AUTOMATION CI/CD IN DEVOPS

FROM COMMIT TO PRODUCTION

WHY CI/CD?

PROBLEMS WITHOUT CI/CD

01

Manual builds → "works for me" errors

02

Late feedback, slow releases

03

Risky "large" deployments

BENEFITS OF CI/CD

01

Frequent small changes, lower risk

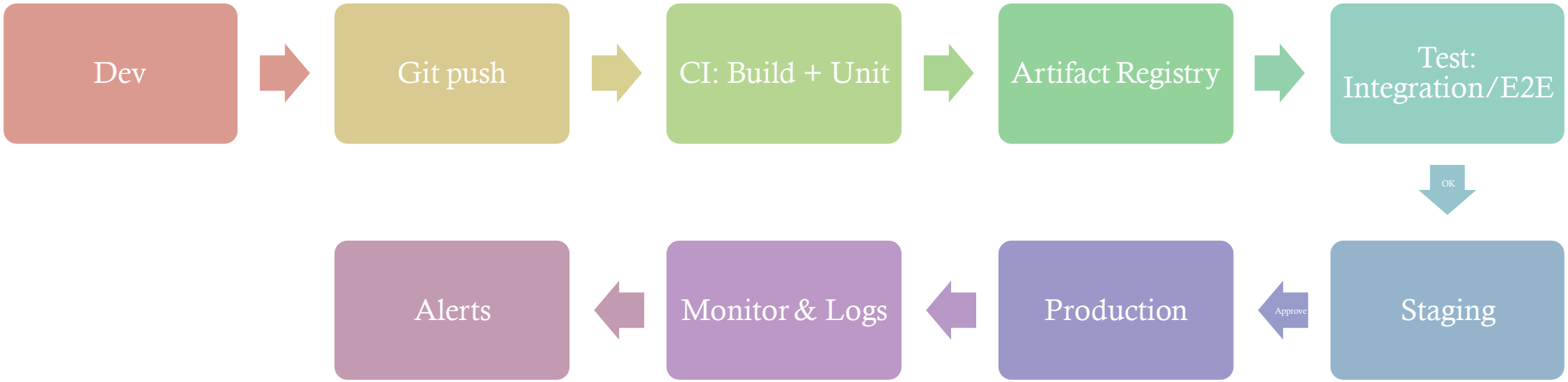
02

Automatic checks (build/test/security)

03

Shorter time from idea to consumer

Emphasis on "fail fast" and quality as a side effect of automation.


Dev
Git push
CI: Build + Unit
Artifact Registry
Test: Integration/E2E
OK
Staging
Approve
Production
Monitor & Logs
Alerts

PRACTICES

01

Trunk-based (for small PRs) or GitFlow

02

Required checks на PR (lint/unit)

03

Semantic versioning + release notes

04

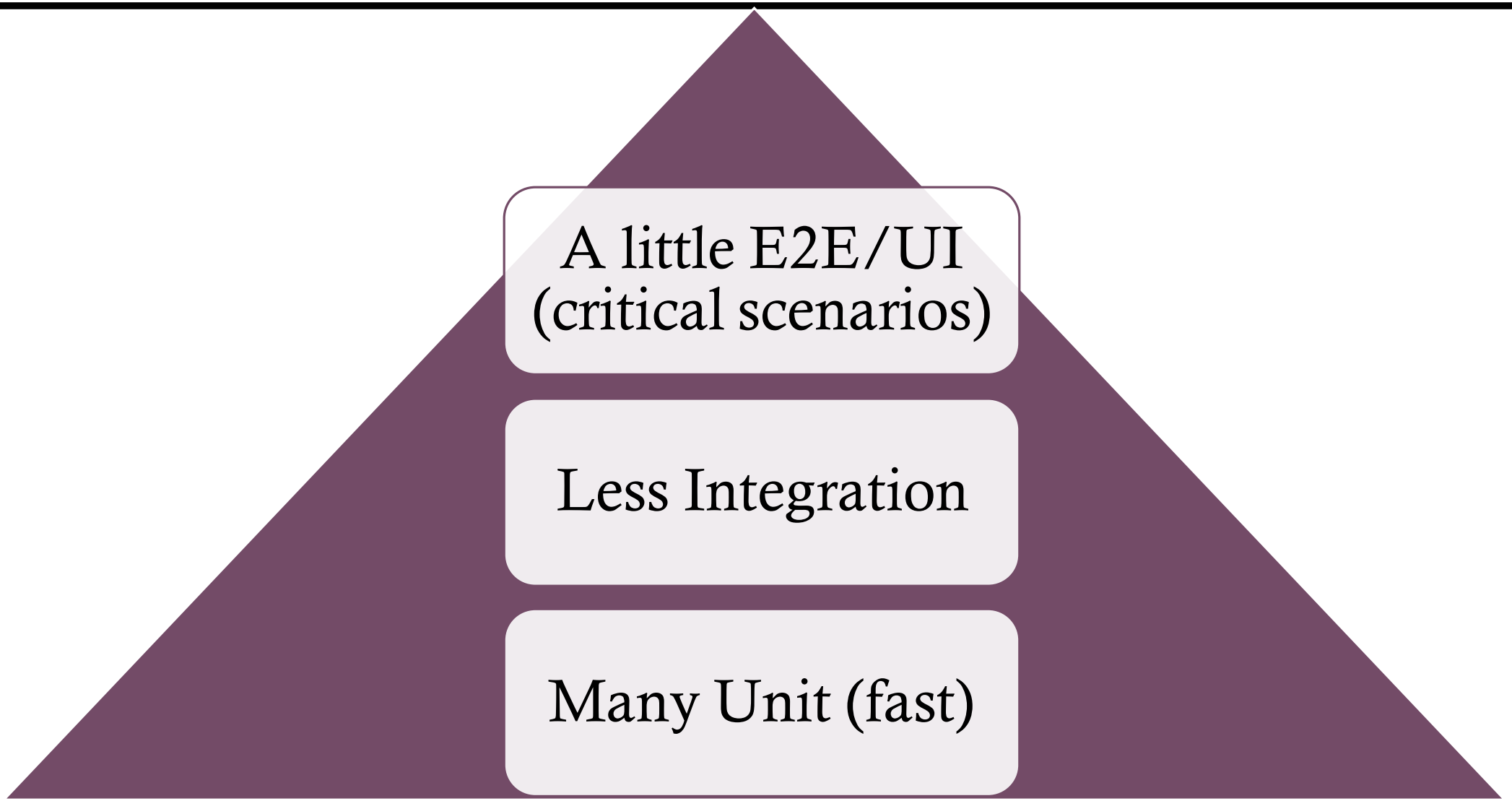
Slide: Artifacts

05

„Build once, deploy many“; Registry (Nexus/Artifactory/Docker Hub)

WHAT WE AUTOMATE

01	02	03	04	05
Installation of dependencies	Compilation/ packaging (Maven/Gradle/npm/pip)	Dependency caching, parallelization	**Slide: Containerization**	Dockerfile → image → push in registry


A little E2E/UI
(critical scenarios)
Less Integration
Many Unit (fast)

RULES

01

Fast feedback (<10 min CI)

02

Parallel execution; stable (non-flaky) tests

03

Coverage as an indicator, not as a goal

DELIVERY VS DEPLOYMENT

01

CD = always ready for release; Continuous Deployment = automatically to prod

02

Blue-Green, Canary, Feature Flags

03

Manual gate for sensitive systems

04

Versioned schema migration

05

Quick rollback plan

WHAT ARE WE MONITORING?

01

Metrics (latency, error rate), logs (centralized), tracing

02

DORA metrics: deploy frequency, lead time, change failure rate, MTTR

03

Meaningful thresholds; noise-free alerts

SUPPLY CHAIN PROTECTION

SECRETS MANAGER (НИКОГА В REPO/ЛОГА)

DEPENDENCY SCANNING / SBOM

STATIC ANALYSIS / SECRET SCANNING

МИНИМАЛНИ ПРАВА ЗА RUNNER-И И DEPLOY KEYS

WHEN JENKINS

On-premises, legacy integrations, high personalization

Master/agents; huge ecosystem of plugins

```
pipeline {
  agent any
  stages {
    stage('Build'){ steps{ sh 'npm ci && npm run build' } }
    stage('Test'){  steps{ sh 'npm test -- --ci' } }
    stage('Package'){
      steps { sh 'docker build -t ghcr.io/org/app:${BUILD_NUMBER} .' }
    }
    stage('Push'){  steps { withCredentials([string(credentialsId:'GHCR_TOKEN', varia
                      sh "echo $T | docker login ghcr.io -u org --password-stdin"
                      sh "docker push ghcr.io/org/app:${BUILD_NUMBER}"
                    }}}
  }
  post { always { junit 'reports/**/*.xml'; archiveArtifacts 'dist/**' } }
}
```

```yaml
  pull_request:
  push: { branches: [ main ] }
jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: '20' }
      - run: npm ci
      - run: npm test -- --ci
      - run: npm run build
  docker:
    needs: build-test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: docker build -t ghcr.io/org/app:${{ github.sha }} .
      - uses: docker/login-action@v3
        with: { registry: ghcr.io, username: ${{ github.actor }}, password: ${{ secrets.GHCR_TOKEN }} }
      - run: docker push ghcr.io/org/app:${{ github.sha }}
```

WHEN ACTIONS?

The code is on GitHub; minimal maintenance; quick start

JENKINS VS GITHUB ACTIONS

- **Setup:** Jenkins (self-hosted) | Actions (SaaS)
- **Flexibility :** Jenkins ↑ (plugins, Groovy) | Actions (enough for 80% of scenarios)
- **Ops weight :** Jenkins ↑ | Actions ↓
- **Ecosystem :** Jenkins plugins | Actions Marketplace
- **Use-case:**
  Enterprise/isolation/legacy | GitHub-centric teams/quick start

DEMO

GOOD PRACTICES AND ANTI-PATTERNS

DO

1. One artifact for all environments
2. Fast CI (<10–15 min), parallel jobs
3. Necessary PR checks;
4. Auto-formatting
5. Centralized logs and monitoring

DON’T

1. Secrets in repo/logs
2. Manual steps in deployment
3. Flaky tests; "broken main" with days
4. Rebuild by environment/machines