

Software Design & Architecture Project: EasyParkPlus Refactoring and Scaling

[Hesham Sakr, Mihai Chindris, Viet Ha Vu]

Quantic School of Business and Technology

Submission Date

October 26, 2025

Abstract

" This document details a software architecture project focused on refactoring and scaling the existing preliminary prototype codebase for EasyParkPlus, a parking lot management company. The project's primary objective was twofold: first, to improve the maintainability and quality of the existing single-facility code by identifying and removing anti-patterns and implementing modern design patterns ; and second, to design a scalable, cloud-native microservices architecture to support the company's expansion to multiple facilities and integrate a new Electric Vehicle (EV) Charging Station Management business activity. The original prototype was successfully refactored using the Factory Pattern and Observer Pattern to decouple business logic from the graphical user interface. A robust architectural plan was then developed using Domain-Driven Design (DDD), resulting in eight Bounded Contexts that directly map to seven independent microservices. The proposed architecture is hybrid, using synchronous REST APIs for real-time operations and an asynchronous message bus (Kafka/MSK) for decoupled workflows, ensuring the system meets business requirements for high availability and future extensibility."

Introduction

The software industry heavily relies on the ability to understand, update, and extend existing codebases. The central focus of this project is the preliminary prototype of the EasyParkPlus Parking Lot Manager, an application initially developed for a single parking lot. EasyParkPlus, now seeking to scale its operations across multiple facilities and introduce EV charging services, engaged our team as software engineering experts to guide this transition. The overarching goal was to transform the preliminary, tightly coupled prototype into a professionally structured, scalable, and maintainable foundation.

Project Phases and Deliverables

This project was executed in three distinct, yet integrated, phases:

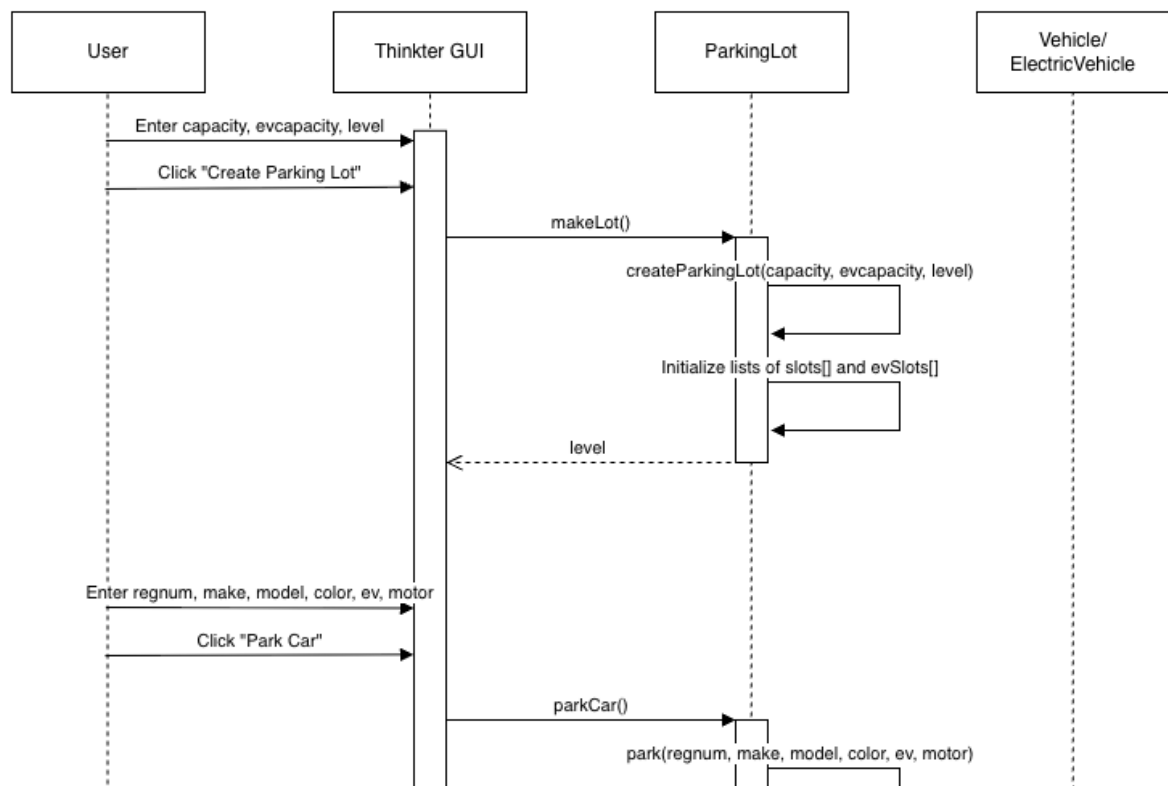
1. **Code Analysis and Refactoring:** This phase involved identifying problematic source code and poor coding practices ("anti-patterns") and removing them, while simultaneously introducing proven solutions via software design patterns. The refactoring provided significant structural and architectural improvements to the system's core vehicle and parking logic.
2. **Domain-Driven Design (DDD) Modeling:** This phase utilized DDD principles to model the expanded system, including the new EV charging capability, across multiple facilities. The outcome was the definition of Bounded Contexts, Ubiquitous Language, and detailed Domain Models.
3. **Microservices Architecture Proposal:** The final phase translated the DDD model into a high-level, microservices-based architecture. This proposal outlines the distinct services, their key responsibilities, the communication endpoints (APIs), and the database strategy to support a cloud-native deployment with the required scalability.

I. Justification for Code Fixes and Chosen Design Patterns

The initial prototype code base suffered from high coupling, global variables, and poor object-oriented design, leading to a God Object anti-pattern in the central ParkingLot class. The refactoring implemented two key design patterns and systematically removed anti-patterns to achieve high maintainability and testability.

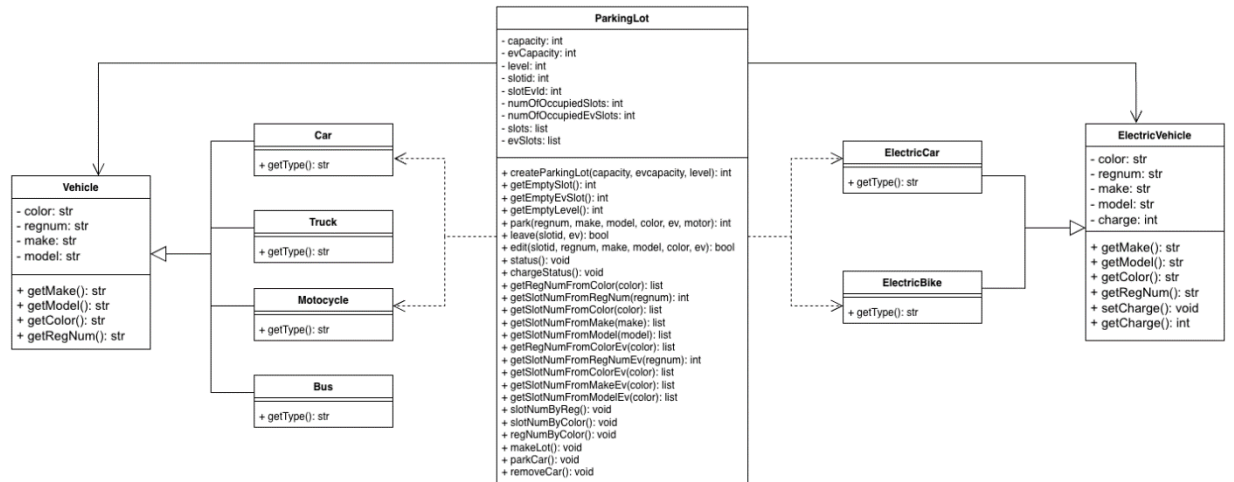
A. Initial design

1. Initial Behavioral UML Diagram



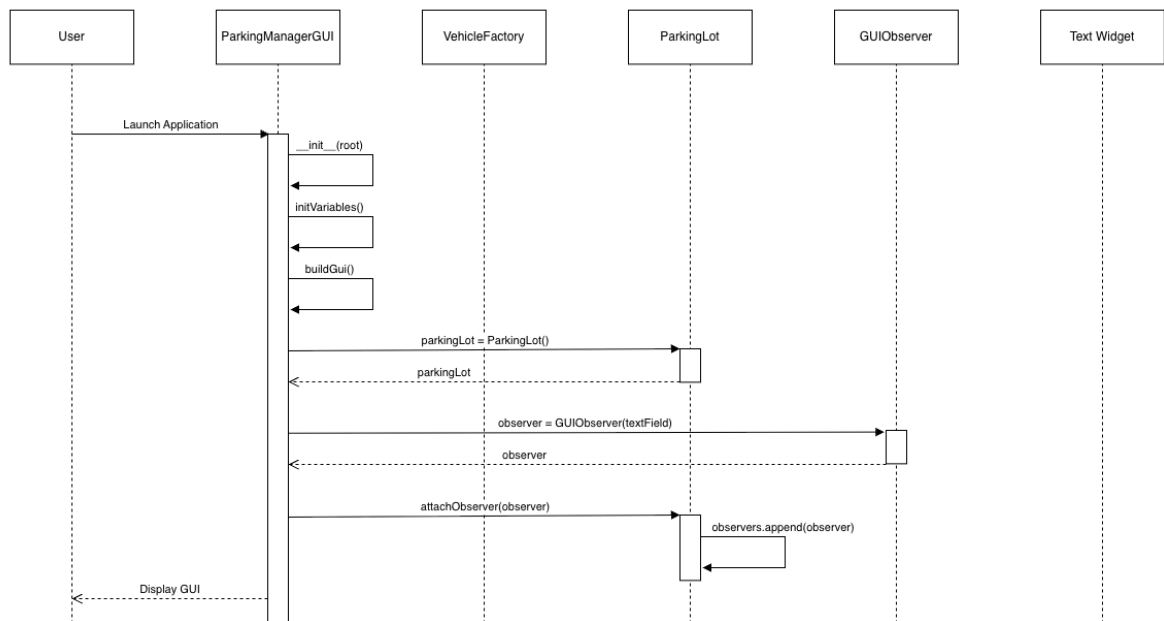
"Refer to the complete diagram: Initial_Behavioral UML Diagram.png"

2. Initial_Structural UML Diagram



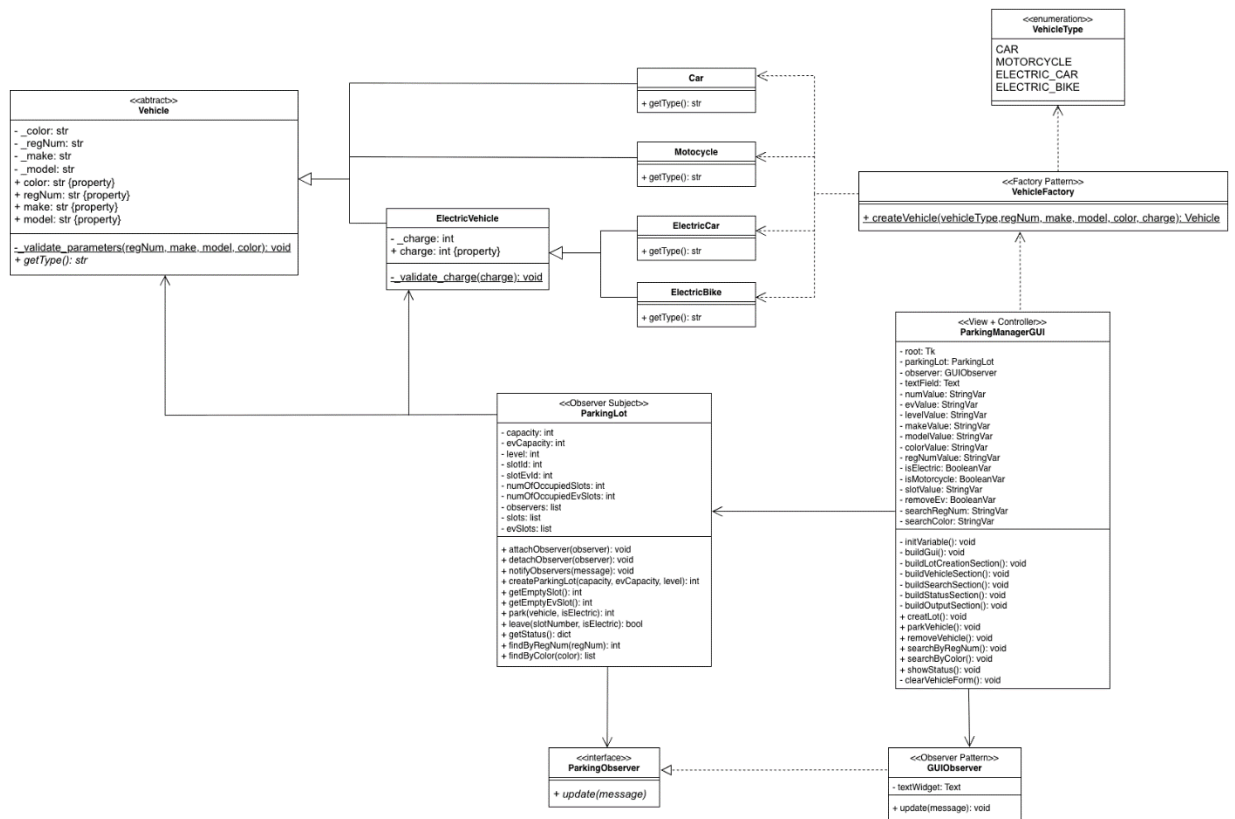
B. Redesign and Modification

1. Redesign Behavioral UML Diagram



"Refer to the complete diagram: Redesigned_Behavioral UML Diagram.png"

2. Redesign Structure UML Diagram



C. Chosen Design Patterns and Justification

We implemented the Factory and Observer patterns to provide significant structural and architectural improvements to the system.

1. Factory Pattern (Creational Pattern)

- Goal: Decouple the client code (the GUI logic) from the specific concrete Vehicle classes it needs to instantiate.
- Implementation: The static `VehicleFactory.createVehicle()` method now handles all instantiation logic (e.g., deciding whether to create a Car, Motorcycle, ElectricCar, or ElectricBike).

- **Justification (Structural Improvement):** This fixes the initial anti-pattern of complex, conditional object creation (nested if statements) spread throughout the `ParkingLot` class. Now, adding a new vehicle type (e.g., a `Scooter`) only requires changes to the Vehicle hierarchy and the `VehicleFactory`, thus adhering to the Open/Closed Principle (OCP).

2. Observer Pattern (Behavioral Pattern)

- **Goal:** Decouple the core business logic (`ParkingLot`) from the presentation layer (Tkinter GUI).
- **Implementation:** `ParkingLot` acts as the Subject, notifying registered Observers (`GUIObserver`) whenever a significant event occurs (e.g., `VehicleCheckedIn`, `VehicleCheckedOut`). The `GUIObserver` then handles the display update.
- **Justification (Architectural Improvement):** This directly addresses the God Object / Tight Coupling anti-pattern. The core business logic (`ParkingLot.park()`) no longer needs to know about or reference the GUI's text field (`tfield.insert`), making the `ParkingLot` class reusable, testable, and focused only on parking operations (Single Responsibility Principle - SRP).

D. Anti-Pattern Removal and Code Fixes

We identified and removed several poor coding practices and design choices that would have weakened the source code.

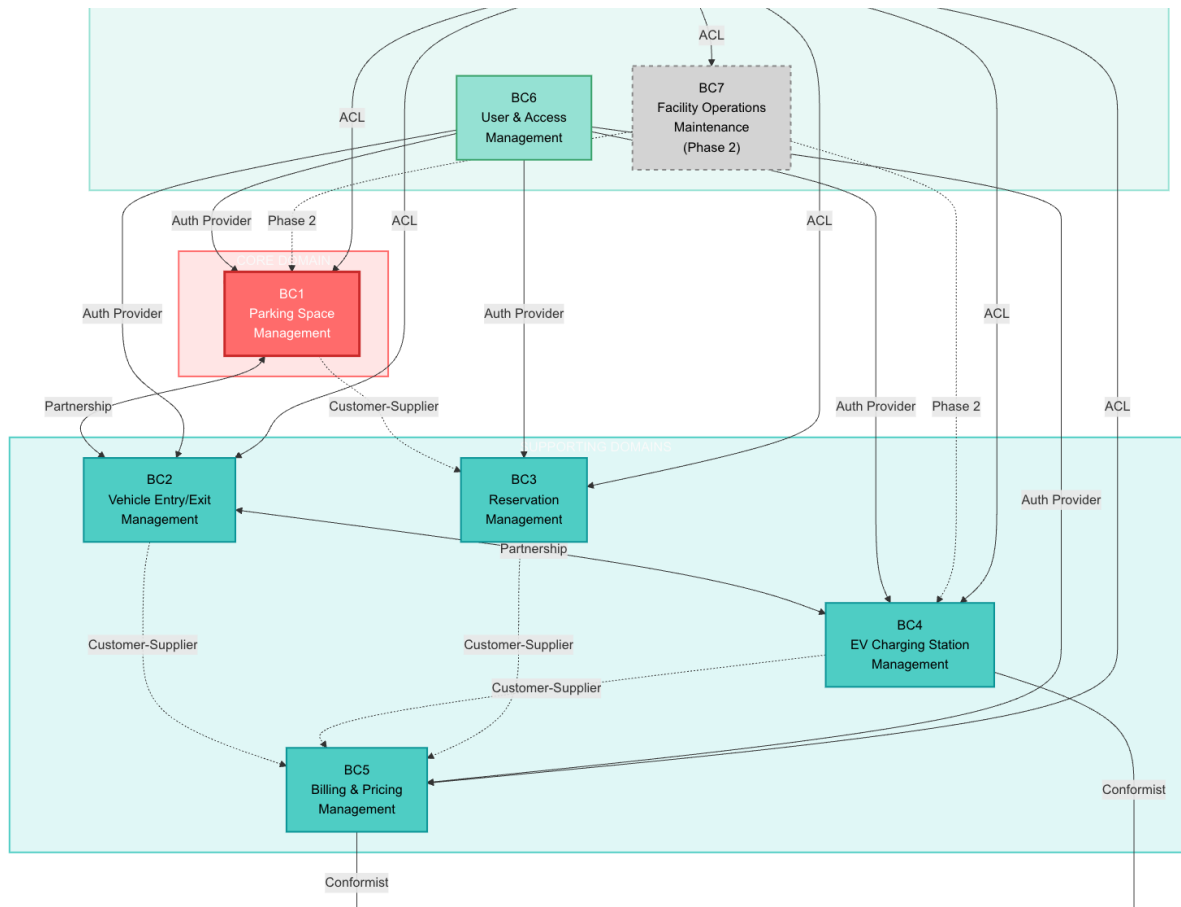
Anti-Pattern Encountered	Structural Fix Implemented	Principle Enforced
Global Variables	All GUI variables moved into the <code>ParkingManagerGUI</code> class as instance attributes (<code>self.numValue</code>).	Encapsulation / SRP (makes code testable)

God Object / Tight Coupling	Code separated into three distinct modules: Vehicle.py, ParkingLot.py, and ParkingManager.py (GUI).	SRP (Each module has one reason to change)
Improper Inheritance	Corrected Python inheritance using class ElectricCar(ElectricVehicle): and the super().__init__() call.	Liskov Substitution Principle (LSP)
Lack of Abstraction	Introduced Vehicle as an Abstract Base Class (ABC) with the abstract method getType().	OCP / DIP (Allows the system to depend on abstractions)
Magic Numbers/Flags	Replaced integer flags (ev=1, motor=0) with explicit Pythonic Booleans (isElectric=True) and sentinel values (-1) with None.	Readability / Clarity
Unnecessary Getters	Replaced Java-style getMake() methods with Pythonic @property decorators (@property def make).	Pythonic Style

II.DDD Modeling: Bounded Contexts and Basic Domain Models


To handle the scaling requirement and the new EV charging feature for EasyParkPlus , we utilized Domain-Driven Design (DDD) to model the system and develop a microservices architecture.

A. High-Level Bounded Context Diagram



The system is decomposed into eight distinct Bounded Contexts (BCs) that align with core business capabilities. The Core Domain is Parking Space Management (BC1).Key

Relationships:Partnership: PSM \leftrightarrow VEM (Parking Space \leftrightarrow Vehicle Entry/Exit) and VEM \leftrightarrow EVC (Entry/Exit \leftrightarrow EV Charging) require tight, coordinated communication for real-time

operations.Customer-Supplier: Core contexts supply data to Billing (e.g., VEM  BPM supplies session duration).Conformist: EVC conforms to the external OCPP 2.0.1 Protocol, and BPM conforms to the Stripe Payment Gateway API.Anti-Corruption Layer (ACL): Analytics (BC8) uses an ACL to consume data from all operational contexts without corrupting its own data model.

B. Basic Domain Models (Aggregates)

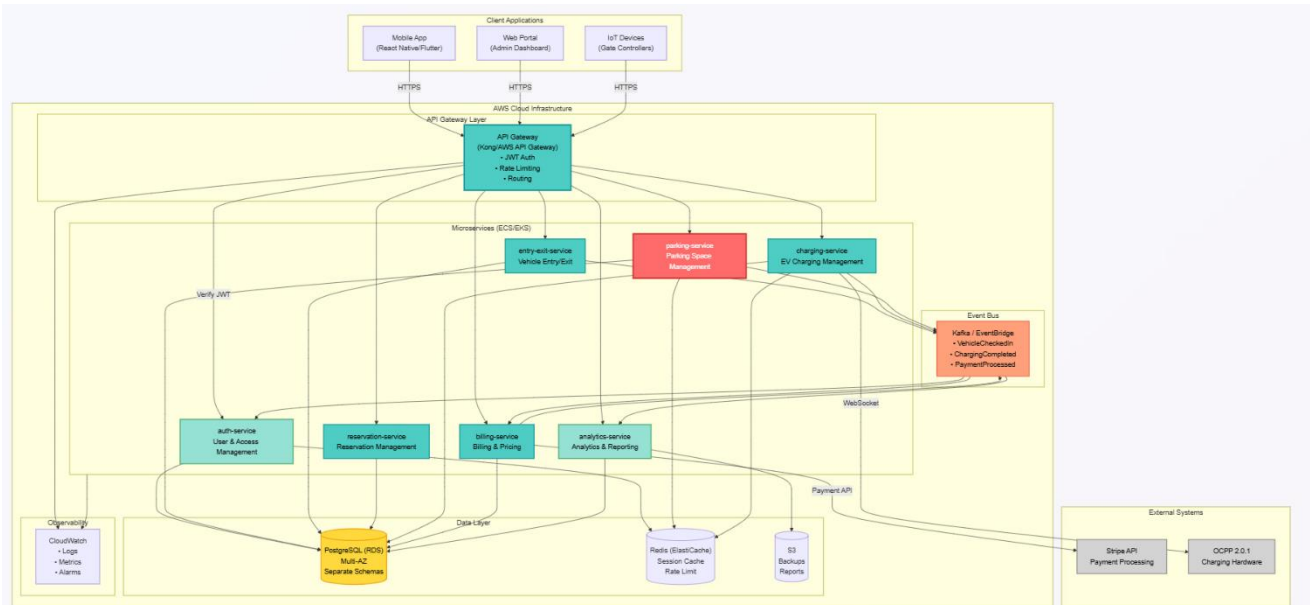
Each Bounded Context (BC) defines key Aggregate Roots (the transactional consistency boundary) and Entities / Value Objects (VOs) using the Ubiquitous Language.

Bounded Context	Core Aggregate Root	Key Entities/Value Objects	Key Business Rule Incorporated
BC1: Parking Space Management	ParkingFacility	ParkingSpace, SpaceType, OccupancyStatus, PowerCapacity (VO)	Facility tracks peakLoadCapacity (350-400 kW) for EV load balancing.
BC2: Vehicle Entry/Exit	ParkingSession	Vehicle, RegistrationNumber (VO), SessionDuration (VO)	Session duration is calculated from entryTime to exitTime.
BC4: EV Charging Station Mgmt	ChargingSession	ChargingStation, EnergyAmount (VO), IdleTime (VO)	Billing is based on energyDelivered + sessionFee + idleFee (if overstayed).
BC5: Billing & Pricing Mgmt	Invoice	PricingPolicy, LineItem, ParkingRate	MVP Rule: dynamicPricingEnabled is FALSE; rates include hourlyRate and dailyCap.
BC6: User & Access Mgmt	User	UserProfile, Subscription, UserRole (VO)	Subscription.crossFacilityEnabled is TRUE, ensuring passes work across all sites.

III. Microservices Architecture: Diagram, Services, and DBs

The Bounded Contexts are mapped directly to microservices to achieve the required scalability. The architecture is hybrid, utilizing REST for real-time actions and Kafka (Amazon MSK) for asynchronous event-driven workflows.

A. Proposed Microservices Architecture Diagram



B. Microservices and Key Responsibilities

Service (Microservice Name)	Bounded Context	Key Responsibilities	Per-Service DB
auth-service	BC6: User & Access Mgmt	User authentication, JWT issuance, subscription management, vehicle registration.	auth_schema in RDS PostgreSQL
parking-service	BC1: Parking Space Mgmt	Facility management, space availability (real-time), IoT sensor data handling.	parking_schema in RDS PostgreSQL
entry-exit-service	BC2: Vehicle Entry/Exit	Session tracking, check-in/out logic, session duration calculation, gate control.	entry_exit_schema in RDS PostgreSQL
charging-service	BC4: EV Charging Station Mgmt	OCPP 2.0.1 communication, load balancing (400kW cap), energy metering, idle time tracking.	charging_schema in RDS PostgreSQL

billing-service	BC5: Billing & Pricing Mgmt	Invoice generation, fee calculation (parking/charging), payment processing via Stripe.	billing_schema in RDS PostgreSQL
reservation-service	BC3: Reservation Mgmt	Reservation creation, cancellation, space allocation, grace period enforcement.	reservation_schema in RDS PostgreSQL
analytics-service	BC8: Analytics & Reporting	Consolidates data from all contexts, calculates KPIs (70% occupancy), generates reports/dashboards.	analytics_schema in RDS PostgreSQL (Optimized for OLAP)

C. APIs/Endpoints (External Facing and Service-to-Service)

1. External-Facing Endpoints (Via AWS API Gateway)

These endpoints primarily serve the Mobile/Web Client.

Service	Endpoint Example (REST)	Purpose
auth-service	POST /auth/login	Issues JWT for authenticated user.
parking-service	GET /facilities/{id}/availability	Real-time space occupancy status.
entry-exit-service	POST /sessions/checkin	Starts a ParkingSession.
charging-service	POST /chargers/{id}/start	Initiates a ChargingSession via the app.
billing-service	GET /pricing	Retrieves the current static PricingPolicy.

2. Internal Service-to-Service Communication

The architecture uses a Hybrid Approach:

Communication Type	Services Involved	Example Workflow	Rationale
Synchronous (REST)	VEM → PSM, BPM → UAM	Space Allocation: entry-exit-service synchronously requests a space from parking-service.	Real-time checks where the client is waiting (low latency: <300ms).
Asynchronous (MSK/Kafka)	P2 → K → S2, P4 → K → S3	Billing: entry-exit-service publishes VehicleCheckedOut event, which billing-service consumes to generate the invoice.	Decoupling and scalability for long-running or non-critical tasks.

IV. Project Methodology: Usage of AI (LLM) During the Project

During the Domain-Driven Design (DDD) modeling and architectural planning phases, the team utilized a Large Language Model (LLM), specifically OpenAI's ChatGPT 4, as a Technical Assistant and Refinement Engine.

A. Type of AI Tool Used

The tool used was a Generative AI (specifically, an LLM) employed primarily for synthesizing complex information, refining domain language, and ensuring compliance with technical industry standards.

AI Tool	Function	Project Phase
Claude (Model: Sonnet 4.5)	Generating code	Generating structural & behavioral diagrams and refactoring code
ChatGPT 4	Technical Assistant/Consultant	DDD Modeling, Architecture Drafting, Q&A Refinement
Models: Sonnet 4.5 and Haiku 4.5	Development Assistant	Organize DDD documentation structure, format domain models and bounded contexts for consistency, and manage the synthesis of business requirements into coherent bounded contexts with aggregates and domain events: OpenCode, terminal user interface

B. Purpose and Rationale for AI Use

The AI was used to accelerate the conceptualization phase by:

1. Synthesizing Industry Standards: Quickly integrating complex, external requirements like the OCPP 2.0.1 protocol and smart grid features into the domain model definitions.
2. Validating DDD Concepts: Ensuring the Ubiquitous Language and Aggregate boundaries were robust and aligned with 2025 DDD best practices.

3. Refining Technical Q&A: Serving as an intermediary to process the technical manager's requirements into clear architectural constraints (e.g., translating "RTO 1 hour, RPO 15 minutes" into "Multi-AZ, 15-min backups" requirements for Hesham).

The AI's output was treated as a draft proposal, which the human team members (Mihai and Hesham) then reviewed, verified, and manually integrated into the final artifacts, such as the Bounded Context Diagram and Domain Models.

C. Sample Prompts Used

The following are samples of the prompts used to guide the AI assistant, demonstrating the type of input and the targeted output:

Role	Sample Prompt	Targeted Output
DDD Lead (Mihai)	"Given the core domains Parking Space Management and EV Charging Station Management, define the initial 8 Bounded Contexts for a multi-facility system that must integrate OCPP 2.0.1 and support a cross-facility subscription model."	The initial Bounded Context Map, including the EVC and UAM contexts, and key integration patterns.
Architect (Hesham)	"We decided on AWS API Gateway, ECS Fargate, and Amazon MSK. Draft the service responsibilities and API specifications for the EV Charging microservice (charging-service), ensuring compliance with a 400 kW max load balancing rule and the asynchronous collection of meter values."	The detailed charging-service API endpoints, its integration with the Event Bus, and its load balancing mechanism.
Refactoring (Ha Vu)	"Analyze the anti-patterns present in a Python class that performs both GUI updates and business logic, and propose how to separate these concerns using the Observer Pattern."	The justification for using the Observer Pattern to solve the God Object anti-pattern and achieve SRP.

Conclusion

This project successfully achieved its mandate to transform a tightly coupled single-facility prototype into a professional, scalable, and cloud-native architecture for EasyParkPlus. By strategically applying modern software engineering principles, the team established a robust foundation ready to support the company's expansion into multi-facility operations and the new EV charging business line.

Key Project Achievements:

- **Code Quality and Maintainability:** The codebase was stabilized by systematically removing major anti-patterns, including the God Object and Global Variables, and separating concerns across distinct modules. The implementation of the Factory Pattern and Observer Pattern ensured high maintainability, testability, and adherence to SRP and OCP.
- **Strategic Domain Alignment (DDD):** The system's complexity was managed using Domain-Driven Design, resulting in eight distinct Bounded Contexts. This modeling effort successfully integrated complex new requirements, such as OCPP 2.0.1 compliance and cross-facility subscription benefits, directly into the Ubiquitous Language and domain models.
- **Scalable Microservices Architecture:** The proposed architecture, built on AWS managed services (ECS Fargate, Amazon MSK), provides the necessary platform to handle moderate traffic (50-100 req/sec) while meeting the strict RTO (1 hour) and RPO (15 minutes) Disaster Recovery targets. The hybrid communication model ensures responsive real-time operations alongside decoupled, scalable backend workflows.

In summary, the refactored code and the comprehensive microservices architecture proposal provide a high-quality, actionable blueprint that validates the strategic requirements set by EasyParkPlus, positioning the company for its March 2026 MVP launch and future growth.