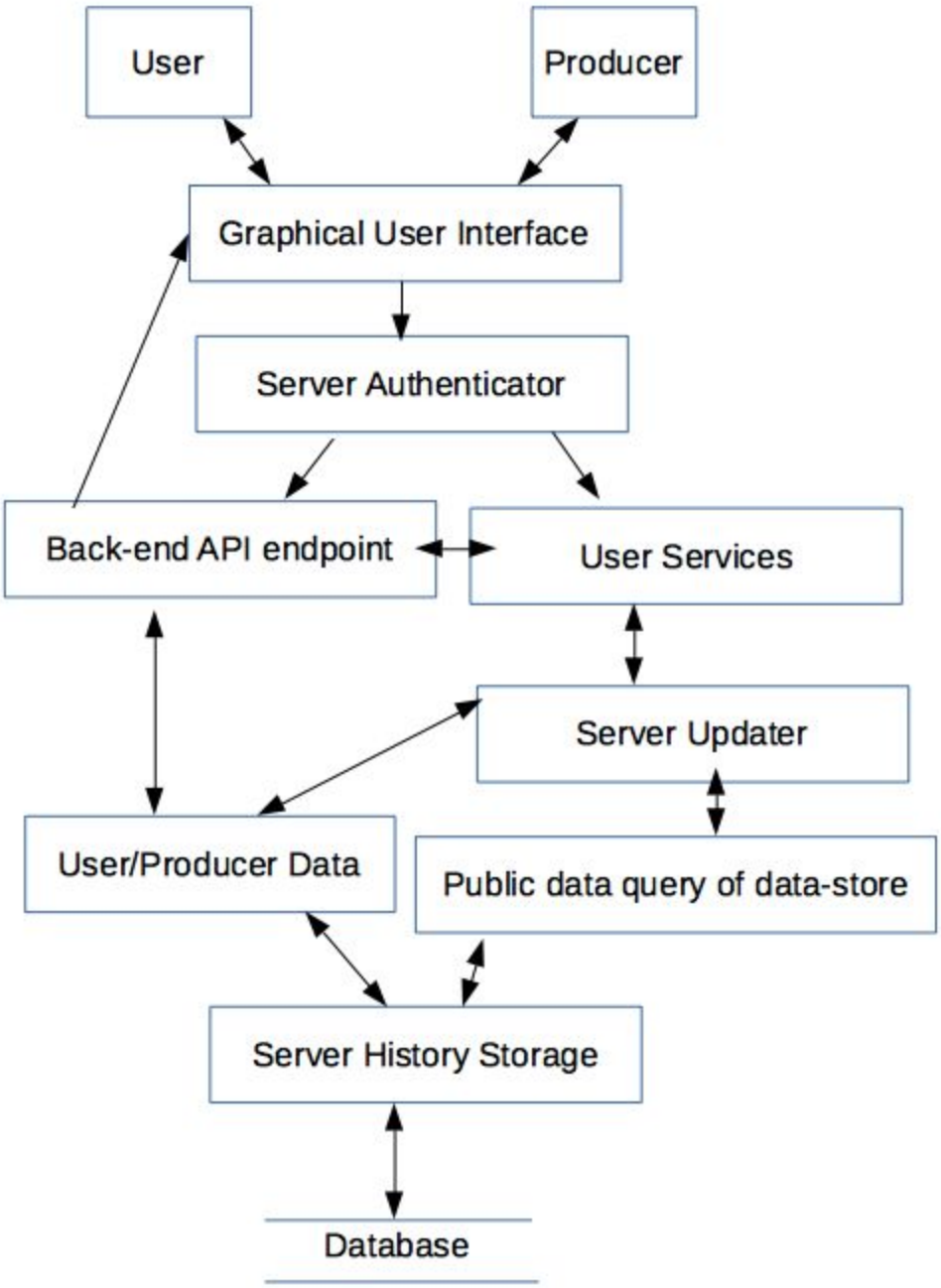
# Carbon Footprint Web Portal / Application
## - System Architecture - HW3

CS 361 Group 2 Project
11/3/19

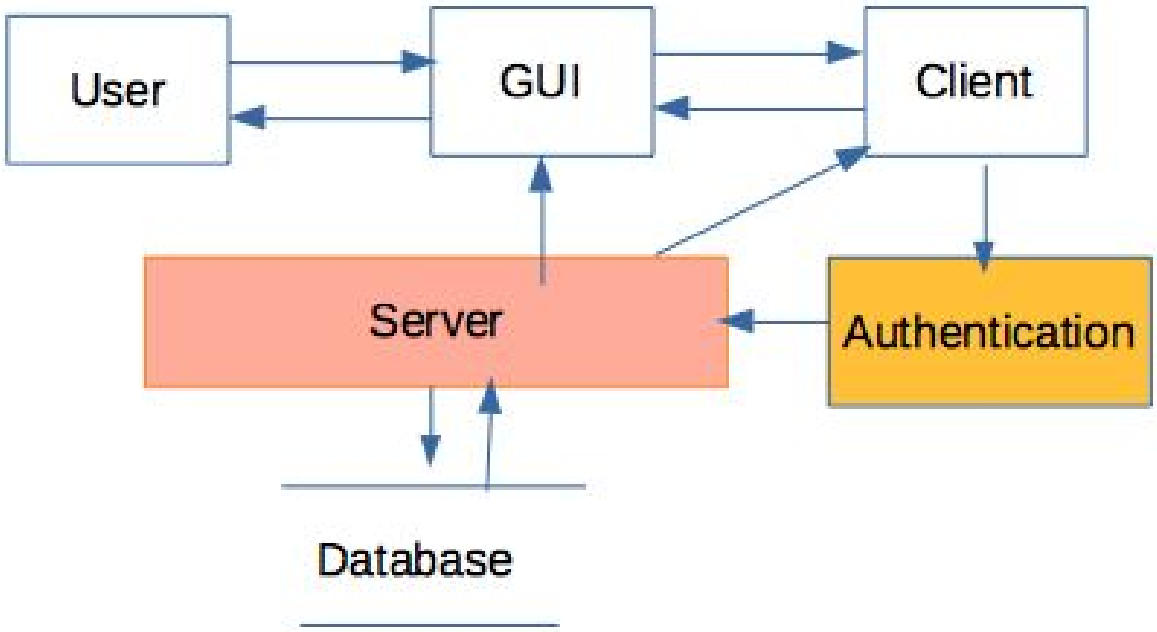

- Alex Densmore
- Anousha Farshid
- Adam Wright
- Ken Wyckoff

## System Architectures

Architecture 1



Architecture 2



## Key Quality Attributes

- ★ Reliability
  - Architecture 1
    The advantage of the divided architecture is that if a single service fails, then it does not cause all of the other services to fail. This would allow the user service to continue to run even if the producer service went down. In this design, the user's past product values are stored in the database separately so that a round trip is not necessary each day, and the producer service would be unnecessary for the running of the user service once a product has been entered. If an update is necessary, or a service goes down, then only that portion of the application needs to be stopped and restarted.
  - Architecture 2
    The disadvantage of the server-centric architecture is that, if the server goes down, the entire system goes down. The GUI cannot be rendered, users cannot be authenticated, and no data flow can occur. This means that if an update to a portion of the application is necessary, or if a failure occurs, the entire application will need to be stopped and restarted.

- ★ Integrity
  - Architecture 1
    With the split architecture the individual services look to a single source of truth for their data and there is no coupling or side effects between each of the services. This should reduce that chances that an important value is updated or exposed by a process that lead to unexpected results or undefined behavior.
  - Architecture 2
    With the server-centric architecture, any integrity issues in the server negatively impact the entire system. For example, a hacker who gained access to the server could then easily steal all user data or completely change what is rendered on the user end (such as adding in their own advertisements or other content or creating other forms of attacks). If the data parser within the server malfunctions, all data entering the database could quickly become corrupted.

- ★ Efficiency
  - Architecture 1
    The compartmentalized architecture has the advantage of spreading tasks out among internal systems. Each software component can be individually optimized as necessary and each hardware component can be upgraded individually. Data can flow between processes such that work can be done simultaneously across the system which minimizes data queue times.

- Architecture 2
  The server-centered architecture's efficiency relies on the speed of the server itself. If the server is running slowly due to large user volume or hardware issues, this makes it hard for even a simple page of the user interface to be rendered quickly. With a split architecture, server slowness may affect the rate at which user data is sent once the user submits a form or at which data is received when the user searches information, but the basic elements of the user interface on the client end would still be able to load quickly. The lack of independence of systems in the server-centered architecture could lead to a very frustrating experience for users in terms of efficiency.

★ Maintainability
- Architecture 1
  User would be suggested to add/edit new inputs based on their purchases/ transportation every day. This will ensure that all users would have updated profile. However, it will not impact other users' profiles or usability.
- Architecture 2
  Any change/update in server data would temporarily affect all users. They may not be able to access their profile during the maintenance time.

★ Interoperability
- Architecture 1
  The compartmentalized architecture is able to be modified in order to communicate with any necessary systems. It would have a separate set of functions for dealing with Internet of Things (IoT) devices. Although these devices would be communicating at different intervals, only a subset of the functions would need downtime to be updated. Producers may input data in a variety of ways, and having their input stream being separated for other functions would allow the compartmentalized architecture to evolve to adopt specific producer functionality or needs.
- Architecture 2
  The server-centric architecture would need to be taken offline completely in order to be updated to communicate with a new and needed system. If any such system introduces or experiences malfunctions, then the entire server-centric architecture is at risk of failing until it is taken offline for updates.

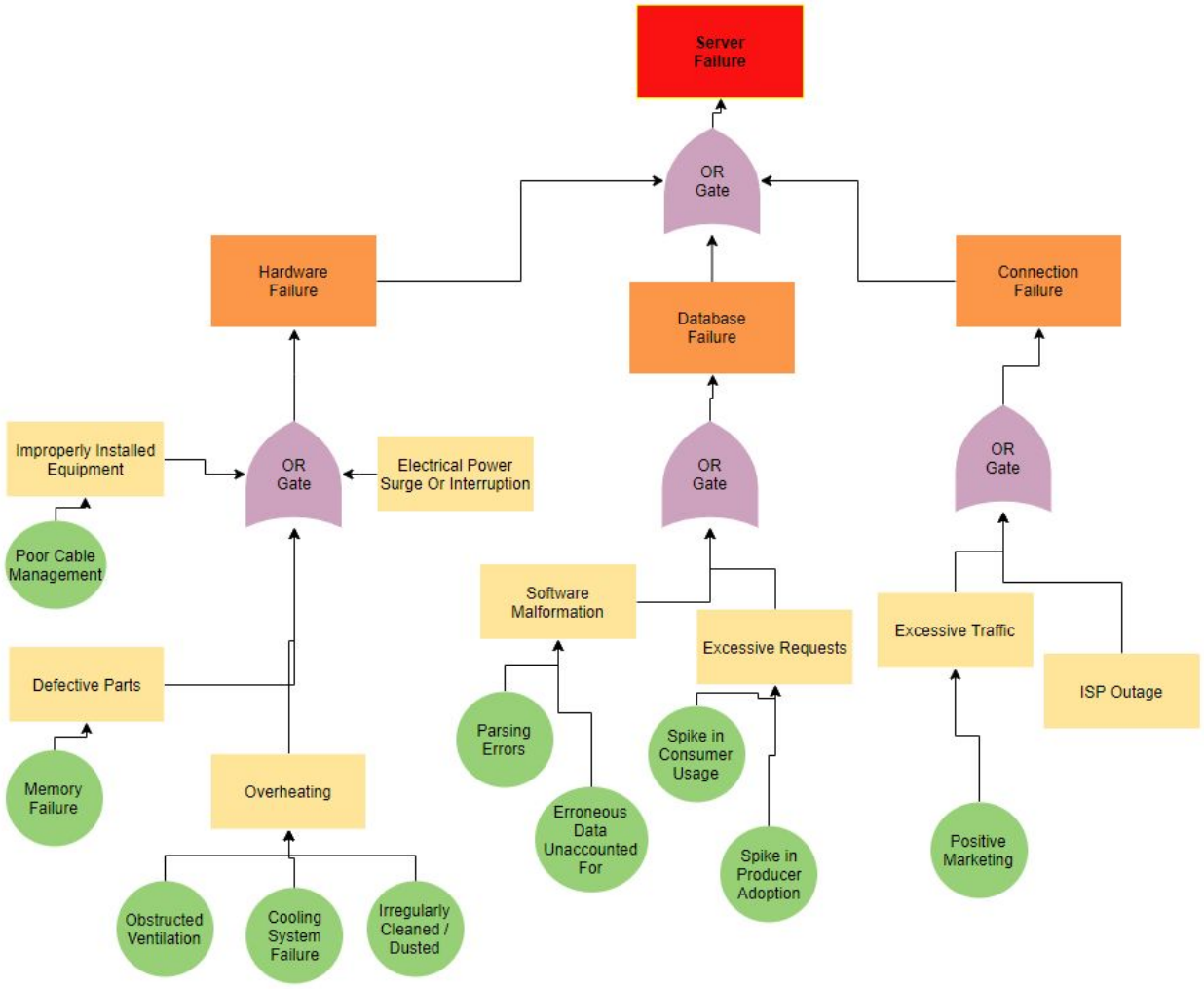
★ Flexibility
- Architecture 1
  The compartmentalized architecture can easily adapt to unusual conditions because each process can fail individually without taking the entire system down with it. This will allow the developers to create new services over time. The new version of the service can have a new API that will allow developers to test the

new service and to make the transition to the new version of the service in a seamless way.

- Architecture 2
  In the server-centric architecture, any unexpected conditions on the server end would rely on the server to resolve them for the other systems not to be dramatically affected. For example, if a process froze the server, no pages on the site would be loadable or usable until the unexpected server dodition was resolved. This may require rebooting the entire system or even reprogramming a component if there is a programming error when updating some features of the system.

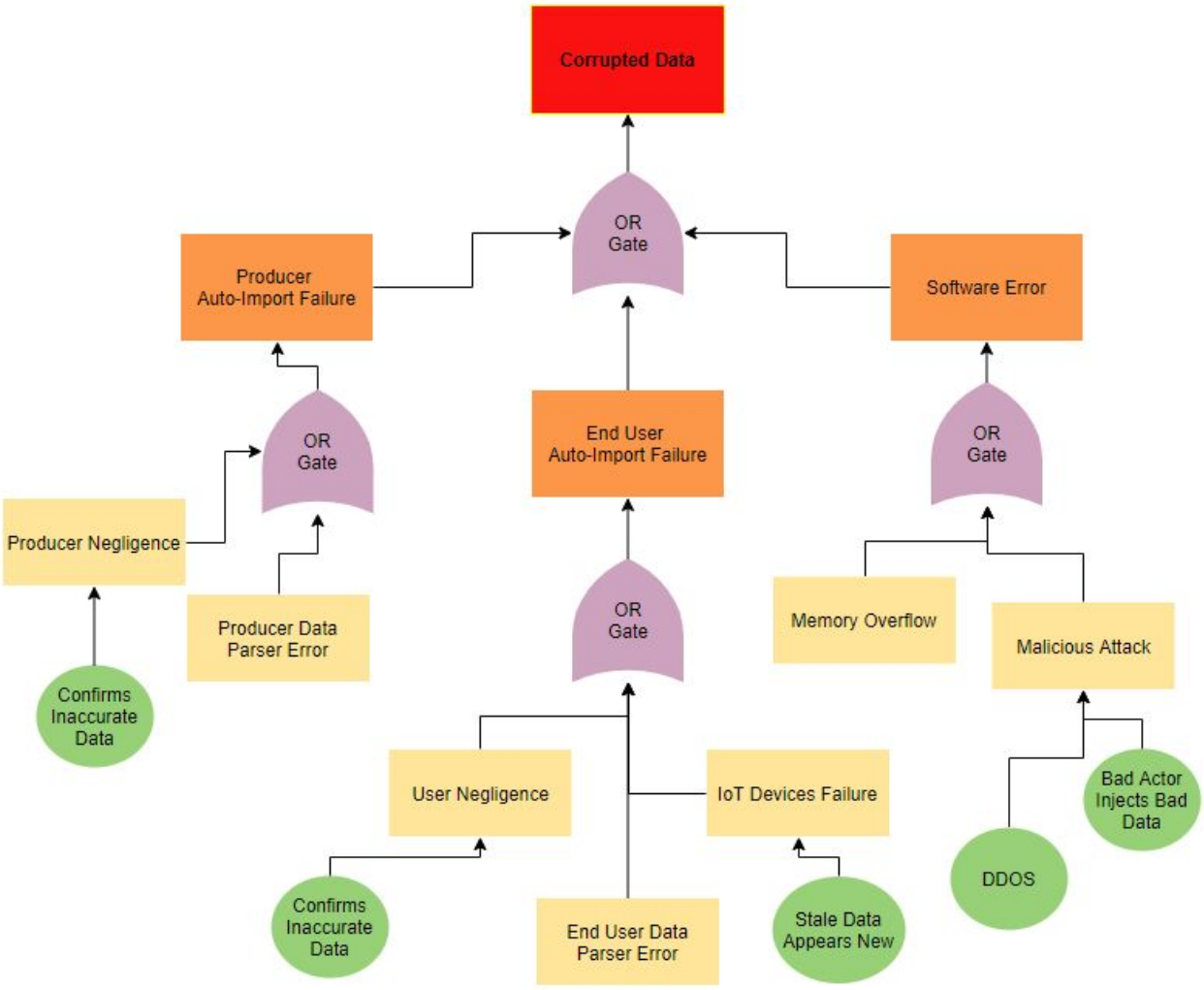
## Failure Modes

Server Failure




The architecture more prone to failure for the Server Failure mode is the monolithic, server-heavy architecture. A connection failure is more likely since all traffic must route through one system. Since all of the hardware is centralized, it is more prone to overheating and more susceptible to electrical surges and outages. Although software can never be perfectly formed, separating databases can maintain some data in the event of an outage.

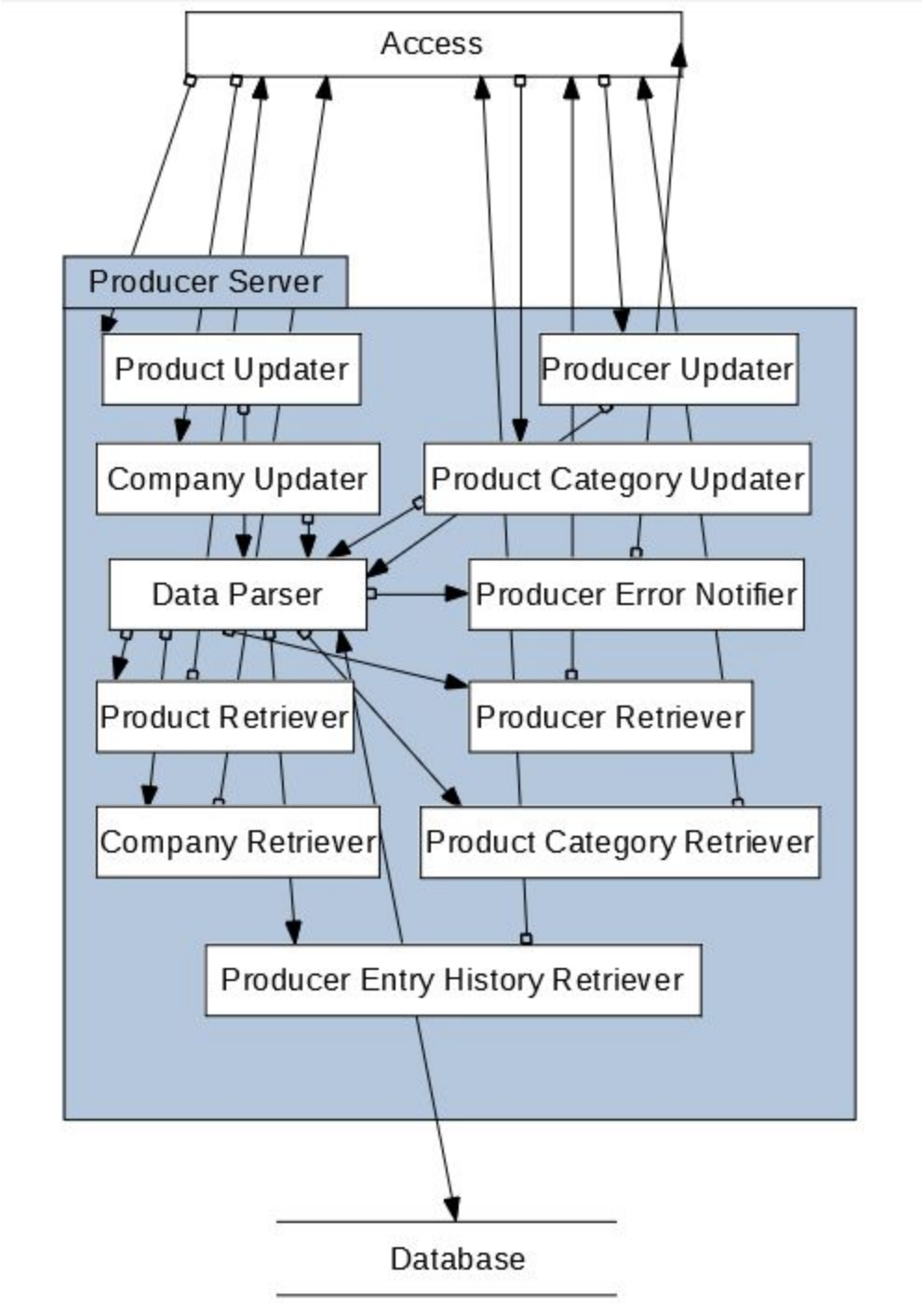
Corrupted Data




The architecture more prone to failure for the Data Corruption failure mode is the monolithic, server-heavy architecture. In this paradigm, all of the data is stored in one database. As such, if corrupted data is within the system, it is shared among all processes. If a malicious actor breaks into the system at any point, he or she has access to the entire system and can thus inject data to user profiles or producer product information alike. Similarly, if the system is targeted by a DDOS attack, then neither end users nor producers can add or update information. No system is immune to negligence, but negligent producers that confirm inaccurate product information immediately affect the end user profiles in this architecture.

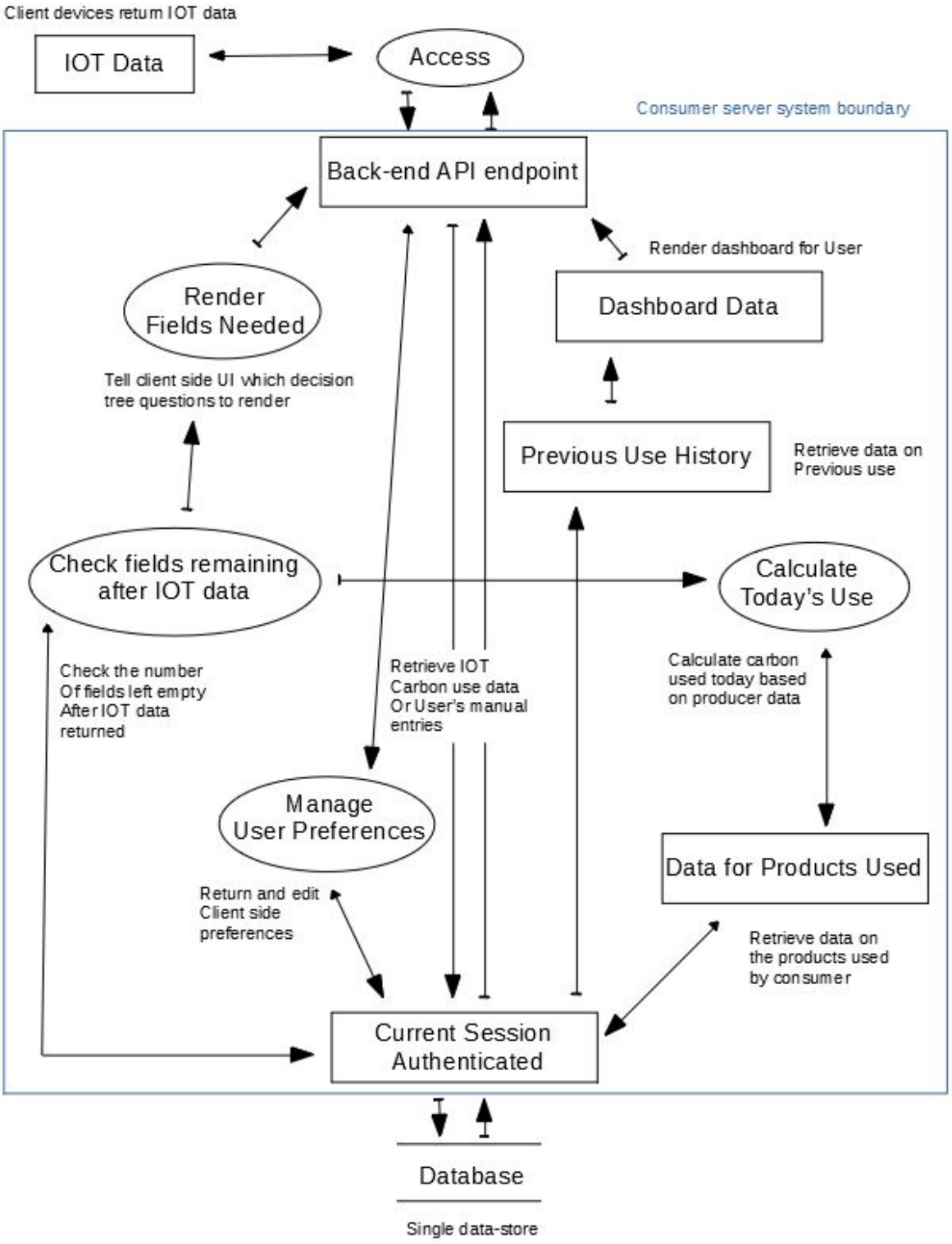
## Architecture Decomposition

**Important Element 1**
Producer Server Data Flow Decomposition



**Important Element 2**

End-User Server Data Flow Decomposition



## Use Case Walkthroughs

### Use Case 1: Measuring End-User Use of Carbon

- ★ The chosen system will allow the user to view the system through the GUI and set up an account and then to login to their account on subsequent visits through the server authenticator.
- ★ The chosen system will allow the user to be able to login to the application and then be presented with the IoT data gathered from their devices, through the back-end API endpoint, which they have associated with their account. User services will allow the user to confirm or update the gathered data through the server updater. The user information will be sent to the server history storage to be stored in the database.
- ★ The chosen system will retrieve the product data that is associated with the products that they have used from the server history storage and then automatically compute the current day's carbon usage using the server updater functions and store those values into the database, for the purpose of displaying to the user in the future.
- ★ The system will display a manual dialog tree for all fields for which there is no IoT data returned for the day. This will insure that all necessary information is entered for the day, while also reducing the burden on the user for entering the entirety of the information manually.
- ★ The system will display a dashboard for the user on the homepage which contains data and visualizations of the user's use of carbon over time, so that they may make behavioral changes which will reduce their carbon usage.
- ★ With the chosen architecture, the user will be able to experience a higher level of reliability and flexibility from the back-end of the application.

### Use Case 2: Measuring Producer Use of Carbon

- ★ The producer will be able to log in or create an account. Authentication data will be passed to the producer server to show whether or not the access attempt is valid.
- ★ The producer will be able to input data about a product, either through an auto-importing process or manual input. The client-side will perform the autoimport function and basic data validation, such as whether any pieces of required data or missing or data is an invalid format for a particular piece of information (such as a year containing letters).
- ★ The producer server will handle parsing of data and communication with the database.

- ★ The producer will be able to view confirmation of the data they have entered returned from the producer server. The server will once again handle parsing of data.
- ★ The user interface which the producer uses will be rendered on the client-side system. This means that client-side code will be responsible for incorporating any data retrieved from the server into a page's html.

**Use Case 3: Users Comparing Carbon Usage Between Different Products**

- ★ The user will request access through GUI.
- ★ The login information will be verified by the server authenticator and the user will have access to the software.
- ★ The system will allow the user to search for a product's information on the database with a request from the user services functions.
- ★ The user services functions will query the public data data-store for information regarding the searched-for product as well as related products, and the public data data-store will request that information from the server history storage.
- ★ Server history storage will request the data from the database, verifying the accuracy of any requested data it currently holds.
- ★ The requested data is sent from server history storage to the public data data-store, which in turn returns the data to user services.
- ★ User services sends the data of the requested product and all related products to the back-end API endpoint.
- ★ The GUI received the data from the back-end API endpoint and displays the information to the user.
- ★ The user is able to compare the searched-for product with products with similar functionality from any producer as well products with any functionality from the same producer.

## Implications

The most significant benefit of the compartmentalized architecture is that it allows for our system in its entirety to evolve as needed. We can expand our Internet of Things interoperability as technology is developed, released, and adopted in real time. Similarly, we can scale the database with redundancies as our collection grows and we can scale our software to allow for an ever-increasing user base. As our user base expands physically across the globe, so too can our hardware. We can install new servers and databases in various locations to increase user connection speeds as well as data storage and data integrity. Although physical expansion could make the system more vulnerable to malicious actors, with more physical locations to access, the compartmentalization would aid in minimizing the damage that can be done.

One issue with our chosen architecture is that the software systems can become overly decentralized to the point of it being difficult to maintain. Software errors, bugs, and glitches will continue to become more time-consuming to track down and correct as our code base expands over time. This can be accounted for with a dedicated managerial team overseeing production with an emphasis on documentation.

Another potential issue could occur when some components are not functioning and the data and functionalities that remain available to users (whether producers or consumers) behave unreliably. For example, if the producer server goes down completely, users may think they are accessing up-to-date data about a product, but they may be accessing stale data. A step to resolve this would be to have protocols in place for handling known error conditions and coordinating the activities of other systems accordingly. Users could be notified as needed about functionalities that are down so they are not surprised or confused by some functionality to which they are accustomed to being missing while the rest of the application seems to be functioning properly.

## Team Member Contributions

The following list indicates team member contributions during the preparation of HW3:

- All - Direct team meetings and continued communication via Google Hangouts and email; writing English sections of architecture documentation.
- Alex Densmore - Drew data flow decomposition diagram for producer server.
- Anousha Farshid - Drew high-level architecture data flow diagrams for both of the candidate architectures.
- Adam Wright - Drew data flow decomposition diagram for end-user server.
- Ken Wyckoff - Created both Fault Trees and Architecture Failure Vulnerabilities

Each team member was also involved in creating this document from the various parts completed individually, along with reviewing said documents for accuracy and assurance.