# Task 1 (Part 1): Design Document

## Task 1 Relational Database Design and Implementation

**By David J Kelley**

(David@AGILaboratory.com)(dkelle24@wgu.edu)

### Introduction

This assignment is essentially walking through the database design process for a given business based on one of two scenarios and in this case, I have selected the first scenario around a HealthFit company focused on healthcare technology. Given that there is not an example of the assignment or design requirements (i.e. reference examples of completed versions of the assignment etc.) I will do the assignment to the level I might professionally, which may or may not be the level required but I suspect it will be. If this was an actual engineering document there would be additional sections that I would normally add including things like executive summaries, target audience and the like but that seems like it would just make it difficult to grade so I’ve chosen to following the assignment outline as close as possible.

### Part 1.Task A - Selecting Scenario 1

To summarize scenario 1: HealthFit Innovations, a growing healthcare technology company, is developing a platform called "HealthTrack" to revolutionize healthcare data collection, analysis, and personalized health recommendations. The platform integrates data from various sources, such as wearable devices, electronic health records (EHRs), medical imaging systems, and patient-reported outcomes, to provide real-time health insights and predictive analytics. As the platform’s user base grows, the increasing volume and diversity of health data are overwhelming the existing database infrastructure, leading to performance bottlenecks and challenges in data integration. To resolve these issues, HealthFit is seeking a database consultant to design a scalable, secure, and efficient relational database that can manage this dynamic data. The consultant will focus on creating a logical database design that includes a data model, entities, attributes, data organization, scalability, security, and integration capabilities to ensure smooth platform operation and enhanced data management.

### Part 1.A.1 - Business Problems

The Scenario 1 problem space can have any number of business cases but at a high level I count 7 cases.

**Case 1 – Real-time Data Integration and Processing**

HealthTrack needs a scalable database that can integrate and process real-time data from various sources, such as wearable devices (e.g., fitness trackers, smartwatches), EHRs, medical imaging systems, and patient-reported outcomes.

**Problem Solved:** The database will facilitate seamless integration and real-time synchronization of data from different sources, enabling HealthTrack to provide accurate, up-to-date health insights.

Those additional cases include “Data Scalability and Performance Optimizations,” “Elimination of Data Silos and Enhanced Data Accessibility,” “Predictive Analytics and Personalized health Recommendations,” “Data Security and Compliance with Healthcare Regulations,” “Improved user Experience with Real-Time Monitoring,” and “Efficient Data Backup and Disaster Recovery.”

### Part 1.A.2 Proposed Data Structure

I laid out a basic UML Object model that includes all of the core objects that are needed to support all of the use cases above, a couple of items to note is that I approached it following these guidelines:

**Rules or Normalization Third Normal form** – this means that for the most part tables are not repetitive we may back away from this more rigid level of normalization because or performance at the production level through views or edge caches or no sql databases and such but at the core of doing complex analysis is information that is standardized structurally and has ‘integrity’ meaning w/o corruption that you might see if you kept things raw. For example if you just have a ‘name’ field then what does that mean? First name last name or last name first name or first initial last name etc. Humans generally are not very consistent and being precise and implementing a reasonable amount of normalization helps with this process.   
**Federated Data Model** – To really get performance out of a structured data model the ‘Federated model’ engineering pattern is probably the plan and I don’t want to do anything that breaks that. This allows the system to potentially be scaled both up and out. However, it is not reflected well at this early stage but will be in a ERD (Entity Relationships Diagram). When building the object model that fact that I want to implement such a large scale database pattern means that I am thinking about ensuring that the data is structured in such a way that it can be easily ‘silo-ed’ on its own. In production this might be multiple clusters holding different parts of the database or that different RDS instances of SQL Server or Postgres will hold different segments.

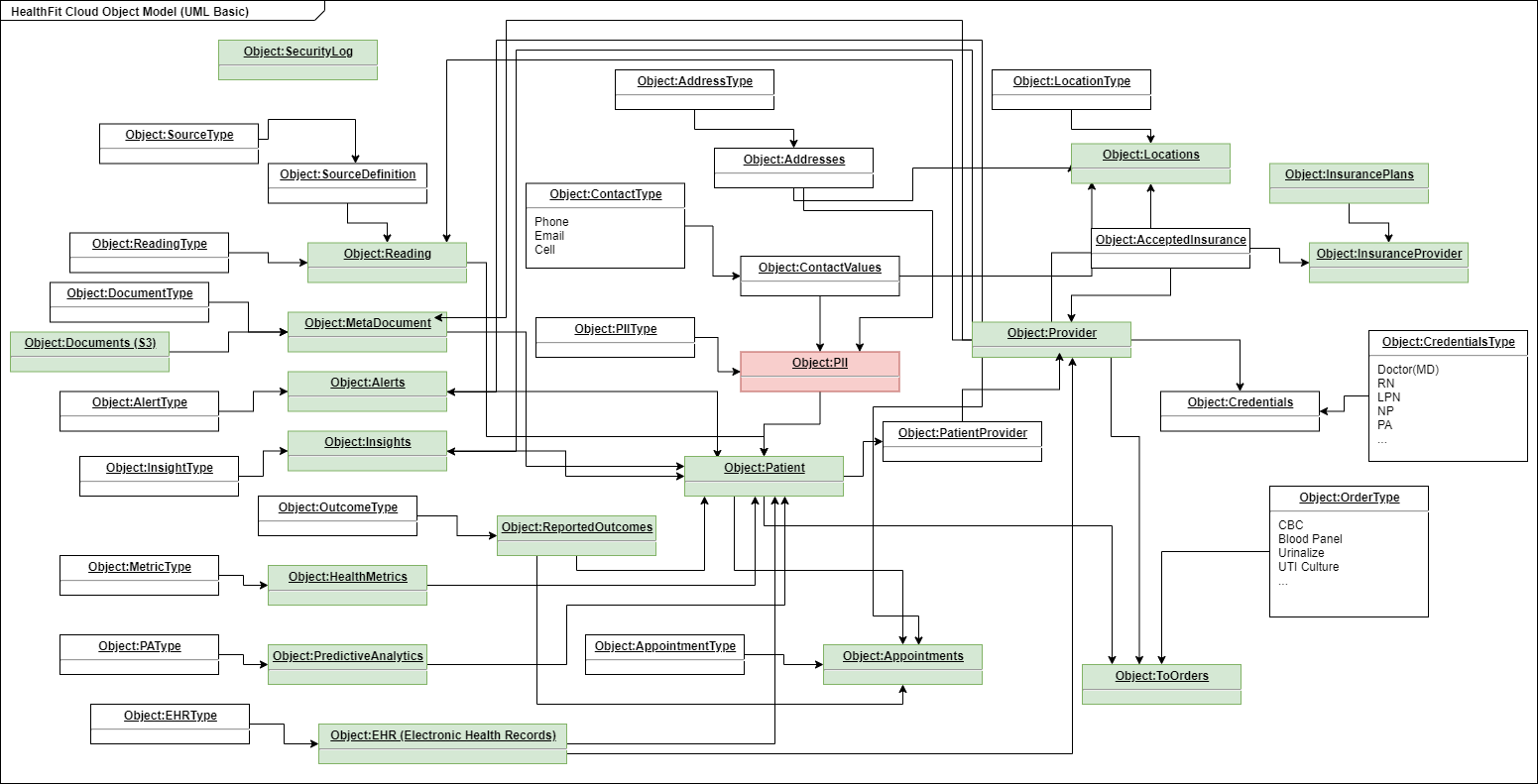
**Missing Relationship tables** – in some cases there are many to many relationships that might be needed between multiple tables, mostly this is not shown only the keyed relationships with critical foreign keys that really define the object for example that ‘Patient’ object is related to the ‘Reading’ table. But there are some important many to many relational ‘tables’ represented as objects in this simple UML diagram specifically the ‘*AcceptedInsurance’* object. In this example the ‘*AcceptedInsurance’* is better thought of as an object on its own but underneath the covers its representing and many to many relationships. Another related point is this is NOT an ERD but and object model and therefore would not normally shows all of those support relationship tables. You could make an argument for not putting type tables as objects, but I thought it was important to visualize a few of the changes to what would be my normal do.

**HIPAA Compliance:** the approach like using a PII table and setting up the data layout the way it is structured so that access to the data can be better controlled and follow regulations like HIPAA with being secure. You would still need to control what admins or data scientists have access to but separating PII from the patient table makes that easier. There maybe other laws one might need to consider depending on numerous issues but HIPAA is one that I know for sure applies to the kinds of data this database system will need to support.

**Object Properties:** Typically, in an object model I will only put property when they help define what something is where I put examples. In UML you might define them more like classes typically but at that point I feel you might as well just do the ERD. This approach helps visualize the structure of the objects and their relationship to the business cases before doing the detail work required for an ERD. One example in this diagram is the ‘ContactType’ object which shows Phone, Email and Cell. Instead of having a phone object or email object, there is a ‘ContactType’ object that can be all of that and the ‘ContactType’ object allows us to differentiate.

Views, Caching: These will be necessary but not at the UML object model level.

Here is the basic UML model:



**Figure A – High Level Object Model** <refer to “*D597 Task 1 Object Model.drawio*”, and “*D597 Task 1 Object Model.png*”>

### Part 1.A.3 Why this structure

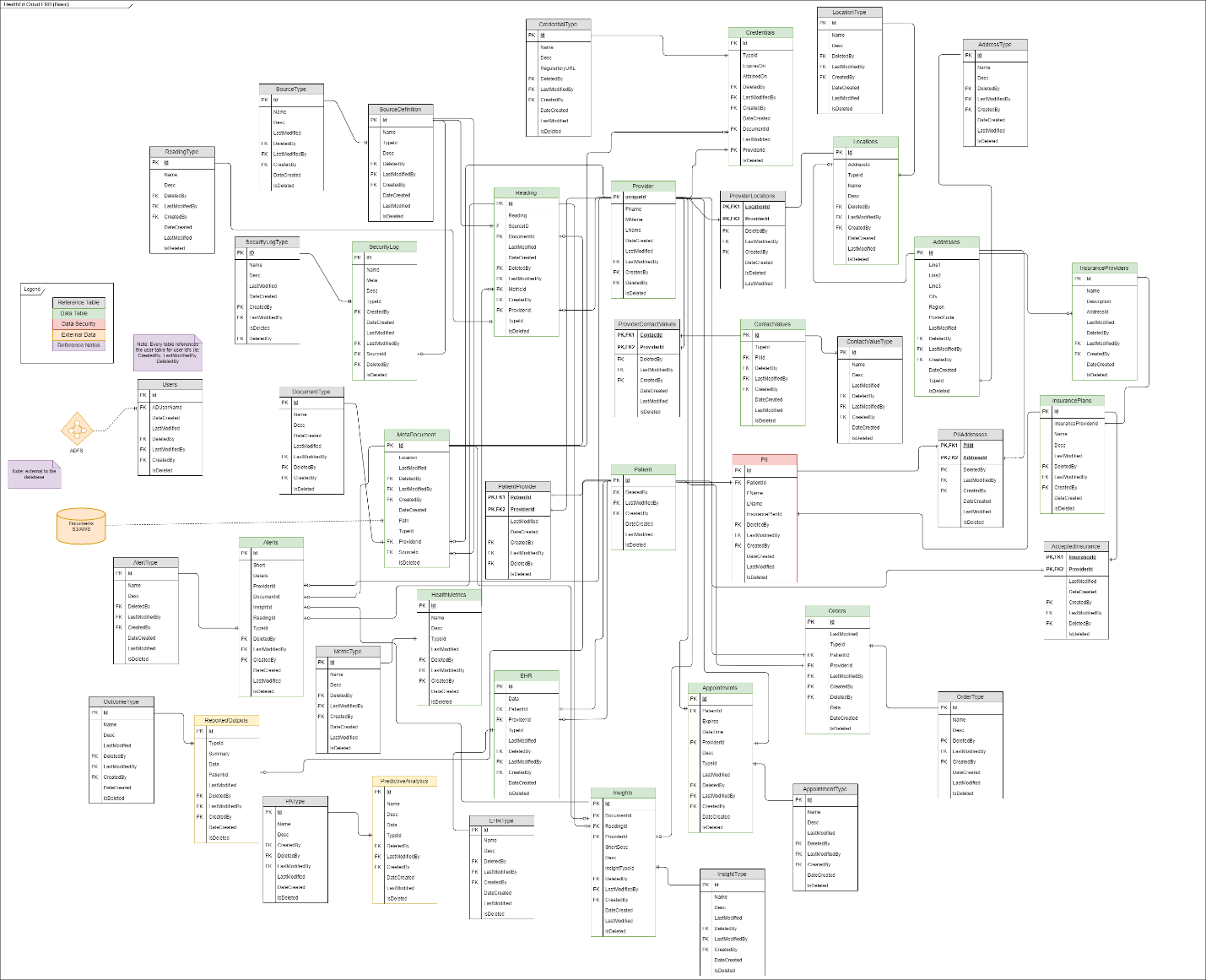
This structure as seen in Figure 1 matches all the proposed cases I mentioned earlier and is as normalized as might be needed for scale and processing. This structured third normal form allows us to better insure data integrity. If you go back to the scenario to really record what you need, there needs to be Patient and Patient information, medical records, PII data, lab results, references to documents like x-rays and all the provider information. But the scenario implies a much richer set of functionality might be needed and is noted above and is addressed with a possible solution in the data architecture here. Once you figure out all the related data to all of those cases you end up with an object model similar to what I have done in figure 1. For someone not familiar with databases in real life this might look like too many but if you have done actual enterprise databases in my experience it is more likely we are missing tables/objects. Additionally, most of the business case(s) need the ability to store and structure readings and sensor data which the above object model accommodates so that these kinds of cases can be done. In some cases, we are using raw data fields which could be schemed XML or JSON in this rev until we understand more details about the data. That means that elements of the database could change in real world scenarios.

### Part 1.A.4 How will the data be used

This system is structured where you can do analytics while keeping PII data separate. This better supports Data Analytics vs automated systems accessing just what they need. This limits exposure of the business while also limiting risk patients. Typically, the data will not be accessed directly but will likely be put into views that are accessible via API’s that are consumed by various apps. There may be ETL processes used to scrub and pull data into this model and there maybe edge caching of different kinds needed at a certain scale where blocks of data is essentially stored in the database and stored in a secondary database on a particular server like an API server. The most important function of how this data will be used in this business is for providers (specifically doctors) to use the data through APIs and visualization tools and other tooling to help patients become healthier or better from injury or illness.

### Part 1.B Create a logical data model

In this section I build not just an entity relationship diagram (ERD) but built out to show a full logical model. In a real implementation at scale this would need to be silo’s and some relationships would need to be enforced programmatically so implementation will take that into account as its being built. This allows the potential to scale out and up using a federated data model (engineering design patter). Additionally, a real implementation at scale would likely need caching of at least reference data in either Redis or in a non-SQL database (Mongo DB etc.) either on application servers or a dedicated caching server. There are cases at scale that may also need to break normalization rules which would need to be addressed in this design. Here is the basic implementation based on the UML Model. Also note two references that are shown as relationships to external data sources. These have to do with users and documents. Documents would not be stored directly in the database but instead meta data and locations to a file server or data lake of some kind with the raw files and then user id’s are used that reference some ADFS or other structure tied to the AD of the given company implementation. This allows for security being managed possibly in a multi-tenant environment where the same user accounts are used across the system and managed outside of the database and allow us to track AD user account activities in the database which has been structured to support that already.



**Figure B – Solution ERD** <refer to the original Draw.io file for further details “*D597 Task 2 ERD.drawio*”>

### Part 1.C Describe DB Objects and Storage

The assignment says, “Describe the database objects and storage, identifying the file attributes within the database solution.” To be clear to me this implies that files need to be stored in the database however I would argue this is not a good design and I would store only the meta data related to files and files would be stored in either an S3 Bucket, a Data Lake maybe Snowflake or something like that. The database above reflects that showing ‘meta’ data and processed data extracted from files but uses a Schema system to map meta data for processing either into EHR references, or readings or other parts of the over all data models. Here is a list and details about each object in this database.

**Note:** that the tables are divided into the following types: Data, Reference, MetaData, and Relational tables. Reference tables are for things used to define types mostly and cleaning up data tables that have categories. MetaData tables are tables that are primarily data about another object in a data table. Relational tables are used to create many to many relationships for example many providers will accept many of the same insurance plans.

Tables:

1. **Addresses** (type: Data): stores all addresses.
   1. **AddressType** (type: Reference): reference types of addresses i.e. home, work, mailing, billing, etc..
2. **Alerts** (type: Data): used to store alerts.
   1. **AlertType** (type: Reference): used as a reference type to indicate the time of alert.
3. **Appointments** (type: Data): used to store patient appointments allowing the system to be used to manage clinic appointments.
   1. **AppointmentType** (type: Reference): Reference data types for appointments.
4. **ContactValues** (type: Data): used to store contact values including phones, emails, urls or other possible contact values except addresses.
   1. **ContactValueType** (type: Reference): Reference data for the types of contact values i.e. work email, home email, cell phone etc.
5. **EHR** (type: Data): used to store electronic health record files using a schema based system.
   1. **EHRType** (type: Reference): Reference data for the types of EHR files stored.
6. **HealthMetrics**(type: Data): stores patient metric data.
   1. **MetricType** (type: Reference): Reference data for the types of metric values.
7. **Insights** (type: Data): used to store insights about patients by providers.
   1. **InsightType** (type: Reference): reference data for the types of insights.
8. **InsuranceProviders** (type: Data): all possible insurance providers.
   1. **InsurancePlans** (type: MetaData): all types types of plans each provider has. This could be extended to include a its own type/reference table.
9. **Locations** (type: Data): Provider locations.
   1. **LocationType** (type: Reference): reference data for the type of provider locations.
10. **MetaDocument** (type: Data): stores all the data about documents loaded into the system either via S3 or Datalake etc.
    1. **DocumentType** (type: Reference): Reference table for the types of documents.
11. **Orders** (type: Data): medical orders/requests from providers for patients.
    1. **OrderType** (type: Reference): Reference data for provider order types.
12. **Patient** (type: Data): Core patient definition.
13. **PatientProvider** (type: Relational): stores all the providers for a given patient.
14. **PII** (type: Relational): stores PII data id’s used to abstract PII data from the patient data helping with compliance and provider laws and policies such as Hipaa regulations.
15. **PIIAddresses** (type: Data): used to reference what address’s a given patient has.
16. **PredictiveAnalytics** (type: Data): used to store predictive analysis data using an XML Schema system.
    1. **PAType** (type: Reference): Predictive Analysis type data.
17. **Provider** (type: Data): defines what providers are in the network/
    1. **Credentials** (type: MetaData): shows what medical credentials each provider has and when they expire.
       1. **CredentialType** (type: Reference): defines the types used for each ‘type’ of credential.
    2. **AcceptedInsurance** (type: MetaData): this table shows what insurance each provider accepts.
    3. **ProviderContactValues** (type: Relational): shows what contact values each provider has.
    4. **ProviderLocations** (type: Relational): shows what locations each provider is associated with.
18. **Reading** (type: Data): stores patient device reading data.
    1. **ReadingType**(type: Reference): reference data for readings.
19. **ReportedOutcomes** (type: Data): Outcomes reported by providers for given patients.
    1. **OutcomeType**(type: Reference): reference type data for reported outcomes.
20. **SecurityLog** (type: Data): used to store security logging information via internal application instrumentation separate from external system logging like db logs or DataDog etc.
    1. **SecurityLogType** (type: Reference): Reference type data for the security log table.
21. **SourceDefinition** (type: Data): defines sources for inputs and other devices etc.
    1. **SourceType** (type: Reference): reference type data for sources.
22. **Users** (type: Data): maps users to a user id from AD usernames. This prevents AD usernames from being in tables and access to this type can be restricted.

You’ll note there are 43 tables but only 22 of them store actual data and the others are for many to many relationships or reference and meta data tables. <Refer to the file “*D597 Task 1 DDL.sql*” for more information>

## Part 1.D Describe DB Scalability concerns and mitigation strategies

The proposed database design addresses scalability concerns by incorporating several strategies aimed at optimizing performance as the volume and variety of data in the HealthTrack platform grows. One key approach is to organize the database with a focus on modularity and normalization, which allows different types of data, such as patient information, device readings, and medical records, to be managed separately. This separation ensures that as the platform's user base and data input increase, the database can scale both vertically (by increasing capacity of individual database servers) and horizontally (by distributing data across multiple servers or instances).   
  
This particular approach is related to the federated data model as an engineering design pattern for database architecture. This architecture minimizes performance bottlenecks and allows for efficient handling of real-time data inputs from wearables and other sources and is the most scalable while retaining highly normalized data.

The recommended database design also includes caching mechanisms to boost performance, particularly for frequently accessed data, such as health metrics and reference tables. By caching this data in faster, non-relational databases (e.g., Redis or MongoDB), the platform can reduces the load on the core relational database and significantly improves response times. This is particularly useful for real-time health insights and predictive analytics, where speed is crucial. Caching allows HealthTrack to efficiently handle large-scale queries without compromising on data integrity or slowing down the user experience.

Another important strategy in the design as articulated is selective denormalization of the database in certain areas to enhance performance especially at the level of local caching. While adhering to third-normal form ensures data integrity, de-normalizing specific tables allows for faster queries by reducing the need for complex joins. This approach is particularly useful for handling large volumes of real-time health data, where performance is a priority. The combination of caching, denormalization, and modular data organization ensures that the database can grow and evolve alongside HealthTrack's expanding user base, while maintaining efficient and secure data management.

Another important note ERD shown above is not spread across core RDMS servers. It is only if the volume is high enough the federated data model could be applied changing the ERD some what in terms of relationships which would have to be programmatically enforced but it would effectively allow the system to scale out by orders of magnitude by siloing across multiple servers or server clusters.

## Part1.E Output the privacy and security measures that should be implemented.

The proposed database design for HealthTrack must incorporate several privacy and security measures to comply with healthcare regulations, protect sensitive patient data, and secure the platform in a multi-tenant environment. The design must ensure compliance with key frameworks like HIPAA and SOC 2, as well as implement additional security protocols that are critical for handling sensitive healthcare data.

HIPAA compliance is a top priority. All electronic health information (ePHI) must be encrypted both at rest and in transit using industry-standard methods such as AES-256. This ensures that even in the event of unauthorized access, the data remains secure and unreadable. Access to ePHI should be governed by role-based access control (RBAC), allowing only authorized personnel—such as healthcare providers or system administrators with appropriate privileges—to access the sensitive information. Furthermore, maintaining detailed audit logs is essential to track all access, modifications, and actions performed on ePHI, ensuring accountability and enabling detection of any unauthorized activity, as required by HIPAA.

To meet SOC 2 compliance standards, the database design should incorporate strong security measures, including firewalls, intrusion detection systems (IDS), and regular vulnerability scanning to protect against unauthorized access. Ensuring the confidentiality of sensitive data is another key requirement, which can be achieved through data encryption and the use of encrypted communication channels, like Transport Layer Security (TLS), for secure data transmission. The availability of the platform must be ensured by implementing redundancy, data backups, and disaster recovery plans. High-availability and failover mechanisms should be put in place to minimize downtime and ensure continuous access to the platform in case of system failures.

In a multi-tenant environment, where multiple organizations or users share the same infrastructure, strict data segregation is necessary to prevent cross-tenant data leakage. This can be achieved through logical separation of data, such as tenant-specific encryption keys or database sharding, where data for each tenant is stored separately. Tenant-based access control should be enforced using attribute-based access control (ABAC), ensuring that users can only access data belonging to their organization. Additionally, regular audits should be performed to maintain compliance with regulations like HIPAA and SOC 2 in a shared environment, ensuring that data privacy and security are maintained for each tenant.

Data masking and de-identification techniques should be implemented to anonymize sensitive data when it is used for research, analytics, or other purposes that do not require personally identifiable information (PII). This reduces the risk of exposing sensitive data while maintaining HIPAA compliance. Storing PII in separate, highly secure tables or databases, with additional encryption and restricted access, allows for easier compliance with HIPAA’s minimum necessary rule, which requires limiting access to the minimum information necessary for a task.

Furthermore, the system should include regular backups and a robust disaster recovery plan. Automated, encrypted backups of the database must be regularly performed and securely stored. The disaster recovery plan should be rigorously tested to ensure that data can be quickly and securely restored in case of system failure or a breach. Finally, maintaining data integrity through database constraints and integrity checks is essential to ensure the accuracy and consistency of the data. Continuous monitoring for security threats and anomalous activity will provide an additional layer of protection, allowing real-time responses to potential issues and safeguarding the system’s overall security.

Take a look at the following diagram:

A diagram of a application

AI-generated content may be incorrect.

**Figure C – Screen Shot of Example Interactive Threat model**   
 <refer to the following documents “*HealthTrackExampleModel.tm7”*, “*HealTrackExampleModelAnalysis.htm”* and “*HealthTrackExampleModelScreenShot.png”*>

This is a screen shot of an interactive model of the basic structure of how the Database we are talking about would be used as I have articulated it earlier. Given API’s and load balancers and other components this could get progressively more complex but this is somewhat outside the assignment parameters as I understand them.

This model shows the basic structure of a single tenant implementation and shows how the main components work and who talks to the data and so forth. Even a simple model like this has a total of 61 potential attack surfaces. When looking at HIPPAA compliance or SOC 2 or other requirements, this kind of detailed threat model is important to help identify where things need to be addressed and mitigated. From the model we might consider DDOS attacks against the database, or spoofing authentication if a non-standard implementation is done. The kinds of mitigations at this level would be not giving access to the database from the internet. The server should be configured inside a trusted authenticated only boundary where only the app service, API service or cache etc. can even see the database and make calls. All of these kinds of things found in a model like this would need to be addressed. Security should be designed in from the start before implementation and tools like this help do that.

By implementing these privacy and security measures, the HealthTrack platform will not only meet regulatory and compliance standards but also provide robust protection for patient data in a secure, scalable, multi-tenant environment.

.