Building Automated Quality Gates into your CI Pipelines

About Me

REPL

Part of **Accenture**

Risci Games

QUALITY BY DESIGN

Designing quality software systems

CRAIG RISI

QUAY FOUR

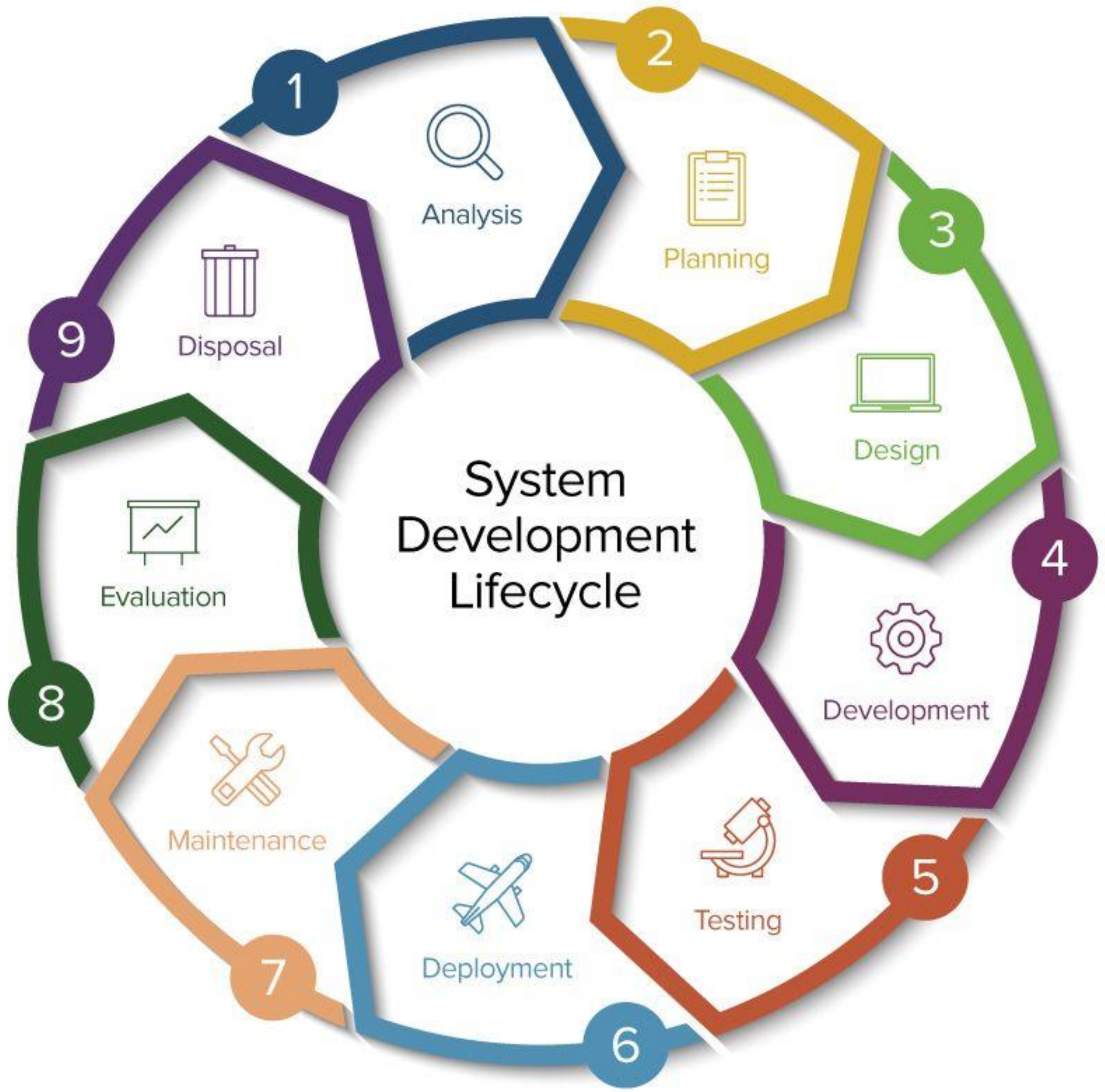
What is a Quality Gate

A change control to ensure the quality of the software at level

Ensures entry and exit criteria can be met

Prevents development from moving to the next stage until certain measures are met

Can be automated to fit a company’s CI/CD needs


System Development Lifecycle
1
Analysis
2
Planning
3
Design
4
Development
5
Testing
6
Deployment
7
Maintenance
8
Evaluation
9
Disposal

Why Quality Gates

MEASURE TO AUTOMATICALLY ENSURE ACCEPTANCE CRITERIA ARE MET

DRIVES SHIFT-LEFT MINDSET

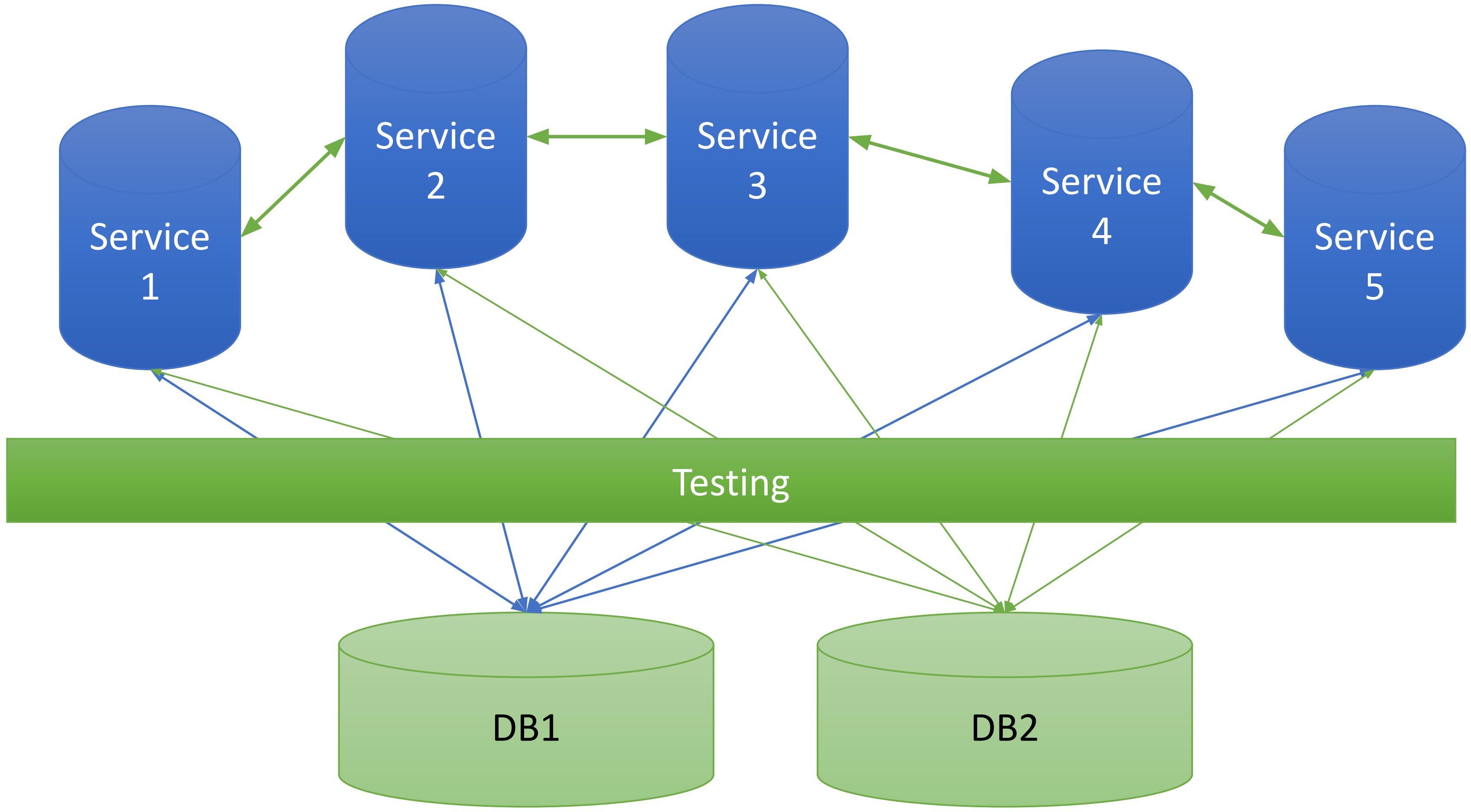
SAFEGUARD AGAINST POOR QUALITY CODE IMPACTING LATER CYCLES

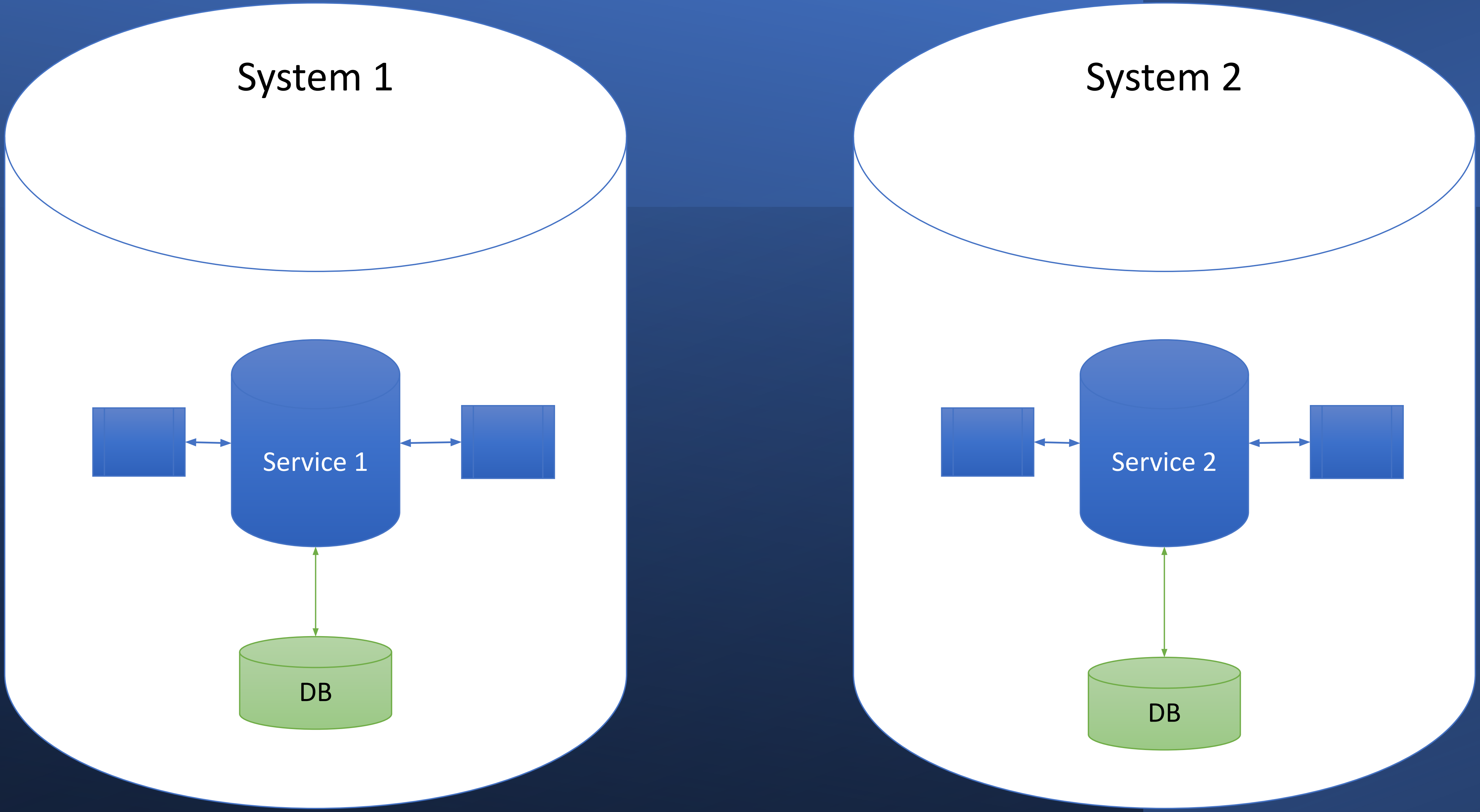
PREVENTS TESTING BEING IMPACTED LATE IN CYCLE

ENSURES PROACTIVE QUALITY

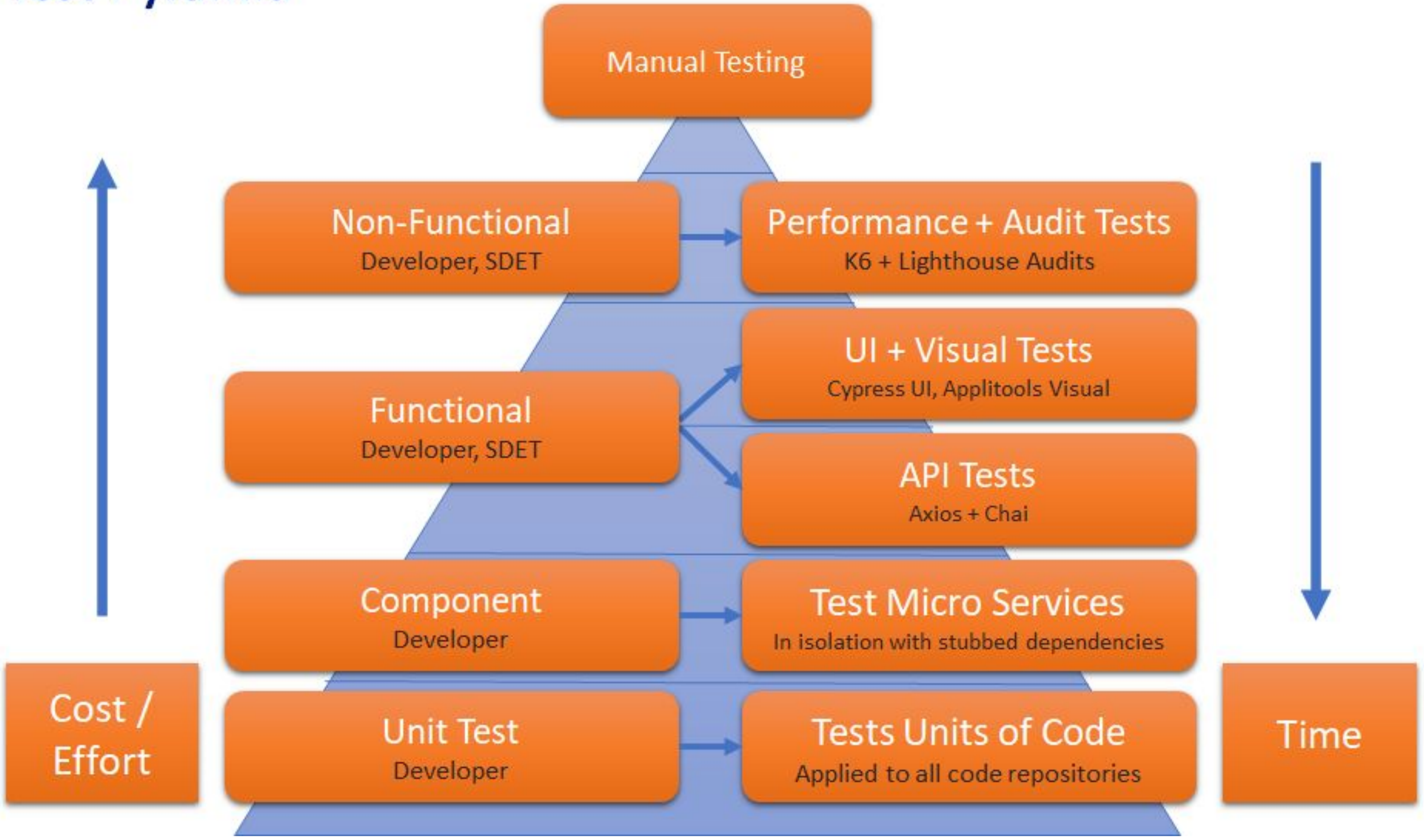
Building blocks for Quality Gates

- Requires well-defined and clear completion criteria
- Testable architecture
- Strong focus on Unit Testing
- Requires automated integration tests
- Works best with small/frequent releases


Service 2
Service 3
Service 4
Service 5
Service 1
Testing
DB1
DB2


System 1
Service 1
DB
System 2
Service 2
DB

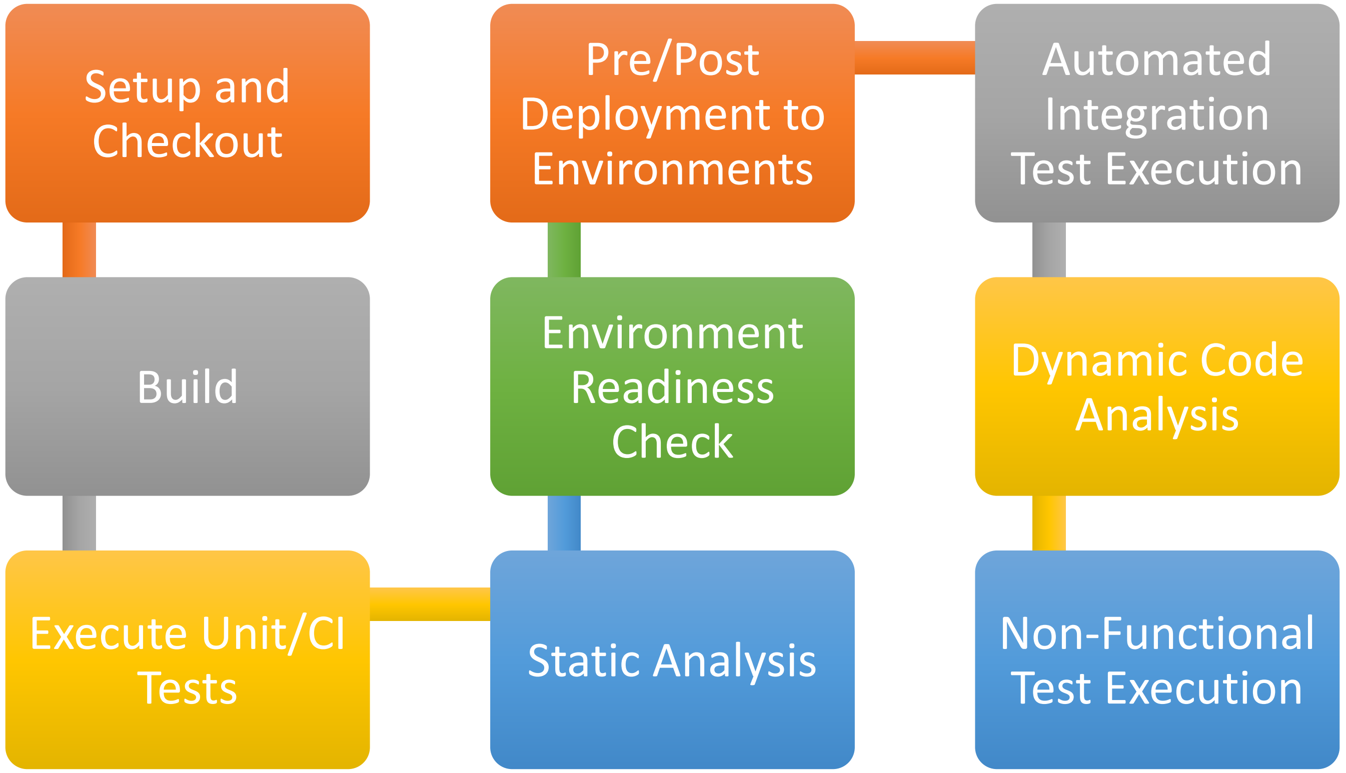
Automation Test Pyramid



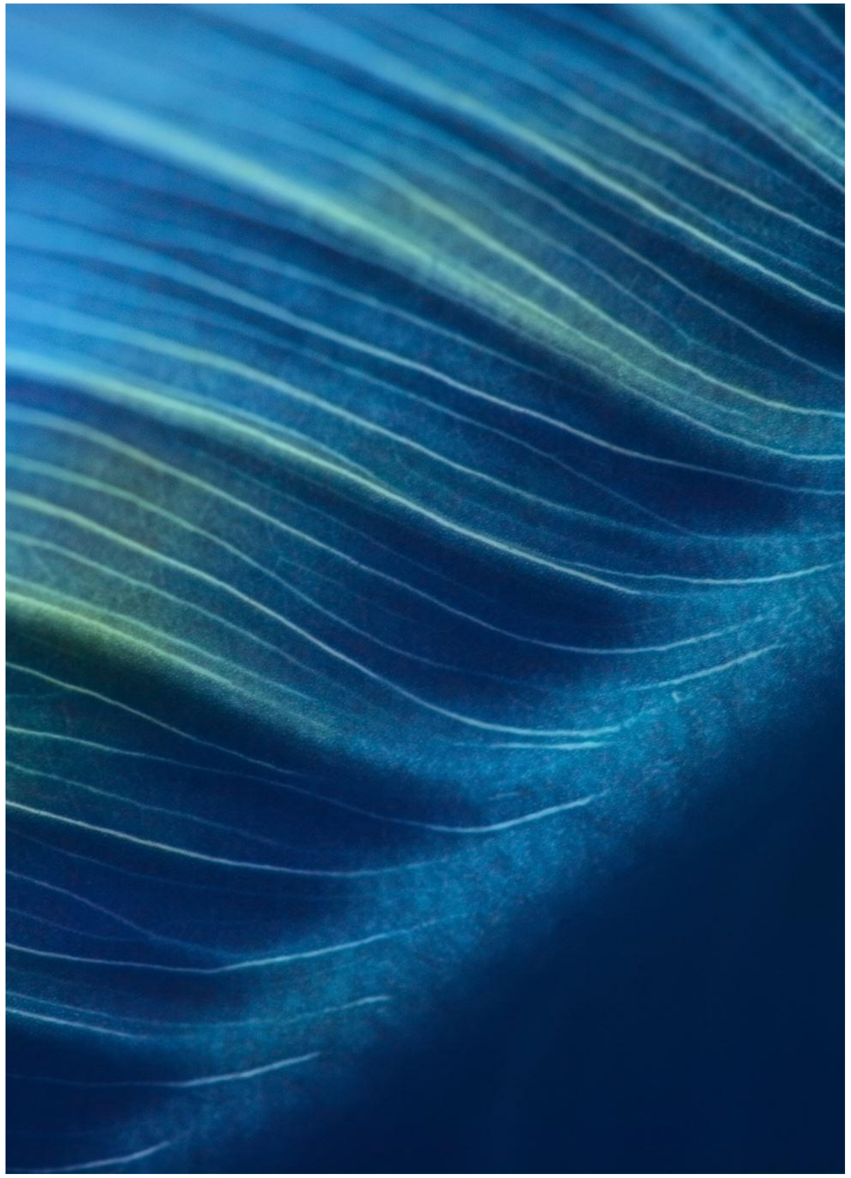
What does a Quality Gate Check

- Build Health
- Infrastructure Health
- Test Results
- Code Coverage
- Security Scans
- Service Performance
- Incident and Issue Management

Types of Quality Gates

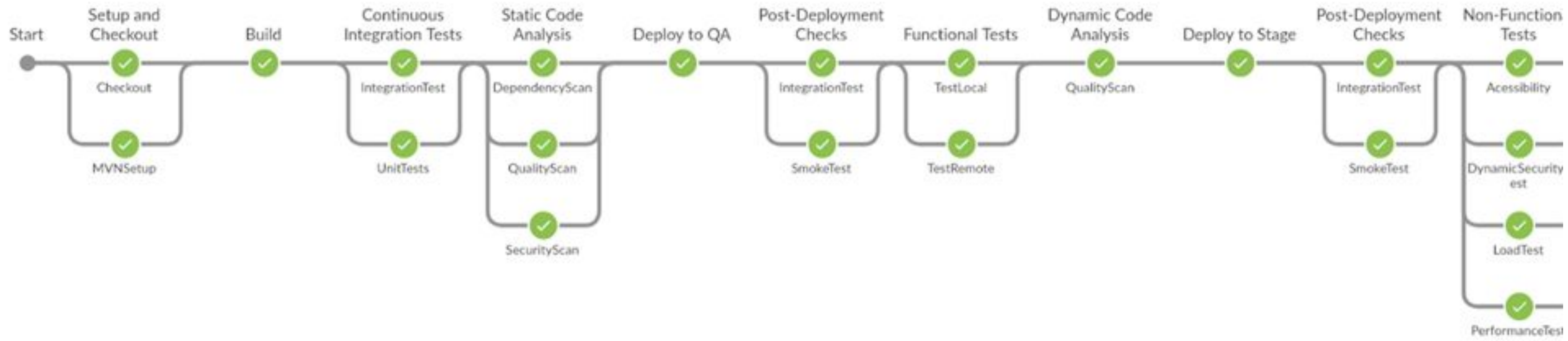


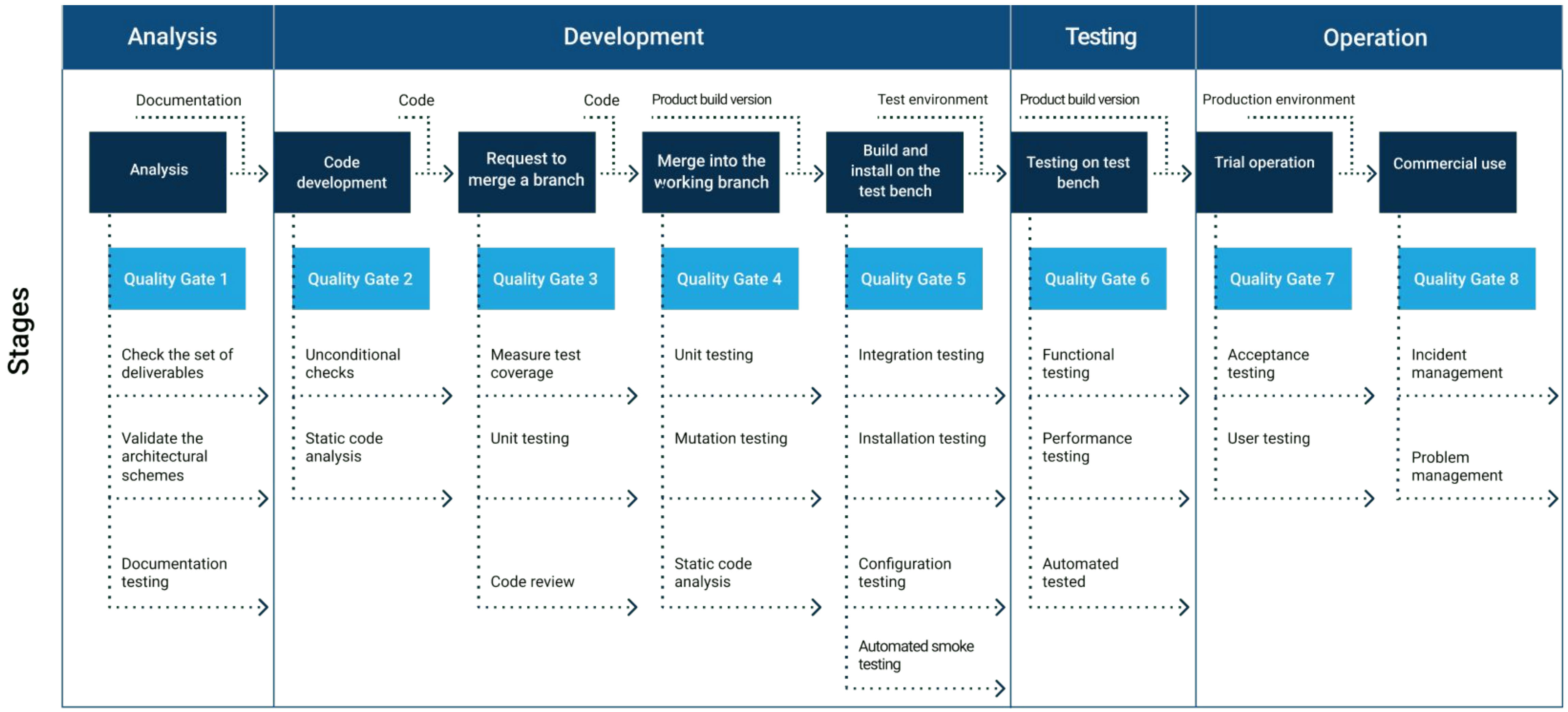
Setup and Checkout
Build
Execute Unit/CI Tests
Static Analysis
Environment Readiness Check
Pre/Post Deployment to Environments
Automated Integration Test Execution
Dynamic Code Analysis
Non-Functional Test Execution



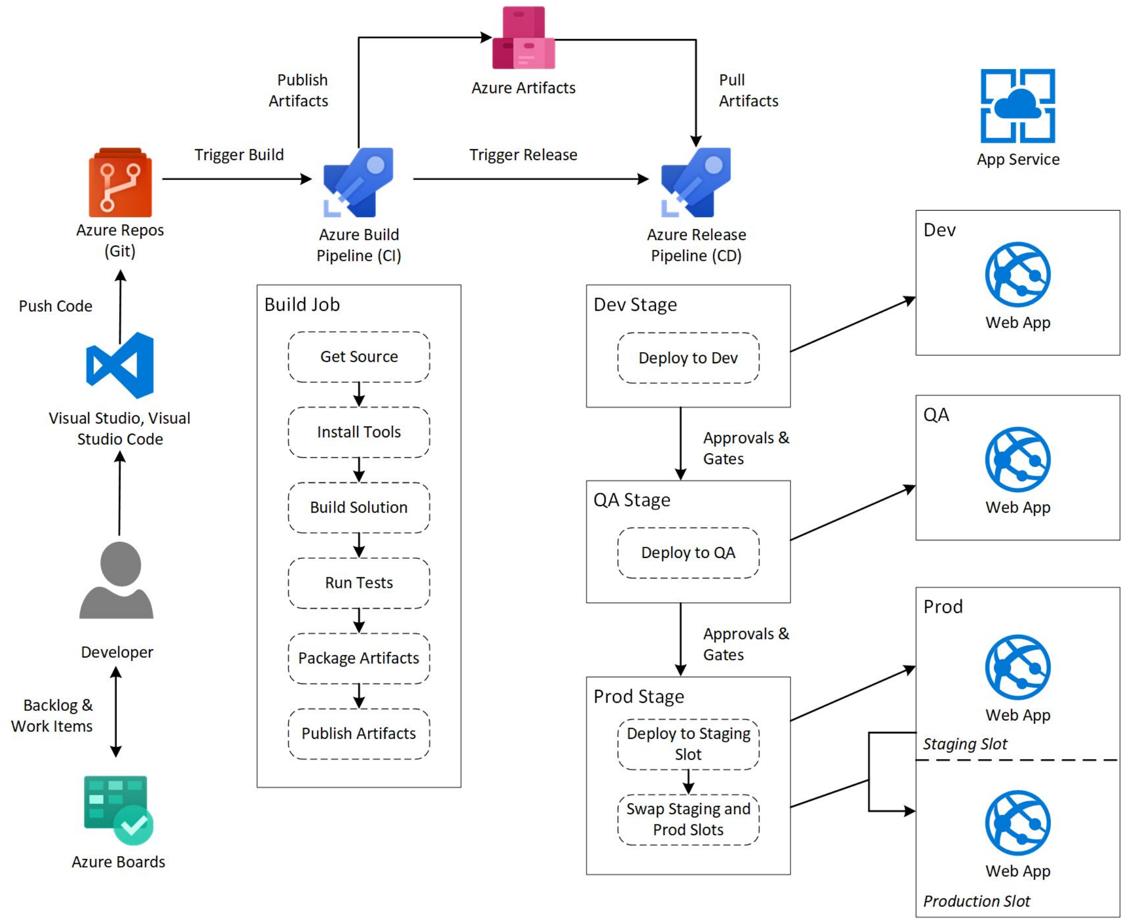
Examples of Quality Checks

- Linting standards need to be met before the code build can be successful.
- 100% successful completion of all tests with a 90% code coverage achieved at a unit testing level.
- Successful completion of scans with 100% code coverage.
- Successful pass of all automated checks

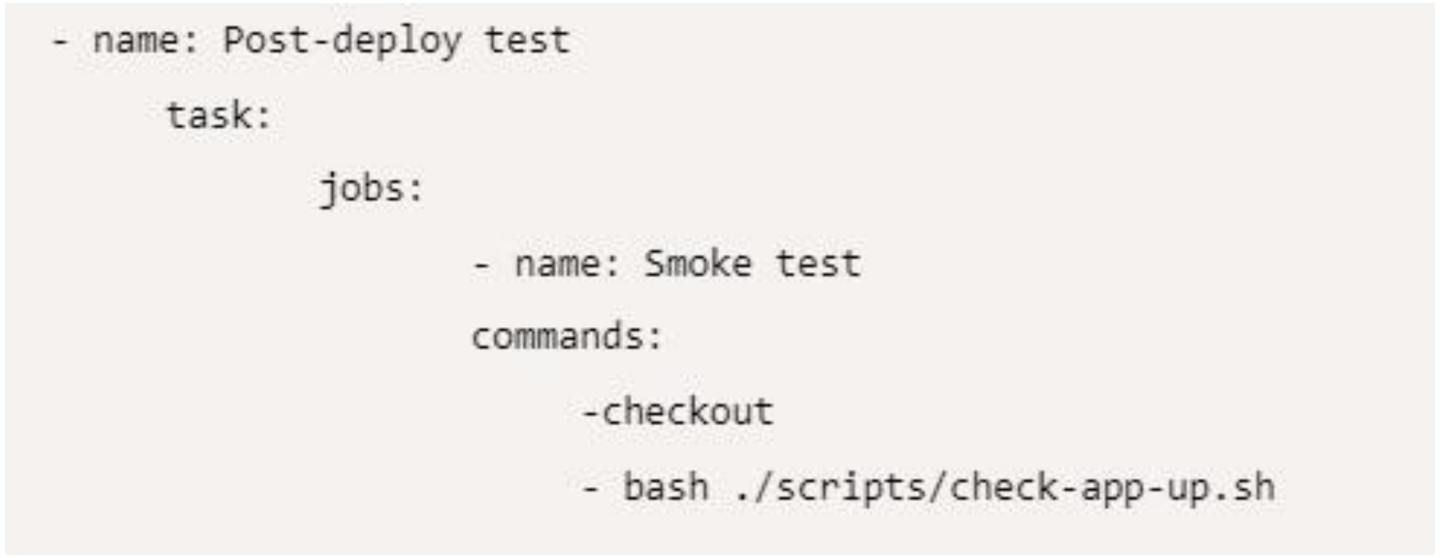

Start
Setup and Checkout
Checkout
MVNSetup
Build
Continuous Integration Tests
IntegrationTest
UnitTests
Static Code Analysis
DependencyScan
QualityScan
SecurityScan
Deploy to QA
Post-Deployment Checks
IntegrationTest
SmokeTest
Functional Tests
TestLocal
TestRemote
Dynamic Code Analysis
QualityScan
Deploy to Stage
Post-Deployment Checks
IntegrationTest
SmokeTest
Non-Function Tests
Acessibility
DynamicSecurity est
LoadTest
PerformanceTest


Stages
Analysis
Development
Testing
Operation
Documentation
Code
Code
Product build version
Test environment
Product build version
Production environment
Analysis
Code development
Request to merge a branch
Merge into the working branch
Build and install on the test bench
Testing on test bench
Trial operation
Commercial use
Quality Gate 1
Quality Gate 2
Quality Gate 3
Quality Gate 4
Quality Gate 5
Quality Gate 6
Quality Gate 7
Quality Gate 8
Check the set of deliverables
Validate the architectural schemes
Documentation testing
Unconditional checks
Static code analysis
Measure test coverage
Unit testing
Code review
Unit testing
Mutation testing
Static code analysis
Integration testing
Installation testing
Configuration testing
Automated smoke testing
Functional testing
Performance testing
Automated tested
Acceptance testing
User testing
Incident management
Problem management

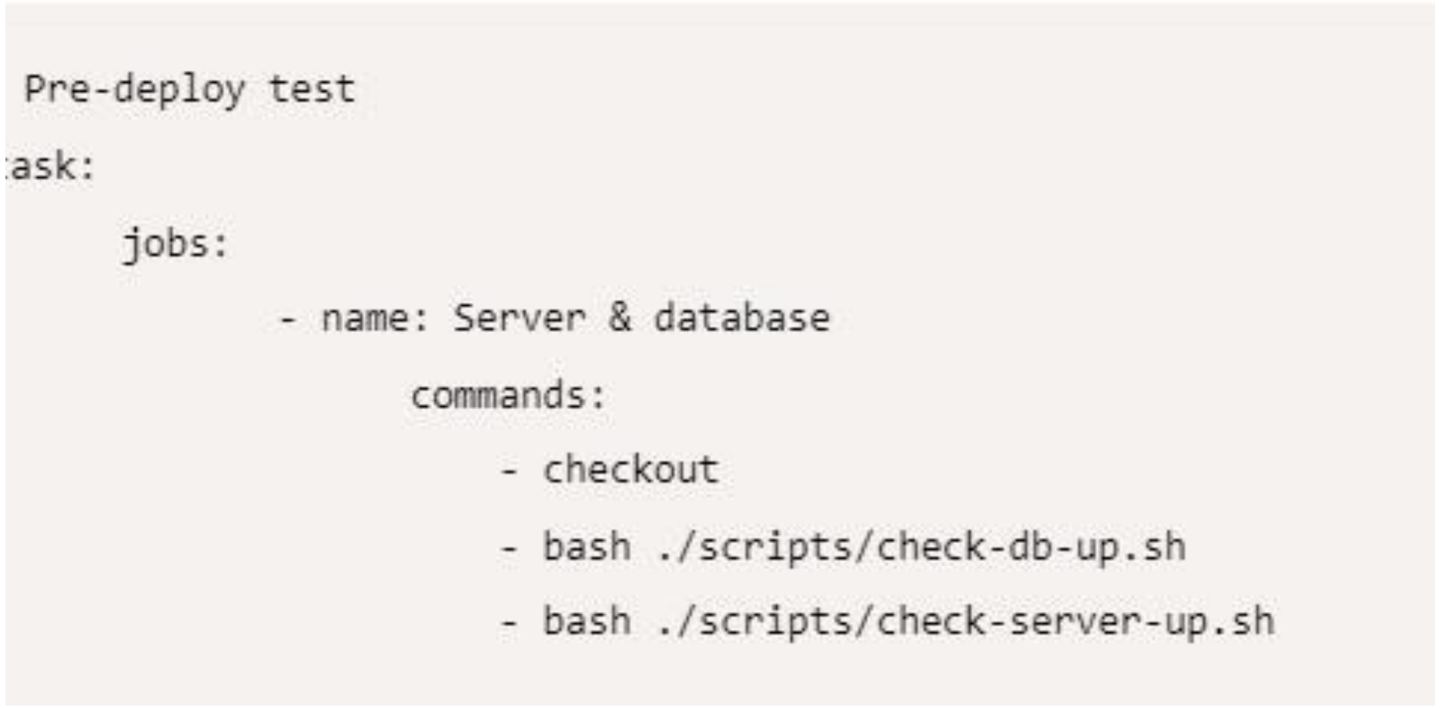
How To Build Quality Gates



Checking Environments Before/After Deployment

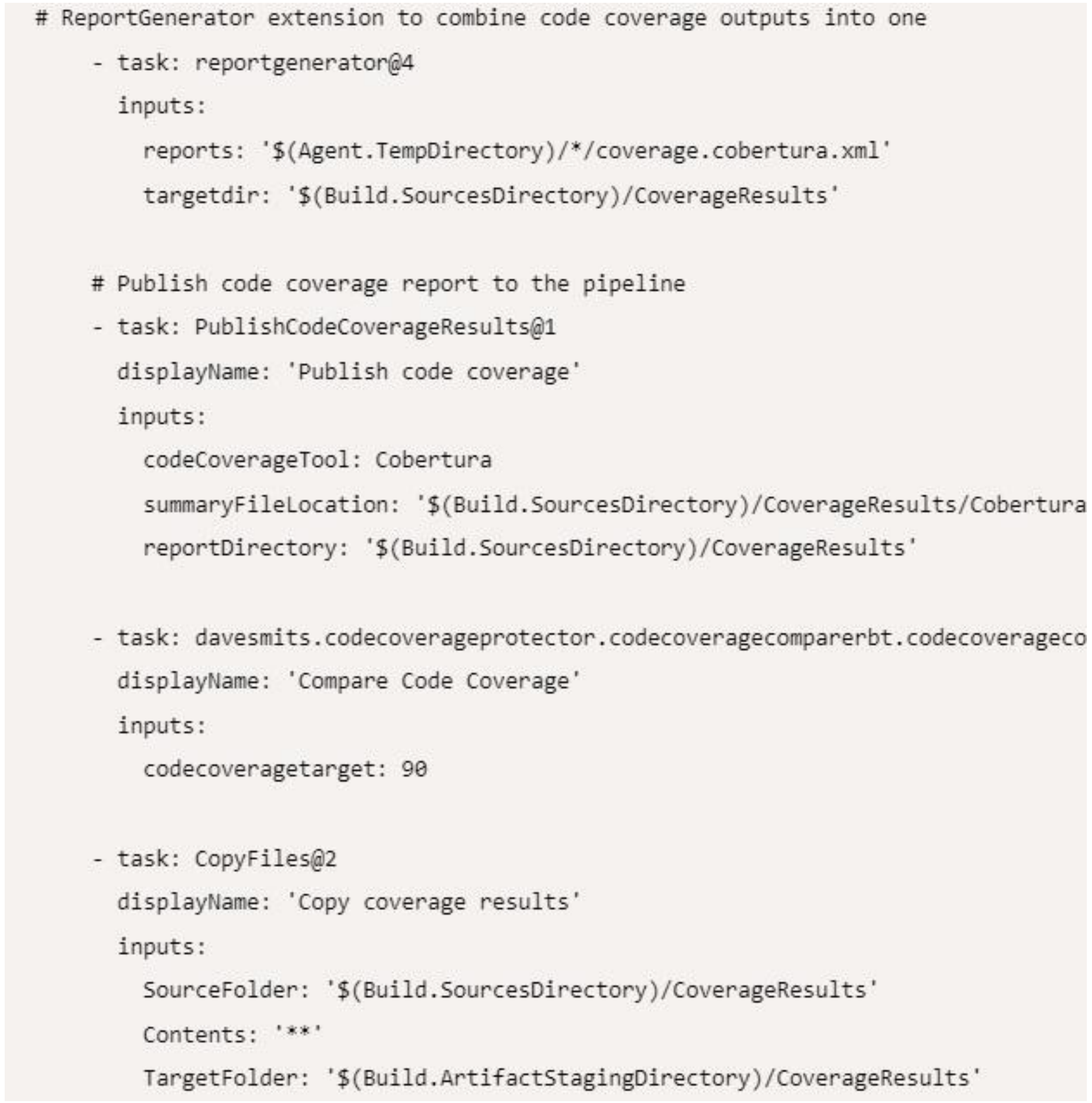


```
- name: Post-deploy test
    task:
        jobs:
            - name: Smoke test
            commands:
               -checkout
               - bash ./scripts/check-app-up.sh
```



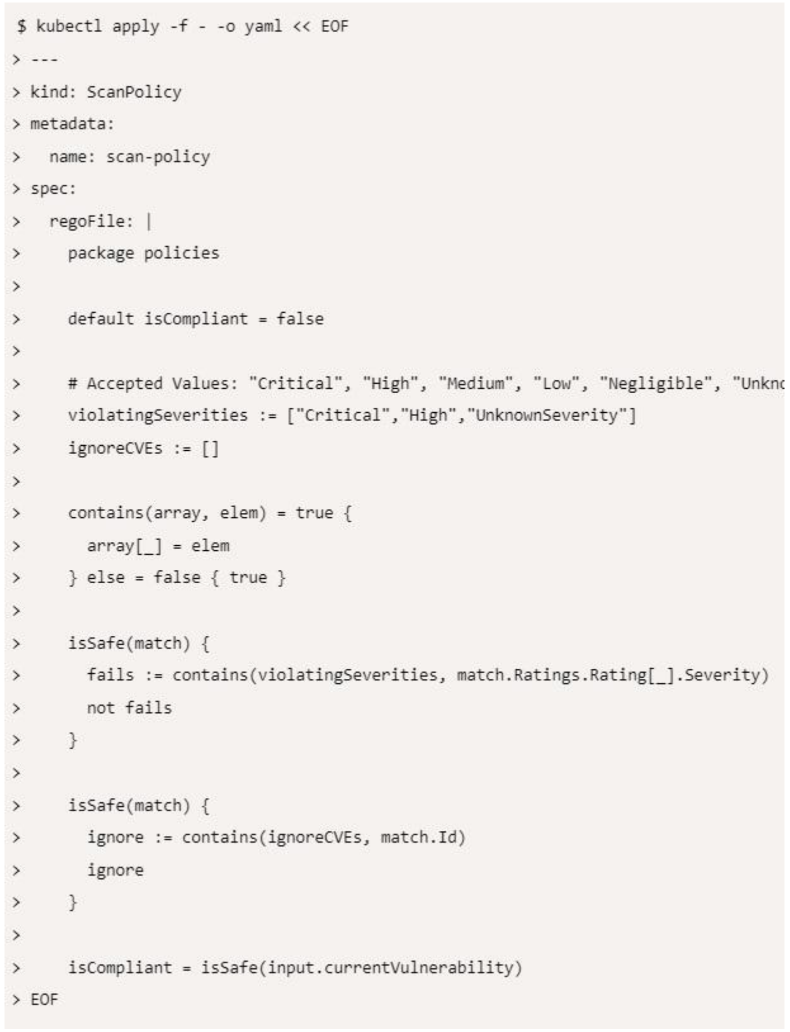
```
Pre-deploy test
ask:
   jobs:
       - name: Server & database
           commands:
              - checkout
              - bash ./scripts/check-db-up.sh
              - bash ./scripts/check-server-up.sh
```

Measuring Code Coverage and Pass Rates

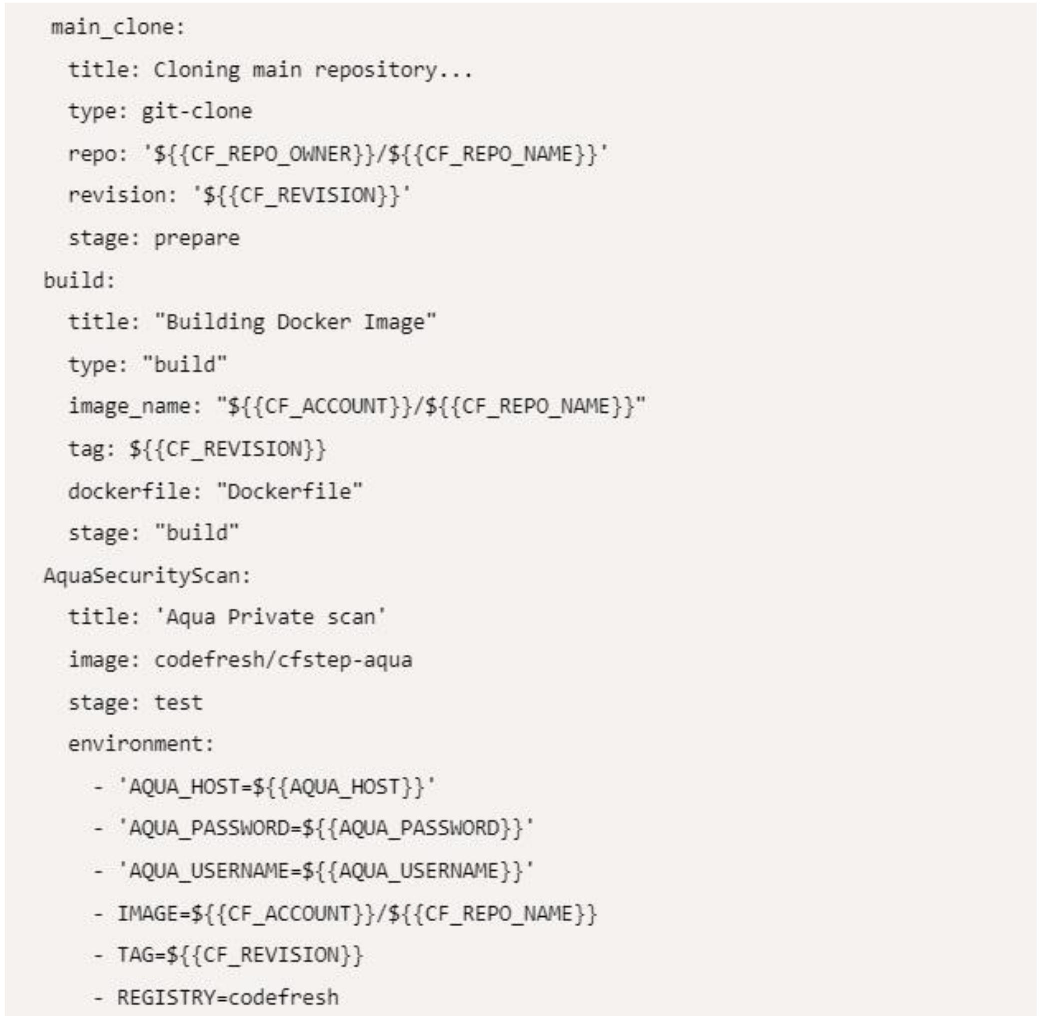


```
# ReportGenerator extension to combine code coverage outputs into one
  - task: reportgenerator@4
   inputs:
    reports: '$(Agent.TempDirectory)/*/coverage.cobertura.xml'
    targetdir: '$(Build.SourcesDirectory)/CoverageResults'

  # Publish code coverage report to the pipeline
  - task: PublishCodeCoverageResults@1
   displayName: 'Publish code coverage'
   inputs:
    codeCoverageTool: Cobertura
    summaryFileLocation: '$(Build.SourcesDirectory)/CoverageResults/Cobertura
    reportDirectory: '$(Build.SourcesDirectory)/CoverageResults'

  - task: davesmits.codecoverageprotector.codecoveragecomparerbt.codecoverageco
   displayName: 'Compare Code Coverage'
   inputs:
    codecoveragetarget: 90

  - task: CopyFiles@2
   displayName: 'Copy coverage results'
   inputs:
    SourceFolder: '$(Build.SourcesDirectory)/CoverageResults'
    Contents: '**'
    TargetFolder: '$(Build.ArtifactStagingDirectory)/CoverageResults'
```

Ensure Successful Scan Results



```
$ kubectl apply -f - -o yaml << EOF
> ---
> kind: ScanPolicy
> metadata:
>   name: scan-policy
> spec:
>   regoFile: |
>     package policies
>
>     default isCompliant = false
>
>     # Accepted Values: "Critical", "High", "Medium", "Low", "Negligible", "Unkn
>     violatingSeverities := ["Critical","High","UnknownSeverity"]
>     ignoreCVEs := []
>
>     contains(array, elem) = true {
>       array[_] = elem
>     } else = false { true }
>
>     isSafe(match) {
>       fails := contains(violatingSeverities, match.Ratings.Rating[_].Severity)
>       not fails
>     }
>
>     isSafe(match) {
>       ignore := contains(ignoreCVEs, match.Id)
>       ignore
>     }
>
>     isCompliant = isSafe(input.currentVulnerability)
> EOF
```



```
 main_clone:
   title: Cloning main repository...
   type: git-clone
   repo: '${{CF_REPO_OWNER}}/${{CF_REPO_NAME}}'
   revision: '${{CF_REVISION}}'
   stage: prepare
build:
   title: "Building Docker Image"
   type: "build"
   image_name: "${{CF_ACCOUNT}}/${{CF_REPO_NAME}}"
   tag: ${{CF_REVISION}}
   dockerfile: "Dockerfile"
   stage: "build"
AquaSecurityScan:
   title: 'Aqua Private scan'
   image: codefresh/cfstep-aqua
   stage: test
   environment:
     - 'AQUA_HOST=${{AQUA_HOST}}'
     - 'AQUA_PASSWORD=${{AQUA_PASSWORD}}'
     - 'AQUA_USERNAME=${{AQUA_USERNAME}}'
     - IMAGE=${{CF_ACCOUNT}}/${{CF_REPO_NAME}}
     - TAG=${{CF_REVISION}}
     - REGISTRY=codefresh
```

Observability

- Collect data from multiple sources
- Store data in a central location
- Use APIs to automate data gathering
- Use logging and monitoring frameworks
- Use data visualization tools
- Set up alerting
- Keep track of data retention policies
- Continuously monitor and optimize

Q&A