REQUIREMENT DOCUMENT FOR

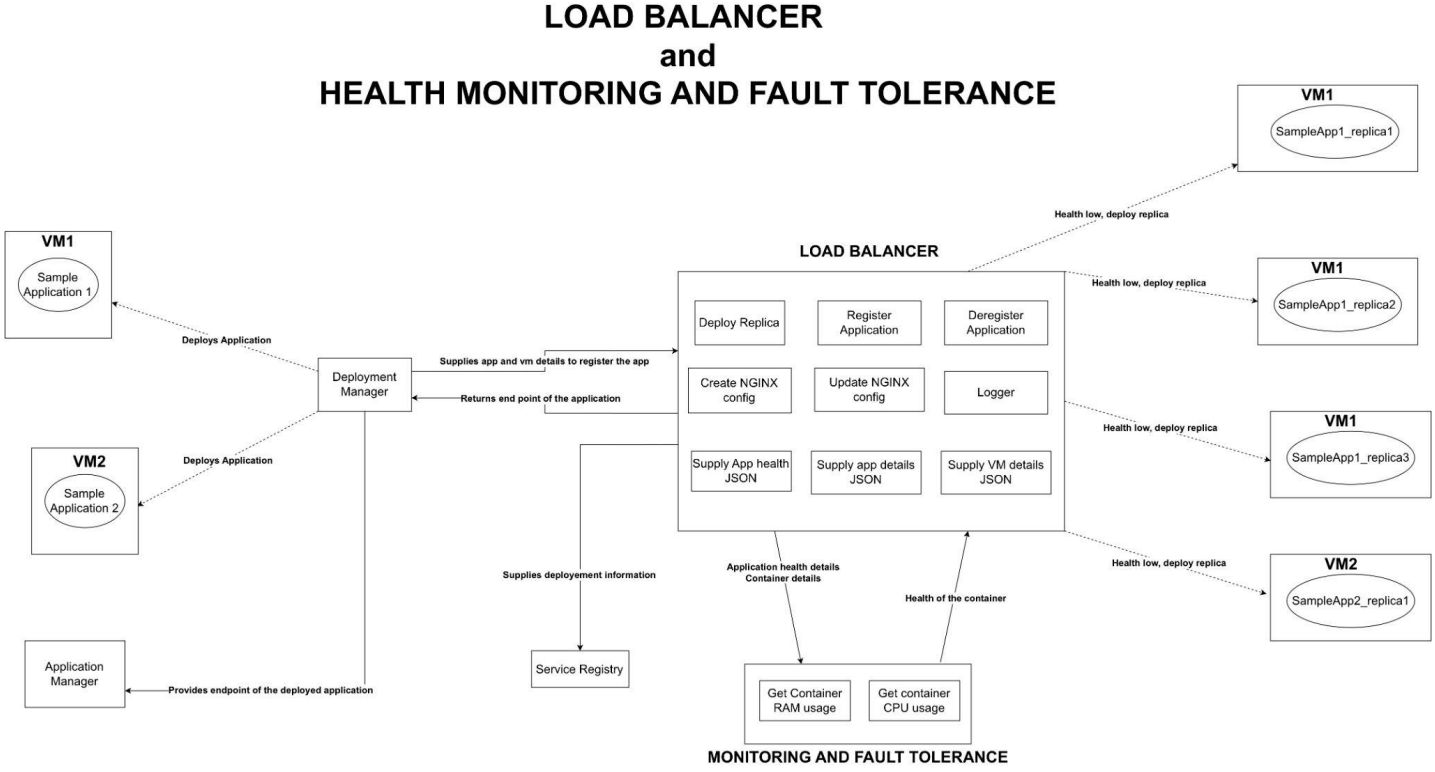
TEAM - 1

Harsimran Singh - 2022201049

Jatin Sharma - 2022201023

Adarsh - 2022201081

1. Load Balancer and Health and Fault Tolerance



- First the application manager gets a request to deploy the application. Then the application manager sends a message to the deployment manager to deploy the application.
- The deployment manager then deploys the application on VM having best health (health information is provided by node manager).
- Till now only the original instance of the application is deployed. Its end point is not yet known by anyone except the deployment manager.
- Now the deployment manager sends the ip:port of the application, name of the application, name of the docker_image of the application, etc to the load balancer.
- The load balancer upon receiving this information creates a config file for this application which contains the original endpoint of this application

and the ip:port of the proxy that it creates for it. This is the final end point of the deployed application.
- This end point is sent to the deployment manager which is further sent to the application manager. Now this endpoint is sent to the end-user.
- The load balancer also updates its config files -
- AppDetails.json (which contains all the details about the deployed application, like its end point, number of instances, container ids, etc, basically the registry of every app)
- VmDetails.json(which contains details of all the VMs in the VM pool like their public ip, the apps they host, the port number available for a new app to be deployed over it)
- AppHealth.json (which contains names of all the apps, and a list corresponding to each app which has the CPU and RAM usage of all the containers of that app)
- Nginx.json (which contains the latest port on which we can create a new NGINX proxy server).

MODULES INSIDE

THE SERVER

- The load balancer server is a flask server which has the following end points -
    - /registerApp: This end point is hit when the deployment manager has deployed the original instance of the application and wants register the application with the load balancer
    - /deregisterApp: This is to stop the app and remove all its registries.
    - /getVmDetails: This endpoint is hit to obtain list of all VMs and see their details, like their ip, port, list of deployed apps on them, etc.
    - /getAppsDetalis - This returns a JSON which contains all the meta data of the app, like its image name, the VM on which it is deployed, etc.
    - /getAppsHealth: This returns a JSON which contains the CPU and RAM usage of all the containers of all the apps (grouped by app name)
    - /logs: this is to obtain the logs of load balancer.

THE LOADBALANCER CLASS

    - LoadBalancer.py contains the LoadBalancer class which contains all the core functionalities like balancing, creating replicas, creating and updating configurations, etc

THE BASH FILES

- Files like load_balancer_start.sh and load_balancer_end.sh are there to initialize the and stop load balancer when the platform starts and stops.

LOGGER AND CONFIG FILES

- Logger.py is there to generate logs and there are other config files like AppDetails.json, etc as mentioned above in the working of load balancer which maintain the app and VM metadata.

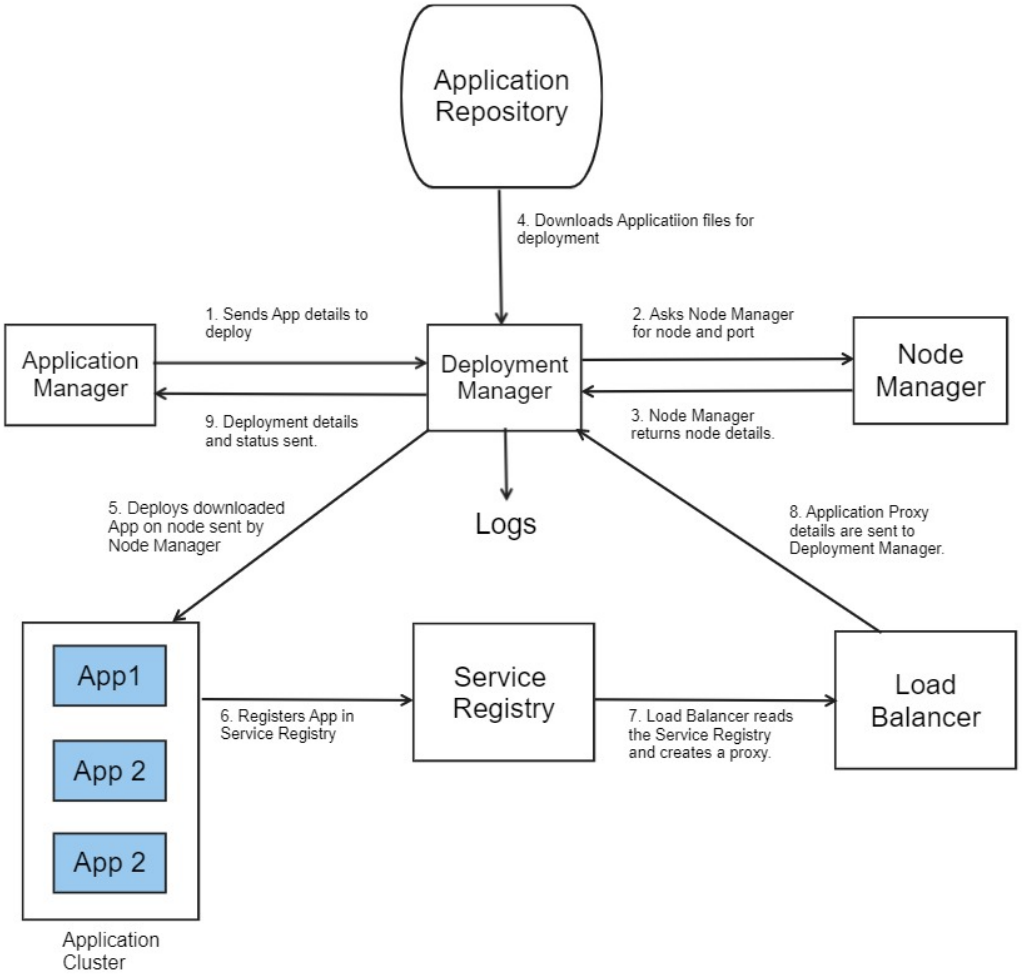
2. Deployment Manager

2.1 Description

Deployment manager module is responsible for deployment of new application instances on the VMs allocated by the Node manager. It requests the Node Manager for new node on which it can deploy the new application instances. It connects to the shared filesystem and to the application repository to get the application artifacts. It generates the files for deployment and runs iton the allocated VM for the deployment of the application.

It then registers it in Service registry and sends requests to the Load balances to create a proxy for the deployed application. This is then returned to the Application Manager.

2.



2.3 Subsystems

• **Shared filesystem connector** – This subsystem helps to connect to the shared filesystem to get the application artifacts and push to the allocated VM for deployment.

• **Deployment Manager** - The Deployment Manager module is tasked with deployment of the application instance requested by the Application Manager.

• **Deployment Files Generator** – This subsystem helps in generating the files dynamically for each application that needs to be run for deploying that application.

2.4 Functional Requirements

• The deployment manager deploys the application instance requested by the Application manager.

• The deployment manager fetches the application artifacts from the shared filesystem.

• The deployment manager dynamically generates the deployment files for each application.

• The deployment manager has a heartbeat API which responds to Fault Tolerance manager.

2.5 Interaction with other modules

• **Application Manager** – Application manager requests deployment of an application instance by sending the application details Deployment Manager after deploying and getting proxy from Load balancer, returns the IP and port for the deployed application instance.

• **Node Manager** – When an application deployment request comes, it requests the Node manager to provide with a node. Node manager provide unique VM with IP.

• **Load Balancer** – Once the application is deployed, it requests the Load Balancer to create a proxy for this deployed application.

• **Service Registry** – It does service registry for the deployed application.

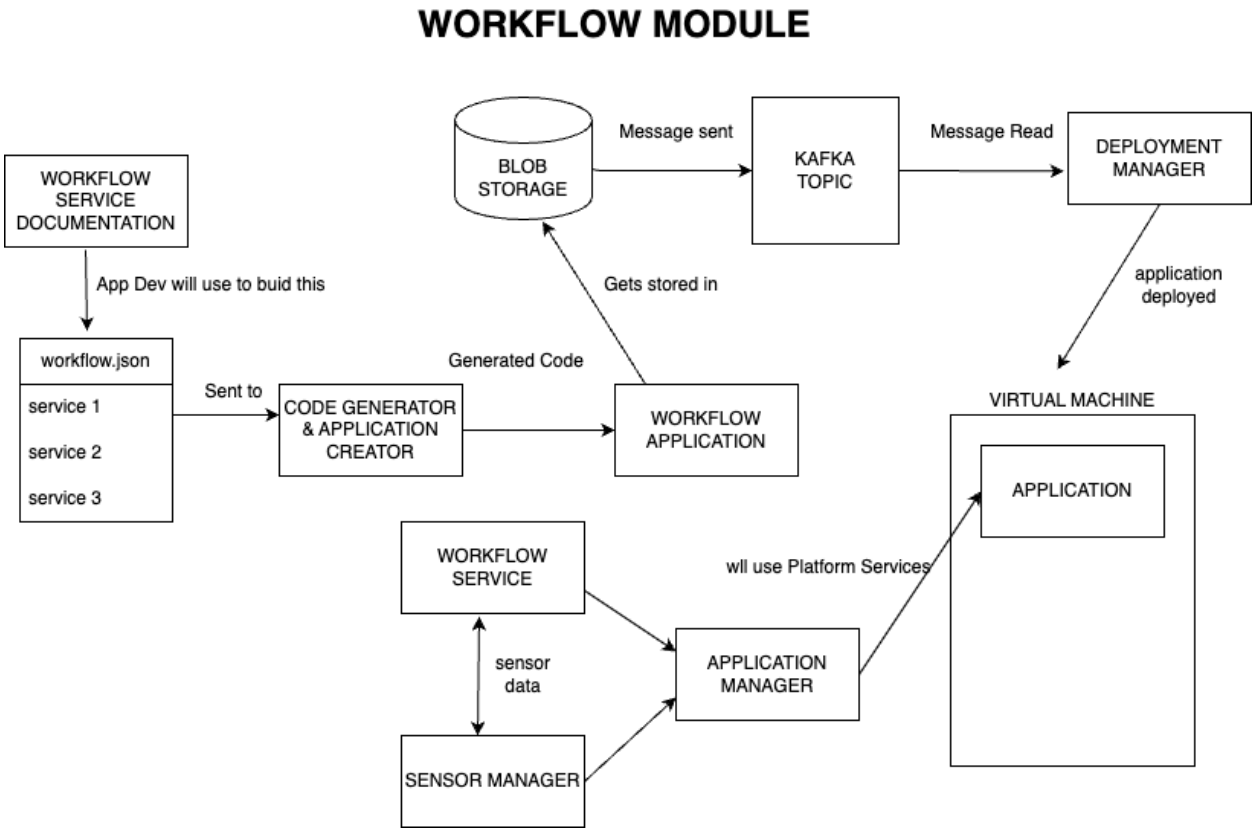
• **Fault Tolerance and Monitoring** – It should response to the heart beat API request by the Fault Tolerance and monitoring.

2.6 Test Cases

Input: Application manager send a deployment request to deployment manager.

Output: New node id is first asked from the Node manager. Deployment manager then does its tasks and deploys the application.

3. Workflow Manager



The Application Developer will make use of existing API i.e Platform Services to create a workflow.json and which will be sent to the Code Generator module. The Code Generator module will generate a code , and store that in the shared Blob Storage. Now, that will be picked by the Deployment Manager module and it will be deployed. The end point of the deployed workflow will be given to the Application Developer and can be used in the application for further use.

Lifecycle:

- When the platform is initialized the Code Generator submodule will be ready to accept the workflow.json file
- After generating code , this will be consumed by Deployment Manager and in return it will send the end point to access that

Functionality

- It serves the purpose of low code by allowing the application developer to just mention the flow of the code.
- It provides the end point of the workflow which can further be used in any application code thus making it reusable.

Interaction with other components

Communicate with application manager : Workflow communicate with application manager to send Application config to upload it in Application DB.