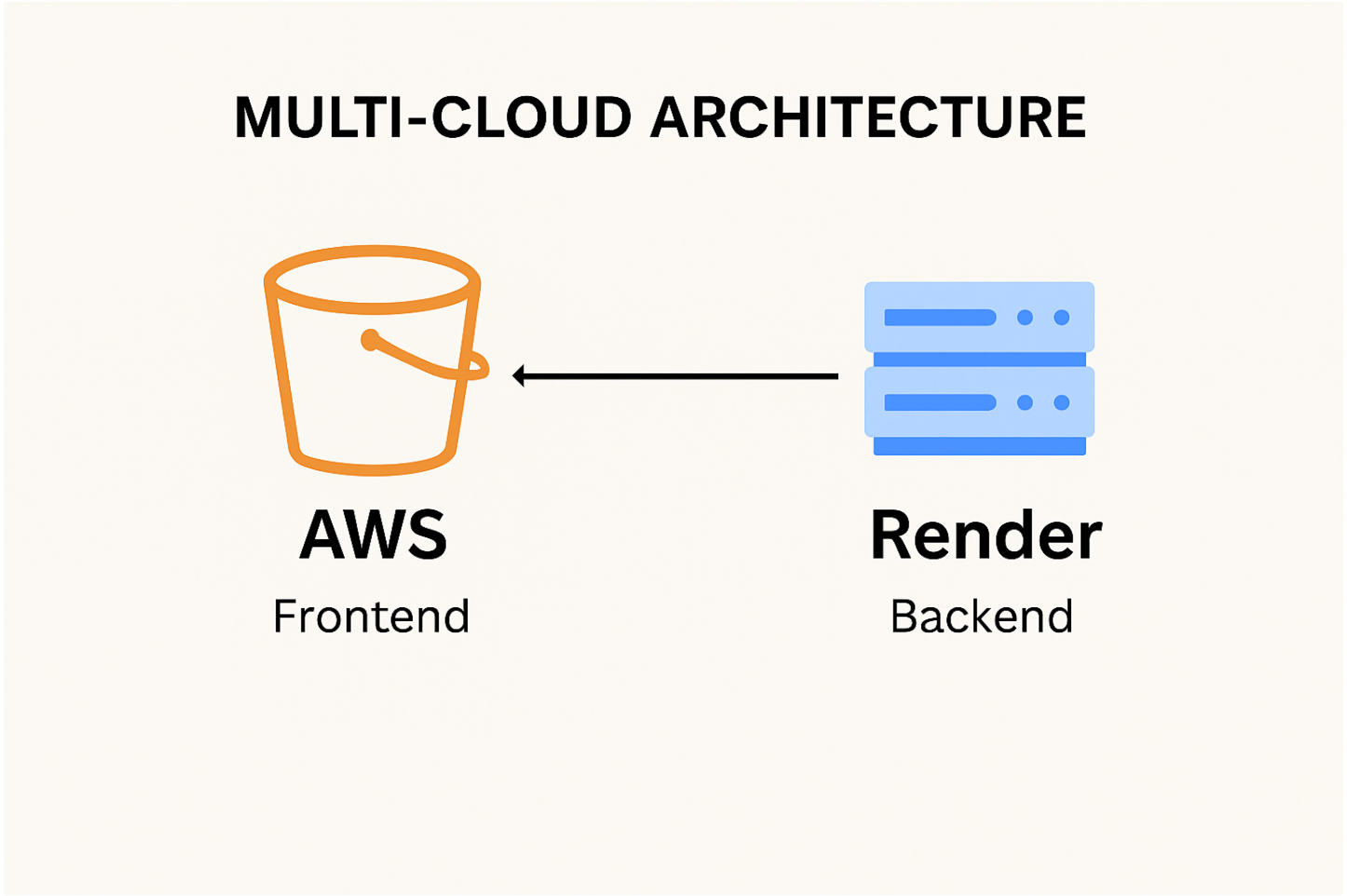
Multi-cloud architecture



Frontend Code

```
<!DOCTYPE html>
<html>
<head>
  <title>Multi-Cloud Site</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Enter Your Name</h1>
  <input type="text" id="nameInput" />
  <button onclick="sendName()">Submit</button>
  <p id="response"></p>

  <script>
    async function sendName() {
      const name = document.getElementById("nameInput").value;
         const  res  =  await  fetch('https://backend-ieii.onrender.com/api/hello?name='  +
name);
      const data = await res.text();
      document.getElementById("response").innerText = data;
    }
  </script>
</body>
</html>
```

Backend Code

```
const express = require('express');
const cors = require('cors');
const app = express();

app.use(cors());

app.get('/api/hello', (req, res) => {
  const name = req.query.name || 'Guest';
  res.send('Hello, ' + name + '! This is from the backend.');
});

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log('Server running on port ' + port);
});
```

Multi-cloud Architecture

IN THIS TASK, I DESIGNED AND IMPLEMENTED A MULTI-CLOUD ARCHITECTURE USING TWO DIFFERENT CLOUD PLATFORMS: AMAZON WEB SERVICES (AWS) AND RENDER. THE OBJECTIVE WAS TO DEMONSTRATE HOW A REAL-WORLD APPLICATION CAN BE DEPLOYED BY DISTRIBUTING RESPONSIBILITIES BETWEEN TWO INDEPENDENT CLOUD PROVIDERS. THIS KIND OF SETUP IMPROVES AVAILABILITY, SCALABILITY, AND REDUNDANCY, WHILE ALSO AVOIDING VENDOR LOCK-IN.

FOR THE FRONTEND, I USED AWS S3 (SIMPLE STORAGE SERVICE) TO HOST A STATIC WEBSITE. I CREATED A BUCKET, CONFIGURED IT FOR WEBSITE HOSTING, AND UPLOADED THE index.html AND style.css FILES. I SET THE PERMISSIONS TO MAKE THE CONTENT PUBLICLY ACCESSIBLE AND OBTAINED A BUCKET ENDPOINT THAT USERS CAN VISIT TO ACCESS THE WEBSITE. THE FRONTEND PAGE ALLOWS USERS TO INPUT THEIR NAME AND SUBMIT IT.

FOR THE BACKEND, I USED RENDER, A CLOUD PLATFORM KNOWN FOR SIMPLE BACKEND HOSTING WITHOUT COMPLEX BILLING SETUPS. I DEVELOPED A SIMPLE NODE.JS EXPRESS SERVER THAT EXPOSED AN API ENDPOINT (/api/hello). WHEN A NAME IS SENT TO THIS ENDPOINT, IT RESPONDS WITH A GREETING MESSAGE. I HOSTED THIS BACKEND ON RENDER, OBTAINED A PUBLIC BACKEND URL, AND INTEGRATED THIS INTO THE FRONTEND CODE USING JAVASCRIPT'S fetch() FUNCTION.

THE MULTI-CLOUD INTERACTION WORKS AS FOLLOWS: WHEN A USER OPENS THE WEBSITE HOSTED ON AWS S3, THEY SEE A SIMPLE UI WITH A TEXT INPUT AND BUTTON. WHEN THE USER ENTERS A NAME AND CLICKS THE BUTTON, A REQUEST IS SENT FROM THE FRONTEND (AWS) TO THE BACKEND HOSTED ON RENDER. THE BACKEND PROCESSES THE REQUEST AND SENDS A RESPONSE BACK, WHICH IS THEN DISPLAYED TO THE USER ON THE SAME PAGE.

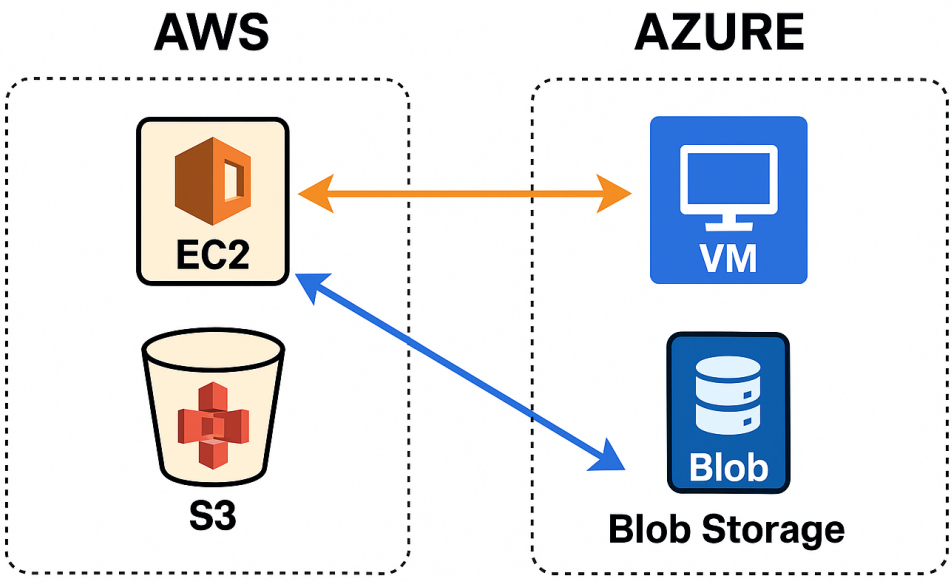
THIS SETUP SUCCESSFULLY DEMONSTRATES HOW TWO SEPARATE CLOUD PLATFORMS CAN BE INTEGRATED TO FUNCTION AS A SINGLE APPLICATION. ONE CLOUD HANDLES

Multi-cloud Architecture

THE STATIC UI DELIVERY, WHILE THE OTHER MANAGES DYNAMIC BACKEND LOGIC. TO MAKE THIS WORK, I HAD TO ENSURE THE BACKEND URL WAS PUBLICLY ACCESSIBLE, AND THAT THE FRONTEND CODE REFERENCED THE CORRECT ENDPOINT. I ALSO TESTED CROSS-ORIGIN REQUESTS TO ENSURE THEY FUNCTIONED WITHOUT CORS ERRORS, WHICH ARE COMMON IN MULTI-CLOUD INTEGRATIONS.

THE ARCHITECTURE IS DOCUMENTED WITH A SIMPLE DIAGRAM THAT I CREATED, VISUALLY REPRESENTING THE FLOW OF USER INPUT FROM AWS TO RENDER AND BACK. I UPLOADED ALL PROJECT FILES TO MY GITHUB REPOSITORY UNDER THE TASK-3 FOLDER, INCLUDING HTML, CSS, BACKEND CODE, AND THE ARCHITECTURE IMAGE.

OVERALL, THIS TASK GAVE ME PRACTICAL INSIGHT INTO HOW FRONTEND AND BACKEND SERVICES CAN BE SEPARATED AND HOSTED ON DIFFERENT CLOUDS TO ACHIEVE BETTER DISTRIBUTION AND SCALABILITY. WHILE THIS WAS A BASIC DEMO, IT LAID THE FOUNDATION FOR MORE COMPLEX MULTI-CLOUD DEPLOYMENTS IN REAL-WORLD PROJECTS.



Multi-Cloud Architecture