Ambitious Solutions

Tech Feasibility Final

Altored Health

Team Members:

Jasmine Flowers

Ian Nieto

John Gornick

Jerry “Tre” Kelly

Client: Jesslynn Stull

Mentor: Bailey Hall

Table of Contents

1. Introduction

1.1 The Big Picture

Healthcare pricing in the United States is notoriously complex, inconsistent, and often opaque. Patients rarely know the true cost of a procedure until months after it has been performed, leaving them with unexpected bills and little ability to plan financially. This lack of transparency makes it difficult for individuals and families to identify affordable, high-quality care, and often results in patients paying far more than necessary. The blurred lines between provider charges, insurer reimbursements, and patient responsibilities lead to confusion, frustration, and in many cases, financial hardship. By addressing this lack of clarity, we aim to empower patients with the information they need to make informed healthcare decisions and reduce the stress associated with unexpected medical costs.

1.2 The Problem

The current strategies to find affordable healthcare are very limited. Patients in need of care are uninformed if they are being overcharged, or in some cases, don't even know what they are being charged for. The medical field needs transparency and community when it comes to pricing, and that is where our sponsor, Jesslynn Armstrong, comes in. Jesslynn is a local entrepreneur and the current Director of Operations at NAU's Venture Studio. She has priceless experience and expertise in running and managing startup companies. After her own personal struggles with the healthcare industry, she was inspired to create Altored Health.

1.3 The Solution

Our Solution is to create a web application that centralizes pricing from various healthcare providers. This web application will display the prices of various operations and procedures, like MRIs, as well as insurance copays from a handful of popular insurers. Key features of our application will include centralized operation pricing, centralized copay pricing, medical bill analysis, and peer-reviewed community sections where users can upload their own processes and experiences.

The healthcare pricing data for our application will be sourced from a centralized database containing comprehensive cost information across various procedures and providers. This information will then be dynamically displayed on the website through an intuitive and user-friendly interface. To ensure feasibility and controlled growth, our initial implementation will focus exclusively on healthcare providers within Arizona. Once the system is proven to function reliably in this limited scope, we will gradually expand coverage to additional states and regions, allowing the platform to scale in a manageable and structured manner.

Similarly, insurance-related data will be retrieved from a dedicated database designed to house key plan details and cost structures. In addition to these static records, our system will integrate user-submitted information, enabling customization of results based on individual circumstances such as insurance provider, plan type, or coverage level. By combining database-driven information with real-time user input, the application will deliver accurate, tailored pricing transparency that empowers patients to make informed healthcare decisions.

We also aim to implement a medical bill analysis feature in the future. This functionality will allow users to upload their medical bills, which the system will scan to identify potential overcharges or inconsistencies. In addition to highlighting questionable charges, the tool may assist users in disputing their bills, providing them with valuable support in managing their healthcare costs.

The application should also serve as a community-based platform, where users can upload and share their experiences with different providers and insurers. By fostering a peer-driven knowledge base, the platform will enable patients to learn from one another, promote transparency, and build trust in the healthcare system. This document will analyze the technical challenges we will face and our strategies to overcome these challenges.

With our needs for our application, we will have to solve some technical challenges. Finding solutions to these challenges will help us understand what technologies will help us build a better product. Afterward,s we will be showing you the solutions we have for these technical challenges. Last ly we will show our process in integrating these technologies into our product.

2. Technological Challenges

2.1 Activating our services in a live deployable setting

In order to create a live web app and decide on what services we can use, we need to choose a service provider to host it. Such a service provider should have a good set of features that enable our app to do what it needs to do. A very important piece of this would be security and compliance standards in the health industry.

2.2 Scraping, Parsing, and Displaying Data

The challenge we face is to parse, scrape, and display hospital pricing data in our application without compromising speed or utility. To achieve this, we first need a robust data parser capable of handling multiple file formats, such as .xlsx, .csv, and .json. This parser must filter out unnecessary or redundant information from the extensive hospital pricing files and then output a clean, structured .json file. Having the data in a consistent and simplified format will make it scrapable and easily readable for subsequent processing. Once the data has been cleaned and converted into the .json format, the next step is to scrape and extract the relevant pricing information. This involves selecting or building a tool that can reliably process the structured data, pulling out the necessary values, and preparing them for display within our application.

Ultimately, the success of this process depends on finding a method that is efficient, accurate, and well-suited to integrate with the technologies driving our application's front end.

2.3 A clean, responsive User Interface

The challenge we face is implementing a clean and responsive user interface that allows users to easily navigate the application. To achieve this, we will need to apply CSS styling to ensure that the interface has a polished and finished look, making it visually appealing and user-friendly. In addition to styling, we must incorporate a framework such as React or a similar technology to provide a responsive front end that delivers speed and efficiency. This combination will allow us to balance design quality with performance, ensuring that the user experience remains smooth even as the application scales in size and complexity.

2.4 A storage medium for storing large amounts of line-by-line data

We need a storage medium, such as a database to store our line-by-line code. It should be stored in a way that is searchable, which will allow users to find healthcare data. Additionally, there should be good security by default to work with healthcare standards.

3. Technology Analysis

Introduction

The main challenge in this project is implementing and deploying a user-based web application. Developing this application requires a front-end team and a back-end team, which we have divided ourselves into. This application will be accessible through web browsers, which will offer accessibility to users. To use our application's features, users will need to create an account on our platform. An account can be created with an email address/ phone number and a password.

3.1 Powerful and scalable backend runtime

3.1.1 Introduction

When we create our web app, the backend is really the deciding factor on a large portion of the technologies, and for proper industry applications, there are only a couple of options. The two main ways are Azure and Amazon Web Services, the decision between the two will dictate what sets of services we have to choose from, and more directly, what backend we will be using.

3.1.2 Desired Characteristics

As we create web software, the main things we are looking for our client in this situation are upfront cost, cost analysis and support. Forthe upfront cost, the option we choose should be as cheap as possible, especially during the design phase, as the product will not be making money during this phase. Cost analysis is a bit different, as it means that it's very clear what our client will pay for the web services. For support, we need some guidance into the relevant systems, which can generally come from online services or people we know.

Secondarily, we would want it to perform well over most regions and have good security with its compliance standards. Performance and region compatibility are important, although we really only need this in the U.S, as this will be a U.S healthcare app. Lastl,y compliance standards are important, as if we deal with sensitive healthcare dat,a then we need it to be secure to follow the law properly.

3.1.3 Alternatives

3.1.3.1 Amazon Web Services

Amazon Web Services has existed for quite some time now and is the most popular industry standard; they have the most servers overall globally and perform quite well in most markets. Additionally, they have many compliance standards with the U.S governmen,t which is the main need since our product is designed for American healthcare issues. As a final note, cost can be very confusing in AWS; they charge based on the services you use, and there are many things that can change without the user knowing that they may charge you for.

3.1.3.2 Microsoft's Azure

Microsoft's Azure has been much more popular recently due to some powerful proprietary features Azure has, which we will be discussing in the other sections of this tech analysis. Microsoft's services are extremely powerful in U.S markets but may struggle in some global markets, which is generally fine for us as we will likely always be the only U.S.-based Application. The U.S compliance standards that Azure has are slightly greater than those of Amazon's. Like AWS, Azure has similar issues understanding where the costs may change; things like server changes and data size changes can immediately charge you extra without any notification.

3.1.4 Analysis

We found most of this information from the documentation on both services, and with some previous experience with both services from the group. With the capabilities of both being quite powerful, the decision is quite close. Microsoft does have more compliance standards than AWS, although Amazon's web services are older and a bit more mature. The main thing that sways our decision is cost and support. Our client has recently been given a grant from Microsoft Azure to help with development, and our client has technical support from colleagues familiar with Azure.

3.1.5 Chosen Approach

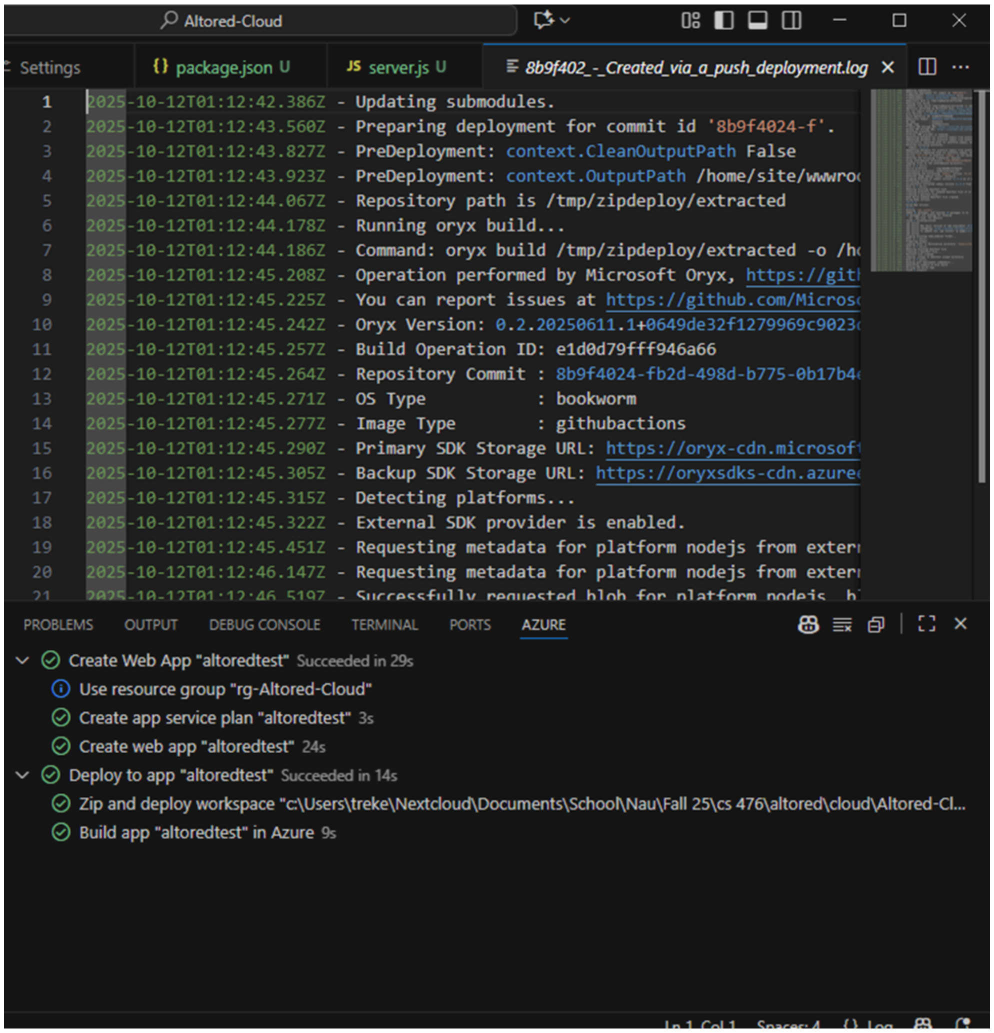
While our project could be programmed and work well in Azure or Amazon Web Services, we will be going with Azure. The main factors that differ for us are the startup cost and client support. Although security and regional support are very important, both services have basically identical performance in those areas. With the extra support and financing our client has with choosing Azure it has clear benefits over AWS. This choice does affect what other technologies we have access to, so it heavily influences our other tech decisions.

Cloud Computing Platforms

CRITERIA	Azure	Amazon Web Services
Compliance Standards	5/5 HIPAA, GDPR, SOC, ISO, FedRAMP; has specific government & healthcare offerings	4/5 HIPAA, GDPR, SOC, ISO, FedRAMP
Region Compatibility	4/5 Great U.S Region Compatibility	5/5 Great Worldwide Region Compatibility
Region Speed	5/5 Very High Speed in the U.S	5/5 Very high Speed in many worldwide regions
Cost Analysis	1/5 Confusing and hard to track	1/5 Also Confusing and hard to track
Upfront Cost	5/5 Startup Funding for our client	3/5 No Startup funding for out client
Team Support	5/5 Client has contacts for support with Azure and azure has good documentation	2/5 No contact for support with AWS although AWS is also well documented.
Total	25/30	20/30

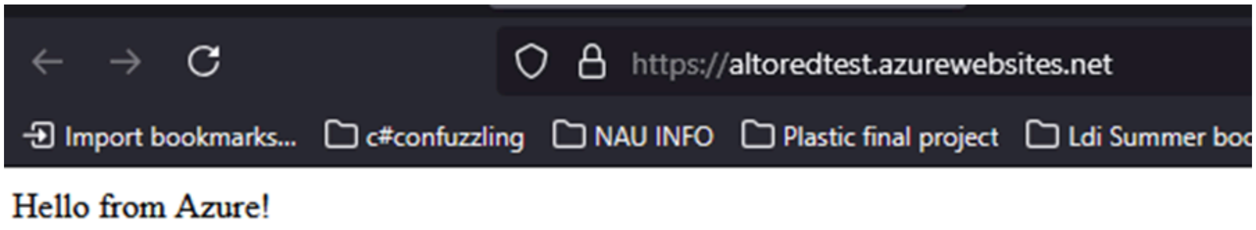
3.1.6 Proof of feasibility

We have created some early tests of Azure's backend to run our main software suite, and from the early tests, we can see that it should work well. We managed to run some test code in C through Azure's web app system, and you can see the deployment in the images below. The first image shows the deployment, and the second shows the web functionality of the code. In the future thoug,h the web app will have a secondary system that takes the info from the backend web app to display in a well-formatted way for users.



```
2025-10-12T01:12:42.386Z - Updating submodules.
2025-10-12T01:12:43.560Z - Preparing deployment for commit id '8b9f4024-f'.
2025-10-12T01:12:43.827Z - PreDeployment: context.CleanOutputPath False
2025-10-12T01:12:43.923Z - PreDeployment: context.OutputPath /home/site/wwwroo
2025-10-12T01:12:44.067Z - Repository path is /tmp/zipdeploy/extracted
2025-10-12T01:12:44.178Z - Running oryx build...
2025-10-12T01:12:44.186Z - Command: oryx build /tmp/zipdeploy/extracted -o /ho
2025-10-12T01:12:45.208Z - Operation performed by Microsoft Oryx, https://gith
2025-10-12T01:12:45.225Z - You can report issues at https://github.com/Microso
2025-10-12T01:12:45.242Z - Oryx Version: 0.2.20250611.1+0649de32f1279969c9023
2025-10-12T01:12:45.257Z - Build Operation ID: e1d0d79fff946a66
2025-10-12T01:12:45.264Z - Repository Commit : 8b9f4024-fb2d-498d-b775-0b17b4
2025-10-12T01:12:45.271Z - OS Type           : bookworm
2025-10-12T01:12:45.277Z - Image Type        : githubactions
2025-10-12T01:12:45.290Z - Primary SDK Storage URL: https://oryx-cdn.microsof
2025-10-12T01:12:45.305Z - Backup SDK Storage URL: https://oryxsdks-cdn.azure
2025-10-12T01:12:45.315Z - Detecting platforms...
2025-10-12T01:12:45.322Z - External SDK provider is enabled.
2025-10-12T01:12:45.451Z - Requesting metadata for platform nodejs from exter
2025-10-12T01:12:46.147Z - Requesting metadata for platform nodejs from exter
2025-10-12T01:12:46.519Z - Successfully requested blob for platform nodejs  h
```

```
Create Web App "altoredtest" Succeeded in 29s
  Use resource group "rg-Altored-Cloud"
  Create app service plan "altoredtest" 3s
  Create web app "altoredtest" 24s
Deploy to app "altoredtest" Succeeded in 14s
  Zip and deploy workspace "c:\Users\treke\Nextcloud\Documents\School\Nau\Fall 25\cs 476\altored\cloud\Altored-Cl...
  Build app "altoredtest" in Azure 9s
```



Hello from Azure!

3.2 Application of a simple, readable format for deploying cloud resources

3.2.1 Introduction

While deploying resources manually through our cloud service is an option, it is often clunky and not streamlined. The sheer quantity of data, different permissions, health data privacy concerns, and other services we will require means that how we deploy these resources will be deeply fundamental to our work process.

3.2.2 Desired Characteristics

The method we choose to deploy resources should have a few essential qualities. Price is the first and most important one. One of our main concerns will be how our technology meshes with the cloud service that we use. For a small project without a ton of funding from the start, a high price tag on simple resource deployment will not be feasible. Simplicity and flexibility will be our other priorities. Simplicity will allow for great levels of accessibility and speed, while flexibility will allow us to work however suits us, not the technology, and allow us to stay efficient.

3.2.3 Alternatives

3.2.3.1 Terraform

From information found on Reddit threads and other computer science-related forums, users often compare Terraform and Bicep as the two main technologies to perform these tasks. Terraform is a service that deploys resources to multiple cloud services. Developed by HashiCorp, this is the most common service for multi-service users, as well as anyone desiring greater integration with providers.

3.2.3.2 Bicep

From information found both through forums and Microsoft's official documentation, Bicep is often considered something of a gold standard for Azure applications. Its largest drawback seems to be that it cannot interact with multiple cloud services, instead focusing solely on one. Bicep was developed by Microsoft and has been out for less time.

3.2.4 Analysis

From information published by both Microsoft and Hashicorp, the similarities and differences between the two seem well documented and often compared. By viewing both Microsoft's official comparison as well as the opinions of many users online, we conclude that Terraform has many advantages over Bicep, but they primarily pertain to compatibility with other cloud services. Terraform is slightly more complex and less supported by Azure itself. Bicep does not have nearly the integration capabilities; however, it is supported more heavily by Microsoft, and is simpler to use.

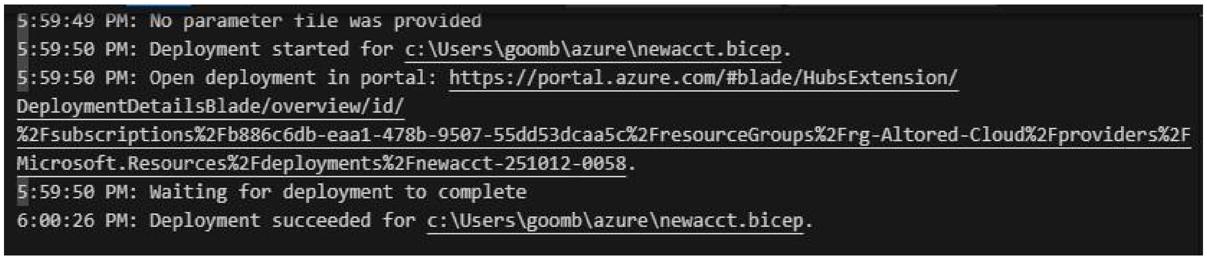
3.2.5 Chosen approach

While both Terraform and Bicep provide free technologies to assist us in what we would need, Bicep has been chosen as the technology we will move forward with. This is primarily due to three factors. Bicep is more heavily supported by Microsoft, giving it a home-field advantage when it comes to Azure support. Bicep also seems to manage more for the user, increasing efficiency. The final reason is related to external factors, such as our team's familiarity with Bicep. The table below shows how each technology performed in our prioritized categories.

CRITERIA	Bicep	Terraform
Cost	5/5	5/5
Azure Compatibility/Integration	5/5	4/5
Flexibility	4/5	3/5
Familiarity	3/5	1/5
Totals	**17/20**	13/20

3.2.5 Proof of feasibility

This image is proof of a successful deployment of a resource group to Azure. This technology has been confirmed to be functional and easily workable.



```
5:59:49 PM: No parameter file was provided
5:59:50 PM: Deployment started for c:\Users\goomb\azure\newacct.bicep.
5:59:50 PM: Open deployment in portal: https://portal.azure.com/#blade/HubsExtension/
DeploymentDetailsBlade/overview/id/
%2Fsubscriptions%2Fb886c6db-eaa1-478b-9507-55dd53dcaa5c%2FresourceGroups%2Frg-Altored-Cloud%2Fproviders%2F
Microsoft.Resources%2Fdeployments%2Fnewacct-251012-0058.
5:59:50 PM: Waiting for deployment to complete
6:00:26 PM: Deployment succeeded for c:\Users\goomb\azure\newacct.bicep.
```

3.3 Scraping Large Amounts of Data

3.3.1 Introduction

One of the most important aspects of this project is scraping data from very large datasets. This section will outline the characteristics we are seeking when evaluating different scraping technologies. We will do this by comparinmg available options and explaining the reasoning behind our chosen approach. To ensure a structured evaluation, we analyzed both technologies using seven key criteria: speed, reliability, scalability, licensing cost, security, ease of integration, and community support. These criteria will also be used to guide the rest of our front end. All findings presented here are based on official documentation, industry forums, and previous experience that our team has.

3.3.2 Desired Characteristics

The ideal scraping technology for our application should efficiently process large datasets with minimal overhead, and should be able to do this while maintaining reliability across runs. We view speed as how quickly the scraper can collect and process data when operating concurrently. When a user wants to use our app, we do not want them to have to wait. Reliability ensures that the system handles retries, manages failed requests, and produces consistent results. We also want the system to be scalable, which will be very important when we need to add more healthcare providers. Licensing cost must be minimal or free to support a sustainable open-source architecture. Security involves ensuring that the framework operates safely within sandboxed environments and follows best practices for protecting data. Ease of integration reflects how seamlessly the scraper can fit into our existing Python stack and pipeline system. Lastly, community support determines the availability of documentation, forums, and long-term maintenance to ensure stability over time.

3.3.3 Alternatives

To evaluate potential scraping technologies, we focused on two primary Python-based tools: Scrapy and Playwright. Scrapy is a framework designed specifically for high-speed and large-scale web scraping. Playwright is primarily used for browser automation and dynamic page rendering. Both tools were evaluated using the seven criteria outlined in the above subsection to determine their suitability for collecting and processing large hospital pricing datasets.

3.3.4 Analysis

The team concluded that Scrapy is better suited for the project's data collection needs. Scrapy provides efficient concurrency management and stays lightweight. We also have had previous experience with Scrapy in personal projects. Although Playwright does excel at automating browsers and executing JavaScript-heavy pages, it has additional overhead that makes it less efficient for large-scale crawling. Scrapy is fast, lightweight, and highly scalable.

3.3.5 Chosen Approach

We will proceed with Scrapy as the primary scraping framework. Scrapy offers a scalable and well-documented environment for parsing large XSLX and CSV datasets. Its architecture supports high-performance crawling and simplifies error handling. This approach ensures that our backend can efficiently collect, normalize, and update hospital pricing data with accuracy and speed while maintaining security and scalability. Scrapy's open-source licensing and active community further support our goals for maintainability and extensibility.

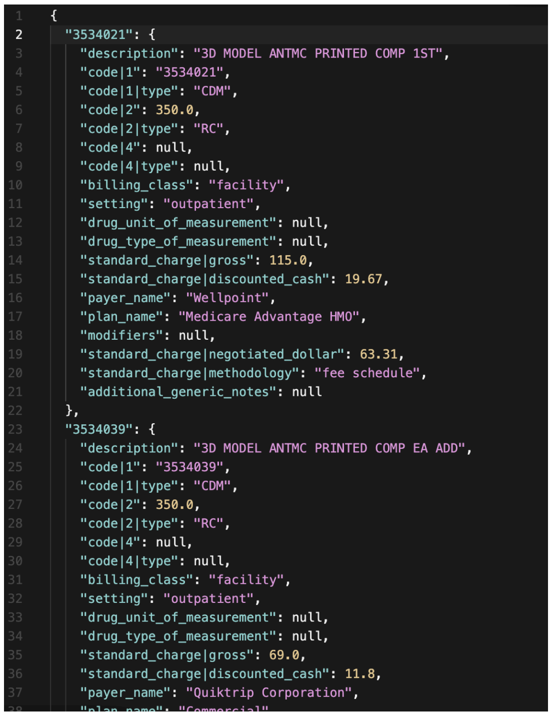
Scraping

CRITERIA	Scrapy	Playwright
Speed	4/5 Made for concurrent crawling with throttling.	3/5 Brower adds overhead.
Reliability	5/5 Built-in retries and pipelines	5/5 Executes JavaScript accurately
Scalability	5/5 Built for large-scale crawls	3/5 Heavy for large deployment
Licensing Cost	Free	Free
Security	4/5 Secure browser sandbox	3/5 Secure browser sandbox, but harder to navigate
Ease of Integration	5/5 Pipelines fits Python stack	3/5 Needs Node or bindings
Community Documentation	5/5 Long-standing Docs	4/5 Growing rapidly
Total	**29/30**	21/30

3.3.6 Proof of Feasibility

Our team has already conducted proof-of-concept tests using Scrapy to parse large XSLX datasets into normalized JSON files. These early results confirm that Scrapy meets our functional and performance requirements, ensuring the feasibility of full-scale deployment.

Output of parsed data:



```
{
  "3534021": {
    "description": "3D MODEL ANTMC PRINTED COMP 1ST",
    "code|1": "3534021",
    "code|1|type": "CDM",
    "code|2": 350.0,
    "code|2|type": "RC",
    "code|4": null,
    "code|4|type": null,
    "billing_class": "facility",
    "setting": "outpatient",
    "drug_unit_of_measurement": null,
    "drug_type_of_measurement": null,
    "standard_charge|gross": 115.0,
    "standard_charge|discounted_cash": 19.67,
    "payer_name": "Wellpoint",
    "plan_name": "Medicare Advantage HMO",
    "modifiers": null,
    "standard_charge|negotiated_dollar": 63.31,
    "standard_charge|methodology": "fee schedule",
    "additional_generic_notes": null
  },
  "3534039": {
    "description": "3D MODEL ANTMC PRINTED COMP EA ADD",
    "code|1": "3534039",
    "code|1|type": "CDM",
    "code|2": 350.0,
    "code|2|type": "RC",
    "code|4": null,
    "code|4|type": null,
    "billing_class": "facility",
    "setting": "outpatient",
    "drug_unit_of_measurement": null,
    "drug_type_of_measurement": null,
    "standard_charge|gross": 69.0,
    "standard_charge|discounted_cash": 11.8,
    "payer_name": "Quiktrip Corporation",
    "plan_name": "Commercial"
```

3.4 Frontend Frameworks

3.4.1 Introduction

A crucial part of this project involves presenting the parsed hospital pricing data through a frontend that is not only efficient, but scalable. This section compares frontend frameworks and evaluates their suitability for this project based on the same set of criteria used in the scraping analysis: speed, reliability, scalability, licensing cost, security, ease of integration, and community support. The information in this section is drawn from official documentation, industry discussions, and early-stage prototypes created by our team.

3.4.2 Desired Characteristics

The ideal frontend framework should enable the creation of a responsive and maintainable user interface that is capable of handling large volumes of data. We will evaluate based off of the criteria from section 3.3, to stay consistent. We mainly want our framework to be hughly scalable, since tons of data will be displayed on our application. We want all of this data to remain uniform.

3.4.3 Alternatives

To identify the most effective frontend solution, we evaluated two frameworks: React.js and Next.js. React.js is a JavaScript library that is designed for building dynamic user interfaces through client-side rendering. Next.js extends React with additional features, such as server-side rendering (SSR) and static-site generation (SSG). These enhance performance and SEO optimization.

3.4.4 Analysis

Both React and Next.js are powerful, modern frameworks capable of supporting scalable and interactive user interfaces. Our team is more familiar with React.js because of its simplicity, flexibility, and our team's prior experience with it. React's component-based design enables modular development which makes it easier to build and maintain complex interfaces. Next.js provides additional capabilities through server-side rendering. These features are not essential for the current system's architecture, which prioritizes client-side responsiveness over SEO optimization.

3.4.5 Chosen Approach

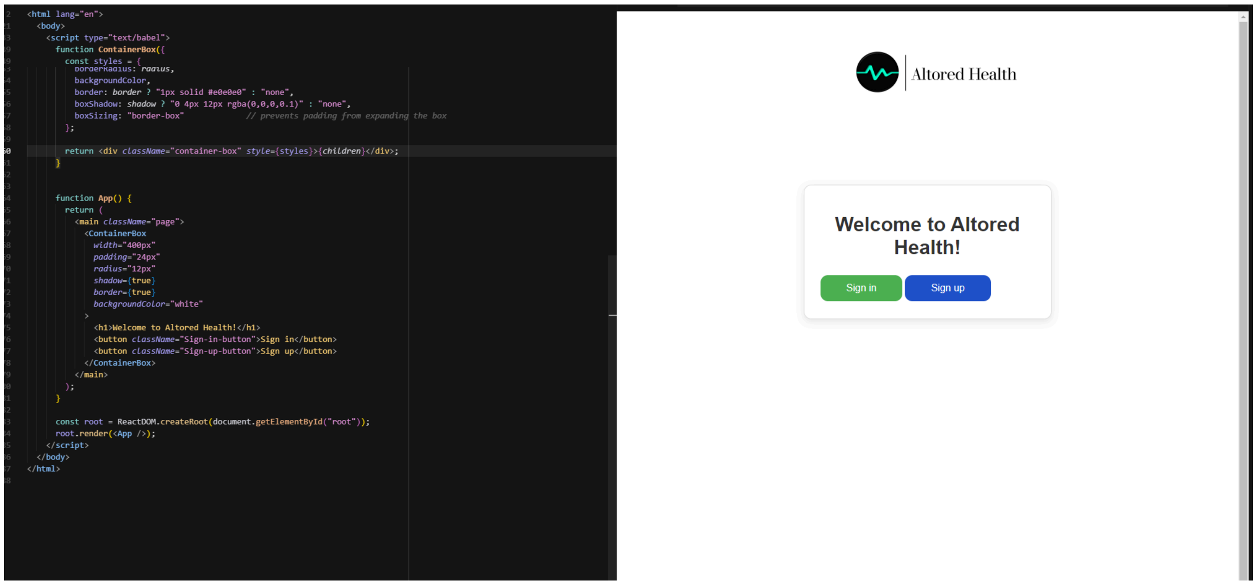
The team will move forward with React.js as the core frontend framework. React allows for the creation of a responsive, data-driven interface that efficiently updates in real time as backend data changes. Its robust ecosystem, strong documentation, and proven scalability in enterprise applications make it a dependable choice for the user-facing layer of this system. The use of React will also facilitate modular design and simplified maintenance as new features and data visualizations are introduced.

Frontend Frameworks

CRITERIA	React	Next.js
Speed	5/5 Good, but carries VDOM overhead	4/5 Strong with SSR
Reliability	5/5 Mature and Proven	5/5 Production-grade SSR
Scalability	5/5 Enterprise-scale	5/5 Adds built-in SSR/SSG
Licensing Cost	Free	Free
Security	5/5 Secure if coded properly	4/5 Good SSR practices
Ease of Integration	5/5 Works with everything	5/5 All-in-one React meta-framework
Community Support	5/5 Largest Ecosystem	5/5 Backed by Vercel, widely used
Total	**30/30**	28/30

3.4.6 Proof of Feasibility

The development team has begun implementing the frontend using React, starting with the creation of a basic user interface. We started with a layout featuring the project logo and two primary components: a Sign In button and a Sign Up button. This will be our first step in constructing the authentication flow and overall user experience.



```
<html lang="en">
  <body>
    <script type="text/babel">
      function ContainerBox({
        const styles = {
          borderRadius: radius,
          backgroundColor,
          border: border ? "1px solid #e0e0e0" : "none",
          boxShadow: shadow ? "0 4px 12px rgba(0,0,0,0.1)" : "none",
          boxSizing: "border-box"            // prevents padding from expanding the box
        };

        return <div className="container-box" style={styles}>{children}</div>;
      }


      function App() {
        return (
          <main className="page">
            <ContainerBox
              width="400px"
              padding="24px"
              radius="12px"
              shadow={true}
              border={true}
              backgroundColor="white"
            >
              <h1>Welcome to Altored Health!</h1>
              <button className="Sign-in-button">Sign in</button>
              <button className="Sign-up-button">Sign up</button>
            </ContainerBox>
          </main>
        );
      }

      const root = ReactDOM.createRoot(document.getElementById("root"));
      root.render(<App />);
    </script>
  </body>
</html>
```

3.5 Collecting data in a well-organized format

3.5.1 Introduction

In order to create a well-organized outline to build our product, our data must be stored in a way that is quick, easy to access, and easy to understand for the development team. Using a database is an ideal way to create such data structures. A database is built to be quick and efficient with large amounts of data, such as ours. Lastly, many of these databases have sets of features that lower the amount of work the backend team would have to do otherwise.

3.5.1 Desired Characteristics

With data collection, we need to take our parsed data and store it within a database that allows us to both access the vast amount of data quickly and efficiently. Additionally, the database we use needs to be versatile, as the needs of our database may change as we parse different sets of data.

3.5.2 Alternatives

3.5.2.1 PostgreSql

The open source PostgreSQL is a powerful tool that keeps data very consistent and organized within well-organized data groups. It can also support NoSQL systems for changing systems, but is much slower than native NoSQL systems.

3.5.2.1 CosmosDB NoSQL

Azure's Cosmos DB NoSQL is an interesting technology that groups JSON documents into localized region servers, allowing mass usage of the datasets, ideal for large web apps. The JSON file's syntax is designed by the developer, allowing data models to evolve quickly.

3.5.3 Analysis

Although using an open source software such as PostgreSQL to manage data is always a good idea, with our use case, Azure's Cosmos DB NoSQL is basically made for it. We will be managing vast libraries of data that can often change unpredictably across many regions with many users. PostgreSQL struggles with large data sets throughout many regions, and struggles with changing data sets. Additionally, added features such as backups can be useful to save the team time.

3.5.4 Chosen approach

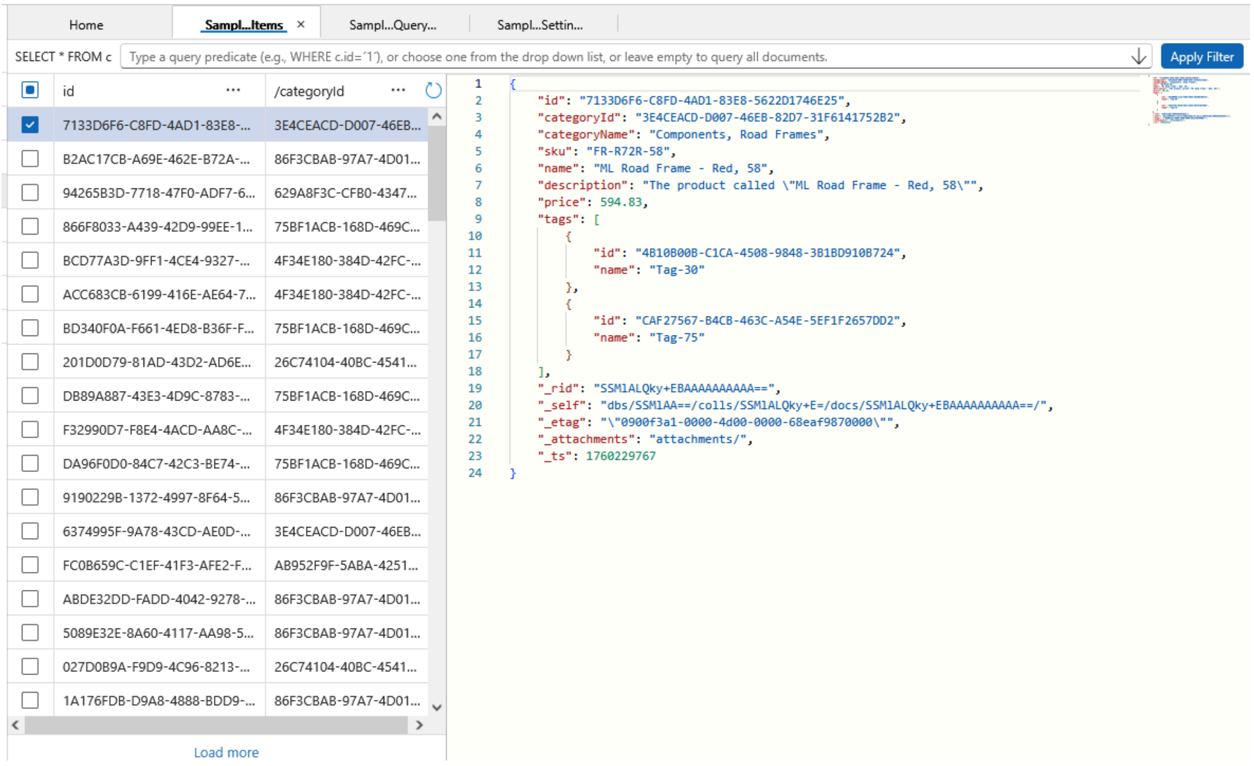
As shown in the table below, we are looking for a couple things: a versatile, organized database that scales well globally. Additionall,y any extra features we don't have to program at this stage of production, such as backups is an added bonus. Mainly due to CosmosDB's more effective use of NoSQL for versatile database control and more powerful region scaling, we will be going with CosmosDB's NoSQL.

DataBase

CRITERIA	Azure CosmosDB NoSQL	PostgreSQL
Versatility	5/5 Developer Defined Structures	2/5 Strictly Organized Structures
Reliability	5/5 Automatically scales globally by using region-based containers	3/5 Needs extra work to perform well globally
Extra Important Features	5/5 Automatic Backups	3/5 Developer needs to create backups manually
Total	**15/15**	8/15

3.5.5 Proof of Feasibility

We have already created a database and container for Azure's CosmosDB NoSQL. We have filled it with some test data, and early testing seems to be going well. As seen in the image below, the JSON files are quite versatil,e and our data can easily be placed in similarly outlined JSON files. Soon, we will be trying to output our data directly to the database to confirm functionality.



SELECT * FROM c

id	/categoryId
7133D6F6-C8FD-4AD1-83E8-...	3E4CEACD-D007-46EB...
B2AC17CB-A69E-462E-B72A-...	86F3CBAB-97A7-4D01...
94265B3D-7718-47F0-ADF7-6...	629A8F3C-CFB0-4347...
866F8033-A439-42D9-99EE-1...	75BF1ACB-168D-469C...
BCD77A3D-9FF1-4CE4-9327-...	4F34E180-384D-42FC-...
ACC683CB-6199-416E-AE64-7...	4F34E180-384D-42FC-...
BD340F0A-F661-4ED8-B36F-F...	75BF1ACB-168D-469C...
201D0D79-81AD-43D2-AD6E...	26C74104-40BC-4541...
DB89A887-43E3-4D9C-8783-...	75BF1ACB-168D-469C...
F32990D7-F8E4-4ACD-AA8C-...	4F34E180-384D-42FC-...
DA96F0D0-84C7-42C3-BE74-...	75BF1ACB-168D-469C...
9190229B-1372-4997-8F64-5...	86F3CBAB-97A7-4D01...
6374995F-9A78-43CD-AE0D-...	3E4CEACD-D007-46EB...
FC0B659C-C1EF-41F3-AFE2-F...	AB952F9F-5ABA-4251...
ABDE32DD-FADD-4042-9278-...	86F3CBAB-97A7-4D01...
5089E32E-8A60-4117-AA98-5...	86F3CBAB-97A7-4D01...
027D0B9A-F9D9-4C96-8213-...	26C74104-40BC-4541...
1A176FDB-D9A8-4888-BDD9-...	86F3CBAB-97A7-4D01...

```
{
    "id": "7133D6F6-C8FD-4AD1-83E8-5622D1746E25",
    "categoryId": "3E4CEACD-D007-46EB-82D7-31F6141752B2",
    "categoryName": "Components, Road Frames",
    "sku": "FR-R72R-58",
    "name": "ML Road Frame - Red, 58",
    "description": "The product called \"ML Road Frame - Red, 58\"",
    "price": 594.83,
    "tags": [
        {
            "id": "4B10B00B-C1CA-4508-9848-3B1BD910B724",
            "name": "Tag-30"
        },
        {
            "id": "CAF27567-B4CB-463C-A54E-5EF1F2657DD2",
            "name": "Tag-75"
        }
    ],
    "_rid": "SSMlALQky+EBAAAAAAAAAA==",
    "_self": "dbs/SSMlAA==/colls/SSMlALQky+E=/docs/SSMlALQky+EBAAAAAAAAAA==/",
    "_etag": "\"0900f3a1-0000-4d00-0000-68eaf9870000\"",
    "_attachments": "attachments/",
    "_ts": 1760229767
}
```

4. Technology Integration

4.1 Brief Statement

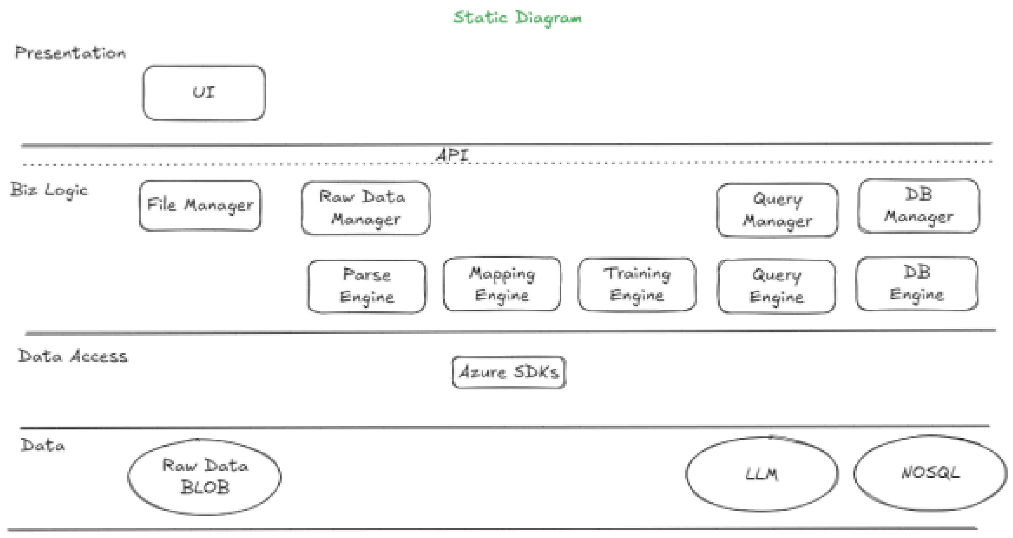
When looking at this project, focusing on the technologies is going to be crucial to ensure success. Therefore, we are going to create a cohesive system to document what it could look like.

4.2 Narrative Walkthrough

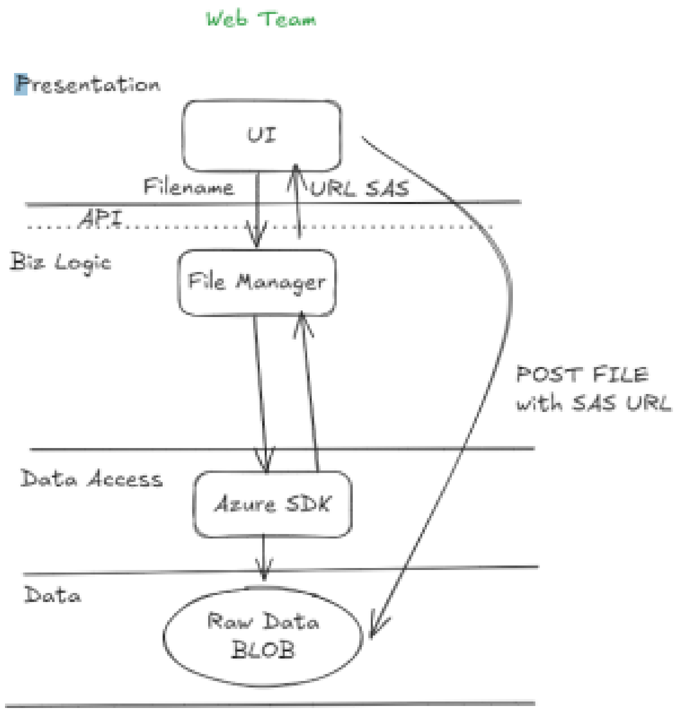
These diagrams show a cloud-based, event-oriented system architecture for the healthcare pricing application. System flow begins when a user action at the Presentation Layer causes a manager at the Business Logic layer. Important events, such as the upload of a file, are handled by calling automatically specialized processing engines—such as the Parse Engine for bills or the DB Engine for queries—interfacing with Azure data services. This architecture constructs a single and self-contained pipeline where data moves effortlessly from user input through processing and storage, with high-end features like machine learning model training and dynamic price lookups.

4.3 System Architecture Diagram

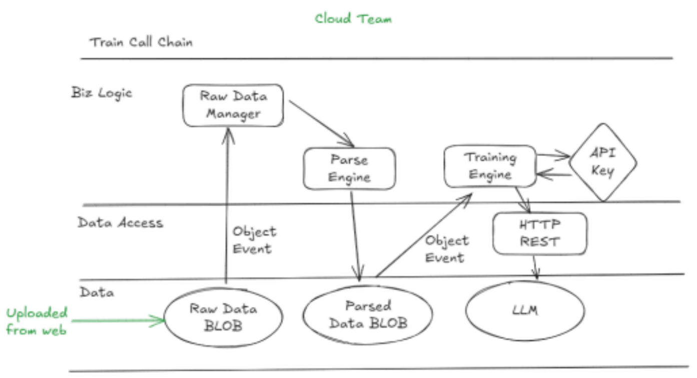
Step 1: This shows the high-level, static architecture of the entire application, organized into layers. Data and control flow from the user-facing Presentation layer down through the Biz Logic and Data Access layers to the final storage in Data repositories like BLOB and NoSQL.

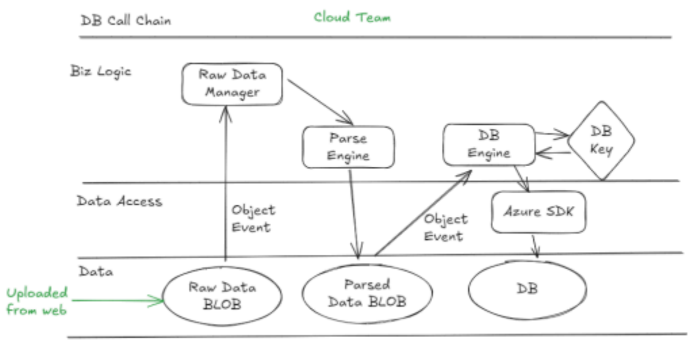


Step 2: This diagram details the specific data flow for a user uploading a file. The File Manager receives the file, uses the Azure SDK to get a secure SAS URL, and then transfers the file directly to Azure BLOB storage for safekeeping.

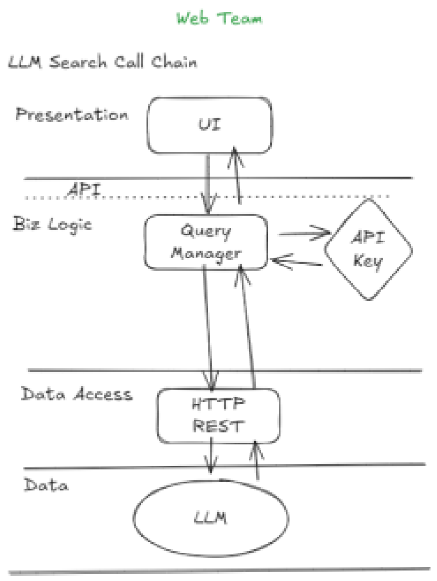


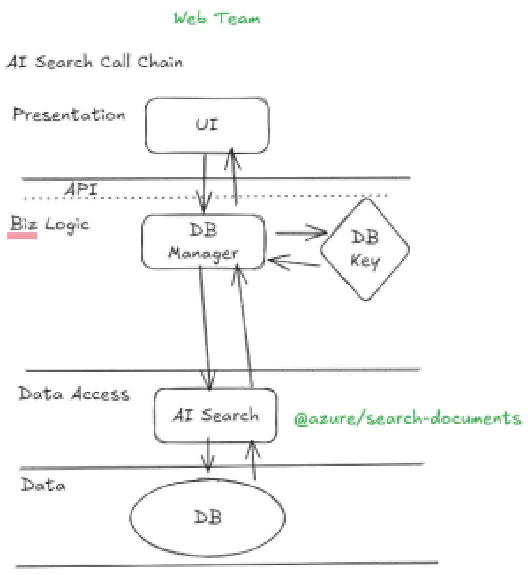
Step 3 and 4: These illustrate two automated processes. When a new file arrives in storage, an event triggers the Parse Engine to process it; the extracted data is then either sent to a Large Language Model (LLM) for training or stored in a database by the DB Engine.


Cloud Team
Train Call Chain
Biz Logic
Raw Data Manager
Parse Engine
Training Engine
API Key
Data Access
Object Event
Object Event
HTTP REST
Data
Uploaded from web
Raw Data BLOB
Parsed Data BLOB
LLM


DB Call Chain
Cloud Team
Biz Logic
Raw Data Manager
Parse Engine
DB Engine
DB Key
Data Access
Object Event
Object Event
Azure SDK
Data
Uploaded from web
Raw Data BLOB
Parsed Data BLOB
DB

Step 4:


Web Team
LLM Search Call Chain
Presentation
UI
API
Biz Logic
Query Manager
API Key
Data Access
HTTP REST
Data
LLM


Web Team
AI Search Call Chain
Presentation
UI
API
Biz Logic
DB Manager
DB Key
Data Access
AI Search
@azure/search-documents
Data
DB

5. Conclusion

5.1 Reiterate the Problem

The U.S. healthcare system places an immense financial burden on patients through its complex and opaque pricing. Individuals are consistently faced with unpredictable, confusing, and often excessive medical bills, leaving them vulnerable to financial hardship and unable to make informed decisions about their own care. This lack of transparency is not merely an inconvenience; it is a fundamental flaw that undermines patient empowerment and financial security. Our project is built on the conviction that by demystifying healthcare costs, we can return a sense of control and clarity to patients and their families.

5.2 Summarize the Document

Recap the main technological challenges analyzed and the key choices made.

5.3 Path Forward

The technical assessment confirms that while the challenges are significant, they are surmountable with a deliberate and stepwise strategy. We have a clear, concrete plan to build a sound, scalable, and stable platform.

5.3.1 Our immediate next steps are:

Detailed System Design

We will create comprehensive technical requirements for the database schema, application architecture, and API integrations, with the problems we've identified as our roadmap.

Phased Rollout (Flagstaff, Arizona, first)

We will begin developing with a minimum viable product (MVP) that focuses exclusively on Flagstaff, Arizona. This will allow us to pilot-test our approach for data aggregation, core functionality (procedure & copay pricing), and user interface within an isolated setting. After we obtain success in Flagstaff, we will scale to larger cities and states.

Prototype and Data Sourcing

Concurrently, we will create a functional prototype of the user interface and apply web scraping pipelines for our core datasets.

By fixing the underlying issues of data capture, integrity, and user attention at the front end, we are creating a sound technical foundation. Disciplined design that leads to a stable, valuable service at our initial Arizona launch will set us up for controlled growth and later deployment of high-end capabilities like bill analysis and community forums. We are confident that this project not only is needed but also is technically feasible, and we are prepared to begin implementing this solution into the real world.