AWS Well-Architected Framework

Container Build Lens

Container Build Lens: AWS Well-Architected Framework

Table of Contents

Container Build Lens

Publication date: **October 20, 2022** (*Document revisions*)

This whitepaper describes the Container Build Lens for the AWS Well-Architected Framework. It helps customers review and improve their cloud-based architectures and better understand the business impact of their design decisions. The document describes general design principles, as well as specific best practices and implementation guidance using the six pillars of the Well-Architected Framework.

This lens whitepaper is intended for those in technology roles, such as chief technology officers (CTOs), architects, developers, and operations team members. After reading this paper, you will understand AWS best practices and the strategies to use when designing container images.

Introduction

The AWS Well-Architected Framework helps you understand the pros and cons of decisions you make while building systems on AWS. Using the framework, you will learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems in the cloud. It provides a way for you to measure your architectures against best practices and identify areas for improvement. We know that having well-architected systems greatly increases the likelihood of business success.

Two of the components that make up the Well-Architected Container Build Lens are the whitepaper and implementation guidance document. The lens whitepaper provides cloud-agnostic questions and best practices on how to build and manage containers and container images. In addition, the paper offers implementation guidance by providing examples for building and managing containers and container images in the AWS Cloud.

The Container Build Lens will focus specifically on the container design and build process. Topics such as best practices for container orchestration architecture design principles and general best practices in software development are considered out of scope for this lens. These topics are addressed in other AWS publications. See the **Resources** sections under *Pillars of the Well-Architected Framework* for more information.

For brevity, we have only covered details from the Well-Architected Framework that are specific to containerized build processes. Consider best practices and questions that have not been included in this document when designing your architecture. We recommend that you read the AWS Well-

Architected Framework whitepaper. This document is intended for those in technology roles (such as developers, architects, and engineers) who wish to understand the fundamental architectural concepts when building and managing containerized applications and container images in AWS Cloud. The lens helps technologists follow a set of established AWS well-architected best practices.

Definitions

The AWS Well-Architected Framework is based on six pillars: operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability. Building containerized applications brings a new dimension of considerations to each of these pillars. The architecture of containerized applications requires you to reconsider how to address security, operational efficiency, stability, and agility of their application stack when compared with traditional applications. In this section, we will present an overview of the container build concepts that will be used throughout this document.

Well-Architected terminology

- **Component** – Code, configuration, and compute resources that together deliver against a requirement. Components interact with other components and are often combined to create a workload.
- **Workload** – A set of components that together deliver business value. Examples of container workloads could include a set of components housed in their own distinct containers that interact with each other to form multi-tier application.

For additional definitions, refer to the AWS Well-Architected Framework whitepaper.

Container technology terminology

- **Container** – Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the host server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment. A container is a runnable instance of an image (see Deep Dive on Containers on AWS getting started resource center). While a container image is immutable, the container adds a read/write layer on top of the image to which your application can write information like temporary files or logs (see Docker overview). When the container stops and is removed, the information written to the temporary read/write layer will be lost.
- **Container image** – Container images are read-only templates used to build out containers. Container images are immutable, meaning they cannot be changed once created. Container

images are created using layers by reading a text file that is called a Dockerfile that contains all necessary information. You can find the Dockerfile reference here.

- **Container runtime** – Software running on a container host (virtual machine or bare metal server) operating system that is responsible for running and managing containers. The container relies on the kernel of the host for all system calls.
- **Dockerfile** – A file or series of files containing commands that describe the content of a container image. Each command represents a layer on the Container image (see the *Container image layers* definition).
- **Container build** – A container image built from a Dockerfile. This process results in a container image containing the necessary components to run your containerized application.
- **Base image** – A starting point container image that is used in the container build process to generate custom or new container images. This image has `FROM scratch` in the Dockerfile.
- **Parent image** – The images that are used to create your image. The parent images are referenced in the `FROM` command of your Dockerfile. It is common practice to reference a parent image rather than a base image.
- **Gold image** – A gold image is the standard image for an organization to start with. The gold image is used in different business units and defines a common baseline for cross-cutting concerns like configuration, networking, and logging. This pattern is similar to the gold Amazon Machine Image (AMI) approach for Amazon Elastic Compute Cloud (Amazon EC2) instance AMIs.
- **Container image layers** – In an image, each layer represents an instruction in the Dockerfile. Layers are applied in sequence to the base image to create the final image. When an image is updated or rebuilt, only the layers that change will be updated, and unchanged layers are cached locally. The sizes of each layer add up to equal the size of the final image (see Docker Glossary).
- **Container image scanning** – A process for security scanning for vulnerabilities in the container image.
- **Container registry** – A service used to store and distribute container images. Container images are versioned in container image repositories inside the registry. Accessing the content inside the registry is standardized in the Open Container Initiative (OCI) distribution specification. Modern registries also allow you to store related artifacts such as Helm charts or other application specification files (artifact spec support). Registries also control and limit access to content stored inside (Auth, RBAC, access logs) and partially also included vulnerability scanning features.
- **Continuous integration/Continuous delivery or deployment (CI/CD)** – CI refers to an automation strategy that makes sure new code changes to an app are built, tested, and pushed to a code repository with little to no manual intervention. Continuous delivery is a

logical extension of CI in that it makes sure that any code change is deployed to the various environments for testing on a defined schedule. Continuous deployment takes continuous delivery a step further by making sure any changes pushed to the shared repository (repo) through the CI process are automatically pushed to production, assuming the changes pass all verification tests in previous stages of the build pipeline.

- **CI/CD pipeline** – A CI/CD pipeline consists of tooling and automation to deliver a CI/CD strategy.
- **Microservice** – An architectural and organizational approach to software development to speed up deployment cycles, foster innovation and ownership, improve maintainability and scalability of software applications, and scale organizations' delivery software and services with an agile approach that helps teams work independently from each other.
- **Multi-stage build** – A multi-stage build is a mechanism to split the build-phase of the image from the final image that will be used to run the application.

Design principles

The Well-Architected Framework identifies a set of general design principles to facilitate good design in the cloud for building and managing container images:

- **Minimize image size:** Reducing the container image size has many advantages, such as reducing the attack surface, improving scaling speed as containers start faster, and minimizing the amount of storage. One recommended way to achieve this is to reduce the number of dependencies and use multi-stage builds. In order to minimize the attack surface, only the necessary binaries to run the container should be included in the image. The same applies for data. Data should be stored in an external system and it is considered best practice to inject configuration data.
- **Build in multiple stages:** Builds of an application should be reproducible everywhere and all dependencies should be encapsulated. This can be achieved by using multi-stage builds. A multi-stage build is a mechanism to split the build-phase of the image from the final image that will be used to run the application. With this mechanism, you can split the build-phase from the final image that will be used to run the application.
- **Standardize across the business:** Based on a lean parent image, a team- or enterprise-wide image can be created that provides optimizations to all teams. An organization could potentially start with a lean image containing company-specific configurations, and teams start adding additional software that is necessary to run the different applications. The advantage of this approach is that it can help enforce security, quality, and compliance standards across multiple teams for the whole organization.
- **Validate and restrict versions:** Image tagging can be used to add additional meta-information to container images. This is useful for codifying version or stability information. For cost management purposes, it is recommended to add additional information like cost center or department to the tag name. The automatically created latest tag shouldn’t be used. A tagging strategy should produce immutable tags. Teams should avoid overwrite tags because this makes it hard to reproduce issues of specific application versions.
- **Implement an image scanning strategy:** It is important to scan container images regularly after they have been built to make sure no new or existing vulnerabilities have surfaced. There are two basic categories to consider when discussing image scanning - static scanning and dynamic scanning. After pushing code to a git repository, a CI/CD chain should automatically build the artifact, the container image, and scan it on push.

- **Containers are designed to perform stateless actions.** If your application is required to store its operational state, then store the state itself in a data store, and refer to it from the application running in your container. Same with source of truth data stores. The source of truth data should be stored outside of the container and your application should ingest only the data that is required to perform its desired actions.

Pillars of the Well-Architected Framework

This section describes the AWS Well-Architected Container Build Lens in the context of the six pillars of the Well-Architected Framework. Each pillar describes a set of definitions, key topics, and considerations that are relevant when building and designing containerized applications in the cloud. These include design principles, definitions, best practices, evaluation questions, considerations, key services, and useful links.

Pillars

- Operational excellence pillar
- Security pillar
- Reliability pillar
- Performance efficiency pillar
- Cost optimization pillar
- Sustainability pillar

For brevity, we have only included questions that are specific to building container workloads. Questions that have not been included in this document should still be considered when designing your architecture. We recommend that you read the AWS Well-Architected Framework whitepaper.

Operational excellence pillar

The operational excellence pillar includes the ability to run and monitor systems to deliver business value and to improve supporting processes and procedures.

The Well-Architected Framework offers advice on the development of processes, runbooks, and game days to test your workloads. We recommend that you investigate the following topics to drive operational excellence within containerized workloads.

These include using a base image that contains only the necessary packages needed for your organization's containers, parity between deployment environments, and the features you should build into your containers and what can be hosted outside of your container.

Best practices

Operational excellence is defined by the ability to run and monitor your systems to ensure they are delivering the business value as intended. Further, it can continuously improve support processes and procedures throughout the lifetime of the system. Operational excellence with regard to your containers and container images is no different. As your application evolves and matures, you must update your container images and running containers to provide additional features to your customers. With these updates and new container deployments, you must have the right mechanisms in place to validate that your workload is attaining its intended business goals. These mechanisms include container health checks and the collection of container logs external to the container. The following guidance is in addition to the best practice guidance laid out by the Operational Excellence Pillar whitepaper.

To drive operational excellence in your containerized workloads, you must understand your container images. This includes the capabilities that are built into the container image, the container image build process, and deployment behavior expectations. You will then be able to design your images to provide insight into your container status and build procedures in order to build containers that best fit your workload.

The following are best practice areas for operational excellence in the cloud:

- Organization
- Prepare
- Operate
- Evolve

Organization

There are no operational excellence best practices for organization specific to the container build process.

Prepare

CONTAINER_BUILD_OPS_01: How do you manage the lifecycle of your containers and images?

Understand the lineage of your container image

It is important to understand what is built into your container image. When you start building a new project, at times it's easier to use a base image from a verified source, such as an official Ubuntu image. While these help developers get up and running faster, often images contain packages and libraries that are not required to run your application and take up additional space. Starting with a minimal container image will save space and speed up the starting of your container when it's deployed into production.

Many customers that we speak with take the additional step of creating parent images. These images form the base for what all containers in the organization are built on top of. These parent images are minimal images that put into place requirements and security controls established by the organization. For example, the parent image can configure an internal package source repository that contains curated and validated library package versions.

Understanding the lineage of your container image helps you efficiently develop, run, manage, and maintain your containers. It also helps maintain your security posture. You can find more details in the Security Pillar whitepaper.

Have parity between your deployment environments

A major benefit of using containers is to provide the ability for the development team to develop new updates and features using an identical artifact that runs in production. As much as possible, development, testing, QA, and production environments in that it will be eventually deployed should be as similar as possible. All environments should share best practices for everything, with the differences between them being the ability to scale and the data operated upon. Best practices for development environments differ between orchestration tools so make sure you are following recommendations based on your containerized platform of choice.

Build the image once and use the same image in all environments

Once the new image has been built with the updates in place for deployment, promote the same image into the next environment, testing, QA, and production, to provide for consistency across all environments. This will reduce the number of changes introduced in each new environment and provide for more consistent behavior.

Use a CI/CD build process

Like with your applications, you should use a CI/CD pipeline to build and test your images through every stage in your development process. The CI process usually starts upon a trigger that is sent

from a version control system (usually git). Whether your application requires compilation or not, there are several steps to take to build the container.

- Check the code out from the code repository.
- Build the application artifact (executable binary or language-specific archive).
- Build the container image from the artifact that you just created.
- Run tests against the built container image.
- Push the container image to the target container registry.

Once built, you will begin continuous deployment where an automated deployment pipeline takes the recently built container image, its manifests, and container package (Helm chart, Kustomization overlay, and so on), and deploys it to the target environment.

Multi-stage builds

Small container images have undeniable performance advantages. The pull from the registry is faster because less data is transferred over the network, and the container startup time is also reduced. This can be achieved by using multi-stage builds. With this mechanism, you can split the build-phase of the image from the final image that will be used to run the application.

```
FROM debian:10-slim AS builder
# First step: build java runtime module
...

ENV M2_HOME=/opt/maven
ENV MAVEN_HOME=/opt/maven
ENV PATH=${M2_HOME}/bin:${PATH}

COPY ./pom.xml ./pom.xml
COPY src ./src/

ENV MAVEN_OPTS='-Xmx6g'

RUN mvn clean package


# Second step: generate container run image
FROM debian:10-slim

...
```

```
COPY --from=builder target/application.jar /opt/app/application.jar

ENV JAVA_HOME=/opt/java
ENV PATH="$PATH:$JAVA_HOME/bin"

EXPOSE 8080

CMD ["java", "-server", "-jar", "application.jar"]
```

In this example, we start building a container image with the build stage **builder** that will be referenced in the target image. In the builder-stage, we generate the complete build environment with all the dependencies (in this case JDK and Maven) and build the JAR-file containing the application. In the second step of the build process, there is an additional `FROM` statement that indicates we start a new build stage. The `COPY` command references the former builder-stage and copies the JAR-file into the current stage of the build. With this approach, it is possible to implement reproducible builds with all necessary dependencies and lightweight target images. However, this build process is more complicated compared to standard builds and usually takes longer.

Implement a minimal container image design to achieve your business and security objectives

It is important to build into your container image only what is necessary. Pictures and other static assets should be stored in a data store, for example Amazon Simple Storage Service (Amazon S3) in AWS, and served through a content delivery network (CDN). This will achieve a minimal container image size, which does not only reduce the storage and running host resource requirements, but it also speeds up the instantiation of a new container from an image if the image itself is not cached locally.

Using package managers to deploy your containerized applications

When building a containerized application, the deployable unit can be not only the container image, but its per-environment configuration that is deployed alongside with the container image to the target environment. To achieve this, users can use packaging tooling such as Helm and Kustomize for Kubernetes, AWS Copilot for Amazon Elastic Container Service (Amazon ECS), Docker Swarm for Docker, and more. That means when building your containerized application, the target artifact will be a package that contains a reference to the container image and its common configurations across all environments. Such configurations can be environment variable, flags, ports, and other specific configurations. This package will then be deployed to different target environments with per-environment customizations. An example of this can be a Helm chart with

an application that references a database endpoint. The Helm chart will contain all the common configurations for development, testing, and production environment, leaving some values to be configured per-environment such as the database endpoint. Then there will be different files such as `values-dev.yaml` and a `values-prod.yaml` that will contain different database endpoints for the development and production environments.

Operate

CONTAINER_BUILD_OPS_02: How do you know whether your containerized workload is achieving its business goals?

An important part of operating any workload is understanding the health of your workload quickly. The sooner an issue can be identified the better.

To ensure the success of your running container, you must understand the health of your containerized workload as well as what your customers are experiencing when they interact with your application. Identify sources of information within your workload and use the information to understand the health of the containerized workload. Identifying where the telemetry within your workload will come from, determining the proper logging levels, identifying the thresholds and information from that telemetry that must be acted upon, and identifying the action that is required when those thresholds are passed will reduce risk and impact on your users.

Implement health checks to determine container state

Implement container health checks. Health checks are one way to determine the health of your running container. They enable your orchestration tooling to direct connection traffic to the container only when it is ready to accept connections, or stop routing connections to the container if the health checks show that the container is no longer running as expected. In the latter case, the orchestration tooling will tear down the misbehaving container and replace it with a new healthy one.

For example, with Amazon ECS you can define health checks as part of the task definition, and perform load balancer health checks for your running application.

For Kubernetes and Amazon Elastic Kubernetes Service (Amazon EKS), you can take advantage of features such as liveness probes to detect deadlock condition, readiness probes to determine if the pod is prepared to receive requests, and startup probes to know when the application running in

the container has started. Liveness probes can either be shell commands or HTTP requests of TCP probes.

Have your logs available outside the running container

Ensure that the logs generated by your running containers are collected and accessible externally. This will enable you to use log monitors to gain more insights into the behavior and functionality of your running container. Your application should be writing its logs to `STDOUT` and `STDERR` so that a logging agent can ship the logs to your log monitoring system.

As with other application workloads, you must understand the metrics and messages that you have collected from your workload. Not only must you understand the data emitted by your containers, but you must also have a standardized log format to easily evaluate the data with your logging tools. Logging collector and forwarder tools give you the ability to standardize your log format across multiple containerized services.

To enable you and your team to pinpoint where issues may be occurring, define your log messages to be consistently structured to enable correlation of logs across multiple microservices in your central logging system.

This understanding enables you to evolve your workload to respond to external pressures such as increased traffic or issues with external APIs that your workload may use in the course of accomplishing its business goals.

Evolve

There are no operational excellence best practices for evolve specific to the container build process.

Resources

This section provides companion material for the Container Build Lens with respect to the operational excellence pillar.

Blogs and documentation

- Health checks and self-healing with EKS
- Implementing health checks (400 level general guidance)
- How can I get my Amazon ECS tasks running using the Amazon EC2 launch type to pass the Application Load Balancer health check in Amazon ECS?

- How do I troubleshoot health check failures for Amazon ECS tasks on Fargate?
- EKS Windows container logging best practices
- Capturing logs at scale with Fluent Bit and Amazon EKS
- Monitor, troubleshoot, and optimize your containerized applications infographic

Partner solutions

- Docker container logging
- Fluent Bit documentation

Whitepapers

Running Containerized Microservices on AWS

Training materials

- One observability workshop
- Kubernetes health checks
- Logging with Amazon OpenSearch Service, Fluent Bit, and OpenSearch dashboards
- Monitoring using Prometheus and Grafana
- Tracing with AWS X-Ray
- Monitoring using Amazon Managed Service for Prometheus / Grafana
- Monitoring Amazon ECS clusters and containers

Security pillar

The security pillar includes the ability to help protect information, systems, and assets while delivering business value through risk assessments and mitigation strategies.

Best practices

One of the greatest benefits of building applications using container images is confidence that the application can run on any sort of infrastructure. Whether the container is deployed on a developer's personal laptop or in public cloud infrastructure, the container will run as expected,

because all the required dependencies are packaged within the image. With this in mind, alignment with security standards and best practices is of utmost importance as containers are deployed in different types of environments.

The best practices laid out in this section of the whitepaper are designed to help you address vulnerabilities that may be introduced during the design and build of a container image.

This section defines how the security pillar relates to Container Build Lens specifically.

The following are best practice areas for security in the cloud:

- Identity and access management
- Detective controls
- Infrastructure protection
- Data protection
- Incident response

Identity and access management

CONTAINER_BUILD_SEC_01: How do you ensure that your container images are using least privilege identity?

By default, containers provide process isolation. This means that processes running inside of a container are isolated from processes and data that exist in other containers as well as the container host's operating system. However, it is important to note that the default behavior is to run the container using the root user when running a container. When the processes inside the container are running as the root user, not only do they have full administrative access to containers, they also have the same administrative level access to the container host. Having an application running within a container through the root user expands the attack surface of the environment. This could provide bad actors with the ability to escalate privilege to the container host infrastructure if the application is compromised. There are multiple ways to mitigate this risk. The most straightforward method is to define the USER directive in the Dockerfile used to compile the image:

```
FROM amazonlinux:2
RUN yum update -y && yum install -y python python-pip wget
```

```
RUN groupadd -r dev && useradd -r -g dev dev
USER dev
...
```

The Dockerfile referenced previously uses a RUN command to add the dev user and group to the image and uses the USER directive to ensure that the dev user is used when running commands inside of the container. Therefore, even if the application hosted in this container is compromised, the attackers would not be able to use the dev user within the container to access other containers or the container host's operating system.

CONTAINER_BUILD_SEC_02: How do you control access to your build infrastructure?

Limit administrator access to build infrastructure (CI pipeline)

Adding continuous security validation in a build pipeline is a major focus for organizations moving to a DevSecOps strategy. This helps ensure that security is built into the application from the beginning of the application's lifecycle as opposed to performing security testing only at the end of the development process. However, it is important to note that securing an organization's build pipeline should be considered a high priority as well, as the pipeline typically accesses databases, proprietary code, and secrets or credentials across dev, test, and prod environments. A compromised build pipeline could provide a bad actor with access to all of the preceding resources in a customer environment. As detailed in the security pillar of the AWS Well-Architected Framework, it is important to follow the best practice of granting the least privileged access to the container build infrastructure. The least privileged best practice should be applied to human identities as well as machine identities. An example might be that a human identity that has access to the container build infrastructure can reach an application's source code, secrets, and other sensitive data.

Detective controls

CONTAINER_BUILD_SEC_03: How do you detect and address vulnerabilities within your container image?

Ensure that your images are scanned for vulnerabilities

After images are built, it is important to maintain a regular cadence of scanning those images to ensure no new or existing vulnerabilities have surfaced. There are two basic categories to consider when discussing image scanning: static scanning and dynamic scanning. Static scanning is performed before the image is deployed. This is important because it allows organizations to detect vulnerabilities in a container image before a container is deployed into an environment. Many registry offerings provide native static image scanning that can scan container images for common vulnerabilities and exposures (CVEs) without having to integrate and maintain a third-party image scanning tool. The scanning process is performed by comparing parent container images, dependencies, and libraries that are used by the container image to known CVEs. Dynamic container scanning is a process that scans the underlying infrastructure where containers run. It is executed post container deployment, and identifies vulnerabilities that may have been introduced by other software installed on the infrastructure itself. These vulnerabilities can be either modifications to an existing running container, or communication with a container that is exposed externally to other processes or hosts.

Infrastructure protection

CONTAINER_BUILD_SEC_04: How do you manage your container image boundaries?

Minimize attack surface

In the context of container workloads, infrastructure protection is often a topic with respect to the container as a vector to access the underlying compute infrastructure. In any security context, reducing attack surface is top of mind. This can be accomplished when designing and building your container in a variety of ways:

- Run distroless images without a shell or a package manager to ensure that bad actors cannot make changes to the image or easily download software packages to aid in their attack.
- Build open-source libraries from source or scan libraries for vulnerabilities to ensure awareness of all of the components of the container image.
- Remove or defang `setuid` and `setgid` bits from the container image to make sure that these permissions are not used in privilege escalation attacks.
- Lint your Dockerfile to help identify violations of best practices for building container images.
- Use a tool such as docker-slim to analyze existing images and remove unnecessary binaries not required by the application.

- Ensure that your container is designed to operate with a read-only root filesystem. This functionality is normally defined at runtime but it is important to consider this facet when designing the container itself.

Understand the lineage of your container image

Aside from reducing attack surface, it is also important to understand where your container images are coming from. If not building images from scratch, you should only run images from trusted registries that have been signed with a trusted signature to ensure integrity. Regarding signing images, it is recommended to utilize signed images to ensure that the contents of the container have not been modified before they are deployed. In general, don't incorporate images directly from a public repository into your container pipeline. Private registries should be used to allow an organization to maintain complete control and visibility over their container image catalog. If using images originating from public repositories, they should be scanned, signed, and stored in a private registry to ensure that the contents of the image are known and verified against existing security standards.

Data protection

CONTAINER_BUILD_SEC_05: How do you handle data within your containerized applications?

Do not hardcode sensitive data into your container image

With respect to handling data in the build and design of the container, it is important that no sensitive information is stored in the container itself. For example, user credentials should never be hardcoded into your container image. Instead, consider using a secret management protocol that is compatible with the container orchestration system being used to manage the container workloads.

Ensure that persistent data is stored outside of the container

Also, if your containerized application writes or consumes persistent data, ensure that data is stored outside of the container. Since containers are intended to be ephemeral, use volumes to store persistent data that will remain intact long after a container's lifecycle has completed.

Incident response

There are no security best practices for incident response specific to the container build process.

Resources

This section provides companion material for the Container Build Lens with respect to the security pillar.

Blogs and documentation

- Container monitoring - Why, how, and what to look out for
- Amazon ECR container image scanning
- Scanning Amazon ECR container images with Amazon Inspector
- Container scanning updates in Amazon ECR private registries using Amazon Inspector
- Building end-to-end DevSecOps CI/CD pipeline
- Logging image scan findings from Amazon ECR in CloudWatch using AWS Lambda function
- Compliance as code for Amazon ECS using Open Policy Agent, Amazon EventBridge, and AWS Lambda
- AWS Secrets Manager controller POC: an EKS operator for automatic rotation of secrets

Partner solutions

- Content trust in Docker
- Notary project - Signature of an OCI artifact
- Cosign - Container signing, verification, and storage in an OCI registry

Whitepapers

- EKS best practices (image security)

Training materials

- AWS container DevSecOps workshop

Reliability pillar

The reliability pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. This includes the ability to operate and test

the workload through its total lifecycle. This paper provides in-depth, best practice guidance for implementing reliable workloads on AWS.

To achieve reliability, a system must have a well-planned foundation and monitoring in place, with mechanisms for handling changes in demand or requirements. The system should be designed to detect failure and automatically heal itself.

Best practices

The reliability pillar encompasses the ability of a workload to perform its intended function correctly and consistently when it's expected to. This includes the ability to operate and test the workload throughout its total lifecycle. In addition to the best practices described in the AWS Well-Architected Reliability Pillar whitepaper, this lens introduces a set of techniques and configurations for the containerized build process, so that you'll be able to reliably operate and manage your containerized applications.

The following are best practice areas for reliability in the cloud:

- Foundations
- Workload architecture
- Change management
- Failure management

Foundations

CONTAINER_BUILD_REL_01: How do you limit the amount of CPU and memory a container consumes?

Use RAM and CPU limits

By default, a running container will use the full RAM and CPU of the host system. This can lead to performance bottlenecks on the host and put your workload in a degraded state.

Setting RAM and CPU limits on your running container will improve the availability of the host system and the workload. In Amazon ECS, update the CPU and memory parameters in the task definition to limit the CPU and RAM a container will consume.

```
         {
  "containerDefinitions": [
      {
        "command": ["/bin/sh -c 'echo HELLO WORLD! >> /usr/share/nginx/html/
index.html'"],
        "entrypoint": ["sh", "-c"],
        "image": "nginx:1.20.1-alpine",
        "name": "hello-world",
        "portMapping": [
            {
              "containerPort": 80,
              "hostPort": 80,
              "protocol": "tcp"
            }
        ]
      }
  ],
  "CPU": "256",
  "memory": "512",
  "executionRoleArn": "arn:aws:iam::012345678910:role/ecsTaskExecutionRole",
  "family": "fargate-task-definition",
  "networkMode": "awsvpc",
  "runtimePlatform": {"operatingSystemFamily": "LINUX" },
  "requiresCompatibilities": [ "FARGATE" ]
  }
```

If you are going to run your container workload on Amazon EKS, update the CPU and memory values in the resources section of your YAML file. The requests and limits keys are used to define how much memory and CPU a specific container will consume when running.

```
        ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: web
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: app
        ports:
        - containerPort: 80
        image: nginx:1.20.1-alpine
        resources:
          requests:
            memory: "64Mi"
            CPU: "250m"
          limits:
            memory: "128Mi"
            CPU: "500m"
```

Workload architecture

CONTAINER_BUILD_REL_02: How do you handle persistent data in a container application?

Use volumes to persist data

There are times when workloads have to store data across multiple containers. For example, an image-processing application that saves images for processing. Given the ephemeral nature of a container workload, data on the container will be lost once the container is restarted and longer exists. Use mounted volumes, whether block or network file system (NFS), to persist file data for an application. Mounted volumes allow for file data sharing among multiple running containers. In addition, mounted volumes should be used to persist logs or configuration files. For persisting data, use external database such as Amazon Relational Database Service (Amazon RDS), Amazon DynamoDB, or Amazon Aurora. Use a database system that provides performance, high-availability, and scalability to your container application when persisting data.

CONTAINER_BUILD_REL_03: How do you automate building and testing of containers?

Create local testing processes

When building a containerized application, you want to be able to test your application as early as possible. That means that you have to think about how developers will be able to test their containerized application locally. First you will have to decide whether the container build for local testing will run on the developer's machine or in a remote machine, because this will have an impact on the tooling that developers use on their machines. Second, you will have to provide a local deployment mechanism. For this, you can use single containers that run as part of an automation script or deploy the containers locally using a local version of your target orchestrator. This can be also part of the testing section of your local build-script. With this approach, you can deploy necessary infrastructure components like databases in a lightweight fashion in order to test your application with the real infrastructure instead of mocked APIs. One example might be a Docker Compose manifest to deploy multiple containers in a single command. For Kubernetes, use minikube to deploy the containerized application and all of its objects (such as Deployment, ConfigMaps, and Secrets).

Design your testing environments to support your container build pipeline

When building a containerized application, it can be easily deployed throughout multiple environments. In order to validate that your application is running properly, you will have to test your containerized applications. With the container's ecosystem, you can have multiple manifests for all of the applications in an environment, and you can easily provision a ready-to-use environment with all dependent services already deployed in it. This process of temporary, or ephemeral testing environments, can be achieved in lower effort given the ease of reproducing fully configured environments that are based on containers. Whether you're using the GitOps methodology for a Kubernetes based application, or a centralized deployment configuration, you should try to create reproducible environments to support testing of your containerized application.

Change management

CONTAINER_BUILD_REL_04: How do I cascade updates to a parent or base image?

Create a standardized parent image

Based on a lean parent image, a team- or enterprise-wide image can be created that provides optimizations to all teams. This could also be multiple parent images depending on the containerized application frameworks and languages. An organization could potentially start with a lean image containing company-specific configurations, and teams can add additional software that is necessary to run the different applications. This could be, for example, a Java Runtime Edition (JRE) or a specific Python version. One disadvantage of this solution is that if a parent image is changed, all images that use it - directly or indirectly - must also be recreated.

Use an image hierarchy approach

Try to maintain an image hierarchy in your container image strategy. A hierarchy or layered approach to container images helps with maintenance, cascading of updates to base images, and allows for the reuse of container images. In addition, it helps maintain the security posture of the broader organization by using the same images that have the security controls image managed by a central team. Operations like patching of a parent image should trigger a rebuild with changes to child images. As a best practice, separate images into the following categories:

- Intermediate base image
- Application server
- Application source code or binary

The intermediate base image is a small and lightweight base OS image. See the following example of a base image, which is the Docker Hub's official public alpine image.

> **Note**
>
> Before using images from a public repository, take steps to review the image for security vulnerabilities.

```
FROM alpine:3.13
```

The application server image is a specific image of the platform application to run the developer's code but does not contain the application's binary code itself. See the following example of an application server image, which is Docker Hub's official NGINX image. It contains the base image of

`alpine:1.13` and the NGINX platform application. The following Dockerfile is built using `docker build -t nginx:1.20.1-alpine`.

```
FROM alpine:3.13
...
ENTRYPOINT ["/docker-entrypoint.sh"]
CMD ["nginx", "-g", "daemon off"]
```

Lastly, the application binary created by the developer should reside in the container image. The container image comprises all the developer's source code to run their application. The following example uses the application server image, and copies the application code into the image.

```
FROM nginx:1.20.1-alpine
COPY . /usr/share/nginx/html
```

Use source control and tagging on all container images

Maintain the Dockerfile for all container images in a source control repository in the image hierarchy and ensure proper tagging of container images. In addition, use a contentious integration process to create a direct correlation between the container's images in source control and the image tag. This best practice is critical to determine what changed in the container image from a prior release.

For example, tag `1.0` indicates that this tag will always point to the latest patch release `1.0.1`, `1.0.2`, `1.0.3`, and so on.

Failure management

CONTAINER_BUILD_REL_05: How do you monitor the health of a container?

Plan for health checks in all containers builds and deployments

It is common to initially develop container applications without thinking of the availability of the services in the container. When running container applications, there is no way of knowing whether the services running within a container are up or not. Adding a health check or probe to the container provides testing of the services in the container. Health check options are available

in Docker using the HEALTHCHECK command, however, `containerd` does not have this option. Examine the orchestrations systems health check and probing options. This could include liveness and readiness probes within Amazon EKS or health checks within a definition file within Amazon ECS.

Resources

This section provides companion material for the Container Build Lens with respect to the reliability pillar.

Blogs and documentation

- Create the AppSpec file
- AppSpec File example
- Task definition parameters
- Dockerfile reference: Healthcheck
- How Amazon ECS manages CPU and memory resources
- Tutorial: Deploy an Amazon ECS service

Whitepapers

- EKS best practices (reliability)
- Amazon ECS best practices - Auto scaling and capacity management

Training materials

AWS containers immersion day

Performance efficiency pillar

The performance efficiency pillar focuses on the efficient use of computing resources to meet requirements, and maintaining that efficiency as demand changes and technologies evolve.

Take a data-driven approach to selecting a high-performance architecture. Gather data on all aspects of the architecture, from the high-level design to the selection and configuration of resource types. By reviewing your choices on a cyclical basis, you will ensure that you are taking

advantage of the continually evolving AWS Cloud. Monitoring will make you aware of any deviance from expected performance and allow you to take action on it. Finally, you can make tradeoffs in your architecture to improve performance, such as using compression or caching, or by relaxing consistency requirements.

Best practices

The performance efficiency pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve. In the context of containers, there are two different aspects of performance: the build-time performance for your container image, and the runtime performance. In this section, we'll focus on build-time performance in order to reduce the time that is necessary to create a container image from a Dockerfile. Reducing the build-time of a container image has a huge impact on developer productivity, and it reduces the amount of time that is necessary to roll out new versions of your application to production. Optimizing the performance at runtime has a direct impact on customer experience, latency, and costs, but is outside the scope of this lens. The following guidance is in addition to the best practice guidance found in the Performance Efficiency Pillar whitepaper.

The following are best practice areas for performance efficiency in the cloud:

- Selection
- Review
- Monitoring
- Tradeoffs

Selection

CONTAINER_BUILD_PERF_01: How do you reduce the size of your container image?

Use small parent images

The OS parent image that is used to create the target images has a huge impact on the final container image size. We can see huge differences when comparing different base images:

```
ubuntu:20.04      72.7MB
debian:10-slim    69.3MB
```

```
alpine:3.14       5.6MB
```

We can use Alpine to build performant containers, but for certain languages there are even more optimized run environments we can leverage. When using statically linked binaries, scratch can be an alternative. In the following example, we have a multi-stage build with a builder image based on Alpine (we will discuss multi-stage builds later in the document). In the first stage, the Go programming language application is built to a statically linked binary, the second stage uses scratch as base image and run the application built in the first stage. With this combined approach of a multi-stage build and using scratch as a base image, we achieve the smallest possible target image for running our application:

```
FROM golang:alpine AS builder

....

RUN go build -o /go/bin/myApplication

FROM scratch
COPY --from=builder /go/bin/myApplication /go/bin/myApplication

CMD ["/go/bin/myApplication"]
```

Run a single process per container

It is highly recommended to limit the number of processes in each container to one. This approach simplifies the implementation of separations of concerns using simple services. Each container should only be responsible for a single aspect of the application that facilitates horizontal scaling of this particular aspect. If it's necessary to run more than one process per container, use a proper process supervisor (like supervisord) and an init system (like tini).

Exclude files with from your build process

The `.dockerignore` file is similar to `.gitignore` and is used to exclude files that are not necessary for the build, or are of a sensitive nature. This can be useful if it's not possible to restructure the source code directory to limit the build context. The following example shows a typical `.dockerignore` file, which excludes files like the compilation target-directory, JAR-files, and subdirectories.

```
*
```

```
!target/*-runner
!target/*-runner.jar
!target/lib/*
!target/quarkus-app/*
```

CONTAINER_BUILD_PERF_02: How do you reduce the pull time of your container image?

Use a container registry close to your cluster

One of the essential factors in the speed of deploying container images from a registry is locality. The registry should be as close to the cluster as possible, which means that both the cluster and the registry should be in the same AWS Region. For multi-region deployments, this means that the CI/CD chain should publish a container image to multiple Regions. An additional way to optimize the pull time of your container image is to keep the container image as small as possible. In the section called “Tradeoffs” multi-stage builds are discussed in detail to reduce the image size.

Review

CONTAINER_BUILD_PERF_03: How do you make sure to get consistent results for your target images?

Using the `latest` tag for the parent image could potentially lead to issues because the latest version of the image might include breaking changes compared to the version that is currently used.

CONTAINER_BUILD_PERF_04: How do you make sure to use updated versions for parent images?

Implement a notification mechanism for updated parent images

If you’re using a team- or enterprise-wide image, you should implement a notification mechanism based as part of your CI/CD chain to distribute the information about a new parent image to

the teams. The teams should build target images with the new parent images and measure the performance impact of the changes by running a proper test suite.

Monitoring

CONTAINER_BUILD_PERF_05: How do you make sure you get consistent performance results over time?

Implement an automated performance testing strategy

System performance can degrade over time. It's important to have an automated testing and monitoring system in place to identify degradation of performance. Every time you build target images based on new parent images, you should measure the performance impact of the changes in the parent image. This also includes the overall build process, because we have to make sure that a testing and monitoring system covers the CI/CD chain. Performance metrics and image sizes have to be collected using services like Amazon CloudWatch and teams must be alarmed if anomalies have been detected.

Tradeoffs

CONTAINER_BUILD_PERF_06: How do you optimize the size of your target image?

Use caching during build

A container image is created using layers. Each statement in a Dockerfile (like RUN or COPY) creates a new layer. These layers are stored in a local image cache and can be reused in the next build. The cache can be invalidated by changing the Dockerfile, which means that all subsequent steps to build the image must be rerun. Naturally, this has a great influence on the speed the image is built. Thus, the order of the commands in your Dockerfile can have a dramatic effect on build performance. In the following example you can see the effect of the proper ordering of statements in a Dockerfile:

```
FROM amazonlinux:2
RUN yum update -y
COPY . /app
```

```
RUN yum install -y python python-pip wget
CMD [ "app.py" ]
```

This simple container image uses `amazonlinux` with tag 2 as parent image. In the second step, the Amazon Linux distribution is updated with the latest patches. After that, the Python application is copied into the container image. Next, Python, pip, wget, and additional dependencies required by the application are installed. In the final step, we start the application. The issue with this approach is that each application change results in cache invalidation for all subsequent steps. A small change in the application results in a rerun of the Python installation, which has a negative impact on build time. An optimized version of the Dockerfile looks like this:

```
FROM amazonlinux:2
RUN yum update -y && yum install -y python python-pip wget

COPY . /app

CMD [ "app.py" ]
```

Now the `COPY` statement of the application is located after `yum install`. The effect of this small adaption is that a change of the application code results in fewer layer changes. In the previous version of the file, each application change results in an invalidation of the layer that installs Python and other dependencies. This had to be rerun after a code change. One additional aspect, which is covered in the optimized version of this Dockerfile, is the number of layers. Each `RUN` command creates a new layer, by combining layers it is possible to reduce the images size.

Use the CPU architecture with best price to performance ratio

AWS Graviton-based Amazon EC2 instances deliver up to 40% better price performance over comparable current generation x86-based instances for a broad spectrum of workloads. Instead of using one build-server for x86 and ARM in combination with QEMU for CPU emulation, it might be a more efficient architecture to use at least one build server per CPU architecture. For example, it is possible to create multi-architecture container images to support AWS Graviton-based Amazon EC2 instances and x86 using AWS CodeBuild and AWS CodePipeline. As described in the blog post Creating multi-architecture Docker images to support Graviton2 using AWS CodeBuild and AWS CodePipeline, this approach includes three CodeBuild projects to create an x86 container image, an ARM64 container image, and a manifest list. A manifest list is a list of image layers that is created by specifying one or more (ideally more than one) image names. This approach is used to create multi-architecture container images.

Resources

This section provides companion material for the Container Build Lens with respect to the performance efficiency pillar.

Blogs and documentation

- Advanced Dockerfile: Faster builds and smaller images using BuildKit and multistage builds
- Use multi-stage builds
- Dockerfile reference
- Run multiple services in a container
- Best practices for writing Dockerfile

Partner solutions

Dockerfile best practices

Videos

Dockerfile best practices

Cost optimization pillar

The cost optimization pillar includes the continual process of refinement and improvement of a system over its entire lifecycle. From the initial design of your first proof of concept to the ongoing operation of production workloads, adopting the practices in this document can enable you to build and operate cost-aware systems that achieve business outcomes and minimize costs, thus allowing your business to maximize its return on investment.

When developing and building applications that will be run as containers, be aware of the cost-effective aspect of the container build process. Similar to writing application code, building a container image for your application can be done quickly, but will probably be much less cost-efficient to build, run, and operate. The best practices and techniques laid out in this section of the whitepaper will guide you to implement changes into the process of building container images that will help you reduce costs for building, running, and operating your container images.

Best practices

Cost optimization is a continual process of refinement and improvement over the span of a workload's lifecycle. In addition to the best practices described in the Cost Optimization Pillar whitepaper, this section covers how the way you're designing and building your containerized applications can have a direct or indirect influence on the cost of running those applications.

The following are best practice areas for cost optimization in the cloud:

- Practice cloud financial management
- Expenditure and usage awareness
- Cost-effective resources
- Manage demand and supply resources
- Optimize over time

Practice cloud financial management

CONTAINER_BUILD_COST_01: How do you design your container build process to avoid unnecessary cost?

Building a containerized application can result in multiple images for the same service. Depending on your organization policy, you might want to keep a subset of your container images to be used in a case of a rollback scenario. An example of such a policy might be that you don't roll back more than three versions, or more than three months in time. That means, that not all container images of a specific application should be kept. Deleting old images can save costs as container registries charge by size of images stored in the registry. You can achieve this deletion policy by creating automation processes, or use service features, for example: Amazon ECR supports a lifecycle policy that can be used to expire (delete) images based on rules such as image age, count, specific tags and more (see Examples of lifecycle policies).

Expenditure and usage awareness

CONTAINER_BUILD_COST_02: How do you design your container build process to avoid unnecessary cost?

Designing efficient container build process

Building containers is a process that consumes compute and storage resources and can lead to unnecessary costs if not using it properly. The build process consumes resources for each build, and there are some considerations that have to be taken for it to be efficient from a cost perspective.

Application dependencies

The container image is usually being built alongside with the application build step. During this build step, all necessary dependencies, libraries, and modules that are being used by the application code are downloaded to the container image. Using unnecessary dependencies will make the build time longer, and will result in wasting compute resources of the build system.

Common container image dependencies

Some operating system packages are needed for multiple applications in the organization for a specific runtime (for example, Python and Java). Building a parent container image that preinstalls all common operating system packages and dependencies for the specific runtime will result in a more efficient build process. Without this common image, each individual container image would be installing the same packages, thus wasting compute and network resources. This practice will also shorten the time for container images built from a specific runtime, since all of its common operating system packages and dependencies are already included in the parent container image. As a result, this will reduce costs for building all other container images that use this parent image.

Cost-effective resources

Container images can affect the overall cost of the system and workloads. In this section, we'll describe what has to be done in order to optimize your container images, and why it is important from cost perspective.

CONTAINER_BUILD_COST_03: Ensure that your container images contain only what is relevant for your application to run

Container image size affects several processes related to running your containerized workload. The following processes can be affected by having unnecessary large container image size.

Containerized application start-up time

Container image size affects the time needed for an image to be pulled from a container registry. Large image sizes (hundreds, or thousands of MB), can lead to a slow startup time of the application, which can lead to:

- Waste compute resources while waiting for images to be pulled.
- Slow scale-out operations.

Container image size also affects the scaling time needed for a containerized application to become ready to receive traffic. This time can translate to a waste of resources. In small-scale replicas of your application, the waste might not be notable, but when dealing with a dynamic autoscaled environment, a 30-second delay between a triggered scale-out event and a container ready to run can result in hundreds of compute minutes wasted per month. To put this delay into an equation, 30 seconds multiplied by 100 pod launches per day, over a period of one month, can result in:

```
30(sec)*100(launches of pods)*30 (days)=90,000 seconds = 1,500 minutes of compute time
 that is wasted.
```

Storage requirements for containers

Consider your instance's storage requirements depending on your container image size. The size of your container image has a direct effect on the instance storage size that the container will run on. This can result in the need for a larger storage size for your instances.

Container image size also affects the storage requirements of the container registry, since the container image will be stored in the registry. Stored images in Amazon ECR are priced per GB-month. For current pricing, refer to the pricing page.

CONTAINER_BUILD_COST_04: How do you reduce your container images size?

A container image consists of read-only layers that represent a Dockerfile instruction. Each instruction creates one additional layer on top of the previous layer. Running multiple consecutive commands can result in a large container image size, even if we delete content in the container image itself. An example of that might be installing a package, and deleting the cached downloaded files that are not needed anymore after installing the package. The following example

shows that we installed `some-package` and then delete the cached files. Even though we used the `rm` command to remove the cached file, the container image contains a layer representing the `rm -rf ...` command, and is still containing a layer with the actual cached files, resulting in a larger container image.

```
RUN apt-get install -y some-package
RUN rm -rf /var/lib/apt/lists/*
```

In order to overcome this, we can concatenate commands, and use a one-liner approach to install packages and remove the cached files in a single command:

```
RUN apt-get install -y some-package && rm -rf /var/lib/apt/lists/*
```

Reducing image layers can be done with several techniques:

- **Building container images from the scratch image** - This can result in creating the minimal container image possible, especially when containerizing executable applications with minimal external dependencies from the OS (like Go, Rust, and C).
- **Use lightweight base images** - For other type of programming languages that need a runtime environment in the container image, using parent and base images of lightweight distributions like Alpine can reduce the image size significantly. For example: a Python container image based on Debian parent image is ~330MB in size whereas a Python container image based on Alpine parent image is ~17MB in size.
- **Reducing the number of `RUN` instructions by chaining commands together** - Installing dependencies and deleting cache in a single command as shown in the previous command. This practice should only be used when the consecutive commands relate one to another.
- **Consider using package managers flags to reduce dependency sizes** - Such as `no-install-recommends` with `apt-get`.
- **Use multi-stage builds** - Multi-stage builds let you reduce image sizes by using build cache from the previous build step and copying only needed dependencies to the final container image. For example, see docker docs.
- Follow Dockerfile best practices such as:
  - Use `COPY` instead of `ADD`.
  - Use absolute path when using `WORKDIR`.

- Exclude files from the build process using `.dockerignore`. Specify exclusion patterns of file, directories or both (similar to `.gitignore`). This makes it easy to exclude unnecessary files from `COPY` or `ADD` commands.

CONTAINER_BUILD_COST_05: How do you design your containerized application to support automatic scaling and graceful termination?

When designing applications that will be containerized, it is important to include signal handling within the code and/or the container itself. Handling signals is a fundamental practice for writing applications, especially when writing applications that will run inside a container. The application should handle system signals and react according to the application logic. Although this is not directly related to cost, handling signals is a key element for using cost saving practices like automatic scaling or using Amazon EC2 Spot Instances. When a scale-in event, or replacement or termination of a Spot Instance occurs, the container orchestrator system or tools will send a `SIGTERM` signal to the application notifying the application to shut itself down gracefully. If it fails to do so, the process may end up being terminated while performing work, which can prohibit the use of auto scaling or spot in general.

CONTAINER_BUILD_COST_06: How do you design your containerized application to support multiple CPU architectures?

Different instance families offer different performance for the same amount of hardware (CPU and memory). An example is using a newer instead of an older generation of instances, or using instances with different CPU architecture, such as ARM. To use a different instance architecture, you have to change your build process. Since the default behavior of the build process is to create a container image that is designed to run on the architecture of the instance that it was built on, you have to create multiple images for each CPU architecture. To create multiple images, run the same build process on an x86 instance, and on an ARM-based instance. Use tagging suffixes to differentiate between the different architectures. You can see an example container image tag when searching images in your registry (see aws-node-termination-handler and search for container images in the Amazon ECR public gallery under the **Image tags** tab) and look for the different *-linux-arm64* and *-linux-amd64* suffixes. Having different container image tags allows

you to schedule and run the x86 version of your container image on an x86 instance, and the ARM version of your container image on an ARM instance. You have to make sure that the correct container image version will run on the correct instance. You can't run the ARM version of your container image on an x86 instance, and vice versa. The Docker image manifest v2 schema 2, and the OCI image index specification were created to make it easier to run your container image on any architecture. This additional specification allows you to create a high-level manifest that contains a list to other already existing container images. This container manifest, contains a reference to all the different container images, specifying their operating system type, architecture, and the digest of the container image that is referenced. Using the manifest, the container runtime will know which image to pull based on what architecture the underlying instance uses. An example of different container image manifests and manifests list, is displayed in the following screenshot from the Amazon ECR public gallery:

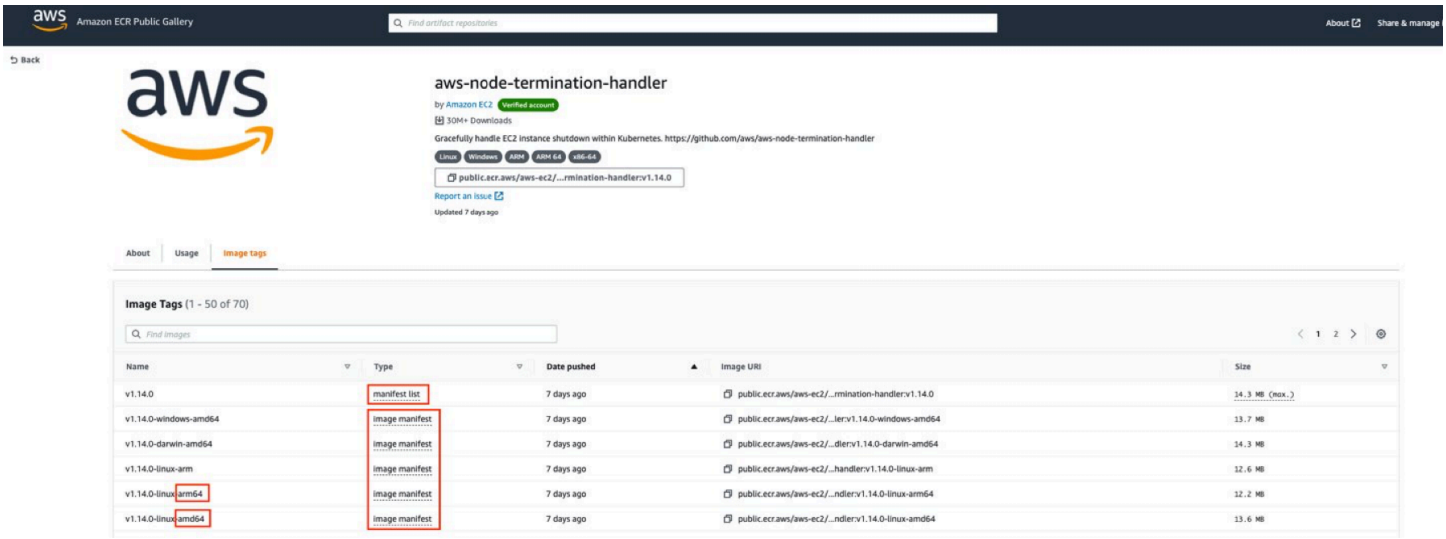

Figure 1. Example of container image manifests and manifest lists for multi-architecture container images

Manage demand and supply resources

CONTAINER_BUILD_COST_07: How do you minimize cost for your containerized application during startup time?

Longer startup times for containerized applications can result in wasted compute resources. Shortening startup times can be done on the application-level (code optimization), or on the container level. For example, if the application needs external dependencies to be present in

the container, it should be already installed during the build process, or it should be included in the parent image, and not downloaded at startup using an entrypoint script or `DOCKERFILE` commands.

CONTAINER_BUILD_COST_08: What systems are you using to create your container build process?

Creating any build process requires developing, maintaining, and operating a build system. This can be done by a variety of methods, such as using OSS tooling for job automation, or using self-developed systems that are able to run build scripts for your application. However, running and maintaining this kind of system involves software development costs, operational costs, compute, and storage costs for running the system. Alternatively you can use build and pipeline services, such as Amazon EC2 Image Builder, AWS CodeBuild, and AWS CodePipeline. Using managed services removes the operational overhead and allows developers to consume pipeline runs and build jobs on a pay-as-you-go basis.

Optimize over time

There are no cost best practices unique to the container build process for optimize over time.

Resources

This section provides companion material for the Container Build Lens with respect to the cost optimization pillar.

Blogs and documentation

- StopTask API sending SIGTERM signal
- Termination of Pods (signals)
- Graceful shutdowns with Amazon ECS

Partner solutions

- Dockerfile best practices
- Multi stage builds
- Don’t install unnecessary packages

Sustainability pillar

The sustainability pillar focuses on environmental sustainability.

Sustainability in the cloud is a continuous effort focused primarily on energy reduction and efficiency across all components of a workload by achieving the maximum benefit from the resources provisioned and minimizing the total resources required. This effort can include:

- The initial selection of an efficient programming language.
- The adoption of modern algorithms.
- The use of efficient data storage techniques.
- The deployment of correctly sized and efficient compute infrastructure.
- The minimization of requirements for high-powered end-user hardware.

Best practices

The sustainability pillar focuses on environmental impacts, especially energy consumption and efficiency, since they are important levers for architects to inform direct action to reduce resource usage. You can find prescriptive guidance on implementation in the Sustainability Pillar whitepaper.

The following are best practice areas for sustainability in the cloud:

- Region selection
- User behavior patterns
- Software and architecture patterns
- Data patterns
- Hardware patterns
- Development and deployment process

Region selection

The Region selection best practice focuses on where you will implement your workloads based on both your business requirements and sustainability goals. There are no sustainability practices unique to the container build process for Region selection. Refer to the Region selection best practice from the sustainability pillar of the Well-Architected Framework for more information.

User behavior patterns

There are no sustainability practices unique to the container build process for user behavior patterns. Refer to the user behavior patterns best practice from the sustainability pillar of the Well-Architected Framework for more information.

Software and architecture patterns

CONTAINER_BUILD_SUSTAINABILITY_01: How do you design your containerized application in a way that reduces the use of the underlying resources?

When designing containerized application, you should keep your build manifests up-to-date and aligned with your application needs. A containerized application image starts from a Dockerfile. The Dockerfile includes all commands required to include the configuration and dependencies for the containerized application. If there are some dependencies that are no longer required, removing them from the Dockerfile can:

- Reduce the time that it takes to build the container image. This affects host resource consumption by the build process.
- Reduce the container image size and therefore reduce the time it takes for this image to be pulled to an instance. This affects host resources usage for running and storing the container images.

Data patterns

There are no sustainability practices unique to the container build process for the data patterns best practice, see the data patterns best practice from the sustainability pillar of the Well-Architected Framework for more information.

Hardware patterns

CONTAINER_BUILD_SUSTAINABILITY_02: How do you support your containerized application to run on energy-efficient hardware?

To be able to use instance types with the least environmental impact (from the Sustainability Pillar whitepaper), you have to ensure your containerized application is able to run on a variety of instance types and architectures. This can be done by creating images that support multi-architecture as described in the Cost Optimization Pillar whitepaper . For example, you can use a build service that supports multi-architecture build servers and combine them to a multi-architecture image using the CI pipeline (see Graviton workshop as an example of using AWS CodeBuild, and AWS CodePipeline alongside Graviton and Amazon EKS). You can also use tools that generate multi-architecture images from a single Dockerfile, such as Docker Buildx.

Development and deployment process

Look for opportunities to reduce your sustainability impact by making changes to your development, test, and deployment practices.

CONTAINER_BUILD_SUSTAINABILITY_03: How do you design your build tooling and services to improve efficiency of the underlying resources?

Use dynamically created build servers for building your containerized workload

Using dynamically created build servers (such as AWS CodeBuild), ensures that while building your containerized images, the needed infrastructure is being provisioned when the build process starts, and being terminated as soon as the build process ends.

Use pre-defined or built runtimes to reduce your build time, and reuse needed dependencies for the build process

When building different types of containerized applications, using common and standardized runtimes for the build process reduces the operational management of creating and maintaining custom images. Also, by using the specific type of runtime for your build server, it verifies that no common dependency is being downloaded and configured as part of the build process. All relevant dependencies are being incorporated into the different runtimes of your build servers, and are being used many times by different build processes for different applications. An example of multiple build runtimes can be found in the AWS CodeBuild documentation.

Update your parent and base image regularly

Update your base and parent images to the latest versions, as sometimes there is a performance improvement that is introduced in newer versions. These improvements are translated into a

sustainability improvement as it affects the resource consumption of the underlying infrastructure, and as a result improves the overall efficiency.

Delete unused or obsolete container images

As described in the Cost Optimization Pillar whitepaper, create mechanisms to verify that unused or obsolete container images are deleted. This can be achieved, for example, by registry lifecycle policies, as exists in Amazon ECR.

Resources

This section provides companion material for the Container Build Lens with respect to the sustainability pillar.

Blogs and documentation

General sustainability documentation:

- Ask an Expert - Sustainability
- AWS Sustainability

AWS sustainable documentation:

- Understand resiliency patterns and trade-offs to architect efficiently in the cloud
- Optimizing your AWS infrastructure for sustainability, part I: Compute
- Automated cleanup of unused images in Amazon ECR

Videos

- AWS re:Invent 2021 - Architecting for sustainability
- AWS re:Invent 2021 - Behind the Scenes: AWS and sustainability| AWS Events

Training materials

Graviton workshop

Scenarios

This section includes a set of customer use cases, applications for AWS features and services, reporting requirements, data structures, testing protocols, and workloads.

We will cover a series of scenarios that represent common patterns and strategies that are used when designing and building containerized applications. We will present the assumptions we made for each of these scenarios, the common drivers for the design, and a reference architecture of how these scenarios should be implemented.

The the section called “Container technology terminology” is a common component across the scenarios in the whitepaper. A CI/CD pipeline consists of the tools and automation to deliver a CI/CD strategy. The pipeline tools are general responsible for building code, running tests, and deploying updated versions of applications automatically.

Building a continuous integration (CI) pipeline that builds your containerized application and base images is critical to automating the software development lifecycle. CI pipelines enable development teams to automate the staging and building of the containerized application and base container images to ensure that development teams are focused on application development and not code deployment.

Scenarios

- Securing containerized build pipelines
- Improving containerized CI/CD pipelines from performance efficiency and cost perspective
- Improving performance for container image build process

Securing containerized build pipelines

A build pipeline is the process of creating a runnable and deployable artifact from the application source code. One of the steps towards adopting the use of container technologies, is updating the build pipeline to include the relevant steps for building containerized applications. When doing so, security measures should be considered for the build pipeline itself. This scenario suggests an architecture for creating a container build pipeline while handling various types of security aspects of the build pipeline itself, which are relevant to containers.

> **Note**
>
> All the security measures, controls, and tasks in this scenario come *in addition* to the relevant security actions needed when running any type of pipeline. An example can be seen in AWS CodePipeline security docs, AWS blogs, and other relevant content and materials.

Reference architecture

The following reference architecture diagram shows a containerized build pipeline with its steps.

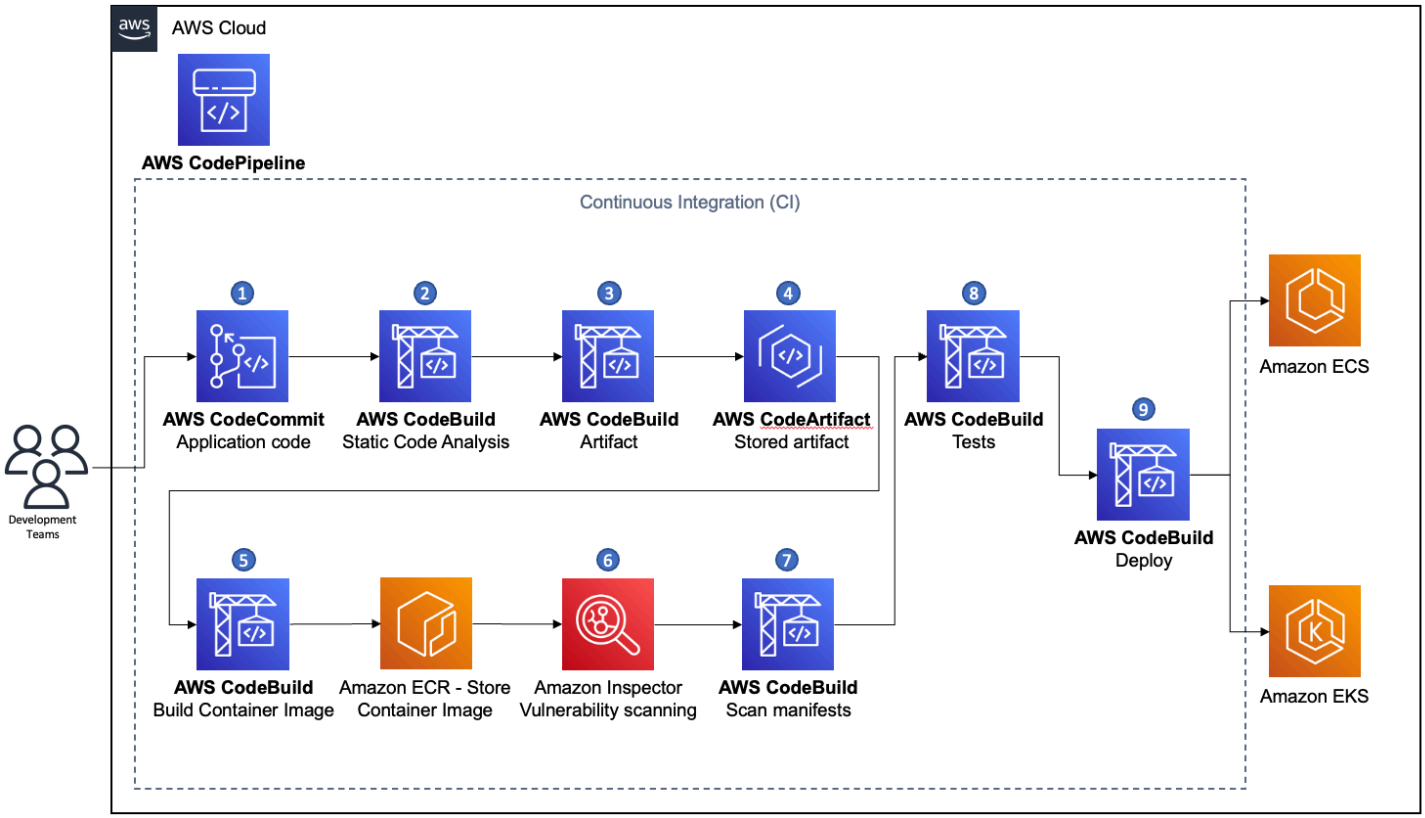

Figure 2. Continuous integration pipeline for containerized applications

1. Developers push a code change to the source code repository.
2. Static code analysis is triggered to identify potential misconfigurations, vulnerabilities of dependencies, ensure good coding practices, and more.
3. Build artifact from code (for example compile and link, or packaging source code).

4. Push and store artifacts in a targeted artifact or binary repository management service (for example AWS CodeArtifact, Nexus repository, or Jfrog Artifactory).
5. Build the container image based on the build manifest file (for example Dockerfile), handle tags and push it to the targeted registry.
    a. If building containerized only pipeline, you might skip storing the compiled artifact, as the target artifact is the container image.
    b. Enable tag immutability on your registry to reduce risk of pushing untrusted tags or versions of already existing container images.
6. Run static code analysis for the container image (for example using built-in registry scanning as exists in Amazon ECR, or use third-party tools such as Clair as used in Quai.io).
7. Based on the targeted orchestrator, the deployment manifests (for example Kubernetes YAML objects definitions, Helm chart packaging files, or Amazon ECS task definitions) should be scanned for vulnerabilities as well. Tools like checkcov can be used to perform such scans.
8. Run acceptance tests against the built container image.
9. Proceed to the deployment process.

One of the steps towards adopting the use of container technologies is adding steps related to building container images to the build pipeline. Usually, the following steps are the minimal necessary steps. Based on the previous list, they usually fall after step 3:

- Build the container image based on the build manifest file (for example Dockerfile), handle tags and push it to the targeted registry.
- Enable tag immutability on your registry to reduce risk of pushing untrusted tags or versions of already existing container images.
- Run static code analysis for the container image (for example using built-in registry scanning as exists in Amazon ECR, or use third-party tools such as Clair as used in Quai.io).
- Based on the targeted orchestrator, the deployment manifests (for example Kubernetes YAML objects definitions, Helm chart packaging files, or Amazon ECS task definitions) should be scanned for vulnerabilities as well. Tools like checkcov can be used to perform such scans.
- Run acceptance tests against the built container image.

Configuration notes

- Based on your targeted containers orchestrator, refer to the security best practices for Amazon EKS, or for Amazon ECS.
- Scan your container images, container manifests, and deployment manifests for vulnerabilities - your build pipeline should include a step for scanning the resulted container images for vulnerabilities.
- Make sure that your container images are built to use a user other than root.
- Ensure that your container images contain only relevant configuration and dependencies to the running application.
- Reduce attack surface for your container images (use minimal parent images, remove package managers, remove shell access). For more information, refer to the Protecting Compute section in the Security Pillar whitepaper.
- Use private container registries to store your trusted parent container images, after they were scanned and signed.

References

- Build a Continuous Delivery Pipeline for your Container Images with Amazon ECR as a Source
- Choosing a Well-Architected CI/CD approach: Open-source software and AWS services
- How to build a CI/CD pipeline for container vulnerability scanning with Trivy and AWS Security Hub
- Building end-to-end AWS DevSecOps CI/CD pipeline with open source SCA, SAST and DAST tools

Improving containerized CI/CD pipelines from performance efficiency and cost perspective

Continuous integration continuous delivery pipelines (CI/CD pipelines) help you automate steps in your software delivery process, such as initiating automatic builds and then deploying to a compute platform of your choice. Usually, a CI/CD-pipeline builds, tests, and deploys your code every time there is a code change, based on the release process models you define. As companies are adopting containers and container technologies more and more, this also increases the usage of their containerized build process. For further optimization, companies might seek ways to improve the efficiency of the overall build process, which includes containerization. There are

several ways to optimize the complete build process and the infrastructure involved. In the following section, we highlight a few popular optimization steps:

- Use managed services.
- Dynamically provision scalable infrastructure.
- Optimize the container images to support multiple runtimes and variations, which include different technologies and programming languages. This will improve not only the application performance efficiency, but also the sustainability impact of the containerized application.

Constantly checking and linting Dockerfile is considered a security best practice, but also has the advantage that it's possible to identify potential obsolete or unused dependencies for an application or for parent images. Organizations can define a standardized golden image, which contains all organization-related configuration that is necessary for all container images. All other images that contain application or tech domain-specific additions have to use this image as parent (direct or indirect) image. This increases the reusability of container images containing only relevant dependencies and configurations for the specific aspect. It is important to point out that this pattern requires images to be rebuilt if the parent image changes in order to reflect the latest changes.

Characteristics:

- You have an existing CI/CD pipeline or you're planning to introduce a pipeline.
- Your pipeline includes a build process for containerized applications.
- You are seeking to improve efficiency and cost of your pipeline solution.
- You want to create a framework that is easy to set up, operate, maintain, and scale, that you can extend with limited impact later.

Reference architecture

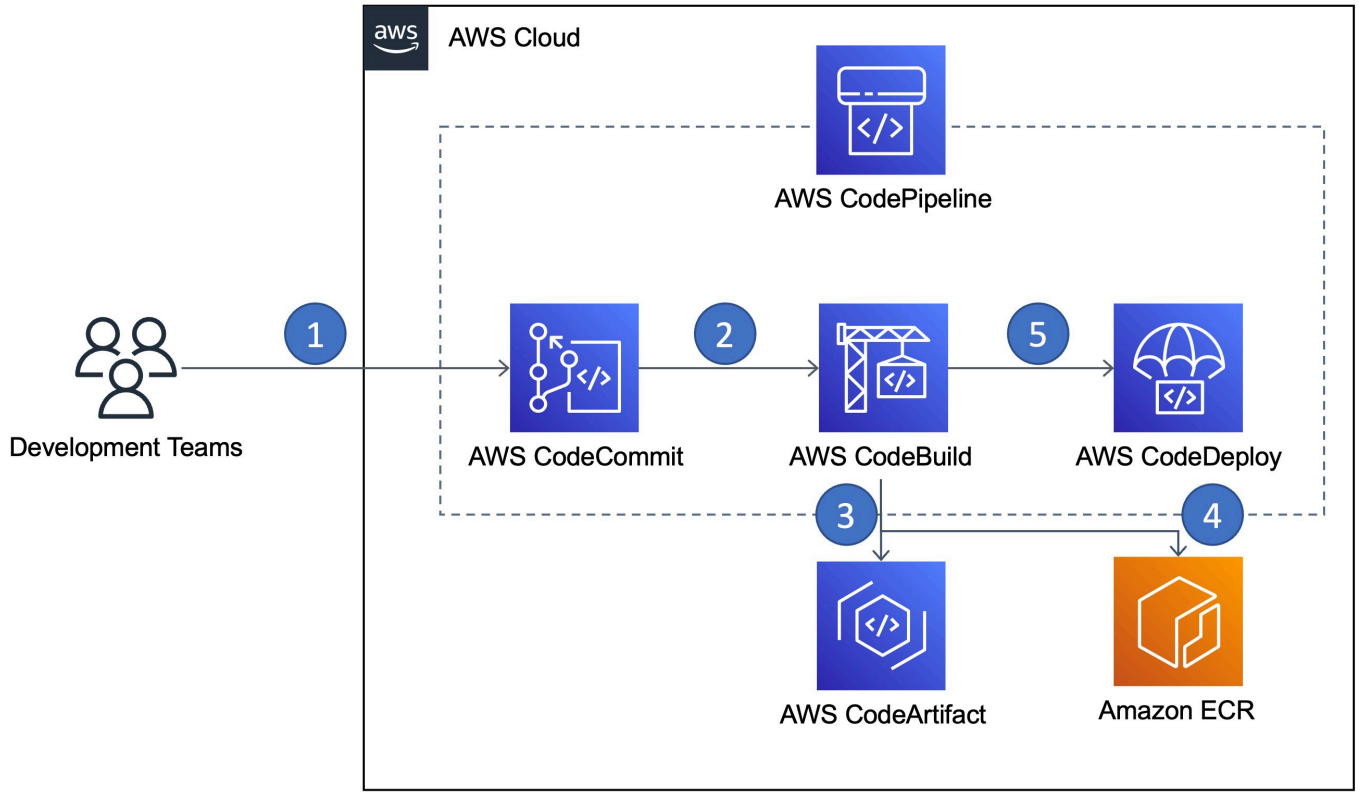

Figure 3. Improving the efficiency of a containerized application's CI/CD pipeline

The reference architecture shown implements a CI/CD pipeline that compiles source code of an application to an executable, stores the executable in an artifact repository, build a container image based on the executable, and stores the image in an image repository. For the pipeline implementation, we use AWS CodePipeline, which automates the build, test, and deploy phases of your release process every time there is a code change, based on the release model you define.

1. The code of the application, which is implemented by the development team is stored in AWS CodeCommit, developers can push their code using standard Git mechanisms.
2. If developers have committed code to a CodeCommit repository, AWS CodePipeline triggers AWS CodeBuild that compiles source code, runs tests, and produces software packages that are ready to deploy.
3. This step is optional and only applies if customers are in a transition phase in, which they have to support legacy workloads, which require artifacts like WAR-files for servlet engines. Those artifacts can be stored in AWS CodeArtifact.
4. The container image is built by AWS CodeBuild and is pushed to Amazon ECR.

5. The completed pipeline detects changes to your image, which is stored in an image repository such as Amazon ECR, and uses CodeDeploy to route and deploy traffic to an Amazon ECS cluster and load balancer. CodeDeploy uses a listener to reroute traffic to the port of the updated container specified in the AppSpec file.

Configuration notes

- To minimize your container images as much as possible, refer to multi-stage builds in the Performance efficiency section, which also verifies that your builds are reproducible. Multi-stage builds allow you to create consistent a build process, which consists of multiple steps for building your container image. By using multi-stage builds, your final container image should contain only relevant binaries, executables, or configurations, which are necessary to run the application.
- After building the images, check that the images are as compact as you expected.
- Use standardized tools like hadolint for Dockerfile linting as described in the Security Pillar.

References

- Building ARM64 applications on AWS Graviton2 using the AWS CDK and Self-Hosted Runners for GitHub Actions
- Build and Deploy Docker Images to AWS using EC2 Image Builder
- stackrox/kube-linter
- Dockerfile Linter

Improving performance for container image build process

It is important for an organization 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 serve their customers better and compete more effectively in the market.

Part of this process is automating the container build process and enabling it to do so at a high velocity. To do so, your organization can adopt methods and processes, such as:

- Using prebuilt images.
- Using the smallest parent image possible.

- Identify only what your container requires to accomplish your business goals.

Reference architecture

- You want a quick image build to enable you to iterate quickly in your development process.
- You want to ensure that your container images and supporting packages maintain your expected security posture.
- You want to use prebuilt images that include many of the packages that your container is dependent upon to speed up the build process.

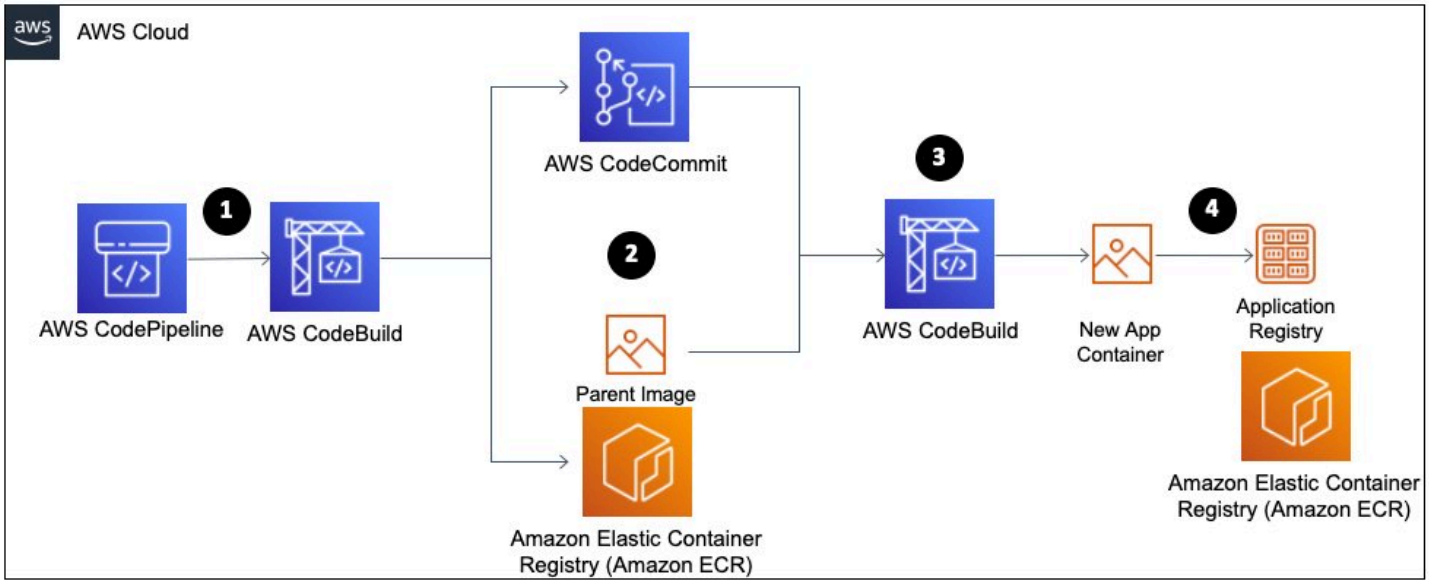

Figure 4. Improving the build pipeline performance of containerized applications

1. A **CodePipeline** pipeline for your application build is triggered, usually by something like a push to a code repository or writing an object to an Amazon Simple Storage Service (Amazon S3) bucket. This kicks off a CodeBuild stage.
2. The **CodeBuild** stage will initiate the building of your containerized application. It pulls the code from the code repository and takes the necessary steps to build your application code.
3. It pulls the parent image from the parent image container repository. It runs through the build steps for your application, as defined in your Dockerfile.
4. It then pushes the built containerized application image in the container registry.

Configuration notes

- Using prebuilt parent images can decrease the time it takes to build the application specific container image. Your image has to install fewer packages as many of them will be included in the parent image.
- Using a base image built in-house, you will have to do less functional testing because the base image has already been tested in previous builds.
- Work with other teams in your organization to determine the baseline security and business requirements that apply to all containerized applications and where there are some overlaps between teams.
- The base image of your container image should have the minimum packages and libraries that are required to build your application container image.
- Identify only what your container requires to accomplish your business goals, store things like picture images and state data outside of your container.
- If your running application is a compiled artifact, place only the artifact and necessary supporting packages in the container, excluding the unnecessary application code.

References

Build a continuous delivery pipeline for your container images with Amazon ECR as a source

Conclusion

The AWS Well-Architected Framework provides best practices across the six pillars for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. The framework provides a set of questions that allows you to review an existing or proposed architecture. It also provides a set of AWS best practices for each pillar. Using the framework in your architecture will help you produce stable and efficient systems, which allow you to focus on your functional requirements.

Contributors

The following have written, reviewed, and contributed to the AWS Well-Architected Container Build Lens.

- Joe Mann – Partner Solutions Architect, Redhat
- Erin McGill – Software Development Engineer, AWS Solutions
- Tsahi Duek – Specialist Solutions Architect, Containers
- Sascha Moellering – Specialist Solutions Architect, Containers
- Thomas Liddle – Solutions Architect, Containers
- Re Alvarez Parmar – Principal Specialist Solutions Architect, Containers
- Roland Barcia – Director, Worldwide Solutions Architects, Serverless
- Liz Duke – Senior Specialist Solutions Architect, Containers, UK/IR
- Dirk Herrmann – Head of Compute SA, AWS Worldwide Specialist Organization Compute
- Satveer Khurpa – Senior Containers and Serverless DLA, AWS Data Lab
- Andreas Lindh – Senior Specialist Solutions Architect, Containers
- Nirmal Mehta – Principal Specialist Solutions Architect, Containers
- Aaron Miller – Principal Specialist Solutions Architect, Containers
- Praveen Nerellapalli – Senior Containers and Serverless DLA, AWS Data Lab
- Theo Salvo – Senior Containers Specialist Solutions Architect
- Viji Sarathy – Principal Specialist Solutions Architect, Containers
- David Surey – Senior Solutions Architect
- Bruce Ross – Senior Lens Leader, AWS Well-Architected

Further reading

For additional information, see the following:

- AWS Well-Architected Framework
- AWS Architecture Center

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Container Build Lens first published.	October 20, 2022

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.