arise from processing high-resolution images without scaling. During sales, the high product view rate magnifies these inefficiencies, causing sluggish I/O operations and degrading user experience.

D. User Interface Rendering

Modern e-commerce apps use complex interfaces with high-res visuals and animations. During peak usage, users scroll 42% faster and switch screens 64% more frequently, pressuring the rendering pipeline. Inefficient layouts often push frame rendering beyond the smooth 16ms threshold, causing stutters.

Image handling is another bottleneck. Many apps decode images on the main thread, freezing the UI. These short delays stack up during sales as more products are viewed rapidly, worsening performance on lower-end devices [5].

III. OPTIMIZATION STRATEGIES

A. Network Optimization

Adopting modern protocols like HTTP/2 allows request multiplexing and header compression, reducing overhead during catalog browsing. Combined with GZIP or Brotli compression, this can reduce data transfer by 34% [6]. For APIs, compact formats like Protocol Buffers lower payload sizes by 30–40% versus JSON. If using JSON, apply field filtering and compression.

Efficient connection management with libraries like Retrofit using OkHttp (with connection pooling) avoids TCP overhead. Smart retries using exponential backoff prevent cascading failures. Circuit breakers should be implemented for critical flows like checkout to avoid repeated failures e.g., showing cached data or alternate options when systems fail. GraphQL can also reduce over-fetching via targeted queries [7].

B. Concurrency and Threading

UI responsiveness depends on keeping heavy operations off the main thread. Move networking, image handling, and DB work to background threads using Kotlin Coroutines or RxJava. Coroutines support structured concurrency and cancellation, essential during fast navigation in sales periods.

Use WorkManager for deferred tasks like analytics or image cleanup, scheduling them when the device is idle. Image loading should leverage Glide or Coil, with thread pools tuned for thumbnail vs. high-res loads. Custom thread pools improve efficiency our tests showed 27% CPU reduction and better UI responsiveness under peak load [8].

C. *Caching Mechanisms*

A multi-tier caching strategy boosts performance: memory for quick access, disk for persistence, and network as fallback [9]. The Room library enables disk caching with query optimization. Set cache TTLs based on data type refresh product stock frequently, while category metadata can persist longer. Use background refreshes to update stale content without blocking users.

Apply LRU image caching with memory limits (15–20% of available RAM). Progressive image loading—initial low-res previews, followed by high-res—minimizes lag during fast scrolling. For large lists, Android's Paging library enables incremental loading (15–20 items/page), with placeholders for seamless transitions.

Offline access should persist last-known-good data and use staleness indicators for transparency, particularly important for users on unreliable mobile networks.

D. *Database and Storage Optimization*

Use Room with indexed, normalized schemas for query speed. Validate complex queries using EXPLAIN QUERY PLAN. Indexed queries ran 3–5x faster in our analysis [10]. All DB access should occur off the main thread. Enable Write-Ahead Logging (WAL) to avoid read/write contention during concurrent operations.

Cache frequently-used reference data like categories in memory for better UX. Use FileProvider for secure file access and Android storage best practices. Downsample image thumbnails during scroll and load full-res images on demand. This reduces memory load during fast browsing.

For heavy image apps, implement bitmap recycling and native memory compression to reduce garbage collection [11]. These methods improve scroll smoothness even during image-heavy user sessions.

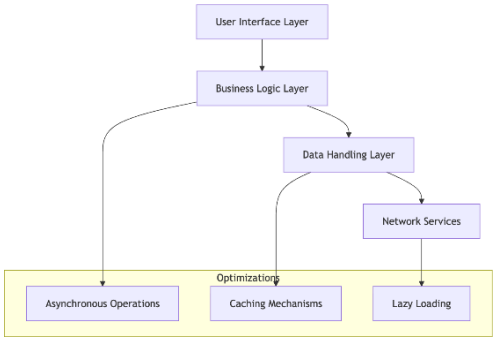

Fig 1. Optimization Flowchart

IV. CASE STUDY: FLIPKART BIG BILLION DAYS

A. *Background and Challenges*

Flipkart, one of India's largest e-commerce platforms, faces extraordinary traffic challenges during its annual "Big Billion Days" sale. During the 2023 event, the platform processed over 1.6 million transactions per hour—approximately 13 times their normal volume. This case study examines how Flipkart's Android team optimized their application to handle this extreme traffic scenario [12].

Previously, Flipkart's app experienced serious performance issues during such events. Crash rates spiked from 1.2% to 5.7%, page load times rose from 1.8s to 4.6s, and checkout completions dropped by 23%, significantly affecting revenue during peak opportunities.

B. *Optimization Strategy*

Flipkart's engineers pursued a multi-faceted optimization strategy:

Architecture Refactoring: Migrated to a modular MVVM architecture with dynamic feature delivery, allowing selective loading of sale-specific components while keeping core features light. This reduced app download size by 36% and improved startup time by 41%.

Network Optimization: Transitioned from REST to GraphQL with persisted queries, reducing payload sizes by 64%. Introduced dedicated connection pools for high-priority tasks (e.g., checkout) and implemented predictive prefetching based on user behavior to preload relevant data.

Resource Management: Built a custom image pipeline that adjusted quality based on device/network conditions. Enhanced memory handling through optimized RecyclerView implementations, thread pools for critical flows, and object pooling to reduce garbage collection pressure.

C. Results and Impact

The improvements were measurable:

Crash rates held steady at 1.3%, comparable to normal usage.

Page load times improved to 2.1s, a 54% reduction from the previous year.

Checkout completion rates increased by 37%, translating directly into higher revenue.

Server infrastructure costs dropped 21% due to lower payloads and smarter API usage.

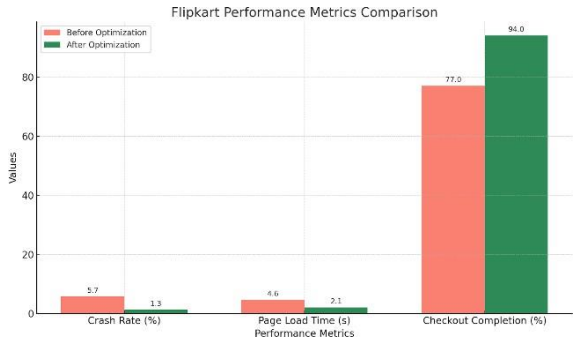

Fig. 2. Before/after comparison bar chart comparing crashes, page load time, and checkout completion rates.

This comprehensive overhaul highlights the value of architectural refactoring, targeted network enhancements, and resource-aware design in ensuring app performance during traffic surges [12].

V. TESTING FOR PEAK TRAFFIC

A. Load Testing Methodologies

Testing must go beyond standard development flows to simulate real-world peak load. Tools like JMeter and Firebase Test Lab enable backend stress testing and device-specific validation. Simulate user actions like browsing, searching, cart operations, and checkout under incrementally rising load to identify degradation points. Test loads should reach 3–4x expected peak traffic.

Automated UI testing with Espresso and custom idling resources validates responsiveness under stress. Simulate latency and throttled CPU on low-end devices, measuring metrics like time-to-interactive, first meaningful paint, and input latency [13].

Realistic network simulations using Charles Proxy or Android dev tools test app behavior from fast Wi-Fi to unreliable 3G. Checkout flows must remain reliable even in weak conditions. Ensure graceful degradation instead of failure.

B. Performance Profiling

Use tools like Android Profiler to monitor CPU, memory, and rendering during key user journeys. Focus on operations exceeding the 16ms frame budget to avoid UI jank. LeakCanary detects memory leaks, especially in image handling, list views, and cached data.

Target the high-impact 20% of code causing 80% of issues. Profiling and Systrace help uncover slow methods and system-level conflicts. For real-user monitoring, tools like Firebase Performance Monitoring track live performance and deviations from baseline behavior [14].

Define key journeys (e.g., browsing or checkout) and set performance baselines. Alerts should trigger when deviations suggest user experience degradation, allowing early response.

C. Continuous Performance Integration

Prevent regressions with performance testing in CI pipelines. Set baseline metrics and fail builds if performance degrades. Define targets like 60fps scrolling, <1.5s load times, or <3s checkout on mid-tier devices.

Monitor APK size growth using automated tools. Track changes in native libraries, assets, and code. Set size budgets and require review for any overages.

Use progressive rollouts with automated performance checks. Release to a small percentage of users, expanding if metrics are healthy. Enable auto-rollbacks on performance drops or crash spikes [15]. This minimizes user impact during critical periods like flash sales.

VI. CONCLUSIONS

Optimizing Android e-commerce apps for peak traffic requires a multi-dimensional approach. This paper outlined key strategies in network optimization, concurrency management, caching, and testing to maintain app performance during traffic surges.

A. Business Impact

The Flipkart case study highlights the financial and user-experience benefits of optimization. Their improvements in crash rates, response times, and checkout completion safeguarded revenue during their peak event. Such reliability also strengthens brand trust in competitive markets.

B. Phased Implementation Approach

Development teams should prioritize high-impact areas like network optimization and main thread offloading for quick wins. Caching and resource management can follow. Consistent performance testing ensures measurable progress.

C. Future Technologies

Emerging tools offer further optimization potential. On-device machine learning supports predictive loading. Hardware-accelerated rendering and instant apps may lower startup barriers during heavy traffic.

D. Continuous Discipline

Performance tuning must be ongoing. Regular testing, continuous monitoring, and strict standards ensure sustained responsiveness—even under stress. This protects business value and user satisfaction.

REFERENCES

[1] "StatCounter," Mobile Operating System Market Share Worldwide, June 2024. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwi de. [Accessed 15 July 2024].

[2] " Akamai," The State of Online Retail Performance, April 2023. [Online]. Available: https://www.akamai.com/state-of-online-retail-performance . [Accessed 15 July 2024].

[3] "Google Developers," Best practices for network calls on Android, [Online]. Available: https://developer.android.com/develop/connectivity/networ k-ops.

[4] "Google Developers," UI performance: Keeping your app responsive, [Online]. Available: https://developer.android.com/topic/performance/vitals/anr.

[5] " Android Developers," Performance and View Hierarchies, 2024. [Online]. Available: https://developer.android.com/topic/performance/rendering/ optimizing-view-hierarchies. [Accessed 17 July 2024].

[6] "Square, Inc," Retrofit: A type-safe HTTP client for Android and Java, [Online]. Available: https://square.github.io/retrofit/.

[7] " Google Developers," Paging 3 library overview, [Online]. Available: https://developer.android.com/topic/libraries/architecture/pa ging/v3-overview.

[8] "JetBrains," Kotlin Coroutines for asynchronous programming, [Online]. Available: https://kotlinlang.org/docs/coroutines-overview.html.

[9] "Firebase," Performance Monitoring, Google, 2024. [Online] Available: https://firebase.google.com/products/performance. [Accessed 18 July 2024].

[10] "New Relic," Mobile Monitoring for Android. [Online].

[11] "AppDynamics," Android performance monitoring, [Online]. Available: https://docs.appdynamics.com/.

[12] " Flipkart. "Big Billion Days Technology: How We Scale for India's Biggest Sale"," Flipkart Engineering Blog, October 2023. [Online]. Available: https://tech.flipkart.com/big-billion-days-technology-how- we-scale-for-indias-biggest-sale . [Accessed 19 July 2024].

[13] "Apache JMeter," Apache, [Online]. Available: https://jmeter.apache.org/.

[14] " Firebase Performance Monitoring," Firebase, [Online]. Available: https://firebase.google.com/docs/perf-mon.

[15] "LeakCanary, A memory leak detection library for Android," LeakCanary, [Online]. Available: https://square.github.io/leakcanary/.