# Pill Dispenser & Reminder - Requirements

November 3rd, 2019

---

## Group 11

Mario Bocaletti

Henry Clay

Corey Nielsen

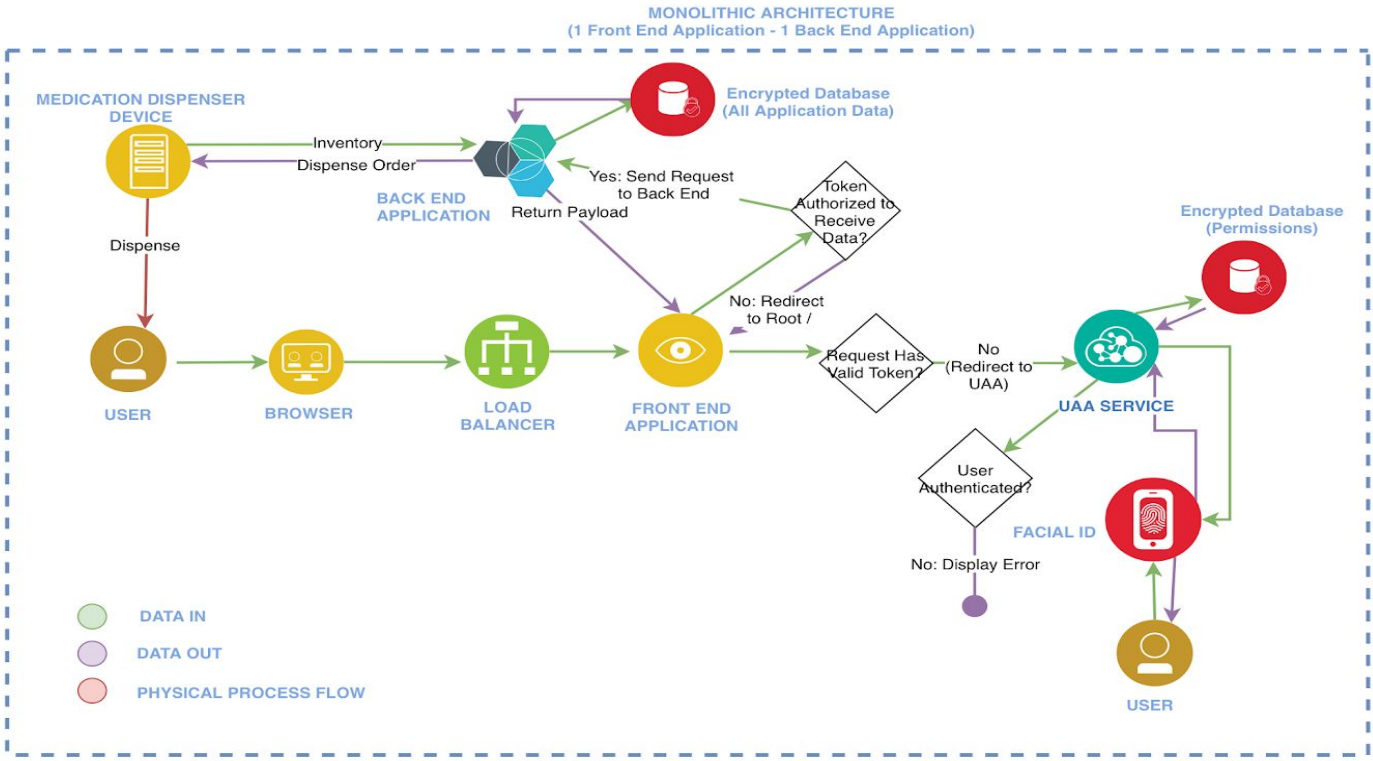
Samantha Tone

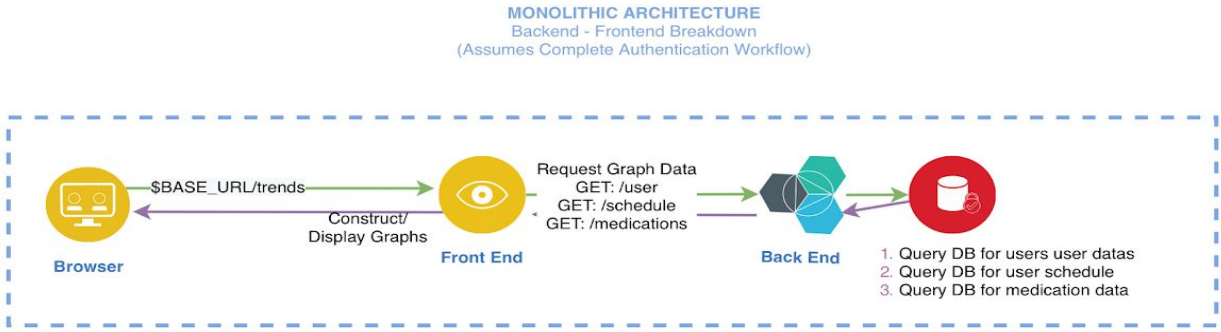
Pavan Thakkar

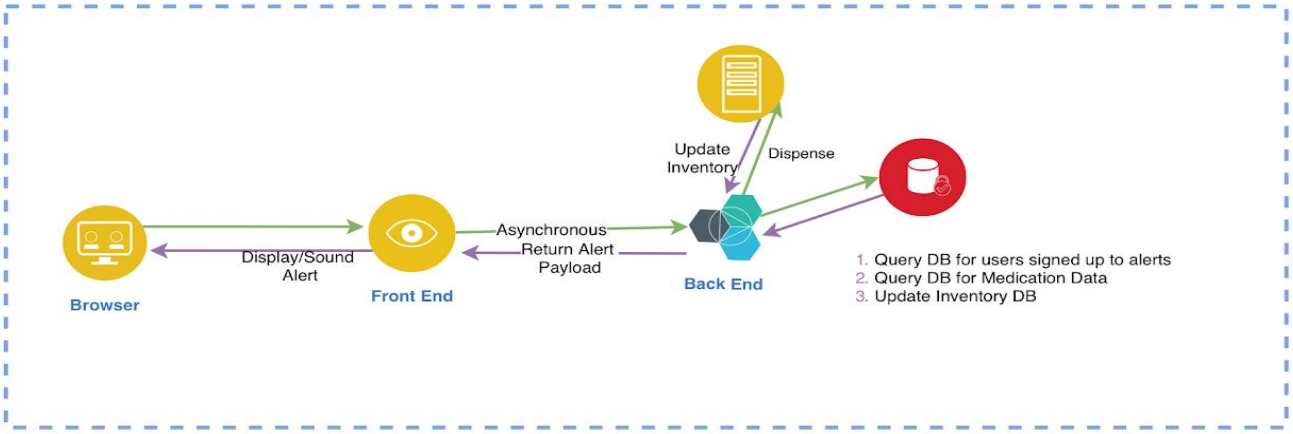
## Customer

Emmanuel Rovirosa

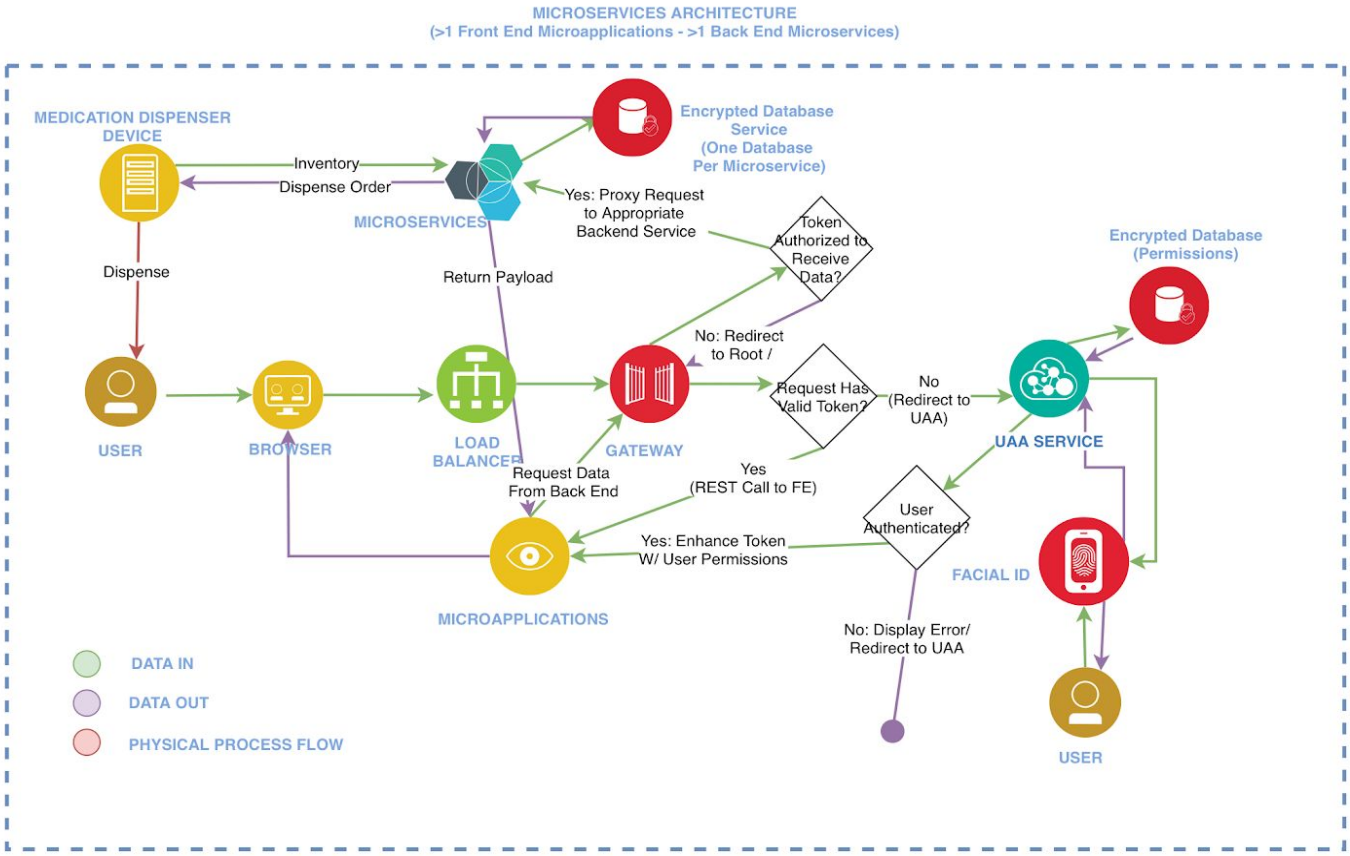
# Architectures


MONOLITHIC ARCHITECTURE
(1 Front End Application - 1 Back End Application)
MEDICATION DISPENSER DEVICE
Inventory
Dispense Order
Dispense
Encrypted Database (All Application Data)
BACK END APPLICATION
Yes: Send Request to Back End
Return Payload
Token Authorized to Receive Data?
No: Redirect to Root /
Encrypted Database (Permissions)
USER
BROWSER
LOAD BALANCER
FRONT END APPLICATION
Request Has Valid Token?
No (Redirect to UAA)
UAA SERVICE
User Authenticated?
FACIAL ID
No: Display Error
USER
DATA IN
DATA OUT
PHYSICAL PROCESS FLOW

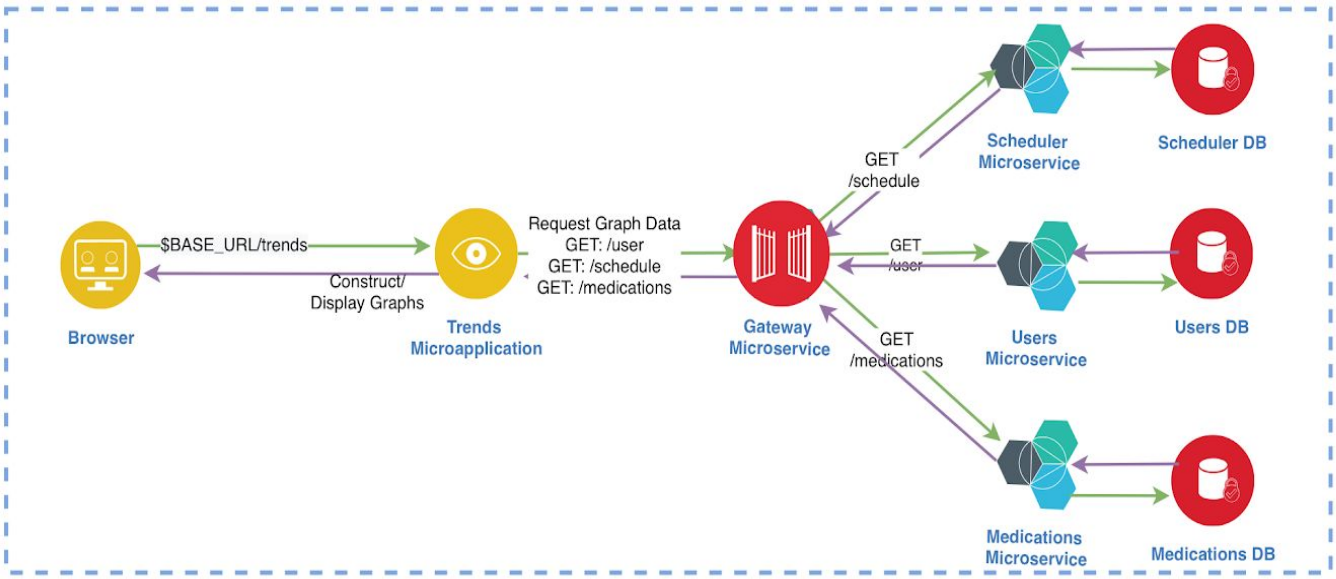

MONOLITHIC ARCHITECTURE
Backend - Frontend Breakdown
(Assumes Complete Authentication Workflow)
$BASE_URL/trends
Construct/ Display Graphs
Request Graph Data
GET: /user
GET: /schedule
GET: /medications
Browser
Front End
Back End
1. Query DB for users user datas
2. Query DB for user schedule
3. Query DB for medication data

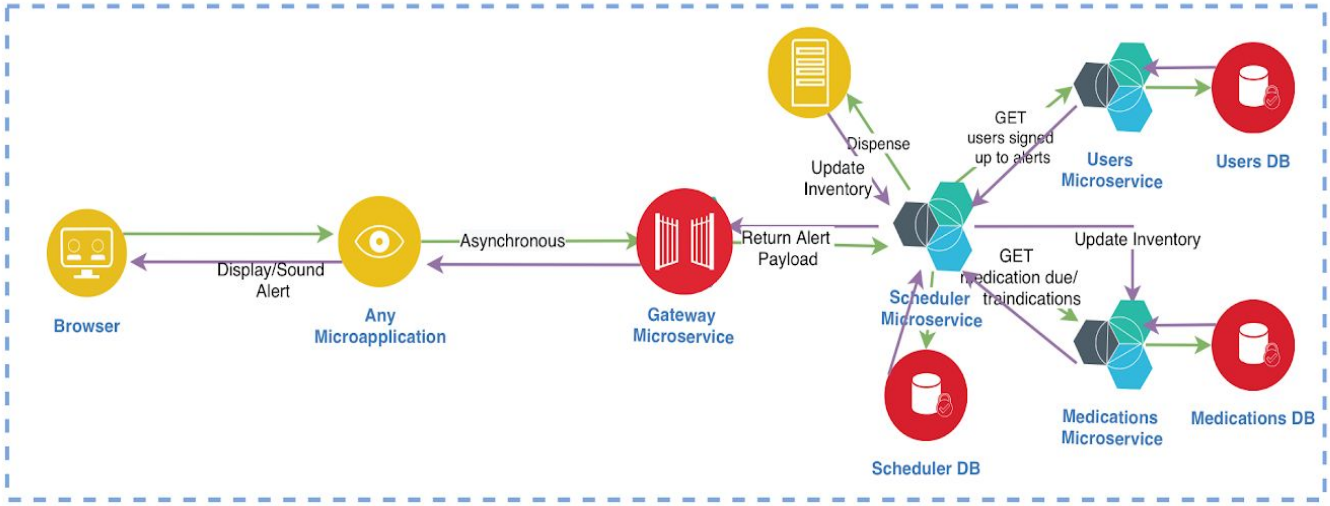

Update Inventory
Dispense
Asynchronous
Return Alert Payload
Display/Sound Alert
Browser
Front End
Back End
1. Query DB for users signed up to alerts
2. Query DB for Medication Data
3. Update Inventory DB

# Service


MICROSERVICES ARCHITECTURE
(>1 Front End Microapplications - >1 Back End Microservices)
MEDICATION DISPENSER DEVICE
Inventory
Dispense Order
Dispense
MICROSERVICES
Encrypted Database Service (One Database Per Microservice)
Yes: Proxy Request to Appropriate Backend Service
Return Payload
Token Authorized to Receive Data?
No: Redirect to Root /
Encrypted Database (Permissions)
USER
BROWSER
LOAD BALANCER
GATEWAY
Request Has Valid Token?
No (Redirect to UAA)
UAA SERVICE
Request Data From Back End
Yes (REST Call to FE)
User Authenticated?
Yes: Enhance Token W/ User Permissions
FACIAL ID
MICROAPPLICATIONS
No: Display Error/ Redirect to UAA
USER
DATA IN
DATA OUT
PHYSICAL PROCESS FLOW

MICROSERVICES ARCHITECTURE
Microapplications - Microservices Breakdown
(Assumes Complete Authentication Workflow)



# Key Quality Attributes

**Reliability**

Monolithic Architecture

The monolithic architecture may struggle with reliability at a large scale. The single Back End service has to handle all requests to the backend API, and a single database has to respond to all the queries. This may cause lag and eventual downtime. Scaling up to multiple instances of the back end would be hard to implement since there is no gateway to route requests to the different running instances. If a system is implemented to scale this architecture to multiple instances, the single database would become a stress point, since all queries from the backend would have to go to the same database.

Microservices Architecture

The microservices architecture is designed with scalability in mind. The inclusion of a gateway service allows multiple instances of individual APIs to exist, so, for example, if the scheduler API receives the majority of requests and results in lag or outages, the scheduler microservice may be scaled up to multiple instances; the less-used services can remain the same. Since each microservice has its own database within a database service, database lookups are not slowed down as much by all queries going to the same database. Moreover, since we also have microapplications serving out the front end, one page being down does not result in complete downtime; i.e, if the trends/graphs page is down, users would still be able to view their schedule.

**Efficiency**

Monolithic Architecture

Building and deploying the monolithic application would be much more efficient in the monolithic application. In addition, querying the back end would be much simpler and potentially faster, since the back end would be able to construct data for responses by performing complex database joins across tables very efficiently. This is not possible in the multi-database approach

Microservices Architecture

Although the microservices architecture is more efficient at a large scale, its inability to perform complex joins across the multiple microservice databases may make it less efficient in some instances. The microservice APIs need to do more work to compensate for the limited database operations. In addition, deployment may be more complex to set up the networking between the gateway and the microservices.

**Integrity**

Monolithic Architecture

The lack of a gateway between the front end and back end may compromise integrity in the monolithic approach. Messages would be vulnerable to being intercepted without the single ingress point to the back end.

Microservices Architecture

Integrity is a strength of the microservices architecture. The user’s authentication details and data from the UAA service are encrypted and decrypted at the gateway layer. Front end applications are unable to communicate with the backend directly, creating a separate security

layer. In addition, having separate databases with separate credentials mean that if one database is compromised, bad actors do not have access to the complete application data.

**Usability**

Monolithic Architecture

The monolithic architecture suffers from failures not being localized. For example, if a table is corrupt in a database, the whole application may go down. On the other hand, since there is a single code base for the front end, users may enjoy more consistency in the application's usability.

Microservices Architecture

If one microservice or microapplication goes down or has a bug, users may still be able to make use of the rest of the application fully. This is a large benefit of the microservices architecture. Microapplications allow designing tailor-made views, which may be completely different from the rest of the application. This may be beneficial in an application such as this one, where professional users such as physicians may have specific UX requirements that would not be useful to general users.

**Maintainability**

Monolithic Architecture

If the application grows to a large size, the codebase may become hard to maintain, since the monolithic architecture would have at most two code repositories. Adding features would be hard since any new feature has to work within the constraints of the whole application. For example, new languages cannot be introduced in any capacity. In addition, common maintenance tasks such as database migrations are difficult when all the data is kept in the same database.

Microservices Architecture

The microservices architecture is easier to maintain, since each microapp and microservice has its own independent codebase and set of dependencies. Finding the piece of code that is responsible for an action is easy if the action can be mapped to a particular microservice with its own repository. Another advantage of microservices is that adding new functionality that does not fit into the present data model of any existing microservice may be accomplished by a new microservice. Introducing new languages and technologies (i.e., cache or queue services) is easier and can be done service by service without having to touch the whole application.

**Flexibility**

Monolithic Architecture

A large monolithic application can be very difficult, if not impossible, to containerize, meaning that infrastructure and resources (i.e., storage, processing, and memory) have to be planned much more carefully, since they have to support the complete application. Data migrations in this architecture may be very difficult, since we're dealing with a single database. A migration to a different infrastructure provider (cloud or physical) would require a clone of the complete application before switching the load balancer, resulting in additional cost, particularly if dealing with physical servers.

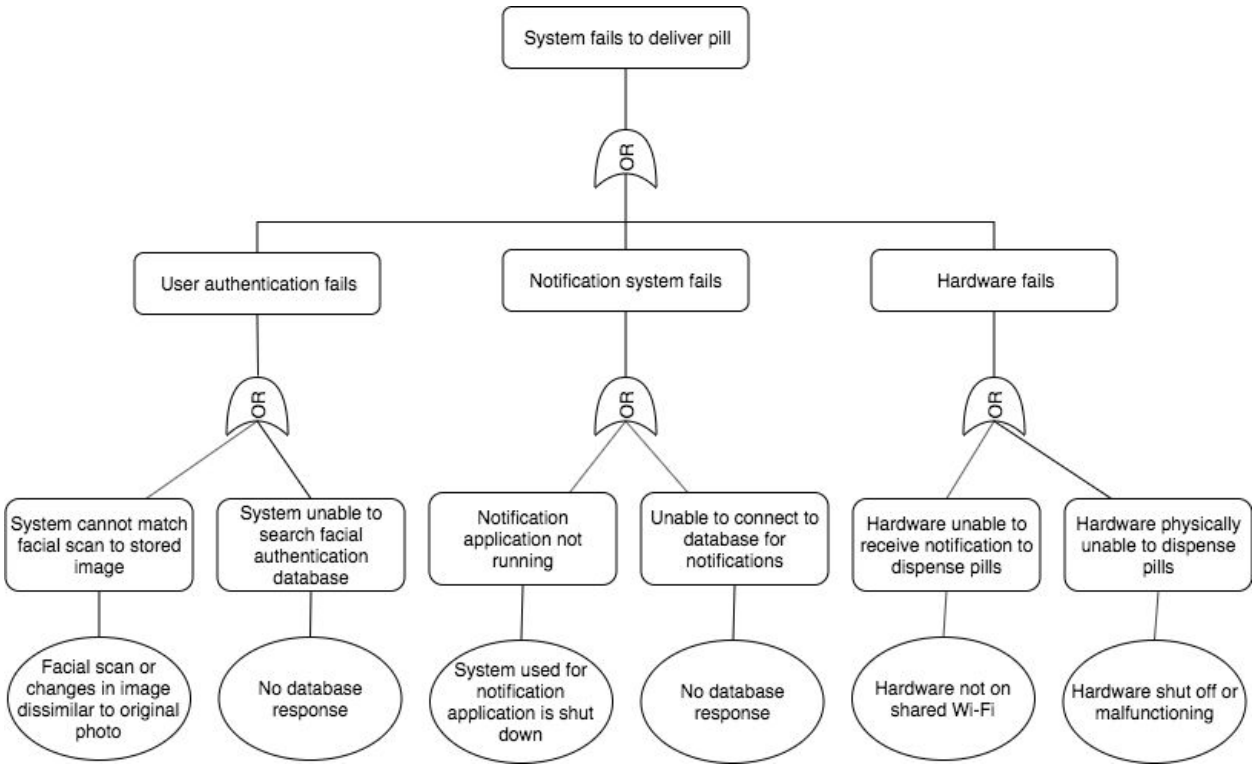
Microservices Architecture

The microservices architecture is more flexible because it is very easy to containerize. This means that the application can be ported to any infrastructure system that supports running

containers (i.e., Docker). Resources such as memory and storage can be allocated in a more flexible manner, since they can be allocated dynamically at the container level without requiring downtime. Serverless solutions are a possibility.

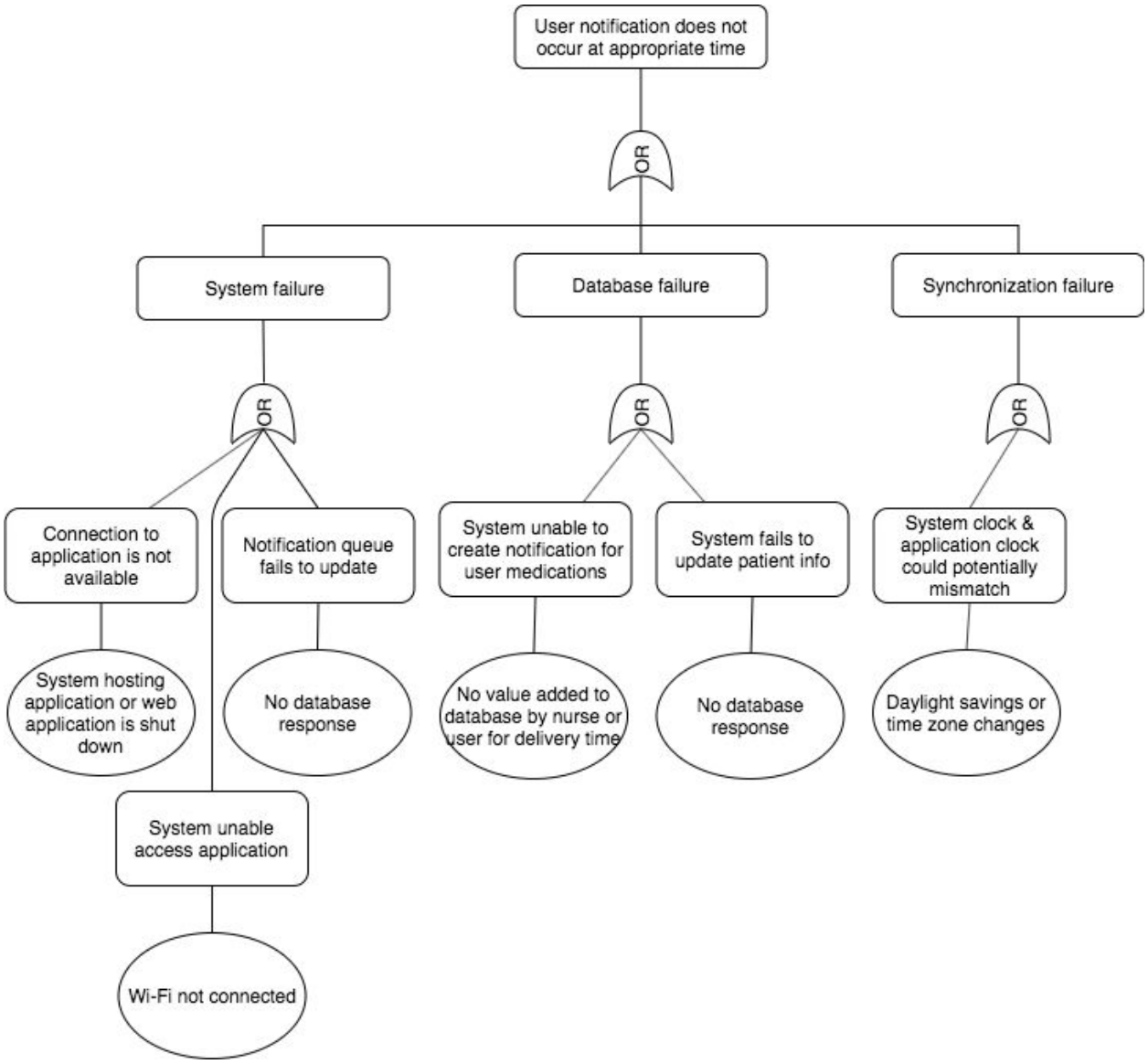
# Failure Modes

## System Fails to Deliver Pill




Several instances are able to cause the system's inability to deliver the pill to the user. We go a bit deeper into notification system failures in our second failure model. Most of the issues in this failure model have to do with connections between our different applications & systems. Since we will be using a web application, it is important that the system in use have that system operating in the background. It is also incredibly important that our hardware system be connected & functioning appropriately to all for delivery even if the notifications & authentication software components are operating correctly. Again, the connectivity of the system is where we'll see the most potential for faults.

## User Notification Does Not Occur




Again, connectivity plays a major role in the failure of the notification system. Ensuring that the application is actively running on the host system as well as ensuring that the system is able to access the database & updates are extremely important for the system to work in actually notifying the user that it is time to deliver the pill. There is also a potential fault where, depending on how both systems update, there good be a time mismatch. There's even some room for error, especially in our hospital scenario, which requires nurse input for delivery times, that we may not ever have a notification to deliver the pill if the nurse is not prompted to enter these delivery times in the first place.

# Use Case Walkthroughs

## Use Case 1: Nurse dispensing pills to individual in hospital room

- Nurse adds patient data such as medication, dispersal times, and information on the person who needs the pills. This would be stored in the respective databases outlined in the microservices architecture.
- Number of pills in the dispenser would also be stored in the medications database.
- The nurse can also add their own face id to the user authentication database so that only authorized personnel can retrieve the pills.
- All of this information is available for the dispenser retrieve so that it knows when to disperse the pills and be able to authenticate the person trying to retrieve the pills.

## Use Case 2: Individual reminding themselves to take pills

- Through the web application, the user can add their information, which will be added to the users database, the face identification will be added to the user authentication database
- User will add pills (medication database) and the dispersal schedule (schedule database) which will each be stored in their own databases as laid out in the microservices architecture.
- Number of pills in dispenser stored in the medications database.
- The dispenser retrieves this information in order to alert the user that it is time to take their pills. The microservices each handle the databases needed to store this information.
- Dispenser will authenticate the user using the user authentication service

## Use Case 3: A family member making sure that another family member has taken their needed pills

- Similar to use case 3, the user adds all of the necessary information for the dispenser to function. The microservices architecture accounts for all the necessary databases.
- The family member gets added as an authorized user on the account that is for the person getting the pills.
- Facial identification could be added to the user authentication database for the family member so that they could also retrieve the pills for the other person.
- Family member can also access account to look at pill schedules by requesting data from the necessary databases so they can look at pill dispersal history.

# Validating the Microservices Architecture

The architecture supports situations where a nurse is dispensing pills to an individual in a hospital room, an individual is reminding themselves to take their pills, and a family member is making sure that another family member has taken their needed pills. So the architecture supports each of the system's use cases.

For use cases #1, 2, 3: all the necessary components appear to be in the system's architecture. The components also seem to be connected correctly. All the arrows are pointing in the right direction. The system does appear to enact the right state changes and performing a use case does not appear to prevent subsequently performing any other use cases.

The architecture seems to support the requirement specifications that we came up with, such as that the system shall dispense medication only for a specific patient per alert, the system shall display which patient the medication alert is for either visually or with a name, the hardware unit shall authenticate the identity of the recipient as a patient or caregiver, the hardware unit shall dispense pills into a holder after authenticating identity, etc.

All of this being said, there still is room for improvement for the architecture, as alluded to in the "Implications" section.

# Implications

- In order for the dispenser to work, it needs to be connected to the internet to access the data needed for authentication and pill dispersal. Without connection, the dispenser can not access the authentication service to confirm the identity of the user and the schedule that tells the machine what/when to dispense. The lack of offline support could render the device useless. In the case where the device is being used for personal use, the dispenser could store some data locally such as the necessary information to authenticate the user and the pill schedule.

  This could be remedied in the hospital environment by having the data on the hospital's servers for the dispenser to access. For the personal use case this may entail having another microservice added to our architecture that manages the offline memory, updating it when the dispenser has access to the internet and ensuring that the data is consistent with the other databases in the system. A failure that could arise out of this addition would be inconsistencies between the offline information and the information in the online database that results in a notification not being pushed or a pill not being dispensed. This would improve usability and make the architecture more reliable and flexible, as the internet is known to go down sometimes.

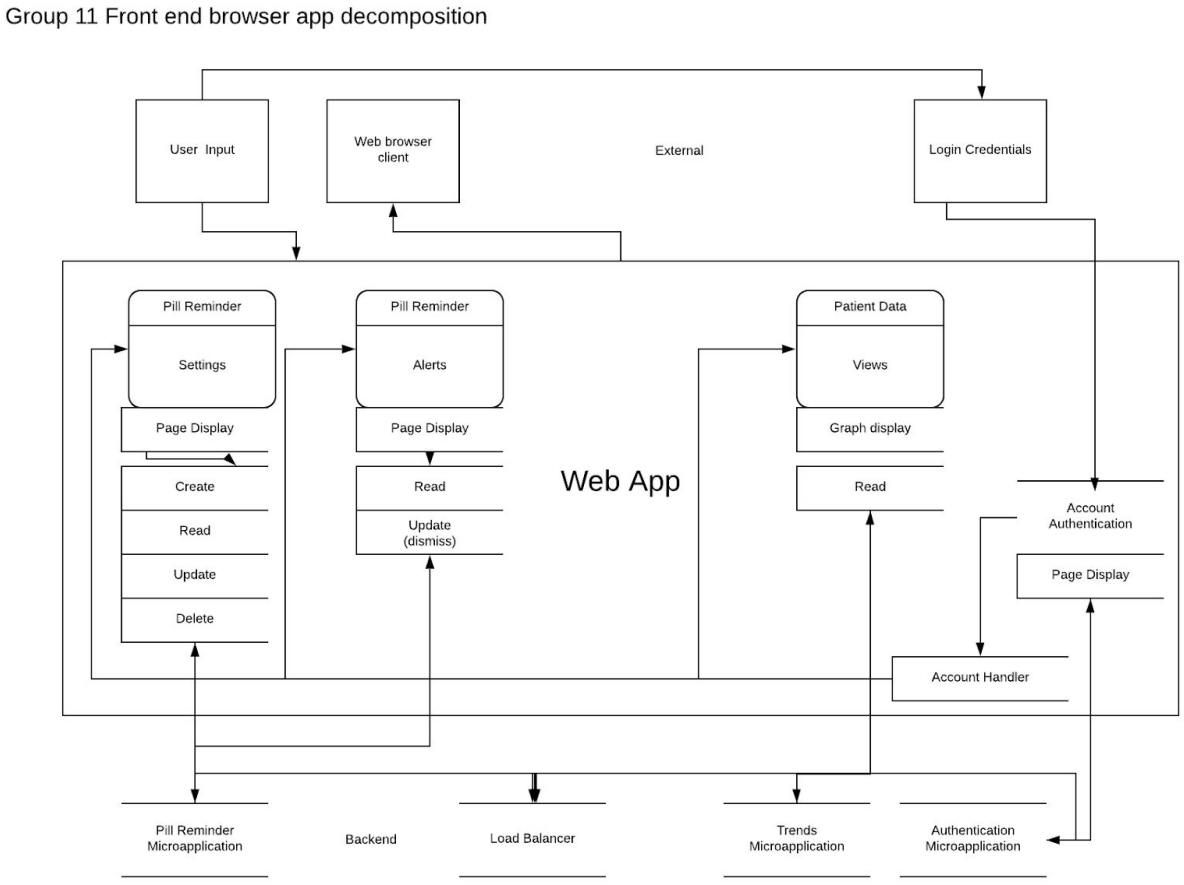
- As mentioned in the fault trees, failures can happen when databases fail to respond because they are down or might not understand the request that was made. In a microservices architecture, this is a possibility since you have several microservices with their own databases communicating with each other in order to fulfill requests. If the microservice is down completely, then a potential solution is something similar to what was mentioned in the previous bullet point. Using the newest data stored locally in the cache or on a server can be used instead. Another option would be using the most recent back-up of the database.  This would improve the reliability of the system.

  In order to prevent a miscommunication between the various microservices, tests should be implemented to make sure that each microservice understands the request it is being given. This can be implemented using tools such as API Fortress that ensure the microservice receives instructions that it can interpret. Having these tools in the system can greatly reduce the risk of faults happening. This would improve the reliability, since it would prevent bad fetches from happening and hopefully prevent faults from happening. But, of course there is always a possibility that the database can't be accessed and the request still fails.

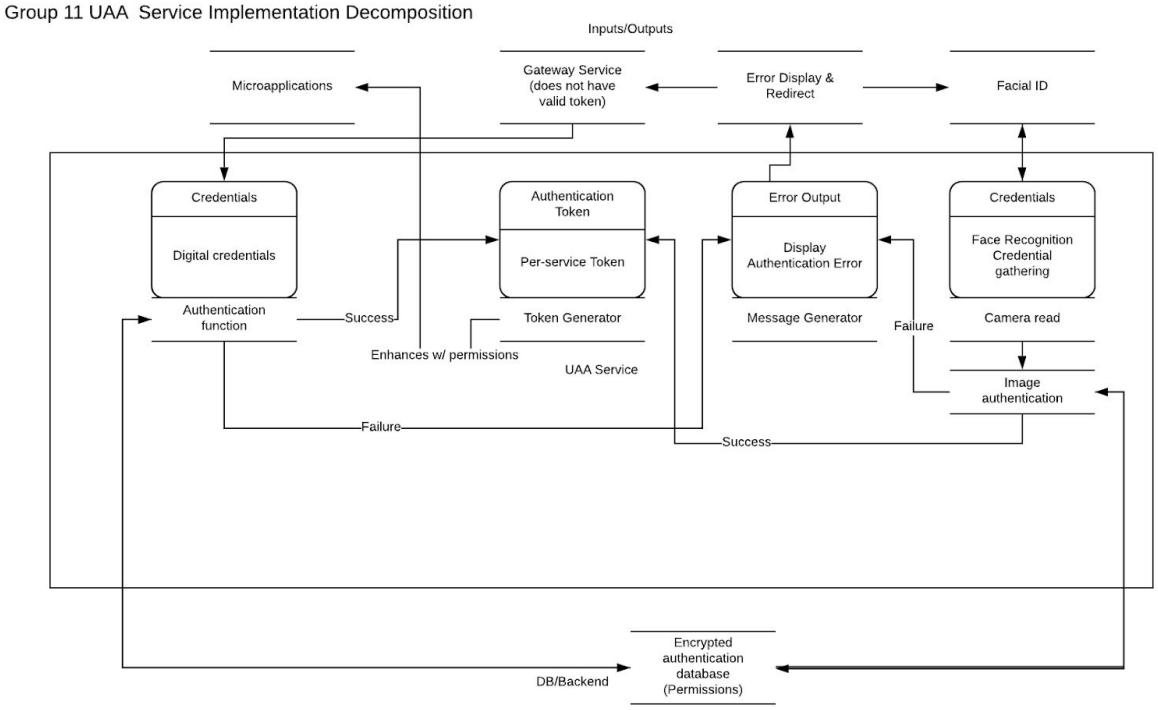
# Lower-Level Dataflow Diagram Decompositions

of the Microservice Architecture (2 Key Parts)

## Web Browser Application



# UAA Authentication Application Decomposition



# Contribution Summary

The following indicates team member contributions during the project stages leading up to & including HW3:

- Mario Bocaletti - Completed the two contrasting architecture diagrams and wrote the quality attributes section.
- Henry Clay - Completed the lower level decomposition dataflow diagrams
- Corey Nielsen - Completed the implications and use case walkthrough pages.
- Samantha Tone - Completed the failure modes section & corresponding fault tree diagrams
- Pavan Thakkar - Completed the validation of a selected architecture (Microservices)