Thinknyx®
You Trust, We Deliver!

Cloud Native Buildpacks for the Absolute beginners - HandsOn

Yogesh Raheja

CERTIFIED kubernetes ADMINISTRATOR

kubernetes AND CLOUD NATIVE ASSOCIATE

Puppet for the Absolute Beginners – Hands-On

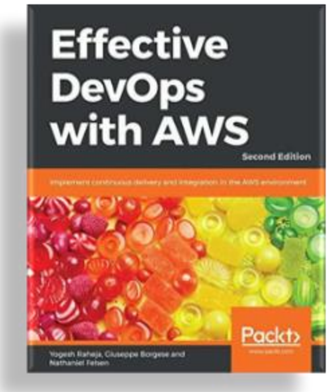
The Ultimate Linux Bootcamp for DevOps SRE & Cloud Engineers

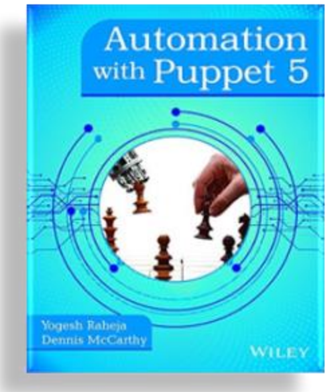
Practical Kubernetes – Beyond CKA and CKAD

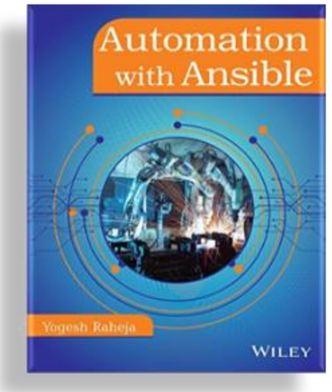
Argo CD for the Absolute Beginners - Hands-On

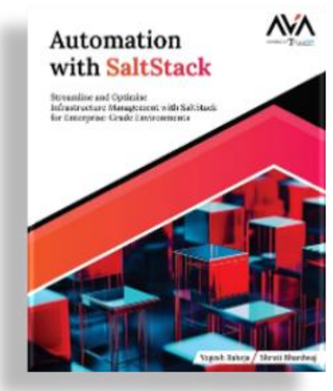
Kubernetes and Cloud Native Associate

Mastering Docker Essentials Hands-on









Yogesh Raheja

Thinknyx Team

Buildpack fundamentals

The key terminologies

Pack CLI

Building various application images using Cloud Native Buildpacks

Developing your own Buildpacks and Builders

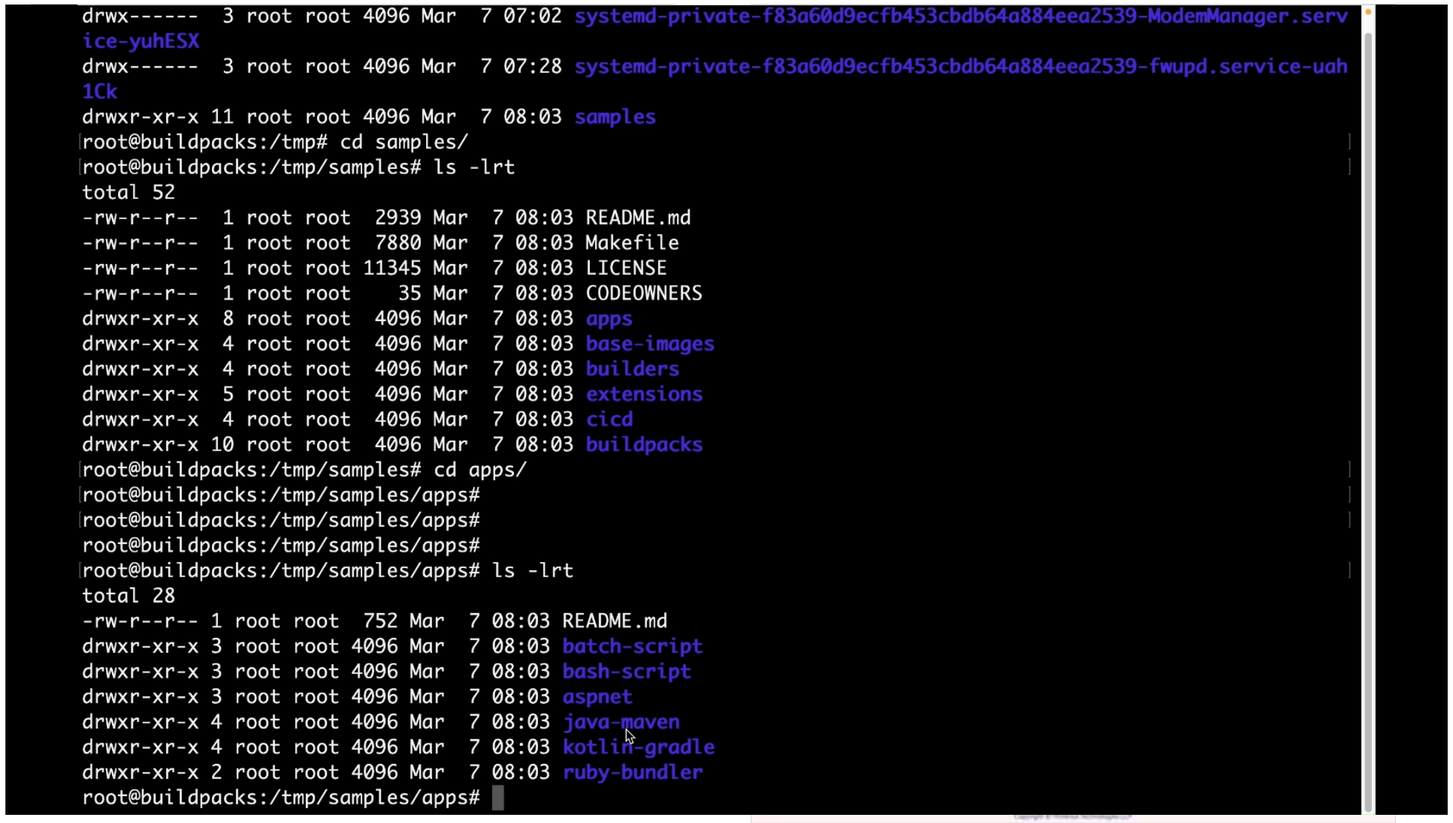
Kubernetes integrations with kpack to build container images

Course Overview

Lectures

Demonstrations

Practice Tests



Cloud Native Buildpacks in production environments

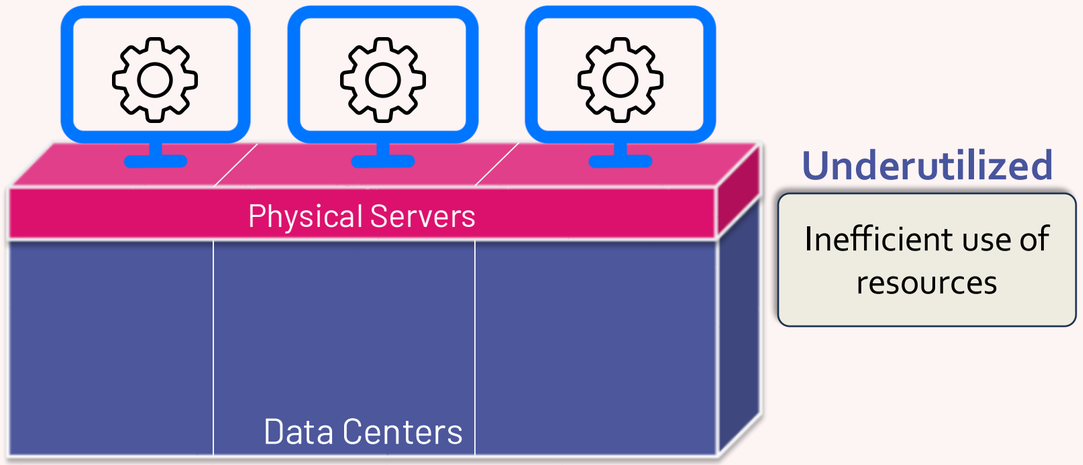
Buildpacks

Buildpacks

- Container Images
- History, Key features, and Benefits of Cloud Native Buildpacks
- Official Documentation

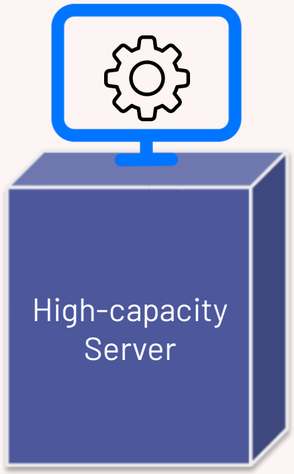
Getting Started with Container Images

Container Images



Underutilized

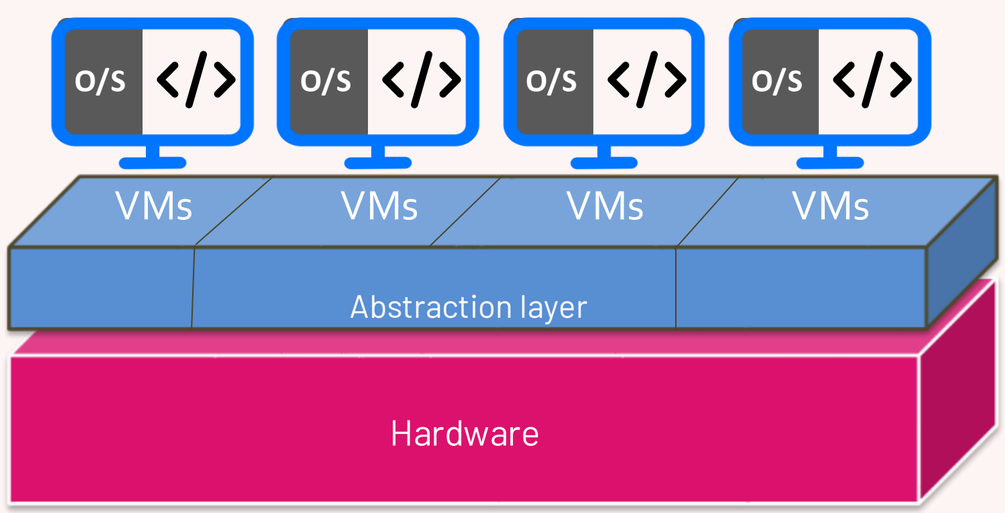
Inefficient use of resources



Resource wastage

Virtualization

Improves resource utilization by running multiple workloads



- ✓ Processors
- ✓ Memory
- ✓ Storage

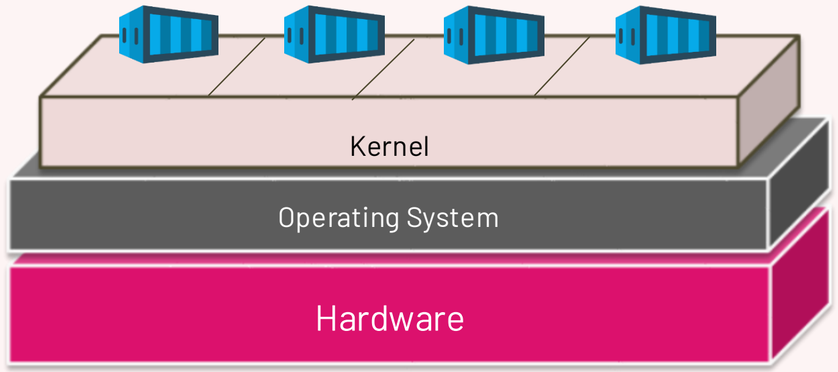
Limitations

- Time consuming
- Resource-intensive

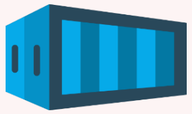
VMs

Container

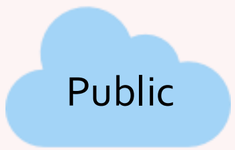
- **Containers are lightweight and modular**
- **They package code with dependencies**
- **Ensures portability and isolation**



Operating System Virtualization



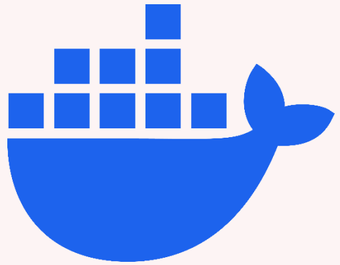
Container



✓ Reduce deployment time
✓ Resource consumption

Docker

- Most widely used solutions
- Provides more than just a runtime
- Complete container lifecycle platform with an intuitive UI

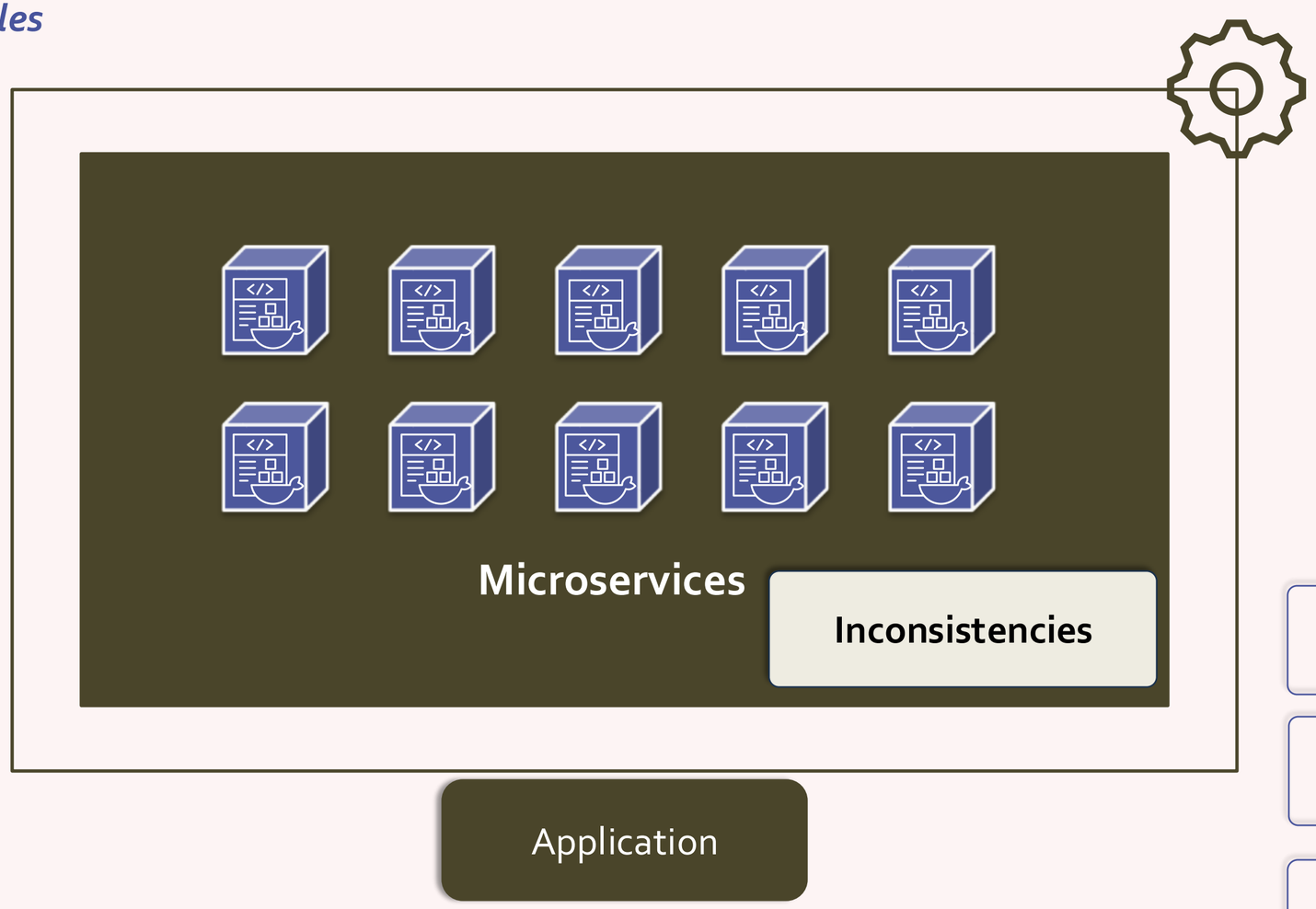


Docker

Container Image | Base Images | Dependencies | Configurations

Docker

- **10's to 100's** of such ***applications*** with **100's to 1000's** of ***microservices*** and their respective ***dockerfiles***



Security Vulnerabilities

Maintainability Challenges

Performance Inefficiencies

Cloud Native Buildpacks
Or
CNB's Safeguards

Buildpacks address these issues by automating the process of converting source code into container images in a standardized and secure manner

What Are Buildpacks and Their Origin

What Are Buildpacks

Buildpacks

Transforming application source code into a container image

- ✓ Write
- ✓ Maintain

Dockerfiles

Buildpacks Analyze

Necessary Dependencies

Consistency

Security

Efficiency

Origin of Buildpacks

HEROKU

2011

Streamline the process of deploying applications by automating the configuration of runtime environments

Manage and configure all dependencies manually

Origin of Buildpacks

- **Buildpacks effectively eliminated much of this complexity**

Programming Language

Framework

Libraries

- ✓ Focus on writing code
- ✓ Underlying infrastructure

Origin of Buildpacks

PaaS

Cloud Foundry

Pivotal Cloud Foundry

GitLab

Google App Engine

Cloud Foundry adopted buildpacks to support multiple languages and frameworks with minimal configuration

Origin of Buildpacks

Kubernetes

January 2018

Pivotal

HEROKU

Buildpacks

CLOUD NATIVE
COMPUTING FOUNDATION

October 2018,

Buildpacks

OPEN CONTAINER INITIATIVE

- ✓ Cross-repository blob mounting
- ✓ Image layer rebasing

Standardize Application Builds

Improve Security

Streamline Deployment Workflows

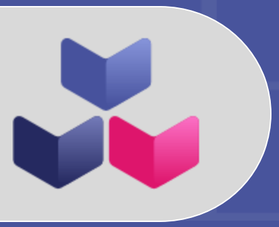
Buildpacks

Containerized Workflows

CI/CD pipelines

- ✓ Define
- ✓ Use
- ✓ Share buildpacks

Key Features of Buildpacks



Key Features

- No Dockerfile Required
- Auto-Detection
- Minimal Application Image
- Advanced Caching
- Reproducibility
- Rebasing
- Multi-Process Support
- Software Bill of Materials
- Modular Design
- Non-Root Builds

Demo

Documentation Walkthrough

Buildpacks

- Buildpacks Terminologies
- Key Terms

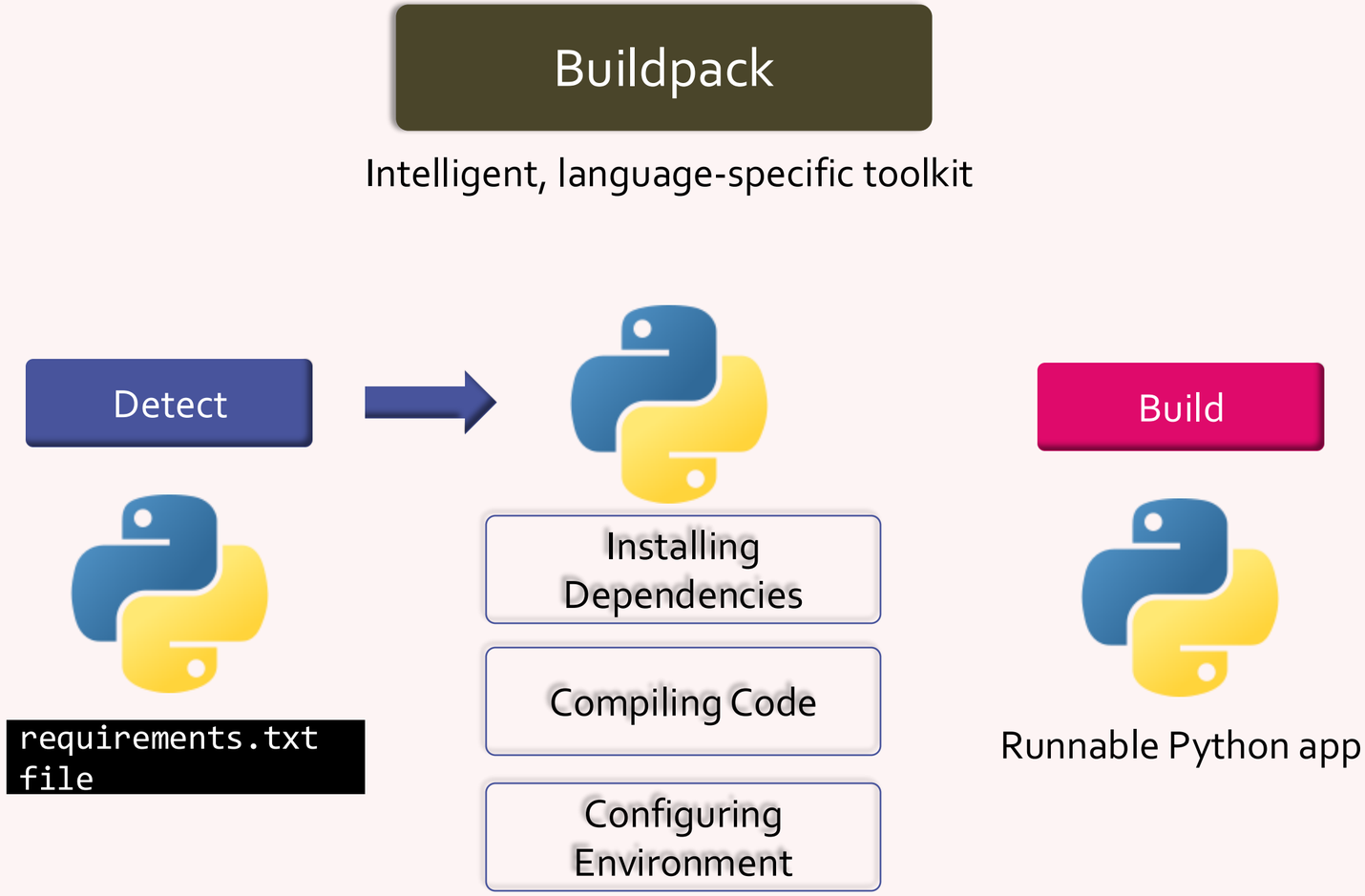
Understanding Cloud Native Buildpacks Terminology

Buildpacks Terminology

Buildpacks

Eliminating the need for developing Dokerfiles

Buildpacks Terminology



Buildpacks Terminology

Buildpack

Intelligent, language-specific toolkit

Detect

buildpack-1

buildpack-2

JVM

Serving Static Assets

Installing Dependencies

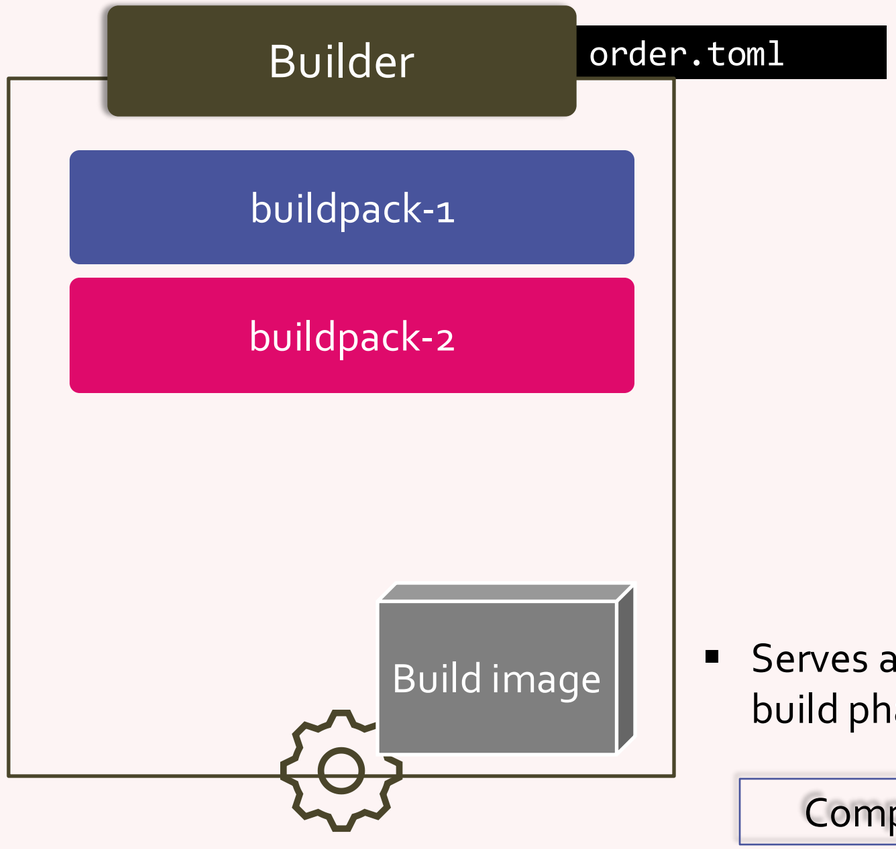
Compiling Code

Configuring Environment

Build

Runnable Python app

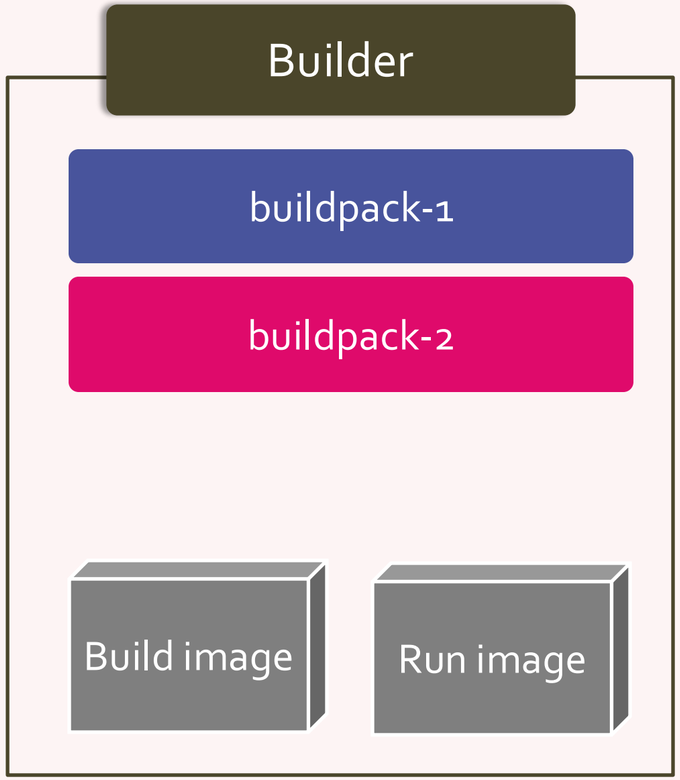
Buildpacks Terminology



- Serves as the foundation for the build phase

Compiler | Language | Package managers

Buildpacks Terminology



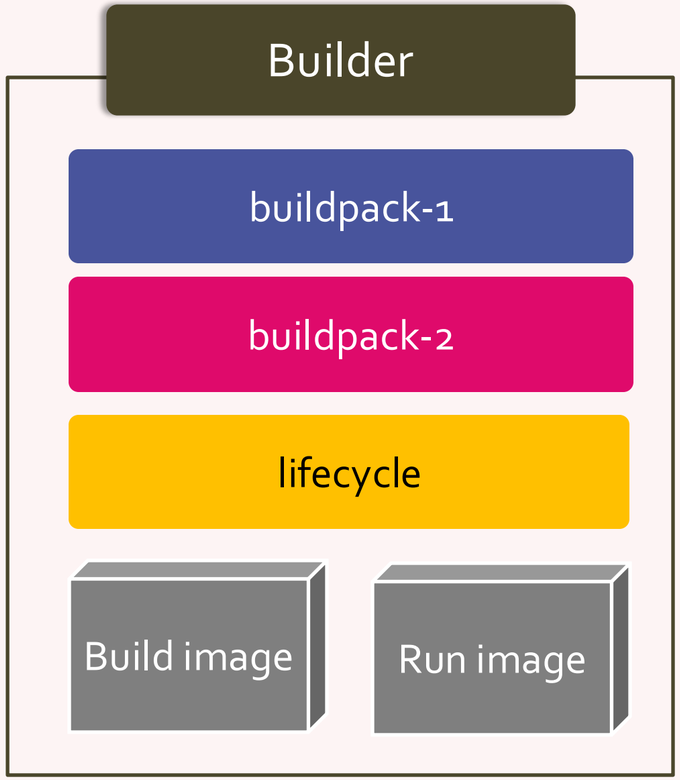
- Designed for final container image, optimized for security & performance

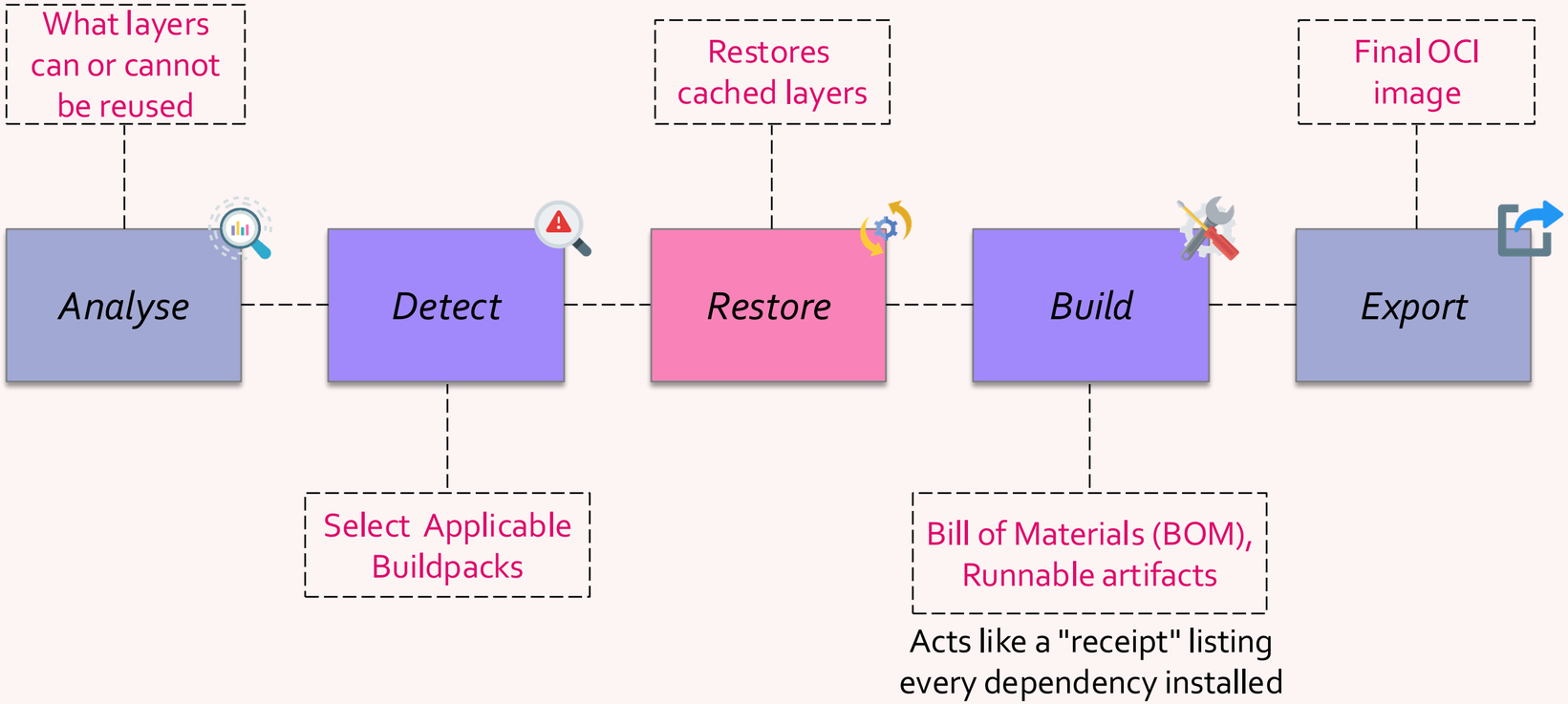
Light Weight OS

Security Patches

Necessary Runtime Libraries

Buildpacks Terminology

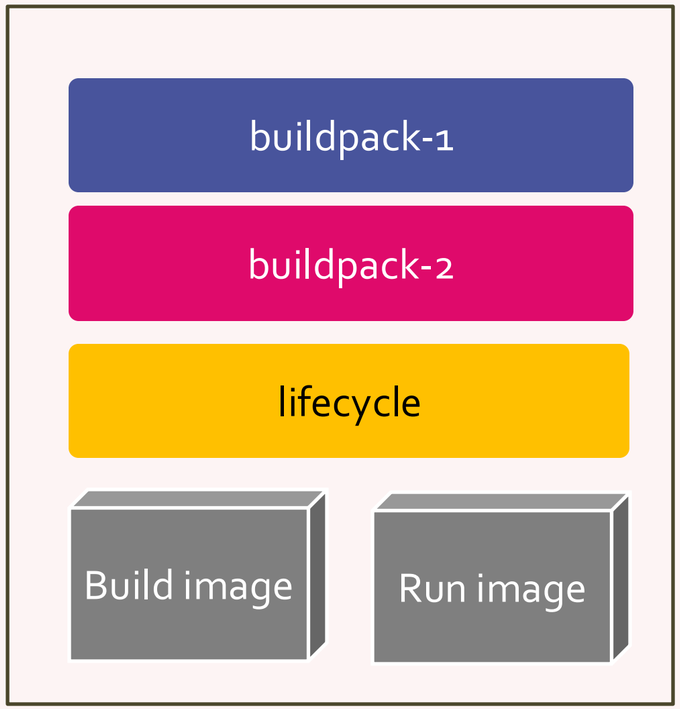


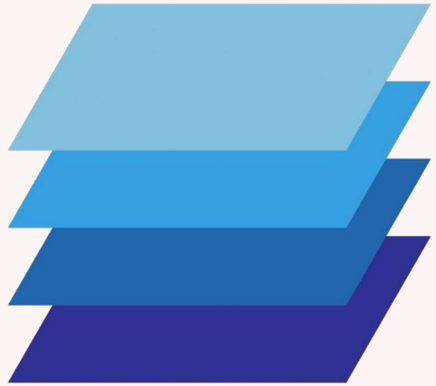


Buildpacks Terminology

Platform

```
pack build my-app --builder paketobuildpacks/builder-jammy-full
```





Final OCI image

Buildpacks Terminology

- Buildpacks enforce security best practices

Decoupled Build

Runtime Environments

- Optimize performance through layer caching
- Streamline compliance with dependency audits via the **BOM**

Dockerfile-free path

Buildpacks

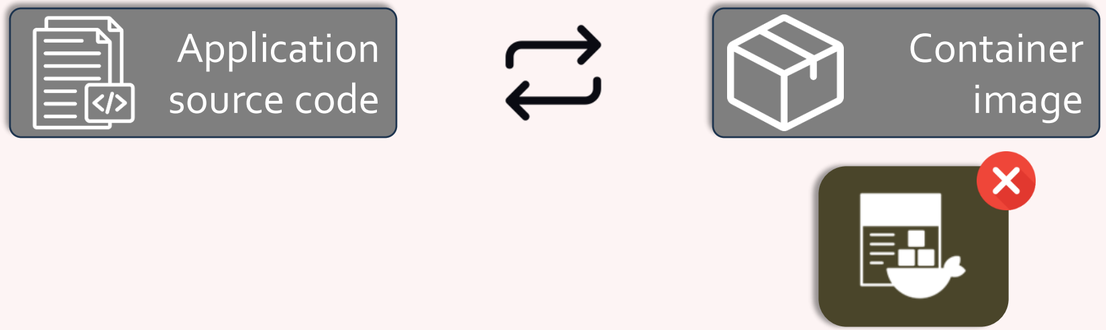
- Installing and using Pack CLI
- Installing Pack CLI on macOS, Linux, and Windows
- Commonly used Pack CLI commands

Getting Started with Pack CLI

Pack CLI

Cloud Native Buildpacks (CNB's)

✓ Pack is a CLI tool maintained by the CNB project to support the use of buildpacks



What Does pack Do?

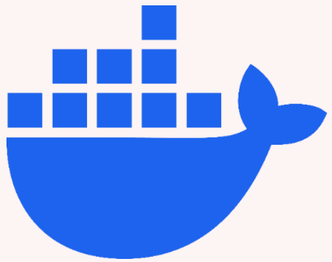
Building Images

Managing Builders

Inspecting Artifacts

Prerequisites

- To use pack, ensure you have



Docker

Podman

Demo

Installing pack CLI on Mac

Demo

Installing pack CLI on Linux

Demo

Installing pack CLI on Windows

Demo

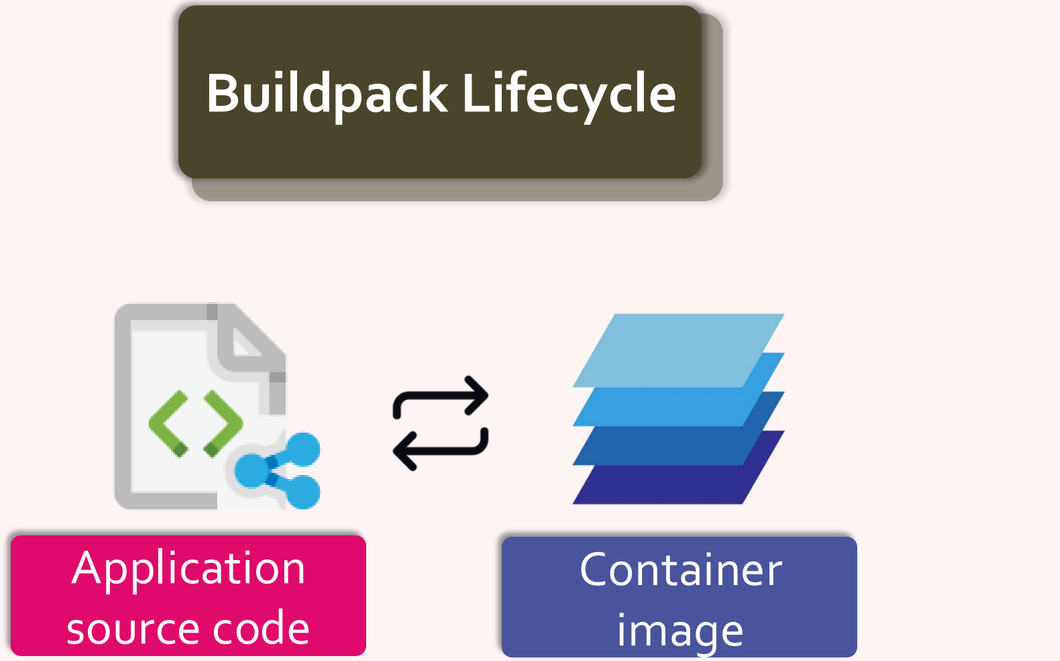
Commonly used pack commands

Buildpacks

- How to build application images
- Understanding the build lifecycle
- Building Python, Java, & Node.js application images

Understanding Build Lifecycle

Build Lifecycle



Build Lifecycle

Detection

Purpose: Identify the buildpacks

Process: Each buildpack in the builder declares its requirements via a detect script

Executes these scripts to identify compatible buildpacks for the source code

Output: ("**build plan**")

Build Lifecycle

Build Lifecycle

Analysis

Purpose: Restore cached metadata

Process: Checks for existing layers in the cache and their associated metadata.

Reduce build time and network usage

Detection

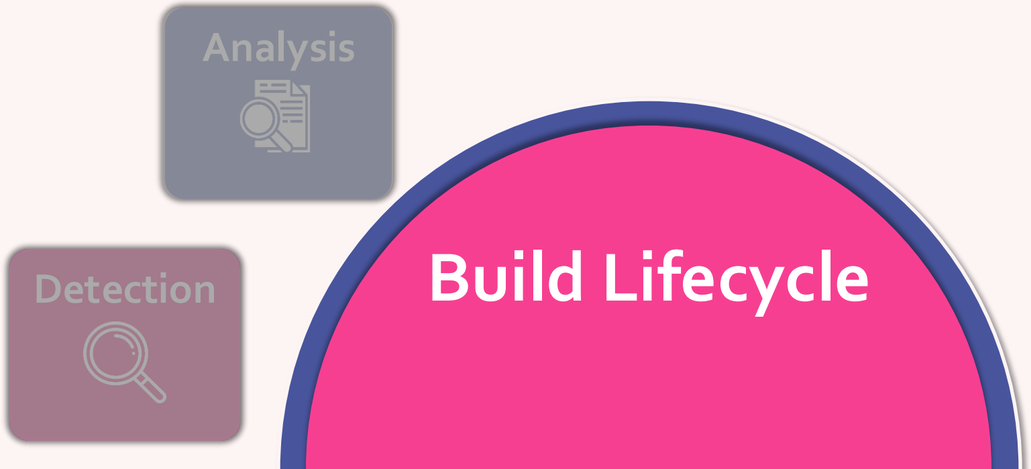
Build Lifecycle

Build Lifecycle

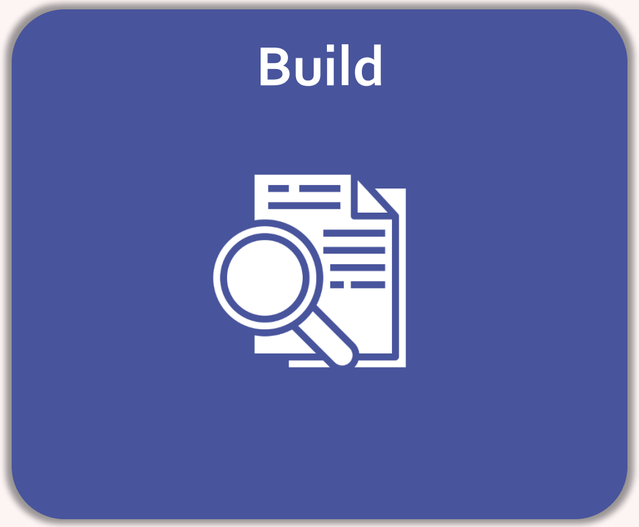
Purpose: Restore cached layers

Process: Layers marked as cacheable in previous builds are restored to the build environment

Avoids redundant downloads or recompilation of dependencies



Build Lifecycle

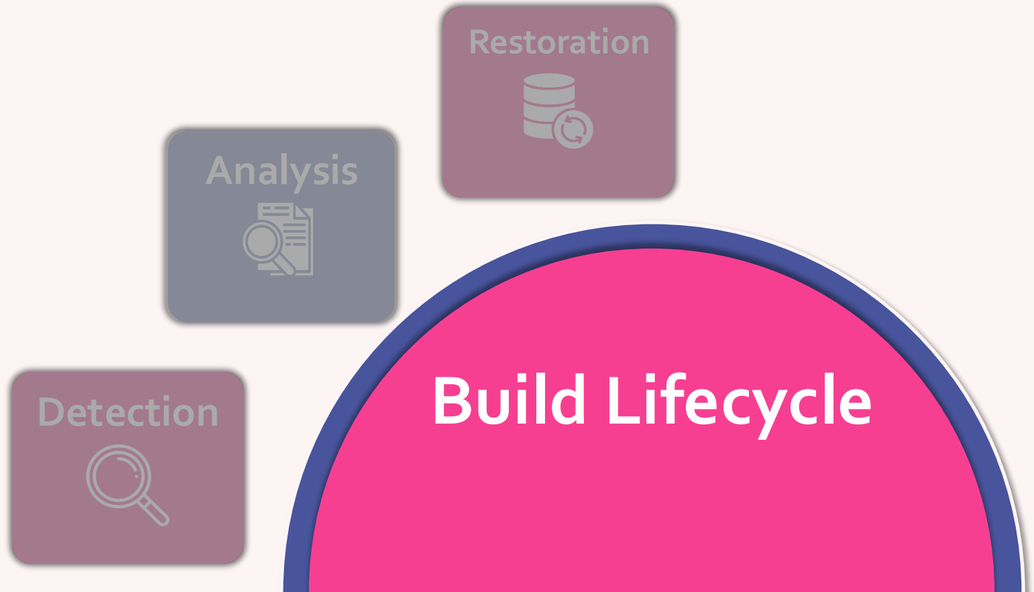


Purpose: Execute to transform source code into a runnable artifact

Process: Selected buildpacks run in order

Modify the filesystem, set environment variables, or define launch processes

Output: A "**launchable**" image with layers



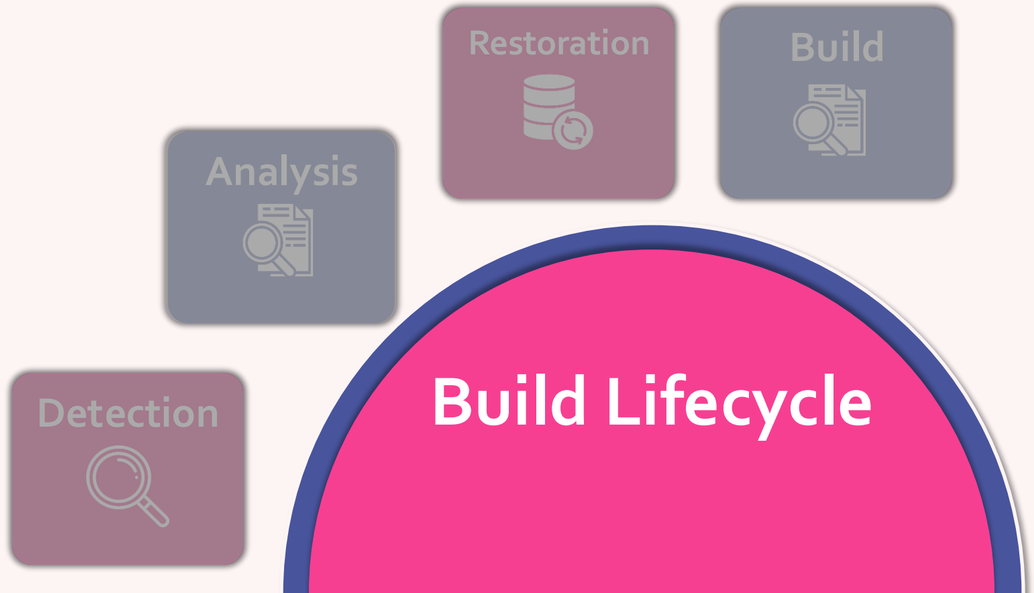
Build Lifecycle

Export

Purpose: Create the final OCI-compliant container image

Process: Combines the buildpack-generated layers, runtime environment, and metadata (**e.g.**, **launch.toml**)

Image is tagged and pushed to a registry



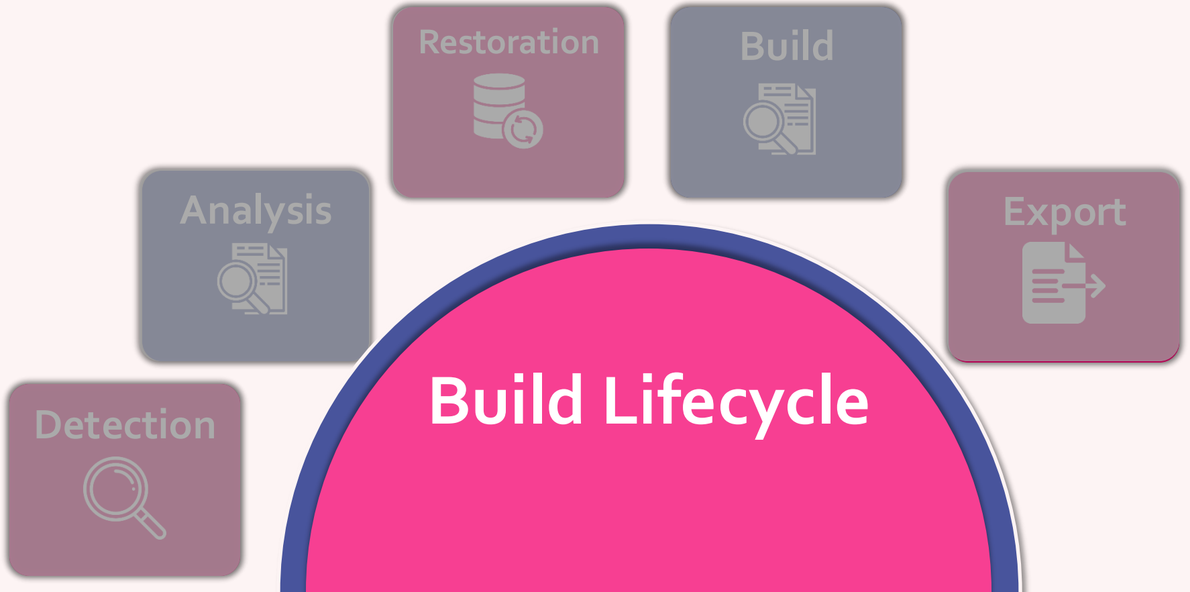
Build Lifecycle

Caching

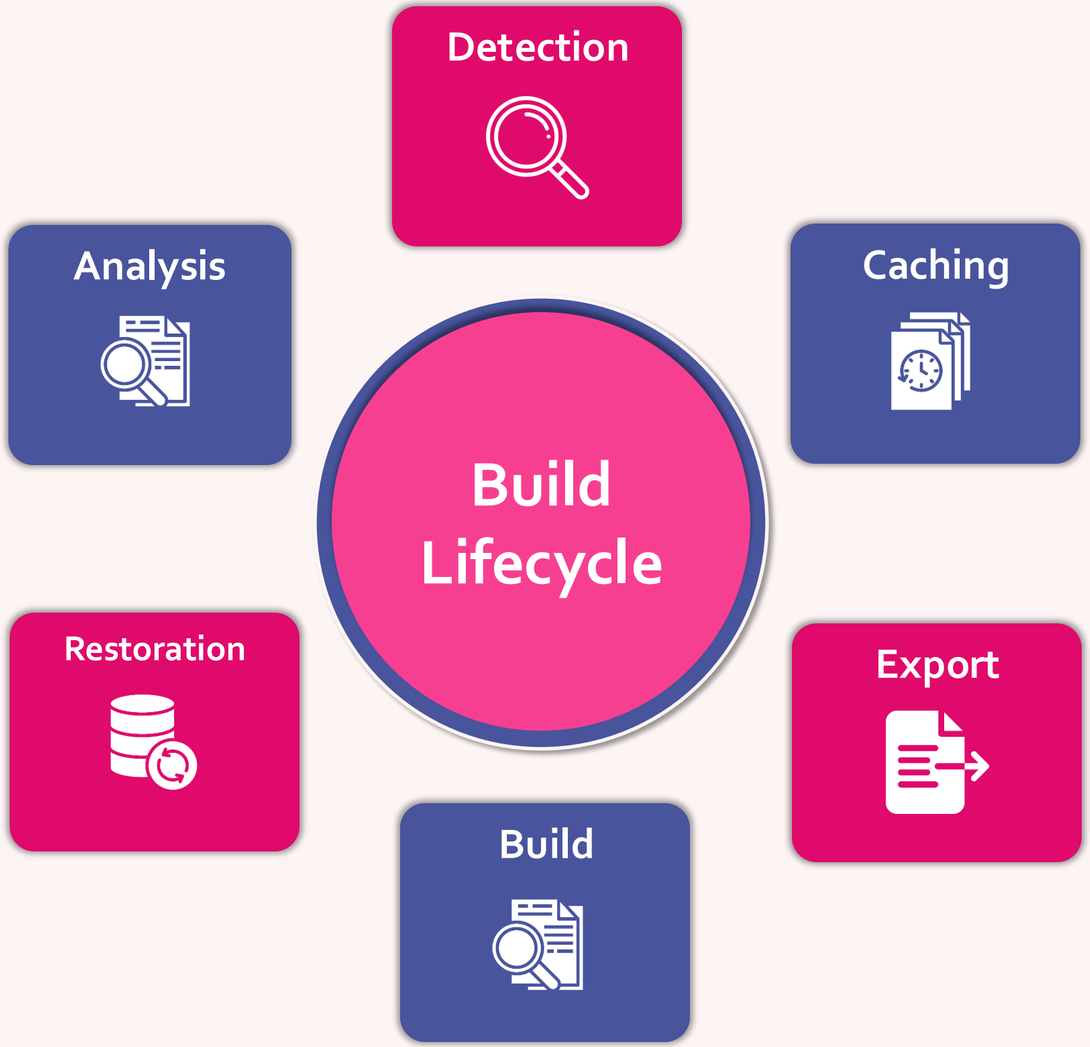
Purpose: Store reusable layers for future builds

Process: Layers marked as cacheable

Subsequent builds skip rebuilding these layers unless changes are detected



Build Lifecycle



Demo

Python Application

Demo

Python Application with Database

Demo

Nodejs Application

Demo

Java Application

Demo

Understanding - How to build your own Buildpacks and Builders

Demo

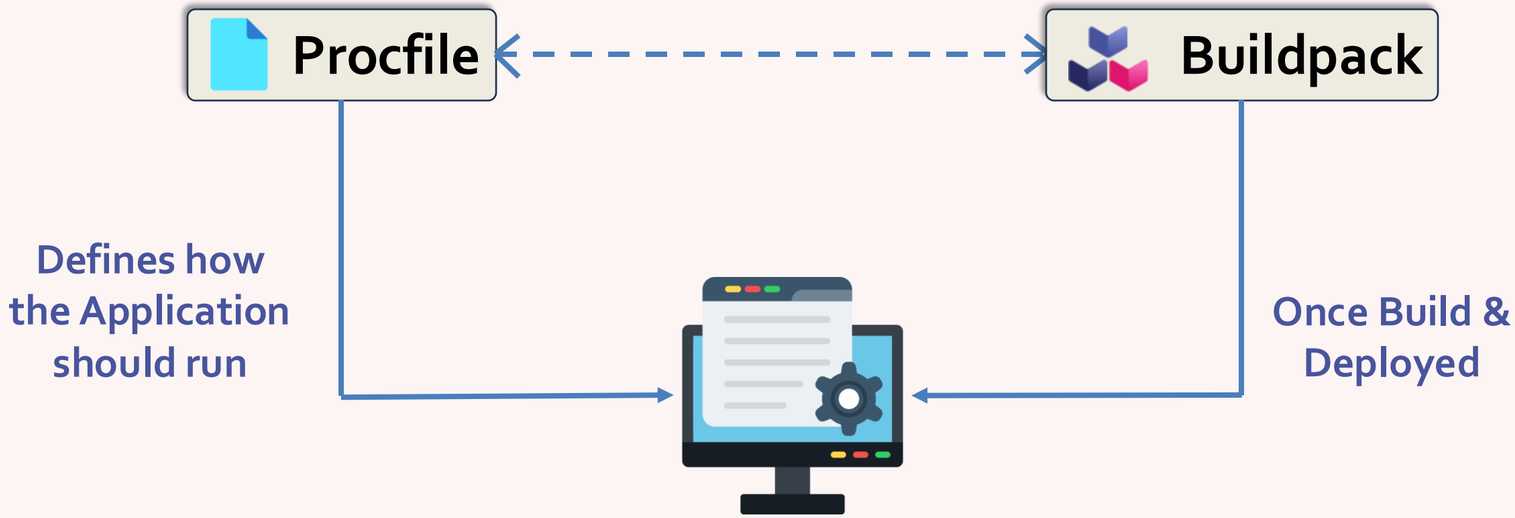
Creating your own Buildpacks

Demo

Creating your own Builders

Procfile in Cloud Native Buildpacks

Procfile

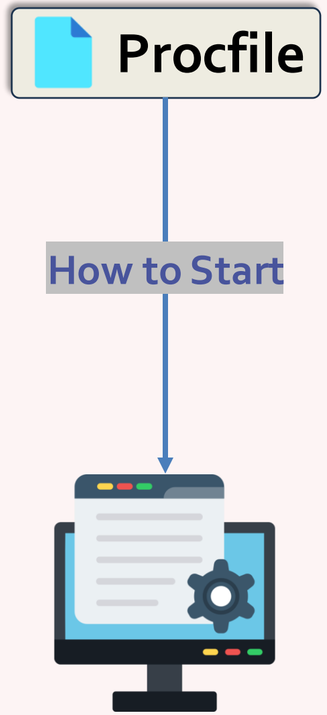


- Procfile explicitly defines the commands required to run your application once built and packaged
- Specifies processes/services
- Web applications and background tasks

Purpose

- Cloud Native Buildpacks automate building processes
  - Setting up Dependencies
  - Compiling Code
  - Generating Container Image

- Procfile defines how to start your application once it’s deployed
  - Specifies the process types



Format

- The file consists of one or more lines
- Each line defines a process type and its associated command

<process_type>: <command_to_run>

web: python app.py → Process type for running the web server

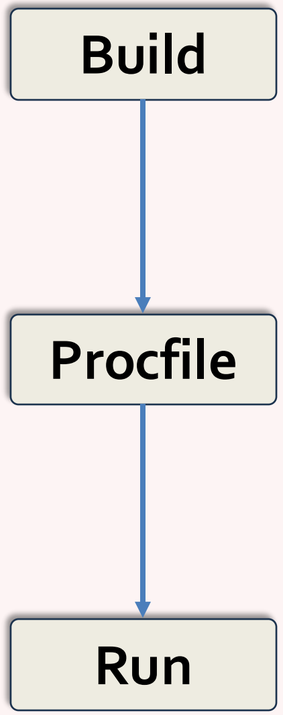
worker: python worker.py → Background process for background jobs

Benefits

- **Multi-Process Applications**
  - **Once the container is built, the Procfile defines how process should run**

- **Platform Compatibility**
  - **Cloud platforms like Heroku use Procfile to start and manage deployed apps**

Workflow



- Buildpack compiles the app, bundles dependencies, and creates a container
- After packaging, Procfile defines how to start the app and its processes
- On deployment, the platform uses Procfile to run specified processes

Summary

Cloud Native Buildpack

Building and containerizing of the app

Procfile

Defines how the app runs post-deployment on buildpack-supported platforms

Key Concepts

ClusterStore

- A cluster-wide buildpack repository for Builders.
- Populated with buildpacks from public or private registries

ClusterStack

- Defines the base OS images
- Ensures consistency

ClusterBuilder

- A cluster-scoped Builder combining ClusterStore, ClusterStack, and lifecycle

Builder Resource

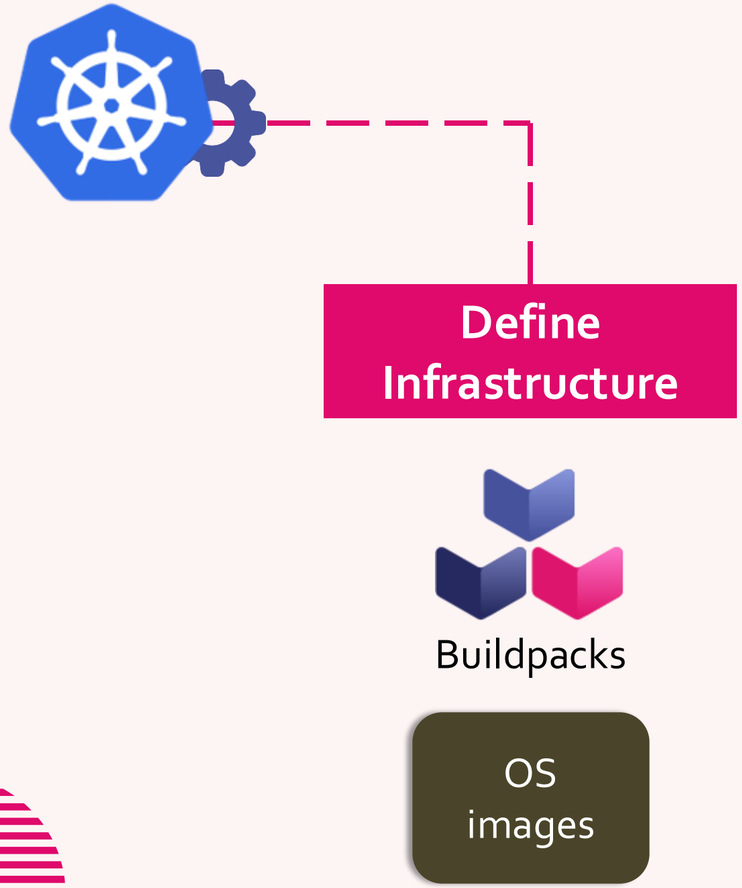
- Specifies ***how*** to build
- Reusable template for image builds

Image Resource

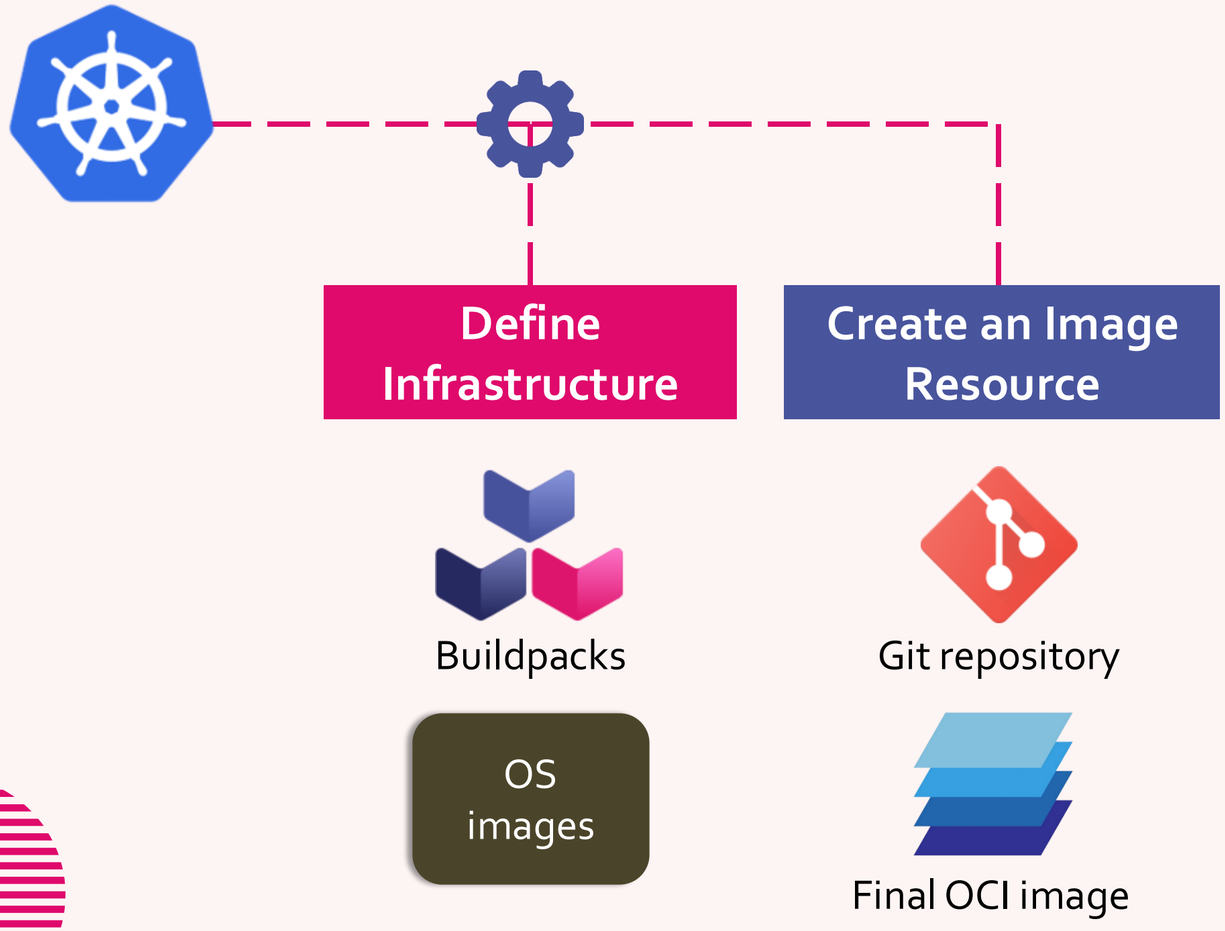
- Defines ***what*** to build
- Auto-rebuilds

Components of kpack

Workflow Overview



Workflow Overview



Workflow Overview

Define Infrastructure

Buildpacks

OS images

Create an Image Resource

Git repository

Final OCI image

Automated Builds

Monitors source code and buildpack updates

Workflow Overview

Define Infrastructure

Buildpacks

OS images

Create an Image Resource

Git repository

Final OCI image

Automated Builds

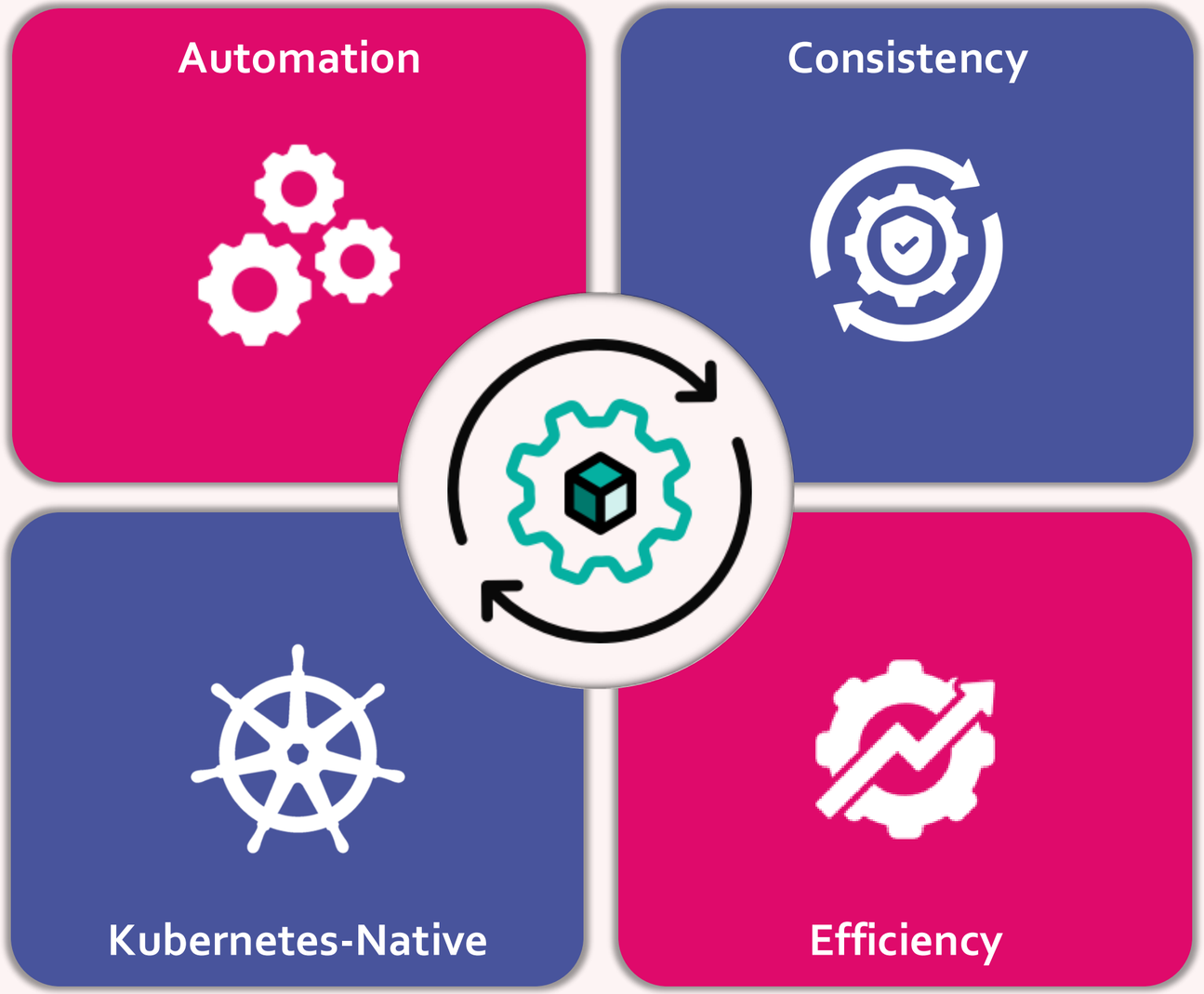
Monitors source code and buildpack updates

Image Reconciliation

Image resources

Including dependencies and security patches

Benefits of kpack



Prerequisites

Kubernetes cluster

v1.20+ recommended

Container registry

Docker Hub, Google Container Registry

Credentials

Stored as Kubernetes Secrets

Overview of the kpack Process

1. Set Up Registry Credentials
2. Create a Service Account
3. Define a ClusterStore
4. Define a ClusterStack
5. Create a Builder
6. Create an Image Resource
7. Automated Builds
8. Run and Use the Image

Demo

Environment Setup for kpack and kpack cli

Demo

Building and Publishing OCI images using kpack cli

Demo

Building and Publishing OCI images using kpack manifest files - YAML Approach

Conclusion

- Fundamental concepts, key terminology, and practical applications
- Understands container images and Buildpacks architecture
- Hands-on experience in leveraging Buildpacks to streamline your containerization workflow
- Advanced topics
  - Developing buildpacks
  - Creating builders
- kpack for Kubernetes
  - Enable automated and declarative container image
  - Builds within a Kubernetes environment