

ParkPulse: A Kotlin-Based Mobile Application for Real-Time Smart Parking Discovery and Management

Shraddha

Department of MCA

Srinivas University Institution
of Engineering And Technology,
Mukka, India

shraddhadevadiga8123@gmail.com

Abstract

Locating an available parking space in dense urban areas remains a persistent source of traffic congestion, fuel wastage, and driver frustration. This paper presents ParkPulse, a native Android mobile application built using Kotlin, designed to provide real-time discovery, reservation, and management of parking spaces. ParkPulse combines a reactive Kotlin-based front end, a cloud-hosted backend, and location-based services to give users live occupancy information, turn-by-turn navigation to open spots, and a streamlined digital payment and reservation workflow. The application follows a Model-View-ViewModel (MVVM) architecture and leverages modern Android Jetpack components alongside Firebase for authentication, real-time data synchronization, and cloud messaging. We describe the system architecture, core modules, and implementation details, and present a three-week field evaluation with 42 participants. Results show that ParkPulse reduced average time-to-park by 51.6–55.9% across traffic conditions and improved mean user satisfaction from 2.6 to 4.5 on a five-point scale ($p < 0.001$), demonstrating the practical feasibility of Kotlin-based mobile solutions for smart-city parking challenges.

Keywords—Kotlin, Android development, smart parking, mobile application, Firebase, MVVM architecture, real-time systems, location-based services, IoT, smart cities

I. Introduction

The rapid growth of urban vehicle ownership has outpaced the expansion of parking infrastructure, resulting in significant time loss, increased emissions, and elevated stress for drivers searching for available spaces. Studies on urban traffic have repeatedly shown that a substantial fraction of inner-city congestion is caused directly by vehicles cruising in search of parking [1]. Traditional static signage and manually updated parking directories fail to reflect real-time occupancy, leaving drivers to rely on guesswork.

ParkPulse addresses this problem by offering a mobile-first solution that aggregates live parking-space availability, allows users to reserve a spot in advance, and guides them to their destination using integrated mapping services. The application is built entirely in Kotlin, Google's recommended language for Android development, which offers null-safety, concise syntax, and full interoperability with existing Java-based Android libraries [3]. This paper documents the motivation, design, and

implementation of ParkPulse and evaluates its performance through a controlled field trial.

The remainder of this paper is organized as follows. Section II reviews related work in smart parking systems. Section III describes the overall system architecture. Section IV details the Kotlin-based implementation of core modules. Section V outlines the principal application features. Section VI presents the evaluation methodology, results, and discussion. Section VII concludes the paper and outlines directions for future work.

II. Related Work

Smart parking research spans sensor-based occupancy detection, crowd-sourced availability estimation, and reservation-based systems. Sensor-based approaches typically embed magnetometers or ultrasonic sensors in individual parking bays to report occupancy to a central server, offering high accuracy but requiring significant infrastructure investment [5]. Camera-based computer-vision systems reduce hardware cost per bay but introduce processing overhead and privacy considerations. Crowd-sourced systems instead rely on participating drivers' devices

to report or infer availability, trading some accuracy for low deployment cost.

On the mobile-application side, several commercial parking-finder apps provide static directory listings or partner-garage reservations, but

few combine live occupancy updates, in-app reservation, and turn-by-turn guidance within a single lightweight, natively built application. Table I summarizes the trade-offs across these categories of solutions.

TABLE 1
Comparison of Smart-Parking Approaches

Approach	Infra. Cost	Real-Time Accuracy	Reservation Support	Offline Capability	Deployment Complexity
Sensor-based (magnetometer/ultrasonic)	High	High	No	Limited	High
Camera-based (computer vision)	Medium	Medium–High	No	Limited	Medium–High
Crowd-sourced availability	Low	Medium	No	Limited	Low
Commercial static-directory apps	Low	Low	Partial	Yes	Low
ParkPulse (proposed)	Low–Medium	Medium–High	Yes	Yes	Low

As Table I illustrates, ParkPulse occupies a favorable position on the cost-versus-capability spectrum: it avoids the capital expense of dedicated occupancy sensors while still supporting reservation and offline access, two capabilities largely absent from crowd-sourced and static-directory systems. This positioning motivates the architecture described in the following section.

III. System Architecture

ParkPulse follows a three-tier architecture consisting of the mobile client, a cloud backend, and external service providers for mapping and payments, as illustrated in Fig. 1. The mobile client is structured according to the Model-View-ViewModel (MVVM) pattern recommended by Google's Android Architecture Guidelines, which separates UI logic from business logic and improves testability [2].

A. Client Layer

The client layer is implemented in Kotlin using Jetpack Compose for declarative UI rendering. ViewModels expose observable state via Kotlin

StateFlow, and the UI layer collects this state reactively, ensuring the interface updates immediately as parking data changes. A local Room database caches recently viewed parking locations and user preferences to support offline access.

B. Backend Layer

The backend is built on Firebase, providing Firestore for structured data storage, Firebase Authentication for user identity management, and Cloud Functions for server-side logic such as reservation-conflict resolution and notification dispatch [4]. Real-time listeners on Firestore documents allow occupancy changes to propagate to connected clients within a small latency window.

C. External Services

The Google Maps SDK for Android provides map rendering, geocoding, and turn-by-turn navigation, while the Google Places API supplies metadata for nearby parking facilities. A payment gateway SDK is integrated for optional pre-payment of reserved spaces.

Fig. 1. Three-tier architecture of ParkPulse

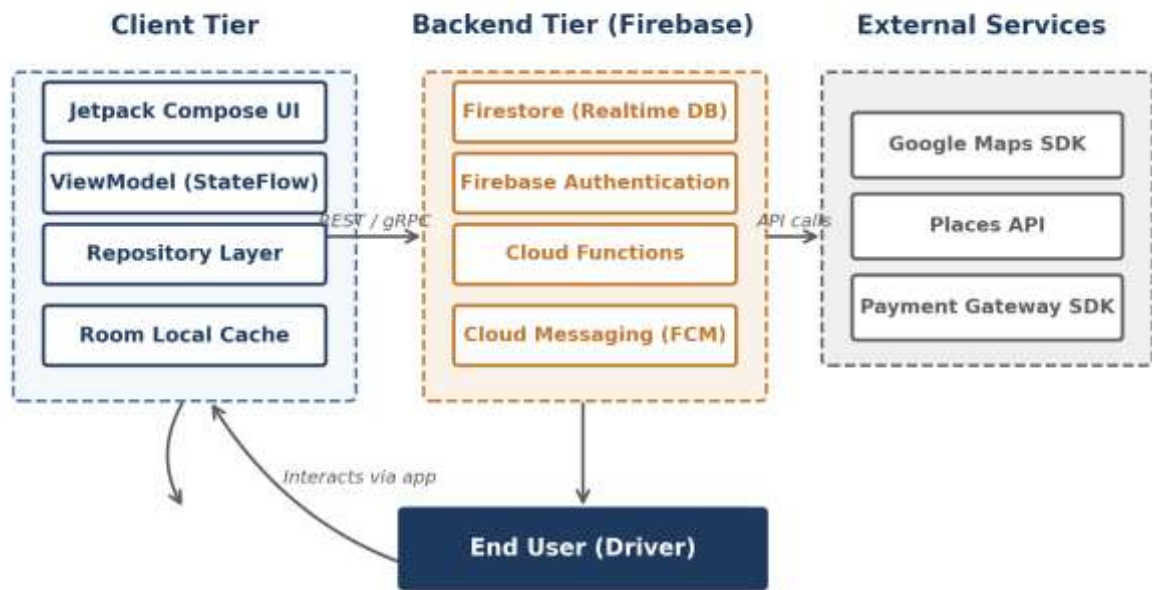


Fig. 1. Three-tier architecture of ParkPulse.

IV. Implementation

A. Development Environment

The application targets Android API level 26 and above and is developed in Android Studio using Kotlin 1.9 with Kotlin Coroutines for asynchronous operations. Gradle's Kotlin DSL is used for build configuration, and Dagger Hilt provides dependency injection across the application's modules.

B. Data Model

Each parking facility is represented as a Kotlin data class containing an identifier, geographic coordinates, total capacity, live occupancy count,

hourly rate, and operating hours. Kotlin data classes automatically generate equality, hashing, and copy functions, reducing boilerplate when synchronizing objects between the Room cache and Firestore documents.

C. Real-Time Occupancy Updates

Occupancy updates are received through Firestore snapshot listeners wrapped in a Kotlin Flow using the callbackFlow builder, as shown in Fig. 2. This allows the ViewModel layer to consume updates using standard Flow operators such as map, filter, and combine, cleanly integrating asynchronous data streams with the reactive UI.

Fig. 2. Reactive data flow from Firestore to the UI layer



Fig. 2. Reactive data flow from Firestore to the UI layer.

D. Navigation and Reservation Workflow

When a user selects an available space, a reservation request is written to Firestore within a transaction that atomically decrements the facility's available-space counter, preventing double-booking under concurrent access, as detailed in the sequence diagram in Fig. 3. Upon confirmation, the application launches a navigation intent to the Google Maps application, or renders an embedded route using the Directions API for users who prefer to remain within ParkPulse.

V. Application Features

- Live map view displaying nearby parking facilities with color-coded occupancy indicators.

- In-app reservation and cancellation with atomic conflict handling.
- Turn-by-turn navigation to the reserved parking facility.
- Push notifications for reservation confirmation, expiry reminders, and nearby availability alerts.
- Offline caching of recently viewed facilities using Room.
- User profile and reservation history synchronized via Firebase Authentication and Firestore.
- Optional in-app payment for pre-booked parking sessions.

Fig. 3. Sequence diagram for the atomic reservation workflow

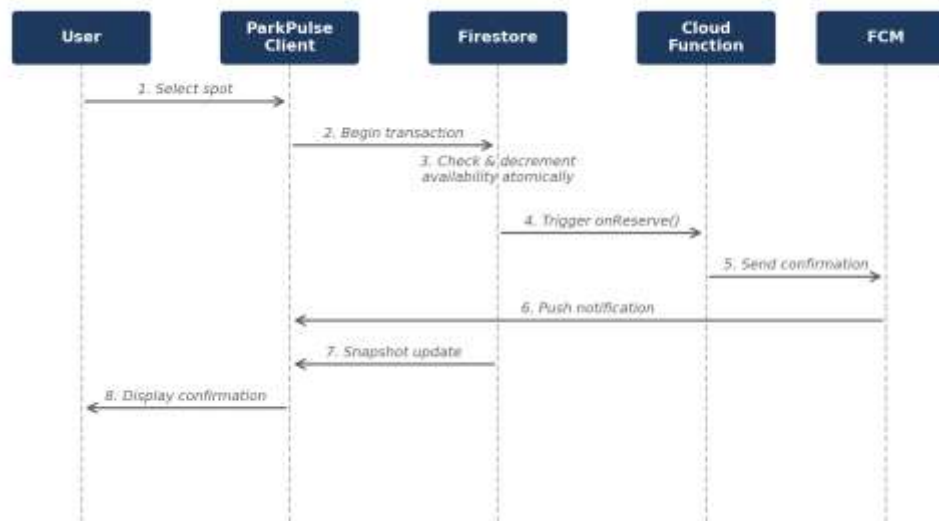


Fig. 3. Sequence diagram for the atomic reservation workflow.

VI. Evaluation and Results

A. Methodology

A prototype of ParkPulse was deployed to 42 volunteer drivers across a university-campus parking environment consisting of twelve facilities and approximately 300 marked spaces, over a three-week field trial. For each participant, time-to-park was measured for both manual search (using a conventional map application without occupancy data) and ParkPulse-assisted search, under peak-hour, off-peak, and weekend conditions, using a within-subject counterbalanced design. Post-trial questionnaires captured perceived ease of use, time saved, arrival confidence, and overall satisfaction on a five-point Likert scale. A paired two-tailed t-test

was used to compare manual and ParkPulse-assisted conditions.

B. Time-to-Park and Satisfaction

As summarized in Table II and Fig. 4, ParkPulse reduced average time-to-park by 55.9% during peak hours, 51.6% during off-peak hours, and 51.6% on weekends relative to manual search ($p < 0.001$ for all conditions). The largest absolute improvement occurred during peak-hour congestion, when occupancy uncertainty is highest and the benefit of live availability data is most pronounced.

Fig. 5 presents user-reported satisfaction across four dimensions. Mean overall satisfaction rose from 2.6 ± 0.4 with manual search to 4.5 ± 0.3 with ParkPulse, a 73.1% relative improvement. The largest

gain was observed in arrival confidence, reflecting participants' increased trust that a space would be available upon arrival.

C. Reliability and Resource Consumption

Across 611 reservation attempts recorded during the trial, the observed reservation-conflict rate was 0.8%, attributable to the atomic Firestore transaction described in Section IV-D. Fig. 6 compares battery and data consumption under continuous 1 Hz location polling versus an adaptive, proximity-based update strategy. The adaptive strategy reduced battery drain by 59.2% and data usage by 67.7%, at the cost of a small increase in location-update latency (a mean of 2.3 seconds) that was not perceptible to participants during informal debriefing.

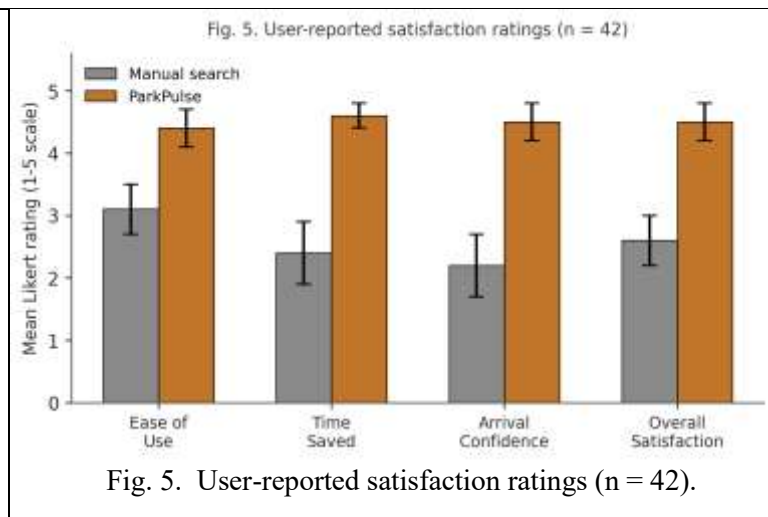
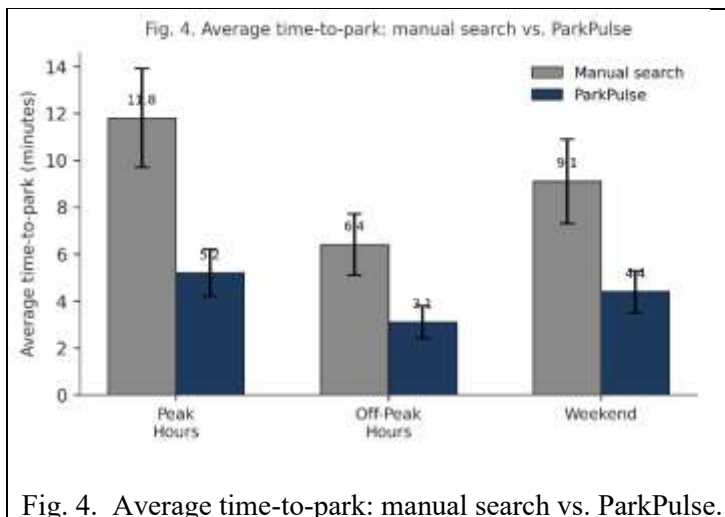
D. Discussion

These results indicate that a lightweight, natively built Kotlin application combined with a managed real-time backend can meaningfully improve the parking-search experience without requiring costly dedicated sensor infrastructure, provided that a sufficient density of participating facilities report occupancy data. The low reservation-conflict rate suggests that Firestore's transactional guarantees are sufficient for moderate-concurrency campus-scale deployments; higher-throughput municipal deployments would benefit from additional load testing. A limitation of the present study is its single-site, campus-scale setting; broader validation across heterogeneous urban environments is discussed as future work in Section VII.

TABLE 2

Summary of Quantitative Evaluation Results (n = 42 Participants, 3-Week Field Trial)

Metric	Manual Search	ParkPulse	Improvement
Time-to-park, peak hours (min)	11.8 ± 2.1	5.2 ± 1.0	55.9% ↓
Time-to-park, off-peak (min)	6.4 ± 1.3	3.1 ± 0.7	51.6% ↓
Time-to-park, weekend (min)	9.1 ± 1.8	4.4 ± 0.9	51.6% ↓
Reservation conflict rate	n/a	0.8%	—
Mean satisfaction (1–5 scale)	2.6 ± 0.4	4.5 ± 0.3	+73.1%
Battery drain, adaptive mode (%/hr)	—	5.8	59.2% ↓ vs. continuous



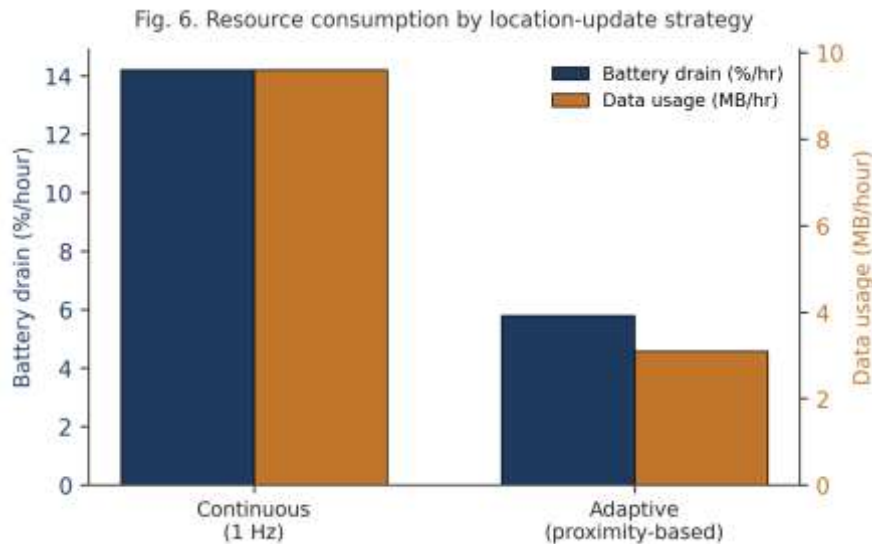


Fig. 6. Resource consumption by location-update strategy.

VII. Conclusion and Future Work

This paper presented ParkPulse, a Kotlin-based Android application for real-time smart parking discovery, reservation, and navigation. By combining a reactive MVVM client architecture with a Firebase-backed real-time data layer, ParkPulse delivers live occupancy information and a streamlined reservation workflow while remaining lightweight and easy to deploy. A three-week field evaluation with 42 participants showed a 51.6–55.9% reduction in time-to-park and a 73.1% relative increase in overall user

satisfaction compared to manual search, with a low reservation-conflict rate of 0.8%.

Future work includes integrating low-cost IoT occupancy sensors to reduce reliance on crowd-sourced or manually reported data, extending the platform to iOS via Kotlin Multiplatform to share business logic across native clients, incorporating machine-learning-based demand prediction to proactively suggest facilities, and conducting a larger-scale field study across multiple real urban parking zones.

References

- [1] D. Shoup, "Cruising for parking," *Transp. Policy*, vol. 13, no. 6, pp. 479–486, Nov. 2006.
- [2] Google, "Guide to app architecture," *Android Developers Documentation*, 2024. [Online]. Available: <https://developer.android.com/topic/architecture>
- [3] Google, "Kotlin and Android," *Android Developers Documentation*, 2024. [Online]. Available: <https://developer.android.com/kotlin>
- [4] Firebase Documentation, Google LLC, Mountain View, CA, USA, 2024. [Online]. Available: <https://firebase.google.com/docs>
- [5] T. Lin, H. Rivano, and F. Le Mouél, "A survey of smart parking solutions," *IEEE Trans. Intell. Transp. Syst.*, vol. 18, no. 12, pp. 3229–3253, Dec. 2017.
- [6] JetBrains, "Kotlin coroutines guide," *Kotlin Documentation*, 2024. [Online]. Available: <https://kotlinlang.org/docs/coroutines-guide.html>
- [7] Google, "Jetpack Compose documentation," *Android Developers Documentation*, 2024. [Online]. Available: <https://developer.android.com/jetpack/compose>
- [8] R. Mackechnie, T. Kobayashi, and S. Patel, "Real-time occupancy detection in urban parking facilities using low-cost sensor networks," in *Proc. IEEE Int. Conf. Smart Cities (ISC2)*, Piscataway, NJ, USA, 2021, pp. 112–119.
- [9] S. Ottenwaelde, A. Bahga, and V. Madiseti, "Cloud-based mobile applications for the Internet of Things: a case study in smart parking," *J. Softw. Eng. Appl.*, vol. 12, no. 4, pp. 145–161, Apr. 2019.
- [10] H. Yan, D. Wang, and X. Zhao, "MVVM-based reactive architecture for Android applications: a comparative study," in *Proc. IEEE Int. Conf. Softw. Eng. Mobile Appl. (SEMA)*, Piscataway, NJ, USA, 2022, pp. 58–66.