Continuous Integration and Delivery in Microservices

CI/CD is a key element of microservices application. Without it, we will not achieve the agility that microservices promises since there are so many moving parts and no longer a single “blob” to deploy. However, there are challenges and differences to be considered when compared to traditional monolith application. For example, in a traditional monolith application, you often have a single pipeline that output the application executable for deployment. Every development team feed into this pipeline and if one team causes an issue, the pipeline stalled. Throughout the years, many strategies such as feature branching has been used to improve the velocity of the pipeline. Contrast with a microservices application where each team or each microservice could have their own pipeline and release cycle. This led to other issues such as compatibility between versions and rollback strategies. To better grasp the challenges, let us revisit the meaning of continuous integration, continuous delivery and continuous deployment

Continuous integration is the process where the code changes are frequently merged in the main branch. Quality is ensured via automated build and test such that the main branch contains production-grade code.

Continuous delivery is the process where the code changes that goes through CI are automatically made ready for deployment into production.

Continuous deployment means that the code changes that pass through the CI/CD process are deployed automatically into productions.

# Challenges

* Independent Teams: Microservices application are often built by small independent team and there could be many of them. This could lead to a situation where no one knows how to deploy the entire application.
* Multiple Technologies: Because Microservices enables and encourages polyglot development, we can end up with multiple technologies for the same application. Hence, each build system can end being different.
* Integration and Load Testing: There is no clear owner of the entire application because each team is content to do their own little things.
* Release management: Every team can deploy update to the production environment. If a strong and reliable process is already in place, there is no need for a central authority but if that is lacking, we might consider it. It is also a good idea to have different policies for different release type (i.e. major versions vs. minor versions vs. patches)
* Versioning: Versioning are key to communicate changes. Although we thrive to keep things backward compatible, sometimes it is just not possible. Versioning of container images, API, libraries will go a long way to ensure the knowledge of what works with what.
* Updates: Updating services should not break other services.

# Good Practices

* Leveraging container for build environment enables the exact same setup for each build regardless who runs it.
* Local development and testing should be done in Docker. Since we will run them in production with images, might as well get them running locally that way as well. Minikube and Docker Compose can help.
* Keep separate container registries for development/testing and production.
* Define company wide convention for container tags, versioning and naming convention for resources deployed in clusters (pods, services, etc.)
* Take advantages of update strategy like rolling update, blue-green deployment, canary release and public test server
  + Rolling update means that as you deploy new services, the new instance starts receiving requests right away. Depending on the services type, this might or may not be suitable. It is true that the services are available right away but at the same time we might have 2 version of the same services running for the same user. Remember, microservices are stateless and are supposed to be non-sticky.
  + Blue-green deployment is the process where we completely deployed the new version in a side by side manner with the old version. When all validations are done, we flip the switch to route all traffic to the new version. This takes more resources but you only run one version at a time.
  + Canary release is where you roll out the updates to only a handful of clients. As the validation continues and whether it is successful or not, the release is made more widespread until it reaches everyone. This is by far the most complicated way but has the benefit of testing your code with real life scenario.
  + Public test server is the process where we create a totally separate and independent environment where the public can run your application. Analogous view can be Windows Insider build or test server for games. This approach might not work for everything. For example, no one will be using your public test server to do online banking because the data are supposed to be segregated. However, running a test build of the latest iteration of an online game could be useful.