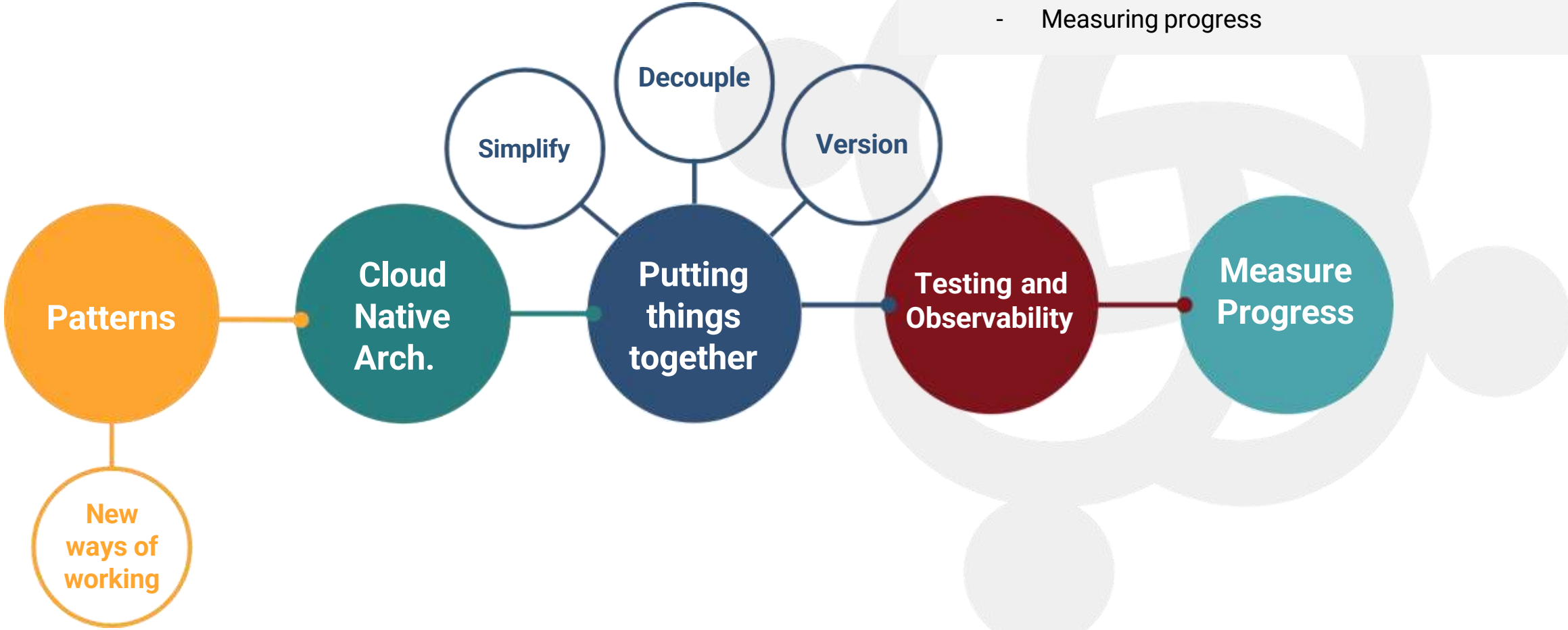
DevOps Workshop Series

Evolving to Continuous Deployment

Our agenda

You will learn:

- Different patterns for safe, continuous deployment
- What is cloud native architecture all about
- How does cloud native enable continuous deployment
- Testing, observability, and release management for continuous deployment
- Measuring progress



CONTINOUS DEPLOYMENT

PATTERNS, PREREQUISITES AND GOTCHAS

Continuous Delivery vs Deployment

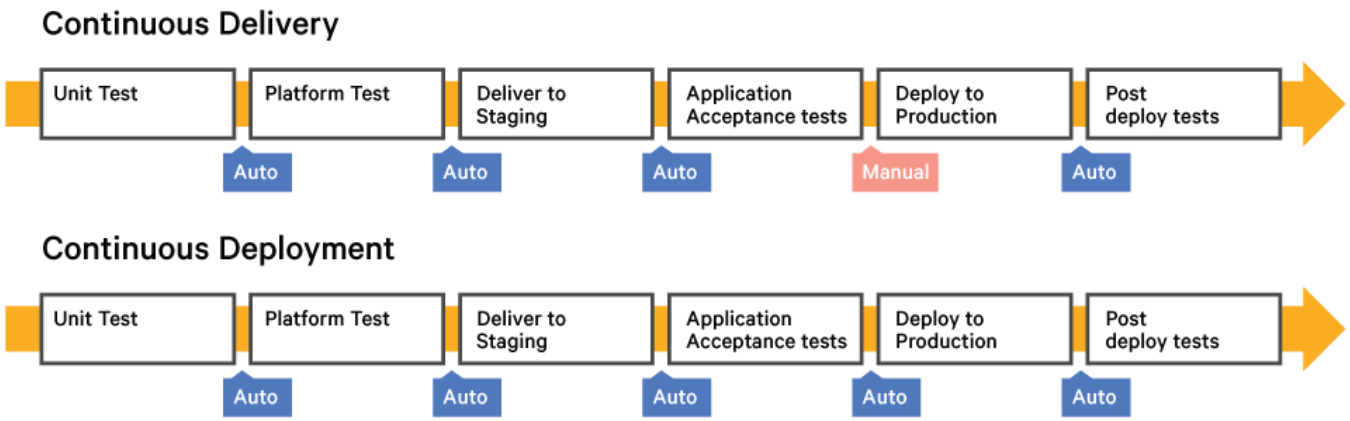
FROM RELEASE READY TO CONTINUOUSLY INTO PRODUCTION

Continuous Delivery

New versions of your application are continuously ready to be released but require a manual validation process to become available in a production environment.

Continuous Deployment

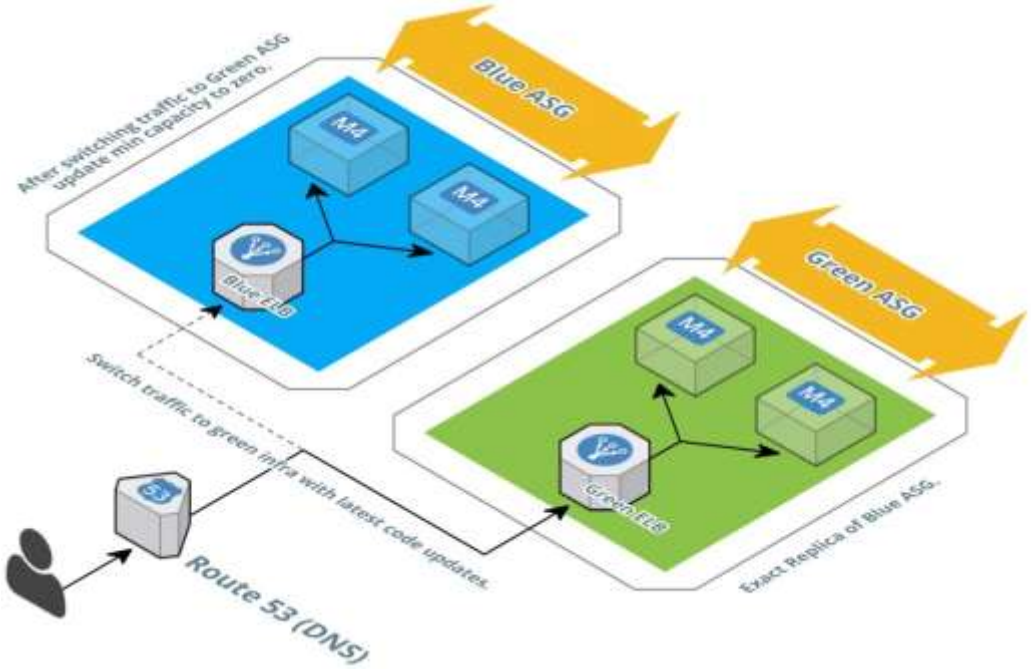
Application releases are fully automated all the way into a production environment, validation of every release requires no human intervention.



Patterns

- Blue/Green
- A/B
- Canary
- Feature Flags

Blue/green



How it works

During release a duplicate environment identical to the existing production receives the new version. Once tested, and validated, traffic is switched over. Rollback is handled by directing traffic back.

Prerequisites

A duplicate production environment or enough automation to spin one up on demand.

Caveats

Having to run two full scale production environments simultaneous is costly. It’s not progressive.

A/B

How it works

A new release is deployed to production, but only made available to a specific subset of users. Once tested and validated, can progressively be exposed to more users, or be considered suitable for full traffic.

Prerequisites

You must be able to target specific users using headers, cookies or perhaps geographical regions. Mature observability is desired considering users will be exposed to a newly released version.

Caveats

As with other progressive delivery patterns, backwards compatibility of data is critical.

Canary

How it works

The new version is deployed to production and starts receiving a percentage of production traffic. Using metrics, tests and other automated mechanisms, the release is validated, and traffic gradually increased.

Prerequisites

Mature mechanisms to automatically a new release as “good or bad”. Full integration of canary evaluations with release process.

Caveats

Canary releases may be slow. Data is always a concern when progressively releasing new versions. Requires active use.

Feature flags

How it works

New features are released to production but not immediately made available to users. Feature flags can be used together with A/B testing, Canary, and other types of deployment patterns.

Prerequisites

Your application must be built to support runtime feature changes, ideally without restarting (e.g., endpoint for config reload). Feature specific telemetry must be available for automated release validation.

Caveats

Not a deployment pattern on its own, rather a mechanism that can be used together with other deployment models.

Cloud Native Architecture

What is

Cloud Native Architecture

Cloud Native Architecture represents a way in which to plan, design, build and deploy applications, so they take advantage of the unique characteristics that cloud computing offers.

Cloud Native principles

- Everything automated
- Loose coupling
- Packaged and Immutable
- High Observability
- Single Responsibility
- 15 Factors

Everything automated

I MEAN **EVERYTHING**

- **Integration, testing, and packaging**
  CI/CD. Unit, functional and other testing.
  Containerization.

- **Infrastructure provisioning and service discovery**
  Infrastructure as Code.

- **Security controls**
  DevSecOps

Loose coupling

MICROSERVICES AND API-FIRST

- Every service hides its implementation behind an API.
- Services can be independently optimized and updated.
- Clearly defined service boundaries and responsibilities (we'll see more of this in single responsibility).

Packaged and immutable

DEPLOYMENT CONSISTENCY, VERSIONABLE AND SELF CONTAINED

- Every new version represents a specific point in time, which is immutable – once built it can't be changed, changes require building a new version.
- Every release is packaged encapsulating all dependencies, in a portable image.
- Containers have become the norm

High observability

VISIBILITY INTO DISTRIBUTED APPLICATIONS

- Consolidated, enriched telemetry
- Traceability across loosely coupled, distributed services
- All services instrumented by default
- Queryable logs, metrics and traces

Single responsibility

CLEAR BOUNDED CONTEXT, A SINGLE REASON TO CHANGE

- Services are responsible for a single, specific, bounded context.
- Each service should have only a single context of reasons for change.
- A service is responsible only for its own data.

The 12+3 Factors

BEST PRACTICES FOR PORTABILITY AND RESILIENCE

- Originally published by Heroku circa 2011 as 12-Factor apps
- Updated with three new factors by Kevin Hoffman from Pivotal in 2016
- 12factor.net
- Beyond the Twelve-Factor App – published by O'Reilly

1. One codebase, one app
2. API First
3. Dependency Management
4. Design, build, release, run
5. Configuration, credentials and code
6. Logs
7. Disposability
8. Backing Services
9. Environment Parity
10. Administrative Processes
11. Port Binding
12. Stateless Processes
13. Concurrency
14. Telemetry
15. Authentication and Authorization

Putting things together

Leveraging **Cloud Native** principles to enable **Continuous Deployment**

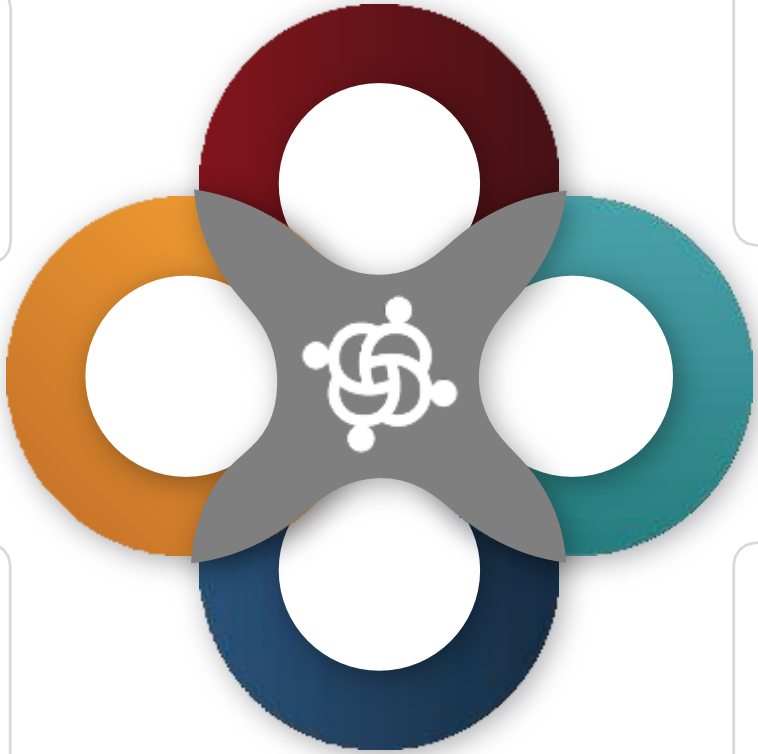
Deployment benefits of Cloud Native

SIMPLIFIED ROLLBACK

Versioning and immutability enable simplified and safe rollback mechanisms, allowing to revert to a predictable, reproducible previous state

ATOMIC, MINIMAL DEPLOYMENTS

Each service can be deployed independently, which minimizes the surface of possible incidents.



SCALE AUTOMATION

Creating ephemeral, production scale environments dynamically is dramatically simplified by the cloud

RELEASE PORTABILITY

Environment parity and portable, self contained images guarantee portability across multiple deployment stages

Orchestrating service deployment

Understand service relations

Loose coupling doesn't mean that there aren't relations between your services, understand how services participate in end-to-end requests and test accordingly.

Version your APIs

Artifact versioning is of little value if service APIs are not also versioned. Service interfaces should be versioned, and consumers should always define which specific version to use.

Event driven deployment

Sequential pipelines and hardcoded deployment patterns will dramatically increase risk. Use event driven, independent deployment stages.

Testing and observability

Testing – cornerstone of Continuous Deployment

Testing before production

- Unit testing
- Integration Testing
- End-To-End Testing
- Regression
- Functional
- Performance
- SAST and DAST

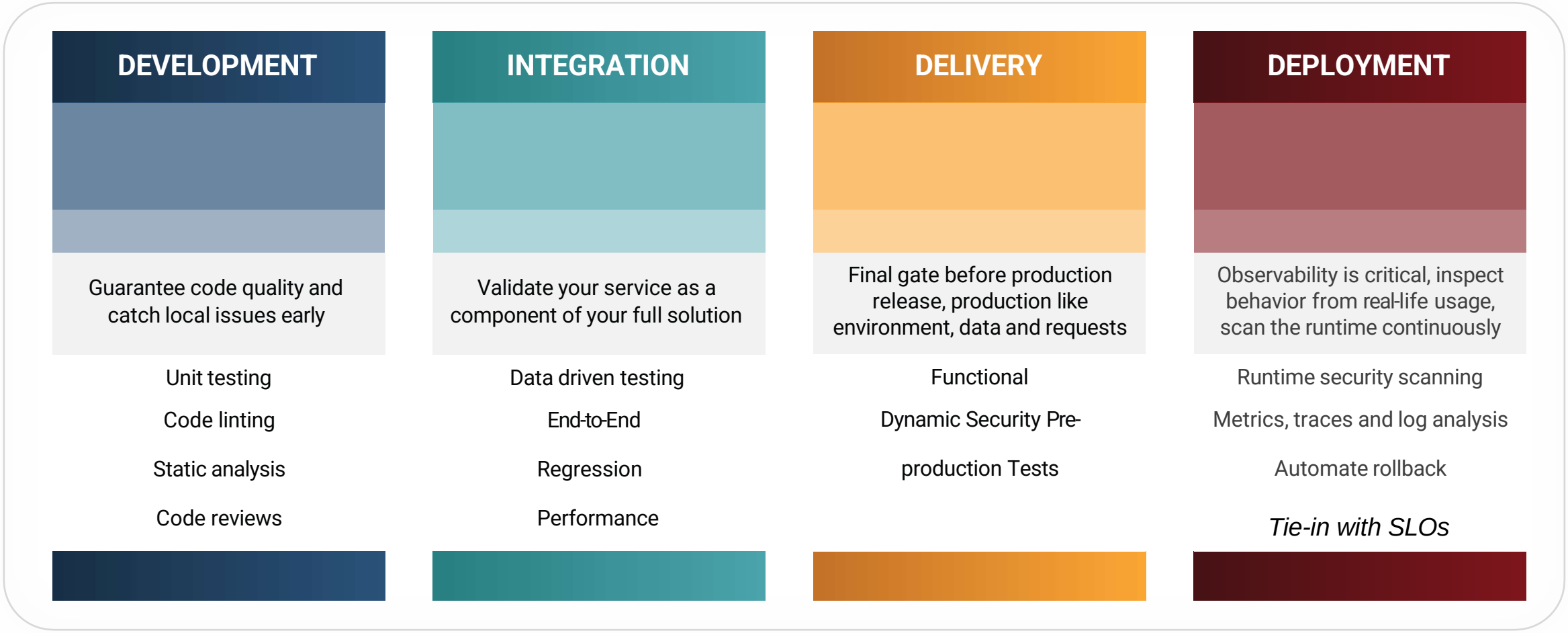
Make sure your services pass security tests, performance remains or is improved, and functionality works as expected

Testing in production

- Canary tests
- Feature metrics
- Runtime security scanning

Real production traffic is the ultimate validation, automated rollback your escape hatch, observability indispensable

Stages for testing and observability



Testing in production

Testing in production is not a replacement for solid testing across all other stages of the integration and delivery pipeline. It requires very mature continuous testing automation before reaching production.

Tests must be carefully designed not to disrupt real users of the service, and effectively target instances of the new versions in progressive deployment scenarios.

Observability is critical, you will want to identify drifts from desired behavior.

Observability, SLOs, and rollbacks

Mature Service Level Objectives, telemetry and system observability are indispensable for safe continuous deployment.

Telemetry will help track error budgets and SLO satisfaction. System observability will simplify the implementation of new means to track desired behavior drift, and these attributes together will be drivers for automated rollbacks.

Steps towards Continuous Deployment

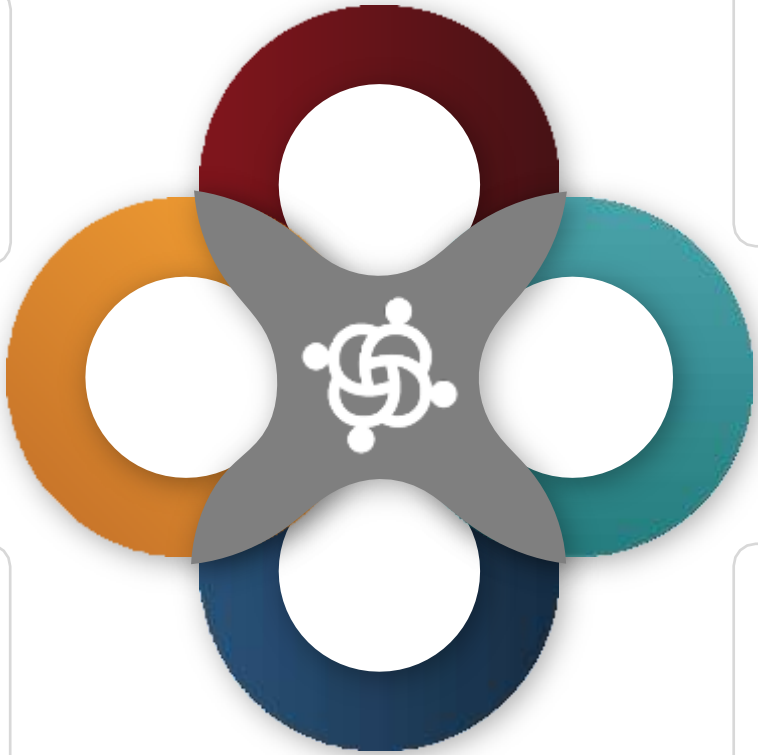
Understanding your solution

PERFORM SERVICE DISCOVERY

Some services may be better suited for continuous deployment as others. Continuous deployment can be approached gradually, understand your services and work with development teams.

VALIDATE API VERSIONING

Deploying microservices in a complex, distributed environment requires mature API versioning, validate that services expose a versioned API, and consumers are using specific versions.



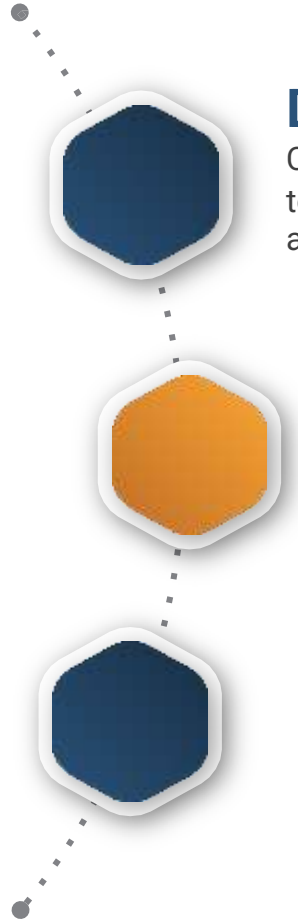
UNDERSTAND RELATIONSHIPS

The level of coupling across your full-service catalog may be diverse, and even in cases of loosely coupled services, there will be relationships and dependencies,

UNDERSTAND YOUR DATA

Data is particularly complicated in terms of continuous deployment. Identify data ownership and think about schema changes thoroughly.

What if we're not fully Cloud Native?



Don't boil the ocean

Carefully select services and applications that are closer to the end goal and work with small groups of applications.

Gain expertise with early adopters

As you gain expertise with early adopters, share that know-how with other teams, and help them navigate the challenges of applying cloud native principles to their solution.

Cont. deployment is not for all

Not everything can or should be continuously deployed, build a system that expects that, and don't try to fit a square peg in a round hole.

The Data challenge

Shift-left for DB changes

- Automation and as-code-declaration must incorporate DB related changes.
- In some scenarios, DB schema changes must be applied even if only a percentage of traffic will be using the new version, any DB change must be either isolated or backwards compatible with previous versions.
- In cases of rollbacks, it must be clearly defined what happens to data written by the new version.

Deployment, rollbacks and human gates

Automated rollback

The ability to automatically rollback a release when things go wrong is indispensable for continuous deployment. The rollback mechanism will vary depending on your deployment strategy.

Switch traffic back to existing environment

Blue/Green compatible

- Previous environment should not be immediately destroyed. A time threshold for production testing and telemetry accumulation must be configured.
- If any parameters fall outside of desired threshold, revert traffic back to the previous environment.
- New environment can be destroyed, however, consider that valuable insights as to causes for rollback will have been generated, make sure telemetry data persists.

Switch traffic away from new version

Canary and A/B compatible

- Revert any changes to traffic routing that would hit the new version or instances of the environment with a problematic feature enabled.
- If multiple features are enabled in an A/B testing deployment, this would effectively remove traffic to all features, not just the problem ones.
- In cases of A/B deployments, you must be able to target instances that have specific features enabled.

Disable features

A/B compatible

- Disable the feature(s) that are not behaving as expected.
- Doesn’t require a full version rollback.
- It’s important to understand what happens if features are disabled when they’re part of a flow, that users may see interrupted.

Cost and measuring progress

Cost of human gated deployment/rollback

%

33

Weeks

Lead time to production. **19% take months!**

#

06

People

Or more

Involved in release validation!

45 min

Rollback

45 minutes for microservices rollback, over an hour for monoliths.

Managing blue/green cost

- Automate provisioning and destruction of environments – ideally don’t leave a full parallel environment running unless it can be used otherwise.

- Leverage parallel environment for disaster recovery and high availability.

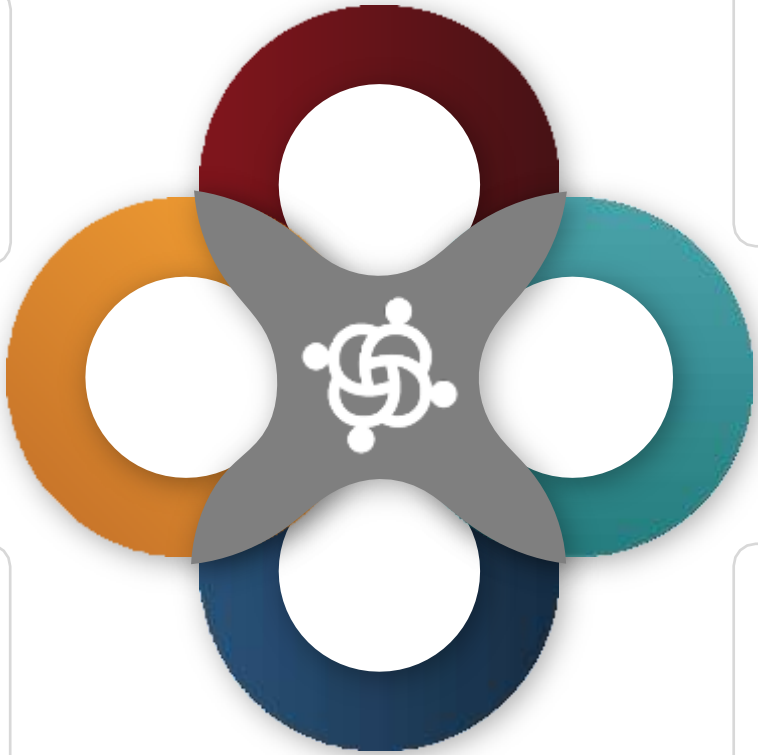
Managing progress

DEPLOYMENT FREQUENCY

You should see an increase in the number of deployments over time. This will reflect the reliability of your automation, your team's maturity and the maturity of your cloud native architecture.

LEAD TIME

The elimination of human gates and handoffs will be clearly reflected in a reduction of your lead time. This metric measures the time it takes for a feature to reach from merge to deployment.



MEAN TIME TO RESTORE

Your ability to handle automated rollbacks in production will have a positive impact in this metric, even in scenarios where issues arise during regular operations.

CHANGE FAIL PERCENTAGE

What is the ration between successful and unsuccessful changes reaching production, as your continuous deployment maturity increases, you should see a decreasing change fail %

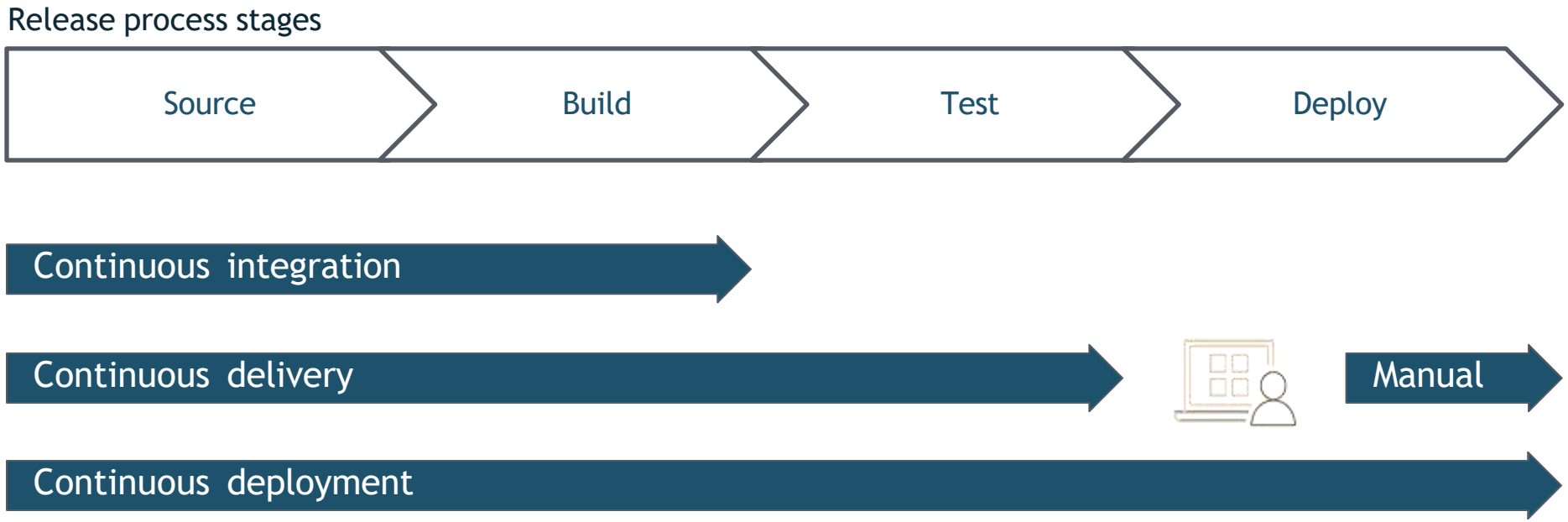
Pillars of releasing modern applications

Continuous integration

Continuous deployment

Infrastructure as code

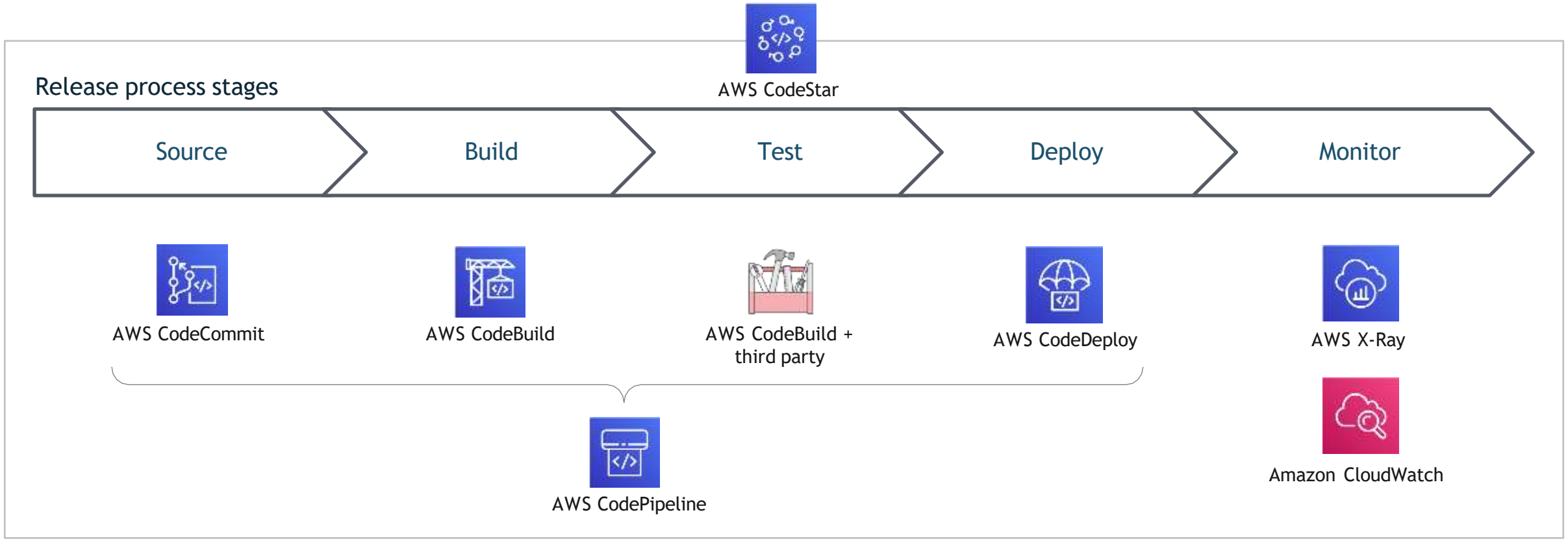
Release process stages



Continuous deployment goals

1. Automatically deploy new changes to staging environments for testing
2. Deploy to production safely without impacting customers
3. Deliver to customers faster: Increase deployment frequency, and reduce change lead time and change failure rate

AWS Code Services



AWS Code Services

Release process stages

Source | Build | Test | Deploy | Monitor

AWS CodeCommit

AWS CodeBuild

AWS CodeBuild + third party

MICRO FOCUS Tricentis

GENYMOTION

Available in aws marketplace

AWS CodeDeploy

AWS X-Ray

Amazon CloudWatch

AWS CodeDeploy

- Automates code deployments to any instance and Lambda
- Handles the complexity of updating your applications
- Avoids **downtime** during **application deployment**
- Rolls back automatically if **failure** detected
- Deploys to Amazon EC2, Lambda, or on-premises **servers**

AWS CodeDeploy: EC2 deployments

```
version: 0.0
os: linux
files:
  - source: /
    destination: /var/www/html
permissions:
  - object: /var/www/html
    pattern: “*.html”
    owner: root
    group: root
    mode: 755
hooks:
  ApplicationStop:
  -   location: scripts/deregister_from_elb.sh
  BeforeInstall:
  -   location: scripts/install_dependencies.sh
  ApplicationStart:
  -   location: scripts/start_httpd.sh
  ValidateService:
    - location: scripts/test_site.sh
    - location: scripts/register_with_elb.sh
```

- Send application files to one directory and configuration files to another
- Set specific permissions on specific directories & files
- Remove/add instance to ELB
- Install dependency packages
- Start Apache
- Confirm successful deploy
- More!

AWS CodeDeploy: Lambda deployments

- Shifts traffic using Lambda function weighted aliases
- Choose canary (“shift 10% of traffic for 10 minutes, then shift rest”) or linear (“shift 10% more traffic every 10 minutes”)
- Validation “hooks” enable testing at each stage of the deployment
- Fast rollback in seconds if case of hook failure or CloudWatch alarms
- Monitor deployment status and history via console, API, Amazon Simple Notification Service (Amazon SNS), and CloudWatch Events

AWS CodeDeploy: Lambda deployments

Enable in your serverless application template

```
Resources:
  GetFunction:
    Type: AWS::Serverless::Function
    Properties:
      DeploymentPreference:
        Type: Canary10Percent10Minutes
        Alarms:
          - !Ref ErrorsAlarm
        Hooks:
          PreTraffic: !Ref PreTrafficHook
```

AWS CodeDeploy: Lambda canary deployment

AWS CodeDeploy

AWS CodeDeploy now automates blue/green deployments to AWS Fargate and Amazon Elastic Container Service (ECS)

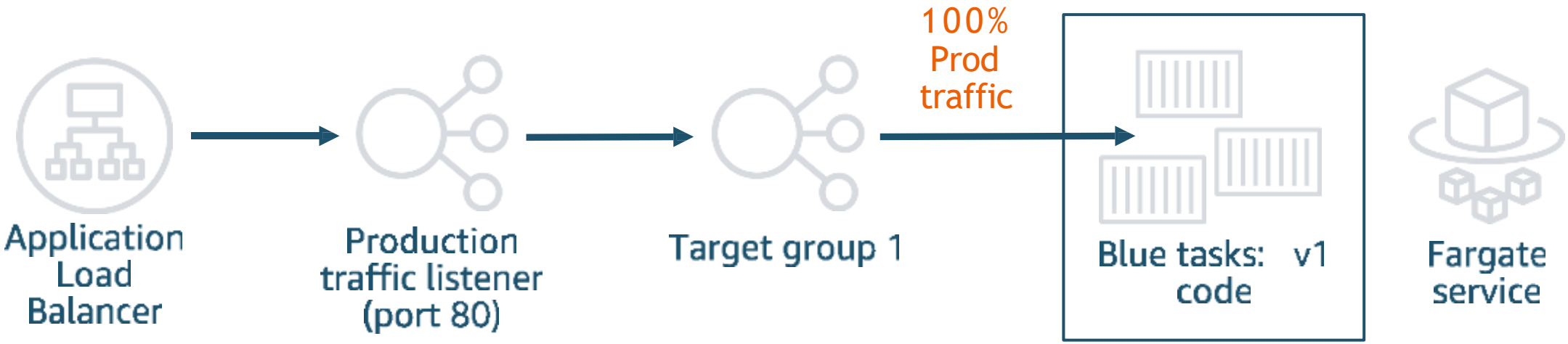
AWS CodeDeploy: ECS blue/green deployments

- Provisions “green” tasks, then flips traffic at the load balancer
- Validation “hooks” enable testing at each stage of the deployment
- Fast rollback to “blue” tasks in seconds in the event of hook failure or CloudWatch alarms
- Monitor deployment status and history via console, API, Amazon SNS, and CloudWatch Events
- Use “CodeDeploy-ECS” deploy action in CodePipeline or “aws ecs deploy” command in Jenkins

AWS CodeDeploy: ECS AppSpec

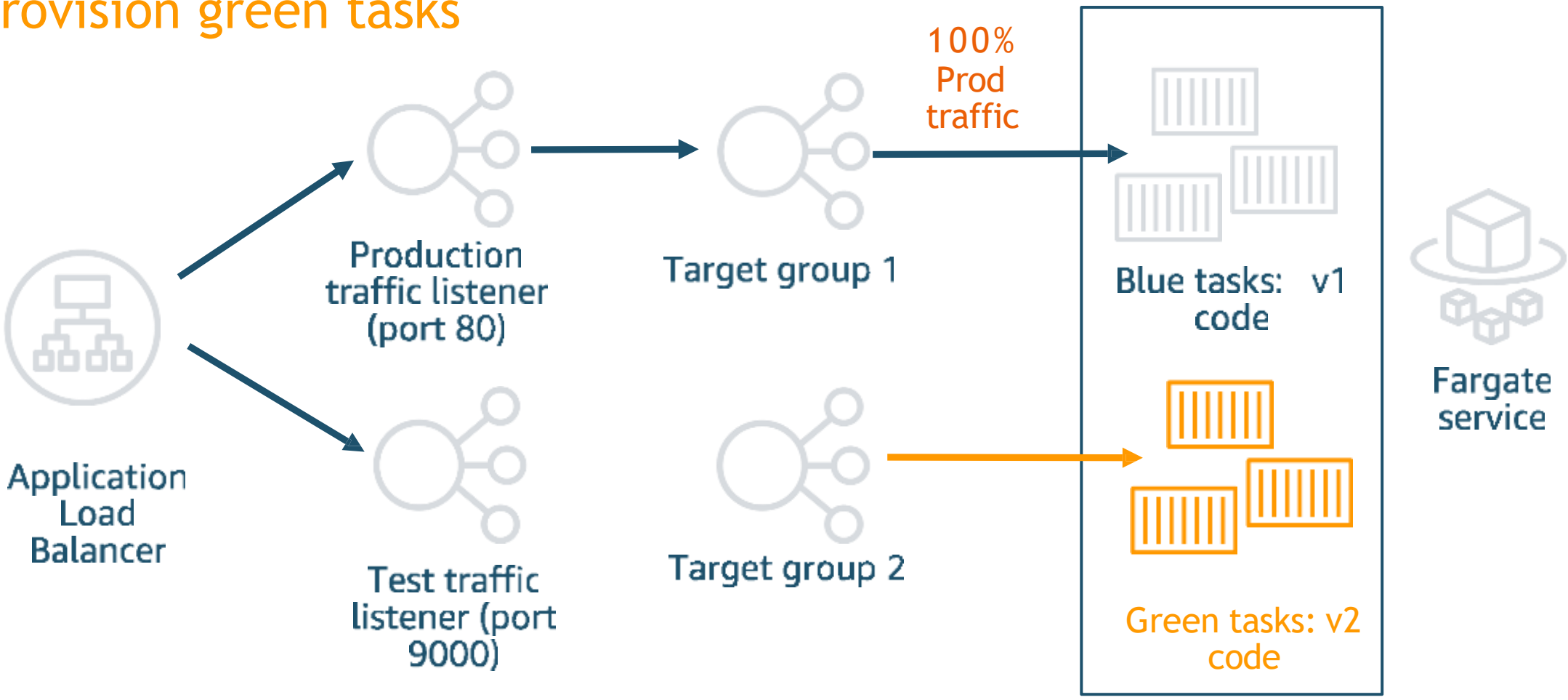
```
version: 1.0
Resources:
    - TargetService:
        Type: AWS::ECS::Service
        Properties:
            - TaskDefinition: "my_task_definition:8"
              LoadBalancerInfos:
                    - ContainerName: "SampleApp"
                      ContainerPort: 80


Hooks:
    - BeforeInstall: "LambdaFunctionToExecuteAnythingBeforeNewRevisionInstalltion"
    - AfterInstall: "LambdaFunctionToExecuteAnythingAfterNewRevisionInstallation"
    - AfterAllowTestTraffic: "LambdaFunctionToValidateAfterTestTrafficShift"
    - BeforeAllowTraffic: "LambdaFunctionToValidateBeforeTrafficShift"
    - AfterAllowTraffic: "LambdaFunctionToValidateAfterTrafficShift"
```

AWS CodeDeploy: ECS blue/green deployment



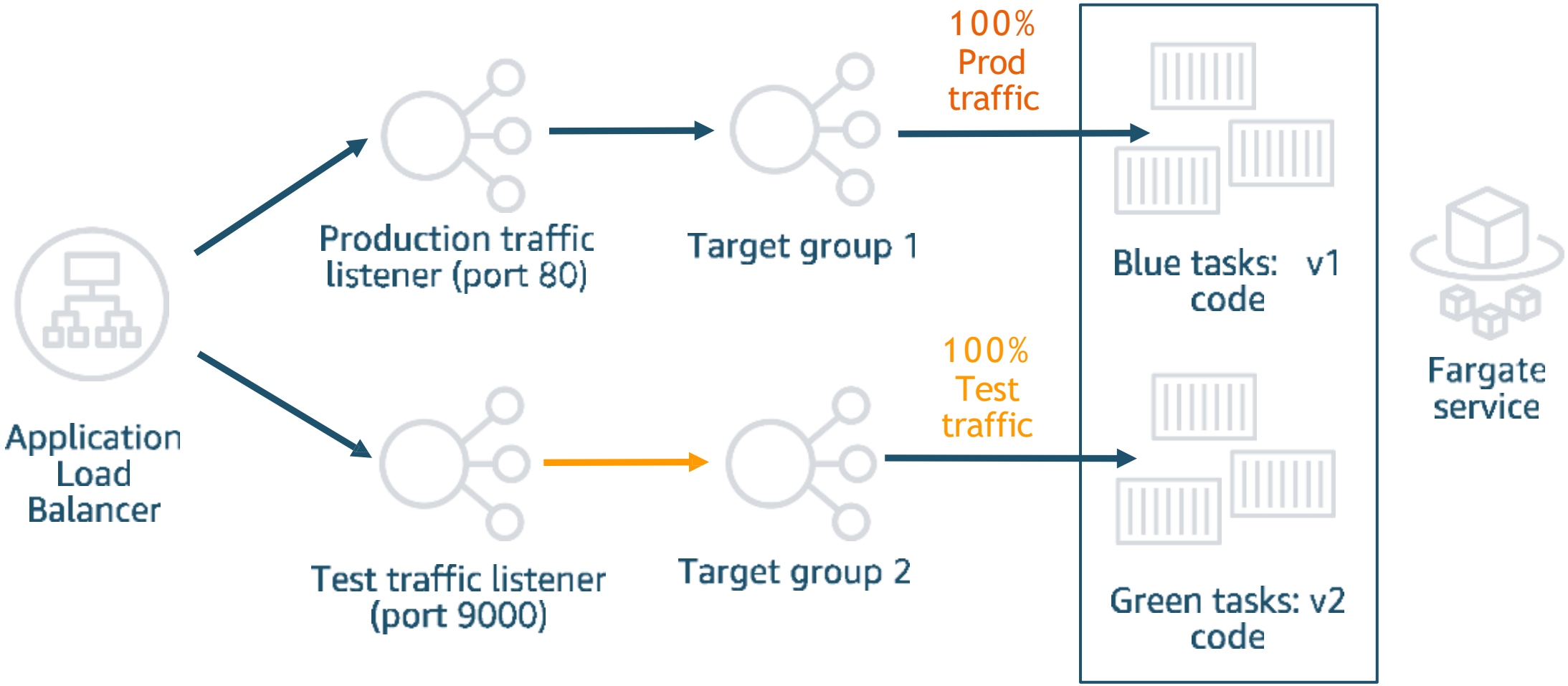
AWS CodeDeploy: ECS blue/green deployment

Provision green tasks



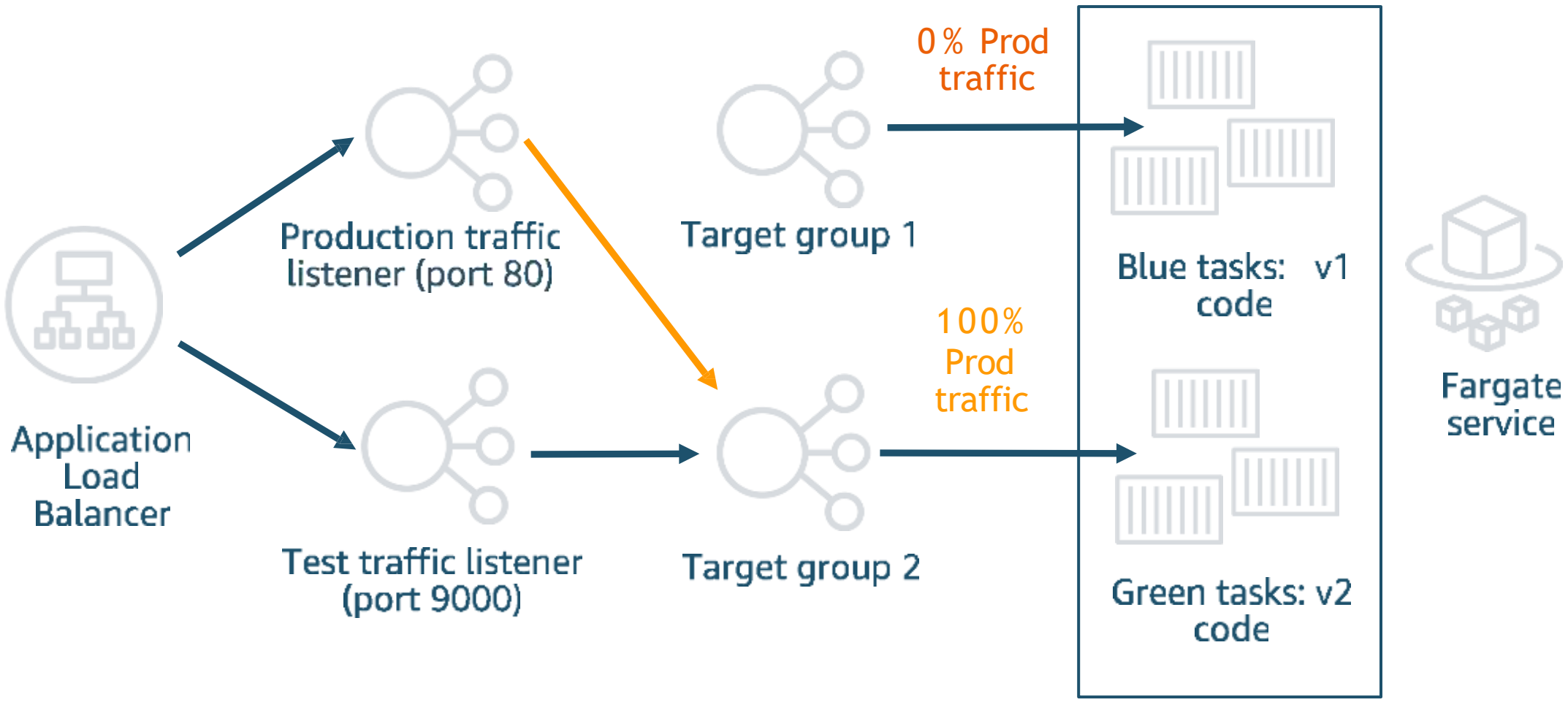
AWS CodeDeploy: ECS blue/green deployment

Run hook against test endpoint before green tasks receive prod traffic



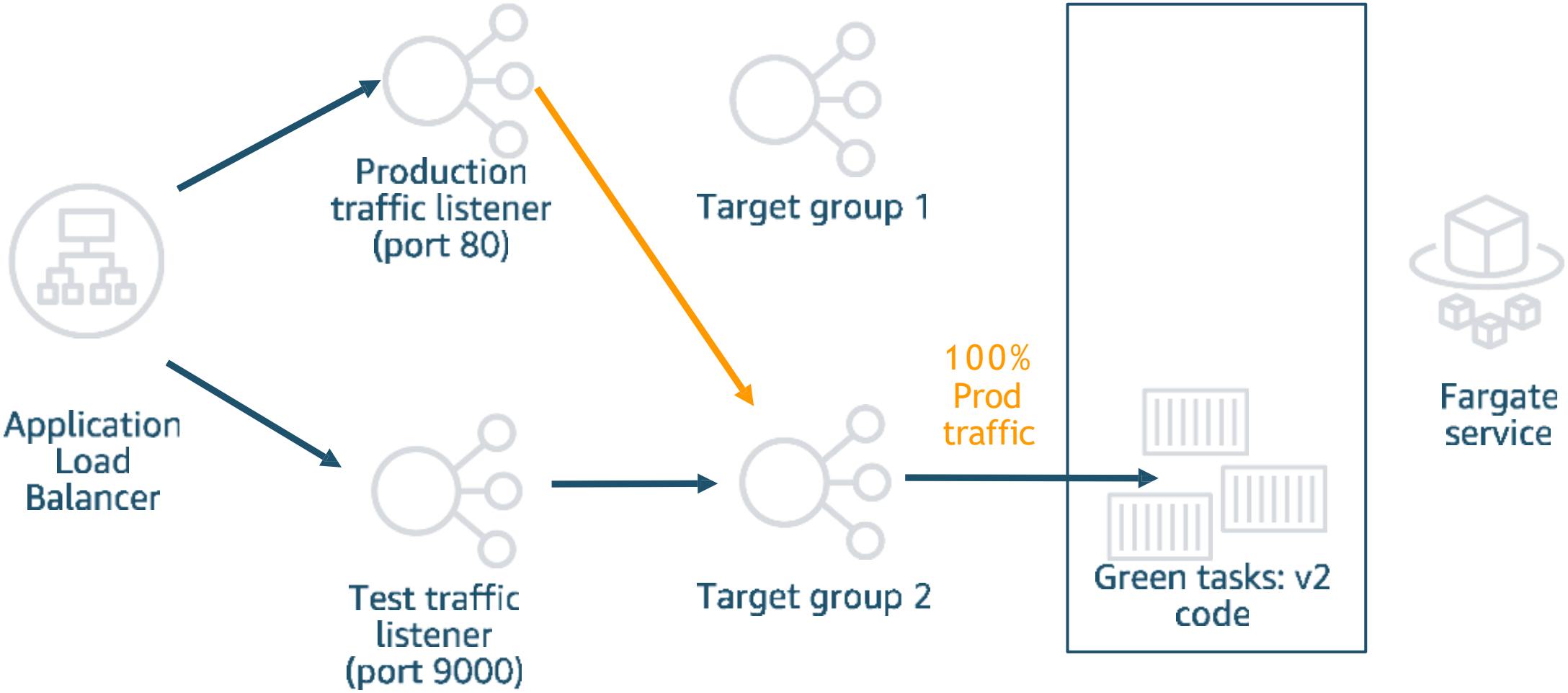
AWS CodeDeploy: ECS blue/green deployment

Flip traffic to green tasks, rollback in case of alarm



AWS CodeDeploy: ECS blue/green deployment

Drain blue tasks



Container image tagging for deployments

- Docker tags are resolved when each container starts, not just during deployments
- Deploying “latest” or “prod” can result in untested code in production after a scale-out event
- Use unique “immutable” tags for deployments

Container image tagging for deployments

Service scales up, launching new tasks

Container image tagging for deployments

Deploy using immutable tags

```
{
    "name": "sample-app",
    "image": "amazon/amazon-ecs-
  sample@sha256:3e39d933b1d948c92309bb583b5a1f3d28f0119e1551ca1fe538ba414a41af48d"
}
```

SHA256 Digest

Build ID

```
{
    "name": "sample-app",
    "image": "amazon/amazon-ecs-sample:build-b2085490-359f-4eaf-8970-6d1e26c354f0"
}
```

Container image tagging for deployments

Compute immutable tags during build

SHA256 Digest

```
export IMAGE_URI=`docker inspect --format='{{index .RepoDigests 0}}'
  my_image:$IMAGE_TAG
```

Example result:

```
amazon/amazon-ecs-sample@sha256:3e39d933b...
```

Build ID

```
export IMAGE_TAG=build-`echo $CODEBUILD_BUILD_ID | awk –F":" ‘{print $2}'`
```

Example result:

```
build-b2085490-359f-4eaf-8970-6d1e26c354f0
```

Container image tagging for deployments

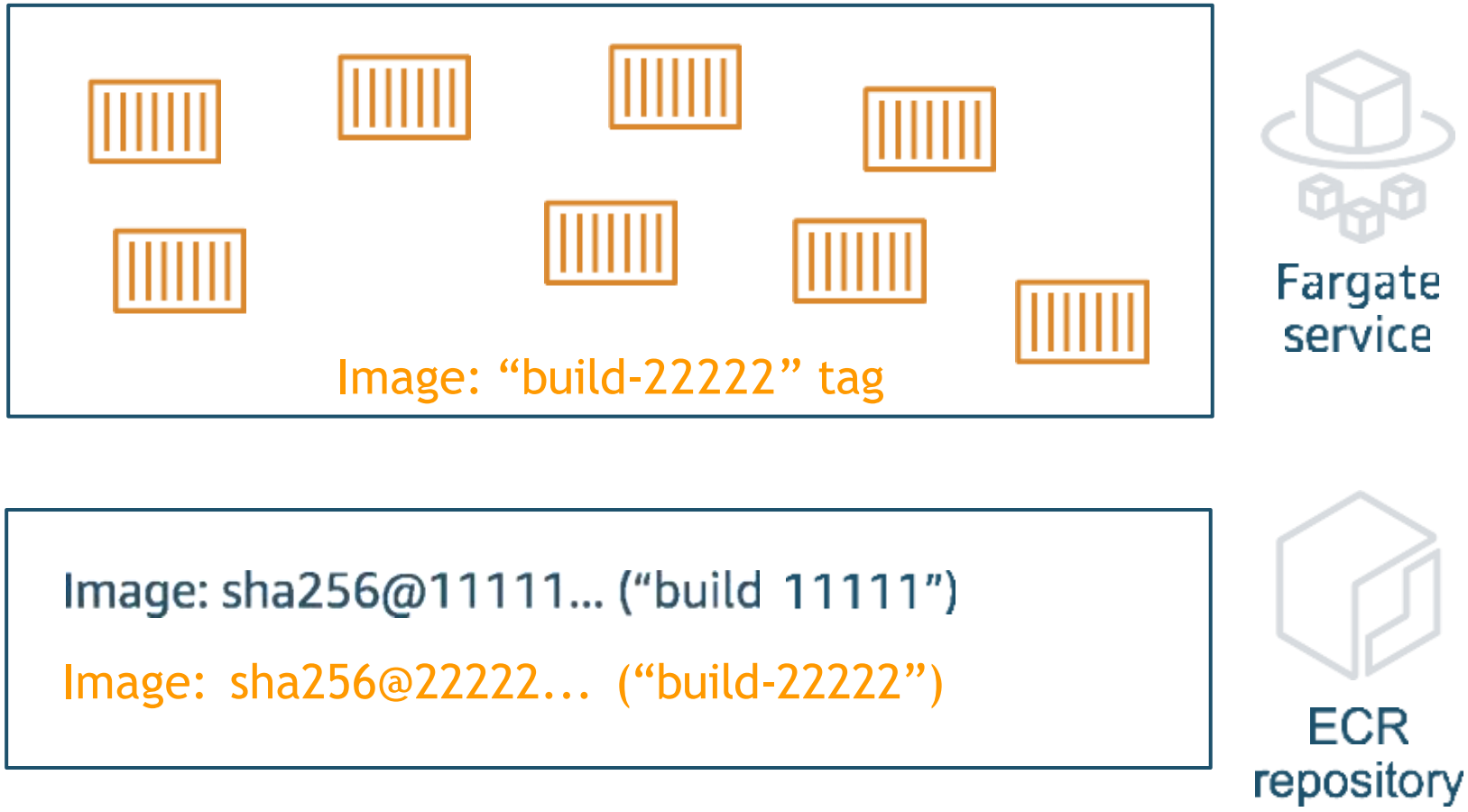
Build pushes new image tagged with new build ID

Container image tagging for deployments

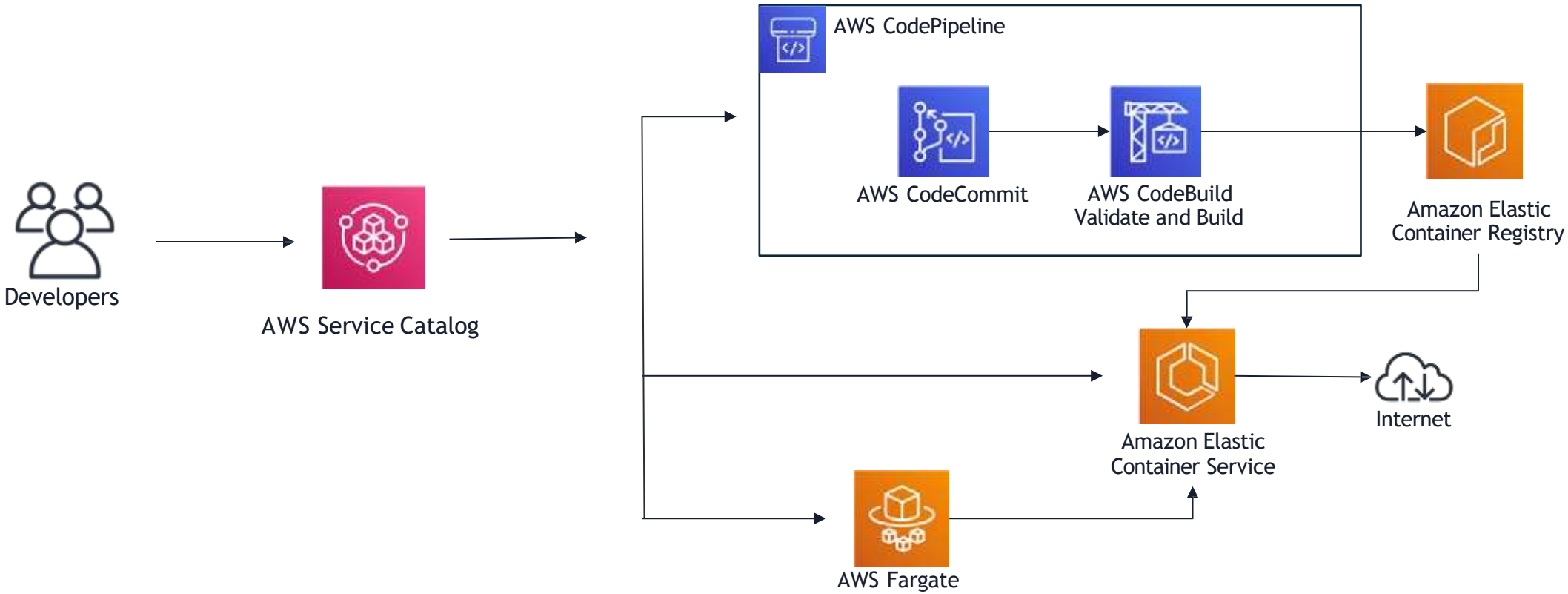
Service scales up, launching new tasks

Container image tagging for deployments

Deployment updates service's task definition, replacing tasks



Build self-service DevOps pipeline to deploy containerized applications



Automate everything

- Defined as code
- Checked into a version control repository, such as AWS CodeCommit
- Able to allow for extensibility through other AWS services or 3rd party tools
- Able to provide FAST feedback on the success and failure of pipeline executions

Automate your security testing…

- Integrate it into your pipelines
- If your pipeline produces AMIs, Docker containers, etc.… scan them with tools like Inspector, Clair, and Twistlock
- If the tool has an API, you can use a custom Lambda action in CodePipeline to trigger it
- If a security test fails, pipeline stops, code doesn’t make it to production

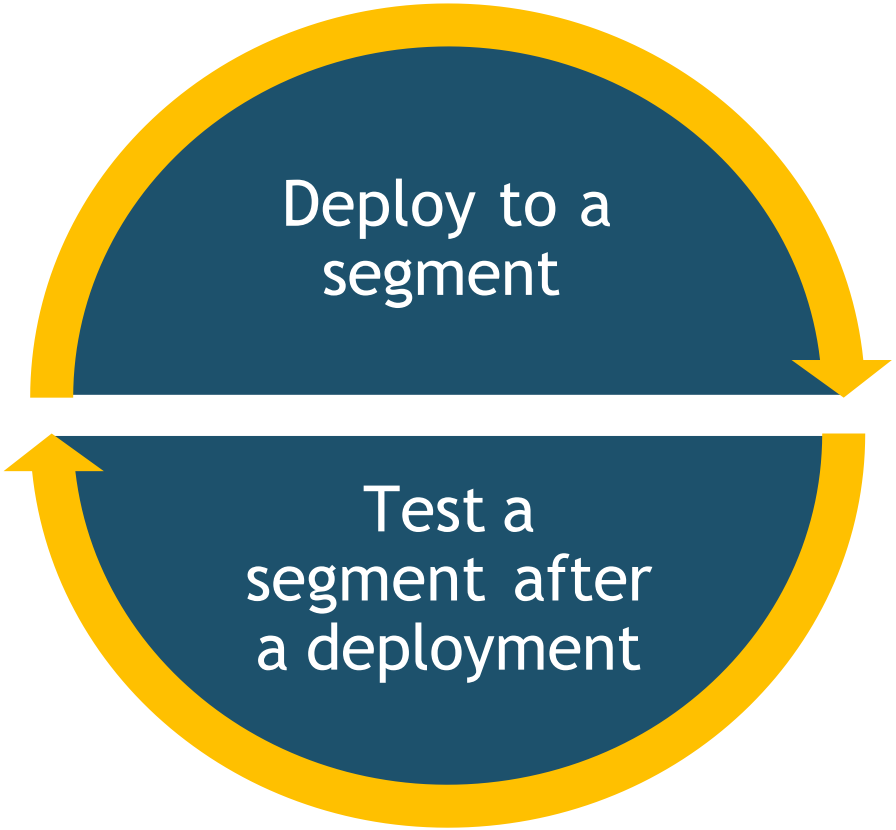
Manage deployment health

- Builds on top of our foundation of automation
- Purpose built to verify that a service is working after a new deployment
- Helps to avoid needing to do this manually

Lower deployment risk by segmenting – one at a time

Option 1

Break production into multiple segments



Lower deployment risk by segmenting – one at a time

Option 2

1. Break production into multiple segments
2. Deploy to a segment
3. Test a segment after a deployment
4. Repeat 2 & 3 until done

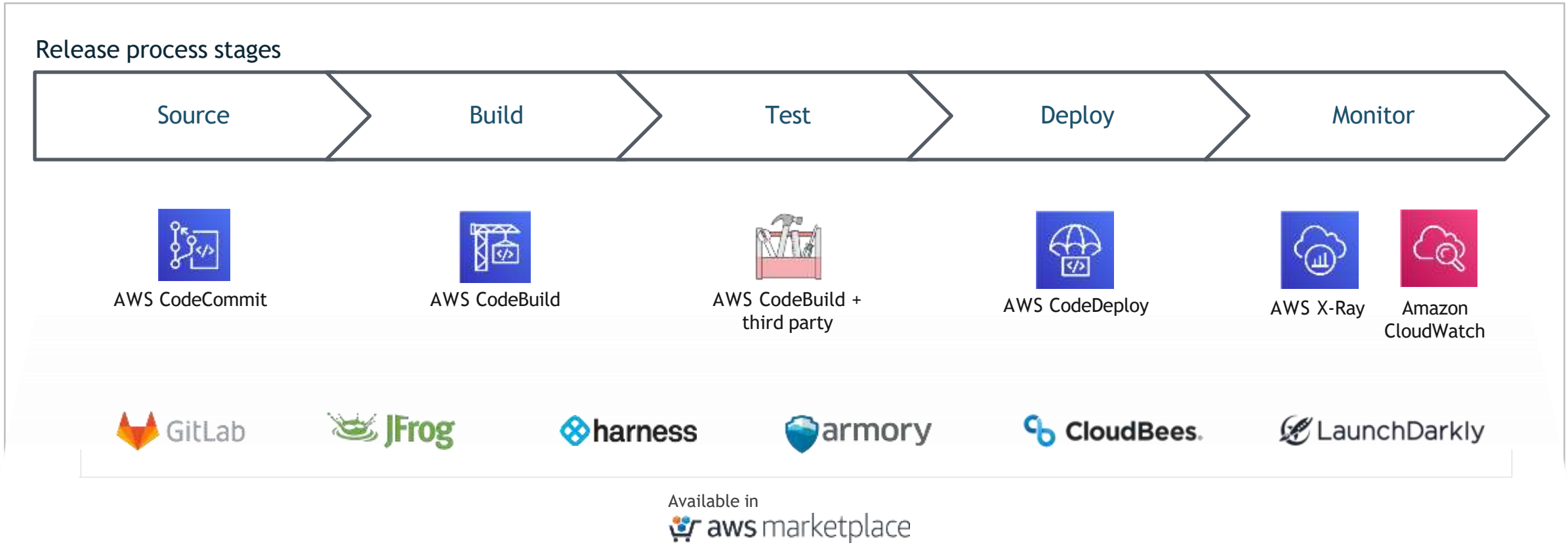
Deploy to multiple regions - Cross-Region deployments

- Allows you to deploy to multiple regions from a single pipeline
- Enables you to achieve lower latency and greater availability

Implement pipeline governance - block non-compliant pipelines

- Use Config Rules and CloudWatch Events to automatically remediate non-compliant pipelines
- Add common action to all pipelines
- Provide developers a best practices pipeline to start from with CloudFormation Templates or Service Catalog
- Use AWS CDK to create pipelines with your best practices embedded in them
- Implement with Approval actions in CodePipeline

AWS Partner Solutions



Move on
Infrastructure as Code