William Griner – willyg@iastate.edu

Benjamin Muslic – muslicb@iastate.edu

Jonathan Duron – jduron24@iastate.edu

Mohamed Elaagip – bigboimo@iastate.edu

4.2 Design Exploration

4.2.1 Design Decisions

Decision 1 - Scalability:

As our userbase grows, our infrastructure needs to be able to grow with it. As such, we've implemented a robust cloud computing infrastructure that can automatically scale to growing needs. We've accomplished this by implementing auto-scaling groups on our ec2 instances, allowing our compute power to scale with the amount of compute power being demanded by our users. We also have set up an elastic load balancer (ELB) to distribute traffic evenly between our ec2 instances as they scale horizontally.

Decision 2 - Security:

Security is implemented through a layered approach using public and private subnets. The public subnets contain only the EC2 instances that need to receive traffic from the internet (through the ELB), while the RDS database instances are placed in private subnets that have no direct route to the internet. For example, if a malicious actor gains access to an EC2 instance in the public subnet, they still can't directly access the database because it's in a private subnet and protected by its security group, which only allows incoming traffic on port 5432 (PostgreSQL) from the EC2 security group. This means even if an EC2 instance is compromised, the attacker would need to also breach the RDS security group rules to reach the database. Additionally, the database instances can still communicate with each other across AZs for replication, while remaining protected from external access.

Decision 3 – Availability:

Availability / failover is very important, because if our servers were to go down, then our app wouldn't function properly. And if our app didn't function properly, then our users would leave, and we would lose business. As such, we've taken multiple measures to ensure our servers are highly available and can handle failover. Failover is when one server fails, the traffic being handled on that server is passed over to another. Failover is primarily handled through multiple layers of redundancy. The application uses two Availability Zones, each containing EC2 instances managed by an Auto Scaling Group, which can automatically replace unhealthy instances and scale based

on demand. The Elastic Load Balancer continuously monitors instance health and automatically redirects traffic away from any failed instances to healthy ones in either AZ. For the database layer, RDS is configured in a Multi-AZ setup where the primary database in one AZ maintains synchronous replication with a standby database in another AZ. If the primary database fails, RDS automatically fails over to the standby database, typically within 60-120 seconds, with no manual intervention required. The application maintains the same connection endpoint throughout this process, ensuring continuous operation. This multi-layered approach to failover (ELB, Auto Scaling, Multi-AZ RDS) ensures high availability even if an entire Availability Zone experiences issues.

4.2.2 Ideation

For at least one design decision, describe how you ideated or identified potential options (e.g., lotus blossom
technique). Describe at least five options that you considered.

Ideation Process:

1. Started with AWS Database Categories

- Relational (RDS, Aurora)
- NoSQL (DynamoDB)
- Document (DocumentDB)
- In-memory (ElastiCache)
- Graph (Neptune)

Five Options Considered:

1. RDS PostgreSQL

- Relational database perfect for structured OBD codes
- Built-in Multi-AZ failover (shown in diagrams)
- Cost-effective for predictable workloads
- Strong querying for historical analysis

1. Aurora

- AWS's enhanced PostgreSQL/MySQL

- Better scaling but more expensive
- More features than needed for OBD app
- Overkill for initial scale

1. DynamoDB

- Serverless NoSQL database
- Great for massive scale
- Pay-per-request model
- Not ideal for related data like vehicle-code relationships

1. Amazon Neptune

- Graph database
- Could model relationships between vehicles, codes, and issues
- Expensive and complex
- Overkill for simple OBD code storage

1. Combination: RDS + ElastiCache

- RDS for persistent storage
- ElastiCache for frequently accessed codes
- Added complexity
- Higher cost for minimal benefit

4.2.3 Decision-Making and Trade-Off

Criteria (Weight 1-5):

1. Data Structure Fit (5) - How well it handles structured vehicle/OBD data
2. High Availability (4) - Failover capabilities and reliability
3. Cost Effectiveness (4) - Price vs features needed
4. Management Overhead (3) - Maintenance effort required
5. Scalability (2) - Ability to handle growth

6. Query Flexibility (3) - Ability to perform complex queries

Decision Matrix (Score 1-5):

Database Option	Data	HA	Cost	Mgmt	Scale	Query	Total
RDS PostgreSQL	5	5	4	4	3	5	89
Aurora	5	5	2	4	5	5	86
DynamoDB	2	5	3	5	5	2	71
DocumentDB	2	4	2	3	4	3	59
Neptune	3	4	1	2	4	4	61
RDS+ElastiCache	5	5	2	2	4	5	79

Formula: Sum of (Weight × Score)

RDS PostgreSQL (Total: 89)

- Data Structure (5/5): Perfect for OBD codes, vehicle info, and user data as structured tables with relationships
- High Availability (5/5): Multi-AZ failover shown in diagrams, automatic backups
- Cost (4/5): Pay for what you use, no oversized infrastructure
- Management (4/5): AWS handles most maintenance, patching
- Scalability (3/5): Can scale vertically, but has limits
- Query (5/5): Complex SQL queries for analyzing OBD data history

Aurora (Total: 86)

- Data Structure (5/5): Same capabilities as PostgreSQL
- High Availability (5/5): Better distributed architecture
- Cost (2/5): 20% more expensive than standard RDS
- Management (4/5): AWS-managed like RDS
- Scalability (5/5): Better auto-scaling, storage
- Query (5/5): Same SQL capabilities as PostgreSQL

DynamoDB (Total: 71)

- Data Structure (2/5): NoSQL isn't ideal for related OBD data
- High Availability (5/5): Fully managed, multi-AZ
- Cost (3/5): Pay-per-request could be unpredictable
- Management (5/5): Zero administration needed
- Scalability (5/5): Unlimited scale
- Query (2/5): Limited complex queries for OBD analysis

DocumentDB (Total: 59)

- Data Structure (2/5): Document model doesn't fit OBD codes well
- High Availability (4/5): Good failover but more complex
- Cost (2/5): Expensive for your needs
- Management (3/5): More complex to manage
- Scalability (4/5): Good scaling but overkill
- Query (3/5): Flexible but not ideal for structured data

Neptune (Total: 61)

- Data Structure (3/5): Graph model could work for vehicle-code relationships, but adds unnecessary complexity for simple OBD code storage. While it can model connections between vehicles, codes, and issues, it's more sophisticated than needed.
- High Availability (4/5): Offers good failover capabilities and replication across AZs (as shown in the architecture diagrams), but requires more complex configuration than RDS.
- Cost (1/5): Very expensive for basic needs. Graph databases are priced higher and the sophisticated features you'd be paying for aren't necessary for OBD code storage.
- Management (2/5): Complex to manage. Requires specialized knowledge of graph databases, query languages, and optimization techniques that aren't justified for this use case.
- Scalability (4/5): Excellent scaling capabilities for complex relationship queries, but this power is overkill for your needs.
- Query (4/5): Powerful for relationship-based queries, but the complexity of graph queries isn't necessary for simple OBD code retrieval and storage.

RDS+ElastiCache (Total: 79)

- Data Structure (5/5): Combines RDS's structured storage with ElastiCache's fast retrieval. Perfect for frequently accessed OBD codes while maintaining structured relationships.
- High Availability (5/5): Both services provide excellent HA features matching the multi-AZ architecture shown in the diagrams.
- Cost (2/5): Expensive because you're running and paying for two separate services when one would suffice.
- Management (2/5): Higher operational overhead managing both caching and database layers. Requires configuration and monitoring of two systems.
- Scalability (4/5): Great scaling with cache layer reducing database load, but adds complexity to the architecture shown in the diagrams.
- Query (5/5): Excellent query performance with cached reads and full SQL capability, but might be overengineered for your needs.

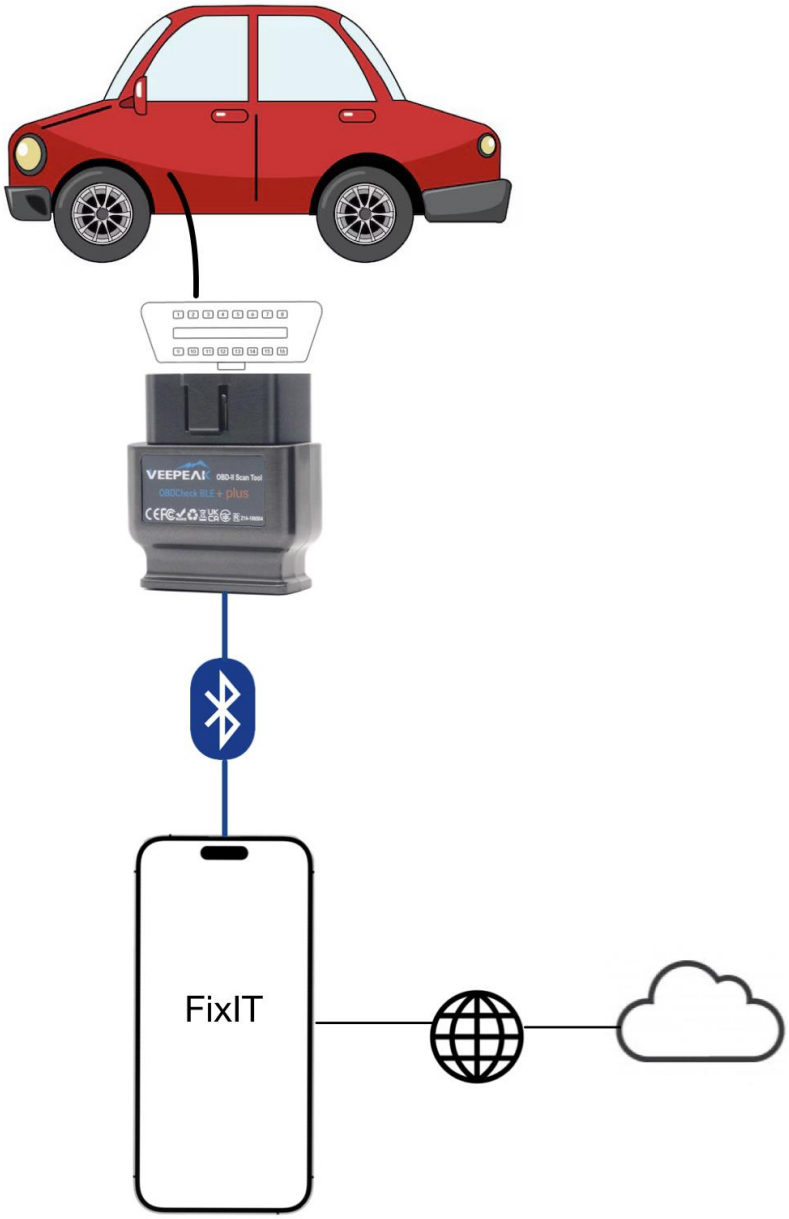
RDS PostgreSQL fits perfectly with:

- The Multi-AZ setup
- Security group configuration (port 5432)
- Integration with EC2 instances
- Structured data requirements

4.3 Proposed Design

4.3.1 Overview

The visual included here demonstrates the connection flow: the car's OBD-II dongle connects to the phone via Bluetooth, and the app communicates with the cloud to provide diagnostic insights.



OBD-II Dongle: This small device plugs into the car's OBD-II port, which is typically found under the dashboard. The OBD-II dongle reads diagnostic data, such as trouble codes, directly from the car's onboard computer. These codes are generated whenever there is a problem, like an engine issue or sensor fault.

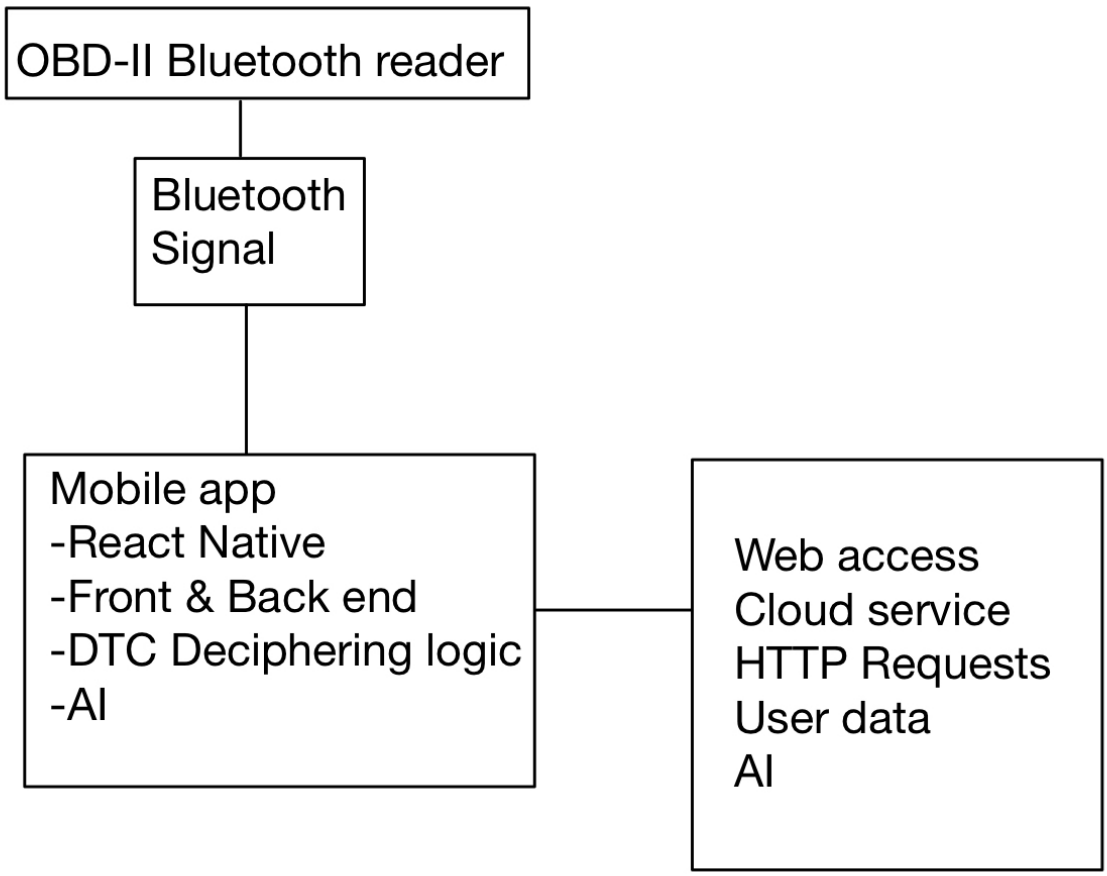
FixIt Mobile App: The FixIt app on the user's smartphone connects to the OBD-II dongle via Bluetooth. Once connected, the app receives the diagnostic codes from the car. The app also sends this information to the cloud for more advanced analysis.

Cloud Server: The cloud server stores, processes, and analyzes the diagnostic data using artificial intelligence (AI). It gathers insights from a vast amount of data and community knowledge to

provide clear, easy-to-understand explanations of what each code means, along with recommendations for fixes. It also stored account information and user histories.

4.3.2 Detailed Design and Visual(s)

The FixIt system integrates an **OBD-II Bluetooth Reader**, a **Mobile App**, and a **Cloud Service** to provide a comprehensive vehicle diagnostic solution. The OBD-II reader captures diagnostic data, the mobile app deciphers and manages this data, and the cloud service offers advanced processing, user data management, and AI-based insights.



Subsystems and Components

1. **OBD-II Bluetooth Reader**

   Role: The OBD-II reader plugs into the car's diagnostic port, retrieving diagnostic trouble codes (DTCs) directly from the vehicle's onboard computer.

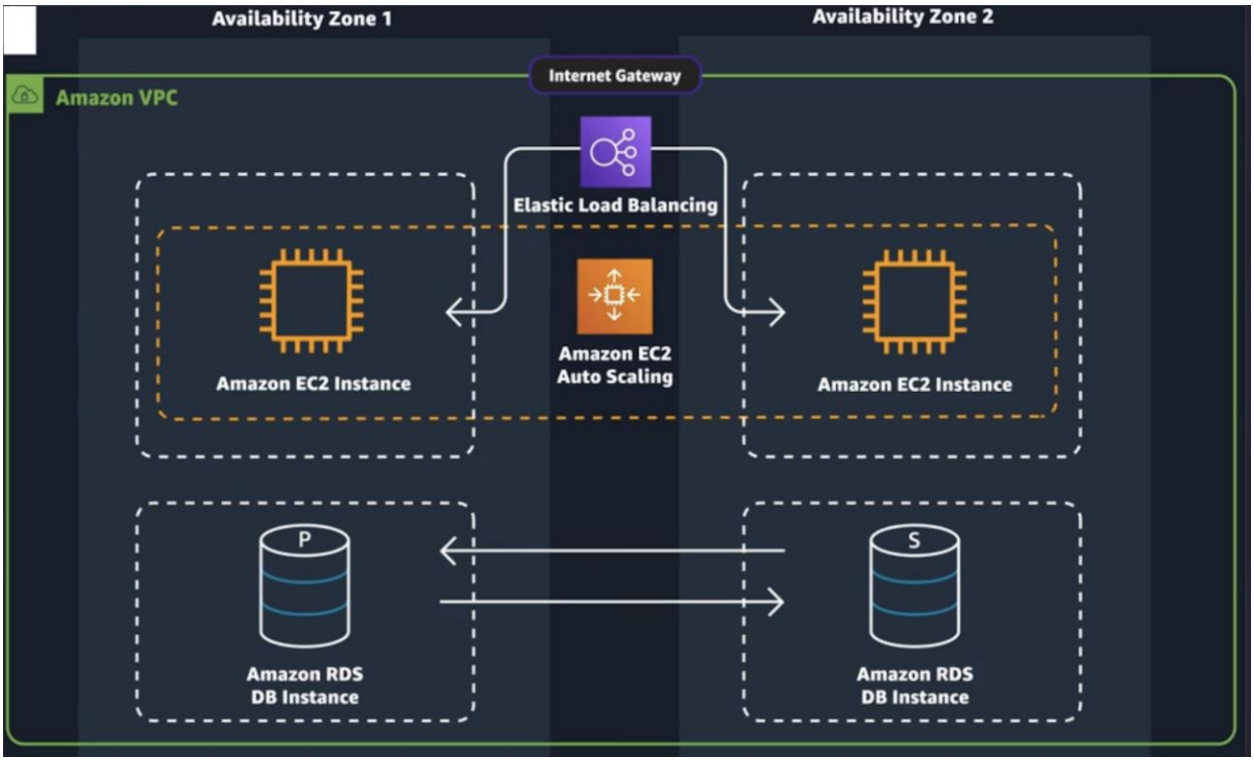
Operation: The reader uses Bluetooth to transmit DTCs to the mobile app. It continuously scans for updates, ensuring that the latest data is available.
Integration: The Bluetooth reader connects to the user's mobile device over Bluetooth Low Energy (BLE), enabling efficient data transfer while minimizing power consumption.

2. **Mobile App (Developed in React Native)**
   Role: The mobile app is the main user interface and processing unit for interacting with the OBD-II data, providing diagnostic insights and connecting with the cloud for additional analysis.
   Components:
      Front-End Interface: Built with React Native, the app supports both iOS and Android devices, ensuring compatibility across platforms.
      DTC Deciphering Logic: A built-in library decodes standard DTCs, providing users with basic interpretations of vehicle codes.
      AI Module (Client-Side): The app uses lightweight AI to provide common fixes for typical DTCs. For more complex diagnostics, it relies on cloud processing.
      Backend Integration: The app communicates with the cloud service using HTTP requests to send and retrieve data. It also handles user authentication through HTTP requests for secure access.
   Integration: The app connects to the OBD-II reader via Bluetooth, processes received data, and uses HTTP requests for user authentication and data exchange with the cloud.
3. **Cloud Service**
   Role: The cloud server is responsible for advanced data processing, user data management, and delivering AI-driven insights beyond what the mobile app can handle locally.
   Components:
      User Authentication: Authentication requests are handled via HTTP, ensuring secure access to user data and personalized diagnostics.
      Web Access and Data Management: The cloud service is accessed through HTTP requests, enabling data storage, user history, and long-term analysis for improved insights.
      AI Processing: The cloud uses machine learning models trained on extensive diagnostic datasets. These models provide in-depth recommendations, predictive maintenance tips, and patterns that enhance user insights.
   Integration: The cloud service communicates with the mobile app using RESTful HTTP requests. Authentication is required for access, ensuring that user data remains secure.

Detailed Integration and Data Flow

1. **OBD-II Bluetooth Reader to Mobile App**:
   The Bluetooth reader collects DTC data and sends it to the mobile app. The app's backend listens for Bluetooth data, processing it and updating diagnostic information as new data arrives.
2. **User Authentication**:
   The mobile app sends HTTP requests to the cloud for user authentication. Users must log in to access personalized data, which ensures security and tailored insights.
3. **Mobile App to Cloud Service**:
   For complex analysis, the mobile app sends the collected DTC data and any additional context (like car model or recent issues) to the cloud via secure HTTP POST requests.
4. **Cloud Service to Mobile App**:
   The cloud server processes incoming data, applies AI algorithms to provide insights, and sends the results back to the app through HTTP GET requests. The app then displays the information, giving users a clear, actionable summary.

Cloud Infrastructure Section



This architecture implements a highly available, auto-scaling application infrastructure deployed across two AWS Availability Zones (AZs) within a single Virtual Private Cloud (VPC). The design follows AWS best practices for fault tolerance and scalability.

Core Components

Network Layer

- Amazon VPC: Provides isolated network infrastructure
- Internet Gateway: Single point of internet connectivity
- Availability Zones: Two independent zones for redundancy

Compute Layer

- Elastic Load Balancer (ELB)
- Distributes incoming traffic across multiple EC2 instances
- Performs health checks on instances
- Supports session stickiness if required
- Auto Scaling Group
- Manages EC2 instance fleet

- Scales based on defined metrics
- Maintains instance distribution across AZs

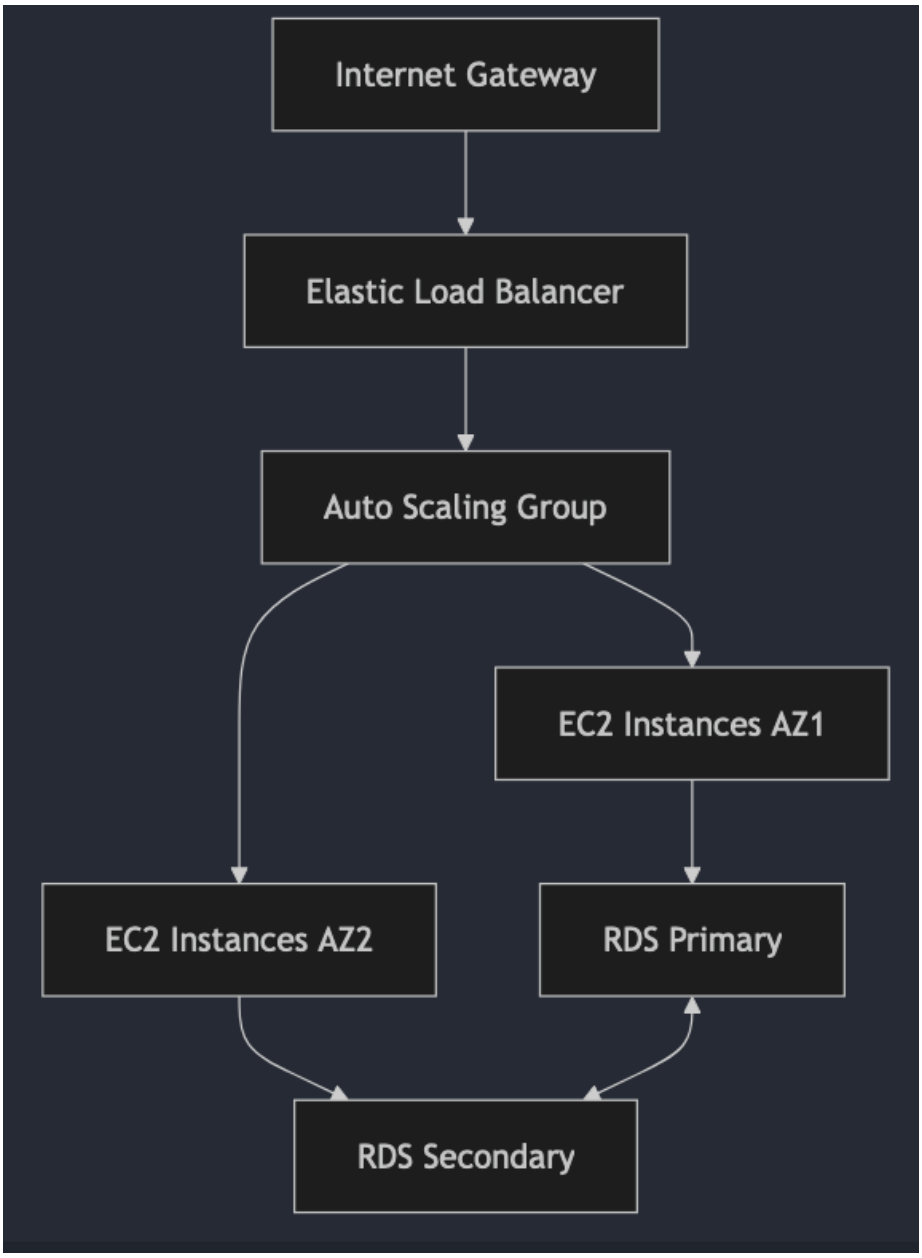
Database Layer

- Amazon RDS
- Primary-Secondary configuration
- Synchronous replication between AZs
- Automatic failover capability

Supporting Services

- IAM: Identity and access management
- S3: Object storage for static assets
- CloudWatch: Monitoring and metrics
- DynamoDB: NoSQL database (if required)

Component Integration Flow



Auto Scaling Configuration

Scale-Out Conditions (ANY trigger scaling)

- CPU Utilization > 70%
- Memory Utilization > 80%
- Request Count > 1000/minute/instance
- Queue Length > 100 messages
- Concurrent Connections > 500/instance

Scale-In Conditions (ALL must be true)

- CPU Utilization < 30%
- Memory Utilization < 40%
- Request Count < 300/minute/instance
- Queue Length < 20 messages
- Concurrent Connections < 100/instance

Capacity Settings

- Minimum: 1 instance per AZ
- Maximum: 3 instances per AZ
- Initial: 1 instance per AZ

Protection Mechanisms

- Cooldown Period: 300 seconds between scaling actions
- Health Check Grace Period: 300 seconds for new instances
- Connection Draining enabled for active sessions
- Scale across both AZs simultaneously
- Step Scaling:
- 70-85% CPU: Add 1 instance
- 85% CPU: Add 2 instances

Load Balancer Settings

- Health Check Path: /health
- Health Check Timeout: 5 seconds
- Health Check Interval: 30 seconds
- Cross-Zone Load Balancing: Enabled
- Security Group: HTTPS (443) from 0.0.0.0/0
- Connection Draining Timeout: 300 seconds

Auto Scaling Group Configuration

- Health Check Type: ELB
- Instance Protection: Disabled for scale-in
- Termination Policy: Oldest Instance First
- VPC Zone Identifier: Both AZ subnets registered

Health Check Configuration

Instance Health Check Endpoints

- Path: /health
- Response Timeout: 5 seconds
- Expected Status Code: 200 OK
- Check Interval: 30 seconds
- Failure Threshold: 2 consecutive failures
- Success Threshold: 2 consecutive successes

Health Check Criteria

- System Health:
- CPU Utilization < 70%
- Memory Usage < 80%
- Available Disk Space > 20%
- Application Health:
- Database Connection Status
- Web Server Status
- Critical Service Status

Grace Period Settings

- Initial Grace Period: 300 seconds
- No health checks during grace period
- Allows for:
- Instance boot completion
- Application startup

- Dependency initialization
- Database connection establishment

Instance Replacement Strategy

- Launch new instance first
- Wait for health check pass
- Traffic shifting:
- Begin traffic routing after health checks pass
- Gradual traffic migration
- Maintain minimum capacity during replacement
- Termination Policy:
- Remove oldest instance first
- Ensures fleet modernization
- Reduces technical debt
- Maintains updated configurations

Unhealthy Instance Handling

- Automatic replacement on health check failure
- Connection draining before termination
    - Connection draining is a feature that helps gracefully remove EC2 instances from service in an ELB/target group
    - When enabled, the load balancer stops sending new requests to the instance that is being drained, while keeping existing in-flight requests running
- Maintain capacity across AZs
- Logging of replacement events
- Alert notifications for repeated failures

4.3.3 Functionality

Connecting the OBD-II Dongle:

- **User Action**: The user plugs the OBD-II Bluetooth dongle into their vehicle's OBD-II port, usually located under the dashboard.
- **System Response**: The dongle powers on and begins communicating with the vehicle's onboard computer, gathering any available diagnostic trouble codes (DTCs).

Launching the FixIt App:

- **User Action**: The user opens the FixIt mobile app on their smartphone, which automatically prompts them to enable Bluetooth if it's not already active.
- **System Response**: The app searches for the OBD-II dongle, pairs with it over Bluetooth, and establishes a connection. A notification in the app confirms the connection and begins receiving diagnostic data.

Gathering and Displaying Basic Diagnostics:

- **User Action**: The user waits a few seconds as the app pulls DTC data from the vehicle.
- **System Response**: The app decodes the basic DTCs using its built-in database, displaying a simplified summary of each code and its associated issue (e.g., "P0420: Catalyst System Efficiency Below Threshold"). The app provides basic insights on what the codes mean and potential fixes.

Requesting Advanced Analysis:

- **User Action**: If the user needs more detailed information or is unsure about the next steps, they can tap an "Analyze Further" button in the app.
- **System Response**: The app securely sends the DTC data and other relevant information (like car make and model) to the cloud for advanced processing. The cloud server applies AI algorithms to analyze the data, leveraging insights from vast databases to identify patterns, probable causes, and recommended maintenance.

Receiving AI-Driven Insights:

- **User Action**: The user waits a few moments while the app fetches results from the cloud.
- **System Response**: Once the cloud analysis is complete, the app displays detailed insights, such as potential issues related to the DTCs, maintenance tips, and suggested steps to resolve the problem. The insights may include information from online communities, past user experiences, or manufacturer data, empowering the user to make informed decisions.

Taking Action Based on Diagnostics:

- **User Action**: Based on the app's insights, the user decides whether to perform simple maintenance themselves, contact a mechanic, or monitor the issue further.

- **System Response**: The app provides options like setting a reminder to check the issue later or accessing DIY guides. It also saves the diagnostics history in the cloud, allowing the user to track issues over time.

4.3.4 Areas of Concern and Development

The current FixIt design effectively satisfies the primary requirements and addresses key user needs. By combining an OBD-II Bluetooth dongle, a user-friendly mobile app, and cloud-based AI insights, the system offers a powerful and accessible diagnostic tool for car owners. Key aspects include:

- **Ease of Use**: The Bluetooth OBD-II dongle and mobile app provide a seamless, user-friendly experience. The app's clear interface simplifies complex diagnostic trouble codes (DTCs), allowing users without technical knowledge to understand and act on vehicle issues.
- **Detailed Insights**: The AI-powered cloud processing meets the need for in-depth diagnostics, going beyond simple code readings to offer actionable insights. This fulfills the requirement for preventative maintenance recommendations and gives users the ability to make informed decisions.
- **Cross-Platform Compatibility**: Using React Native ensures the app is accessible on both iOS and Android, widening the user base and meeting the requirement for universal accessibility.

Primary Concerns for Delivering a Product That Meets Requirements

1. **Bluetooth Connectivity Stability**: Maintaining a stable Bluetooth connection between the OBD-II dongle and the mobile app is crucial. Interference or connectivity issues may cause data loss or delay in real-time diagnostics, impacting the user experience.
2. **Cloud Processing Latency**: Since in-depth analysis relies on cloud processing, there is a potential latency in data transmission and response. Slow or unstable internet connections could delay diagnostic results, frustrating users who expect quick insights.
3. **Data Security and Privacy**: Since user data and vehicle diagnostics are sent to the cloud, securing this data is paramount. Meeting data security standards is essential to protect user privacy and build trust.

4.4 TECHNOLOGY CONSIDERATIONS

1. OBD-II Bluetooth Dongle

- **Description**: The OBD-II Bluetooth dongle plugs into the car's diagnostic port and communicates with the mobile app over Bluetooth, relaying vehicle diagnostic data.
- **Strengths**:
    - **Wireless Connectivity**: Bluetooth provides convenient, wireless communication with the mobile app.
    - **Low Power Consumption**: Bluetooth Low Energy (BLE) helps reduce battery usage.
    - **Ease of Use**: Simple plug-and-play design allows users to set it up easily.
- **Weaknesses**:
    - **Limited Range**: Bluetooth range is limited, so the user must be close to the car for reliable connectivity.
    - **Potential Connectivity Issues**: Bluetooth connections can sometimes be unstable, leading to data loss or delay.
    - **Limited Data Rate**: Bluetooth may not be able to handle large volumes of data at high speeds compared to WiFi.
- **Trade-Offs**:
    - **Bluetooth vs. WiFi**: Bluetooth was chosen for its lower power consumption and ease of setup, though it has a shorter range and lower data throughput compared to WiFi.
- **Alternative Solutions**:
    - **WiFi OBD-II Dongle**: WiFi would provide a faster and potentially more stable connection, especially for larger data transfers. However, it requires more power and may drain the vehicle's battery faster.
    - **Wired OBD-II Connection**: A wired connection would eliminate connectivity issues but reduce ease of use and limit user flexibility.

2. React Native for Mobile App Development

- **Description**: React Native is a cross-platform development framework that allows us to build a single codebase for both iOS and Android devices.
- **Strengths**:

    - **Cross-Platform Compatibility**: Enables the app to work on both iOS and Android without separate codebases, saving development time and resources.
    - **Strong Community and Libraries**: React Native has a large developer community and many third-party libraries, allowing faster development and easier troubleshooting.
    - **Native Performance**: Though it's a cross-platform framework, it provides near-native performance, suitable for a real-time app.
- **Weaknesses**:
    - **Limited Access to Some Native Features**: Certain device features may be harder to implement than with fully native development.
    - **Potential Performance Bottlenecks**: For highly complex or graphics-intensive apps, React Native may not perform as well as fully native apps.
- **Trade-Offs**:
    - **React Native vs. Native Development**: React Native was chosen for cross-platform compatibility and faster development time. While native development offers more optimized performance, it requires maintaining separate iOS and Android codebases, which is time-intensive.
- **Alternative Solutions**:
    - **Flutter**: Flutter is another cross-platform framework known for its smooth UI but has a smaller library ecosystem than React Native.
    - **Native Development**: Developing separate iOS and Android apps would allow for full use of native features but would significantly increase development and maintenance effort.

3. Cloud Computing for Data Processing and Storage

- **Description**: The cloud server handles complex data processing, user data management, and AI-driven diagnostics. It enables long-term data storage and machine learning capabilities that exceed mobile device processing power.
- **Strengths**:
    - **Scalability**: The cloud can handle large amounts of data and scale as the user base grows.
    - **Advanced Processing Power**: Cloud servers can perform complex AI and machine learning analyses, which would be difficult on a mobile device.
    - **Data Storage**: The cloud stores user history and diagnostic data, enabling personalized and historical analysis.

- **Weaknesses**:
    - **Dependency on Internet Connectivity**: Cloud processing requires a stable internet connection, which could delay diagnostics if connectivity is poor.
    - **Latency**: Sending data to the cloud and waiting for a response introduces latency, which may impact real-time user experience.
    - **Data Privacy Concerns**: Storing and processing user data in the cloud requires strong security measures to protect user privacy.
- **Trade-Offs**:
    - **Cloud vs. Local Processing**: Cloud processing allows for more complex analysis but introduces latency and requires internet access. Local processing on the mobile device would reduce latency but limits the complexity of AI processing.
- **Alternative Solutions**:
    - **Edge Computing**: Processing data closer to the device (e.g., on a local server or a powerful in-car device) would reduce latency and dependency on internet connectivity. However, it would require additional hardware and might limit scalability.
    - **On-Device Processing**: For some diagnostics, lightweight AI could be implemented directly on the mobile app. This would allow real-time analysis without internet dependency but limits the depth of analysis possible with cloud AI.

4. AI and Machine Learning for Diagnostics

- **Description**: AI models in the cloud analyze vehicle data, recognize patterns, and provide personalized insights to help users understand and address issues.
- **Strengths**:
    - **Predictive Insights**: Machine learning models can provide predictive maintenance recommendations based on patterns in vehicle data.
    - **Enhanced User Experience**: AI improves the user experience by offering tailored suggestions rather than just raw data.
    - **Continual Improvement**: AI models can improve over time as more data is collected, making the system more accurate and helpful.
- **Weaknesses**:
    - **Complexity**: Developing, training, and deploying machine learning models can be time-intensive and requires specialized knowledge.

    - **Resource Intensive**: AI processing demands significant computational resources, which could lead to higher cloud costs.
    - **Data Dependency**: AI models rely on a large volume of data to be effective, which may be challenging to collect initially.
- **Trade-Offs**:
    - **AI vs. Rule-Based System**: AI provides dynamic insights and can improve over time, whereas a rule-based system would be simpler and more predictable but unable to adapt or improve based on new data.
- **Alternative Solutions**:
    - **Rule-Based System**: Instead of AI, a rule-based system could be used to provide insights based on predefined conditions. While simpler, it would lack the flexibility and predictive power of machine learning.
    - **Hybrid Approach**: Combining rule-based logic on the app with AI-based analysis in the cloud could offer a balance, providing fast insights locally while using AI for more complex cases in the cloud.

5. HTTP Requests for Data Transmission and User Authentication

- **Description**: HTTP requests facilitate communication between the mobile app and the cloud server, handling data transmission and user authentication.
- **Strengths**:
    - **Standard Protocol**: HTTP is widely supported and compatible with most networks, making it easy to implement.
    - **Secure Communication**: HTTPS (secure HTTP) provides encryption, which is essential for user data and authentication.
    - **Reliability**: HTTP is a reliable protocol with established methods for error handling and recovery.
- **Weaknesses**:
    - **Latency**: HTTP requests introduce some latency, which could impact real-time diagnostics if responses are delayed.
    - **Internet Dependency**: HTTP requires an internet connection, so the app won't function fully offline.
- **Trade-Offs**:
    - **HTTP vs. MQTT**: While HTTP is simpler and more widely used for apps, MQTT (a lightweight messaging protocol) could reduce latency and improve reliability in low-bandwidth environments. However, MQTT would require additional setup and may not be as straightforward to implement.

- **Alternative Solutions**:
    - **MQTT Protocol**: MQTT is a low-overhead protocol that could reduce latency in data transmission. However, it may require additional infrastructure for cloud integration.
    - **WebSockets**: WebSockets offer a persistent connection, allowing for faster, two-way communication, but they may be more complex to implement than standard HTTP.

4.5 DESIGN ANALYSIS

Design 1: Male OBD Connector with CAN H/L Connected to MCP2515 and Arduino Board

- **What We Did**: In the first design, we built a setup using a male OBD connector wired to the CAN high and CAN low pins on an MCP2515 CAN bus module, which was connected to an Arduino board. This allowed us to receive CAN data directly from the car's OBD-II port.
- **Results**: This setup worked in terms of receiving raw CAN data, which validated that we could collect diagnostic information. However, we encountered a major issue: the lack of resources to decipher DTC (Diagnostic Trouble Codes) in a user-friendly way. Additionally, the amount of hardware required (OBD connector, MCP2515, Arduino) was excessive and impractical for common users.
- **Why It Didn't Work**: The setup was too complex for the average user, requiring too many components and manual configuration. Without a straightforward method to decipher DTC codes, it lacked essential functionality.
- **Conclusion**: This design was ultimately impractical due to complexity and lack of accessible DTC decoding.

Design 2: OBD Dongle (Wi-Fi) to ESP32 Microcontroller

- **What We Did**: In the second design, we switched to using a WiFi-enabled OBD dongle paired with an ESP32 microcontroller. This setup simplified the hardware somewhat and allowed us to successfully receive and decipher DTC codes. The

ESP32 connected to the OBD dongle over WiFi, and we used a local IP address to view the diagnostic data.

- **Results**: We achieved a functional setup, where the ESP32 could gather DTC codes from the OBD dongle, and we were able to decipher these codes. However, there were significant limitations: the user had to connect their phone to the ESP32's WiFi, which sacrificed internet connectivity, making the design inconvenient. Additionally, the DTC information could only be viewed on the local IP address, limiting accessibility.
- **Why It Didn't Work**: The requirement for the user to disconnect from their WiFi and connect to the ESP32 made the setup too cumbersome. While technically functional, this setup was too complex and unintuitive for a smooth user experience. Our plan to send DTC information from the ESP32 to the cloud required using the phone as a router, adding further complexity.
- **Conclusion**: While this design technically worked, the reliance on WiFi-based connection sacrificed usability and was too complicated for the average user.

Design 3: OBD Dongle (Bluetooth) to Mobile Phone

- **What We Did**: Our current approach is to use a Bluetooth OBD-II dongle that connects directly to the user's mobile phone, bypassing the need for additional hardware like the ESP32. This design is simpler and more user-friendly, allowing us to use the FixIt mobile app to gather DTC codes from the Bluetooth OBD dongle.
- **Results So Far**: This design simplifies the setup significantly and improves usability, as the user only needs to connect to the Bluetooth OBD dongle through the mobile app. We have successfully connected the Bluetooth OBD dongle to the phone but still need to implement DTC decoding within the mobile app. Currently, the DTC decoding logic exists in C for the ESP32, and we need to port this logic to JavaScript/React Native for use within the mobile app.
- **Why This Design Is Promising**: This setup is the most straightforward for users, requiring only the OBD dongle and the mobile app. It eliminates the need for intermediary hardware, making the solution more accessible and user-friendly. Additionally, the Bluetooth connection does not interfere with the user's internet connectivity, allowing for potential cloud integration without additional complexity.
- **Conclusion**: Although still a work in progress, this design appears to be the best solution for user needs, balancing functionality and simplicity. The primary challenge now is to adapt the DTC decoding logic for the mobile app.

Plans for Future Design and Implementation

1. **Porting DTC Decoding Logic to React Native**: Our immediate focus is on translating the existing DTC decoding logic from C (used on the ESP32) to JavaScript/React Native, allowing us to decode DTC codes directly within the mobile app. This will be essential for a seamless user experience, as users will see clear diagnostic information in the app without needing additional hardware.
2. **Implementing Cloud Connectivity for Advanced Analysis**: Once we achieve local DTC decoding in the app, we plan to integrate cloud connectivity. This will enable advanced analysis by sending DTC data to the cloud for AI-driven insights, expanding the diagnostic capabilities and providing users with more comprehensive information.
3. **User Interface and Experience Enhancements**: We will focus on improving the app's user interface, making it intuitive for users to view diagnostics, access additional insights, and manage their vehicle data. This includes designing a simple, visually appealing display for DTC codes and integrating user authentication to access personalized cloud features.
4. **Testing for Stability and Usability**: To ensure the app is reliable and easy to use, we will conduct extensive testing on various Android and iOS devices, checking for Bluetooth connectivity stability, responsiveness, and usability in real-world conditions.