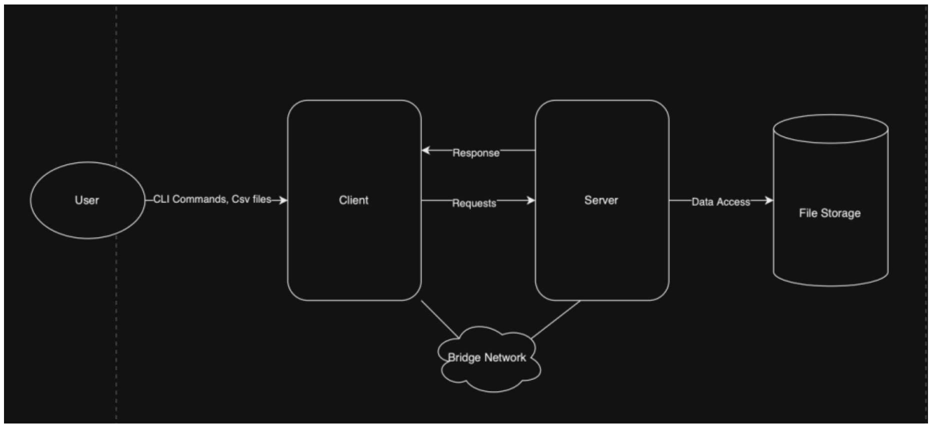
Design Document

Illustration of the Design:

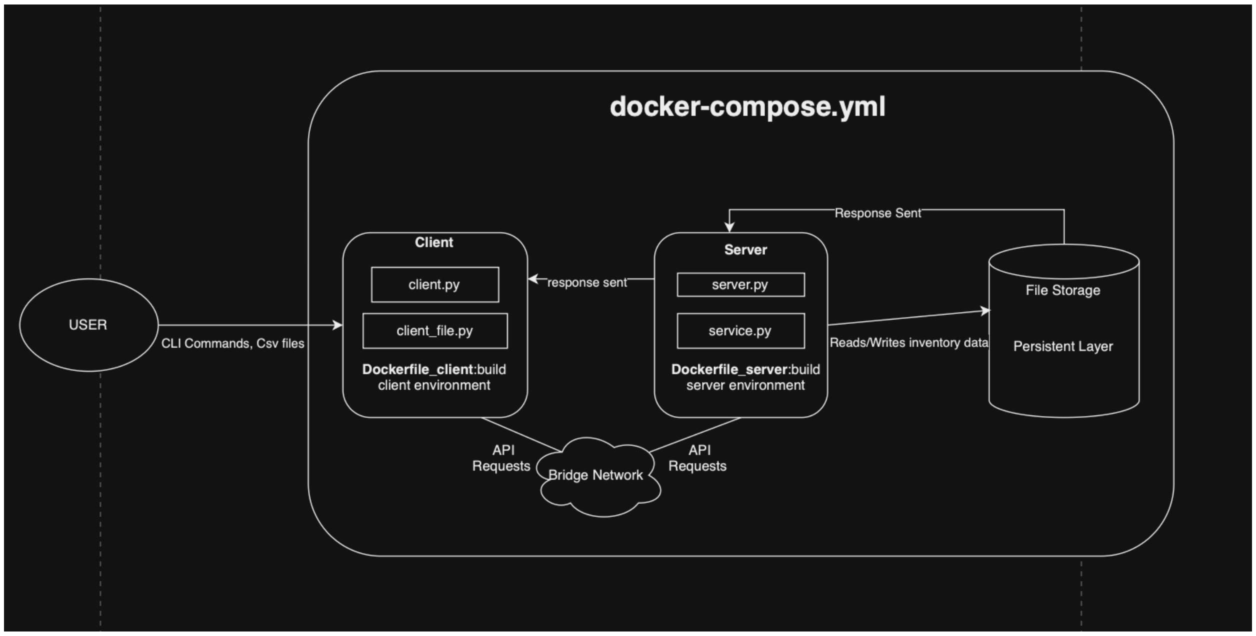
High Level Design:

The design consists of a **High-Level Design (HLD)** where the **User** interacts with the **Client Container**, which includes client.py and client_file.py. This container communicates with the **Server Container**, which houses server.py and service.py. The server interacts with **File Storage**, either through volumes or external storage, while a **network bridge** facilitates communication between the Client and Server Containers, ensuring isolation.



Low Level Design:

In the **Low-Level Design (LLD)**, specific API calls are made from client_file.py to server.py for various operations such as define_stuff, add, get_count, and undefine. This diagram illustrates the flow of data and requests between these components, showcasing the modular architecture that supports efficient communication and processing.



Benefits of the Design:
This design offers several benefits, primarily through the **isolation of components**. By separating the client and server into distinct containers, resources can be allocated and managed more efficiently, allowing the server to scale independently of the client. This modular approach simplifies maintenance and upgrades, as changes to one component can be made without impacting the entire system. Additionally, enhanced security is achieved by reducing the attack surface, as isolated components limit exposure to vulnerabilities.

Value of the Benefits:
The benefits of this design translate into tangible value. Improved performance is achieved through optimized resource usage, leading to enhanced responsiveness, especially under heavy loads. The modular structure simplifies troubleshooting, allowing for quicker identification of issues within specific components, thus reducing downtime. Furthermore, future-proofing the system becomes easier, as independent upgrades or replacements of components can be made to adapt to changing requirements or emerging technologies.

Drawbacks of the Design:
However, the design does have its drawbacks. The increased complexity of managing multiple containers can complicate deployment and maintenance processes, potentially leading to operational overhead. Moreover, separating the client and server may introduce network latency, affecting the performance compared to a single-container architecture. Ensuring compatibility between client and server versions can also pose challenges as updates are implemented, necessitating careful dependency management.

Impact of the Drawbacks:
On the other side, the operational overhead from managing multiple containers can strain administrative resources, requiring greater effort in orchestrating deployments and monitoring performance. Network latency may result in bottlenecks, negatively impacting the user experience when frequent data processing is needed. Additionally, integration challenges may arise, as keeping components in sync and ensuring seamless communication can demand more rigorous version control and testing protocols.

Value Analysis:
Conducting a cost-benefit analysis indicates that the advantages of modularity, security, and scalability outweigh the drawbacks associated with complexity and network latency, particularly in production environments where performance and reliability are critical. The long-term value of investing in a complex yet scalable and secure architecture manifests in maintenance ease, upgrade flexibility, and overall system resilience. Ultimately, a well-architected system enhances adaptability to evolving business needs, establishing itself as an asset in a fast-paced technological landscape.

R1-HLD:
In the high-level design, client_file.py is represented as a client component that initiates requests to the API Gateway. It fits into the inventory service as the entry point for user interactions, sending HTTP requests for inventory operations like adding, removing items etc. It communicates with the API Gateway, which routes requests to the Inventory Service.

R1-LLD:
client_file.py was added as a new script without requiring changes to the existing client.py, server.py, or service.py. It reads the CSV file and sends HTTP requests using client.py functions. No modifications were required to the core API or logic of the inventory service.

R1-User Story:
As a user, user wants to manage inventory efficiently. The client_file.py script allows the user to send requests to add, remove items etc. from the inventory, streamlining the management process. When the script is run, it initiates an HTTP request to the API Gateway, which processes the request and updates the inventory accordingly.
client_file.py allows users to batch-process actions such as defining items, adding inventory, and querying item counts by reading instructions from a CSV file and automating the requests to the inventory service.

R2-HLD:
Before: The architecture consists of a single container running both the client and server components. There is a direct interaction between client.py and server.py, and both share the same resources.
After: The architecture is modified to include separate containers for the client (client.py, client_file.py) and the server (server.py, service.py). A network bridge is established to facilitate communication between the two containers, isolating their resources.

R2-LLD:
Updated the docker-compose.yml to define separate services for the client and server. Each service is isolated in its own container, with a bridge network for communication. Using file storage, configured volumes in the Docker setup to manage data effectively.

R2-User Story:
As a user, when submitting an input file through the client application, client_file.py processes the data and transforms it into API calls (like define_stuff and add). These calls are sent to the **Server Container**, which executes the requests using server.py and service.py. The server then interacts with file storage as needed, ensuring that my inventory data is updated and accurate.