Architektur

Control-Plane in drei Diensten

argocd-server: gRPC/REST-API fuer Web-UI, CLI und CI/CD; Auth, RBAC-Enforcement, Git-Webhook-Empfang.
argocd-repo-server: haelt lokale Kopien der Git-Repos und rendert die Manifeste (Helm, Kustomize, Plain-YAML).
argocd-application-controller: vergleicht den Live-State im Cluster laufend mit dem Soll aus Git, meldet OutOfSync und fuehrt Syncs samt Lifecycle-Hooks aus.

Nebenkomponenten

applicationset-controller: erzeugt Applications in Serie (siehe Generators).
notifications-controller: Trigger/Templates fuer Alerts.
dex: gebuendelter Identity-Broker fuer SSO (OIDC, SAML, LDAP, GitHub).
Redis: reiner Wegwerf-Cache – kann jederzeit ohne Datenverlust neu aufgebaut werden.

HA-Betrieb

argocd-server ist stateless: 3+ Replicas, dazu `ARGOCD_API_SERVER_REPLICAS` setzen.
repo-server horizontal skalieren; `--parallelismlimit` drosselt paralleles Rendern (OOM-Schutz).
application-controller (StatefulSet): Cluster-Sharding via `ARGOCD_CONTROLLER_REPLICAS`; Algorithmen `legacy`, `round-robin`, `consistent-hashing` (experimentell). Richtwert 1000 Apps: 50 Status- / 25 Operation-Processors.
HA-Manifeste brauchen ≥ 3 Nodes (Pod-Anti-Affinity).

```
argocd login argocd.example.com --sso
argocd app list
argocd app get shop --refresh
argocd app diff shop
kubectl -n argocd get applications
```

Application & AppProject

Application

Die Kern-CR: Source (`repoURL`, `path`, `targetRevision`) + Destination (`server`, `namespace`) + `project`.
`syncPolicy.automated` aktiviert Auto-Sync: `prune` loescht aus Git Entferntes, `selfHeal` revertiert manuelle Cluster-Aenderungen zurueck auf den Git-Stand.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: shop
  namespace: argocd
spec:
  project: team-shop
  source:
    repoURL: https://git.example.com/shop/deploy.git
    targetRevision: main
    path: overlays/prod
  destination:
    server: https://kubernetes.default.svc
    namespace: shop
  syncPolicy:
    automated: {prune: true, selfHeal: true}
```

AppProject

Blast-Radius pro Team: `sourceRepos` (erlaubte Git-Quellen), `destinations` (Cluster + Namespaces), `clusterResourceWhitelist` (Allow-List fuer cluster-scoped Kinds), `namespaceResourceBlacklist`.
Projekt-Rollen buendeln Policies; JWT-Tokens dazu (`argocd proj role create-token`) fuer CI-Zugriff.
Das default-Projekt erlaubt initial alles – fuer Prod einschraenken oder eigene Projekte schneiden.

App-of-Apps & ApplicationSet

App-of-Apps (Bootstrapping)

Eine Parent-Application zeigt auf einen Ordner (oder ein Chart) voller Application-Manifeste – ein Sync bootstrappt den ganzen Cluster deklarativ.
Achtung: Wer ins Parent-Repo pushen darf, kann Applications in beliebige Projekte legen – Push-Zugriff dorthin ist Admin-Level.

ApplicationSet-Generators

Template + Generator = Applications in Serie: List (feste Wertepaare), Cluster (alle in Argo CD registrierten Cluster), Git (Directories/Files eines Repos), SCM-Provider und Pull-Request (Repo-/PR-Discovery via Provider-API), Plugin (eigener RPC-Dienst).
Matrix kombiniert zwei Generators (z.B. Cluster × Git), Merge ueberlagert Parameter mehrerer Generators.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata: {name: workloads, namespace: argocd}
spec:
  goTemplate: true
  generators:
    - matrix:
        generators:
          - clusters: {}    # alle registrierten Cluster
          - git:
              repoURL: https://git.example.com/deploy.git
              revision: HEAD
              directories:
                - path: apps/*
  template:
    metadata:
      name: '{{.name}}-{{.path.basename}}'
    spec:
      project: workloads
      source:
        repoURL: https://git.example.com/deploy.git
        targetRevision: HEAD
        path: '{{.path.path}}'
      destination:
        server: '{{.server}}'
        namespace: '{{.path.basename}}'
      syncPolicy:
        automated: {prune: true, selfHeal: true}
```

Sync: Phasen, Waves & Hooks

Phasen, dann Waves

Reihenfolge pro Sync: PreSync → Sync → PostSync; SyncFail laeuft nur bei Fehlschlag. Innerhalb einer Phase ordnen Waves (`argocd.argoproj.io/sync-wave`, Default 0, negativ erlaubt): niedrigste Wave zuerst, dann Kind, dann Name.
Die naechste Wave startet erst, wenn die vorige synced + healthy ist; zwischen Waves liegen 2 s Delay (`ARGOCD_SYNC_WAVE_DELAY`).

Hooks

`argocd.argoproj.io/hook`: `PreSync` (z.B. DB-Migration), `Sync`, `PostSync` (Smoke-Test), `SyncFail` (Cleanup), `Skip` (nicht anwenden).
Aufraeumen via `argocd.argoproj.io/hook-delete-policy`: `HookSucceeded`, `HookFailed`, `BeforeHookCreation` (Default).

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migrate
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/hook-delete-policy: HookSucceeded
    argocd.argoproj.io/sync-wave: "-1"
spec:
  template:
    spec:
      containers:
        - name: migrate
          image: registry.example.com/shop/migrate:1.4.2
      restartPolicy: Never
```

Sync-Options

Pro App oder pro Ressource

`ServerSideApply=true`: K8s Server-Side Apply statt Client-Side – fuer grosse Manifeste (Annotation-Limit) und geteilte Ownership.
`Prune=false` (Ressourcen-Annotation) schuetzt vor Loeschung; `Prune=confirm` verlangt manuelle Bestaetigung.
`PruneLast=true`: prunen erst, nachdem alles Neue healthy ist.
`Replace=true`: `kubectl replace` statt apply – destruktiv, nur gezielt einsetzen.
`ApplyOutOfSyncOnly=true`: nur abweichende Ressourcen anfassen – entlastet den API-Server bei grossen Apps.
Dazu: `CreateNamespace=true`, `Validate=false`, `SkipDryRunOnMissingResource=true` (CRD kommt im selben Sync), `RespectIgnoreDifferences=true`, `PrunePropagationPolicy=foreground`.

```yaml
spec:
  syncPolicy:
    automated: {prune: true, selfHeal: true}
    syncOptions:
      - ServerSideApply=true
      - ApplyOutOfSyncOnly=true
      - PruneLast=true
      - CreateNamespace=true
---
# pro Ressource (Annotation):
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Prune=false
```

Health & Custom Health

Health-Modell

Stati: Healthy, Progressing, Degraded, Suspended, Missing, Unknown. App-Health = schlechtester Zustand der direkten Kind-Ressourcen.
Built-in-Checks u.a. fuer Deployment/StatefulSet/DaemonSet (Generation + Replicas), LoadBalancer-Service und Ingress (`status.loadBalancer.ingress` gefuellt), PVC (`Bound`), Job/CronJob.

Custom Health (Lua)

Fuer CRDs ohne Built-in-Check: Lua-Skript in `argocd-cm`, Key `resource.customizations.health.<group>_<kind>`. Die Ressource liegt als `obj` vor, Rueckgabe `hs.status` + `hs.message`. Standard-Lua-Libs sind aus Sicherheitsgruenden per Default deaktiviert.

```
# argocd-cm (ConfigMap, data:)
resource.customizations.health.cert-manager.io_Certificate: |
  hs = {}
  hs.status = "Progressing"
  hs.message = "Warte auf Issuance"
  if obj.status ~= nil and obj.status.conditions ~= nil then
    for _, c in ipairs(obj.status.conditions) do
      if c.type == "Ready" and c.status == "True" then
        hs.status = "Healthy"
        hs.message = c.message
      end
    end
  end
  return hs
```

Drift & Diff

ignoreDifferences

Erwartete Abweichungen ausblenden statt Dauer-OutOfSync: `jsonPointers` (Feldpfad), `jqPathExpressions` (Listenelemente selektieren), `managedFieldsManagers` (alles eines Field-Managers, z.B. `kube-controller-manager`).
Gilt per Default nur fuer den Diff – mit Sync-Option `RespectIgnoreDifferences=true` auch beim Apply. System-weit: `resource.customizations.ignoreDifferences` in `argocd-cm`.
Seit v3.0 ignoriert der Vergleich `status` bei allen Ressourcen (`ignoreResourceStatusField: all`).

```
spec:
  ignoreDifferences:
    - group: apps
      kind: Deployment
      jsonPointers:
        - /spec/replicas    # HPA steuert Replicas
    - group: "*"
      kind: "*"
      managedFieldsManagers:
        - kube-controller-manager
```

RBAC & SSO

RBAC (argocd-rbac-cm)

Casbin-Policies in `policy.csv`: `p, <subj>, <resource>, <action>, <object>, <effect>`; Gruppen-Mapping via `g, <gruppe>, <rolle>`. Built-ins: `role:readonly`, `role:admin`.
`policy.default` gilt fuer jeden authentifizierten Nutzer und laesst sich nicht per deny-Regel aushebeln – restriktiv setzen.
Seit v3.0: `logs, get` ist eigenstaendig noetig (nicht mehr von `applications` geerbt); `update`/`delete` wirken nur noch auf die App selbst (Sub-Ressourcen fine-grained).

```
# argocd-rbac-cm, policy.csv
p, role:team-shop, applications, get,  team-shop/*, allow
p, role:team-shop, applications, sync, team-shop/*, allow
p, role:team-shop, logs,         get,  team-shop/*, allow
g, oidc:shop-devs, role:team-shop
```

SSO

dex als mitgelieferter Identity-Broker (OIDC, SAML, LDAP, GitHub, ...) oder direktes OIDC ueber `oidc.config` in `argocd-cm`.
Seit v3.0 mappt dex-RBAC auf `federated_claims.user_id` statt `sub` – bestehende Policies beim Upgrade anpassen.

Multi-Cluster

Cluster registrieren

CLI: `argocd cluster add <context>` legt im Ziel-Cluster einen ServiceAccount an. Deklarativ (GitOps-konform): Secret mit Label `argocd.argoproj.io/secret-type: cluster` und Feldern `name`, `server`, `config`.
Der eigene Cluster ist `https://kubernetes.default.svc`. Repos analog: `secret-type: repository`. Viele Cluster: Controller-Sharding (siehe HA-Betrieb).

```
apiVersion: v1
kind: Secret
metadata:
  name: prod-cluster
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: cluster
stringData:
  name: prod
  server: https://prod.example.com:6443
  config: |
    {"bearerToken": "<token>",
     "tlsClientConfig": {"caData": "<b64-ca>"}}
```

Notifications

Trigger + Template + Service

`argocd-notifications-cm` definiert Services (Slack, E-Mail, Webhook, PagerDuty, ...), Trigger (wann) und Templates (was). Ein mitgelieferter Katalog deckt die Standardfaelle ab; Apps abonnieren per Annotation.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  annotations:

    notifications.argoproj.io/subscribe.on-sync-succeeded.slack:
    deploy-info

    notifications.argoproj.io/subscribe.on-health-degraded.slack:
    deploy-alerts
```

Anti-Patterns

Was du nicht tun solltest

kubectl neben Argo CD: manuelle Edits sind Drift; mit `selfHeal` sofort revertiert – Aenderungen gehoeren nach Git.
Alles im default-Projekt: kein Blast-Radius-Limit – pro Team ein AppProject mit `sourceRepos`/`destinations` schneiden.
Auto-Sync mit prune ohne Schutz: kritische Ressourcen mit `Prune=false` oder `Prune=confirm` markieren, `PruneLast` nutzen.
Secrets im Klartext in Git: Argo CD bringt kein Secrets-Management mit – Sealed Secrets, External Secrets Operator o.ae. vorschalten.
Parent-Repo des App-of-Apps offen: Push-Zugriff dorthin ist faktisch Argo-CD-Admin.
ignoreDifferences als Pflaster: wer echten Drift ausblendet, verliert die Kernzusage von GitOps.
policy.default zu breit (z.B. `role:admin`): gilt fuer jeden authentifizierten Nutzer, deny-Regeln greifen nicht dagegen.

argocd-cheatsheet.de

Argo CD Cheatsheet von OMNI52

Weiterfuehren, nicht selbst neu erfinden

GitOps-Rollout & Argo-CD-Governance

OMNI52™ hilft Plattform-Teams, Argo CD fuer regulierte Enterprise-Umgebungen sauber aufzustellen.

HA-Setup mit Controller-Sharding, AppProject- und RBAC-Schnitt mit SSO-Anbindung, ApplicationSet-Strukturen fuer Multi-Cluster-Fleets, Sync-Strategien mit Waves und Hooks, Secrets-Integration ohne Klartext in Git. Output landet als GitOps-Repo bei euch im Cluster: Helm-Charts, Runbooks, Diagnose-Skripte. Euer Team operiert weiter, nicht wir.

OMNI52 GmbH, Ebern · istio-consulting.de · kostenloses 30-min Initial-Assessment

Aus der OMNI52 Cheatsheet-Fabrik

Verwandte Sheets: kubernetes-cheatsheet.de (Kubernetes Core) · rke2-cheatsheet.de (RKE2-Distribution) · rancher-cheatsheet.de (Rancher-Management) · istio-cheatsheet.de (Service-Mesh-Layer).