# IVR Modernization Middleware – Milestone 2

---


Team-B Members

- Ramya Inavolu
- Seshwar Bheemineni
- Sujal Rane
- Pati Veera Surya Umanjani
- Thrupthi Chandana G
- Uma Maheswari Naidu
- Varshitha Kolla
- Joise S Arakkal
- Parasaram Neha Sri
- Alankrith
- Laasya

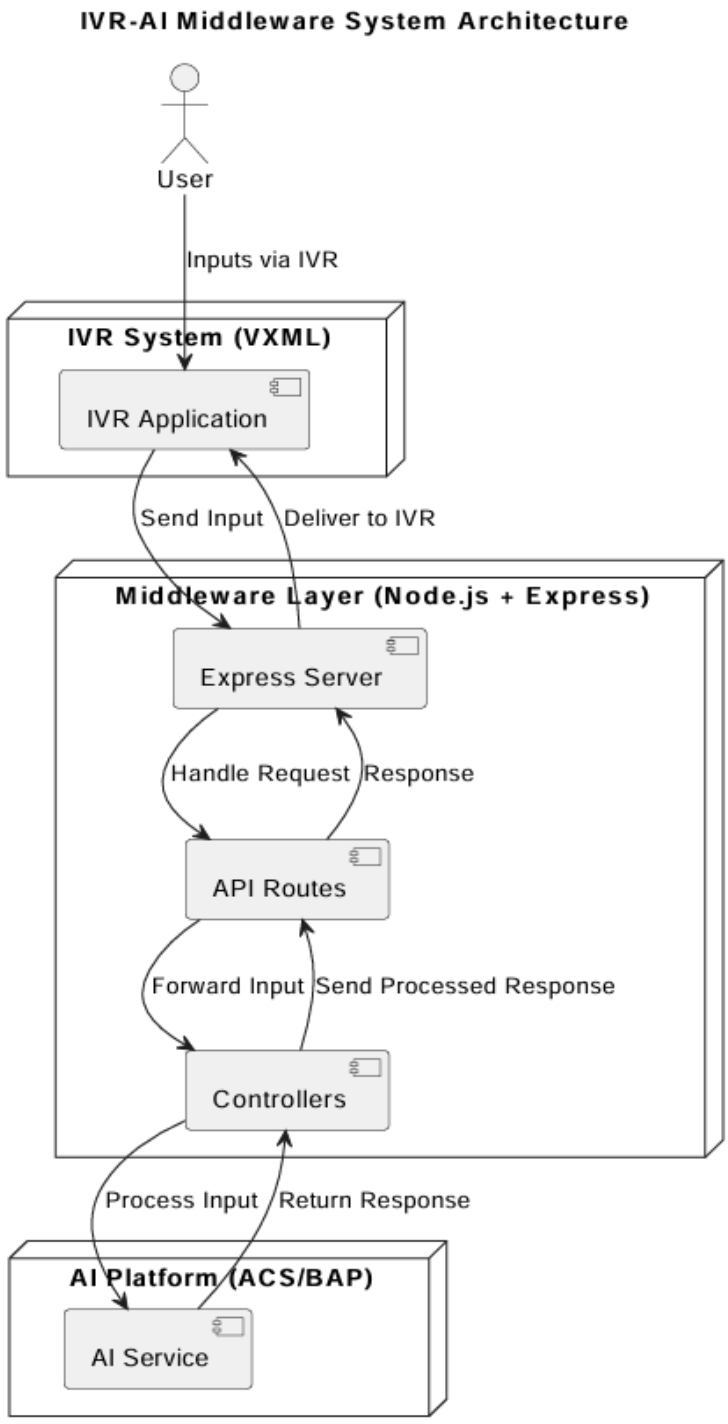

## Introduction

**Goal:** Modernize IVR with ACS/BAP

**Milestone 2:** Node.js middleware with mock ACS/BAP services.

## System Architecture

The architecture shows how a user interacts with an IVR system (VXML), which collects inputs and passes them to a Node.js middleware layer.
The middleware (Express server, routes, controllers) processes requests and forwards them to the AI platform (ACS/BAP).
The AI generates a response, which flows back through the middleware and IVR, delivering the output to the user.


IVR-AI Middleware System Architecture
User
Inputs via IVR
IVR System (VXML)
IVR Application
Send Input
Deliver to IVR
Middleware Layer (Node.js + Express)
Express Server
Handle Request
Response
API Routes
Forward Input
Send Processed Response
Controllers
Process Input
Return Response
AI Platform (ACS/BAP)
AI Service

# Setup

## System Setup (Installation Guide)

Prerequisites

- **Node.js** → Required to run middleware. Install from nodejs.org.
- **npm** → Installed automatically with Node.js, used for dependencies.
- **Thunder Client** → VS Code extension used to test APIs without Postman.

## Installation Steps

### Step 1: Clone the Repository

```
git clone
https://github.com/springboardmentor545-lgtm/TeamB-IVR-Modernization.git
cd TeamB-IVR-Modernization.git
```

### Step 2: Install Dependencies

```
cd TeamB-IVR-Modernization
npm install
```

### Step 3: Navigate to Middleware Project

1. Go inside the middleware project folder:

   ```
   C:\Users\adhik\Documents\Projects\TeamB-IVR-Modernization\middleware-pr
   oject
   ```

2. Check files are present:
   ```
   dir
   ```

3. Start the server:
   Using npm:

   ```
   npm start
   ```

Using nodemon:

```
npx nodemon index.js
```

**Step 4: Test API with Thunder Client**

- Open **Thunder Client** in VS Code → Click **New Request**
- Change method to **POST**
- Enter URL:
  http://localhost:3000/ivr/input
- Go to **Body** → **JSON** → Paste sample JSON

```
{
  "sessionId": "9876543210",
  "digit": "1"
}
```

- Click **Send**

**Step 5: Test BAP Route**

- Open **Thunder Client** → Create **POST Request**
- URL:
  http://localhost:3000/ivr/input
- Body → JSON:

```
{
  "sessionId": "9876543210",
  "digit": "3"
}
```

- Click **Send**

## Commands

```
npm install
npm start
```

Use **nodemon** for auto-reload when running the backend server.

## Folder Structure

```
middleware-project/
├── package.json
├── index.js              # Main entry point
├── /routes               # All API endpoints
│    └── ivrRoutes.js
│    └── acsRoutes.js
│    └── bapRoutes.js
├── /controllers          # Logic for routes
│    └── ivrController.js
│    └── acsController.js
│    └── bapController.js
├── /services             # Mock services (simulate ACS/BAP)
│    └── acsService.js
│    └── bapService.js
├── /docs
│    └── API.md           # API documentation
├── README.md
```

---

## System Setup – Execution Guide

Ensure **Node.js** and **npm** are installed
Check with:

```
node -v / npm -v
```

1. Navigate to project directory:
   `cd path/to/your/project-directory`

2. Install dependencies:
   `npm install`
3. Start middleware server:
   `nodemon index.js` or `npx nodemon index.js`
4. Server runs on: http://localhost:3000/

## Sample URLs & Output

**Sample URL:** http://localhost:3000/ivr/input
**Output Example:**

```
{"status": "success", "message": "IVR input received", "input": "User pressed 1"}
```

**Sample URL:** http://localhost:3000/acs/response
**Output Example:**

```
{"status": "success", "message": "ACS response processed", "response_code": 200}
```

## API Documentation – IVR Input Endpoint

**Method:** POST
**URL:** http://localhost:3000/ivr/input

**Description**
Main entry point for IVR requests. Validates user input and routes it to ACS or BAP services.

**Request Body Example:**

```
{
  "sessionId": "abc123",
  "digit": "1"
}
```

| Field | Type | Required | Description |
|---|---|---|---|
| sessionId | String | Yes | A unique identifier for the user's call session. |
| digit | String | Yes | The digit pressed by the user on the IVR (Interactive Voice Response) system. |

## Success Responses

**Case 1: Digit = "1" (Balance Inquiry)**

```
{
  "sessionId": "abc123",
  "response": "Your account balance is $500"
}
```

**Case 2: Digit = "2" (Agent Transfer)**

```
{
  "sessionId": "abc123",
  "response": "Transferring you to an agent"
}
```

#### Case 3: Digit = "3" (Payment Service)

```
{
  "sessionId": "abc123",
  "response": "Payment service is currently active"
}
```

### Error Responses

#### Case 1: Invalid Digit

```
{ "error": "Invalid option selected" }
```

#### Case 2: Missing Parameters

```
{ "error": "Missing sessionId or digit" }
```

#### Case 3: Internal Server Error

```
{ "error": "Something went wrong" }
```

## Overview of /acs/process Endpoint

**Method:** POST

**URL:** http://localhost:3000/acs/process

**Description:**

Handles ACS-specific logic for account balance inquiry and agent transfer

#### Request Body

```
{
  "sessionId": "abc123",
  "userInput": "1",
```

```
  "channel": "voice"
}
```

**Successful Response Example:**

```
{
  "status": "success",
  "message": "Command processed successfully",
  "nextAction": "playAudio",
  "audioFile": "welcome.wav"
}
```

**Standard Error Status Codes:**

- **400** – Bad Request
- **401** – Unauthorized
- **404** – Not Found
- **429** – Too Many Requests
- **500** – Internal Server Error

# ACS Endpoints

The ACS  integration acts as middleware that allows legacy IVR systems (built on VXML) to interact with modern conversational AI platforms. ACS is implemented as a mock service to simulate request–response flows between the IVR and ACS.

The implementation follows a layered architecture with three main components :

**Routes, Controller, and Service.**

## acsRoutes.js

- Defines the API endpoints that external systems, such as IVR, can call.
- Directs incoming requests (e.g., /acs/response) to the appropriate controller.
- Contains no logic other than path definitions.
- Analogy: Functions like a reception desk that directs requests to the correct department.

### acsController.js

- Serves as the decision-making layer between routes and services.
- Validates incoming data and ensures it is correctly structured.
- Forwards valid requests to the service layer.
- Ensures a consistent JSON response format.
- Analogy: Works like a manager who checks documents, forwards them to the right team, and communicates the result.

### acsService.js

- Simulates ACS behavior and generates mock responses for milestone testing.
- Processes the request using the provided sessionId and userInput.
- Returns a structured response including a platform identifier (platform: "ACS") to distinguish ACS replies from other systems.
- Analogy: Operates like a processing machine that takes in input and delivers well-formatted output.

## Example Flow

i. IVR sends a request:
{ "sessionId": "12345", "userInput": "press 1 for balance" }
ii. acsRoutes.js directs the request to acsController.js.
iii. acsController.js validates inputs and passes them to acsService.js.
iv. acsService.js returns a mock ACS response

```
{
    "platform": "ACS",
    "sessionId": "12345",
    "received": "press 1 for balance",
    "message": "ACS processed: press 1 for balance"
}
```

v .The final structured JSON response is delivered back to the IVR

**Error Handling**

To ensure clarity and maintainability, the `/acs/response` API returns well-defined error payloads for every major error scenario. Each error response uses a consistent JSON structure with clear status codes.

**Standard Error Status Codes**

- **400** **Bad Request:** Invalid input provided.
- **401** **Unauthorized:** Authentication failure.
- **404** **Not Found:** Session or resources missing.
- **429** **Too Many Requests:** Rate limit exceeded.
- **500** **Internal Server Error:** Unexpected middleware or ACS failure.

**JSON Error Structure**

Each error response includes:

- `code`: An error code string.
- `message`: A human-readable description.
- `details`: Additional context or advice.

**Sample Error JSONs:**

```
{
    "errorCode": "INVALID_INPUT",
    "message": "The digit provided is not supported.",
    "docsLink": "https://api.docs/errors#INVALID_INPUT"
}
```

The `/acs/response` endpoint ensures smooth interaction between the middleware and ACS by handling user inputs efficiently. With clear request/response structures and robust error handling, it provides reliability, consistency, and ease of integration for future enhancements.

# Project Contribution

| Section | Person |
|---|---|
| Project Demo | Alankrith |
| Title & Team Details + Formatting | Joise S Arakkal |
| Objective of Milestone 2 + Introduction | Thrupthi Chandana G |
| Architecture Diagram | Seshwar Bhemineni |
| Testing Middleware | Varshitha Kolla |
| System Setup (Installation Guide) | Uma Maheshwari Naidu |
| System Setup (Execution Guide) | Pati Veera Surya Umanjani |
| API Documentation – IVR Endpoints | Parasaram Neha Sri |
| API Documentation – ACS Endpoints | Ramya Inavolu |
| Team Task Division + Challenges | Byreddy Lasya Sre Reddy |
| Learnings + Conclusion + Future Scope | Sujal Rane |

# Challenges

- Setting up Express.js server with proper routes
- Handling Axios API calls & managing async responses
- Debugging CORS issues
- Structuring project into modular format
- Managing error handling for failed API requests

## Learnings

- Understood how middleware connects and processes requests
- Importance of modular structure for scalability
- Gained deeper understanding of async programming
- Learned to configure CORS
- Improved collaboration using Git

# Conclusion

Through the development of the middleware for IVR Modernization, our team gained both technical and collaborative learnings that proved highly valuable. On the technical side, we understood the crucial role of middleware as the bridge between legacy IVR systems built on VXML and modern conversational AI platforms such as ACS and BAP. The project reinforced the importance of well-defined API communication, where structured request and response formats ensure seamless interoperability.

Working with **Node.js and Express** gave us hands-on experience in asynchronous programming, routing, and modular application design. By organizing the project into routes, controllers, and services, we ensured maintainability and scalability. The use of **Axios** for API calls highlighted the importance of error handling, retries, and resilience in systems that depend on external services. Testing with **Thunder Client** helped us validate not only the correctness of the API flows but also the importance of capturing both success and error scenarios to improve robustness.

Collaboration was another major learning. By dividing the middleware into modules, each team member contributed meaningfully to the project while also understanding how their part interacted with the larger system. This improved our coordination, communication, and collective problem-solving skills. We realized that building middleware is not only a coding task but also about aligning multiple perspectives into a cohesive solution.

In terms of outcomes, Milestone 2 successfully demonstrated that requests from IVR can be routed through the middleware, processed by mock ACS/BAP services, and returned with meaningful responses. This validated our design and proved that modernization of legacy IVR systems is achievable without replacing existing infrastructure. By the end of this milestone, we had a working middleware with modular architecture, mock endpoints, and clear API documentation — establishing a strong foundation for further development.

Looking forward, the next milestone will involve integrating with **real ACS/BAP platforms**, which will introduce authentication, live conversational flows, and stronger security requirements. Performance and scalability will become a priority, ensuring that the middleware can handle production-level workloads with low latency. We also aim to enhance error handling, add monitoring and logging features, and strengthen dialogue flow mapping for more natural user experiences. In the long run, this middleware framework can evolve into a versatile solution that supports multiple AI platforms, enabling enterprises to modernize their IVR systems with flexibility and minimal redevelopment.

Overall, this milestone has not only validated the feasibility of IVR modernization via middleware but also enriched our understanding of real-world system integration. It has given us the confidence and technical grounding to move toward a production-ready solution in the upcoming phases.