

The 12-Factor-App

Principles that need to be considered when building modern applications

=====

Going from an Idea to Execution should only take as much time as you need to develop the code for it, because provisioning and hosting can be achieved in a matter of hours if not minutes.

Most cloud platforms can provision resources in a matter of minutes. And most of the time you do not even need to provision a server - because of serverless technologies - you can simply write code, push code and you are done.

Today's applications are expected to have an uptime of 99.999% meaning there should be no reason to take down an application for any reason whatsoever. Meaning there is no time for application downtime for maintenance outages, patching servers or adding additional resources or scaling, That is breaking free from the underlying infrastructure via portability without having to change the source code of the application.

Tools and processes needed to build modern cloud native applications that can scale and adapt to changing requirements. (Scalable, resilient , Easy to manage using modern tools and techniques)

The twelve-factor app is a methodology for building software-as-a-service apps that:

1. Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
2. Have a clean contract with the underlying operating system, offering maximum portability between execution environments;
3. Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;
4. Minimize divergence between development and production, enabling continuous deployment for maximum agility;
5. Can scale up without significant changes to tooling, architecture, or development practices.

Application summary requirements for application development keeping certain principles in mind

1. Portability - applications should be portable and NOT be tightly coupled with underlying infrastructure.
2. Dev Prod Parity - minimal divergence/ reduce divergence to enable continuous deployment
3. Scalability - Application Must be Easily scalable to enable many instances at once ,Must be built with easy Horizontal scaling in mind
4. Cloud ready - Application must be suitable for deployment on modern Cloud platforms.

In order to achieve the above Application requirements , the application MUST be developed with certain Principles in mind. **These Principles are known as the 12-Factor-App principles.**

1. Codebase
2. Dependencies
3. Concurrency
4. Processes
5. Backing Services
6. Config
7. Build, Release and Run
8. Port Binding
9. Disposability
10. Dev Prod Parity
11. Logs
12. Admin Processes

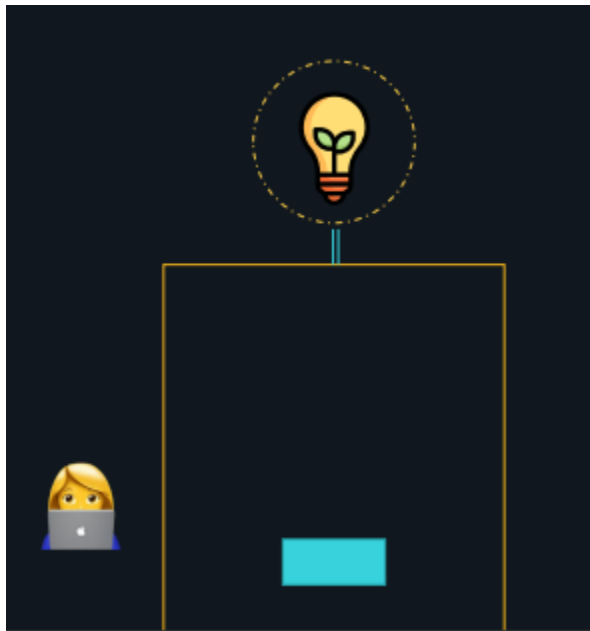
Here is an example using a Simple Flask Python Application to achieve

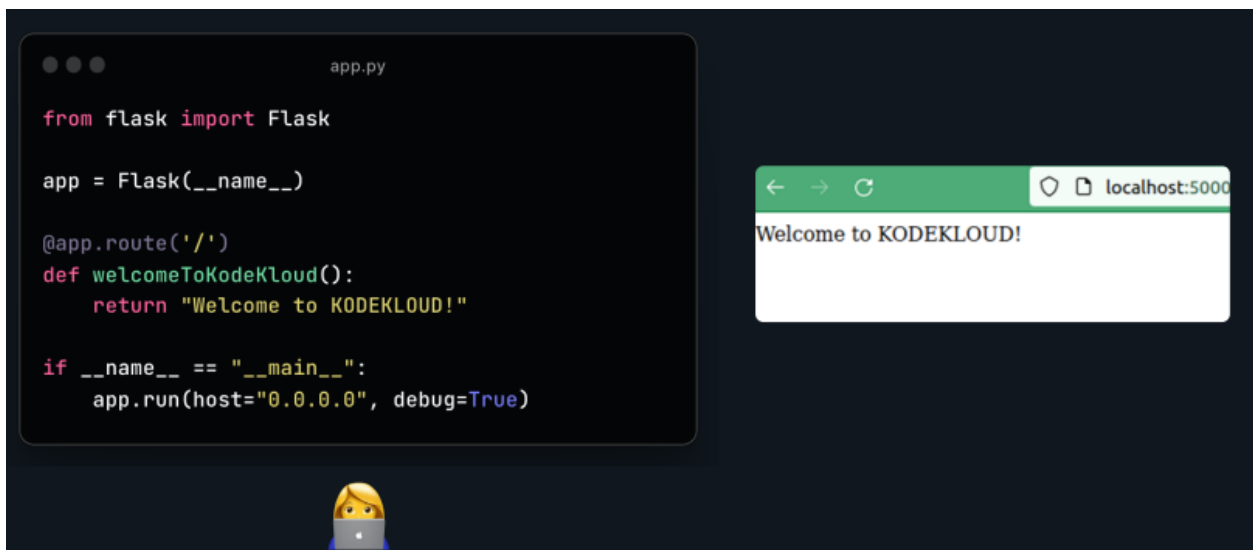
- Portability
- Dev Prod Parity - Continuous Deployment
- Scalability
- Modern Cloud Ready Platform

Keeping in mind some known DevOps principles

- Continuous Integration and Continuous Delivery
- Microservices
- Infrastructure as Code
- Monitoring and Logging
- Communication and Collaboration

Story Line using a Basic Idea





Portability

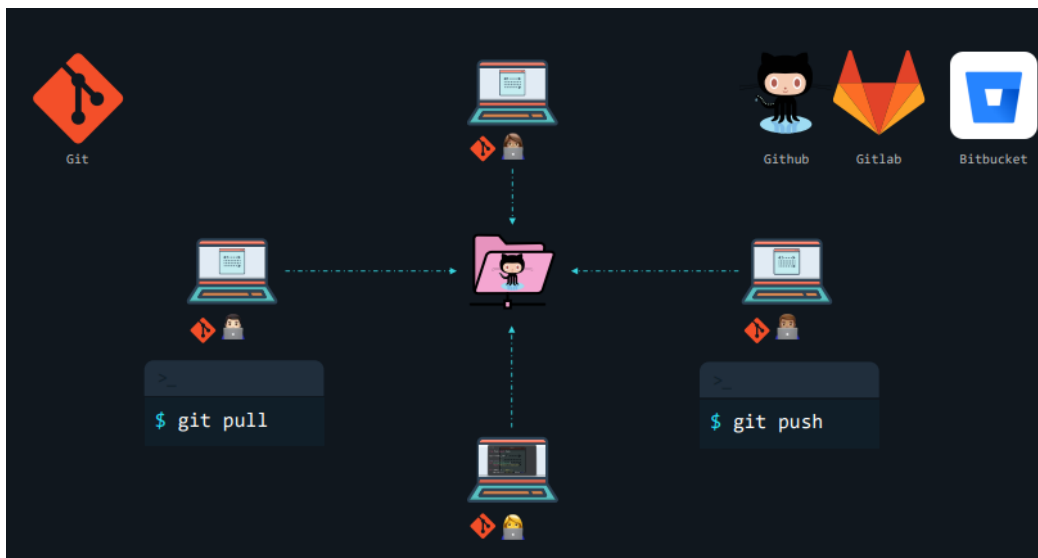
Continuous Deployment

Scalability

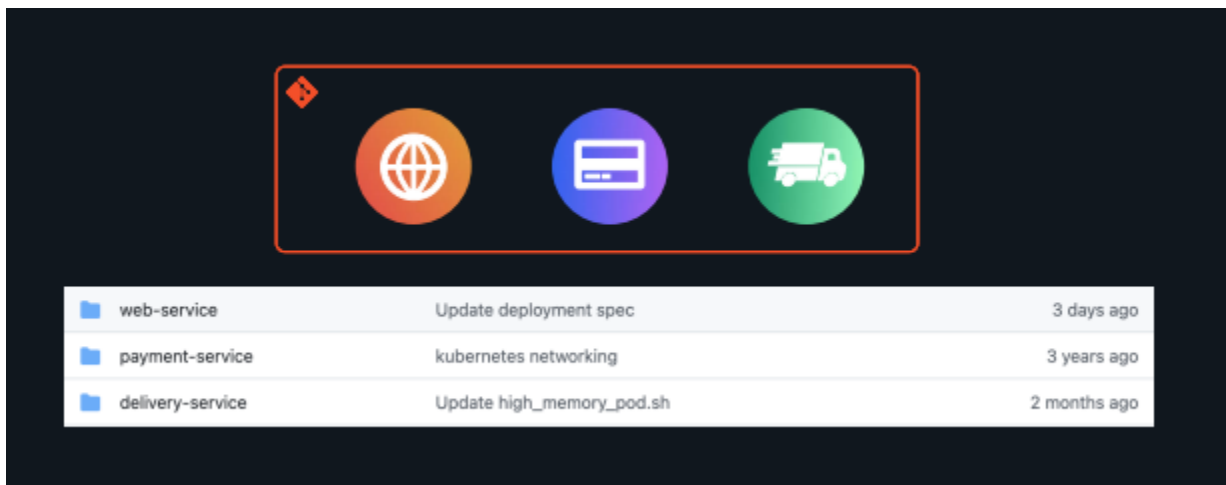
Modern Cloud Platforms

1.CodeBase

- The 12-factor-app states that we must be having a single code base for your application
- The code base must be tracked with revision control
- Working on different environments but on the same code base
- **Solution :**
 - A solution that helps collaboration on code conflicts
 - Mostly using Git
 - And a cloud based central location to push back changes

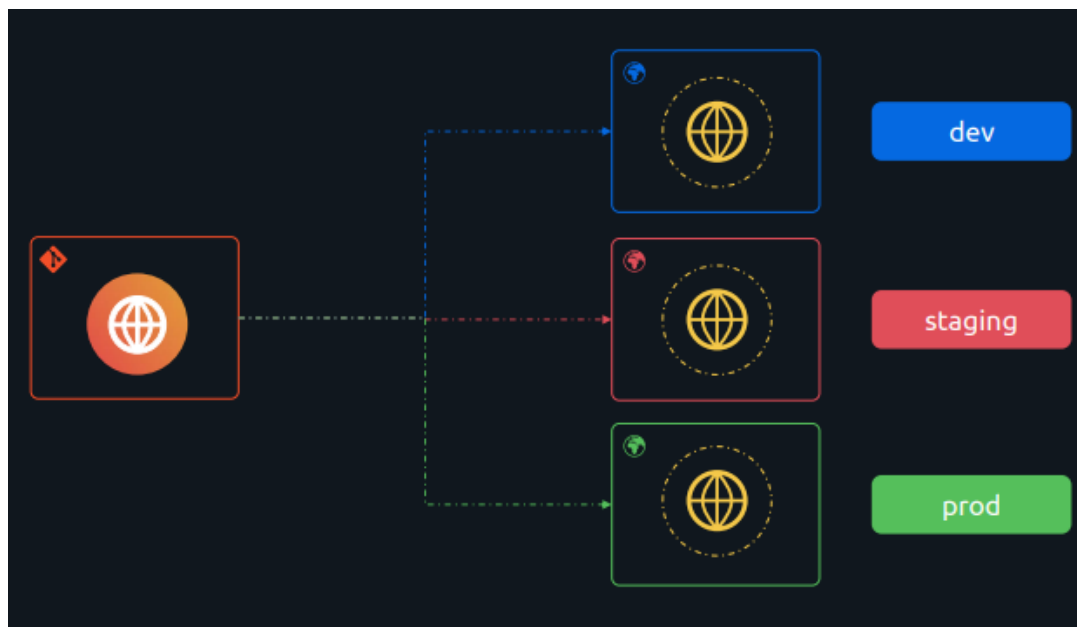


- Multiple apps sharing the same code is a violation of twelve-factor and should be separated into individual code bases.



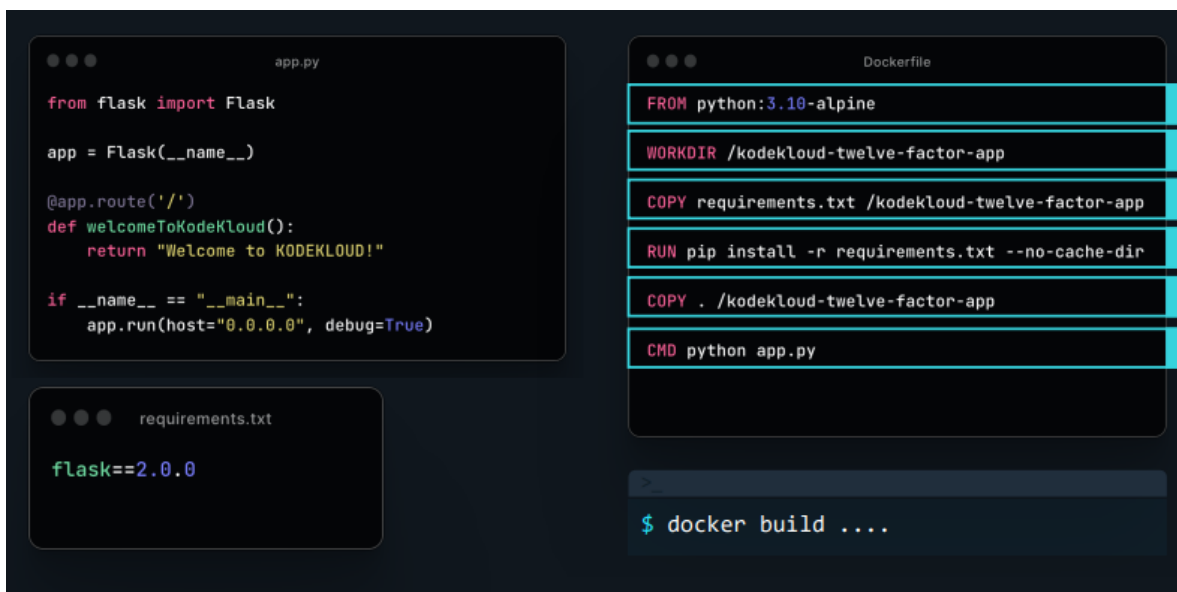


- Each code base should then have multiple deployments - that is Dev, Staging , Prod



2. Dependencies

- Explicitly declare and Isolate dependencies.
- Meaning a 12-factor app never assumes the existence of system wide packages and dependencies where you will be running your app.
- **Solution**
 - Use versioned declarative formats for setup automation and to minimize time and cost for new developers joining the project;
 - Use of a versioned declaration will install all dependencies required for your application to run during the build process
 - Use an isolation mechanism such as Docker to isolate dependencies that an application requires to run on the system. Docker allows applications to run in a self contained environment that is isolated from the host system.
 - A Dockerfile is used to package the application with its dependencies into a docker container.



The screenshot displays three files in a dark-themed editor. The top-left file, `app.py`, contains Python code for a Flask application with a single route `/` that returns "Welcome to KODEKLOUD!". The bottom-left file, `requirements.txt`, specifies the dependency `flask==2.0.0`. The right file, `Dockerfile`, defines the container build process: it starts from `python:3.10-alpine`, sets the working directory to `/kodekloud-twelve-factor-app`, copies `requirements.txt`, installs the dependencies with `pip install -r requirements.txt --no-cache-dir`, copies the application code, and sets the command to `python app.py`. Below the Dockerfile, a terminal snippet shows the command `$ docker build`.

```
app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def welcomeToKodeKloud():
    return "Welcome to KODEKLOUD!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)

requirements.txt
flask==2.0.0

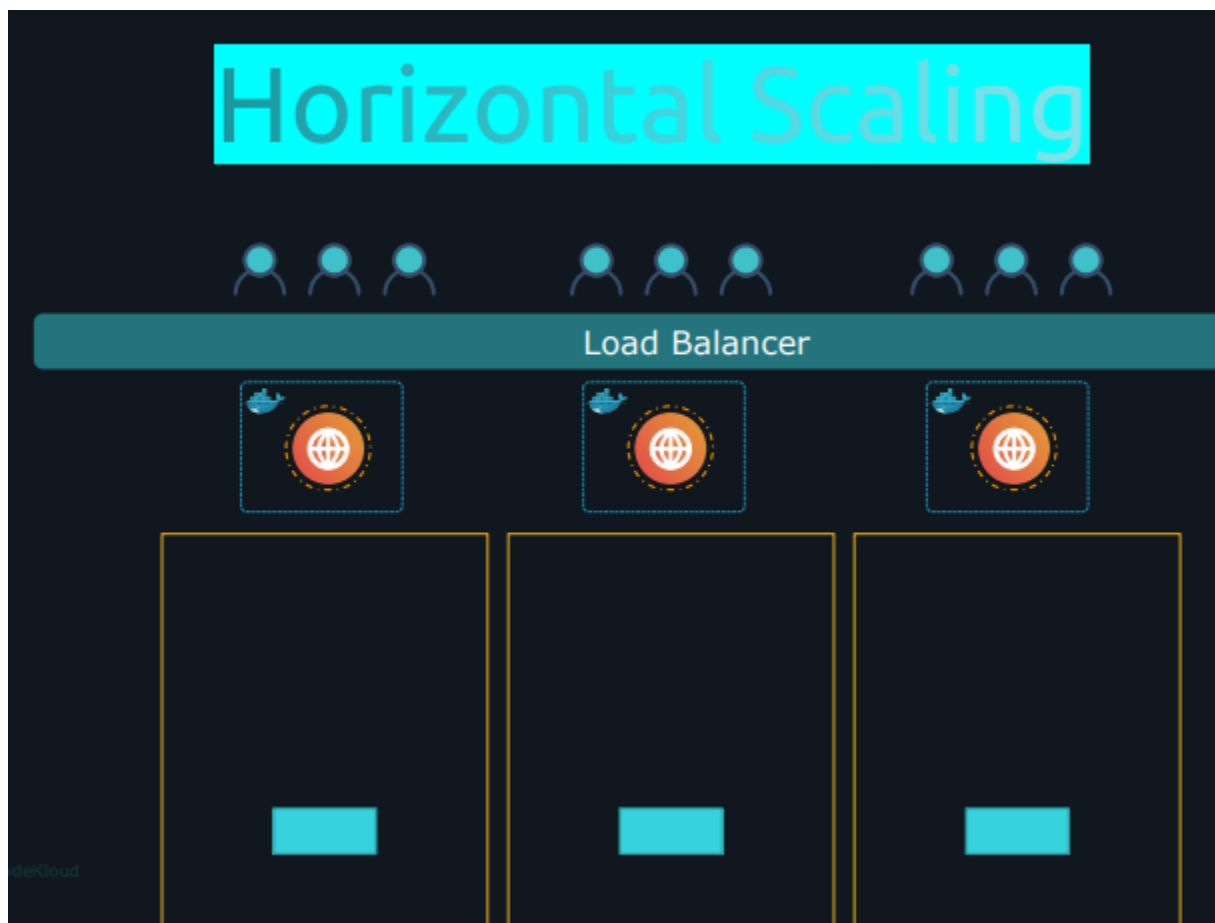
Dockerfile
FROM python:3.10-alpine
WORKDIR /kodekloud-twelve-factor-app
COPY requirements.txt /kodekloud-twelve-factor-app
RUN pip install -r requirements.txt --no-cache-dir
COPY . /kodekloud-twelve-factor-app
CMD python app.py

$ docker build ....
```



3. Concurrency

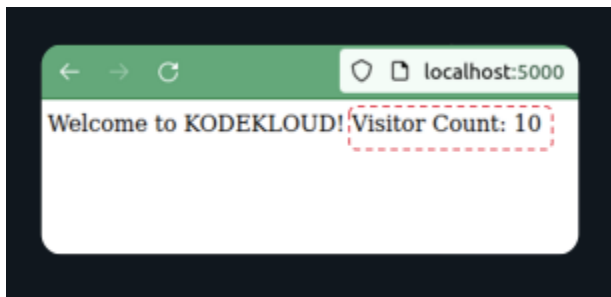
- What happens when we have more users visiting the application?
- One solution can be Vertical scaling. However vertical scaling will cause downtime and will ultimately hit maximum limits or resources that can be added to the host system
- **Solution**
 - In a 12 factor system the recommendation to design and build applications with Horizontal scaling capability as opposed to Vertical scaling,
 - That is running multiple instances of the application concurrently
 - A capability of a 12 factor system is to be able to provision more instances of an application with great ease and within minutes.
 - A load balancer can then direct traffic to the multiple instances of the application.



4. Processes & Backing Services

Using an additional feature that shows total visitor count every time a user visits the application and increments each time a new visitor logs in.

- Works well when we only have one instance running because the visitor count is stored inside the memory of that running instance
- When we run multiple instances, they will all have their own versions of this count variable stored locally inside them.
- This is true also for users session information that keeps a user logged in until a task is completed and a future request is directed to another instance.



```
app.py

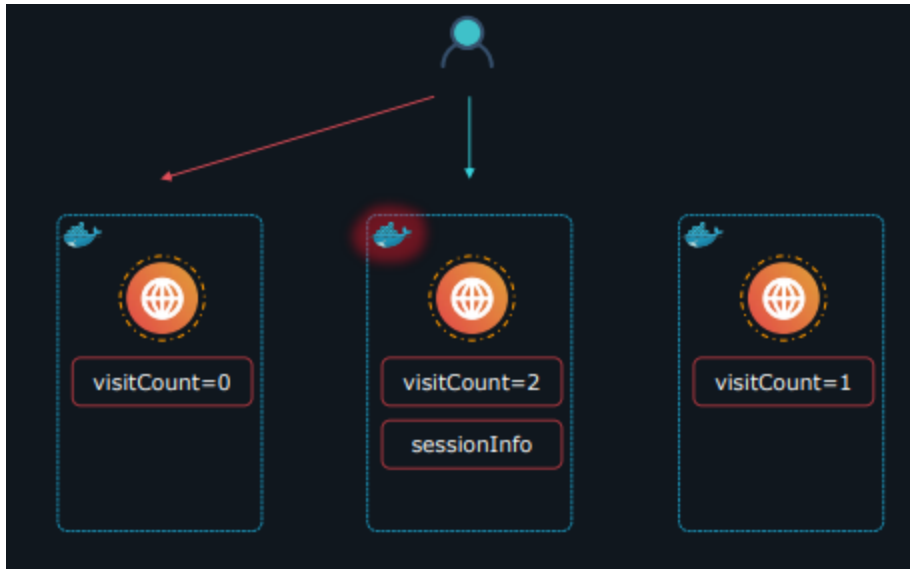
from flask import Flask

app = Flask(__name__)

visitCount = 0

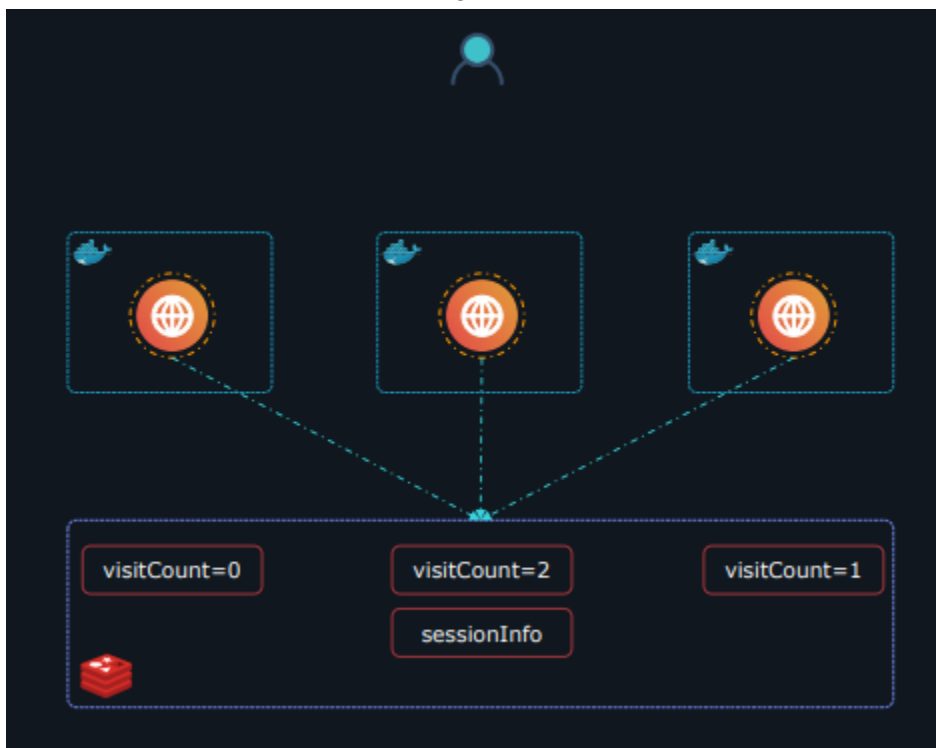
@app.route('/')
def welcomeToKodeKloud():
    global visitCount
    visitCount+=1
    return "Welcome to KODEKLOUD! Visitor Count: " + str(visitCount)

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

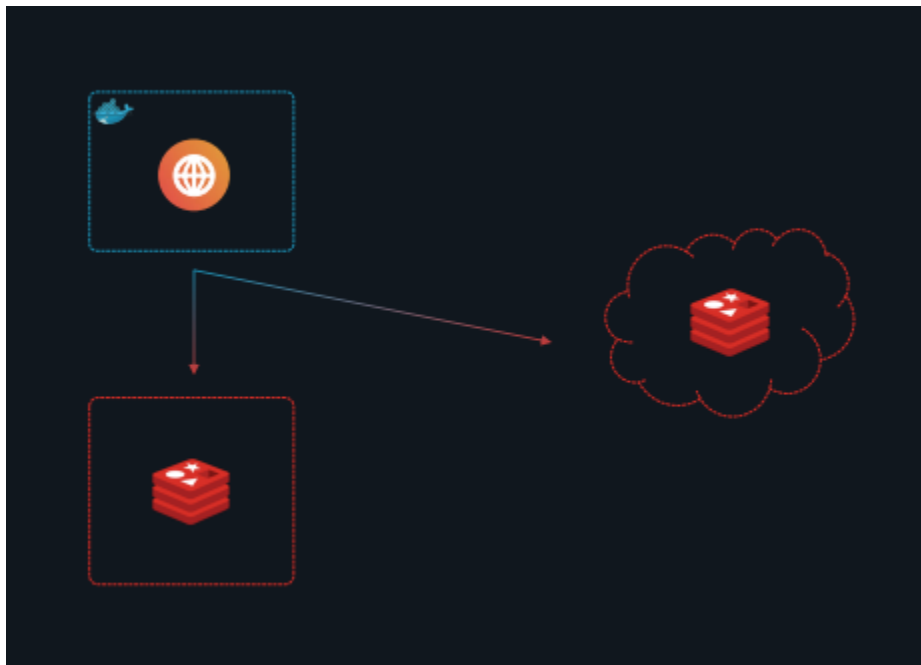


- Solution

- With Concurrency comes the need for Backing services.
- Meaning the application design should be built as an independent **stateless** app.
- All session information should be stored externally in a Backing service that can be easily accessed by all running instances and all processes having the same set of data allowing many instances of the application to run as required while ensuring we store nothing locally.
- The external Backing services can be a database or a caching service like **Redis**



- All Backing services must be treated like attached resources.
- Meaning application must be able to just work by pointing to another backing service resource instance wherever it may be without having to change the application code.



```
app.py

from flask import Flask
from redis import Redis

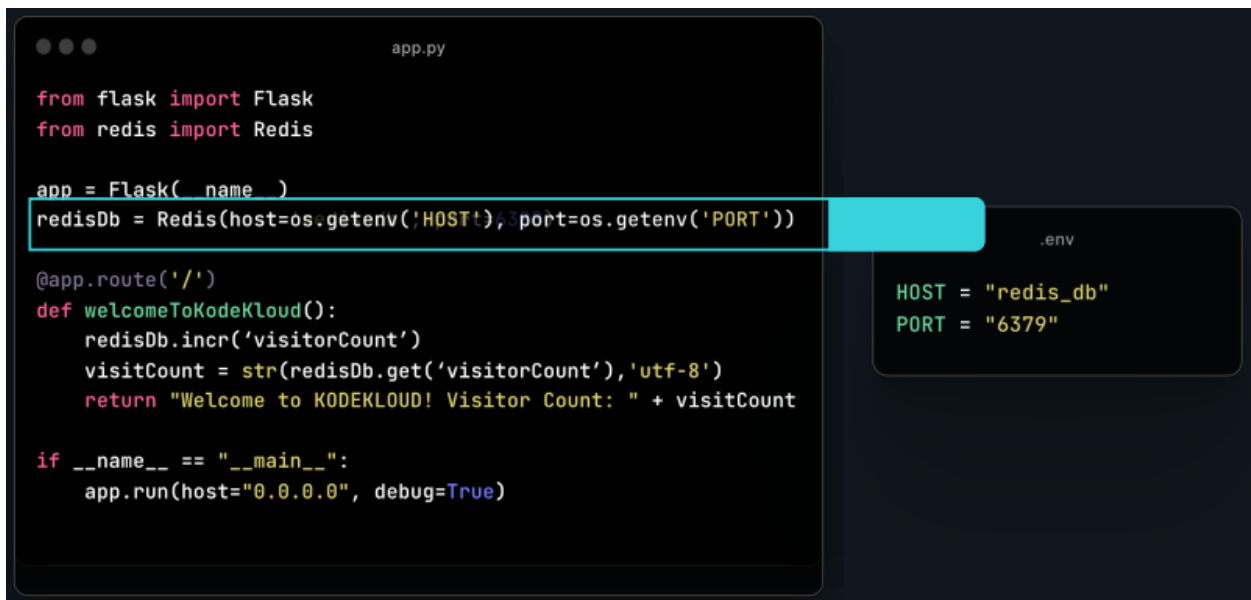
app = Flask(__name__)
redisDb = Redis(host='redis-db', port=6380)

@app.route('/')
def welcomeToKodeKloud():
    redisDb.incr('visitorCount')
    visitCount = str(redisDb.get('visitorCount'), 'utf-8')
    return "Welcome to KODEKLOUD! Visitor Count: " + visitCount

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

5. Config

- Using the application Redis configuration example above, the configuration includes hard coded host string and port value.
- In the 12 factor applications, configuration information must not be hard coded , as this presents problems in inconsistencies and errors while deploying the application to different environments e.g. dev, staging and production.
- **Solution**
 - The 12 factor App stores configs in environmental variables which allows deployments for different environments.
 - Secrets are also used to prevent exposure of sensitive information to the public since the application would fetch configuration information as variables.
 - The configs are then defined and stored in a separate configuration file from the main application code.
 - Lastly, ensure environment configurations are under version control.
 - An example would be kubernetes secrets configurations - loading environment specific configuration data in a separate file - without the need for code level changes and without exposing sensitive information to the public).



The image shows a code editor with two files. The main file, `app.py`, contains Python code for a Flask application that connects to Redis using environment variables. A line in the code is highlighted with a red box. To the right, a smaller window shows the `.env` file with environment variables for `HOST` and `PORT`.

```
app.py
from flask import Flask
from redis import Redis

app = Flask(__name__)
redisDb = Redis(host=os.getenv('HOST'), port=os.getenv('PORT'))

@app.route('/')
def welcomeToKodeKloud():
    redisDb.incr('visitorCount')
    visitCount = str(redisDb.get('visitorCount'), 'utf-8')
    return "Welcome to KODEKLOUD! Visitor Count: " + visitCount

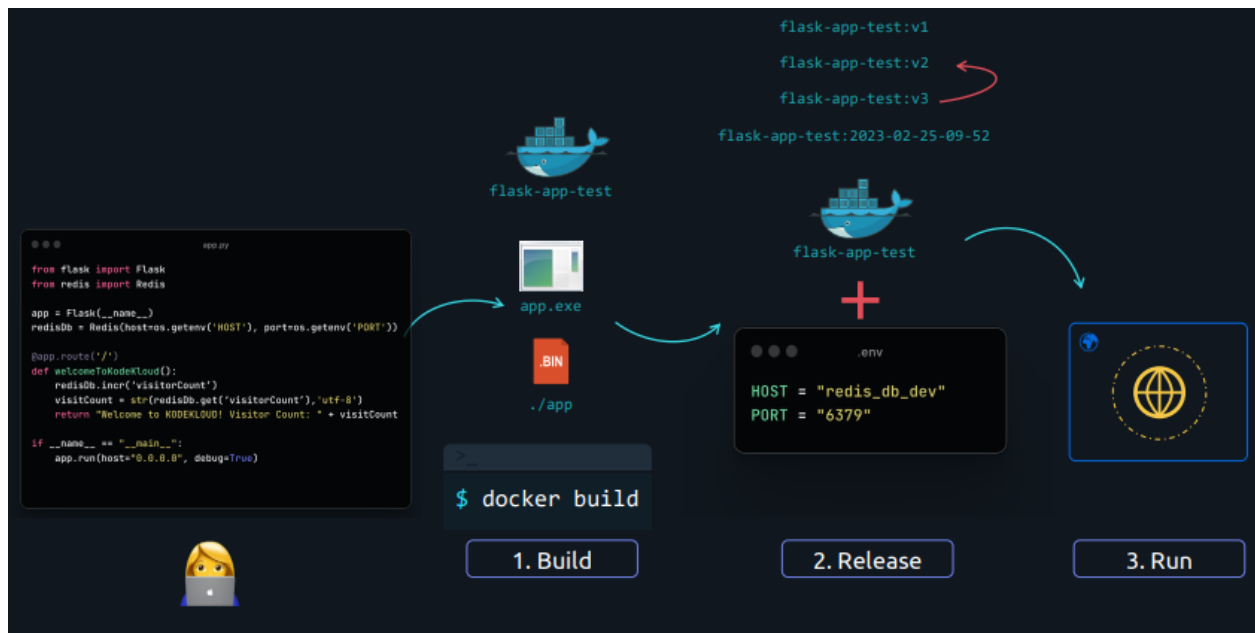
if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

```
.env
HOST = "redis_db"
PORT = "6379"
```



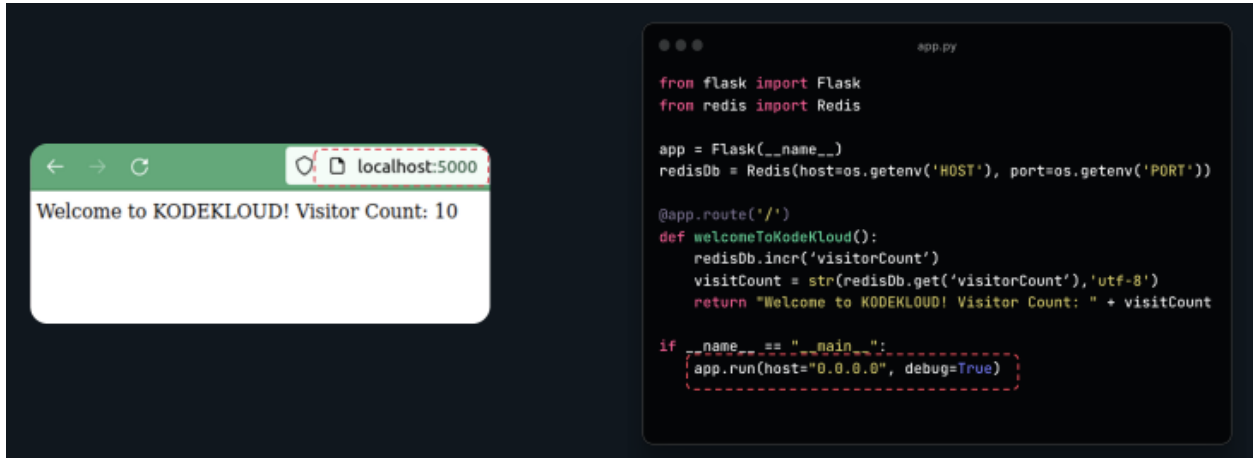
6. Build, Release and Run

- This is a key principle in the 12 factor app methodology.
- The 12 factor app uses a strict release process by separating between the build, release and run stages, and this improves our overall ability to manage and maintain our software.
- **Solution**
 - Build Phase: From the developers code, we Build, converting the code from a text format into an executable format.
 - Once built, the executable image + environment config file becomes the Release object.
 - Every release should have a unique Release ID or a timestamp associated with it, so that it is easy to recognize when it was created.
 - Run Phase, where you run the release in the respective environment so that the exact same build is used to run in different environments ensuring that we have the same code base running in the environments in a consistent fashion .
 - Any minor change in the code results in a new build process that should create a new release and a new deployment.
 - By separating our Build and Run process we can effectively manage our build artifacts and deployments.
 - By having a distinct Build Phase, we can store our build artifacts in a designated location allowing the ability for easy Rollbacks to previous releases or ability to Re-deploy to a specific release as required.



7. Port Binding

- The 12-factor-app is completely self-contained meaning, and does not rely on a specific web server to function.
- Meaning having the ability to export and bind multiple application ports on the server.



The top part of the image shows a web browser window at `localhost:5000` displaying "Welcome to KODEKLOUD! Visitor Count: 10". To the right is a code editor showing the `app.py` file:

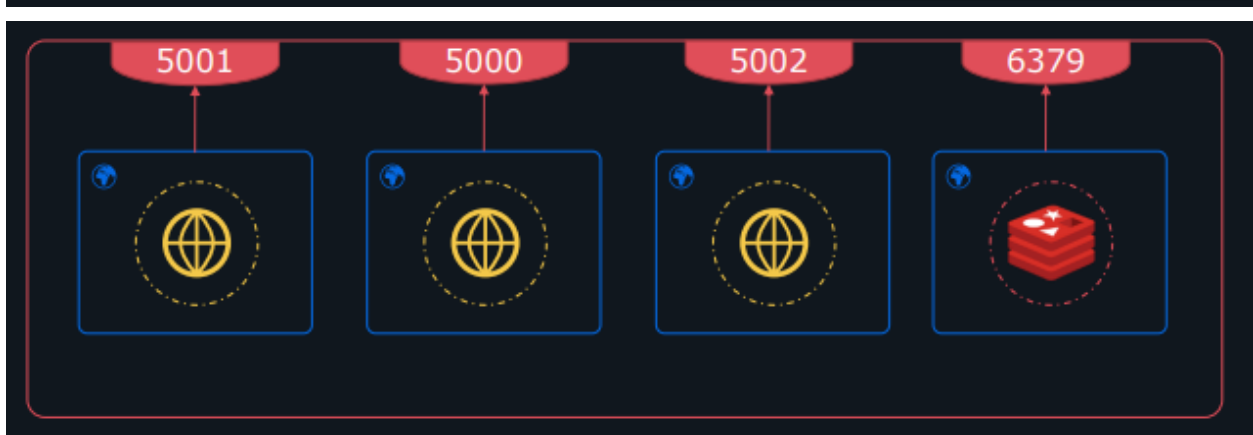
```
app.py

from flask import Flask
from redis import Redis

app = Flask(__name__)
redisDb = Redis(host=os.getenv('HOST'), port=os.getenv('PORT'))

@app.route('/')
def welcomeToKodeKloud():
    redisDb.incr('visitorCount')
    visitCount = str(redisDb.get('visitorCount'), 'utf-8')
    return "Welcome to KODEKLOUD! Visitor Count: " + visitCount

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```



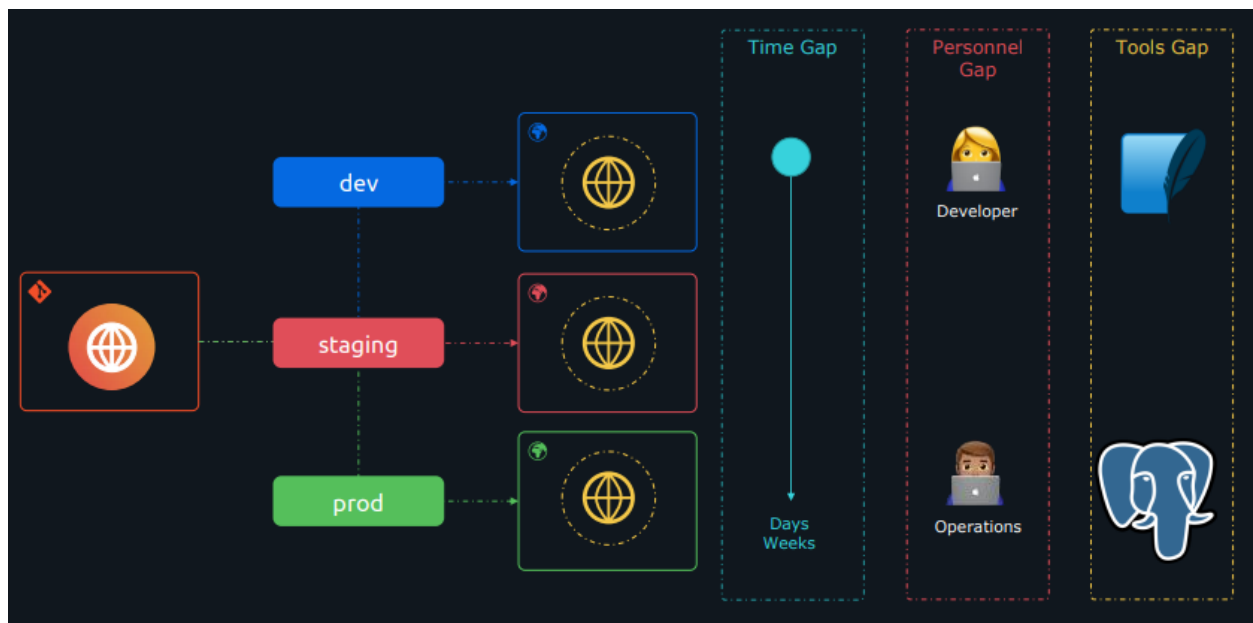
The bottom part of the image is a diagram showing four application instances, each represented by a blue box with a globe icon. Above each box is a red pill-shaped label with a port number: 5001, 5000, 5002, and 6379. The first three boxes have a globe icon, while the fourth box has a red cube icon. Arrows point from each box to its corresponding port label.

8. Disposability

- The 12-factor-app's processes are disposable, meaning Applications can be started or stopped at a moment's notice by striving to reduce startup time.
- The 12-factor-app should be able to horizontally scale to provide additional resources in a moment's notice or ideally in a matter of seconds, when requirements increase.
- For this to happen processes must strive to reduce startup time.
- The same should be true when decreasing instances when the load reduces, processes should be disposable when no longer required with Graceful shutdowns.
- **Solution**
 - Docker
 - Kubernetes, Cloud provides e.g. AWS, Azure

9.Dev / Prod Parity

- The 12-factor-app is designed for continuous deployment by keeping the gap between development and production small.
- Challenge
 - Time gap - between development and the time it takes to the production stage.
 - Personnel gap - different set of people Developing the code and a different set of people deploying the code.
 - Operations may have little to no knowledge of the changes coming from development making it hard to identify issues caused by new changes.
 - Difference set of tools, a test DB and a production DB



- **Solution**
 - 12-factor-app resists the urge to use different services between development and production
 - Use of CI/CD principles, and tools to build Pipeline configurations, to reduce the time it takes to move changes from Dev to Production in consistent fashion.
 - Involvement of developers and operations in deployment, incorporating Dev and Ops in the same team having the same mindset and aiming to keep the same tools as much as possible.

10. Logs Management

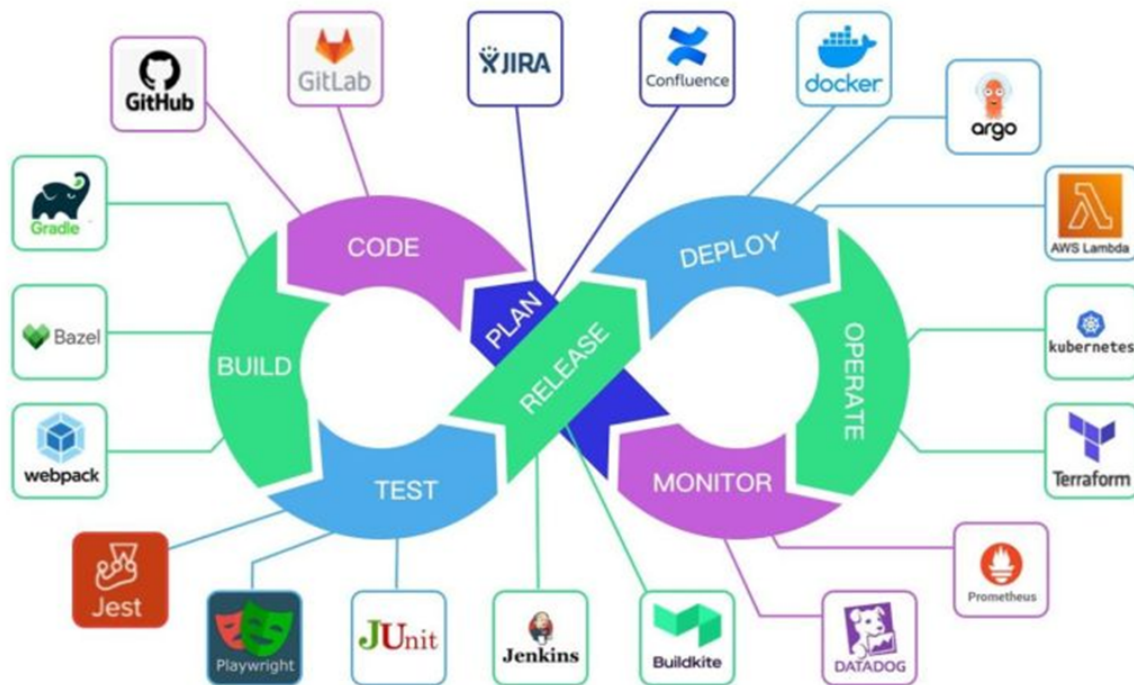
- The 12-factor-app states that Logs should be stored in a centralized location in structured format, where a centralized solution picks (e.g Fluentd).
- Traditionally applications followed different approaches to storing logs. E.g. writing logs to a local file.
- In a 12-factor-app applications should send logs to a central location for centralized management of logs.
- Tightly coupling logs to the app itself is discouraged.
- Application should never concern itself with routing or storage of its output stream
- Application must maintain a stdout put in structured json format which is then processed by an external agent to transfer the logs to a centralized location enabling consolidation and easier analysis
- All logs should follow a structured format that will make it easy to query and analyze.
- **Solution**
 - ELK , Splunk , Prometheus



11. Admin Processes

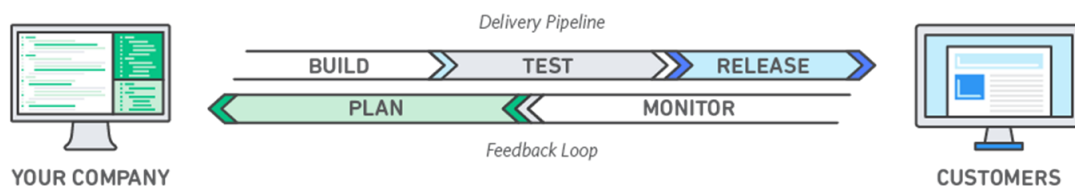
- The 12-factor-app states that Administrative tasks should be kept separate from the application process.
- The administrative development set-up should be as identical as the processes running in production.
- The administrative process should be automated, scalable and reproducible.

What is CI/CD Pipeline



DevOps Model Defined

- DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes.
- This speed enables organizations to better serve their customers and compete more effectively in the market.



Tools and Process used at Co-op

12-Factor-App Principles	Implementation at Co-op
Codebase Dependencies Concurrency Processes Backing Services Config Build, Release and Run Port Binding Disposability Dev Prod Parity Logs Admin Processes	CodeBase <ul style="list-style-type: none"> • GitHub , TFS • Collaboration - Code reviews, Branch management , Issue Triage • Microservices designs • Using Docker as an Isolation mechanism Configs and Dependencies <ul style="list-style-type: none"> • Deployment File separate from code • Use of secrets in Kubernetes • Separate configs • Using Docker as an isolation mechanism • Portability and scalability using Docker
DevOps Principles	
Communication and Collaboration Continuous Integration Continuous Delivery Microservices Infrastructure as Code Monitoring and Logging	Backing services <ul style="list-style-type: none"> • Use of Redis for caching sessions Dev Prod Parity <ul style="list-style-type: none"> • Continuous Integration and Continuous Delivery • Jenkins for CICD Pipeline Integrations • Argo CD for CICD GitOPs Deploy Operations • Kubernetes to achieve Infrastructure as Code
Future Implementations for Modern Cloud Ready Platforms	
<ul style="list-style-type: none"> • Red Hat Openshift 	