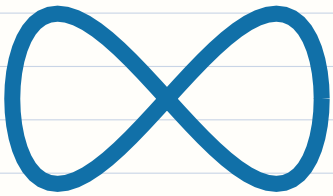
GrowthSchool

CI/CD END-TO-END

COMPLETE NOTES



COMMIT - BUILD - DEPLOY

From a git commit to a live app on AWS EKS - the GitOps way

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

A complete, interview-ready walkthrough of one real pipeline, using a school LMS app (ClassKira) as the example. Every page has a plain-English definition, a numbered diagram, real snippets, common mistakes, and a revision snapshot. Learn the 10-step flow so well you can draw it from memory in an interview.

HOW TO USE THESE NOTES

Learn the flow as a story: commit -> test -> build -> push -> update tag -> sync -> deploy.
If you can retell those 10 steps in order, you can pass almost any CI/CD interview.

1. What is CI/CD?

GrowthSchool

IN SIMPLE TERMS
CI/CD is an automated assembly line for software: every time you change the code it is tested, packaged, and shipped to users automatically - so releases are fast, safe, and repeatable.

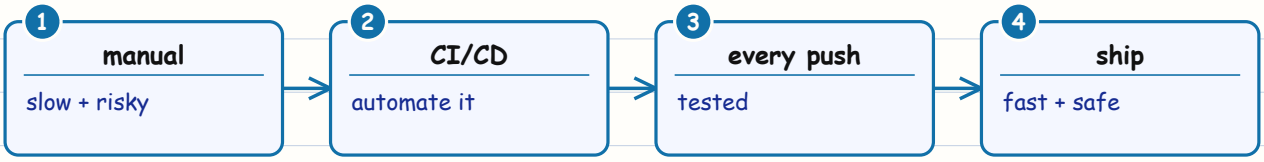
* The Idea

- Instead of manually testing and deploying (slow and error-prone), a pipeline does it for you.
- Every code change is automatically tested, packaged, and shipped - the same way every time.

* Why It Wins

- Faster releases, fewer bugs reaching users, and instant rollback when something is wrong.
- Humans review the change; the machine does the repetitive, risky steps.

Manual vs automated



REAL-WORLD EXAMPLE

```
# the whole promise of CI/CD, in one line:
git push   ->   tested, built, and deployed automatically
```

COMMON MISTAKE
Thinking CI/CD is "just a build script". It is a safety system: tests gate the build, and the deploy is reproducible. Skipping the gates defeats the point.

REVISION SNAPSHOT

ASK YOURSELF	What three things happen automatically on every code change?
DO THIS	Explain CI/CD to a friend in one sentence using the ClassKira app.

2. CI vs CD vs GitOps

GrowthSchool

IN SIMPLE TERMS
CI (Continuous Integration) = automatically build and test every change. CD (Continuous Delivery/Deployment) = automatically ship it. GitOps = Git is the single source of truth that a tool keeps your cluster in sync with.

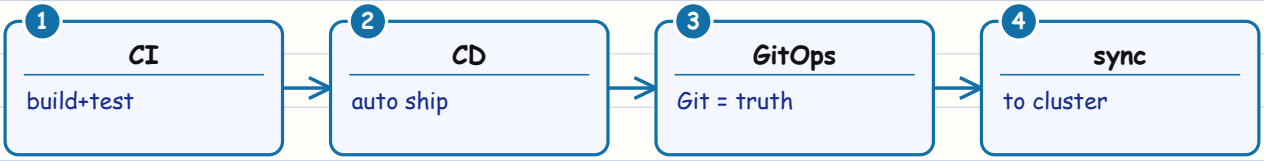
* The Three Terms

- CI = Continuous Integration: build + test every change quickly.
- CD = Continuous Delivery/Deployment: automatically ship the tested change to an environment.

* GitOps

- GitOps means the desired state lives in Git; a tool (Argo CD) makes the cluster match it.
- You never run kubectl by hand in production - you change Git, and the cluster follows.

CI -> CD -> GitOps



REAL-WORLD EXAMPLE

```
CI  : GitHub Actions runs tests + builds the image
CD  : the new image tag is shipped automatically
GitOps: Argo CD syncs the cluster to whatever Git says
```

COMMON MISTAKE
Confusing Continuous Delivery (auto to staging, human approves prod) with Continuous Deployment (auto all the way to prod). Know which one your team uses.

REVISION SNAPSHOT

ASK YOURSELF	In one line each: what is CI, CD, and GitOps?
DO THIS	Say which step Actions owns and which step Argo CD owns.

3. The ClassKira Pipeline

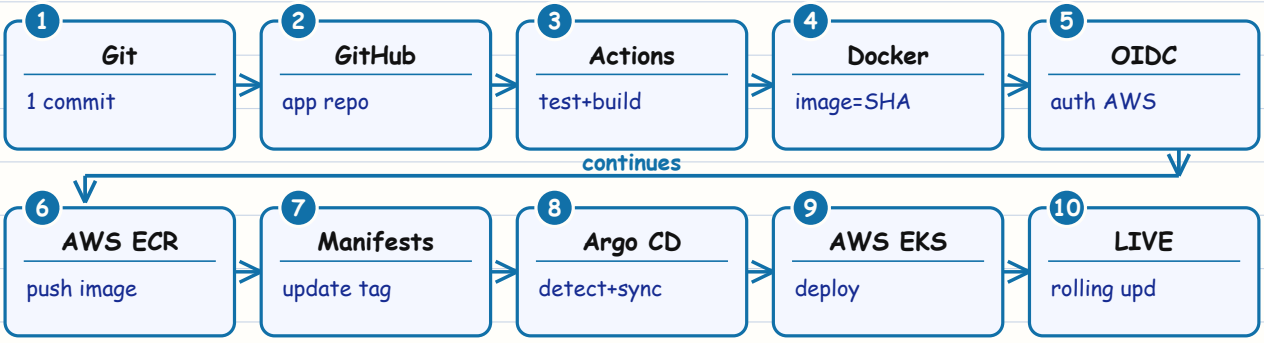
GrowthSchool

IN SIMPLE TERMS
This is the big picture of the whole ClassKira pipeline - one diagram showing how a single commit travels from a developer's laptop all the way to a live app.

* Follow One Commit

- ClassKira is our example school LMS. A developer fixes a bug and pushes once.
- That single push travels through ten automated steps and ends as a live update for students.

ClassKira LMS: one commit -> live in production (follow 1 -> 10)



REVISION SNAPSHOT

ASK YOURSELF	Can you name all ten steps from commit to live, in order?
DO THIS	Trace one ClassKira commit through the diagram out loud.

4. Step 1: Git - Commit

GrowthSchool

IN SIMPLE TERMS
Git is where a developer saves changes to the app code and the Kubernetes manifests (the YAML files that describe how the app should run).

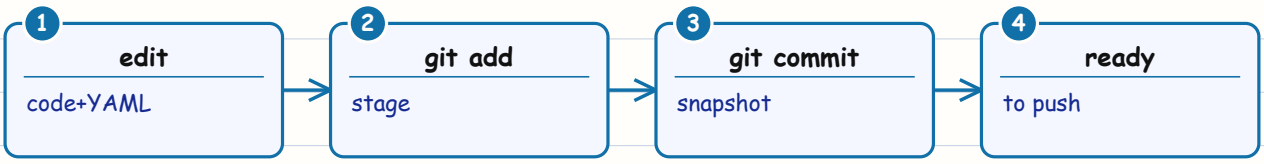
* What You Change

- You edit the app code (the bug fix) and commit it with a clear message.
- The Kubernetes manifests (which image to run) live in Git too - config as code.

* Good Habits

- Small, well-described commits make the pipeline history easy to read and to roll back.
- Use a feature branch, not main, so CI can check it before it merges.

Commit the change



REAL-WORLD EXAMPLE

```
git switch -c fix/login-timeout
git add .
git commit -m "fix(auth): extend login timeout"
git push -u origin fix/login-timeout
```

REVISION SNAPSHOT

ASK YOURSELF	What two kinds of files does GitOps keep in Git?
DO THIS	Make one small, well-named commit on a branch.

5. Step 2: GitHub - Repo & PRs

GrowthSchool

IN SIMPLE TERMS
GitHub is the shared online home for the repository, where teammates review changes through pull requests before they are merged into the main branch.

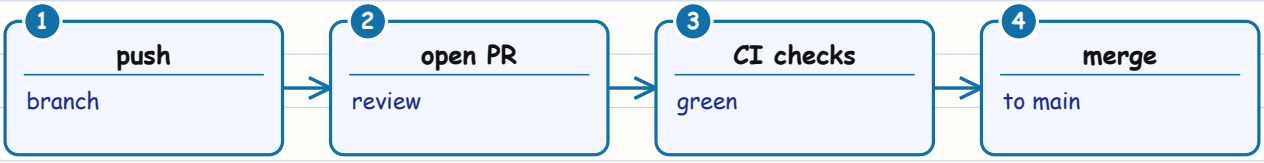
* The Shared Home

- GitHub stores the repository and opens your branch for review as a pull request (PR).
- Teammates comment; CI checks must pass before the PR can merge to main.

* The Trigger

- Opening or updating a PR (and merging to main) triggers the GitHub Actions pipeline.

GitHub pull request



REAL-WORLD EXAMPLE

```
gh pr create --base main --title "fix: login timeout" --fill
# CI runs automatically on the PR; merge when green
gh pr merge --squash
```

COMMON MISTAKE
Pushing straight to main with no PR or checks. One bad commit then ships to students. Protect main: require a PR plus passing CI.

REVISION SNAPSHOT

ASK YOURSELF	What event triggers the pipeline to run?
DO THIS	Open a PR and watch the CI checks appear on it.

6. App Repo vs Manifests Repo

GrowthSchool

IN SIMPLE TERMS
The GitOps pattern uses two repositories: an "app repo" (the source code) and a "manifests repo" (the Kubernetes YAML that says which image version to run).

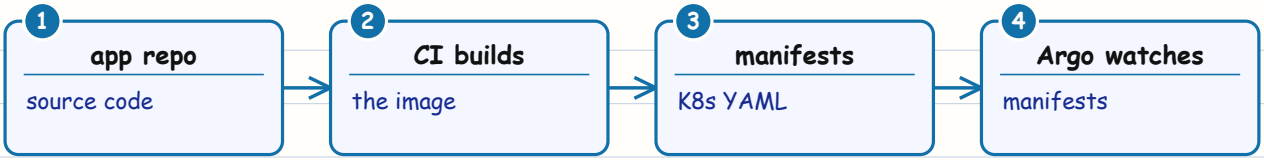
* Two Repositories

- App repo: the ClassKira source code and its Dockerfile. CI builds an image from it.
- Manifests repo: the Kubernetes YAML (Deployment, Service) that says which image tag to run.

* Why Separate

- Argo CD watches only the manifests repo, so a config change (image tag) is what triggers a deploy.
- This clean split is the heart of GitOps.

Two-repo GitOps



REAL-WORLD EXAMPLE

```
app-repo/       src/ ... Dockerfile
manifests-repo/ deployment.yaml   # image: <ECR>/classkira:<TAG>
# Argo CD is pointed at manifests-repo
```

REVISION SNAPSHOT

ASK YOURSELF Which repo does Argo CD watch, and why that one?

DO THIS Sketch the two repos and what each contains.

7. Step 3: GitHub Actions

GrowthSchool

IN SIMPLE TERMS
GitHub Actions is GitHub's built-in automation that runs your pipeline steps automatically whenever you push code or open a pull request.

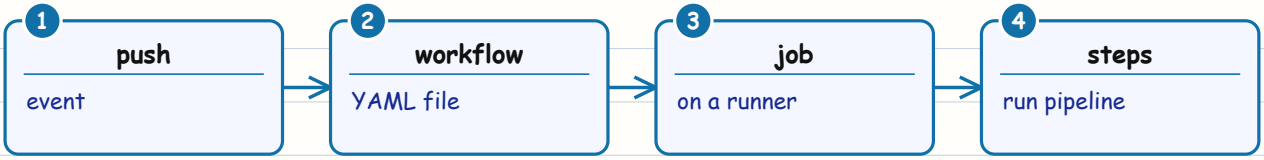
* The Automation Engine

- A workflow is a YAML file in .github/workflows. It lists jobs, each with steps.
- It runs on GitHub-hosted runners whenever the trigger event happens.

* Our Pipeline Job

- One job will: check out code, run tests, build the image, log in to AWS, and push to ECR.

GitHub Actions



REAL-WORLD EXAMPLE

```
name: ci-cd
on:
  push: { branches: [main] }
permissions:
  id-token: write   # needed for OIDC
  contents: write   # to commit the new tag
```

REVISION SNAPSHOT

ASK YOURSELF	Where do GitHub Actions workflows live in the repo?
DO THIS	Create a workflow that runs on every push to main.

8. Step 3: Test & Validate

GrowthSchool

IN SIMPLE TERMS
The CI stage runs your tests and checks (lint, unit tests) to prove a change is safe BEFORE anything is built or shipped.

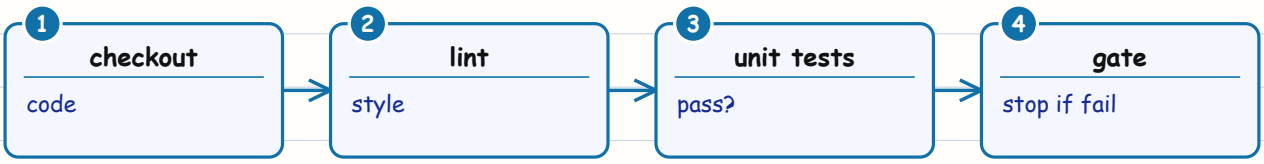
* Prove It Is Safe

- Before building anything, run lint and unit tests. If they fail, the pipeline stops here.
- This "quality gate" keeps broken code from ever becoming an image or a deployment.

* Validate Config Too

- Also validate the Kubernetes YAML (e.g. kubeconform) so a bad manifest cannot ship.

CI quality gate



REAL-WORLD EXAMPLE

```
- uses: actions/checkout@v4
- run: npm ci
- run: npm run lint
- run: npm test           # fails the job on any failing test
- run: kubeconform manifests/
```

COMMON MISTAKE
Building and pushing before tests pass. Always gate the build on green tests - otherwise you ship bugs fast, which is worse than shipping slow.

REVISION SNAPSHOT

ASK YOURSELF	What happens to the pipeline if a unit test fails?
DO THIS	Add a failing test and confirm the pipeline stops.

9. Step 4: Build the Docker Image

GrowthSchool

IN SIMPLE TERMS
A Docker image is the packaged app. CI builds it and labels it with the exact commit id (SHA), so every image is traceable back to its code.

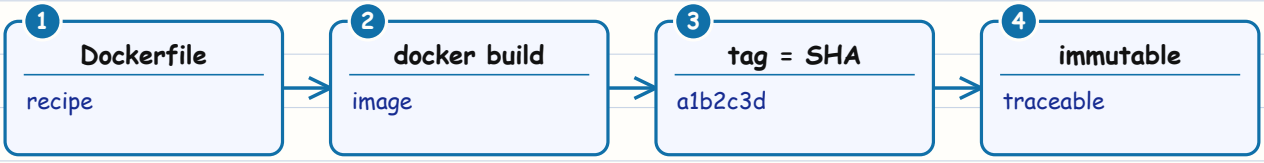
* Package the App

- The Dockerfile turns the tested code into an image. We tag it with the commit SHA.
- A SHA tag (e.g. classkira:a1b2c3d) is immutable and traceable to the exact code.

* Never "latest"

- Using latest hides which version is running and breaks rollbacks. Always use the SHA (or version).

Build the image



REAL-WORLD EXAMPLE

```
SHA=$(git rev-parse --short HEAD)
docker build -t classkira:$SHA .
# later tagged for ECR:  <acct>.dkr.ecr.<region>.amazonaws.com/classkira:$SHA
```

COMMON MISTAKE
Tagging the image latest and deploying that. Two deploys then run different code under the same name. Tag by commit SHA so every image is unique and traceable.

REVISION SNAPSHOT

ASK YOURSELF	Why tag the image with the commit SHA instead of latest?
DO THIS	Build an image tagged with your short commit SHA.

10. Step 5: GitHub OIDC to AWS

GrowthSchool

IN SIMPLE TERMS
OpenID Connect (OIDC) lets GitHub Actions log in to AWS with a short-lived token instead of stored access keys - keyless, and much safer.

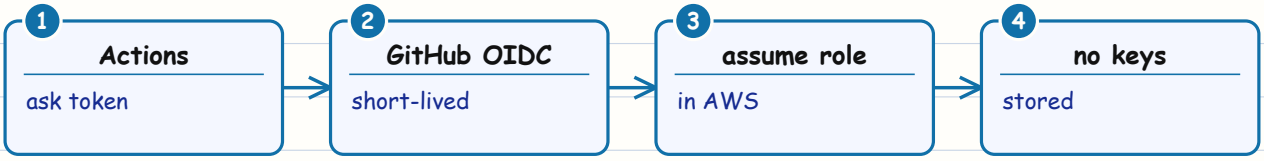
* Keyless Login

- Instead of storing AWS access keys in GitHub, Actions requests a short-lived OIDC token.
- AWS trusts GitHub as an identity provider and lets the workflow assume a scoped IAM role.

* Why It Matters

- No long-lived secrets to leak or rotate. The token lasts minutes and is scoped to this repo.
- This is the modern, secure way to connect CI to a cloud - and a favourite interview topic.

Keyless OIDC login



REAL-WORLD EXAMPLE

```
- uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::<acct>:role/classkira-ci
    aws-region: us-east-1
# no access keys anywhere - just a short-lived OIDC token
```

COMMON MISTAKE
Storing static AWS_ACCESS_KEY_ID / SECRET in GitHub Secrets. They leak and never expire.
Use OIDC + an IAM role trusting your repo instead.

REVISION SNAPSHOT

ASK YOURSELF	How does GitHub Actions log in to AWS without stored keys?
DO THIS	Explain OIDC keyless auth in two sentences.

11. Step 6: Push Image to AWS ECR

GrowthSchool

IN SIMPLE TERMS
AWS ECR is AWS's private image registry. After building, the pipeline pushes the tagged image there so the cluster can pull it later.

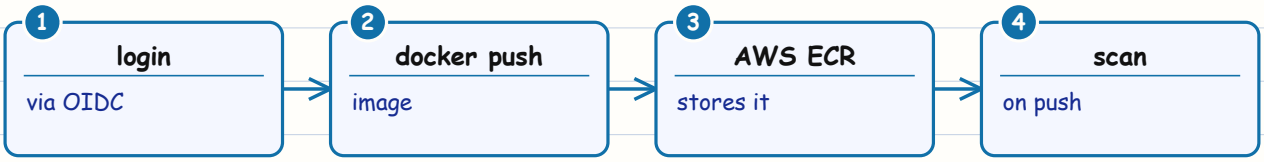
* Store the Image

- With the OIDC role assumed, the workflow logs in to ECR and pushes the SHA-tagged image.
- ECR is private; only the pipeline and the cluster (via IAM) can pull from it.

* Scan It

- Enable scan-on-push so vulnerabilities are caught before the image can ever be deployed.

Push to AWS ECR



REAL-WORLD EXAMPLE

```
aws ecr get-login-password | docker login --username AWS \
  --password-stdin <acct>.dkr.ecr.us-east-1.amazonaws.com
docker push <acct>.dkr.ecr.us-east-1.amazonaws.com/classkira:$SHA
```

REVISION SNAPSHOT

ASK YOURSELF	What just happened after a successful docker push to ECR?
DO THIS	Push a test image to a private ECR repo using OIDC.

12. Step 7: Update the Image Tag

GrowthSchool

IN SIMPLE TERMS
The pipeline automatically edits the Kubernetes manifest to point at the new image tag, and commits that change back to Git.

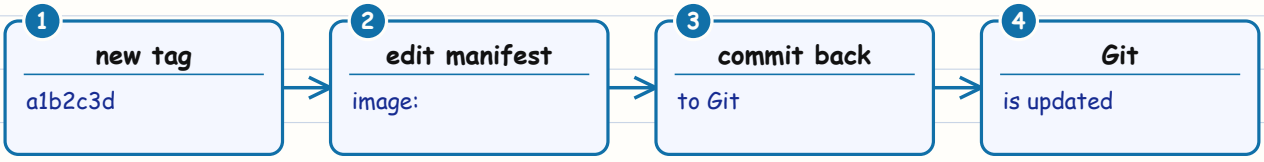
* Commit the New Tag

- The pipeline edits the manifest in the manifests repo to point at the new SHA tag,
- then commits and pushes that change back to Git. Git now holds the new desired state.

* This Is the Handoff

- CI is done. From here CD (Argo CD) takes over - it only reacts to the Git change.

Update image tag



REAL-WORLD EXAMPLE

```
# in the manifests repo, bump the tag and commit
sed -i "s|classkira:.*|classkira:$SHA|" deployment.yaml
git commit -am "deploy: classkira $SHA"
git push   # this commit is what triggers the deployment
```

COMMON MISTAKE
Having CI run kubectl apply directly to the cluster (push-based). That bypasses Git history and Argo CD. In GitOps, CI only updates Git; the cluster is changed by Argo CD.

REVISION SNAPSHOT

ASK YOURSELF	After CI pushes the tag commit, what part of the system reacts next?
DO THIS	Update an image tag in a manifest and commit it.

13. What is Argo CD?

GrowthSchool

IN SIMPLE TERMS
Argo CD is a GitOps tool that watches your manifests repo and continuously makes the cluster match exactly what Git says it should be.

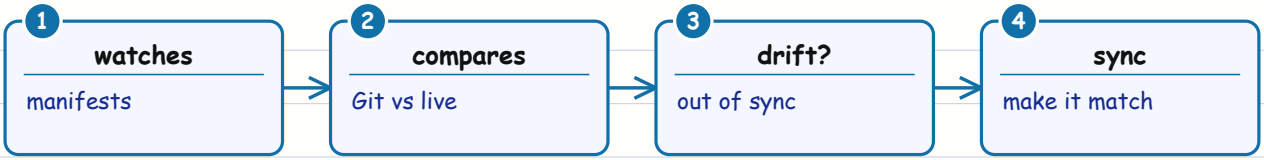
* The GitOps Deployer

- Argo CD runs inside the cluster and continuously watches the manifests repo.
- It compares what Git says (desired) with what is running (live) and reports "Synced" or "OutOfSync".

* Pull, Not Push

- The cluster pulls its own desired state from Git. Nothing outside needs cluster credentials.
- This is safer and fully auditable - every change is a Git commit.

What Argo CD does



REAL-WORLD EXAMPLE

```
argocd app get classkira
# Health: Healthy   Sync: Synced (or OutOfSync)
# source: manifests-repo   destination: EKS cluster
```

REVISION SNAPSHOT

ASK YOURSELF	What two states does Argo CD compare, and what does it watch?
DO THIS	Explain pull-based GitOps vs pushing with kubectl.

14. Step 8: Argo CD Detects the Chan

GrowthSchool

IN SIMPLE TERMS
When the image tag changes in Git, Argo CD notices the difference (drift) and syncs the new desired state to the cluster.

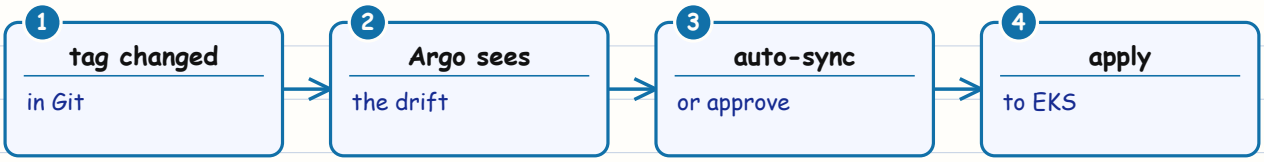
* Drift Detected

- The tag commit makes Git differ from the cluster, so Argo CD marks the app OutOfSync.
- With auto-sync on, it proceeds automatically; otherwise a human clicks "Sync" to approve.

* Controlled Release

- You can require manual sync for production to keep a human approval in the loop.

Argo detects change



REAL-WORLD EXAMPLE

```
# auto-sync policy (in the Argo CD Application)
syncPolicy:
  automated: { prune: true, selfHeal: true }
# or leave it manual and click Sync for prod
```

REVISION SNAPSHOT

ASK YOURSELF	Why does Argo CD suddenly show OutOfSync after the tag commit?
DO THIS	Toggle auto-sync vs manual sync and note the difference.

15. Step 9: Deploy to AWS EKS

GrowthSchool

IN SIMPLE TERMS
Deploying means Argo CD applies the updated manifest to AWS EKS, which then pulls the new image from AWS ECR.

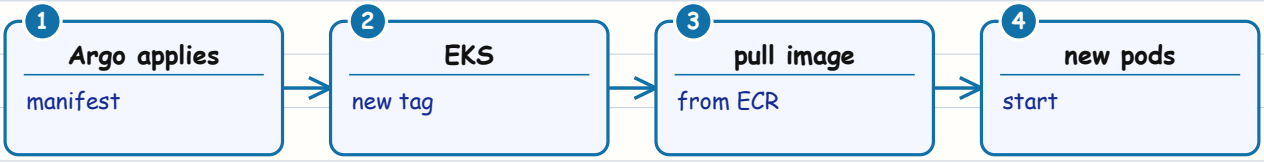
* Apply the Desired State

- Argo CD applies the updated Deployment to AWS EKS. EKS sees a new image tag.
- EKS pulls that exact image from AWS ECR (using the node/pod IAM permissions).

* New Pods Start

- Kubernetes creates new pods running the new image, keeping the old ones until the new are ready.

Deploy to EKS



REAL-WORLD EXAMPLE

```
kubectl get deploy classkira -o wide      # shows the new image tag
kubectl get pods -l app=classkira         # new pods appearing
kubectl describe pod <pod>                # events: pulling from ECR
```

COMMON MISTAKE
Forgetting the cluster needs permission to pull from a private ECR. If pods show ImagePullBackOff, fix the ECR pull permissions / IAM, not the image.

REVISION SNAPSHOT

ASK YOURSELF	Where does EKS pull the new image from, and how is it allowed to?
DO THIS	Watch new pods pull the image after a sync.

16. Step 10: Rolling Update - Live

GrowthSchool

IN SIMPLE TERMS
A rolling update replaces the old pods with new ones gradually, so users never see downtime while the new version goes live.

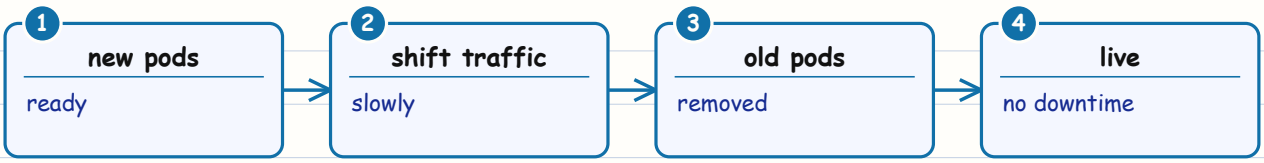
* Zero-Downtime Release

- Kubernetes brings up new pods, waits for readiness, shifts traffic, then removes old pods.
- Students keep using ClassKira the whole time - they just get the fixed version.

* Confirm Success

- Watch the rollout finish and check health/metrics before calling the release done.

Rolling update



REAL-WORLD EXAMPLE

```
kubectl rollout status deploy/classkira
# Waiting for rollout... deployment successfully rolled out
curl -f https://classkira.com/health
```

REVISION SNAPSHOT

ASK YOURSELF	How does a rolling update avoid downtime for users?
DO THIS	Watch a rollout complete and verify the health endpoint.

17. End-to-End Recap

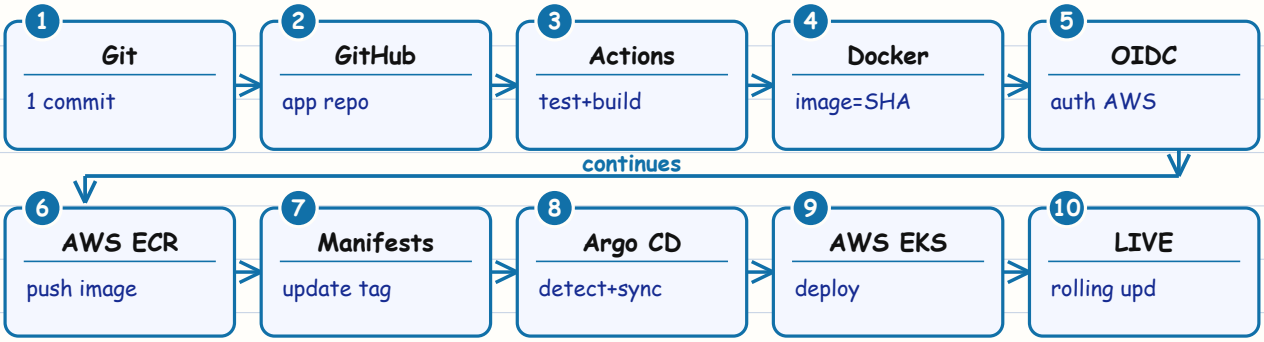
GrowthSchool

IN SIMPLE TERMS
This recaps the entire journey end to end in ten numbered steps, so you can retell the whole flow from memory.

* The Whole Story

- Commit -> push -> test -> build image (SHA) -> OIDC login -> push to ECR ->
- update tag in Git -> Argo CD syncs -> EKS pulls image -> rolling update -> live.

ClassKira LMS: one commit -> live in production (follow 1 -> 10)



REVISION SNAPSHOT

ASK YOURSELF	Retell all ten steps from memory, in order.
DO THIS	Draw the pipeline on paper without looking at the notes.

18. Rollback & Safety

GrowthSchool

IN SIMPLE TERMS
Rolling back is easy in GitOps: you revert the commit in Git, and Argo CD automatically returns the cluster to the previous version.

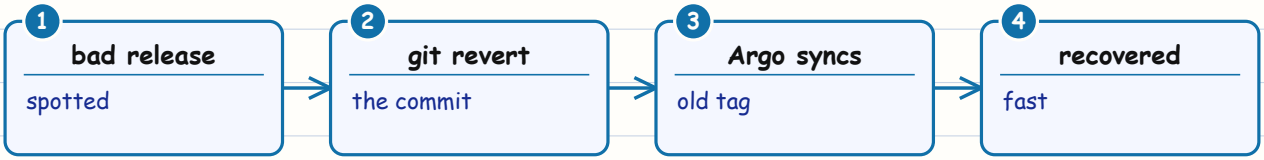
* Undo Is a Git Revert

- A bad release? Revert the tag commit in the manifests repo. Git now points at the old image.
- Argo CD syncs the cluster back automatically - no scary manual cluster surgery.

* Safety Nets

- Keep old images in ECR, protect main, and require review so bad tags rarely merge at all.

Instant rollback



REAL-WORLD EXAMPLE

```
git revert <bad-tag-commit>   # in the manifests repo
git push
# Argo CD detects the change and rolls back to the previous image
```

COMMON MISTAKE
Rolling back by editing the cluster directly. That drifts from Git; the next sync would undo your fix. Roll back IN GIT so Git stays the source of truth.

REVISION SNAPSHOT

ASK YOURSELF	What is the safest way to roll back in GitOps?
DO THIS	Revert a tag commit and watch Argo CD restore the old version.

19. Security in the Pipeline

GrowthSchool

IN SIMPLE TERMS
Pipeline security means no stored secrets (use OIDC), least-privilege permissions, and scanning images for vulnerabilities before they ship.

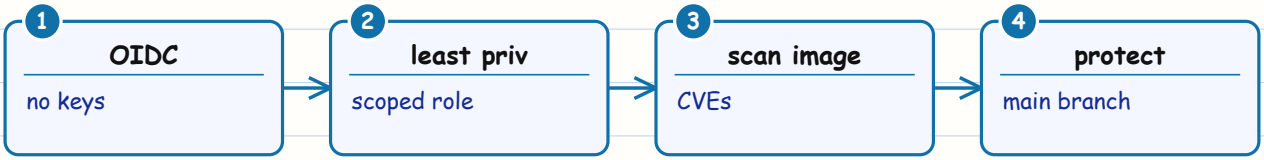
* No Stored Secrets

- Use GitHub OIDC (keyless) for AWS. Give the CI role least privilege - only ECR push it needs.
- Protect the main branch and require PR reviews so only approved code can ship.

* Trust the Image

- Scan images on push, pin base images, and run containers as non-root in the manifests.

Secure the pipeline



REAL-WORLD EXAMPLE

```
# IAM role trust: only this repo + branch can assume it
sub: repo:org/classkira:ref:refs/heads/main
# role policy: ecr:PutImage on the classkira repo only
```

COMMON MISTAKE
Giving the CI role broad admin "to make it work". A leaked pipeline then owns your account. Scope the OIDC role to exactly the ECR actions and repo it needs.

REVISION SNAPSHOT

ASK YOURSELF	Name three ways this pipeline stays secure.
DO THIS	Scope a CI IAM role to only ECR push for one repo.

20. Common Mistakes & Fixes

GrowthSchool

IN SIMPLE TERMS
These are the mistakes that break real pipelines - and the quick checks that fix each one.

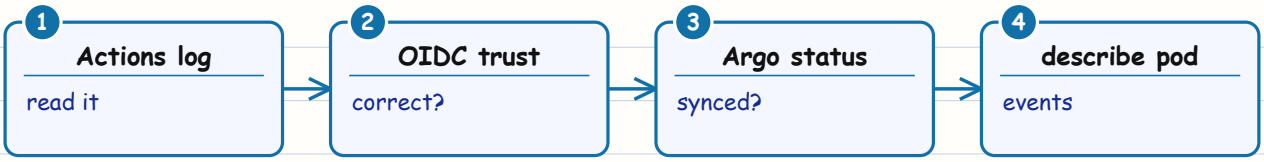
* CI Side

- Build before tests (ship bugs); using latest tag (no rollback); storing static AWS keys (leaks).
- OIDC "not authorized to assume role" usually means the IAM trust policy sub/branch is wrong.

* CD Side

- CI running kubectl directly (bypasses GitOps); ImagePullBackOff (ECR pull perms/wrong tag);
- Argo stuck OutOfSync (manual sync off, or a bad/typo'd manifest).

Debug the pipeline



REAL-WORLD EXAMPLE

```
# quick triage
gh run view --log            # read the Actions logs
argocd app get classkira     # sync + health status
kubectl describe pod <pod>   # Events explain pull failures
```

REVISION SNAPSHOT

ASK YOURSELF	What usually causes "not authorized to assume role" with OIDC?
DO THIS	Match each symptom above to its one-line fix.

21. CI/CD Cheat Sheet

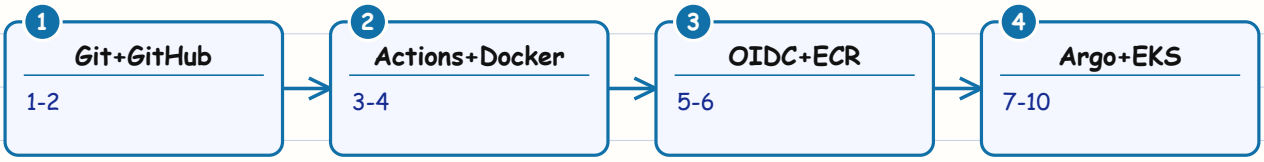
GrowthSchool

IN SIMPLE TERMS
A one-page quick reference of the tools, the commands, and the order of the whole pipeline.

* The Order (memorise this)

- 1 commit 2 push/PR 3 test+validate 4 build image (SHA) 5 OIDC login
- 6 push to ECR 7 update tag in Git 8 Argo detects 9 deploy to EKS 10 rolling update.

Remember the order



REMEMBER IT AS A STORY
Code it (Git+GitHub) -> prove it (test) -> pack it (Docker) ->
store it (ECR) -> point to it (tag in Git) -> ship it (Argo->EKS).

ONE-PAGE PIPELINE REFERENCE

```
git commit / git push              docker build -t app:$SHA .
gh pr create / gh pr merge         aws ecr ... / docker push
npm test / kubeconform             sed -i tag + git commit
configure-aws-credentials (OIDC)   argocd app get / sync
kubectl rollout status             git revert  (rollback)
```

REVISION SNAPSHOT

ASK YOURSELF	Can you list the 10 steps in order without looking?
DO THIS	Recite the pipeline as the 6-word story above.

22. Interview Questions & Answers

GrowthSchool

IN SIMPLE TERMS
The exact questions interviewers ask about CI/CD and GitOps, with crisp answers you can say out loud.

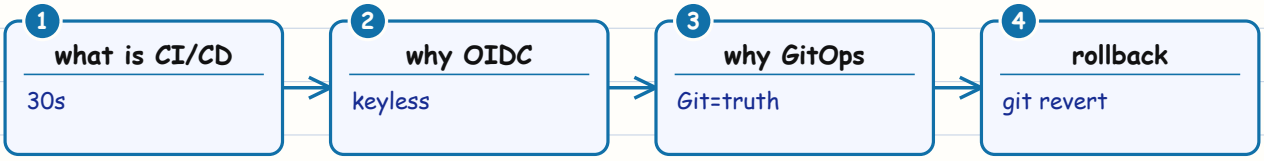
* Be Ready For

- Q: CI vs CD? A: CI builds+tests every change; CD ships it automatically.
- Q: Why OIDC over keys? A: short-lived, scoped tokens - no secrets to leak or rotate.

* The GitOps Ones

- Q: What triggers the deploy? A: a commit that changes the image tag in the manifests repo.
- Q: How do you roll back? A: git revert the tag commit; Argo CD syncs the cluster back.

Nail the interview



REAL-WORLD EXAMPLE

```
# 30-second whiteboard answer
commit -> Actions test+build -> OIDC -> ECR ->
tag commit -> Argo CD sync -> EKS rolling update -> live
```

REVISION SNAPSHOT

ASK YOURSELF	Answer all four questions above out loud in under a minute.
DO THIS	Whiteboard the flow while narrating each step.

23. Capstone: Build the Pipeline

GrowthSchool

IN SIMPLE TERMS
Build the whole ClassKira pipeline yourself, end to end, as your capstone project.

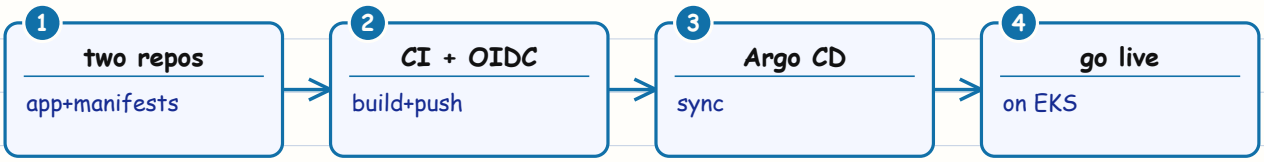
* Set It Up

- Create an app repo (with Dockerfile) and a manifests repo for ClassKira.
- Add a GitHub Actions workflow: test, build (SHA tag), OIDC login, push to ECR, commit the tag.

* Wire the Deploy

- Install Argo CD on an EKS cluster and point it at the manifests repo.
- Push a change and watch it flow all the way to a rolling update.

Capstone plan



STUDENT LAB
Deliverables: keyless OIDC (no stored keys), image tagged by SHA in ECR, an auto-committed tag
in the manifests repo, and an Argo CD app that syncs a rolling update to EKS end to end.

REVISION SNAPSHOT

ASK YOURSELF	Did one commit reach production with zero manual cluster commands?
DO THIS	Complete the capstone, then break a release and roll it back via git revert.