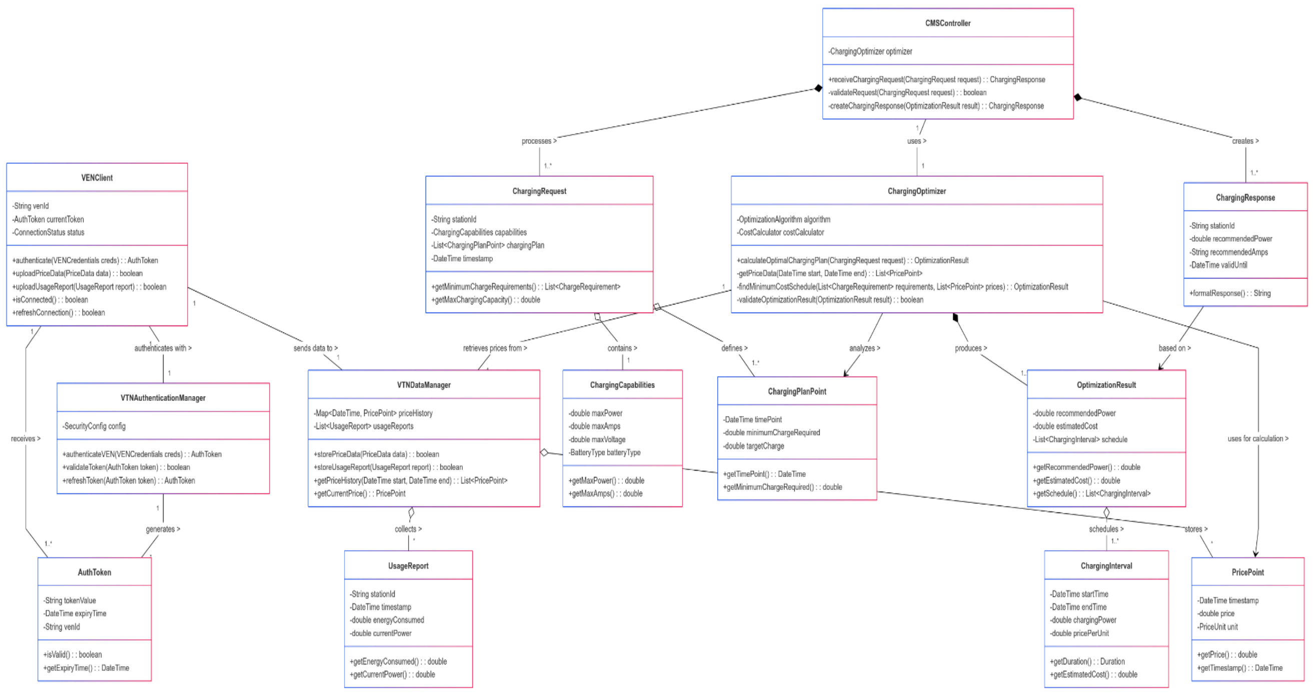
Product Design

Team 46 **Hiten Garg, Monosij Roy, Shlok Sand, Aditya Shankar, Eshwar Sriramoju**

Design Model

https://drive.google.com/file/d/1bxqQH-TKHcYuAZOxHR8V0pm3BHJSfVZi/view?usp=sharing


CMSController
-ChargingOptimizer optimizer
+receiveChargingRequest(ChargingRequest request) : : ChargingResponse
-validateRequest(ChargingRequest request) : : boolean
-createChargingResponse(OptimizationResult result) : : ChargingResponse
processes >
1..*
uses >
1
1
creates >
1..*
VENClient
-String venId
-AuthToken currentToken
-ConnectionStatus status
+authenticate(VENCredentials creds) : : AuthToken
+uploadPriceData(PriceData data) : : boolean
+uploadUsageReport(UsageReport report) : : boolean
+isConnected() : : boolean
+refreshConnection() : : boolean
ChargingRequest
-String stationId
-ChargingCapabilities capabilities
-List<ChargingPlanPoint> chargingPlan
-DateTime timestamp
+getMinimumChargeRequirements() : : List<ChargeRequirement>
+getMaxChargingCapacity() : : double
ChargingOptimizer
-OptimizationAlgorithm algorithm
-CostCalculator costCalculator
+calculateOptimalChargingPlan(ChargingRequest request) : : OptimizationResult
-getPriceData(DateTime start, DateTime end) : : List<PricePoint>
-findMinimumCostSchedule(List<ChargeRequirement> requirements, List<PricePoint> prices) : : OptimizationResult
-validateOptimizationResult(OptimizationResult result) : : boolean
ChargingResponse
-String stationId
-double recommendedPower
-String recommendedAmps
-DateTime validUntil
+formatResponse() : : String
1
1
receives >
authenticates with >
sends data to >
retrieves prices from >
contains >
defines >
analyzes >
produces >
based on >
uses for calculation >
1..*
1..
VTNAuthenticationManager
-SecurityConfig config
+authenticateVEN(VENCredentials creds) : : AuthToken
+validateToken(AuthToken token) : : boolean
+refreshToken(AuthToken token) : : AuthToken
VTNDataManager
-Map<DateTime, PricePoint> priceHistory
-List<UsageReport> usageReports
+storePriceData(PriceData data) : : boolean
+storeUsageReport(UsageReport report) : : boolean
+getPriceHistory(DateTime start, DateTime end) : : List<PricePoint>
+getCurrentPrice() : : PricePoint
ChargingCapabilities
-double maxPower
-double maxAmps
-double maxVoltage
-BatteryType batteryType
+getMaxPower() : : double
+getMaxAmps() : : double
ChargingPlanPoint
-DateTime timePoint
-double minimumChargeRequired
-double targetCharge
+getTimePoint() : : DateTime
+getMinimumChargeRequired() : : double
OptimizationResult
-double recommendedPower
-double estimatedCost
-List<ChargingInterval> schedule
+getRecommendedPower() : : double
+getEstimatedCost() : : double
+getSchedule() : : List<ChargingInterval>
generates >
collects >
schedules >
stores >
1..*
*
AuthToken
-String tokenValue
-DateTime expiryTime
-String venId
+isValid() : : boolean
+getExpiryTime() : : DateTime
UsageReport
-String stationId
-DateTime timestamp
-double energyConsumed
-double currentPower
+getEnergyConsumed() : : double
+getCurrentPower() : : double
ChargingInterval
-DateTime startTime
-DateTime endTime
-double chargingPower
-double pricePerUnit
+getDuration() : : Duration
+getEstimatedCost() : : double
PricePoint
-DateTime timestamp
-double price
-PriceUnit unit
+getPrice() : : double
+getTimestamp() : : DateTime

VENClient	**Class state** ● VEN identification information (venId) ● Authentication status (currentToken) ● Connection status (status) E.g., maintains the identity of the Virtual End Node and its current authentication state with the VTN. **Class behavior** ● **authenticate(VENCredentials creds)**: AuthToken - Authenticates with VTN ● **uploadPriceData(PriceData data)**: boolean - Sends pricing to VTN ● **uploadUsageReport(UsageReport report)**: boolean - Sends usage data to VTN ● **isConnected()**: boolean - Checks connection status ● **refreshConnection()**: boolean - Reestablishes connection if needed
VTNAuthenticationManager	**Class state** ● Security configuration (config) E.g., contains settings for token validation, security policies, and authentication parameters. **Class behavior** ● **authenticateVEN(VENCredentials creds)**: AuthToken - Validates VEN and issues token ● **validateToken(AuthToken token)**: boolean - Verifies token validity ● **revokeToken(AuthToken token)**: boolean - Invalidates token ● **refreshToken(AuthToken token)**: AuthToken - Issues new token before expiration

AuthToken	**Class State** ● **Token value** (tokenValue) ● **Expiration time** (expiryTime) ● **VEN identifier** (venId) **E.g.,** a security token that confirms the VEN's authenticated status with expiration information. **Class Behavior** ● **isValid()**: boolean - Checks if the token is still valid ● **getExpiryTime()**: DateTime - Returns token expiration time
VTNDataManager	**Class State** ● **Price data repository** ● **Usage report storage** E.g., manages the storage and retrieval of pricing data and charging usage information in the VTN system **Class Behaviour** ● **storePriceData(PriceData data): boolean** - Saves pricing information ● **storeUsageReport(UsageReport report): boolean** - Saves usage data ● **getPriceHistory(DateTime start, DateTime end): List<PricePoint>** - Retrieves historical prices ● **getCurrentPrice(): PricePoint** - Gets current price information
CMSController	**Class state** ● Request processing configuration E.g., Manages the receipt and processing of charging requests from the Charging Management System. **Class behavior** ● **receiveChargingRequest(ChargingRequest request):** ChargingResponse - Processes CMS requests ● **validateRequest(ChargingRequest request): boolean** - Checks request validity ● **createChargingResponse(OptimizationResult result)**: ChargingResponse - Forms response with optimization data

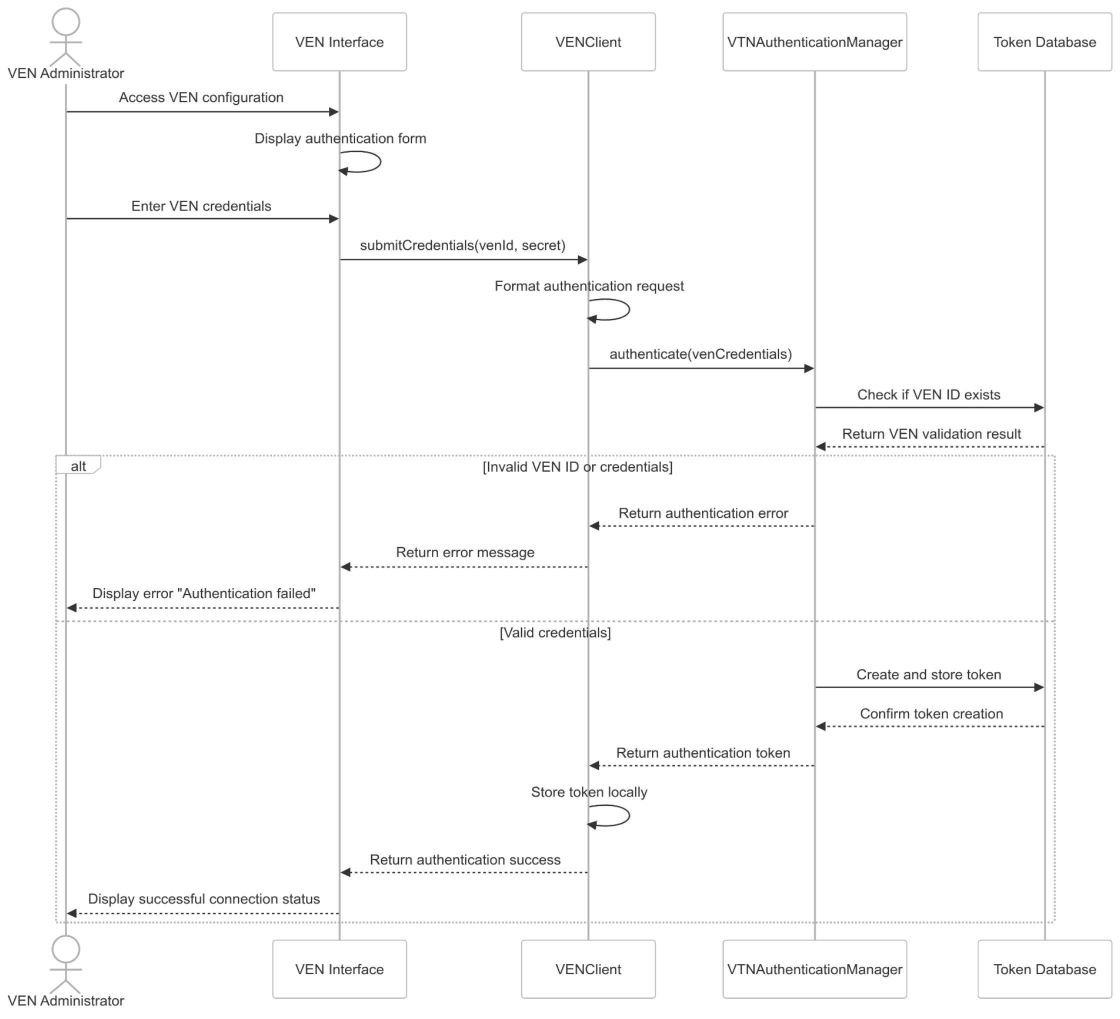
ChargingRequest	**Class state** ● Station identifier (stationId) ● Station capabilities (capabilities) ● Charging plan requirements (chargingPlan) ● Request timestamp (timestamp) E.g., contains an array of time points with minimum charge requirements (60%, 70%, etc.) and station capabilities. **Class behaviour** ● **getMinimumChargeRequirements()**: List<ChargeRequirement> - Extracts charge targets ● **getMaxChargingCapacity():** double - Returns maximum charging capacity
ChargingResponse	**Class State** ● **Station identifier** (stationId) ● **Recommended power level** (recommendedPower) ● **Recommended amperage** (recommendedAmps) ● **Response validity period** (validUntil) **E.g.,** contains "20000W/24A" recommendation for optimal charging. **Class Behavior** ● **formatResponse()**: String - Formats the charging recommendation for CMS
ChargingOptimizer	**Class State** ● **Optimization algorithms** ● **Cost calculation models** **E.g.,** contains the logic for determining the most economical charging schedule based on price data and requirements. **Class Behavior** ● **calculateOptimalChargingPlan(ChargingRequest request)**: OptimizationResult - Main optimization method ● **getPriceData(DateTime start, DateTime end)**: List<PricePoint> - Retrieves relevant pricing ● **findMinimumCostSchedule(List<ChargeRequirement> requirements, List<PricePoint> prices)**: OptimizationResult - Core algorithm

	• **validateOptimizationResult(OptimizationResult result)**: boolean - Ensures optimization is valid
OptimizationResult	**Class State** • **Recommended charging power** (recommendedPower) • **Cost estimation** (estimatedCost) • **Charging schedule** (schedule) **E.g.,** contains the optimal charging power recommendation of 20000W along with the calculated schedule. **Class Behavior** • **getRecommendedPower()**: double - Returns optimal power setting • **getEstimatedCost()**: double - Returns estimated cost with this plan

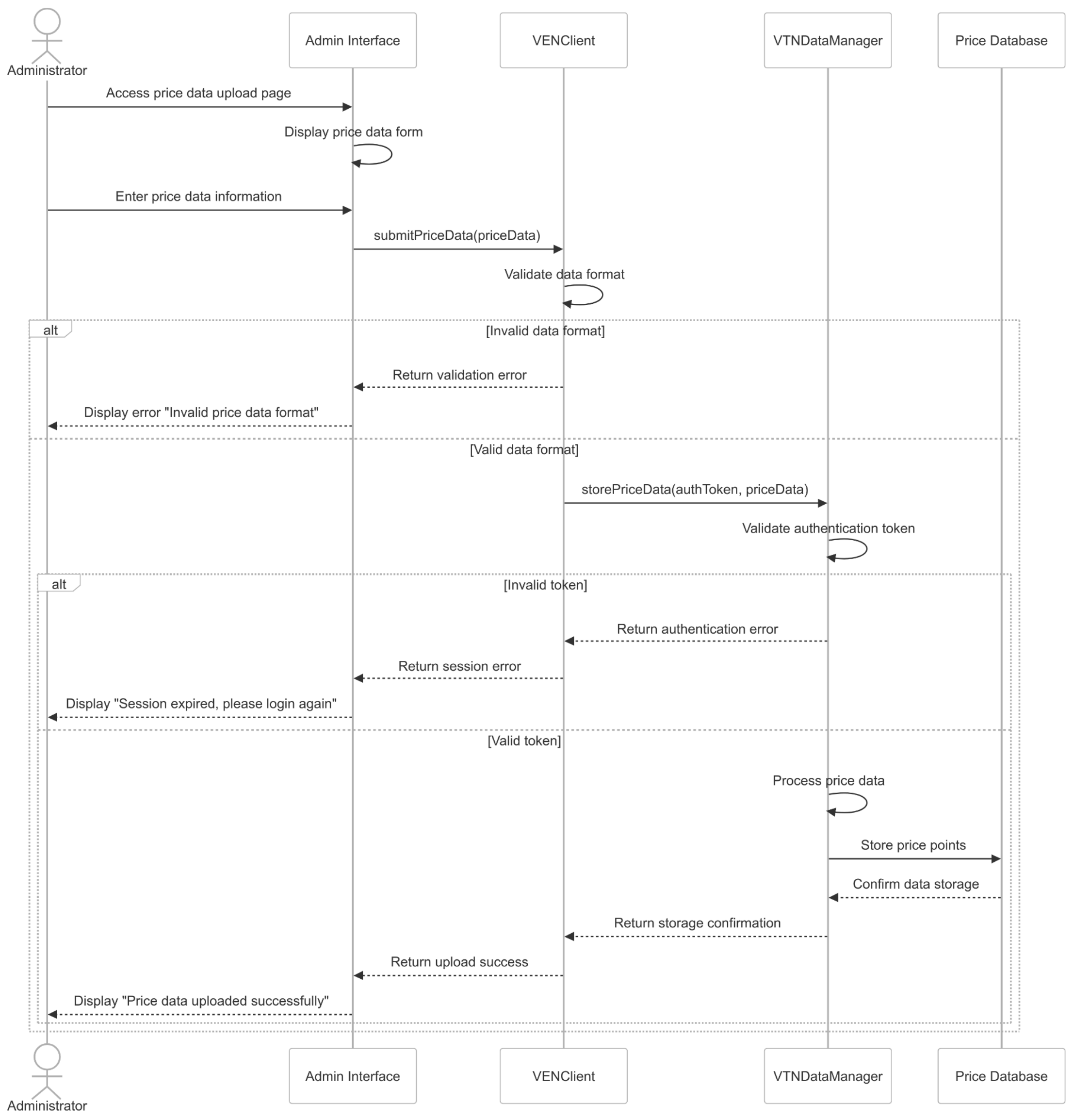
.

Sequence Diagram(s)

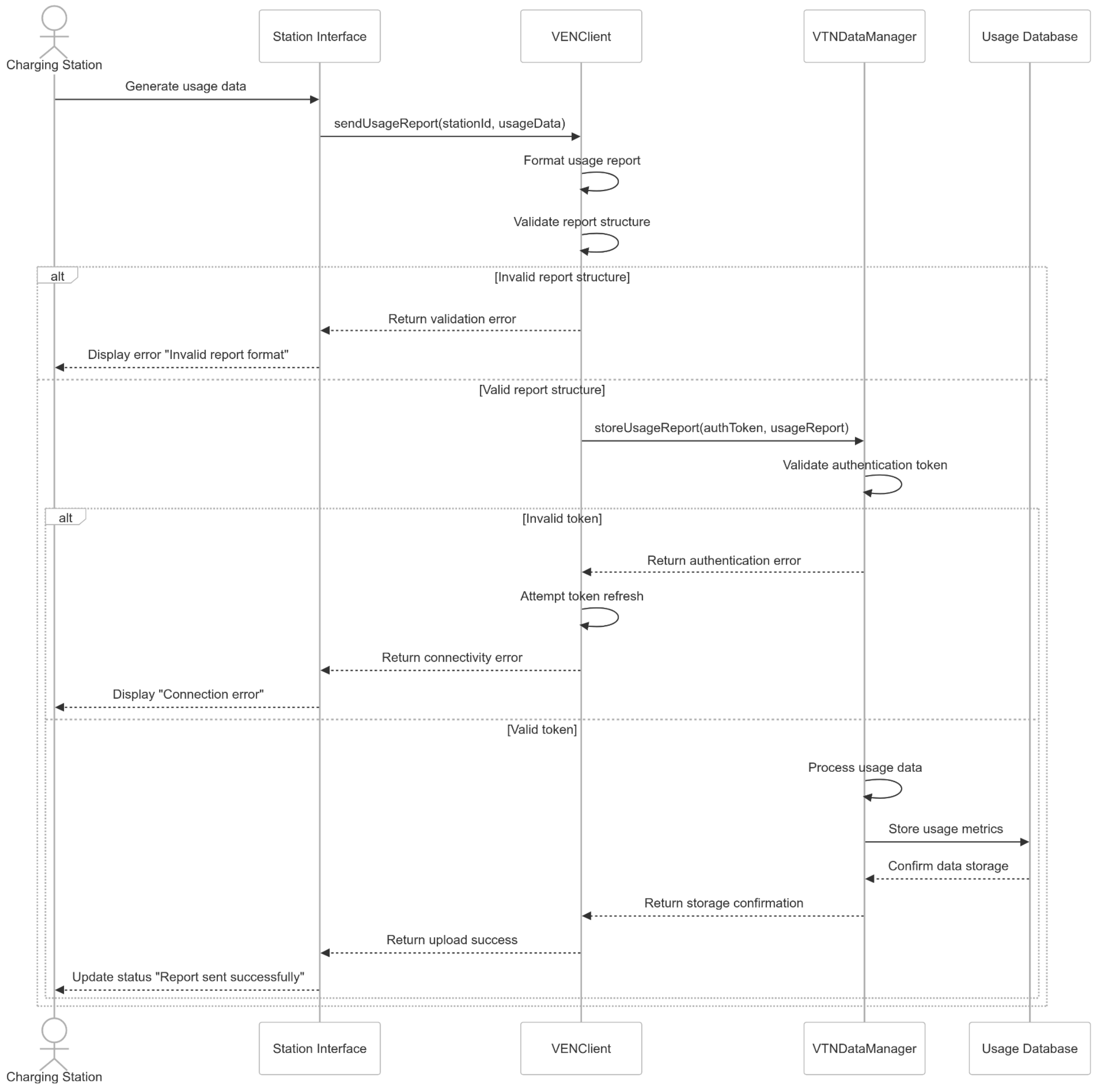
1) VEN Authentication Sequence Diagram


VEN Administrator
VEN Interface
VENClient
VTNAuthenticationManager
Token Database
Access VEN configuration
Display authentication form
Enter VEN credentials
submitCredentials(venId, secret)
Format authentication request
authenticate(venCredentials)
Check if VEN ID exists
Return VEN validation result
alt
[Invalid VEN ID or credentials]
Return authentication error
Return error message
Display error "Authentication failed"
[Valid credentials]
Create and store token
Confirm token creation
Return authentication token
Store token locally
Return authentication success
Display successful connection status
VEN Administrator
VEN Interface
VENClient
VTNAuthenticationManager
Token Database

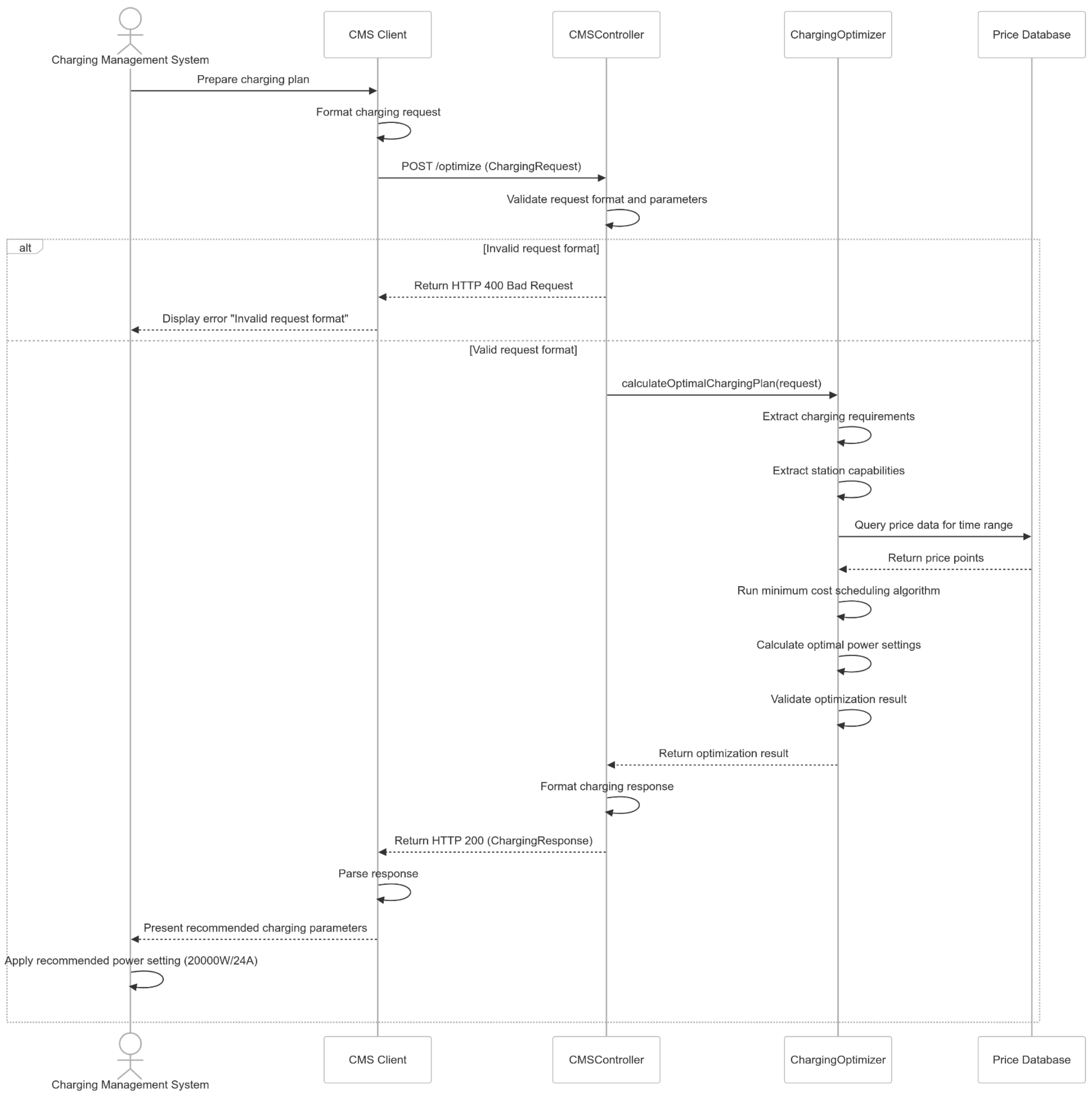
2) Price Data Upload Sequence Diagram



3)Usage Report Upload Sequence Diagram



4) Charging Optimization Sequence Diagram


Charging Management System
CMS Client
CMSController
ChargingOptimizer
Price Database
Prepare charging plan
Format charging request
POST /optimize (ChargingRequest)
Validate request format and parameters
alt
[Invalid request format]
Return HTTP 400 Bad Request
Display error "Invalid request format"
[Valid request format]
calculateOptimalChargingPlan(request)
Extract charging requirements
Extract station capabilities
Query price data for time range
Return price points
Run minimum cost scheduling algorithm
Calculate optimal power settings
Validate optimization result
Return optimization result
Format charging response
Return HTTP 200 (ChargingResponse)
Parse response
Present recommended charging parameters
Apply recommended power setting (20000W/24A)

Design Rationale

VEN-VTN Authentication

Chosen Design

A dedicated VENClient that authenticates with VTNAuthenticationManager using token-based authentication, with tokens stored in a database.

Alternatives Considered

1. ***Certificate-Based Authentication***

  - *Pros: Stronger security, no need for token management, industry standard for machine-to-machine authentication*
  - *Cons: Complex certificate management, higher implementation complexity, challenging certificate renewal process*
2. ***API Key Authentication***
  - *Pros: Simplicity of implementation, permanent credentials, widely used pattern*
  - *Cons: No built-in expiration, security risks if compromised, limited metadata capacity*
3. ***Basic Authentication with Username/Password***
  - *Pros: Simplest implementation, universally understood, minimal development effort*
  - *Cons: Poor security if not using HTTPS, credentials sent with every request, no inherent expiration*

Rationale for Final Decision
We selected token-based authentication because:

1. *It provides better security than API keys or basic authentication through automatic expiration*
2. *It's significantly simpler to implement than certificate-based authentication*
3. *The stateless nature of tokens reduces database load while still maintaining security*
4. *It aligns with industry standards for machine-to-machine communication*
5. *The refresh mechanism allows for continuous operation without manual intervention*

The main trade-off was accepting slightly lower security than certificate-based authentication in exchange for significantly reduced implementation complexity and easier debugging during development.

Price Data Management

Chosen Design
A VTNDataManager that stores price data in a dedicated database, with price points organized by timestamp.

Alternatives Considered

1. ***Flat File Storage***
  - *Pros: Simple implementation, easy backup, no database dependencies*
  - *Cons: Poor query performance, difficult to scale, limited indexing capabilities*
2. ***In-Memory Data Structure***
  - *Pros: Fastest possible access, simplest implementation for prototyping*
  - *Cons: Data loss on restart, memory limitations, no persistence guarantees*
3. ***Relational Database with Normalized Schema***
  - *Pros: Referential integrity, familiar technology, strong consistency guarantees*
  - *Cons: Potentially slower for time-series data, more complex queries for time ranges*

Rationale for Final Decision
We chose a time-series database approach within the VTNDataManager because:

1. *Price data is fundamentally time-series data with regular patterns*

2. *Time-series databases provide optimized performance for our primary query pattern (retrieving prices for a time range)*
3. *It offers better scaling characteristics as the volume of price data grows*
4. *The approach simplifies aggregation operations needed for optimization calculations*
5. *It balances query performance with implementation complexity*

While an in-memory approach would have been faster for development and small datasets, it couldn't provide the persistence and reliability needed for production. A normalized relational approach was rejected due to unnecessary complexity for our relatively simple data model and potential performance limitations for time-range queries.

Charging Optimization Algorithm

Chosen Design
A specialized ChargingOptimizer that implements a dynamic programming algorithm to find the minimum cost charging schedule.

Alternatives Considered

1. ***Greedy Algorithm Approach***
   - *Pros: Simplest implementation, fast execution, low computational resources*
   - *Cons: Only optimal for certain pricing structures, can miss global optimum*
   -
2. ***Heuristic Approach (Genetic Algorithm)***
   - *Pros: Can handle complex constraints, parallelizable*
   - *Cons: Non-deterministic results, complex implementation, potential performance issues*

Rationale for Final Decision
We selected the dynamic programming approach because:

1. *It guarantees an optimal solution, which directly translates to cost savings*
2. *The problem space is well-suited to dynamic programming (discrete time intervals, subproblem overlap)*
3. *Performance testing showed it could compute results within our 100ms target for 24-hour schedules*
4. *The approach handles changing price data and charging requirements without reconfiguration*
5. *The implementation complexity is manageable compared to alternatives like machine learning*

The greedy algorithm was initially prototyped but proved suboptimal with complex pricing structures. Machine learning was considered unnecessary complexity for what is essentially a deterministic optimization problem with clear constraints.

System Architecture

Chosen Design
A layered architecture with clear component separation (VENClient, CMSController, ChargingOptimizer) following service-oriented design principles.

Alternatives Considered

1. ***Microservices Architecture***
   - *Pros: Independent scaling, technology flexibility, clear service boundaries*
   - *Cons: Deployment complexity, service communication overhead, potential consistency issues*
2. ***Monolithic Application***
   - *Pros: Simpler development and deployment, lower initial overhead*
   - *Cons: Scaling limitations, tight coupling, potential maintenance challenges as system grows*
3. ***Event-Driven Architecture***
   - *Pros: Loose coupling, good for handling asynchronous processes*
   - *Cons: Complex event flows, learning curve, potential debugging challenges*

Rationale for Final Decision
We chose a layered architecture with service boundaries because:

1. *It provides a balance between the simplicity of a monolith and the scalability of microservices*
2. *The clear separation of concerns improves testability and maintenance*
3. *It allows us to evolve toward microservices in the future if needed*
4. *The design matches our team's expertise, reducing implementation risk*
5. *It accommodates our current scale without unnecessary complexity*

While microservices would offer better scalability, our current transaction volume doesn't justify the added complexity. The monolithic approach was rejected due to concerns about maintaining separation of concerns as the system grows. Event-driven architecture was considered but deemed unnecessary for our primarily request-response based interactions.

CMS Integration

Chosen Design
A REST API endpoint in CMSController that receives POST requests from CMS and returns optimized charging parameters.

Alternatives Considered

1. ***WebSocket Connection***
   - *Pros: Real-time updates, reduced overhead for frequent communication*
   - *Cons: More complex implementation, stateful connection management*

Rationale for Final Decision
We selected a REST API approach because:

1. *It's widely understood and easy to implement on both client and server sides*
2. *The request-response pattern matches our optimization workflow naturally*
3. *It provides good developer tooling for testing and debugging*
4. *The performance is sufficient for our use case (infrequent requests with simple data structures)*

5. *It's easy to secure using standard patterns like API keys or tokens*

Error Handling and Resilience

Chosen Design
Comprehensive error handling at each system layer with retry mechanisms for network operations and token refresh capabilities.

Alternatives Considered

1. ***Minimal Error Handling***
   - *Pros: Simpler implementation, faster development*
   - *Cons: Poor reliability, difficult debugging, potential service disruptions*
2. ***Circuit Breaker Pattern***
   - *Pros: Better handling of downstream failures, prevents cascading failures*
   - *Cons: Added complexity, more sophisticated implementation required*
3. ***Global Error Handler***
   - *Pros: Centralized error management, consistent handling*
   - *Cons: Less context-specific handling, potential for overlooking specific error cases*

Rationale for Final Decision
We implemented comprehensive error handling because:

1. *Reliability is critical for an automation system interacting with energy infrastructure*
2. *Domain-specific error handling at each layer provides better context for resolution*
3. *The retry mechanisms balance immediate failure with persistence for transient issues*
4. *The approach provides better diagnosability through specific error information*
5. *It establishes a foundation for monitoring and alerting on system health*

While a circuit breaker would provide better resilience against persistent downstream failures, it added complexity that wasn't justified for our current scale. However, the design allows for adding circuit breakers in the future if needed. The minimal approach was rejected due to reliability concerns, as even temporary failures could lead to suboptimal charging decisions.