# TECHNICAL CASE STUDY: POCKETBASE ARCHITECTURE

An Analysis of Single-File Backend Systems, Operational Constraints, and Integration Strategy

| | |
|---|---|
| **Subject:** | PocketBase Open-Source Backend Engine |
| **Analysis Date:** | July 2026 |
| **Focus Areas:** | Embedded SQLite Concurrency, Extensibility via JavaScript (Goja), Next.js Architectural Synergy |

## 1. Executive Summary

Modern web architecture heavily leverages distributed systems, abstracting backend functionality into decoupled infrastructure components (e.g., managed databases, external identity providers, and object storage endpoints). While this mitigates point-of-failure vulnerabilities for high-scale enterprise systems, it induces significant configuration complexity and infrastructural overhead for early-stage applications and Minimum Viable Products (MVPs).

PocketBase addresses this complexity by consolidating core backend requirements—Realtime Database, User Authentication, File Storage, and an Administrative UI—into a **single executable Go binary** utilizing an embedded SQLite engine. This case study evaluates the underlying architecture, documents non-apparent runtime constraints, and outlines technical pros and cons for software engineers.

## 2. Architectural Foundations

PocketBase combines a low-footprint, compiled Go application layer with an embedded transactional database. Unlike standard Backend-as-a-Service (BaaS) architectures such as Firebase or Supabase—which require independent container networks or orchestrators to link separate API servers, web consoles, and PostgreSQL instances—PocketBase operates entirely inside a single operating system process memory space.

### Core Infrastructure Pillars

- **Embedded Realtime Storage:** Leveraging SQLite via WAL (Write-Ahead Logging) mode, database transactions are executed via local file-system read/write calls, bypassing TCP network handshakes entirely. Realtime events are handled via standard Server-Sent Events (SSE).
- **Unified Authentication:** Built-in management for authentication protocols including email/password registration, token verification, and OAuth2 provider integration (Google, GitHub, GitLab, OIDC).
- **Local & Remote Asset Management:** Files are metadata-mapped directly inside database collections. Binary files can be written to the local disk or transparently proxied to an upstream S3-compatible bucket via built-in configuration drivers.
- **Declarative Security:** Access control is governed by granular, collection-level API Rules evaluated directly by the engine using string-based evaluation (e.g., checking client properties via token state metadata: `@request.auth.id != ""`).

## 3. The Extensibility Engine & Runtime Limitations

While native performance is achieved by compiling custom modules directly into the Go source code, PocketBase natively provides an internal **JavaScript/TypeScript extension mechanism** via the `pb_hooks` engine. This engine exposes over 80 system event hooks, allowing scripts to intercept requests, validate payloads, and execute arbitrary business logic before or after operations (e.g., `onRecordCreate`, `onBootstrap`).

> **Critical Architectural Caveat (Goja Interpreter):** The PocketBase JavaScript runtime does **not** operate on Node.js, V8, or Bun. Instead, it embeds `Goja`, an ECMAScript 5.1/6 pure Go interpreter. Consequently, standard npm packages requiring native Node modules (such as `fs`, `crypto`, `net`, or HTTP-client packages dependent on internal Node runtimes) will fail to execute. Developers must use PocketBase's globally injected Go wrapper methods (e.g., `$http.send()`, `$os.readFile()`) to interface with the host environment.

## 4. Production Hardening & Operational Constraints

Moving a single-file architecture into production introduces specific structural limitations that contrast sharply with traditional cloud-native strategies.

### SQLite Write Serialization & Concurrency Bottlenecks

SQLite utilizes file-level locking mechanisms. Although PocketBase enhances write throughput by enabling Write-Ahead Logging (WAL) mode—allowing concurrent reads while a write operation is actively committing—**write transactions remain strictly serialized** at the database level. For high-throughput transaction flows (e.g., high-frequency ingestion, metrics monitoring, continuous multi-user synchronization), write-locks will cause query queueing and eventual time-out errors.

### Filesystem Ephemerality & Hosting Constraints

Because state is tied to the physical `data.db` file on disk, deployment platforms with ephemeral filesystems (e.g., standard Heroku Dynos, base instances on Render, or serverless edge runtimes) are architecturally incompatible. Every code deployment or container recycling cycle resets the local block storage, destroying the underlying database file. PocketBase requires **persistent block storage** or hosting on a dedicated Virtual Private Server (VPS), introducing a requirement for local automated cron-based backup pipelines.

### Horizontal Scaling Limitations

PocketBase cannot be horizontally scaled by deploying multiple stateless instances behind a standard Round-Robin load balancer. Because instances do not share an out-of-process distributed caching or database layer, scaling must be handled **vertically** (allocating higher CPU/RAM compute resources to a single host node).

## 5. Next.js Integration Mapping

When integrating PocketBase into modern frontend frameworks like Next.js (specifically under the Pages Router model), the architecture differs significantly from standard client-to-database architectures:

- **Client-Side Interaction:** Initializing the PocketBase JS SDK on the browser enables real-time subscriptions and low-latency reads directly out of the SQLite engine, maximizing efficiency.

- **Server-Side Rendering (SSR) Double-Hop Penalty:** When fetching data inside server-side contexts (e.g., `getServerSideProps` or internal API routes), queries must execute an additional network hop: `Client → Next.js Server Node → PocketBase Instance`.
- **API Gateway Rate Limiting:** Since server-side calls originate from a concentrated server IP address, they can inadvertently trigger PocketBase's built-in anti-abuse rate limiters unless explicitly bypassed or configured in the system panel.

## 6. Vercel Incompatibility & Decoupled Deployment Model

A frequent structural misconception is the attempt to run a PocketBase binary directly on stateless serverless hosting providers like **Vercel**. This is technically impossible due to two structural friction points:

> **Vercel Structural Incompatibilities:**
> - **Stateless Compute Architecture:** Vercel executes code via ephemeral serverless edge components and cloud functions. Because these environments possess no fixed disk memory, local writes to the SQLite `data.db` file are discarded instantly when the runtime context scales down or redeploys.
> - **Execution Lifecycle Restrictions:** Serverless platforms do not permit long-running background daemons. A continuous, compiled Go binary listening continuously on a specific network port (e.g., `./pocketbase serve`) cannot maintain an active process footprint inside a standard short-lived invocation model.

### The Recommended Decoupled Pipeline

To combine the deployment performance of Vercel for frontends with the low-overhead simplicity of PocketBase, developers must leverage a decoupled topology:

1. **Frontend Layer (Vercel):** Code assets containing the Next.js application are built and pushed directly to Vercel's global CDN network.
2. **Backend Layer (Persistent VPS or Fly.io):** The PocketBase binary is executed on persistent infrastructure (such as a cheap Virtual Private Server or a container instance with attached volume block storage via Fly.io/ Railway).

### JavaScript Initialization Matrix (Next.js Pages Router Setup)

To facilitate authenticated operations across this decoupled boundary, initialize the PocketBase client using a persistent environment reference:

```
// lib/pocketbase.js
import PocketBase from 'pocketbase';

// Resolves to the persistent backend deployment URL configured in Vercel settings
const pb = new PocketBase(process.env.NEXT_PUBLIC_POCKETBASE_URL || 'http://127.0.0.1:8090');

export default pb;
```

## 7. Comprehensive Evaluation Matrix

### Architectural Advantages

- **Infrastructural Economy:** Drastically reduces operational cost; a complete suite runs comfortably on a low-tier $4/month VPS.
- **Zero Network Database Latency:** In-process data reading bypasses internal VPC network roundtrips completely.
- **Trivial Local Portability:** Local database configuration is identical to production; backing up requires a simple copy of the `pb_data` folder.
- **Native Embedded Tooling:** Shipped with standard, production-grade logging, automated migrations, and job scheduling.

### Technical Risks & Limitations

- **Single Point of Failure:** Loss of host VPS availability or storage degradation takes down the entire application layer.
- **Single-Maintainer Venture:** Built and curated primarily by one developer (Gani Georgiev); no enterprise-grade support SLAs.
- **Pre-v1.0 Lifecycle Status:** Currently pre-v1.0.0, introducing risks of breaking API shifts and manual schema migration adjustments.
- **Non-Standard JS Runtime:** Goja-based hook execution prevents direct usage of standard Node/npm package dependencies.

## 8. Reference Directory


**[1] Base Video Reference:** Better Stack YouTube Channel. *"This Free Go Alternative to Firebase is Just One File."* Available at: https://youtu.be/WZoC1HA1vec

**[2] Official Project Repository:** PocketBase Core Go Source. Available at: https://github.com/pocketbase/pocketbase

**[3] Embedded JavaScript Interpreter:** Goja ECMA5.1+/6 Go Runtime Engine. Available at: https://github.com/dop251/goja

**[4] Database Specification:** SQLite Documentation on Write-Ahead Logging (WAL Mode). Available at: https://www.sqlite.org/wal.html