NEWS · JULY 5, 2026

# Eine virtuelle Maschine ist keine Versionsverwaltung — und was Sie stattdessen tun sollten

By Dr. Holger Flick




Wer oft genug auf Entwicklerkonferenzen spricht, führt immer wieder dasselbe Gespräch. Es gibt viele talentierte Entwickler — oft Delphi-Entwickler, oft Menschen, die seit zwanzig Jahren ernsthafte Software ausliefern —, die, sobald das Thema auf Versionsverwaltung kommt, einen neuen und kreativen Grund parat haben, warum sie keine einsetzen. Als **GitKraken Ambassador** habe ich inzwischen die meisten davon gehört.

Einer taucht immer wieder auf und lässt einen tatsächlich innehalten, weil er verantwortungsbewusst *klingt*:

> "I use virtual machines for everything, so my code is safe."

Auf Deutsch: "Ich benutze für alles virtuelle Maschinen, mein Code ist also sicher." Es ist ermutigend, dass Sicherheit überhaupt eine Rolle spielt — die meisten aus der "Ich benutze keine Versionsverwaltung"-Fraktion denken gar nicht erst darüber nach. Aber es muss klar gesagt werden: **Eine virtuelle Maschine ist keine**

**Versionsverwaltung, sie ist nicht einmal ein brauchbarer Ersatz dafür, und wer VMs so einsetzt, erzeugt still und leise genau das Problem, das er vermeiden will.**

Lassen Sie mich erklären, warum — behutsam, denn der Instinkt dahinter ist ein guter — und dann zeige ich Ihnen ein Setup, das Ihnen in etwa fünfzehn Minuten das Original liefert, auf Ihrer eigenen Hardware, ohne irgendeinem Cloud-Anbieter vertrauen zu müssen.

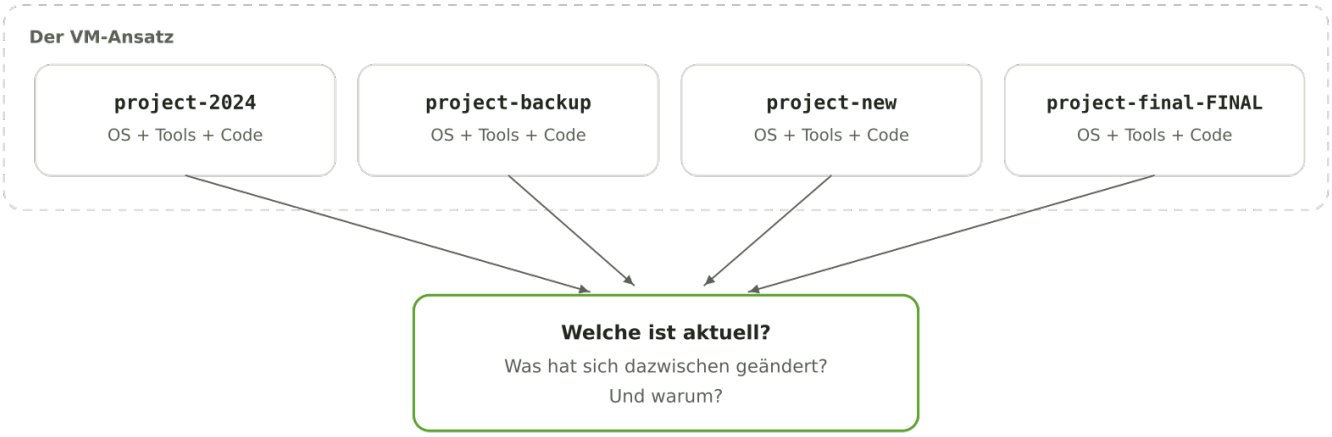
## Der Instinkt ist richtig. Der Mechanismus ist falsch.

Zuerst gebührt Anerkennung, wo sie verdient ist. Wer auf diese Weise zu VMs greift, hat eine echte Wahrheit verinnerlicht: **Ihr Quellcode ist ein wertvolles Gut und verdient Schutz.** Das ist mehr als die halbe Miete.

Die meisten, die auf Versionsverwaltung verzichten, tun das, weil sie *noch* nie etwas verloren haben. Wer zu VMs greift, versucht wenigstens aktiv, auf Nummer sicher zu gehen.

Das Problem ist, dass "sicher" mit dem falschen Mechanismus verdrahtet wurde. Eine VM schützt den *Schnappschuss einer Maschine*. Versionsverwaltung schützt die *Geschichte Ihres Codes*. Das ist nicht dasselbe, und genau dieser Unterschied ist der springende Punkt.

Hier ein hilfreiches Gedankenmodell. Stellen Sie sich vor, Sie haben zwölf virtuelle Maschinen, jede mit einer Kopie Ihres Projekts zu irgendeinem Zeitpunkt in der Vergangenheit. Was haben Sie damit tatsächlich?




Zwölf eingefrorene VM-Kopien — und keine Möglichkeit zu erkennen, welche zählt oder warum

Sie haben **mehrfache Kopien derselben Sache, keine davon zentral verwaltet.** Jede VM ist eine schwere, undurchsichtige Kiste von etlichen Gigabyte. Um herauszufinden, worin sich zwei davon unterscheiden, müssten Sie beide booten, beide Projekte öffnen und die Dateien von Hand vergleichen. Um die Frage "Warum habe ich diese Funktion im März geändert?" zu beantworten, müssten Sie sich erinnern, welche VM der März war — und dann raten. Es gibt keine Geschichte *mit Absicht* — nur einen Haufen eingefrorener Momente ohne roten Faden.

Das ist kein Versionsverwaltungssystem. Das ist ein Schuhkarton voller Polaroids.

## Was eine VM schlicht nicht kann

Lassen Sie mich konkret werden, denn "Glauben Sie mir, es ist schlimmer" ist kein Argument. Hier ist, was Versionsverwaltung Ihnen gibt und was eine Wand aus VM-Snapshots strukturell nicht leisten kann:

**Sie kann Ihnen nicht sagen, was sich geändert hat.** Ein Commit ist ein präziser Nachweis auf Zeilenebene:

Diese fünf Zeilen in dieser Unit haben sich geändert, von diesem Zustand zu jenem, an diesem Datum, aus diesem Grund. Ein VM-Snapshot ist ein Monolith — das Betriebssystem, der Compiler, Ihre IDE-Einstellungen und Ihr Code, alles miteinander verschmolzen. Daraus lässt sich keine Absicht herausdiffen.

**Sie kann Ihnen nicht sagen,* warum*. Das ist der Punkt, den die meisten unterschätzen. Eine gute Commit-Mes-*

*sage hält die Entscheidung fest* — "auf manuelles Parsen umgestellt, weil FireDACs Datumsbehandlung an den

regionalen Einstellungen des Kunden scheiterte."* Sechs Monate später ist dieser Satz Gold wert. In einer VM gibt es keinen Ort, an dem man diesen Satz aufschreiben könnte.

**Sie erlaubt Ihnen kein furchtloses Experimentieren.** Mit Git erstellen Sie einen Branch, probieren ein

riskantes Refactoring aus und mergen es entweder — oder werfen es weg. Sofort, ohne Kosten. Das VM-Äquiv-alent lautet: "Klone die komplette 40-GB-Maschine und hoffe, dass du dich erinnerst, welcher Klon die gute Idee war." Das macht in der Praxis niemand, was bedeutet, dass diese Entwickler faktisch *nicht experimentieren* —

und damit langsamer und zaghafter sind, als sie sein müssten.

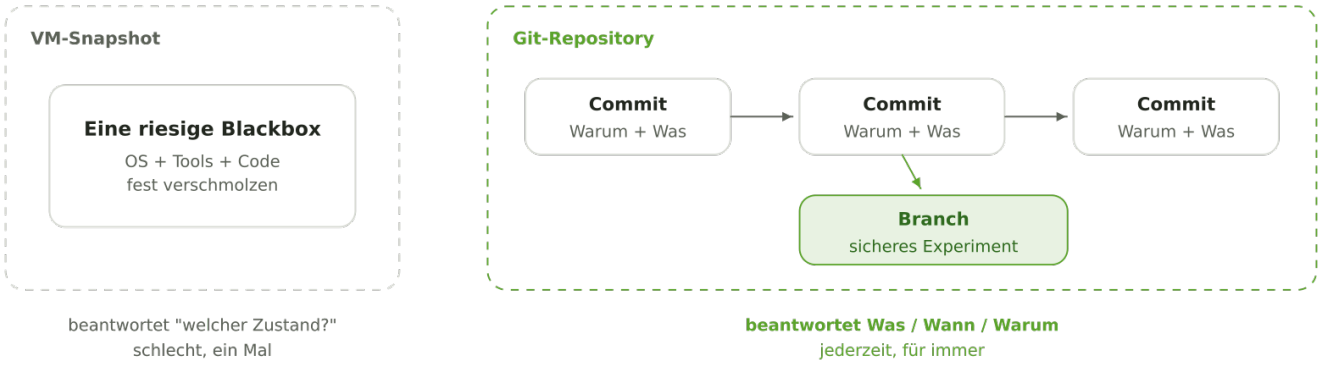
**Sie lässt sich nicht zentral verwalten.** Zwölf VMs auf zwölf Platten bedeuten zwölf Dinge, die synchron zu

halten sind, und keine einzige verlässliche Quelle der Wahrheit. In dem Moment, in dem Sie in zweien davon Code bearbeiten, haben Sie einen Fork, den Sie nie wieder zusammenführen werden. Ein Git-Repository hat genau eine kanonische Historie, auf die sich jeder Clone einigt.

**Sie skaliert nicht einmal auf eine einzelne Person herunter.** "Aber ich arbeite doch allein" ist der Einwand,

den ich als Nächstes höre, und er stellt die Sache auf den Kopf. Sie arbeiten sehr wohl mit jemandem zusammen: **mit Ihrem früheren Ich.** Das Ich vom letzten Frühjahr hat Entscheidungen getroffen, an die sich das

Ich von heute nicht mehr erinnert. Versionsverwaltung ist das einzige Werkzeug, mit dem das Gegenwarts-Ich das Vergangenheits-Ich befragen kann — und eine klare Antwort bekommt.




Ein VM-Snapshot ist eine undurchsichtige Kiste; ein Git-Repository ist eine verbundene Historie mit Absicht

Nichts davon bedeutet, dass VMs schlecht wären. VMs sind *hervorragend* — um Build-Umgebungen zu

isolieren, auf einem sauberen Betriebssystem zu testen oder einen alten Compiler zu betreiben, den Sie nativ nicht mehr installieren können. Nutzen Sie sie weiterhin genau dafür. Verlangen Sie nur nicht von ihnen, die Zeitmaschine für Ihren Quellcode zu sein — für diesen Job wurden sie nie gebaut.

## "Aber ich vertraue GitHub und keinem anderen Cloud-Anbieter"

An dieser Stelle wird das Gespräch meist erst richtig interessant, denn direkt hinter dem ersten Einwand steckt ein zweiter, völlig legitimer:

> "I don't trust GitHub, or any other cloud provider, with my code."

Also: "Ich vertraue meinen Code weder GitHub noch irgendeinem anderen Cloud-Anbieter an." Und hier ist der Satz, den ich laut sagen möchte, weil ihn zu viele Menschen nie zu hören bekommen:

> **Das müssen Sie auch nicht.**
> Versionsverwaltung bedeutet nicht GitHub. Git ist ein **verteiltes, offenes, selbst hostbares** System.
> GitHub ist *ein* gehostetes Produkt, das darauf aufbaut. Sie können jeden Vorteil professioneller Versionsverwaltung haben, ohne dass auch nur ein einziges Byte Ihres Codes jemals Hardware verlässt, die Ihnen physisch gehört.

Sagen Sie das jemandem mit genau dieser Sorge laut ins Gesicht, und Sie können zusehen, wie die Erleichterung eintritt. Der ganze Einwand beruht auf einer falschen Gleichung: *Git = GitHub = mein proprietärer Code in Microsofts Cloud.* Zerlegen Sie diese Gleichung, und der Einwand verpufft. Sie können Ihren eigenen Git-Server betreiben — auf einem übrig gebliebenen PC, einem NAS oder einem Raspberry Pi auf Ihrem Schreibtisch, auf Wunsch komplett vom Internet getrennt.

Ich habe früher schon darüber geschrieben, wie das mit **Gitea** geht, und das ist eine großartige Wahl. Heute möchte ich Sie aber auf **Forgejo** hinweisen, denn für einen Privacy-first-Entwickler, der niemandem vertrauen will, passt es besonders gut — und ich möchte Ihnen das vollständige Docker-Setup zeigen.

## Warum ausgerechnet Forgejo

Forgejo ist ein leichtgewichtiger, selbst gehosteter Git-Dienst — Repositories, Issues, Pull Requests, ein Web-Codebrowser, das volle Programm. Wenn Sie das an Gitea erinnert, dann deshalb, weil es tatsächlich ein Community-getriebener Fork von Gitea *ist* — und genau das ist für alle relevant, die Cloud-Anbietern misstrauen:

- **Es wird von einem gemeinnützigen Verein getragen** (Codeberg e.V.), nicht von einem Unternehmen, das übernommen werden, den Kurs wechseln oder anfangen kann, Ihr Vertrauen zu monetarisieren. Für jemanden, dessen Kerneinwand "Ich traue keiner Firma die Verwahrung meines Codes zu" lautet, ist das Governance-Modell das eigentliche Feature.
- **Es läuft vollständig auf Ihrer Hardware.** Kein Abo, kein Konto, keine Telemetrie, die Sie nicht bestellt haben, keine Internetabhängigkeit. Ihr Code liegt auf Ihrer Platte, Punkt.
- **Es ist wirklich leichtgewichtig.** Es läuft problemlos auf einem Raspberry Pi. Sie brauchen keinen dicken Server für einen professionellen Git-Host.
- **Es fügt sich nahtlos in das gesamte Git-Ökosystem ein.** Standard-Git über SSH und HTTPS — jeder Git-Client der Welt, GitKraken eingeschlossen, spricht nativ mit ihm.

Mit anderen Worten: Es beantwortet den "Ich traue der Cloud nicht"-Einwand *vollständig*, denn es gibt keine Cloud. Es gibt nur Ihre Maschine.

## Forgejo mit Docker Compose einrichten

Machen wir es konkret. Das hier ist ein vollständiges, produktionsreifes Setup, das Sie auf jeder Linux-Maschine in Ihrem Netzwerk betreiben können — einem übrigen PC, einem NAS, einem Pi. Das Wesentliche dauert

deutlich unter fünfzehn Minuten; der Rest dieses Abschnitts ist der komplette Quelltext, damit Sie sich per Copy-and-paste zu einem laufenden Server durcharbeiten können.

## Voraussetzungen

- Docker und Docker Compose installiert
- Eine Maschine mit mindestens 1 GB RAM (512 MB funktionieren; 1 GB ist komfortabel)
- Fünfzehn Minuten

## Die Docker-Compose-Datei

Legen Sie ein Verzeichnis für Ihre Forgejo-Installation an — etwa `~/forgejo` — und erstellen Sie darin eine Datei namens `docker-compose.yml`:

```
services:
  forgejo:
    image: codeberg.org/forgejo/forgejo:9
    container_name: forgejo
    environment:
      - USER_UID=1000
      - USER_GID=1000
      - FORGEJO__database__DB_TYPE=postgres
      - FORGEJO__database__HOST=db:5432
      - FORGEJO__database__NAME=forgejo
      - FORGEJO__database__USER=forgejo
      - FORGEJO__database__PASSWD=forgejo_secret_password
      # Damit Links und SSH-Clone-URLs den echten Host verwenden, nicht "localhost":
      - FORGEJO__server__DOMAIN=192.168.1.100
      - FORGEJO__server__ROOT_URL=http://192.168.1.100:3000/
      - FORGEJO__server__SSH_DOMAIN=192.168.1.100
      - FORGEJO__server__SSH_PORT=2222
    restart: always
    volumes:
      - forgejo_data:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
    ports:
      - "3000:3000"   # Weboberfläche
      - "2222:22"     # SSH für Git-Operationen
    depends_on:
      - db
    networks:
      - forgejo_network

  db:
    image: postgres:16-alpine
    container_name: forgejo_db
    restart: always
    environment:
      - POSTGRES_USER=forgejo
      - POSTGRES_PASSWORD=forgejo_secret_password
      - POSTGRES_DB=forgejo
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - forgejo_network

volumes:
  forgejo_data:
  postgres_data:

networks:
  forgejo_network:
    driver: bridge
```

Ein paar Dinge sind hervorzuheben, weil genau daran die meisten hängen bleiben:

- **Ersetzen Sie `192.168.1.100` durch die tatsächliche LAN-IP** (oder den Hostnamen) der Maschine, auf der das Ganze läuft. Wenn Sie `DOMAIN`, `ROOT_URL` und `SSH_DOMAIN` von Anfang an setzen, funktionieren die Clone-URLs, die Forgejo Ihnen anzeigt, auch von Ihren anderen Rechnern aus — statt auf `localhost` zu zeigen.
- **Ändern Sie `forgejo_secret_password`** in etwas Echtes, bevor Sie das starten. Es taucht an zwei Stellen auf — beim Forgejo-Dienst und bei der Datenbank — und beide müssen übereinstimmen.
- Die Datenbank und alle Repositories liegen in **benannten Docker-Volumes**, sodass Ihre Daten Container-Neustarts und Upgrades überleben. (Genau dorthin richten Sie auch Ihr *echtes* Backup — denken Sie daran: Versionsverwaltung ist kein Backup, eine Lektion, die ich nicht müde werde zu wiederholen.)

## Forgejo starten

Öffnen Sie ein Terminal in Ihrem Verzeichnis `~/forgejo` und führen Sie aus:

```
docker compose up -d
```

Geben Sie dem Ganzen einen Moment, öffnen Sie dann einen Browser und rufen Sie auf:

```
http://192.168.1.100:3000
```

Sie sehen den Einrichtungsbildschirm von Forgejo. Die Datenbankfelder sind bereits aus Ihrer Compose-Datei vorausgefüllt — bestätigen Sie sie einfach, legen Sie Ihr Administratorkonto an, und fertig. Sie haben jetzt einen privaten, selbst gehosteten Git-Host, der niemandem Rechenschaft schuldet außer Ihnen.

## Ein bestehendes Delphi-Projekt hineinbringen

Sobald Forgejo läuft, ist der Umzug eines Projekts der übliche Git-Dreisatz:

```
# In den bestehenden Projektordner wechseln
cd /path/to/your/delphi/project

# Ein Git-Repository initialisieren
git init

# VOR dem ersten Add eine .gitignore für Delphi anlegen (siehe unten)
git add .
git commit -m "Initial commit"

# Den Forgejo-Server als Remote hinzufügen (das leere Repository vorher in der Web-UI anlegen)
git remote add origin http://192.168.1.100:3000/yourusername/yourproject.git

# Pushen
git push -u origin main
```

## Eine sinnvolle `.gitignore` für Delphi

Committen Sie keine kompilierten Binaries, DCU-Dateien oder IDE-Kram — das ist Build-Ausgabe, kein Quellcode. Legen Sie das hier in Ihr Projektstammverzeichnis:

```
# Kompilierte Delphi-Ausgabe
*.dcu
*.dcp
*.dcpil
*.dpu
*.bpl
*.bpi
*.lib
*.exe
*.dll
*.map
*.drc

# Lokale IDE-Einstellungen
*.local
*.identcache
*.tvsconfig
*.dsk

# Sicherungsdateien
*.~*
__history/

# Ausgabeverzeichnisse
Win32/
Win64/
Android/
iOS/
OSX/

# Paket-Cache
*.cache
```

## Jetzt machen Sie es sich bequem: GitKraken + der Versionsbaum

Und hier kommt der Teil, der aus "Ich habe einen Git-Server aufgesetzt" ein "Ich benutze Versionsverwaltung tatsächlich gern" macht. Ein Server ist nur die halbe Miete — Sie wollen auch einen Client, der die Historie *sichtbar* macht. Denn der ganze Grund, warum ich Sie von den VM-Snapshots wegziehe, ist, dass Sie die

Zeitachse Ihres Projekts endlich *sehen* können. Dieser Client ist für mich GitKraken.

Das eine Feature, das am zuverlässigsten Skeptiker bekehrt, ist der **Versionsbaum** — GitKrakens visueller

Commit-Graph. Er zeigt Ihr gesamtes Repository als klare, interaktive Zeitachse: jeder Commit, jeder Branch, jeder Merge, jedes Tag, so angeordnet, dass Sie die Geschichte des Projekts auf einen Blick lesen können. Das ist *genau* das, was Ihre Wand aus VMs Ihnen nie geben konnte. Statt "welche eingefrorene Kiste war noch

mal der März" bekommen Sie eine beschriftete, navigierbare Linie, durch die Sie sich klicken können. Für viele Entwickler ist der Moment, in dem sie ihre eigene Historie so dargestellt sehen, genau der Moment, in dem Versionsverwaltung endlich Klick macht.

Und weil so viele von uns im Editor leben, integriert sich GitKraken direkt in **Visual Studio Code**. Sie

bekommen denselben Versionsbaum, Commit-Graph und Staging-Workflow direkt neben Ihrem Code, ohne in eine separate App wechseln zu müssen. Committen, branchen, reviewen und in der Historie stöbern — genau dort, wo Sie ohnehin arbeiten.

Ein paar praktische Hinweise, damit Sie nicht überrascht werden:

- **Fügen Sie Forgejo als benutzerdefinierte/self-hosted Integration** in GitKraken hinzu, authentifizieren

  Sie sich einmal, und GitKraken behandelt Ihren privaten Server wie jeden anderen Host — Clonen, Pushen, Pullen, Branchen, alles visuell.

- **Der kostenlose Tarif von GitKraken deckt Self-hosted-Server nicht ab.** Sie brauchen mindestens den Pro-Plan, um sich mit einer selbst gehosteten Forgejo-Instanz zu verbinden. Wenn Sie es mit Ihrem Handwerk ernst meinen, ist das eine leichte Entscheidung — der Komfort zahlt sich von selbst aus, gerade während Sie die Gewohnheit aufbauen.
- Eine Komfortlücke, damit die Erwartungen stimmen: GitKrakens **Pull-Request-Funktionen erstrecken sich nicht auf Self-hosted-Server**, PRs erstellen Sie also in der Forgejo-Web-UI. Eine Kleinigkeit, die den Alltag aus Committen und Historie-Stöbern nicht berührt — und genau dort liegt der Wert.

## Das ehrliche Fazit

Zurück zum Anfang: "Ich benutze VMs, mein Code ist also sicher."

Der Instinkt ist richtig: Ihr Code verdient Schutz. Aber ein Stapel virtueller Maschinen liefert Ihnen **mehrfache, undurchsichtige, unkoordinierte Kopien** — einen Schuhkarton voller Polaroids —, während Sie in Wirklichkeit **eine verwaltete Historie wollen, die beantwortet, was sich wann und warum geändert hat.** VMs sind ein wunderbares Werkzeug für die Aufgabe, für die sie gebaut wurden; die Zeitmaschine Ihres Codes zu sein gehört nicht dazu.

Und wenn der Grund, warum Sie die "offensichtliche" Antwort gemieden haben, darin liegt, dass Sie GitHub oder irgendeinem Cloud-Anbieter nicht vertrauen — gut so. Auch das ist ein gesunder Instinkt, und Sie müssen ihn nicht aufgeben. Forgejo in einem Docker-Container gibt Ihnen die volle professionelle Versionsverwaltungserfahrung auf Hardware, die Ihnen gehört, verbunden mit nichts, das Sie nicht kontrollieren. Kombinieren Sie das mit GitKrakens Versionsbaum — in der eigenen App oder direkt in VS Code — und Sie haben ein Setup, das mit dem jedes Großunternehmens mithält, nur ohne das Großunternehmen und ohne die Cloud.

Fünfzehn Minuten. Ihre Hardware. Ihre Historie. Keine Cloud, keine Ausreden.

## Brauchen Sie Starthilfe?

Wenn Sie Delphi-Entwickler sind — allein oder im Team — und diesen Sprung schon lange vorhaben, aber nicht wissen, wo Sie anfangen sollen: Genau das ist die Art von Arbeit, die ich mache. Einen selbst gehosteten Git-Host aufsetzen, bestehende Projekte sicher migrieren, ein vernünftiges Branching etablieren und einen Client wie GitKraken so einrichten, wie er gehört. Melden Sie sich.

*Im Jahr 2026 sollte kein professioneller Entwickler sein Lebenswerk einem Ordner voller eingefrorener virtueller Maschinen anvertrauen. Ändern wir das — ein Repository nach dem anderen.*

INDEPENDENT · AD-FREE · READER-SUPPORTED

# Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

## Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

## Ways to support

KO-FI

### Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

### Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

### Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

**Support on Ko-fi —**