DATABASE ARCHITECTURE AND PAYMENT INTEGRATION IN AN AUTOMATED PARKING MANAGEMENT SYSTEM

Arofat Babaniyazova
Department of Applied Informatics
Kimyo International University in Tashkent (KIUT)
Tashkent, Uzbekistan
babaniyazovaarofat@gmail.com

Abstract. *This paper examines the data management and payment subsystems of an automated parking facility built around license plate recognition. The system employs a lightweight SQLite database operating in Write-Ahead Logging (WAL) mode to record real-time spot occupancy, full entry–exit histories, advance bookings and electronic payment transactions. A position-based tariff engine calculates fees at the moment of vehicle departure, and payment links for two widely used Uzbek e-wallet services — Payme and Click — are generated automatically. Advance reservations are handled through a REST API and a Telegram bot; a background scheduler cancels no-show bookings after a configurable grace period and logs a penalty charge. A JWT-secured role model governs write access to all sensitive endpoints. Evaluation over a two-week trial at a 24-space outdoor lot showed that the database layer sustained sub-200 ms API response times under 200 simultaneous WebSocket connections and recovered without data loss from three simulated power-failure interruptions, confirming the suitability of WAL-mode SQLite for small-to-medium parking deployments.*

Keywords: *automated parking, SQLite, WAL mode, electronic payment, Payme, Click, REST API, WebSocket, Telegram bot, JWT authentication.*

1. Introduction

Automating the financial and administrative core of a parking facility is at least as important as automating vehicle identification. Even when a license plate is recognised correctly, the value of that recognition is lost if the downstream system cannot reliably record the event, calculate the correct fee, present a payment option the driver is willing to use, and confirm the transaction — all within a few seconds [1].

In Uzbekistan the dominant retail payment channels are the Payme and Click e-wallet platforms, which together cover the majority of smartphone users. Any parking solution intended for real-world deployment must integrate with these services. At the same time, the system must operate at low cost on modest hardware: a Raspberry Pi or a small office server rather than a cloud-hosted database cluster [2].

This paper describes the data-layer and payment architecture developed as part of a broader automated parking project. The specific contributions are: (1) a four-table relational schema designed for high-frequency concurrent reads and writes in WAL-mode SQLite; (2) a tariff engine that computes fees at departure and generates platform-specific payment links; (3) a booking subsystem with automatic no-show detection and penalty logging; and (4) an empirical evaluation of database reliability under realistic concurrent load.

2. Related Work

Smart parking systems have been widely studied from the perspective of sensor networks and occupancy prediction [3, 4], but the data persistence and payment layers receive comparatively little attention in the literature. Geng and Cassandras [3] propose a reservation-based parking model and discuss the need for a central database to track space availability, yet leave the database design unspecified. Barone et al. [4] describe a city-scale parking management architecture that relies on a cloud relational database, which is inappropriate for small independent lots with limited connectivity.

SQLite is frequently recommended for embedded and edge applications due to its zero-configuration deployment and ACID compliance [5]. Its WAL mode allows concurrent readers to proceed without blocking a writer, making it well suited to parking systems where a camera thread writes events while a web server reads occupancy state for multiple clients simultaneously [5].

Link-based payment flows — where the system generates a URL that the driver opens on a personal device — have become practical as smartphone penetration has risen, but published work on their implementation in parking-specific software remains sparse [6].

3. Database Design

The system uses a single SQLite file opened in WAL mode with a connection pool of four threads. WAL mode was chosen because it permits concurrent reads during a write transaction, which is critical when the camera thread inserts an entry event while the web server is serving occupancy data to browser clients [5].

The schema comprises four tables, summarised in Table 1.

Table	Key columns	Purpose
spots	spot_number, status, plate, entry_time	Real-time occupancy state for every parking space
history	plate, spot, entry_time, exit_time, amount, type	Permanent record of every entry–exit event and fee
bookings	plate, owner_name, phone, spot, arrival_time, status	Advance reservations and their lifecycle

Table	Key columns	Purpose
payments	ref_id, amount, method, status, paid_at	Transaction records linked to a history row via ref_id

Table 1. Database schema overview.

The spots table is updated on every plate recognition event; its status column takes one of three values: free, occupied, or reserved. The history table is append-only: rows are inserted at entry and updated with exit_time, duration and amount at departure. The bookings table records advance reservations; the booking_checker background task polls it every five minutes and transitions stale bookings to no_show when the grace period expires. The payments table stores one row per payment attempt linked to its history row by a unique reference identifier.

All four tables carry composite indexes on the most frequently queried columns. Parameterised queries are used throughout to prevent SQL injection.

4. Tariff Engine and Payment Integration

When the license plate recognition module signals a vehicle departure, the backend retrieves the corresponding history row, computes the parking duration in minutes, applies the configured hourly rate, and enforces a minimum charge covering the first fifteen minutes. The resulting amount is written back to the history row and a new payments record is created with status pending.

For Payme and Click, the system constructs a platform-specific redirect URL embedding the payment reference and amount. The URL reaches the driver through a Telegram message, a QR code at the exit barrier, or a link in the operator dashboard. The driver completes payment on their personal device; the platform calls a webhook endpoint which updates the payments row to paid and triggers the gate. Cash payment is also supported: the operator marks the row as cash_paid through the dashboard. Table 2 compares the supported methods.

Method	Integration	Confirmation	Avg. time
Cash	Operator dashboard	Manual approval	< 1 min
Payme	Redirect URL / QR	Webhook callback	1–2 min
Click	Redirect URL / QR	Webhook callback	1–2 min
Telegram	Inline payment button	Webhook callback	1–2 min

Table 2. Supported payment methods and confirmation flows.

5. Advance Booking and No-Show Handling

Drivers may reserve a space through the REST API or by sending a structured message to the Telegram bot. A booking record stores the driver's name, phone number, vehicle

plate, desired space number and expected arrival time. The corresponding spots row is immediately set to reserved so the space is not assigned to a walk-in vehicle.

The booking_checker coroutine wakes every five minutes and queries for active bookings whose arrival_time lies more than two hours in the past. Each such booking is transitioned to no_show: the spots row reverts to free, a penalty amount is written to payments with status pending_penalty, and the driver receives a Telegram notification. This automatic expiry prevents reserved spaces from being held indefinitely [3].

Drivers who arrive within the grace period are processed by the normal entry flow: the camera recognises the plate, the system matches it to an active booking, and the barrier opens. The booking status transitions to completed and the spot to occupied.

6. Security and Access Control

All write endpoints are protected by JSON Web Tokens signed with HS256 [7]. Tokens carry a role claim (superadmin or operator) and expire after eight hours. The superadmin role has unrestricted access; the operator role may process entries and exits but cannot modify pricing or export the full history log. WebSocket connections present their token as a URL query parameter; unauthenticated connections receive read-only occupancy data.

Passwords are stored as bcrypt hashes. Parameterised queries prevent SQL injection throughout. The system stores only the plate string and event timestamps; no images are retained on disk, reducing privacy exposure in line with Uzbekistan's Law on Personal Data Protection (2019) [8].

7. Evaluation

The system was deployed at a 24-space outdoor parking lot in Tashkent for two weeks. During that period 512 vehicle passes were recorded. A separate stress test replayed the two-week event log at 40× real-time speed while 200 WebSocket clients polled the occupancy endpoint every three seconds. Key results:

- API response time remained below 200 ms at the 95th percentile throughout the stress test, with a median of 34 ms.
- No data loss or database corruption occurred across three simulated abrupt power-failure interruptions. WAL recovery restored the database to a consistent state in all three cases.
- The booking_checker correctly identified and expired all 11 no-show bookings created during the trial; no false positives were recorded.
- Payment webhooks from Payme and Click were acknowledged within 400 ms in all test calls, satisfying the platform's 30-second timeout requirement.

8. Conclusion

The data-layer and payment architecture described in this paper shows that a four-table WAL-mode SQLite database combined with link-based integration to Uzbek e-wallet services is sufficient to underpin a production-grade automated parking system on

commodity edge hardware. The schema design separates high-frequency occupancy writes from the historical log, enabling concurrent readers without performance degradation. The advance booking subsystem with automatic no-show expiry reduces manual operator workload, while JWT-based role control and bcrypt password hashing provide an adequate security baseline for small-to-medium facilities.

Future work will focus on migrating the history table to a lightweight time-series store to improve analytical query performance as the dataset grows, and on adding end-to-end encryption to the Telegram notification channel.

REFERENCES

[1] R. E. Barone et al., "Architecture for parking management in smart cities," IET Intell. Transp. Syst., vol. 8, no. 5, pp. 445–452, 2014.

[2] Raspberry Pi Foundation, Raspberry Pi 4 Model B — Product Brief, 2023. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[3] Y. Geng and C. G. Cassandras, "New smart parking system based on resource allocation and reservations," IEEE Trans. Intell. Transp. Syst., vol. 14, no. 3, pp. 1129–1139, 2013.

[4] R. E. Barone et al., "Smart parking: Using a Vanet-based system to make decisions for reducing CO2 emissions," in Proc. IEEE ITSC, 2011, pp. 1236–1241.

[5] D. R. Hipp, "SQLite, a self-contained, serverless, zero-configuration, transactional SQL database engine," SQLite Consortium, 2022. [Online]. Available: https://www.sqlite.org/wal.html

[6] A. Thakur et al., "Smart parking system for smart cities using IoT and mobile application," in ICT Analysis and Applications, Springer, 2020, pp. 319–327.

[7] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015.

[8] Republic of Uzbekistan, "Law on Personal Data Protection," 2019.