

HIGH-LEVEL DESIGN DOCUMENT

Laravel Automotive Management System on Kubernetes

Project Title	Laravel K8s Automotive Management System
Document Type	High-Level Design (HLD)
Version	1.0
Project Type	DevOPs
Status	Final Draft

1. Introduction

1.1 What is This Document?

This is a High-Level Design (HLD) document — think of it as the "blueprint" or "map" of the project before we start writing code. Just like an architect draws a floor plan before building a house, software engineers write an HLD before building a system.

This document explains how the Laravel Automotive Management System is structured, what technologies it uses, how it is deployed on Kubernetes, and why each decision was made. Even if you are new to these technologies, this document will guide you step by step.

Simple Explanation

An HLD answers three core questions: *WHAT* are we building? *HOW* is it structured? *WHY* did we make these choices?

1.2 What Are We Building?

We are building a web-based Automotive Management System — a software tool that helps a car dealership or automotive service centre manage its day-to-day operations. Instead of using paper records or spreadsheets, this system provides a clean web interface where staff can:

- Add, update, and search vehicle inventory
- Manage customer records and contact information
- Book and track vehicle service appointments
- Generate reports on sales and service activity
- Control who can access which parts of the system (admin, sales, technician roles)

1.3 Key Terms — Plain English Glossary

Before diving in, here is a quick-reference glossary. Do not worry if these terms are unfamiliar — they will make more sense as you read on.

Term	Short Meaning	Real-World Analogy
Laravel	A PHP framework to build web apps quickly	Like a ready-made toolbox for building websites
Kubernetes (K8s)	A system that runs and manages containers	Like a smart manager who keeps all your workers (containers) running
Docker	A tool to package an app into a portable container	Like a shipping container — same box works anywhere
Container	A lightweight, isolated package with your app inside	Like a lunchbox — self-contained, portable
Pod	The smallest unit Kubernetes manages (holds containers)	Like a desk in an office — one or more workers share it
Nginx	A fast web server / reverse proxy	Like a receptionist — directs incoming visitors (requests)

MySQL	A relational database for structured data	Like a very organised filing cabinet
Redis	An in-memory data store for fast caching	Like a sticky-note pad — quick to read/write, temporary
MVC	Model-View-Controller architecture pattern	Model = data, View = what user sees, Controller = logic in between
HLD	High-Level Design — big-picture architecture doc	Like a city map — shows the layout, not every doorknob
PVC	PersistentVolumeClaim — permanent disk storage in K8s	Like a filing cabinet that survives office moves
Ingress	Kubernetes resource that manages external web traffic	Like a building's main entrance and reception desk
ConfigMap	Kubernetes object holding non-secret configuration	Like a settings file pinned to a noticeboard
Secret	Kubernetes object holding sensitive data (passwords)	Like a locked safe — only authorised pods get access
HPA	Horizontal Pod Autoscaler — auto-add or remove pods	Like hiring extra staff during a busy season automatically

2. System Overview

2.1 Why Did We Build This?

Most small automotive businesses still rely on spreadsheets or paper-based records to manage vehicles and customer data. This is error-prone, slow, and hard to scale. Our project solves this by providing a modern, web-based management system.

Beyond just building a web app, we went one step further: we deployed it on Kubernetes. This means the system can automatically recover from crashes, handle more users during busy periods, and be updated without any downtime — just like large-scale production systems used by real companies.

Why Kubernetes?

Traditional deployments break if the server crashes. Kubernetes automatically restarts failed components, distributes load across multiple copies, and lets you update the app without users ever seeing a downtime page.

2.2 Project Goals

- Build a working full-stack web application using Laravel (a popular PHP framework).
- Learn and apply Docker containerization — packaging each part of the app into its own container.
- Deploy on Kubernetes — a real-world industry-standard container orchestration platform.
- Implement auto-scaling so the app can handle 10 users or 1000 users without manual effort.
- Keep sensitive data (passwords, API keys) secure using Kubernetes Secrets.
- Follow DevOps best practices used in real software engineering jobs.

2.3 What the System Can Do — Feature List

Feature	Description
Vehicle Inventory	Add, edit, delete, and search vehicles by make, model, year, price, or status
Customer Management	Store customer profiles, contact info, and link them to purchases/service history
Service Bookings	Schedule, view, and update service appointments with status tracking
Invoicing	Generate and track invoices and payment status per customer
User Roles & Access	Admin, Sales, and Technician roles with different access permissions
Reports & Dashboard	Visual dashboards showing sales totals, bookings, and inventory summaries
Audit Logs	Track who made what changes and when — important for accountability

3. Architecture Overview

3.1 What is the "Architecture" of a System?

Think of architecture as the structural design of a building. Before construction begins, an architect decides: how many floors, where the rooms go, how the plumbing connects, and where the exits are. Software architecture works the same way — it defines the major components of the system and how they connect to each other.

Our Architecture in One Sentence

The user's browser talks to Nginx (the doorman), which passes requests to Laravel (the brain), which reads/writes data from MySQL (the filing cabinet) and uses Redis (the sticky notes) for speed — and Kubernetes keeps everything alive automatically.

3.2 The Three-Tier Design

Our system is organised into three distinct tiers (layers). Each tier has a clear, single responsibility:

- Tier 1 — Presentation (What the user sees): The browser renders HTML pages generated by Laravel's Blade templating engine. Users interact with forms, tables, and dashboards.
- Tier 2 — Application (The brain): Laravel handles all the business logic — validating input, querying the database, enforcing rules (e.g., only admins can delete vehicles), and returning the right data.
- Tier 3 — Data (The memory): MySQL stores all permanent data (vehicles, customers, bookings). Redis stores temporary data for speed (user sessions, cached queries).

3.3 Full Layer Breakdown

Layer	What It Does	Technology Used	Analogy
User's Browser	Shows the web interface to the user	HTML, CSS, JavaScript	Customer at the counter
Ingress	Receives all traffic, routes it correctly	Nginx Ingress Controller	Building entrance / receptionist
Web Server	Serves static files (images, CSS, JS)	Nginx Container	Filing clerk
Application	Processes requests, runs business logic	Laravel on PHP-FPM Container	The brain / manager
Queue Worker	Handles background jobs (emails, reports)	Laravel Artisan Worker Container	Back-office staff
Cache & Sessions	Speeds up repeated operations	Redis Container	Sticky-note pad on the desk
Database	Stores all permanent data	MySQL Container + PVC	The filing cabinet
Orchestration	Keeps everything running, scales automatically	Kubernetes Cluster	The building manager

Image Registry	Stores Docker images (app packages)	Docker Hub	Software warehouse / library
-----------------------	-------------------------------------	------------	------------------------------

3.4 How a Request Flows Through the System

Here is what happens step-by-step when a user logs in and views the vehicle list:

- Step 1: User opens a browser and types the app URL. The browser sends an HTTP request to the internet.
- Step 2: The Nginx Ingress Controller receives the request and checks the routing rules.
- Step 3: Nginx routes the request to the Laravel PHP-FPM container inside the Kubernetes cluster.
- Step 4: Laravel checks if the user is authenticated (session stored in Redis).
- Step 5: Laravel queries MySQL for the list of vehicles using its Eloquent ORM.
- Step 6: If the query was recently run, Redis cache serves the result instantly (no DB hit).
- Step 7: Laravel passes the data to a Blade template, which renders an HTML page.
- Step 8: The HTML response travels back through Nginx to the user's browser.
- Step 9: The user sees the vehicle listing page in their browser.

4. Component Design

This section explains each major component (building block) of the system in plain English, followed by the technical details.

4.1 Laravel Application — The Brain

Laravel is the core of the entire application. It is a PHP framework that follows the Model-View-Controller (MVC) pattern, which means:

- Model: Represents data and interacts with the MySQL database (e.g., a Vehicle model fetches vehicle rows).
- View: The Blade template files that produce the HTML the user sees in their browser.
- Controller: Receives a request, asks the Model for data, and passes it to the View to render.

Think of It Like a Restaurant

The Controller is the waiter (takes orders), the Model is the kitchen (prepares the food / fetches data), and the View is the plate that reaches the customer (what users see).

In our deployment, Laravel runs inside a Docker container based on the php:8.2-fpm-alpine image. Configuration values (like database host, app URL) are injected at startup from Kubernetes ConfigMaps and Secrets — meaning we never hardcode passwords into the code.

4.2 Nginx — The Doorman

Nginx is a lightweight, high-performance web server. In our setup, it has two jobs:

- Serve static files (images, CSS, JavaScript) directly without bothering Laravel — this is much faster.
- Forward all PHP requests (any URL that needs dynamic content) to the Laravel PHP-FPM container.

Why Separate Nginx and Laravel?

If Laravel had to serve every image and CSS file, it would slow down. By having Nginx serve those files directly, Laravel only handles the smart work — like a team where one person handles admin tasks so the expert can focus on expert work.

4.3 MySQL — The Filing Cabinet

MySQL is our relational database — it stores all the structured data in organised tables with rows and columns (just like a spreadsheet, but much more powerful and reliable).

- Database name: automotive_db
- Key tables: vehicles, customers, service_bookings, invoices, users, audit_logs
- All database tables are created and managed by Laravel Migrations — meaning the database structure is version-controlled just like code.
- MySQL data is stored on a Kubernetes PersistentVolumeClaim (PVC) — a dedicated disk that survives even if the MySQL container restarts or crashes.

□ What is a PVC?

Normally, when a container restarts, all its data is wiped. A PersistentVolumeClaim is like plugging in an external hard drive — the data lives on the drive, not inside the container, so it survives restarts.

4.4 Redis — The Sticky Note Pad

Redis is an in-memory data store — meaning it keeps data in RAM (extremely fast) rather than on disk. We use it for three purposes:

- Session Storage: When a user logs in, their session data is stored in Redis instead of files. This allows multiple Laravel pods to share session data without conflicts.
- Caching: Frequently accessed data (like the vehicle list) is stored in Redis so we avoid hitting MySQL every single time. Redis can serve a cached result in microseconds.
- Queue Backend: When Laravel needs to do a slow task (like sending an email or generating a report), it adds a job to a Redis queue instead of making the user wait.

4.5 Queue Worker — The Back-Office Team

A separate Kubernetes Deployment runs Laravel queue workers. These workers watch the Redis queue and process background jobs one by one. They run inside the same Docker image as the main Laravel app, but with a different startup command: `php artisan queue:work`.

□ Why Separate Workers?

Imagine a restaurant where waiters also had to cook every meal themselves — they would be overwhelmed. Queue workers are like dedicated kitchen staff: they handle the heavy lifting in the background so web requests remain fast for users.

5. Kubernetes Resource Design

This section explains how the application is organised inside the Kubernetes cluster. If you are new to Kubernetes, think of it as a very smart operating system for running containers across multiple computers.

5.1 Namespace — Our Dedicated Space

All resources for this project live inside a Kubernetes Namespace called `automotive-system`. A namespace is like a folder on a computer — it groups related resources together and keeps them isolated from other projects on the same cluster.

5.2 Deployments — Running Our Containers

A Kubernetes Deployment tells Kubernetes: "Run X copies of this container, and if one crashes, restart it automatically." Here are all the Deployments in our project:

Resource	Type	Replicas	What It Does
<code>laravel-app</code>	Deployment	2 (min)	Runs the main Laravel PHP-FPM web app
<code>nginx-web</code>	Deployment	2 (min)	Runs the Nginx web server / reverse proxy
<code>queue-worker</code>	Deployment	1	Runs Laravel background job workers
<code>mysql</code>	StatefulSet	1	Runs MySQL database with permanent storage
<code>redis</code>	Deployment	1	Runs Redis cache and queue backend
<code>laravel-scheduler</code>	CronJob	On schedule	Runs Laravel scheduled tasks (e.g., reports)
<code>laravel-service</code>	Service	N/A	Internal network route to the Laravel pods
<code>mysql-service</code>	Service	N/A	Internal network route to the MySQL pod
<code>redis-service</code>	Service	N/A	Internal network route to the Redis pod
<code>automotive-ingress</code>	Ingress	N/A	External HTTPS access point for the whole app
<code>laravel-config</code>	ConfigMap	N/A	Stores non-secret settings (APP_URL, DB_HOST)
<code>laravel-secret</code>	Secret	N/A	Stores passwords and APP_KEY (encrypted)
<code>mysql-pvc</code>	PVC	10Gi	Permanent disk storage for MySQL data
<code>laravel-storage-pvc</code>	PVC	5Gi	Permanent disk for uploaded files / logs

laravel-hpa	HPA	2–10 pods	Auto-scales Laravel pods based on CPU usage
-------------	-----	-----------	---

5.3 Auto-Scaling — Handling Busy Days

The Horizontal Pod Autoscaler (HPA) watches the CPU usage of the Laravel pods. If average CPU goes above 70%, it automatically adds more pods (up to 10). When traffic drops, it removes extra pods to save resources.

□ Real-World Analogy

Imagine a bank that opens more cashier windows when queues get long, and closes them again when the branch is quiet — automatically, without a manager needing to intervene. That is exactly what HPA does for our app.

6. Data Architecture

6.1 What Data Does the System Store?

The system stores all of its data in a MySQL relational database. Below is a summary of the key tables and what they contain:

Table Name	Key Columns	What It Stores
users	id, name, email, role, password	System users (admin, sales, technician)
vehicles	id, make, model, year, vin, price, status	Car inventory — every vehicle in the dealership
customers	id, name, email, phone, address	Customer profiles and contact details
service_bookings	id, customer_id, vehicle_id, date, status	Appointment bookings for servicing
invoices	id, customer_id, amount, status, issued_at	Bills issued to customers, payment tracking
audit_logs	id, user_id, action, entity, created_at	Record of every significant action taken by users

6.2 How Is Data Kept Safe from Loss?

By default, data inside a Docker container disappears when the container is deleted or restarted. To prevent this, we use Kubernetes PersistentVolumeClaims (PVCs) — dedicated disk volumes that exist independently of containers.

- mysql-pvc (10Gi): Attached to the MySQL container. All database records survive container restarts.
- laravel-storage-pvc (5Gi): Attached to the Laravel container at /var/www/html/storage. User-uploaded files, logs, and compiled views are stored here permanently.

□ Think of It This Way

A PVC is like a USB hard drive plugged into your container. Even if you throw the container away and plug in a new one, the hard drive (and all your data) is still there.

7. Security Design

Security might sound complex, but it boils down to one principle: only the right people and services should be able to access the right things. Here is how we enforce that:

7.1 No Hardcoded Passwords

One of the most common beginner mistakes is hardcoding passwords and API keys directly into code files. If someone ever reads your code (e.g., on GitHub), they would have access to your database.

We avoid this entirely by storing all sensitive values — the database password, Laravel APP_KEY, and Redis auth token — inside Kubernetes Secrets. These are injected into containers as environment variables at runtime, so they never appear in our source code or Docker images.

Kubernetes Secret vs ConfigMap

ConfigMap = non-sensitive settings (like the app URL or database host). Secret = sensitive data (like passwords). Secrets are stored encoded and only shared with pods that specifically request them.

7.2 Network Isolation

- Only the Nginx Ingress is exposed to the internet — everything else is completely private inside the cluster.
- MySQL and Redis are configured as ClusterIP Services — they have no external IP address and cannot be reached from outside the cluster.
- Kubernetes NetworkPolicies ensure that only the Laravel pod can talk to MySQL and Redis. Other pods cannot access these services at all.

7.3 Application-Level Security (Built into Laravel)

Security Feature	What It Does & Why It Matters
CSRF Protection	Prevents attackers from tricking a logged-in user into submitting a form on their behalf (Cross-Site Request Forgery). Laravel adds a hidden token to every form and verifies it.
Password Hashing	User passwords are never stored as plain text. Laravel uses bcrypt (a strong one-way hash) so even if the database is stolen, passwords cannot be read.
Role-Based Access	Each user has a role (admin, sales, technician). Laravel Gates and Policies ensure a technician cannot access billing data, and a salesperson cannot delete system users.
SQL Injection Guard	All database queries go through Laravel's Eloquent ORM, which uses parameterized queries — making SQL injection attacks impossible.
XSS Prevention	Laravel's Blade templates automatically escape output, preventing attackers from injecting malicious JavaScript into pages that other users see.
Input Validation	All user input is validated using Laravel Form Requests before it reaches the business logic or database.

7.4 HTTPS / TLS Encryption

All traffic between the user's browser and the server is encrypted using TLS (HTTPS). The Nginx Ingress Controller handles TLS termination. SSL certificates are managed automatically by cert-manager using the free Let's Encrypt certificate authority. Any user who accidentally visits the HTTP version of the site is automatically redirected to HTTPS.

8. Deployment Architecture

8.1 How the App is Packaged — Docker Images

Each service in the system is packaged into a Docker image. A Docker image is like a snapshot of a perfectly configured computer — it contains the operating system, runtime, dependencies, and application code, all in one portable file.

Image Name	Base Image	What It Contains
laravel-app:latest	php:8.2-fpm-alpine	Laravel source code, PHP extensions, Composer packages
nginx-automotive:latest	nginx:1.25-alpine	Nginx config + compiled static assets from the public/ folder
mysql:8.0	Official MySQL image	Used as-is — no modification needed
redis:7-alpine	Official Redis image	Used as-is — no modification needed

Why Alpine-Based Images?
Alpine Linux is a tiny (5MB) Linux distribution. Using it as the base image makes our Docker images much smaller — faster to download, push, and start. Less size = less attack surface too.

8.2 Step-by-Step Deployment Sequence

When deploying the system to a Kubernetes cluster for the first time, follow this order. Order matters because later steps depend on earlier ones (e.g., Laravel needs MySQL to be ready before it can run database migrations):

- Step 1: Create the Namespace, ConfigMaps, and Secrets — the foundation config layer.
- Step 2: Deploy MySQL (StatefulSet + PVC) and wait until its readiness probe passes.
- Step 3: Deploy Redis and wait until it is ready.
- Step 4: Deploy the Laravel app. An InitContainer automatically runs database migrations (php artisan migrate) before the main app starts.
- Step 5: Deploy Nginx web server.
- Step 6: Deploy the Queue Worker and Scheduler CronJob.
- Step 7: Apply ClusterIP Services for all internal components.
- Step 8: Apply the Ingress resource and update DNS to point to the cluster's external IP.
- Step 9: Apply the Horizontal Pod Autoscaler (HPA) for the Laravel Deployment.

8.3 Zero-Downtime Updates

When you push a new version of the app, Kubernetes uses a Rolling Update strategy. Instead of shutting everything down and restarting, it:

- Starts one new pod with the updated image.
- Waits until the new pod passes its health check (readiness probe).
- Only then removes one old pod.
- Repeats until all pods are running the new version.

Result

Users never see a "site is down for maintenance" page. The update happens invisibly in the background — just like how Netflix deploys updates without you ever noticing.

9. Non-Functional Requirements

Non-functional requirements describe HOW WELL the system performs, not just WHAT it does. Think of them as the quality standards the system must meet.

Quality Area	Requirement	How We Achieve It	Simple Explanation
Availability	99.5% uptime	Multi-replica pods + K8s self-healing	If one copy crashes, another keeps running
Scalability	Handle 500+ concurrent users	HPA scales app pods 2–10 based on CPU	More users = more pods, automatically
Performance	Page load under 2 seconds	Redis caching + Nginx static serving	Serve cached data fast, skip slow DB calls
Security	No hardcoded credentials	All secrets in Kubernetes Secrets	Passwords never in code or images
Maintainability	Config changes w/o rebuild	ConfigMap-driven Nginx and app config	Change settings without rebuilding Docker images
Observability	Monitor health & logs	K8s liveness/readiness probes + log stream	K8s knows if app is sick and can restart it
Portability	Runs on any K8s cluster	Standard YAML manifests, no cloud lock-in	Works on Minikube, GKE, EKS, or any K8s cluster
Recoverability	Recovery within 5 minutes	Pod auto-restart + PVC data persistence	App auto-heals; data survives crashes

10. Technology Stack

Here is every technology used in the project, with the version and a brief note on why it was chosen.

Category	Technology	Version	Why Chosen
Backend Framework	Laravel	v10.x	Most popular PHP framework; great for rapid web app development
PHP Runtime	PHP-FPM	v8.2	High-performance PHP process manager; ideal for containerized deployments
Frontend Templates	Blade / Tailwind	Laravel built-in	Simple, powerful templating; Tailwind provides utility-first CSS
Web Server	Nginx	v1.25	Lightweight, fast, excellent as reverse proxy for PHP-FPM
Database	MySQL	v8.0	Industry-standard relational DB; excellent Laravel compatibility
Cache & Queues	Redis	v7.0	Blazing-fast in-memory store; supports sessions, cache, and job queues
Containerisation	Docker	v24+	Universal container platform; enables consistent environments everywhere
Orchestration	Kubernetes	v1.28+	Industry gold standard for running containers at scale
Ingress Controller	Nginx Ingress	Community	Widely adopted; simple annotation-based routing configuration
TLS Certificates	cert-manager	v1.13	Automates free SSL certificates via Let's Encrypt ACME protocol
Package Manager (PHP)	Composer	v2.x	Standard PHP dependency manager, used to install Laravel and packages
Version Control	Git / GitHub	Latest	Industry standard for source code management and collaboration

11. Assumptions & Constraints

11.1 Assumptions (What We Expect to Be True)

These are conditions we assume to be in place for the system to deploy and run correctly:

- A working Kubernetes cluster is available — either local (Minikube) or cloud-based (GKE, EKS, AKS, etc.).
- kubectl is configured with a valid kubeconfig file for the target cluster.
- Docker is installed and Docker Hub credentials are available for pushing/pulling images.
- The cluster has internet access for pulling Docker images and obtaining Let's Encrypt SSL certificates.
- The deployment machine has sufficient resources: at least 4 CPU cores and 8GB RAM for a local cluster.

11.2 Constraints (Known Limitations)

These are the known limitations of this project, especially within the scope of a final-semester academic submission:

- Single MySQL replica — a production-grade system would use a MySQL Operator or a managed database service (like AWS RDS) for high availability. This is out of scope for this project.
- Local PersistentVolumes are tied to a specific node — a cloud storage class (like EBS or GCP Persistent Disk) would be needed for a truly portable, multi-node production deployment.
- Redis runs without replication — a production system should use Redis Sentinel or Redis Cluster mode. Not required for this project's scale.
- No CI/CD pipeline is included in this submission — deployments are performed manually using kubectl apply commands.
- Monitoring/alerting (e.g., Prometheus + Grafana) is not included, though the architecture is designed to support it.

12. Document Revision History

Version	Author	Change Summary	Status
1.0	Yashit , Yugal , Yashashvi, Yuvraj	Initial HLD — Complete architecture documentation	Final Draft

End of Document

This document is part of a Final Semester Academic Project. Source code is available at <https://github.com/YashEC-19/laravel-k8s-automotive>