nuonic digital
ecommerce

Cloud- & DevOps-Track · Head in the Cloud 2026

Zwei Lebenszyklen, ein Projekt

Shopware nativ betreiben, Infrastruktur containerisieren — Deployments mit Deployer & Stack Files bei mittwald

Alex Jank · Co-Founder & Managing Partner, nuonic Digital

KONTEXT

Wer redet hier — und warum dieser Talk?

Shopware-Agentur

München & Hamburg. Wir bauen und betreiben Shopware-6-Shops für den DACH-Mittelstand — Retainer und Wartung, nicht Go-live-und-weg.

Ein Recipe für alle Shops

Eine zentrale Deployer-Konfiguration, per GitHub Actions in jedes Projekt eingebunden. Dieselbe Pipeline, ob ein Shop oder zwanzig.

mittwald als Plattform

Wir haben von drei Hostern auf einen konsolidiert. Native PHP-Runtime, managed Datenbanken und Container — alles in einem Projekt.

Disclaimer: *mittwald hat unser Setup schon zweimal selbst verbloggt. Dieser Talk erzählt es nicht nach — er begründet, warum es so aussieht.*

DAS PROBLEM

Ein Shop ist kein PHP-Skript — es ist ein verteiltes System

MySQL / MariaDB

Bestell- & Stammdaten

Redis

Sessions, Cache, Lock-Store

OpenSearch

Produktsuche & Indexierung

RabbitMQ

Message Queue (Messenger)

Worker

Async-Consumer, Scheduled Tasks

PHP-FPM

Die Storefront- & Admin-App

DAS PROBLEM

Zwei naheliegende Antworten — beide schlecht

Klassisches Shared Hosting

- Kein Redis, kein OpenSearch, keine Queue — die halbe Architektur fehlt einfach.
- Kein SSH-naher Deploy-Workflow, keine atomaren Releases, kein sauberes Rollback.
- Funktioniert für die Visitenkarten-Website. Nicht für einen ernsthaften Shop.

Alles in Container / eigenes k8s

- Du wirst zum Plattform-Team: Base-Image-CVEs, FPM-Tuning, TLS, Patching — alles deins.
- Image-Build pro Deploy, Media-Volumes, OPcache selbst konfigurieren.
- Volle Portabilität — aber für den Betriebsaufwand, den die meisten Shops nie brauchen.

01

DIE THESE

Hybrid ist eine Entscheidung, kein Kompromiss

Die App dort betreiben, wo sie am besten läuft. Die Infrastruktur dort, wo sie hingehört.

nuonic digital
ecommerce

DIE THESE

Die Entscheidungsregel in drei Schritten

01

Managed first

Was die Plattform managed anbietet, nimmst du managed: MySQL, Redis. Backups, Monitoring, Updates sind nicht mehr dein Problem.

02

Container für die Lücke

Was es nicht managed gibt, läuft als Container im selben Projekt: OpenSearch, RabbitMQ. Infrastructure as Code, versioniert.

03

Nativ für die App

Shopware selbst läuft auf der nativen PHP-Runtime — getunt, gepatcht, OPcache-optimiert. Kein Image-Lifecycle an deinem Bein.

`managed → Container-Lücke → native App`

DIE THESE

Konkret: Was läuft wo — und warum

Komponente	Betriebsart	Warum
Shopware (PHP-FPM)	**Native Website-App**	Getunte Runtime, OPcache-Preload, Patching & TLS abgegeben
MySQL	**Managed**	Backups, PITR, Monitoring inklusive — der Default
Redis	**Managed**	Sessions/Cache/Locks; gesichert & überwacht ohne Zutun
OpenSearch	**Container**	Nicht managed verfügbar — also als Code im Stack File
RabbitMQ	**Container**	Nicht managed verfügbar — Messenger-Transport über AMQP
MariaDB	**Container (nur wenn zwingend)**	Kein managed Backup — Konsistenz musst du selbst bauen

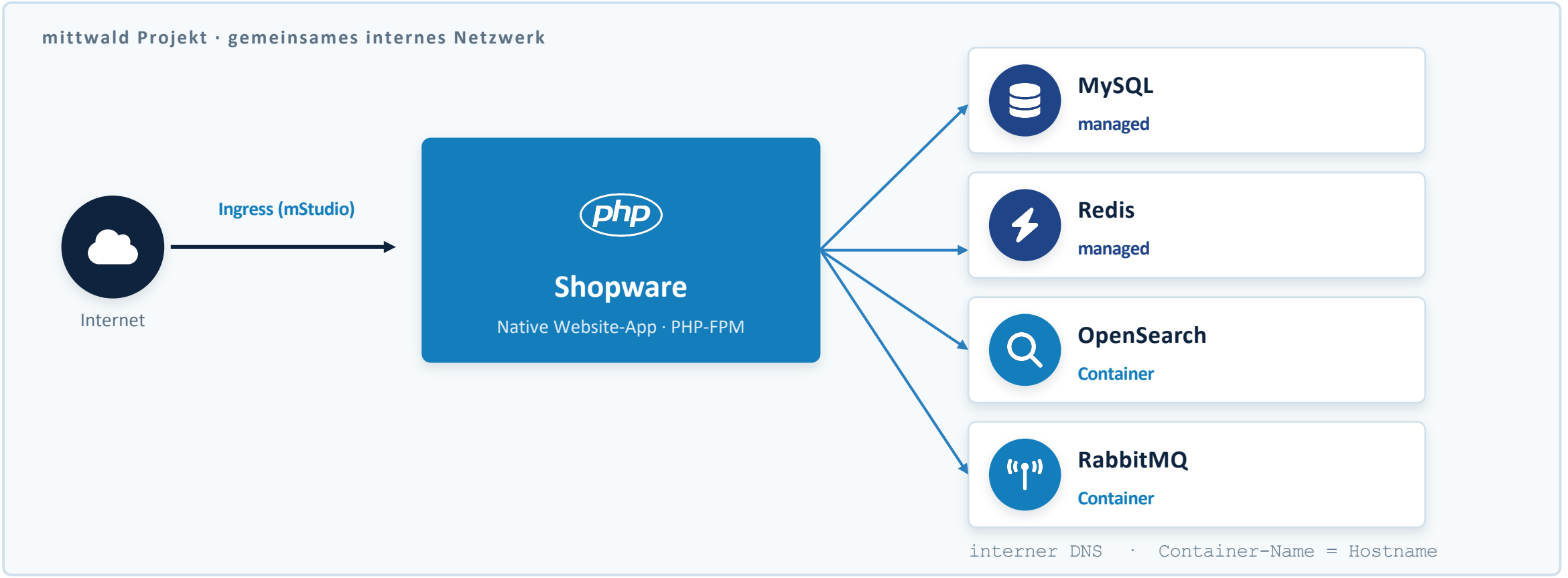
02

ARCHITEKTUR

Ein Projekt, ein Netzwerk

App und Container teilen sich dasselbe Projekt-Netzwerk. Erreichbarkeit über internen DNS — keine öffentlichen Ports für die Infrastruktur.

Ein mittwald-Projekt von innen



Verdrahtung: interner DNS in der .env

Der Container-Name ist der Hostname. Keine IPs, keine öffentlichen Ports — die App referenziert die Dienste schlicht über ihren Namen im selben Netzwerk.

shopware/.env.local

```
# .env.local — alles über internen DNS, kein Port nach außen
DATABASE_URL="mysql://user:pass@mysql:3306/shopware"
REDIS_URL="redis://redis:6379"

# OpenSearch-Container (Hostname = Service-Key 'opensearch')
SHOPWARE_ES_ENABLED=1
SHOPWARE_ES_HOSTS="http://admin:PASSWORT@opensearch:9200"
SHOPWARE_ES_INDEXING_ENABLED=1
SHOPWARE_ES_INDEX_PREFIX=sw

# RabbitMQ-Container als Messenger-Transport (AMQP)
MESSENGER_TRANSPORT_DSN="amqp://guest:guest@rabbitmq:5672/%2f/messages"
```

03

CONTAINER-SEITE

Stack File als Code

Ein Docker-Compose-kompatibles YAML in Git. Deployt über die offizielle mittwald GitHub Action — idempotent und reproduzierbar.

nuonic digital
ecommerce

Das Stack File: Docker-Compose, das du committest

deploy/stack.yaml

```
# stack.yaml — im Git-Repo, deployt via Action
services:
  opensearch:
    image: "opensearchproject/opensearch:2"
    # ...
  rabbitmq:
    image: "rabbitmq:3-management"
    # ...
volumes:
  opensearch-data: {}
  rabbitmq-data: {}
```

Idempotent

Die Action schreibt den Stack exakt wie im YAML. Mehrfaches Anwenden ändert nichts — nur Diffs werden umgesetzt.

Code ist die Wahrheit

Manuelle Änderungen im mStudio werden beim nächsten Deploy überschrieben. Also: Stack konsequent als Code führen.

Reproduzierbar

Neuer Kunde, gleicher Stack? Copy-Paste plus Deploy. Aus stundenlangem Server-Setup wird unter eine Stunde.

OpenSearch im Container — Persistenz nicht vergessen

deploy/stack.yaml · opensearch

```
services:
  opensearch:
    image: "opensearchproject/opensearch:2"
    envs:
      discovery.type: single-node
      plugins.security.ssl.http.enabled: false   # Container kann kein HTTPS -> intern http
      OPENSEARCH_INITIAL_ADMIN_PASSWORD: "{{ .Env.OS_ADMIN_PASSWORD }}"  # Pflicht ab 2.12
    ports: ["9200/tcp"]
    volumes:
      - "opensearch-data:/usr/share/opensearch/data"   # <- ohne Volume sind die Indizes weg
volumes:
  opensearch-data: {}
```

Merke: Ohne Volume auf /usr/share/opensearch/data verlierst du bei jedem Recreate den kompletten Suchindex. Das Volume wird automatisch ins Projekt-Backup einbezogen.

CONTAINER-SEITE

OpenSearch RAM zähmen — der wichtigste Handgriff

OpenSearch greift sich per Default, was es kriegen kann. Ohne Begrenzung füllt die JVM problemlos zweistellige GB. Du brauchst ZWEI Grenzen — nicht eine.

1 · JVM-Heap begrenzen

```
OPENSEARCH_JAVA_OPTS:
  "-Xms512m -Xmx512m"
bootstrap.memory_lock: true
```

-Xmx kappt den Heap; memory_lock verhindert Swapping.

2 · Container-Limit (cgroup)

Heap ≠ harte Grenze. Off-Heap (Lucene-mmap, Thread-Stacks) liegt obendrauf. Ohne ein hartes Memory-Limit am Container kann der Prozess das Heap-Limit überschießen — und der OOM-Killer holt ihn.

Heap-Limit UND Container-Limit — beides, nicht entweder/oder.

RabbitMQ — Messenger-Transport im Container

deploy/stack.yaml · rabbitmq

```
services:
  rabbitmq:
    image: "rabbitmq:3-management"
    envs:
      RABBITMQ_DEFAULT_USER: "{{ .Env.MQ_USER }}"
      RABBITMQ_DEFAULT_PASS: "{{ .Env.MQ_PASS }}"
    ports:
      - "5672/tcp"     # AMQP
      - "15672/tcp"    # Management-UI
    volumes:
      - "rabbitmq-data:/var/lib/rabbitmq"
```

3-management-Image

Bringt die Management-UI auf 15672 mit — Queues und Consumer im Blick, ohne Extra-Tooling. Könnt ihr dann per Subdomain / Projekt-Domain mappen.

Transport für Messenger

Shopware schiebt asynchrone Tasks über die AMQP-DSN hier rein. Deshalb das amqp-PHP-Ext im CI.

Volume = persistente Queue

Ohne /var/lib/rabbitmq-Volume sind nach einem Recreate alle noch nicht verarbeiteten Nachrichten weg.

skip_recreation: Datendienste nicht bei jedem Deploy killen

.github/workflows/stack.yml

```
- name: Deploy Stack to mittwald
  uses: mittwald/deploy-container-action@v1
  with:
    api_token: ${{ secrets.MITTWALD_API_TOKEN }}
    stack_id:  ${{ vars.STACK_ID }}
    stack_file: "deploy/stack.yaml"
    skip_recreation: "mariadb,opensearch,rabbitmq"   # stateful -> nicht neustarten
```

Ohne skip_recreation

Jedes Stack-Deploy startet auch die Datendienste neu — unnötige Downtime, kalte Caches, Risiko bei laufenden Indizierungen.

Mit skip_recreation

Nur Dienste mit echtem Diff werden angefasst. Datenbanken, Suche und Queue laufen ungestört weiter.

CONTAINER-SEITE

MariaDB im Container — wenn's wirklich sein muss

Manche Projekte fordern MariaDB statt managed MySQL. Geht per Container — aber das managed Backup entfällt. Konsistente Backups baust du selbst:


```
# cron im mariadb-Container: täglicher konsistenter Dump ins gemountete Volume
0 3 * * *  mariadb-dump --single-transaction --quick shopware \
             | gzip > /backup/shopware-$(date +\%F).sql.gz    # /backup = mittwald-Volume
```

Backup-Realität: warum managed der Default bleibt

Managed MySQL / Redis

- ✓ Backups & Point-in-Time-Recovery inklusive
- ✓ Monitoring und Updates ohne dein Zutun
- ✓ Konsistenz ist garantiert — kein Eigenbau
- ✓ Ein Knopf weniger, an dem du dich verbrennst

Container-DB (MariaDB)

- — Volume-Snapshot ≠ konsistentes Backup
- — Dump-Cron, Rotation & Restore-Test: dein Job
- — Mehr Kontrolle — und mehr Verantwortung
- — Nur bei echtem Zwang, nicht aus Gewohnheit

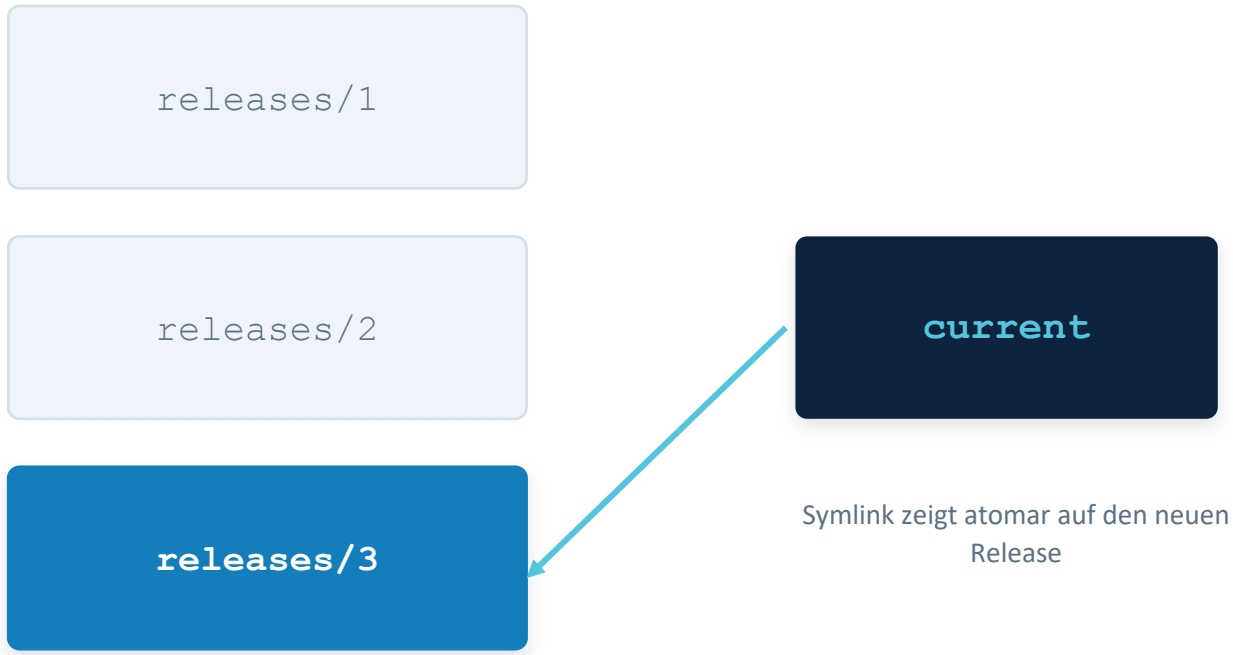
04

APP-SEITE

Deployer: atomar, zero-downtime

Die App läuft nativ — und wird mit atomaren Releases ausgerollt. Eine zentrale Recipe, in jedes Projekt eingebunden.

Deployer: atomare Releases per Symlink-Swap



releases/

Jeder Deploy ein eigenes, vollständiges Verzeichnis.

shared/

Media, Logs, .env überleben Releases — per Symlink eingehängt.

current → releases/N

Der Umschalter. Ein Symlink-Swap = der Go-live, in Millisekunden.

Rollback

Symlink auf den alten Release zurück. Sofort, ohne Re-Deploy.

APP-SEITE

Eine Recipe für alle Shops

Unsere Deployer-Konfiguration liegt in einem eigenen Repo und wird zur Deploy-Zeit in jedes Projekt eingecheckt. Ein Fix dort verbessert sofort jeden Shop.

GitHub Actions

```
# .github/workflows/deploy.yml  (reusable, in jedem Shop-Repo aufgerufen)
- uses: actions/checkout@v6              # 1) der Shop selbst
- uses: actions/checkout@v6              # 2) die zentrale nuonic-Recipe
  with:
    repository: 'nuonic-digital/infra.nuonic.deployer'
    path: 'deployer-config'
    token: ${{ secrets.INFRA_GITHUB_TOKEN }}
```

`src/recipe/shopware6.php` baut auf Deployers offizieller Shopware-Recipe auf und ergänzt unsere Tasks — einmal gepflegt, überall ausgerollt.

APP-SEITE

Build im CI — nicht auf dem Server

Build-Strategie

```
// src/recipe/shopware6.php
set('update_code_strategy', 'archive');   // gebautes Artefakt als Archiv hochladen

task('sw-build-without-db:build', function () {
    // Assets/Theme im CI bauen — ganz ohne DB-Verbindung
    runLocally('APP_ENV=ci shopware-cli project ci .');
});

task('deploy:update_code', function () {
    upload('./', '{{release_path}}');      // fertig gebauten Stand hochladen
});
```

Konsequenz: Der Server braucht keine Build-Toolchain und keinen DB-Zugriff zum Bauen. Composer, Node, shopware-cli laufen im Runner — der Server bekommt nur das fertige Ergebnis.

APP-SEITE

Shared Files & Environments sauber trennen

shopware6.php

```
set('shared_files', [
    '.env.local',
    '.env.prod.local',
    '.env.staging.local',
    'install.lock',
    'public/.htaccess',
    'public/.user.ini',
]);

set('labels_env', fn() =>
    get('labels')['env']);   // prod | staging
```

Secrets überleben Releases

Die .env-Dateien liegen in shared — kein Secret im Repo, kein Verlust beim Deploy.

Pro Environment

labels_env unterscheidet prod und staging — eine Recipe, mehrere Umgebungen.

Staging-Schutz

Auf staging läuft automatisch system:setup:staging — keine echten Mails, keine Live-Zahlungen.

APP-SEITE

Migrations & Co. an den Deployment-Helper delegieren

shopware6.php

```
task('sw:deployment-helper:run', function () {
    $env = [];
    if (test('[ -f {{release_path}}/.shopware-project.{{labels_env}}.yml ]')) {
        $env['SHOPWARE_PROJECT_CONFIG_FILE'] = '.shopware-project.'.get('labels_env').'.yml';
    }
    if (test('[ -f {{release_path}}/vendor/bin/shopware-deployment-helper ]')) {
        run('cd {{release_path}} && vendor/bin/shopware-deployment-helper run', env: $env);
    }
});
```

Der Helper übernimmt Migrations, Plugin-Updates, Theme-Aktivierung & Cache — gesteuert über `.shopware-project.{env}.yml`. Ist er vorhanden, überspringt die Recipe ihr manuelles plugin:update:all.

APP-SEITE

Der OPcache-Trick: der Preis der nativen Runtime

Das Problem

OPcache cacht absolute Pfade. Nach dem Symlink-Swap zeigt current auf den neuen Release — aber OPcache liefert weiter den alten Pfad aus. Ergebnis: stale Code oder Fatal Errors.

Die Lösung

OPcache gezielt zurücksetzen — direkt nach dem Symlink-Swap. In Container-Welten löst ein neuer Prozess das implizit; auf der nativen Runtime machst du es explizit mit einem Hook.

shopware6.php · opcache

```
task('opcache:clear', function () {
    // cachetool resettet den OPcache
    run('cd ~ && {{bin/php}} \
        ./cachetool.phar opcache:reset');
});

// nach dem atomaren Swap zurücksetzen
after('deploy:symlink', 'opcache:clear');
after('sw:cache:clear', 'opcache:clear');
```

APP-SEITE

Cache-Invalidierung: Redis gezielt flushen

shopware6.php · redis

```
set('redis_flush_dbs', [0, 1]);

task('sw:redis-flush', function () {
    $cmd = ['redis-cli'];
    // optional Passwort/Port aus der Umgebung
    if ($pw = getenv('REDIS_PASSWORD'))
        $cmd[] = '-a '.$pw;
    foreach (get('redis_flush_dbs') as $db)
        run("...-n $db FLUSHDB SYNC");
});
```

Gezielt statt FLUSHALL

Nur die definierten DBs (0, 1) werden geleert — andere Daten in Redis bleiben unangetastet.

Frischer Cache nach Deploy

Neuer Code, neue Konfiguration — alter Cache-Stand muss weg, sonst gibt's Geister.

Port & Passwort flexibel

Funktioniert mit managed Redis genauso wie mit abweichenden Ports pro Umgebung.

APP-SEITE

Ehrlich bleiben: Zero-Downtime hat eine Bedingung

Atomare Releases sind notwendig, aber nicht hinreichend. Im Swap-Fenster läuft der alte Release kurz gegen das bereits migrierte Schema.

Rückwärtskompatibel = sicher

✓ Spalte/Tabelle additiv hinzufügen

✓ Neue Felder nullable oder mit Default

✓ Alter & neuer Code laufen beide sauber

✓ Der Symlink-Swap bleibt unkritisch

Destruktiv = Downtime-Risiko

— Spalte droppen/umbenennen bricht alten Code

— Typänderungen mitten im Swap-Fenster

— Lösung: Expand/Contract in zwei Deploys

— Erst additiv ausrollen, später aufräumen

05

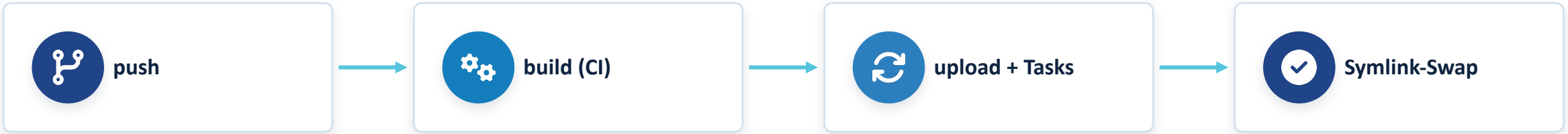
CI/CD

Zwei Tools, zwei Kadenzen

App und Infrastruktur ändern sich unterschiedlich oft. Also: zwei entkoppelte Pipelines — nicht eine, die alles vermischt.

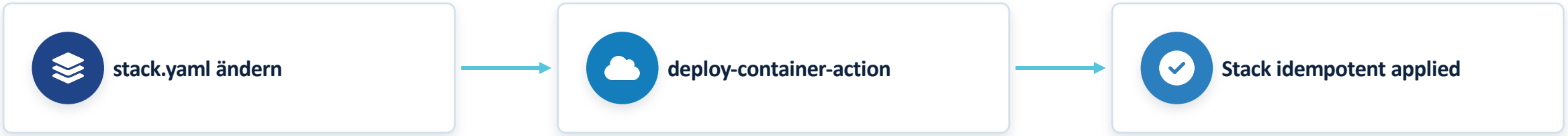
Zwei entkoppelte Pipelines

A · Code-Änderung · oft



`Deployer · deployphp/action`

B · Infrastruktur-Änderung · selten



Du deployst nicht OpenSearch neu, weil sich ein Theme-Pixel geändert hat.

deploy.yml unter der Haube

.github/workflows/deploy.yml

```
on: { workflow_call: ... }   # reusable
jobs:
  deploy:
    runs-on: ubuntu-latest-m
    steps:
      - uses: 1password/load-secrets-action@v4
      - uses: shivammathur/setup-php@v2
        with: { extensions: amqp }   # RabbitMQ
      - run: composer install --no-dev
      - uses: deployphp/action@v1.0.26
        with: { dep: deploy env=prod }
```

Secrets aus 1Password

SSH-Key, Composer-Tokens, Redis-Credentials kommen zur Laufzeit aus 1Password — nicht roh in GitHub.

Composer-Auth

Bearer-Token für packages.shopware.com plus Repman für private Pakete.

amqp-Extension

Wird gesetzt, damit die App den RabbitMQ-Transport sprechen kann.

06

BETRIEB & FALLSTRICKE

Die Narben

Was in keinem Blogpost steht: die Dinge, die uns im Betrieb tatsächlich erwischt haben — und wie wir sie heute abfangen.

nuonic digital
ecommerce

Vier Narben aus dem echten Betrieb

OpenSearch-OOM

Ohne hartes Container-Limit überschießt der Prozess das Heap-Limit — OOM-Kill. Heap UND Container begrenzen.

ext-redis beim Hostwechsel

PHP-Extension-Konflikte beim Umzug zwischen Hostern. Extension-Matrix vorher prüfen, nicht erst beim Deploy.

Theme-Asset-Hash-Mismatch

Im Swap-Fenster zeigt HTML auf Asset-Hashes des anderen Release → kurzzeitig 404.

Worker-Lebenszyklus

Messenger-Consumer hängen am alten Code, bis sie neu starten. Nach Deploy gezielt restarten.

Backup-Verantwortung: wer macht was?

Was	Wer	Hinweis
Managed MySQL / Redis	**mittwald**	Backup, PITR, Monitoring — vollständig abgedeckt
Container-Volumes (OpenSearch, RabbitMQ)	**mittwald**	Volume wird ins Projekt-Backup einbezogen
OpenSearch-Index	**reproduzierbar**	Kein Backup nötig — per es:admin:index neu aufbaubar
Container-DB (MariaDB)	**DU**	Konsistenter Dump-Cron ins Volume — plus Restore-Test
Code & Stack File	**Git**	Single Source of Truth — jederzeit re-deploybar

ABWÄGUNG

Warum nicht einfach alles in Container?

Pro Container-everything

- Ein mentales Modell für alles
- Volle Portabilität, kein Lock-in
- Dev/Prod-Parität bis aufs Image
- Frei in der Versionswahl jeder Komponente

Warum Hybrid hier gewinnt

✓ Getunte FPM-Runtime + OPcache-Preload

✓ OS-Patching, TLS, Updates abgegeben

✓ Keine PHP-Base-Image-CVEs an deinem Bein

✓ Reifes Deploy-Modell für media-lastige Apps

Der Preis: *Lock-in. Den haben wir bewusst genommen — drei Hoster zu einem. Wenn Plattform-Unabhängigkeit euer härtestes Kriterium ist, ist Container-everything die richtige Wette.*

ZUM MITNEHMEN

Vier Sätze für den Heimweg

01

Entscheidungsregel

managed → Container für die Lücke → native App. In dieser Reihenfolge.

02

Zwei Lebenszyklen trennen

App via Deployer (oft), Infra via Stack File (selten). Entkoppelt, nicht vermischt.

03

IaC ab Tag 1

Stack File in Git, zentrale Recipe für alle Shops. Reproduzierbar statt handgepflegt.

04

Die teuren Lektionen sind Betrieb

RAM-Limits, OPcache-Reset, Backup-Konsistenz — daran entscheidet sich's, nicht am Setup.

nuonic digital ecommerce

nuonic digital ecommerce

Danke. Fragen?

Stack File, Deployer-Recipe & die Hooks teile ich gern — sprecht mich an.

Alex Jank

Co-Founder & Managing Partner · nuonic Digital

kontakt@nuonic.de · nuonic.de