leapfrog

Technical Discovery

HANDBOOK

Guideline to effective technical discovery

Technical Discovery

HANDBOOK

Guideline to effective technical discovery

Credits

Authors

Amit Joshi

Bipen Chhetri

Chumlung Limbu

Drishya Bhattarai

Sagar Chalise

Sawan Vaidya

Swechhya Bista

Editor

Anish Krishna Manandhar

Designer

Bibek Shrestha

Special Thanks

Chris Bohnert

Suja Manandhar

De Architects Team (DART)

All Leapfroggers

Table of Content

About

Leapfrog's teams frequently undergo discovery at various points of software development. Leapfrog has adopted Design Thinking and Discovery as the primary approach for general discovery. However, such a framework has yet to be put forward for the technical aspect of discovery. This document attempts to conceptualize technical discovery and provides a general guideline that any software engineer can follow to conduct an effective Software technical discovery.

> While the rules in the handbook do not apply 100 percent of the time, they do characterize the best programming practices in the great majority of cases. You should not slavishly follow these rules, but voilate them only occasionally and with good reason. Learning the art of programming, like most other disciplines, consists of first learning the rules and then learning when to break them.
>
> **Josh Bloch**
>
> Author of Effective Java

Who is this book for?

This book is intended primarily for software architects and technical leads. It provides a general guideline for how a software engineer should make architectural decisions in different engineering areas during the discovery process.

The notion here is that we have the high-level business requirement document in place. This book does not illustrate the UI/UX design discovery; however, many decision-making and technical requirements are discovered as the design thinking process reveals what the product needs.

What is Technical Discovery?

Simply put, technical discovery is the process of determining the technical requirements of a project. It is a crucial part of the product discovery and design phase. It helps find the right balance of technology, resources, time, and effort for product development and delivery. It is a continuous process that helps shape the technical architecture in a particular direction. This process is always progressive and has to consider technical and human aspects, plus continuous trade-offs against each other.

With this in mind, we can define **software technical discovery as time-bound research and analysis of a product's technical aspect, with the goal of providing optimum engineering solutions to the product that aligns with product vision and business outcomes**.

Stages in Technical Discovery

Technical discovery is divided into four major stages. These stages can overlap with one another and can be carried out in iterations. However, we separate them in this playbook to represent the high-level activities taken during the technical discovery process.

> ! These stages can repeat during the process and do not have to be chronological. They represent generalized activities commonly done during technical discovery and are arranged in the best possible way. These steps can be skipped per the project needs and requirements or extended during the project development phase.

1. ANALYZE

- Requirement elicitations and specification
- Understand problem domain and functional requirements
- Non-functional requirement and quality attributes analysis
- Research architecture/ technology on similar products

2. ABSTRACT

- Convert business requirements to high level system design
- Create macro-level architectural blueprint
- High-level database design, brief description on systems, services, platforms and relationship among modules

3. ARCHITECT

- Convert high-level system to component level design
- Design detailed data models, API
- Select tech stacks/libraries/POCs on various layers
- Design infrastructural architecture
- Requirement breakdowns

4. ESTIMATE

- Requirement verification, validation, estimations
- Project timeline and milestones
- Team composition
- Cost analysis

Figure 1: Stages in technical discovery

To elaborate more, we will run a sample use case of the product **“Connected Care”** through different stages of technical discovery.

“Connected Care” provides Chronic Health Monitoring solutions to care coordinators. For this use case, the system will provide the following significant solutions:

1. Remotely monitor patients' vitals and symptoms
2. Triage information from different data sources like Smart Devices and Electronic Health Records (EHR)
3. Coordinate care collaboration among the care team
4. Concierge the communication between patients and physicians.

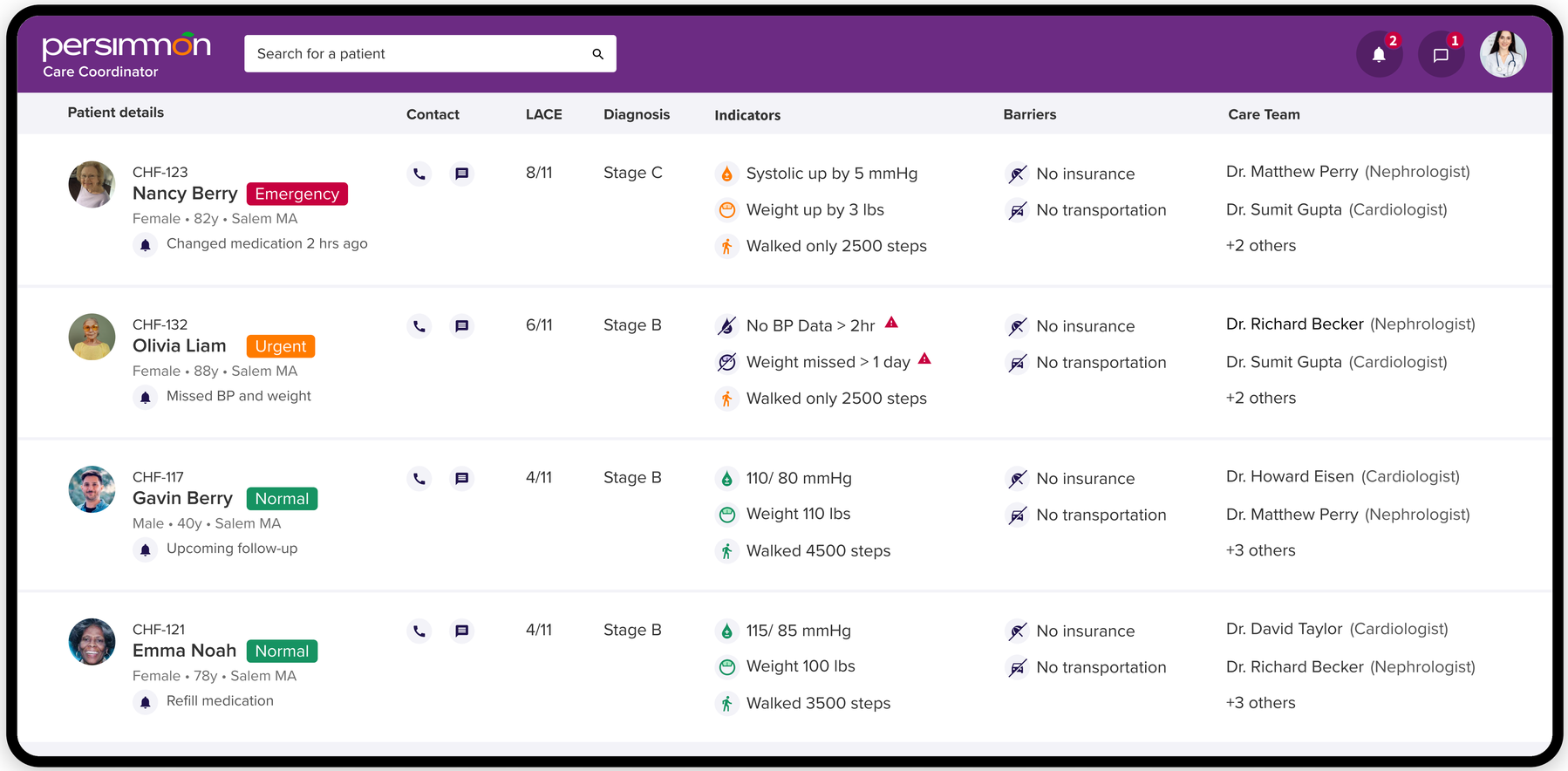


Figure 2: Care Coordinator view of the patient list

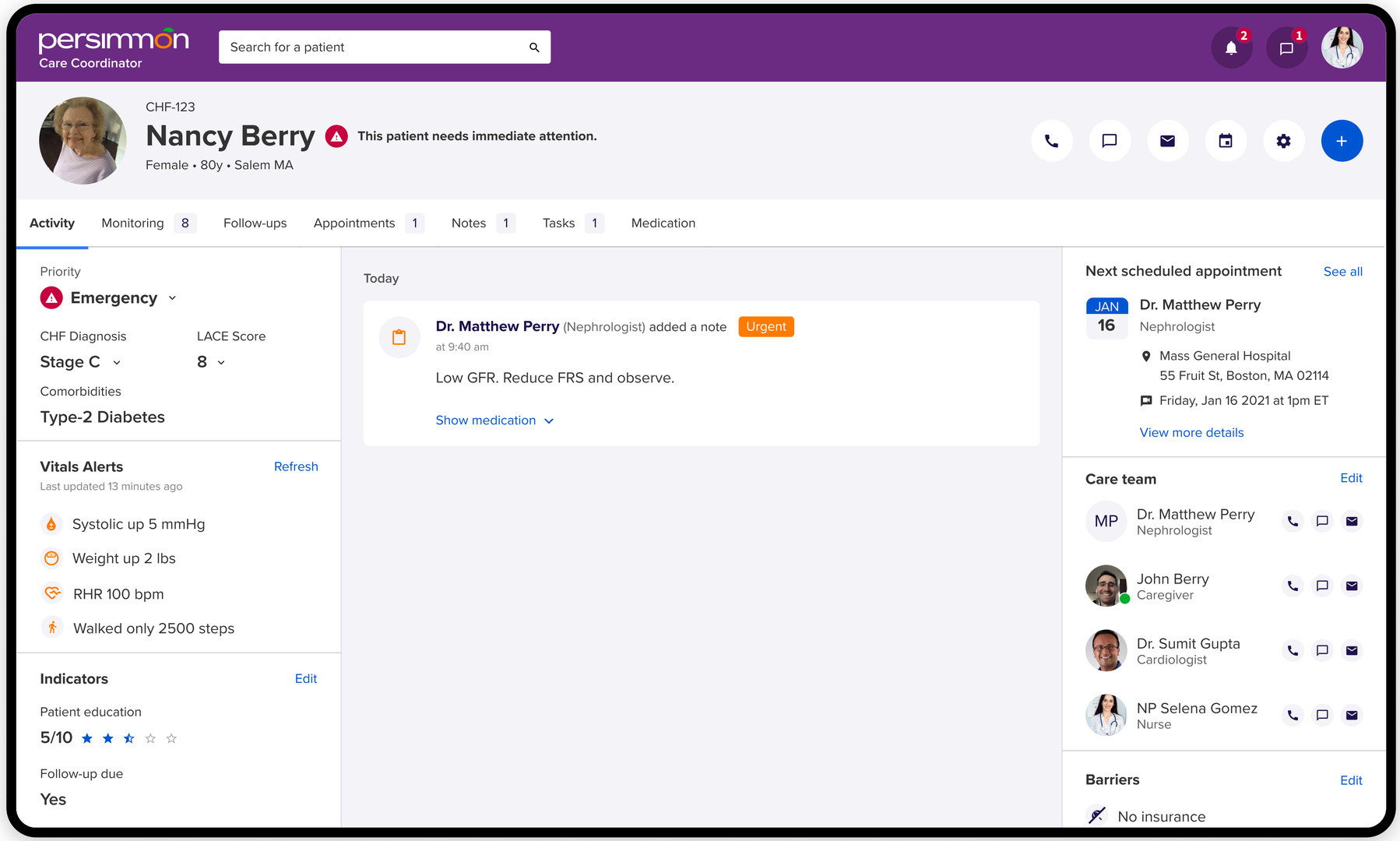


Figure 3: Care Co-Ordinator view of Patient Details

Analyze

“Analyze” is the first and preliminary stage of the technical discovery process. This stage is about analyzing and understanding the problem domain, the nature of the product, client needs, and the majority of the requirements. This stage is a requirement analysis phase where we learn about the functional requirements (product features, business use cases) and the **non-functional requirements** or **quality attributes** like security, scalability, reliability, maintainability, performance, etc.

Non-functional requirements describe how a system must behave and establish constraints on its functionality.

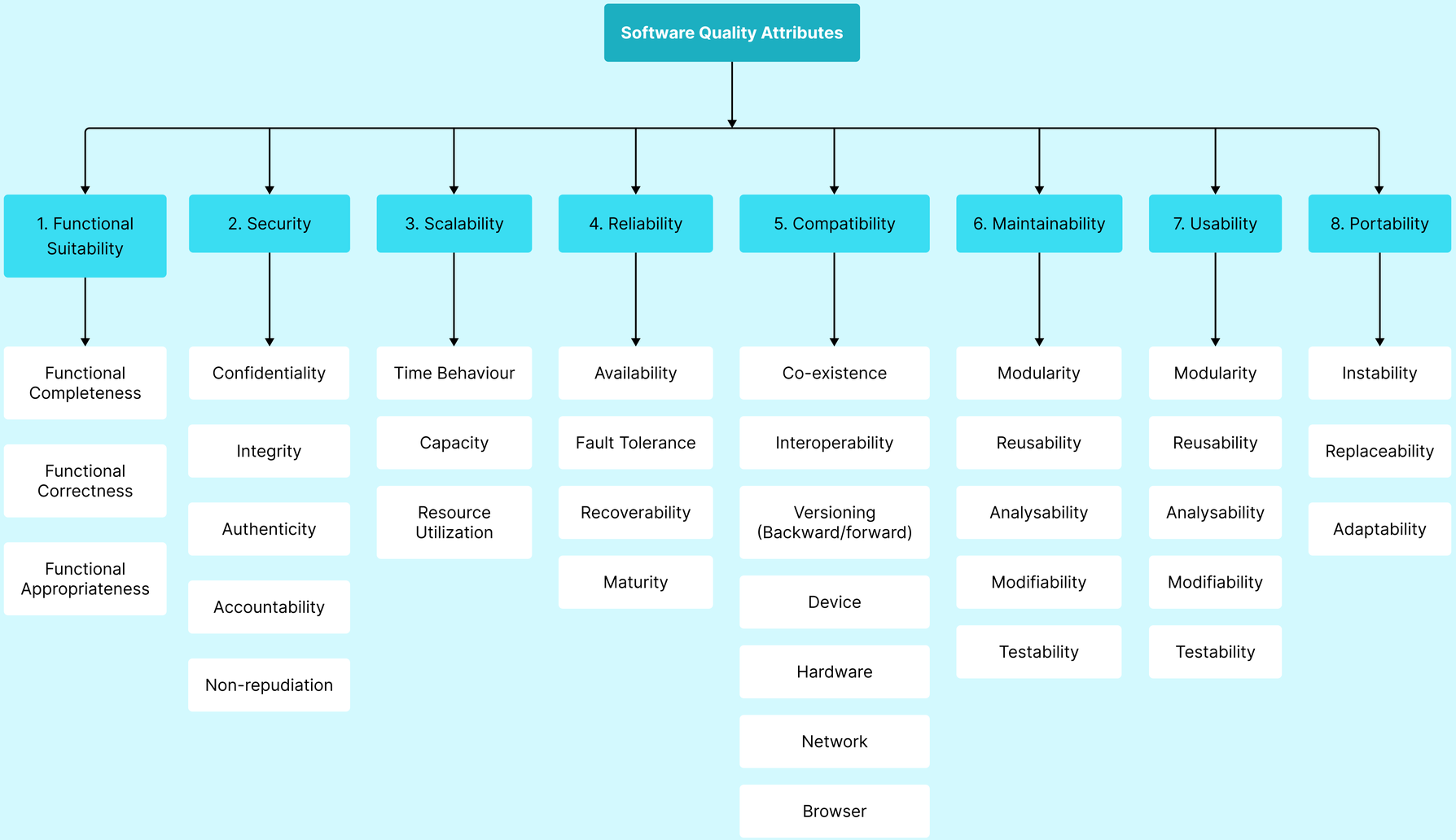


Figure 4: Software Quality Attributes (ISO/IEC FCD 25010)

Below are some of the questions that can be considered during analyze phase. Every question doesn't require to be answered and can be analyzed with cautious reasoning based on functional requirements.

Scalability

- What can be the total number of users?
- What is the acceptable response time to perform an action?
- How big can data volume grow in the following 5/10 years?
- How many users can access the system concurrently during peak hours?
- Does the system have to be available in multiple geographical regions? What is the user base like in each geographical region?
- Any feature that is likely to be primarily used?
- What is the acceptable page load time of a full screen?
- What is the impact of the data presented on screen being outdated or inaccurate?
- Does it have to be consistent in real-time or eventually?

Security

- Is the system tailored for a single organization?
- Does the system need to interact with external APIs?
- Does the system have external as well as internal users?
- Can authenticated users only do specified actions?
- Does any software compliance(HIPAA, FDA, PIA, SOC2, Etc.) need to be addressed during development and delivery?
- Does the system need to audit user actions at any required time?

Reliability & Recoverability

- What is the amount of Mean time between failures **(MTBF)**, Mean time to failure **(MTF)**, and Mean time to repair **(MTR)**?
- What if the system completely collapses under an availability zone? Do we need multiple availability zones?
- Under what extreme scenarios does the system guarantee uptime? What should be the guaranteed uptime?

Maintainability

- How convenient is it to test the software flow?
- How convenient is adding, modifying and maintaining new/existing features?
- What is the rate of bugs occurring after releases?

These are only the sample questions under a few quality attributes and can vary based on the product's compliance, scalability, performance, maintainability, security, and **other non-functional requirements**.

The questions mentioned above may seem only answered in yes/no format or numeric values, but these answers will serve as guidelines for better system design.

Let’s run these questions on our use case **“Connected Care”**.

The Problem

In care management, some patients require more supervision and care from their healthcare providers. In such cases, healthcare providers can benefit by using digital platforms that will allow them to monitor their patients and their healthcare status online as needed. It will enable them to view the patients' history, monitor their vitals, and communicate with them, eliminating the need for physical visits or appointments.

End Users

In the long run, Connected Care should be able to support various staff from the health care provider's organization. For example, Administrators, Doctors, Managers, Nurses, Etc. Each of these could have different roles and permissions in the Connected Care application. Grouping staff into different units, such as departments and teams, could also be helpful.

In Phase 1, however, we will begin with **one user- a healthcare provider** who will be onboarded to the platform by the developer.

SaaS-based

In Phase 1, the product is not a Software as a Service (SaaS), but as we learn more about the domain and improve the stability of the tech stack, we can market the product in a SaaS manner in the future.

Multitenancy

The product will be a single tenant in Phase 1. But, this product will definitely need to support multiple clients in the long run.

Nature of Data

There are two types of data sources:

1. Structured data from EHR (Electronic Health Record) systems. These data points are entered into an EHR system by various staff. They are most likely organized relationally and can be accessed via a service called Redox. We can assume that each EHR will contain patient data count in the range of 1000-2000. The data per EHR system will be manageable.

2. The second type of data will be sourced from various healthcare devices' data collection platforms, for example, Google Heath, Apple Health, Fitbit, Etc. These data will reside on servers of specific vendors. Connected Care will likely have to acquire permission from each patient and these data sources to begin ETL. The size of these data will likely be significant as there could be thousands of collected data points per patient per day.

Tentative User Base Request over a period of time

Number of people using smart wearables is ever-growing.

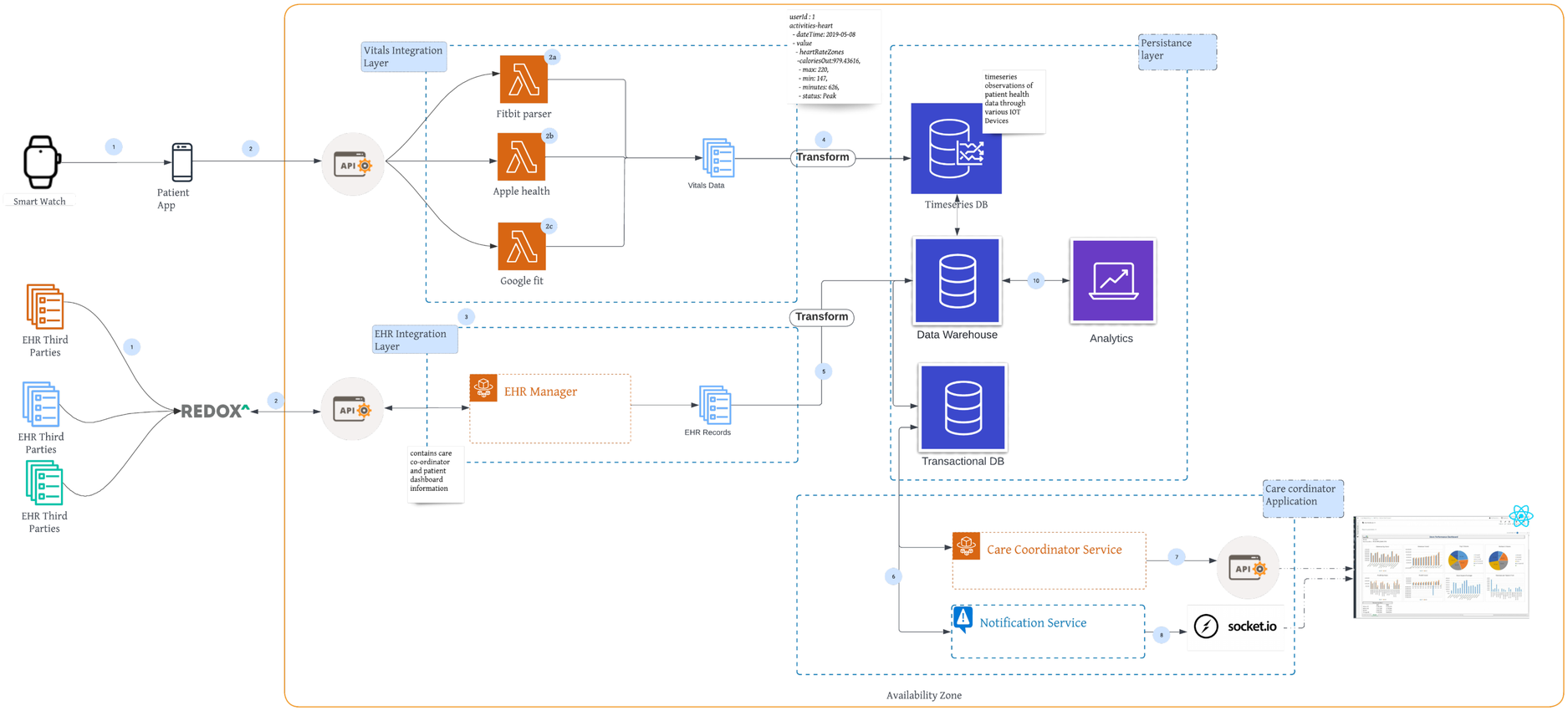
Compliance

- The team working on this project needs to undergo HIPAA compliance training.
- Following HIPAA standards, the production servers must be encrypted at rest and in transit. PHI data protection must be considered a high priority.
- In order not to constrain the development team by HIPAA needs, it is recommended to create test data and mock APIs for development purposes.

Abstract

During this stage, an abstract view of the product's overall architecture and main features is generated. This stage is the system design phase, where we try to design different components, elements, and interfaces of the system that satisfy the specified requirements irrespective of any technologies and stacks.

The following diagram represents a sample system design of the **Connected Care App.**



1 **EHR:** Third party system **Redox** provide easy connection to different EHRs. The implementation becomes independent of the EHR being used.

REDOX

SMART WATCH: Apple HealthKit collects data from iPhone, the built-in sensors on Apple Watch, compatible medical devices, and apps that use HealthKit. Similarly, Fitbit provides a set of public Web APIs that developers may use to retrieve Fitbit user data collected by the Fitbit trackers & smartwatches.

2 API call to gets vital and Patient data from respective dataset.

3 Data jobs to consume and extract Vital and EHR records from the APIs and push it to database layer

4 Transfrom data and records to map with database field

6 Get all records for care coordinator records and notification from the transactional DB to be displayed in the frontend

7 API to get required data from services to frontend

8 Web socket to pass notification to frontend

9 Dashboard to view the patient info and analytics. Reactjs will be used to build the dashboard frontend part.

10 Database for analytic purpose

Figure 5: Sample System Design [Source]

This step includes primary architectural components that drive the overall system design of the product. We specify visual notations and lightweight methods to describe software architecture. There are different tools to visualize software design and architecture, for example, C4Model, TOGAF, Archimate, etc.

The diagram below represents the different layers of a C4 model for a sample application.

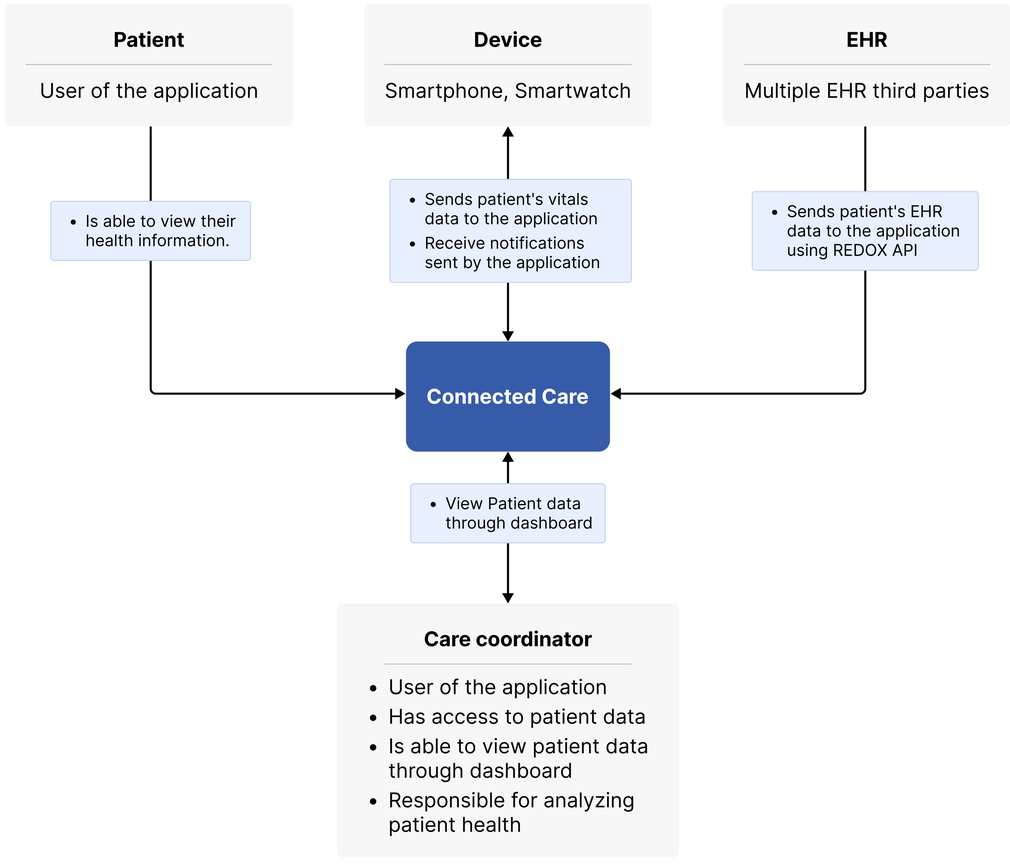


Figure 6.1: Level 1 - Context Diagram of Connected Care Application

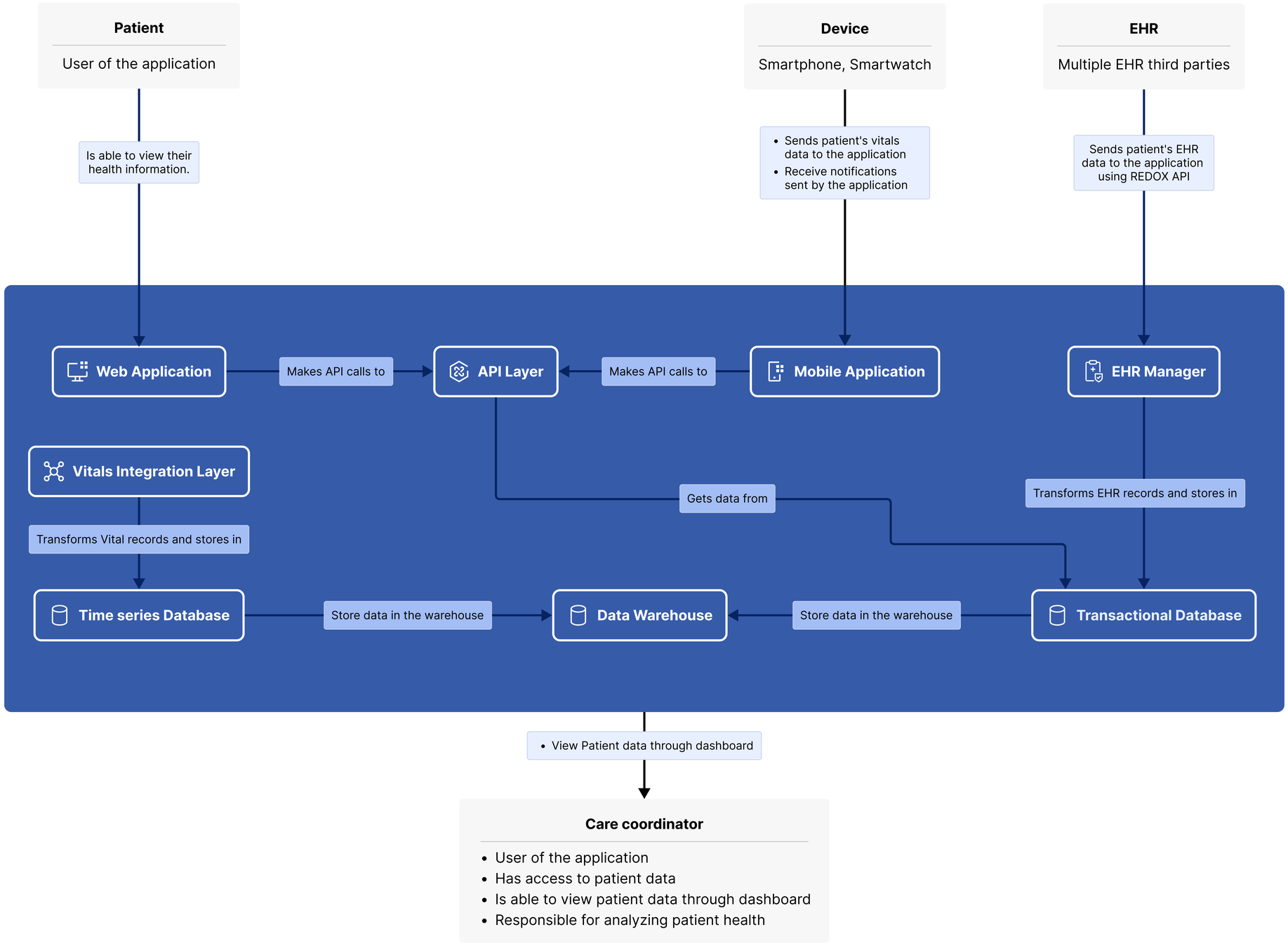


Figure 6.2: Level 2 - Containers Diagram of Connected Care Application

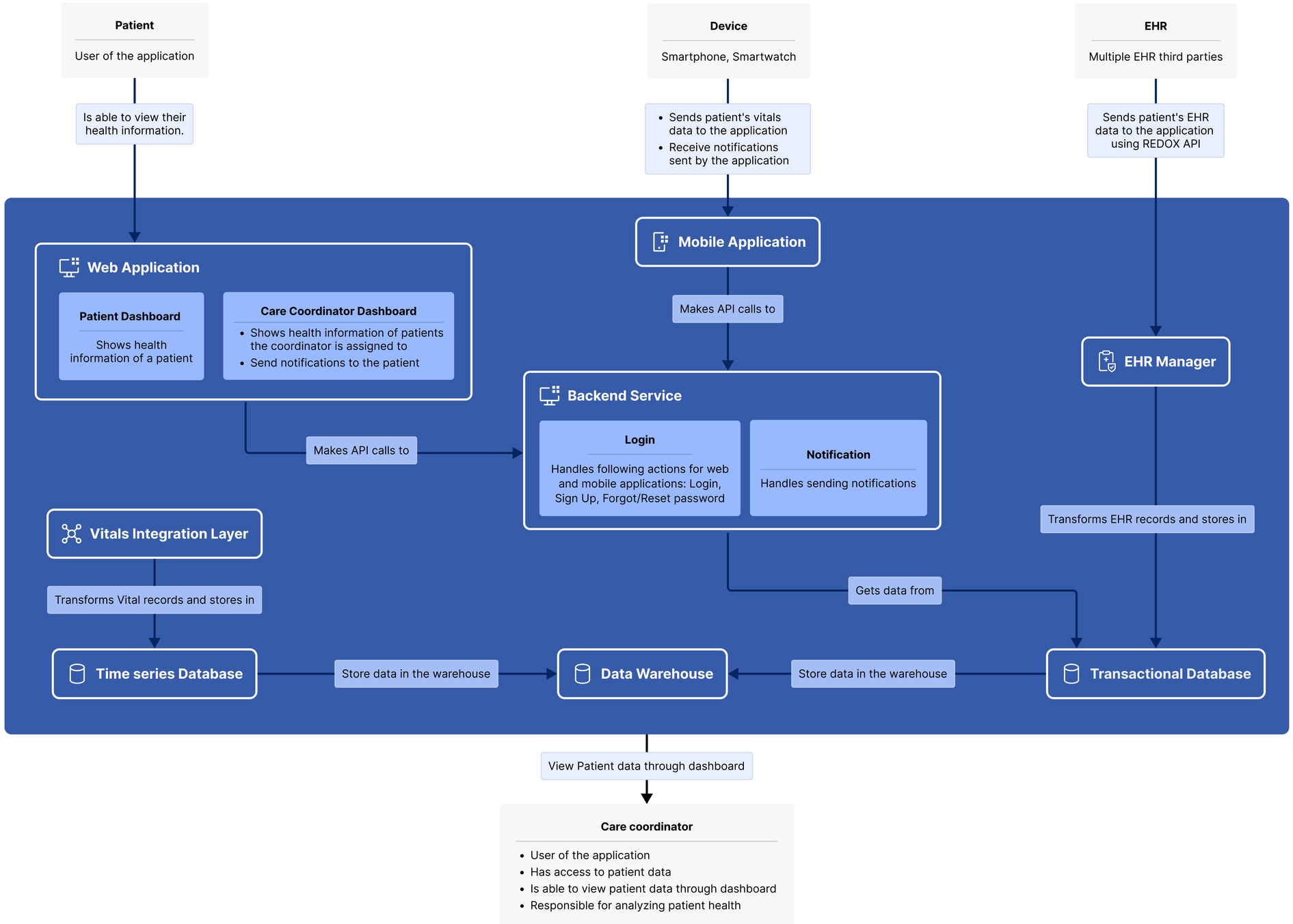


Figure 6.3: Level 3 - Contexts Diagram of Connected Care Application

Microservice Design Patterns

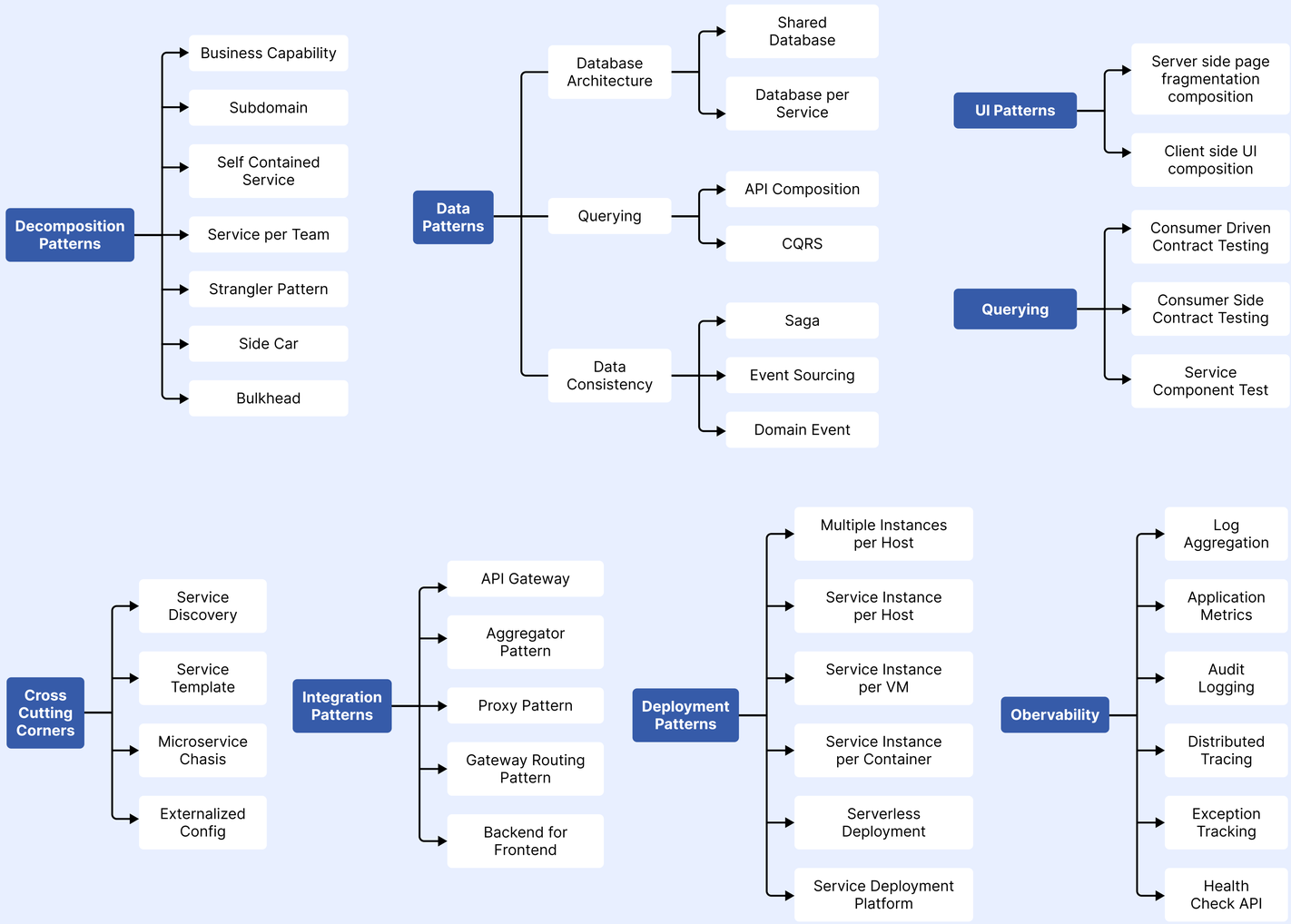


Figure 7: Microservice Patterns

Diagram Checklist

Notation, notation, notation

A software architecture diagram review checklist

General

Does the diagram have a title?	Yes	No
Do you understand what the diagram type is?	Yes	No
Do you understand what the diagram scope is?	Yes	No
Does the diagram have a key/legend?	Yes	No

Elements

Does every element have a name?	Yes	No
Do you understand the type of every element? (i.e. the level of abstraction; e.g. software system, container, etc)	Yes	No
Do you understand what every element does?	Yes	No
Where applicable, do you understand the technology choices associated with every element?	Yes	No
Do you understand the meaning of all acronyms and abbreviations used?	Yes	No
Do you understand the meaning of all colours used?	Yes	No
Do you understand the meaning of all shapes used?	Yes	No
Do you understand the meaning of all icons used?	Yes	No
Do you understand the meaning of all border styles used? (e.g. solid, dashed, etc)	Yes	No
Do you understand the meaning of all element sizes used? (e.g. small vs large boxes)	Yes	No

Relationships

Does every line have a label describing the intent of that relationship?	Yes	No
Where applicable, do you understand the technology choices associated with every relationship? (e.g. protocols for inter-process communication)	Yes	No
Do you understand the meaning of all acronyms and abbreviations used?	Yes	No
Do you understand the meaning of all colours used?	Yes	No
Do you understand the meaning of all arrow heads used?	Yes	No
Do you understand the meaning of all line styles used? (e.g. solid, dashed, etc)	Yes	No

Figure 8: Software Architecture Diagram Checklist [Source]

Architect

This stage is where major architectural planning and decision-making are done. It consists of shaping and implementing the majority of findings from the technical discovery stage.

ER Diagram/Data Model

Once we have clarified most requirements, the ER Diagrams that show Entities, Attributes, and their relationships are created. Tools like Lucid Charts and PGAdmin can be used for ER diagram generation.

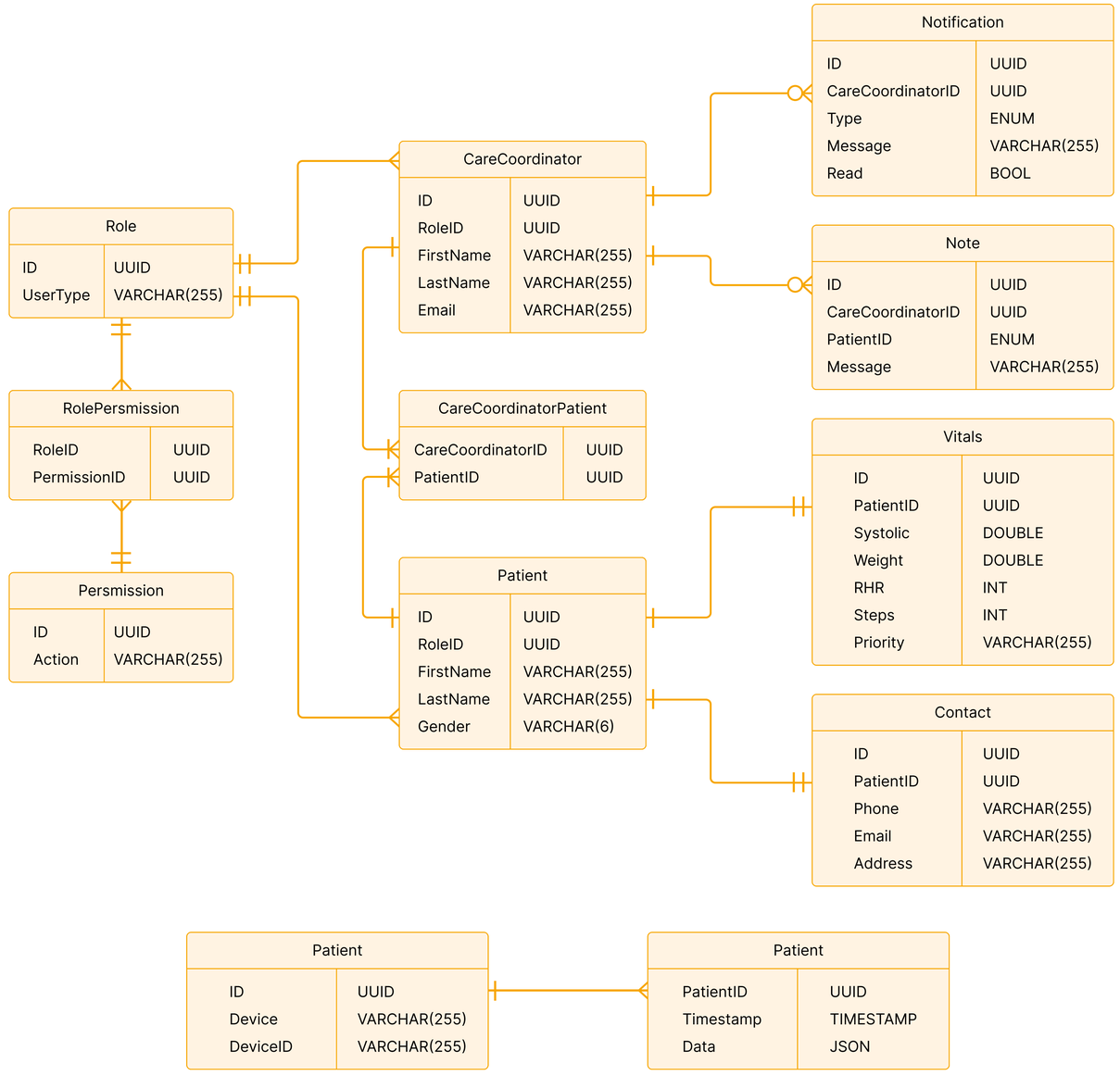


Figure 9: ER Diagram

API Design

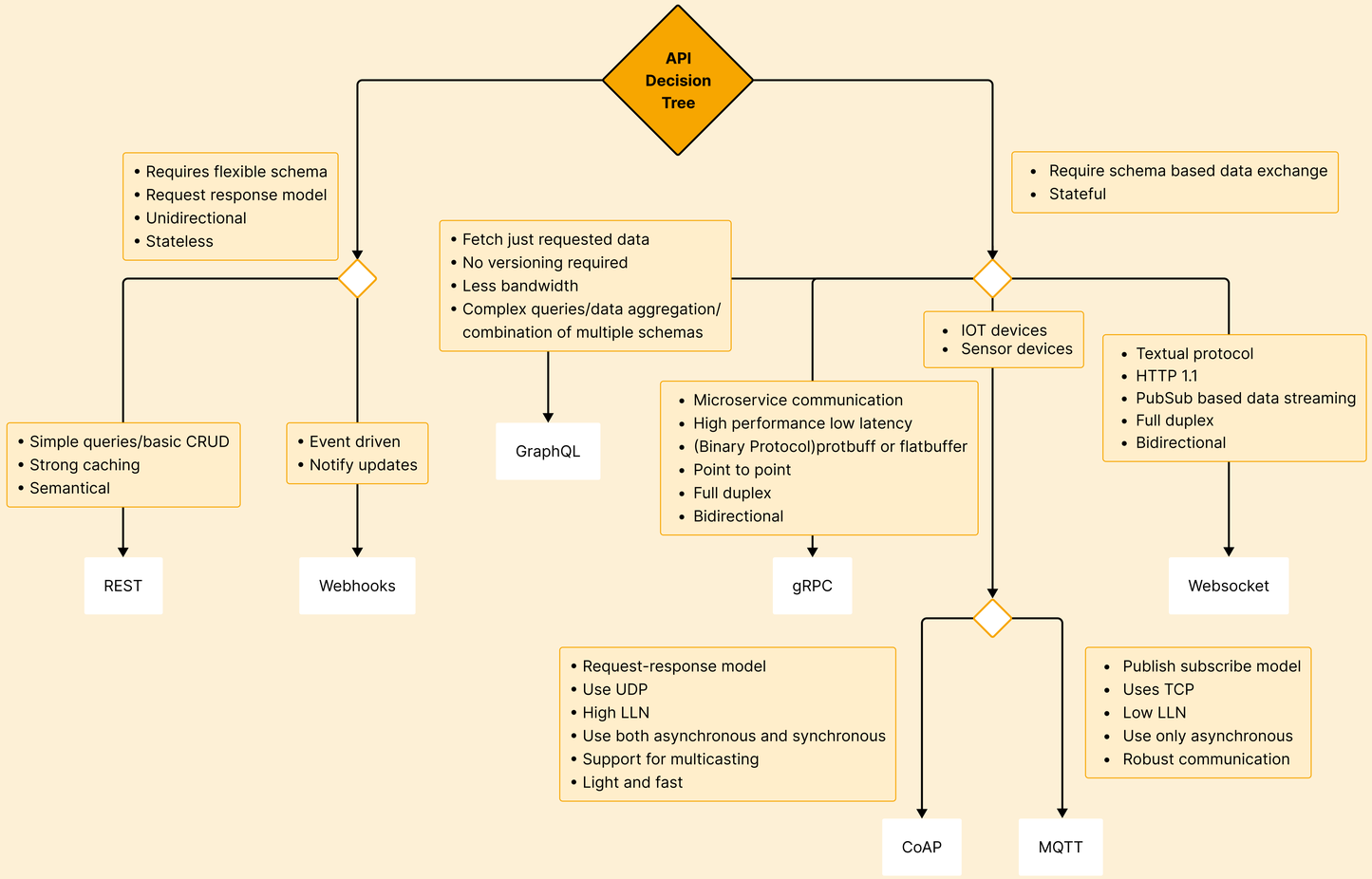


Figure 10: API Technology Decision Tree

The above figure represents various possible API technology based on different natures, volumes, and data integration.

As for Connected Care, we will be going with REST API as it provides good flexibility. Data is not tied to resources or methods, so REST can handle multiple types of calls, return different data formats and even change structurally with the correct implementation of hypermedia.

Tentative High-Level Base APIs

```
CareCoordinator:
GET /api/{version}/organization/{orgId}/care-coordinator

Patient:
GET /api/{version}/organization/{orgId}/patient

Notification:
GET /api/{version}/organization/{orgId}/notification
POST /api/{version}/organization/{orgId}/notification
DELETE /api/{version}/organization/{orgId}/notification/{id}

Notes:
GET /api/{version}/organization/{orgId}/note
POST /api/{version}/organization/{orgId}/note
DELETE /api/{version}/organization/{orgId}/note/{id}
```

Note: We will get Vitals and EHR records from the third-party services API that we use.

User Flow/User Journey

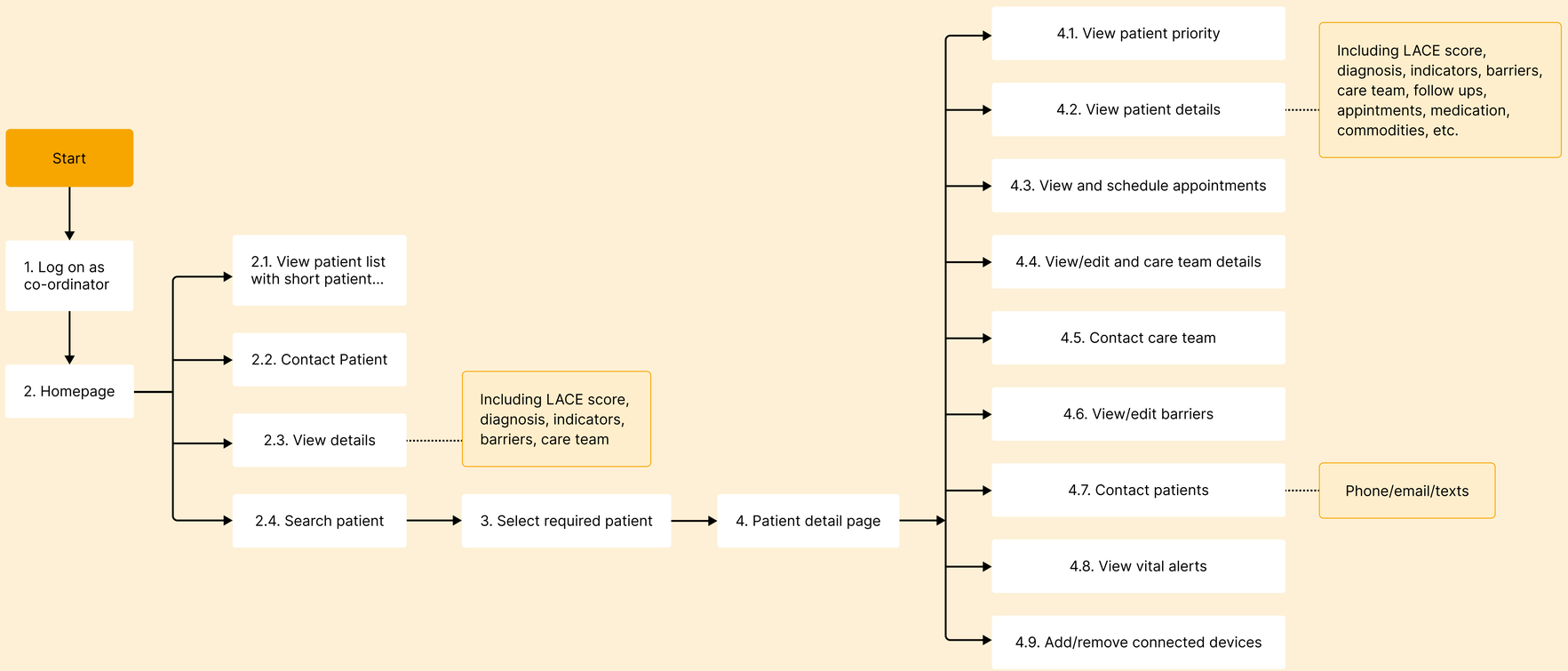


Figure 11: User flow diagram for Care Co-ordinator

Tools and Technologies [source]

After the majority of requirements are finalized and an abstract view is ready, we decide on implementation details for each layer. This stage will consist of deciding the favorable tools and technologies across different layers like:

1. Frontend

This step includes a selection of front-end tools and technologies. When choosing a front-end technology, we should ensure that it works well with the technologies used in the backend. The main factors we need to regard when choosing a front-end technology for a project are the goals and complexity of the project and matching the requirement with technology.

Suppose the project you are building has a simple, read-only client-side requirement. In that case, a traditional web application can be used, where your application needs to function in a browser without javascript support and the most of the application logic is on the server-side. However, if your application must expose a rich user interface with many features and has application logic on the web browser, then a single-page application(SPA) can be used. Angular, React, Ember and Vue are popular frameworks used in developing SPA.

Suppose you are developing a simple, lightweight application and involves a small engineering team. In that case, a monolith architecture can be used, which will be much easier to build, make changes, and deploy. If the application you are building is complex, constantly evolving, and involves many team members, then microservice architecture would be ideal.

> ! In the Connected Care app, we have a rich UI with lots of features. Since a dashboard screen needs continuous components updates, we will be using SPA, particularly React, as it is an efficient, declarative, and flexible open-source JavaScript library for building simple, fast, and scalable front-ends of web applications. We can write highly customizable, modular and testable components and have strong community support.

2. Backend

This step includes the selection of backend technologies like Nodejs, Django, Etc, as well as deciding any backend architecture or patterns to follow. The patterns may include microservices, serverless functions, caching layers, or cloud-native design patterns.

Here are some of the backend architectures to follow:

2.1 Serverless Architecture

It is the approach in which a developer can run the backend services on demand without worrying about the infrastructures running behind the scenes. In this approach, the code is invoked by serverless runtime as often as needed. The consumer is then billed by the number of times and duration of the function's runtime.

There are two primary service models for the backend in serverless architecture: Function as a Service (FaaS) and Backend as a Service (BaaS).

FaaS is the core technology in a serverless architecture. AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions are prime examples of this service model.

BaaS is the service model in which all the necessary aspects of the backend, such as database management, cloud storage, user authentication, and hosting, are all outsourced as a package. Like FaaS, a developer need not worry about the infrastructure underneath. Back4App and Firebase are some examples of this service model.

2.2 Container Architecture

It is the approach where each application and its dependencies are packaged into a single “container” and can be deployed in any operating system. They are more lightweight and efficient when compared to Virtual Machines.

Containers give developers complete control over the environment, programming language, and framework. However, the management of the containers adds a whole layer of complexity to this architecture. Software like Docker Swarm and Kubernetes help manage the containers at a large scale.

2.3 Microservices Architecture

It is the approach where the application contains loosely coupled, highly maintainable, and testable services which can be independently deployed. This architecture allows developers to use different programming languages, frameworks, and data storage technologies. Also, each microservice is relatively small and independent, allowing easier understanding and faster deployments. On top of that, the system built following this architecture is fault-tolerant. Having one service down does not affect the entire system.

However, a distributed system brings management, operation and overall architectural complexities.

Also, microservices are costly in terms of infrastructure. Each service needs its own independent set of infrastructures to maintain isolation, which can add up significantly.

Which works best in what scenario?

Serverless:

- When you have short-lived, event-driven workloads
- For the applications where most of the logic is on the client side
- When you need lower overhead costs in terms of infrastructure
- Rapidly changing and scaling applications

Containers:

- When you need more control over the application's environment
- For the applications that need to be moved between local and cloud environments
- Real-time applications that need applications to run for a long time (e.g.: in case of using web sockets)

Microservices:

- For deploying new features and functionalities without affecting the whole system
- To enable a high degree of team autonomy
- When an application needs to be rewritten using new technology stacks

So when it comes to choosing, these approaches are not mutually exclusive. You can use serverless for some parts of the application, such as sending notifications and containerizing the parts that need to be available 24/7. Similarly, microservices are not an isolated, standalone pattern. Both containerization and serverless approaches can be used in different services within.

Each approach has its shining moments and trade-offs. The best way would be to keep an open mind, know what fits when and implement it accordingly.

Authentication and Authorization

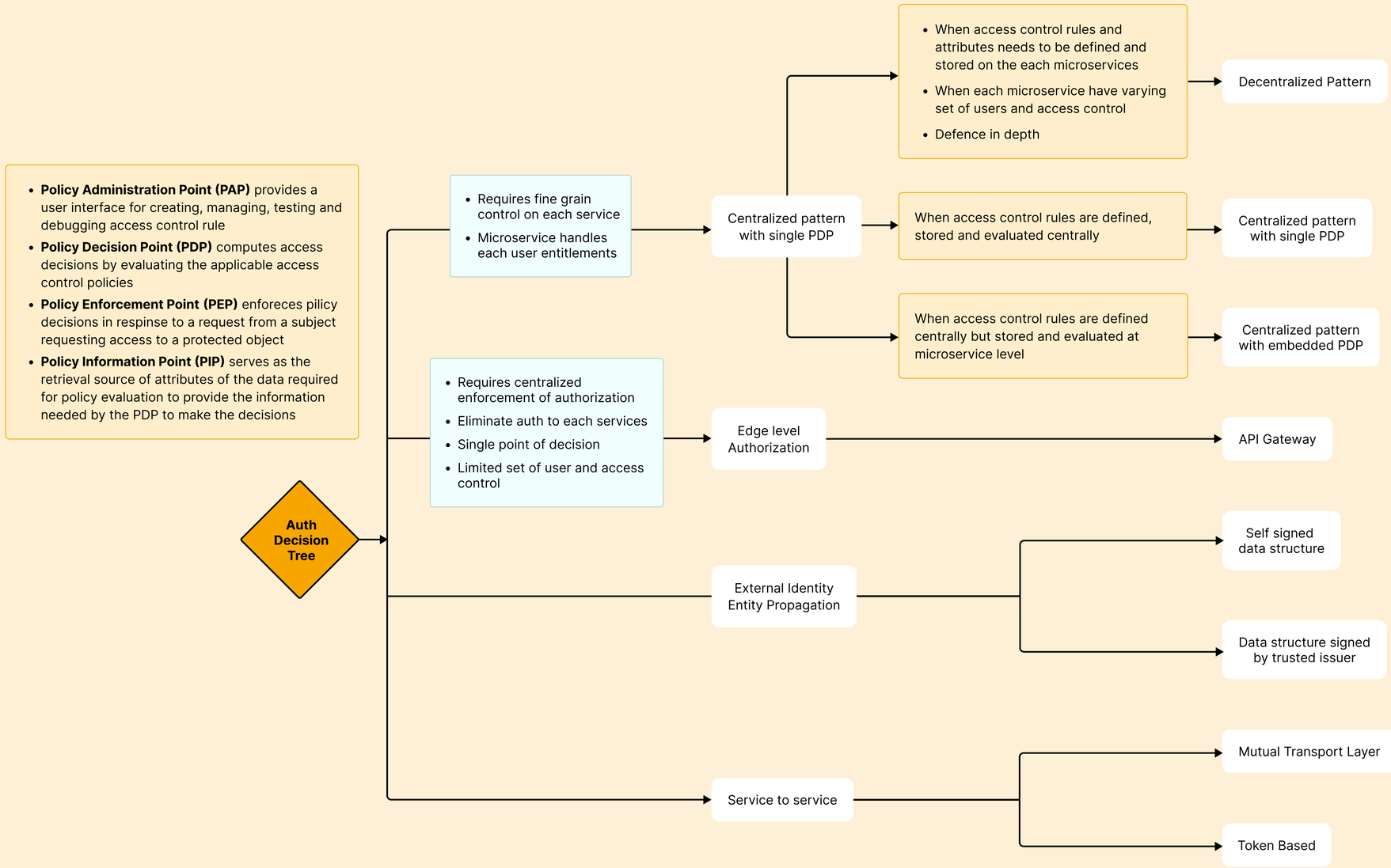


Figure 12.1: Authentication and Authorization

This step includes deciding on best-fit authentication and authorization patterns.

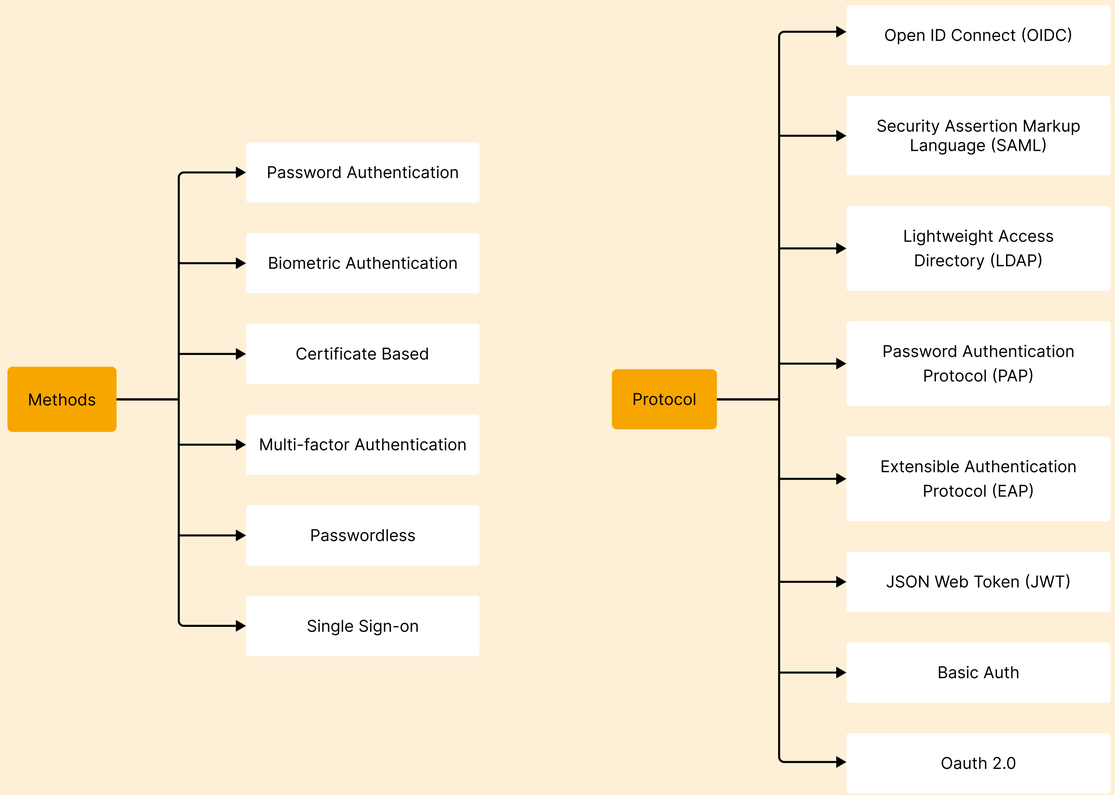


Figure 12.2: Authentication and Authorization

3. Database

This step includes the selection of database choices that best fit the requirements. It may require multiple databases as well as considerations of systems according to data for the application. E.g., persistent data, session data, Etc. There is a wide range of database technologies of different natures; relational (SQL) is usually for tabular structural related data or non-relational (NoSQL) for dynamic nature such as a key store, document-based. This step will also include decision-making of database patterns, whether multi-tenant, database per service or any other database patterns. The scalability and the different nature of data should be considered when recommending the tool or the flow of the data.

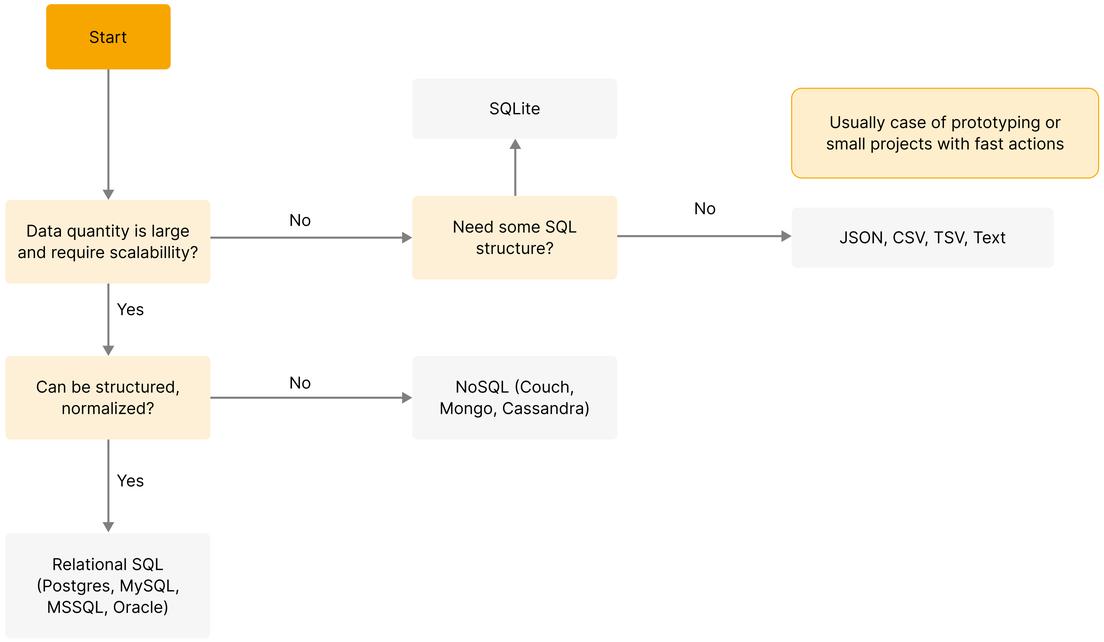


Figure 13.1

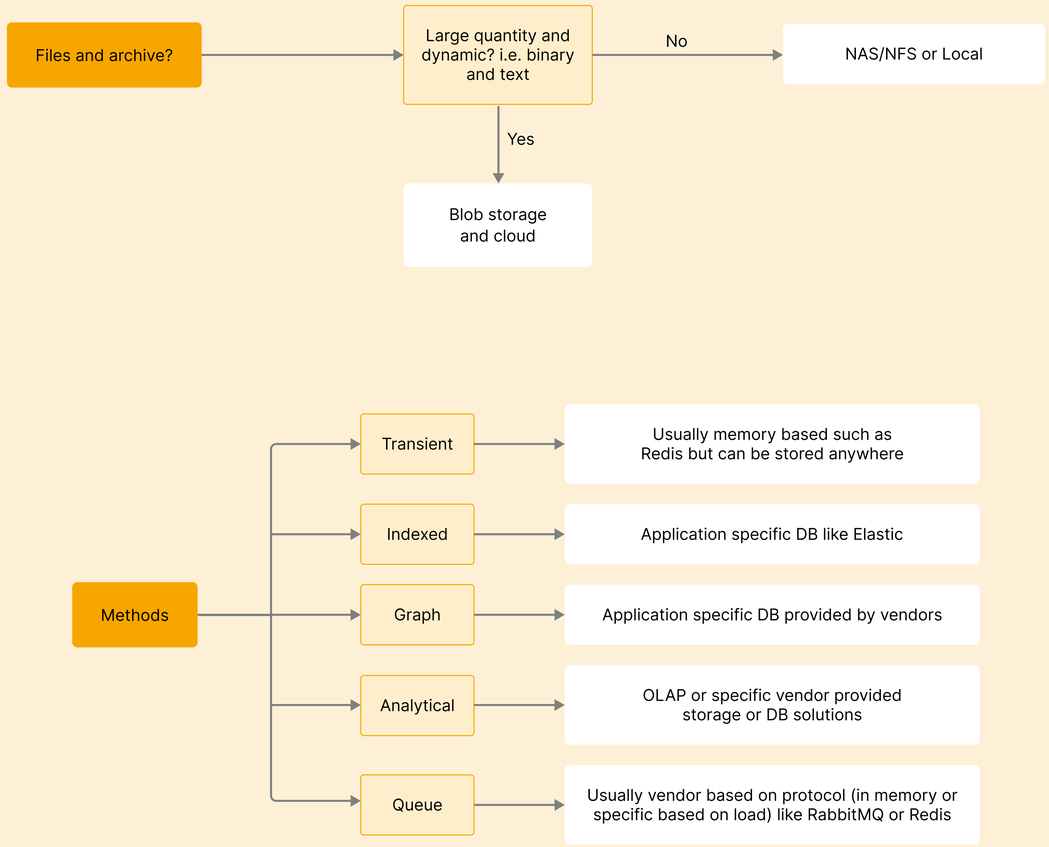

Files and archive?
Large quantity and dynamic? i.e. binary and text
No
NAS/NFS or Local
Yes
Blob storage and cloud
Methods
Transient
Usually memory based such as Redis but can be stored anywhere
Indexed
Application specific DB like Elastic
Graph
Application specific DB provided by vendors
Analytical
OLAP or specific vendor provided storage or DB solutions
Queue
Usually vendor based on protocol (in memory or specific based on load) like RabbitMQ or Redis

Figure 13.1

Select your database using PACELC Theorem

PACELC theorem is an extension of the **CAP (Consistency, Availability, Partition Tolerance)** theorem. It states that in the case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

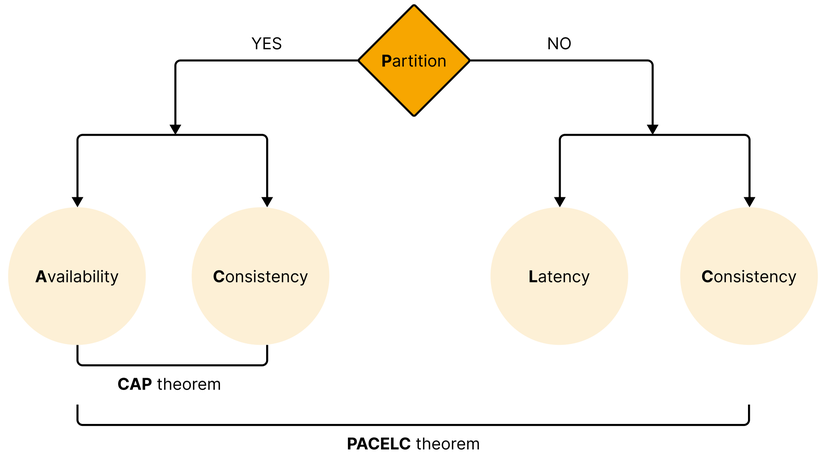


Figure 14.1: PACELC theorem

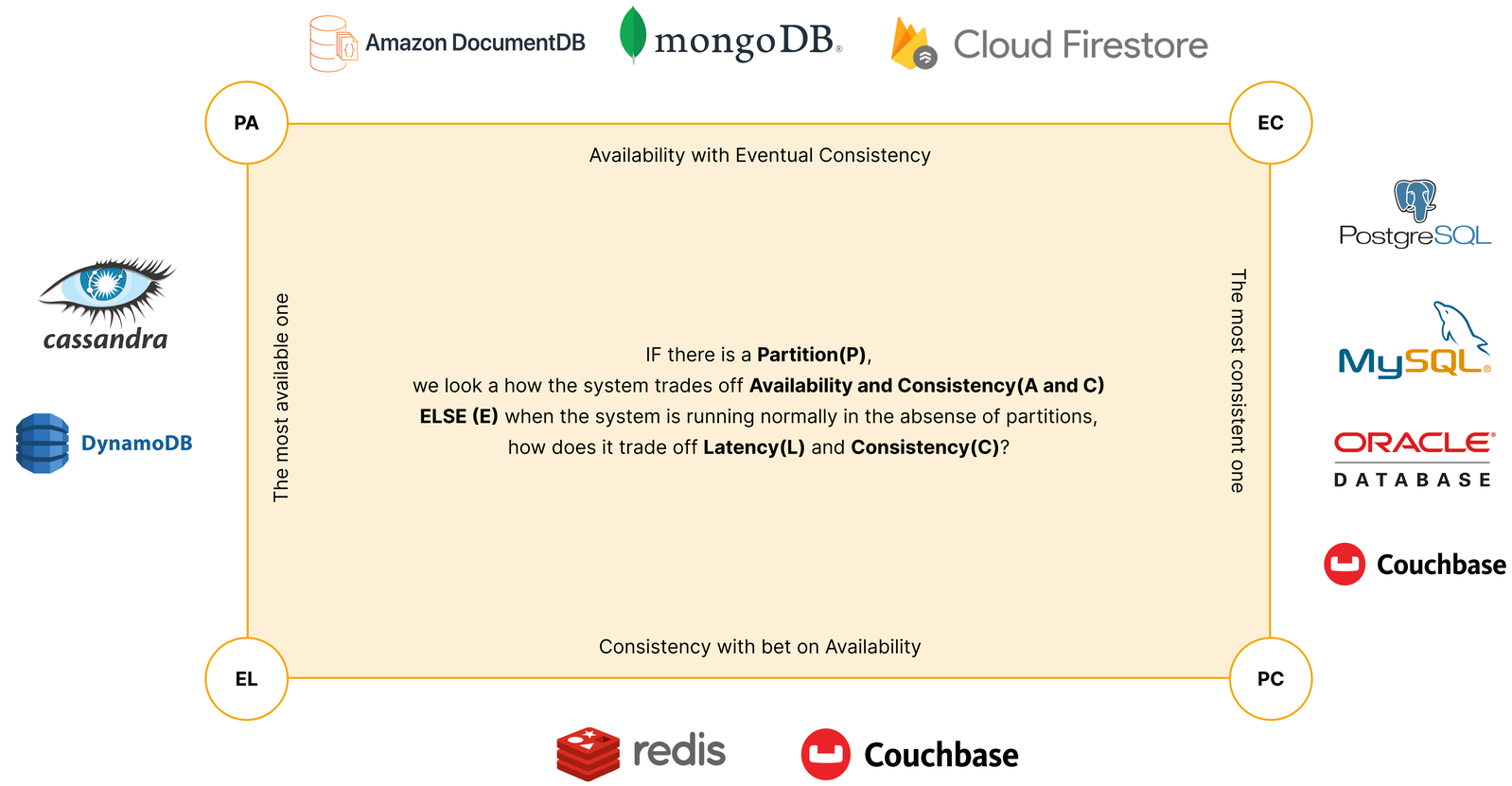


Figure 14.2: PACELC theorem

In the case of the “Connected Care App,” we will be following the Polyglot persistence concept that refers to using multiple data storage technologies for varying data storage needs across an application or within smaller components of an application.

Using a single database to satisfy a program's requirements can result in a non-performant, "jack of all trades, master of none" solution. Relational databases, for example, might work when the data is smaller but becomes problematic when the data involved grows larger. A graph database might solve the problem of relationships in the case of the graph model, but it might not solve the problem of database transactions.

Hence we will use relational DB (Postgres) for transactional data, and for Vitals/EHR records which are immutable and time-based events, we will use time series DB.

4. File Storage

File Storage is the choice of storing multiple types of files, i.e., uploads/downloads and assets and binaries in the cloud. There may be three choices for cloud file storage i.e

- File Storage: Usually Network Attached Storage(NAS) or File Systems in VM. Amazon EFS can be such a choice.
- Block Storage: Usually dedicated scalable storage such as Dedicated Attached Storage(DAS) or Storage Area Network (SAN). E.g., Amazon EBS
- Object Storage: The most famous storage option for scalability, availability, and ease of use. E.g., Amazon S3

In most cases, we will be using Object Storage for applications unless there is a specific recommendation for File or Block Storage. The container-based microservice and sharing of file resources between the services also make a choice for object storage more valid. This choice should consider the security and read/write operations well.

Also, object storage and CDN make static assets such as images, javascript, and CSS more scalable and shareable.

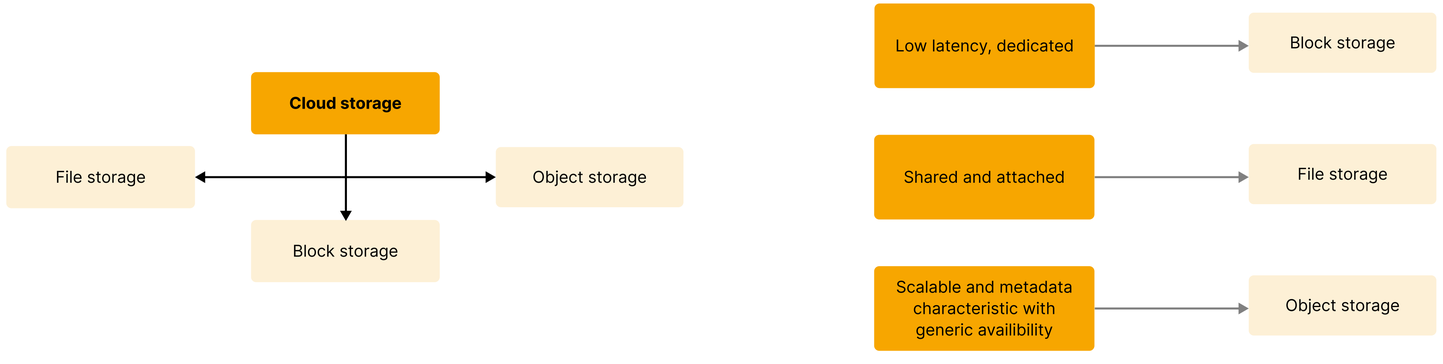


Figure 15: File Storage

ETL Pipeline

Modern data can have different sources as well as events. Also, data is usually different, so it needs conversion and structure according to need. There are different tools according to requirements, such as streaming or batch processing, single process or multiple-step by steps, and SQL-like processing or raw data. The pipeline and movement of data also depend on robustness and fault tolerance. Also, there are tools for drag and drop, which can be used as per requirement.

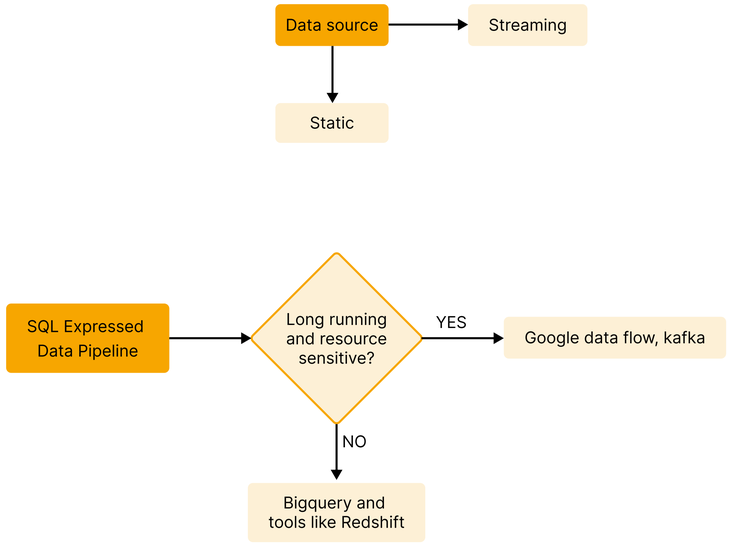


Figure 16

Decision:

It makes sense to use AWS Glue as a tool for ETL. All our infrastructure resides in AWS. Managing as well as using Glue would be easier for our other infrastructures as well. In addition to that, here are some of the criteria that have made us choose Glue:

- Integration with other AWS Services.
- Most of our data might be streaming with fast-paced inputs from Smart Devices, Health devices, and Glue easily scales with the volume of incoming requests/data.
- We chose a wholesome tool in Glue for our ETL process, which is easier to manage than a single ETL component for different functionality.

5. Logging and Monitoring

Logging and monitoring should provide details on each incoming request entry, trace any log levels inside the application, health check the system, and monitor and alert in case of any exceptions. This step includes the selection of technologies that helps the visualization of application logs, metrics, distributed tracing, co-relating, and structuring logs. Zipkin, ELK, Datadog, Grafana, Prometheus, and other commonly used tools for logging and monitoring.

6. Security [source]

This step includes considering security aspects on different software layers, including encryption at rest and transit, security policies, VPNs, API securities, encryption libraries, user roles management, etc. Security decisions can be considered based on the famous seven application security principles.

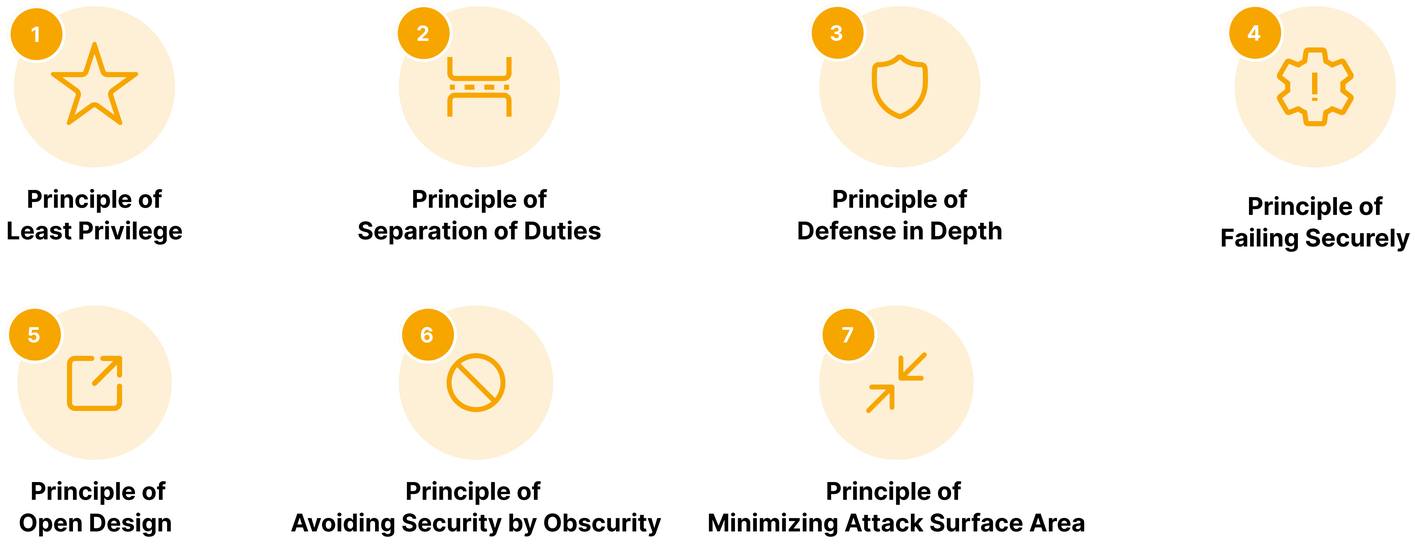


Figure 17: Famous 7 application security principles [source]

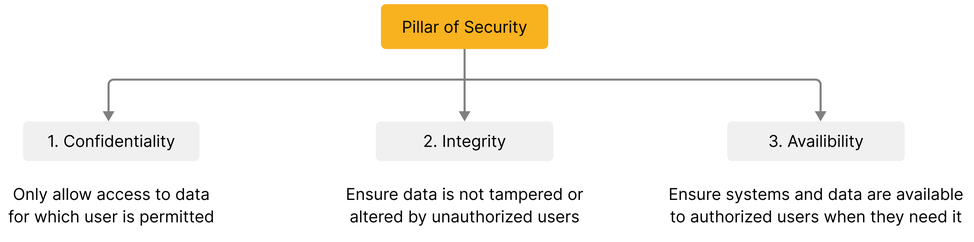


Figure 18: Pillars of security

7. Network Selection

This step includes identifying what type of network is suitable for the given system, in-house/cloud/hybrid, as well as selection/designing of the network's internal components. Different network layer considerations include checking the need for a separate private network, size of the network, private and public subnets, accessibility to and from the internet, need for NAT gateways, VPN, private and public DNS, and much more. Another essential part is to clear out which resources of the system reside where, like, does the database needs to be accessible via the internet? Some of the questions to answer (answers are for "yes" cases):

1. Does the application require a dedicated network?

2. Will your application need IaaS (VMs), PaaS(DBs), and other that requires a self-managed network?
⇒ A dedicated network is required to host the application /components.

3. Will the application reside in Cloud?

4. Does the application need to access private on-premises resources or other private data centers?
⇒ Create a S2S VPN connection. If it requires high bandwidth and reliable connection setup Direct (Wired) Connection (AWS Direct Connect).

5. Does the application/components need to be exposed to the internet/public?
⇒ Internet Gateway

6. Does the application's private component need to access the internet?
⇒ NAT Gateway

7. Does the application completely resides behind a private network and is required to be accessed by multiple clients?
⇒ May require VPN Server with managed Clients (OpenVPN).

8. Does the application need to access (other) cloud resources?
⇒ Virtual Network Peering.

9. Does the application have separate public and private accessible components?
⇒ Multizone Public/Private Subnets.

10. Do you require custom DNS management?
⇒ DNS Service (Route 53)

11. Does the application (or any component) require a static public IP Address?
⇒ Buy/Subscribe to public IPs.

12. Is system based on micro architect and requires service level access blocking?
⇒ Network micro-segmentation

13. What level of network security is a priority in the application (DDOS, WAF, IDS, IPS etc)?

14. What level of network feature is a priority in the application(Load Balancing, scalability, availability, logging, monitoring, diagnosis, etc)?

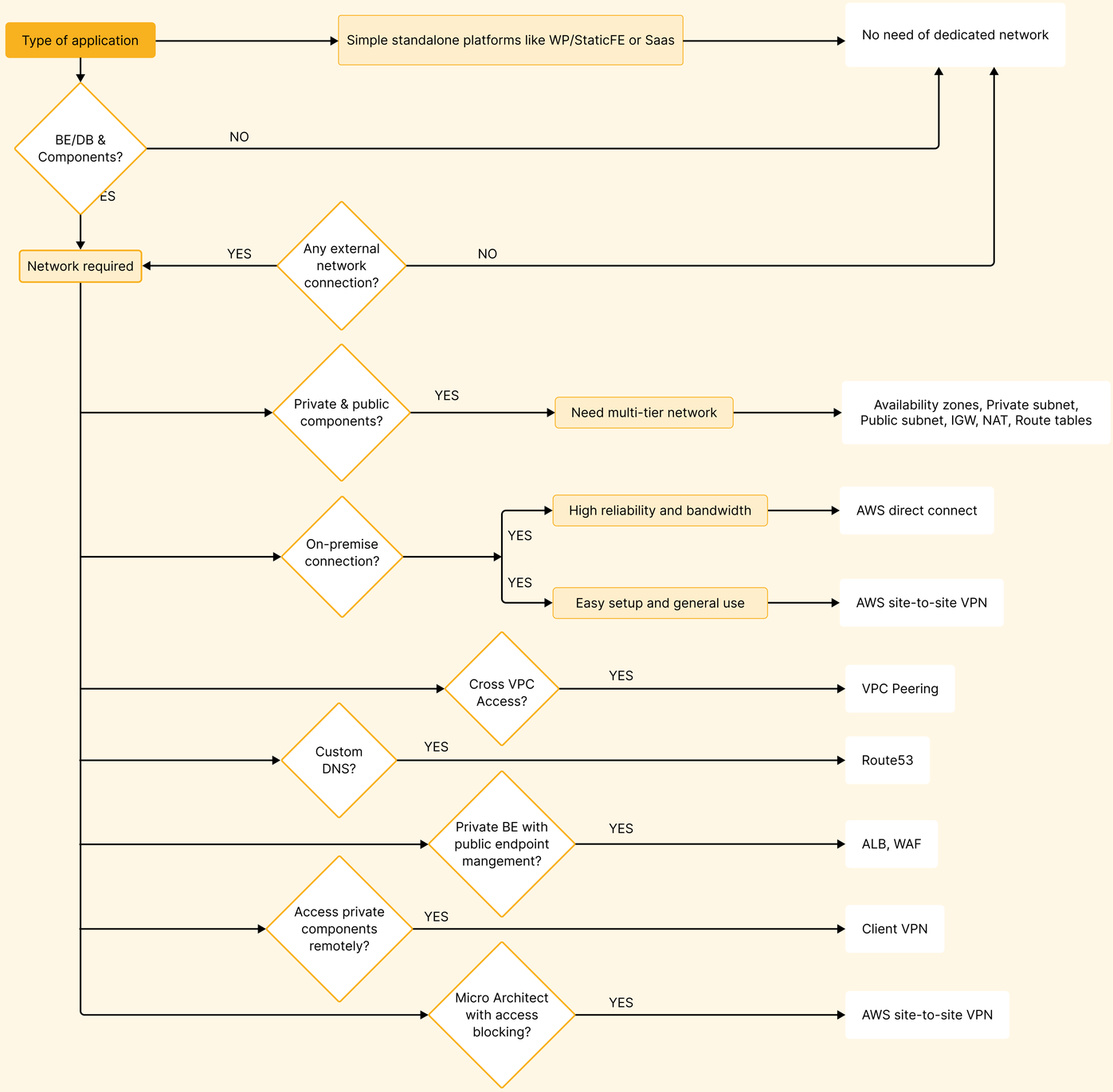


Figure 19: Network selection

8. Disaster Recovery and Backup Plan

This step includes how the architecture will handle the disaster and its recovery mitigation. Usually, this should include the procedure to handle or recover system failure, loss of data, and downtime. Some areas to consider for Disaster Recovery are:

1. Infrastructure as a Code and other automation scripts

2. Plan to deploy a full fresh system or system from existing data

- Networks
- Application Server
- Database Server
- CDN and file storage
- Connection to third-party services

3. Backup and recovery of data

- Database.
- Object/File Storage

4. Disasters to consider

- Network failure
- Application failure
- Application/Web Server failure
- Database failure
- Host machine/OS failure
- (Cloud) AZ/Region/Whole Cloud failure
- (On-prem) Hardware/Internet/Power failure

5. Time frame to complete each stage in case of disaster.

Disaster recovery is based on the Recovery Time Objective (RTO) and Recovery Point Objective (RPO). RTO helps us understand how long the system can afford to be offline or how long it should take to recover its functionality. Whereas RPO denotes how much data the system can handle to lose, we might need near real-time data to be recovered or could withstand the loss of specific interval data. The DRP should identify the middle ground between RPO and RTO.

Figure 20: Recovery objectives: RTO and RPO [source]

9. Infrastructural Architecture

Once the tech stacks are finalized, the holistic view of cloud infrastructure is made, representing the interconnection between various system components. Infrastructural Architecture should include a representation of Network design, Client/User Server Connection Flow, Application, Database, storage services, external services, and connectivity between them. It is also better to show how decoupled services interact, whether the server is public or private, network boundaries, recommended instance sizes, etc.

The following image represents infrastructure architecture in the AWS cloud. This will vary based on cloud selection, whether it's the Google cloud platform, Azure, On-Premise, or the on-premises solution.

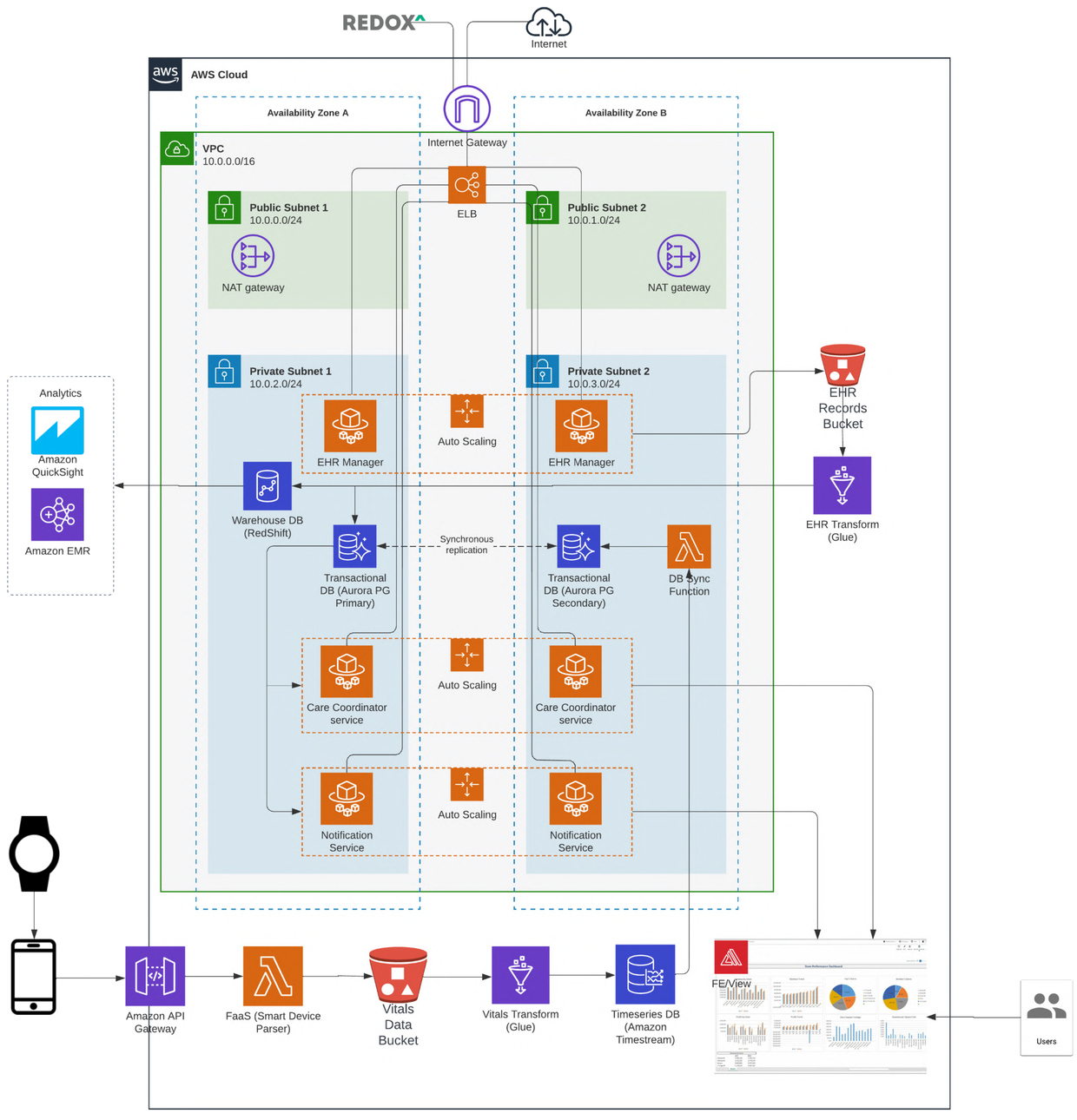


Figure 21: TechDND Doc [source]

Estimate

This is the stage where we will be estimating all the work that needs to be done. We are clear on the requirements, system architecture, and technologies by this time. The requirements are broken down into Epic, Stories, Tasks, and estimated. The requirements are grouped into milestones, and the estimates can be done using different ways like story points, hours, and T-shirt sizing.

While these estimations are subject to change as per the change in requirements, and R&D required, these numbers will provide a close approximation of the timeline and cost.

SN	Epic	Task	Description	Priority	Tshirt Size (By Time)	StoryPoint/ Hours	Buffer/ hours	Risk	Remarks	References	
T0001	ETL: Redox to Warehouse	Boilerplate setup for AWS Glue and related codebase	Boilerplate script includes scripts to setting things up, repository, folder structure, base entities in code, linters, loggers, exception handlers, autodoc generators (if any), CI/CD, Licenses, containerization scripts and so on	High	M	X	0.00				
T0002		Write script to create Warehouse DB structure (aka Migration script)	Include scripts that could be abstracted with libraries such as alembic (Python), knex (javascript) or basic SQL scripts. The purpose of these scripts is to create the schema or skeleton of the database. It should follow the ER diagram, if available	High	S	X	0.00				
T0003		Write Glue service to read from the Redox service and write to the Data Warehouse	The Glue service should read from the Redox Proxy Service and convert the data to a format specified for the warehouse. The format can be a direct database representation of the API structure, which can be transformed later in the warehouse itself	High	L	X	8.00				
T0004	ETL: Device Data to Time Series Database	Boilerplate setup for AWS Glue and related codebase	Boilerplate script includes scripts to setting things up, repository, folder structure, base entities in code, linters, loggers, exception handlers, autodoc generators (if any), CI/CD, Licenses, containerization scripts and so on	High	M	X	0.00				
T0005		Write script to create Time Series DB structure (aka Migration script)	Include scripts that could be abstracted with libraries such as alembic (Python), knex (javascript) or basic SQL scripts. The purpose of these scripts is to create the schema or skeleton of the database. It should follow the ER diagram, if available	High	S	X	0.00				
T0006		Write Glue service to read from the Device service and write to the ~~Data~~	The Glue service should read from the Device Proxy Service and convert the data to a format specified for the warehouse. The format can be a direct database	High	L	X	8.00				

Figure 22.1: Sample Estimations [source]

N82

		START DATE													
Date		31-Jan	7-Feb	14-Feb	21-Feb	28-Feb	7-Mar	14-Mar	21-Mar	28-Mar	4-Apr	11-Apr	18-Apr	25-Apr	2-Ma
Weeks		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Sprint		Sprint 0		Sprint 1		Sprint 2		Sprint 3		Sprint 4		Sprint 5		Sprint 6	
Project Manager		50%	50%	50%	50%	50%	50%	25%	25%	25%	25%	25%	25%	25%	25%
Developer 1		100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Content Writer		0%	0%	100%	100%	100%	100%	50%	50%	50%	50%	50%	50%	50%	50%
QA		100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Developer 2		100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Developer 3		100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Team Lead		100%	100%	100%	100%	50%	50%	50%	50%	50%	50%	50%	50%	50%	50%
Developer 4		100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Developer 5		100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
UI/UX Engineer		25%	25%	25%	25%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Tasks	Sprint 0		Sprint 1		Sprint 2		Sprint 3		Sprint 4		Sprint 5		Sprint 6	
	Common component														
	ETL: Redox to Warehouse														
	ETL: Device Data to Time Series Database														
	ETL: Data Integration of Warehouse with Transactional DB														
	Write Help Contents														
	Webservice: Care Coordinator														
	Webservice: Notification Service														
	Application: Analytic features														

Figure 22.2: Sample Project Plan [source]

Infrastructure Cost Estimate

An overview of the estimated cost will be derived for the Infrastructure Architecture. This is a tentative cost based on the estimated user base, incoming requests, and use of resources like database, file server, app server, and other cloud components. It is preferred to separate the overall cost upon each deployment environment like Dev, UAT, and Production. Below is a simplified example of cost estimation.

TOTAL COST (YEARLY)	49,546$

DEV(us-west-2)	50												
SERVICE	S3 (API Backend)				HTTP API GATEWAY (APP Backend)				ECS service (APP Backend)				
	100	300	0.1		500	1							
	RESOURCE	Unit	Cost Per Unit	COST	RESOURCE	Unit	Cost Per Unit	COST	RESOURCE	Unit	Cost Per Unit	COST	
	Buckets	2	0	0	Gateways	2	0	0	Tasks	2	0	0	Re
	Storage(GB)	20	0.023	0.92	Api Requests	1520500	0.000001	3.041	Memory(GB)	2	0.004445	12.9794	St
	Write Requests	152050	0.000005	0.76025					vCPU	1	0.04048	59.1008	
	Read Requests	456150	0.0000004	0.18246									
	Data Out to Internet(GB	1.46484375	0.09	0.263671875									
Cost (Monthly)				1.86271				3.041				72.0802	
Cost (Yearly)				22.35252				36.492				864.9624	
Total (Yearly)	7889.048791												

QA(us-west-2)	50												
SERVICE	S3 (API Backend)				HTTP API GATEWAY (APP Backend)				ECS service (APP Backend)				
	100	300	0.1		500	1							
	RESOURCE	Unit	Cost Per Unit	COST	RESOURCE	Unit	Cost Per Unit	COST	RESOURCE	Unit	Cost Per Unit	COST	
	Buckets	2	0	0	Gateways	2	0	0	Tasks	2	0	0	Re
	Storage(GB)	20	0.023	0.92	Api Requests	1520500	0.000001	3.041	Memory(GB)	2	0.004445	12.9794	St
	Write Requests	152050	0.000005	0.76025					vCPU	1	0.04048	59.1008	
	Read Requests	456150	0.0000004	0.18246									
	Data Out to Internet(GB	1.46484375	0.09	0.263671875									

Figure 22.3: Infrastructure Cost Estimations [source]

Reference

1. The non-functional requirements cheat sheet
2. ISO/IEC 25010 software and data quality
3. The C4 Model
4. OpenAPI Specification
5. Software Architecture Quality Attributes
6. Quality attributes in Software Architecture
7. CAP and PACELC theorems in plain English

Leapfrog Technology, Inc is a global full-service technology services company. Founded in 2010, we have offices in Seattle, Boston and Kathmandu. With a high performing team of over 400 engineers, designers, and product analysts, we help companies solve technology innovation and digital transformation challenges. We have instilled a culture of continuous learning in our people. Our mission is to be the best and most trusted technology services company to execute clients' digital vision.

info@lftechnology.com
lftechnology.com

VER 2022.08.15

Technical Discovery Handbook

Guideline to effective technical discovery