DevOps Assembly Lines

Problem:
In today's fast-growing world of containerization of applications and moving the application architecture from Monolithic to Micro-services, the standard concept of a single pipeline is not relevant.

To fulfill the requirements of such scenarios, the concept of DevOps Assembly lines comes into the picture for any business or project in an organization.

Note: With the above statement, it does not mean that monolithic architecture is not good or valid anymore. It is still used depending on the limitations, & and requirements of the application

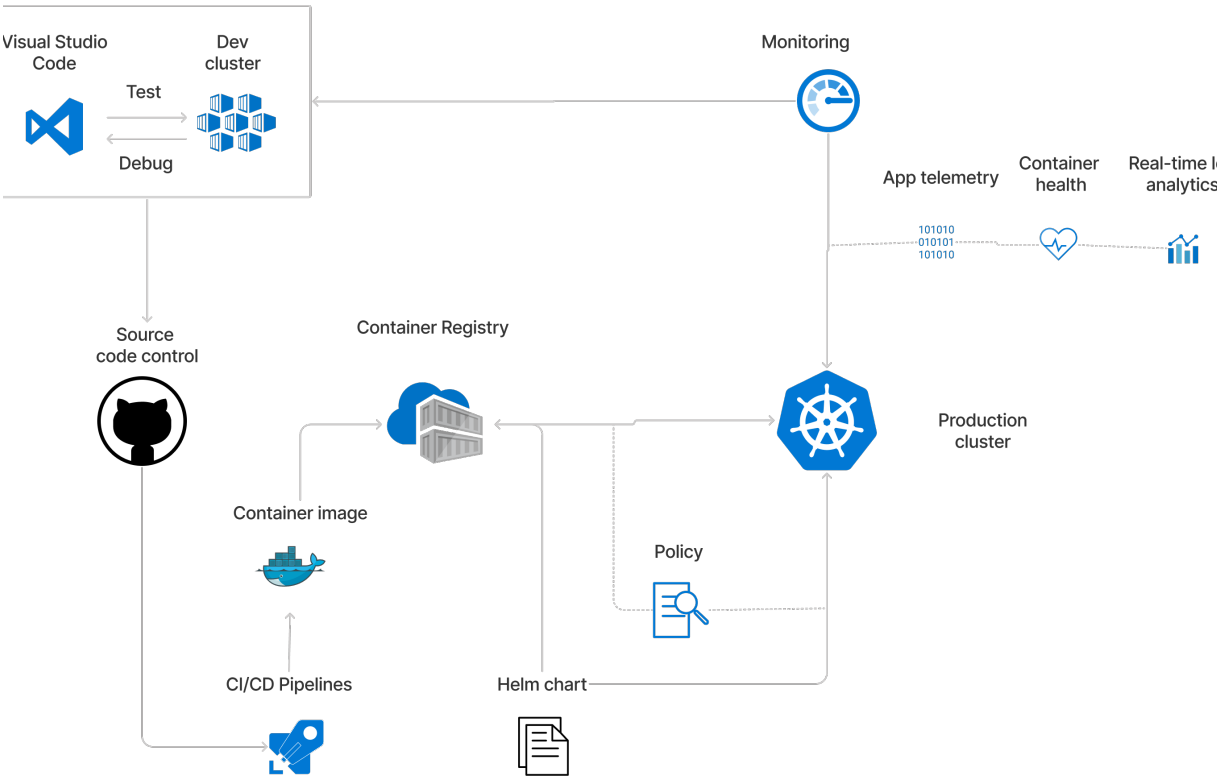
---

Solution/Architecture

We have defined the problem & topic, which we will be discussing in this blog i.e DevOps Assembly lines. Now comes the next question what exactly it is and how will it be used in the micro-services methodology.

Let's first re-iterate in short what is DevOps. I will not be going in detail and assuming that you have a basic idea of the DevOps pipeline structure.

In standard monolithic architecture of application where a single code base is used for all of the component in the application. A single DevOps pipeline will be responsible for deploying the application source code to the production server where it will be accessible by the end customer as shown in below diagram.

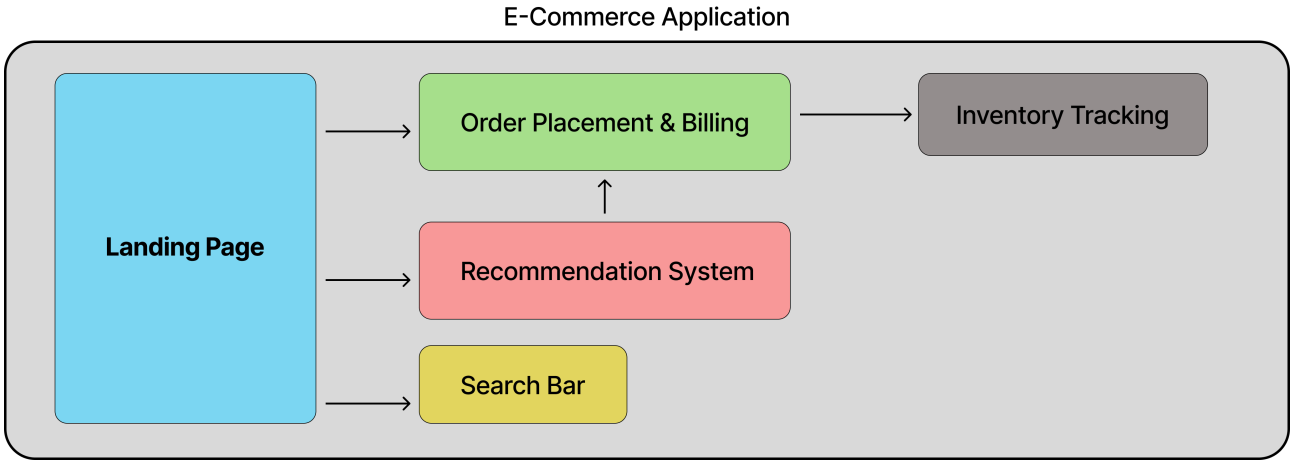


Now, we know what a single standard pipeline looks like in a monolithic architecture. Next, we will see what a microservices application looks like from the top.

In a micro-service design of the application deployment, certain defined rules must be followed otherwise, there is no productive outcome from the business/organization point of view.

Let's take an example of an E-commerce website application that uses the micro-services architecture. The website may consist of the following components:

1. Landing Page of the company
2. Order Placement & billing
3. Recommendation
4. Inventory Tracking



Each of the above components has its own independent source code, timeline, and tracking of code updates, and each other component communicates through the exposed API endpoints.

In other words, the application components are not tightly coupled with one another, allowing the developer and operations team to manage each component independently without much relying on the progress of the other components.

The above architecture is really handy for troubleshooting when we need to figure out why a particular component isn't working without having to delve further into the code base of other components of the application.

In the aforementioned example, we do not have a **single "source code"** location for the entire program rather, we would have **various sub-repos or different source codes** for each component that must be published to the production server.

In this scenario, from the perspective of the end user, the complete program (including all components) must be operational to service the demands of the business otherwise, it might be useless to the end user.

However, with this understanding, the following question arises: what will a DevOps pipeline look like in the given scenario, and how can we build it?

Technical Details and Implementation

Now that we are aware of the issue, we can build the DevOps pipeline we need for the micro-service apps.

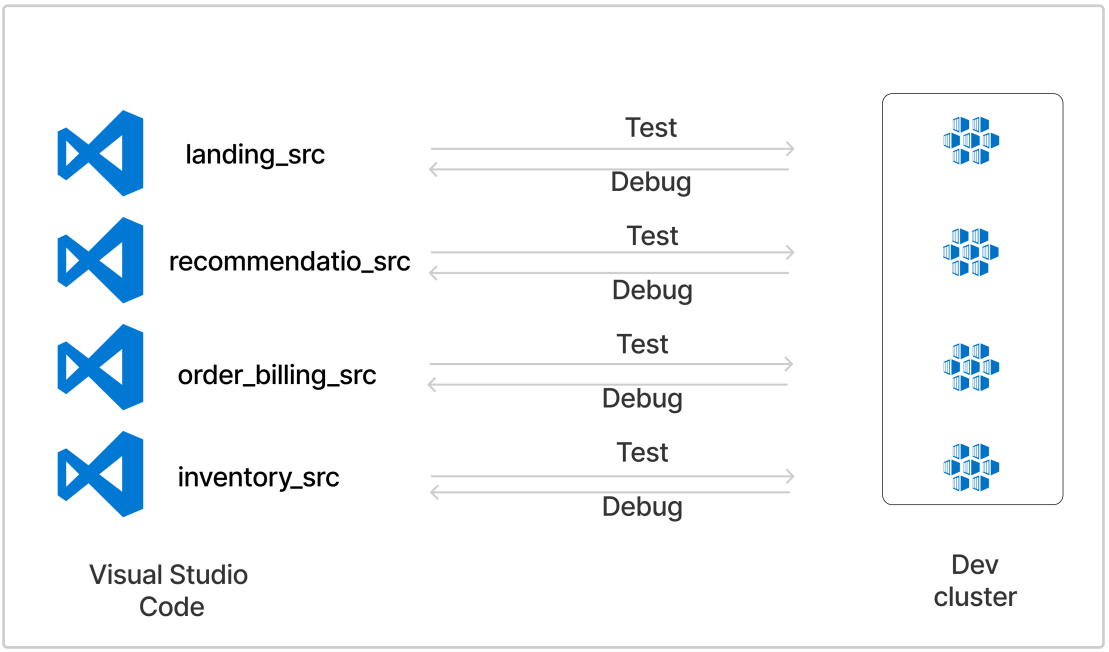
With the help of a diagram at each stage and an explanation of what we are doing specifically, let's see how it will look.

- ***Development***

At initially, each component would have its own team of developers, repository, or both, each of which would be solely in charge of that component. These components may include developers of ML recommendation systems, front-end web developers, and back-end Java developers.

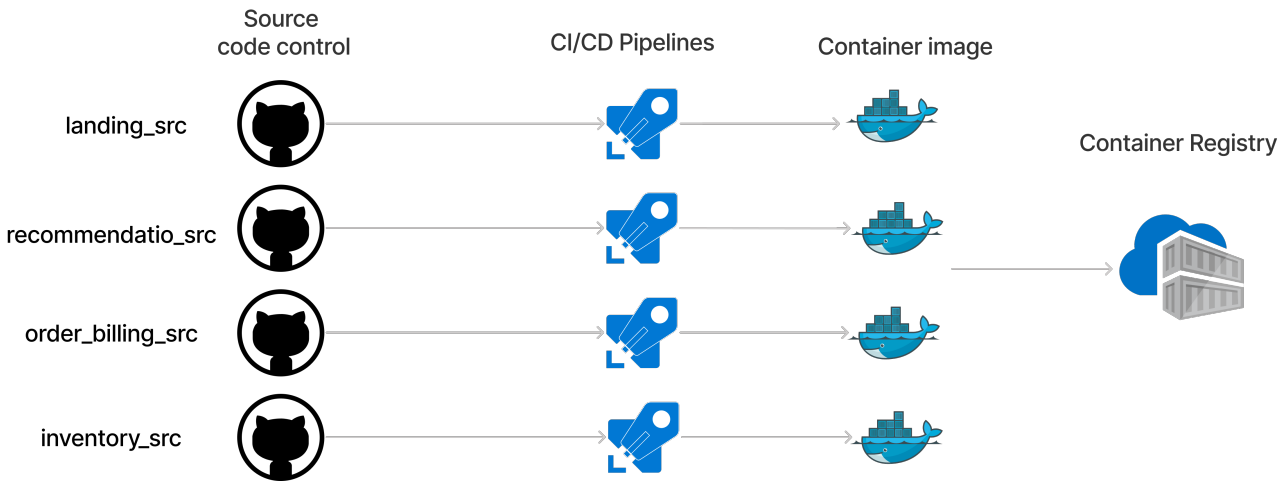
These elements are as follows in our sample scenario:
- Web developer for landing page
- Java developer for order & billing page
- DB engineer for inventory
- Java or python developer for search page, and
- ML engineer for recommendation system.

Additionally, each developer will maintain a copy of their individual code base in a separate repository, allowing for independent progress tracking.

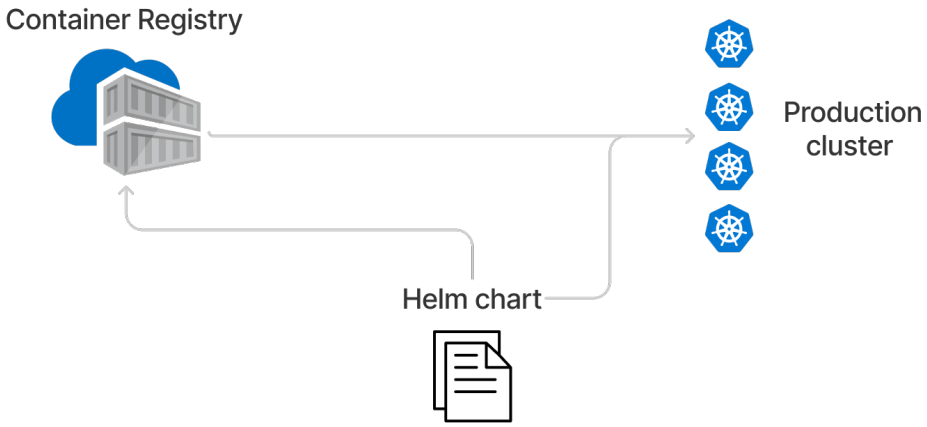


- ***Packaging***

The CI/CD pipeline will be used to build a container image or other artifacts after the code has been developed and merged into that component's main branch. These artifacts are then posted or stored to the artifactory or container registry.

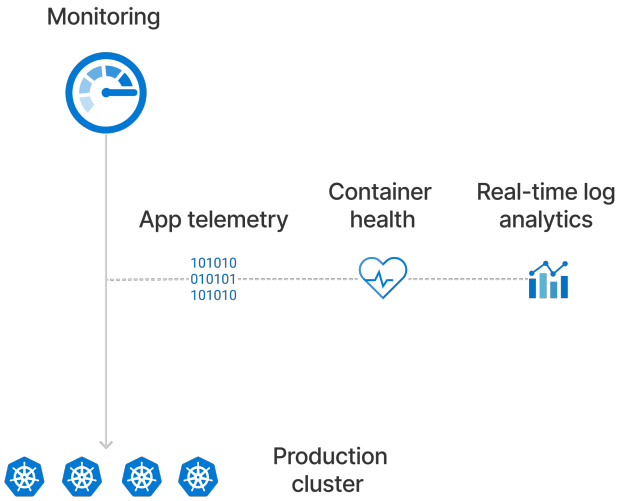


- The application's components are now available and prepared to be installed on the production servers. In this case, we can deploy the Container images/Artifacts onto the production servers using Helm Charts, Ansible, Tekton, or any other CD (Continuous Deployment) tool.
    - It would consist of four components for the complete e-commerce application in our example.



- The organization's work does not cease once the application is successfully deployed to the production servers and begins successfully handling end-user requests. To further enhance the application, it now has to be continuously monitored.

    - Each micro-service in the entire application stack needs to be monitored more closely for how it interacts with the client and with other micro-services.
    - The engineering/developers team will again receive feedback based on the monitored data for use in future work of improvement.



In our example, we've used something interesting: **multiple pipelines working together within a single pipeline.** Each of these pipelines takes on the task of deploying specific components of a single application as they move through the different stages of Development, Testing, and Deployment.
This approach is distinct from the typical DevOps pipeline, which is commonly used for monolithic applications and typically handles all these stages (Development, Testing, and Deployment) as a single pipeline.

This innovative concept of nesting multiple pipelines within a primary one is known as ***'DevOps Assembly Lines.'*** It's a strategy that significantly improves the management and efficiency of the deployment process.

Challenges in the above solution

In the deployment of any application, there are important considerations that demand attention:

1. **Initial Deployment to Production Servers:**

When deploying an application for the first time on production servers, it's crucial to ensure a smooth transition.
In many cases, different teams work independently on various components of the application. While this autonomy is valuable, it can lead to challenges if one component isn't ready when others are.

To prevent this situation, we can employ conditional statements within the Continuous Deployment (CD) pipeline. These conditions ensure that the entire application won't be pushed to production until all dependent components are ready.

2. **Subsequent Updates:**

Even when applying updates to different components of the application is feasible independently with this micro-service design, we need to consider compatibility.
It's essential to verify that updates don't introduce compatibility issues with components already running in production.

To address this, we can implement conditional checks before the final deployment to verify compatibility with the older, already-deployed components.

By addressing these challenges in the deployment process, we can ensure a smoother and more reliable application delivery. If you using Azure DevOps tools in your organisation then for more in-depth information on how to write the pipeline code, you can refer to the official documentation.

Business Benefit

In today's era of microservices architecture, where applications are structured differently from traditional monolithic setups, adhering to the standard DevOps pipeline may no longer be suitable. Instead, there's a need for more optimized and enhanced solutions to reap the full benifits of microservices