NEWS · OCTOBER 9, 2026

Next.js-Merge-Konflikte in GitKraken: Ein echter Konflikt, Schritt für Schritt

Ein echter Merge-Konflikt in einem Next.js-Header -- zwei Teammitglieder, drei kollidierende Hunks, eine Lockfile -- Schritt für Schritt in GitKraken Desktop gelöst, einschließlich der Lösung, die den Build besteht und trotzdem ein Feature löscht.

By Dr. Holger Flick

Ihr Pull Request war heute Morgen noch grün. Dann ist der PR einer Kollegin zuerst auf `main` gelandet, und jetzt trägt Ihrer den kleinen Hinweis, den jeder Teamentwickler kennt: Dieser Branch hat Konflikte, die gelöst werden müssen. In einer Next.js-Codebasis sind das meistens dieselben drei Verdächtigen -- eine gemeinsam genutzte Layout- oder Header-Komponente, `package.json` und die Lockfile, die niemand je lesen möchte.

Merge-Konflikte haben einen Ruf, den sie nur zum Teil verdienen. Meiner Meinung nach ist der Konflikt selbst selten das Problem. Das Problem ist, ihn blind zu lösen: auf `<<<<<<<`-Marker in einem Texteditor zu starren, eine Seite zu wählen, damit das Rot verschwindet, und zu hoffen, dass der Build schon meldet, wenn etwas schiefgelaufen ist. Kleiner Spoiler: Der Build meldet es nicht immer.

In diesem Beitrag gehen wir einen echten Konflikt durch -- kein geschöntes Zwei-Zeilen-Beispiel -- in einer kleinen Next.js-App, von der frühen Warnung bis zum Merge-Commit, mit GitKraken Desktop. Jede Termi-

nalausgabe und jedes Build-Ergebnis unten stammt aus einem echten Durchlauf. Wir sehen uns auch die eine Lösung an, die kompiliert, den Type-Check besteht und trotzdem ein Feature wegwirft. Genau ihretwegen habe ich diesen Beitrag geschrieben.

Die Ausgangslage: zwei Teammitglieder, ein Header

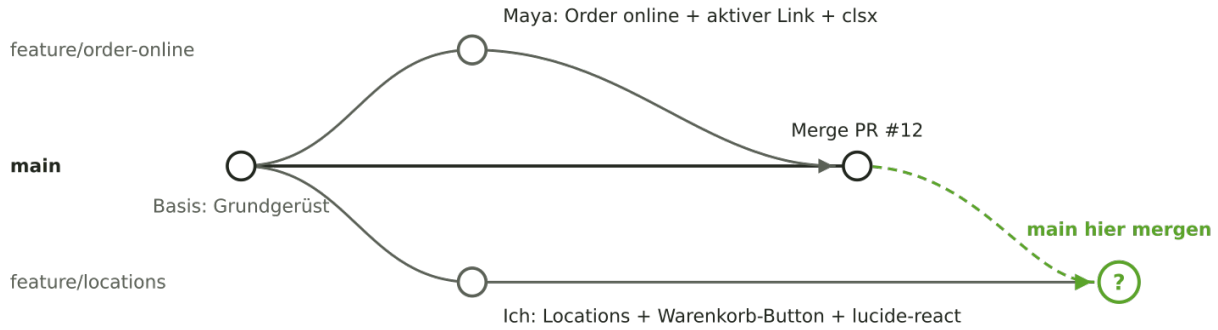
Bevor wir etwas lösen können, brauchen wir einen Konflikt, der sich lohnt, also bauen wir einen realistischen. Die App ist die Website einer fiktiven Restaurantkette -- Los Pollos Hermanos, natürlich --, gebaut mit Next.js 16 und dem App Router. Ihre `SiteHeader`-Komponente auf `main` ist so schlicht wie nur möglich:

```
import Link from "next/link";

const links = [
  { href: "/", label: "Home" },
  { href: "/menu", label: "Menu" },
];

export function SiteHeader() {
  return (
    <header className="flex items-center justify-between border-b px-6 py-4">
      <Link href="/" className="font-bold">
        Los Pollos Hermanos
      </Link>
      <nav className="flex gap-4">
        {links.map((link) => (
          <Link key={link.href} href={link.href}>
            {link.label}
          </Link>
        ))}
      </nav>
    </header>
  );
}
```

Nun übernehmen zwei Personen am selben Morgen zwei Tickets. Nehmen wir an, meine Kollegin Maya arbeitet an `feature/order-online`. Sie fügt einen Link „Order online“ hinzu und hebt die aktive Seite hervor. Dafür braucht sie `usePathname`, das nur in einer Client Component funktioniert, also ergänzt sie die Direktive `"use client"`. Außerdem fügt sie `clsx` hinzu, einen winzigen Helfer für bedingte Klassennamen, und blendet die Navigation auf kleinen Bildschirmen aus.

Ich arbeite an `feature/locations`. Ich füge einen Link „Locations“ hinzu, einen kleinen Warenkorb-Button mit einem Icon aus `lucide-react`, und gebe der Navigation etwas mehr Abstand. Keiner von uns hat etwas falsch gemacht. Wir haben beide nur die zentralste Komponente der App angefasst, und genau das passiert in echten Teams jede Woche.

Mayas PR wird zuerst gemergt. Die Historie sieht jetzt so aus, und mein Branch ist derjenige, der aufholen muss.



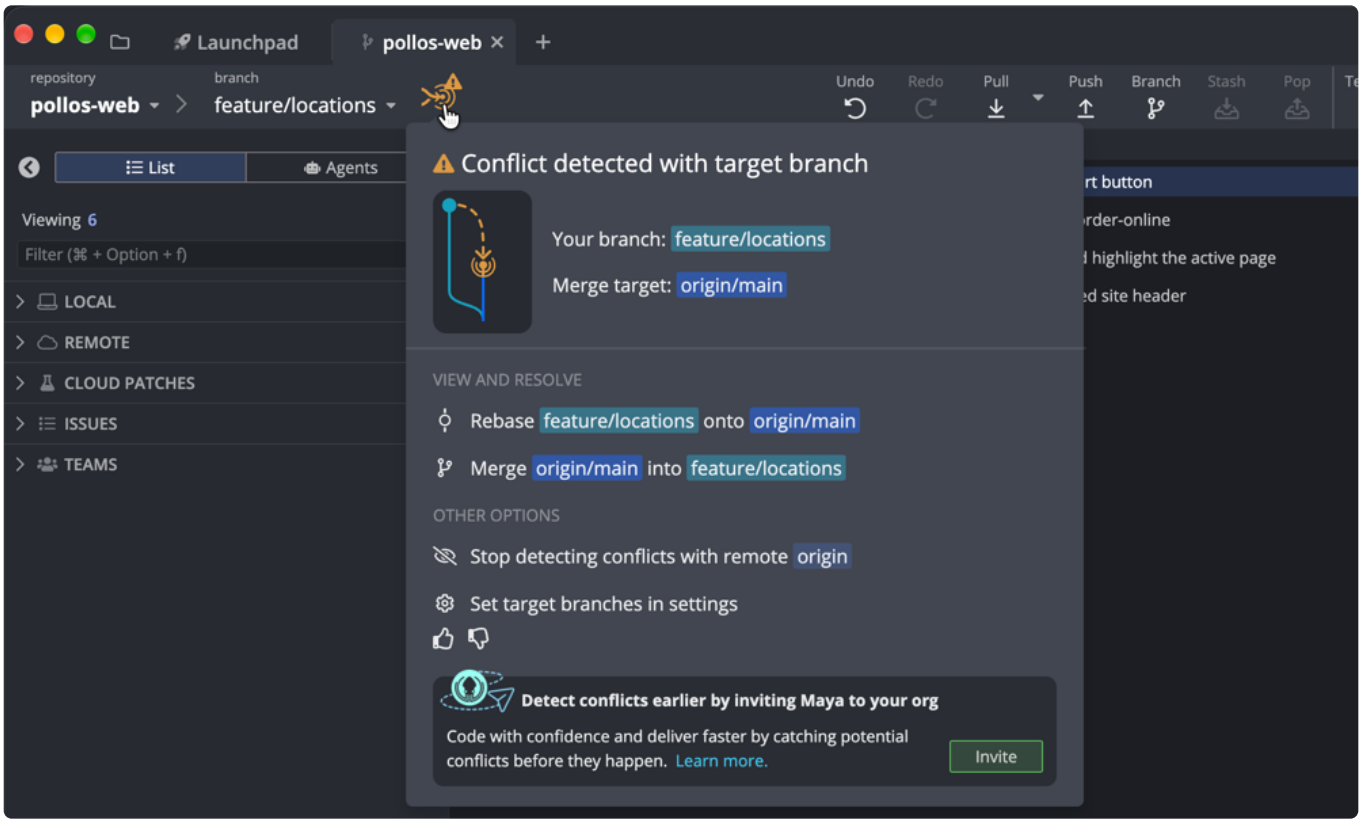
Mayas PR ist zuerst auf main gelandet -- jetzt muss main in meinen Branch gemergt werden

Der gestrichelte Pfeil ist der Merge, den wir gleich durchführen. Beide Branches sind vom selben Basis-Commit ausgegangen, und dieser gemeinsame Vorfahre ist das, was den Rest dieses Beitrags überhaupt möglich macht.

Schritt 1: Den Konflikt sehen, bevor Sie mergen

Der billigste Konflikt ist der, von dem Sie wissen, bevor Sie den Merge starten, also suchen wir zuerst nach ihm. GitKraken Desktop hat eine Funktion namens Conflict Prevention, die Ihren Branch gegen seinen Ziel-Branch

prüft und ein Warnsymbol anzeigt, wenn ein Merge wahrscheinlich kollidieren wird. Sind Ihre Teammitglieder Mitglieder Ihrer GitKraken-Org, geht sie einen Schritt weiter und meldet auch überlappende Änderungen in deren bereits committeter, aber noch nicht gemergter Arbeit -- mit der Möglichkeit, Ihre Änderungen als Cloud Patch zu teilen oder eine Zusammenfassung der Überschneidung für Ihr Team zu kopieren.



Conflict Prevention, bevor der Merge überhaupt beginnt: mein Branch, sein Merge-Ziel und die beiden Wege, den Konflikt zu lösen -- Rebase oder Merge.

In meinem Szenario sitzt das Warnsymbol direkt neben dem Branch-Namen, bevor ich auch nur einen Merge-Befehl getippt habe. Ein Klick darauf zeigt „Conflict detected with target branch“, nennt meinen Branch und das Merge-Ziel `origin/main` und bietet direkt im Menü an, meinen Branch auf `origin/main` zu rebasen oder `origin/main` in ihn zu mergen. Es hat sogar bemerkt, mit wessen Arbeit ich kollidiere, und schlägt vor, Maya in die Org einzuladen, damit die nächste Überschneidung schon auffällt, solange ihre Änderungen noch auf ihrem Branch liegen. Im Team ist das der Moment für eine kurze Nachricht an Maya („Ich arbeite auch am Header, ich merge main jetzt rein“), statt die Überschneidung erst im Code-Review zu entdecken. Beachten Sie, dass Conflict Prevention eine kostenpflichtige Funktion ist; die GitKraken-Dokumentation führt sie für das Pro-Abo und höher.

Sie fragen sich vielleicht, ob reines Git das auch kann. Es kann, mit etwas Zeremonie. Seit Git 2.38 berechnet

`git merge-tree --write-tree` einen Merge komplett im Speicher und liest laut Dokumentation weder aus

Working Tree noch Index, noch schreibt es dorthin („does not read from or write to either the working tree or index“). Das ist die echte Ausgabe für unsere Branches:

```
$ git merge-tree --write-tree --name-only feature/locations main
8dcbff5bfcf79df7704fc8d925cb591dbe81f487
components/site-header.tsx
package.json
pnpm-lock.yaml

Auto-merging components/site-header.tsx
CONFLICT (content): Merge conflict in components/site-header.tsx
Auto-merging package.json
CONFLICT (content): Merge conflict in package.json
Auto-merging pnpm-lock.yaml
CONFLICT (content): Merge conflict in pnpm-lock.yaml
```

Der Befehl endet mit Status 1, wenn der Merge kollidieren würde, was ihn für Skripte und CI nützlich macht. Der Unterschied zu GitKraken liegt schlicht darin, wer daran denken muss zu fragen. Die CLI antwortet, wenn Sie sie aufrufen; Conflict Prevention beobachtet weiter, während Sie arbeiten.

Schritt 2: Mergen -- und ansehen, was Git schon für Sie erledigt hat

Von einem Konflikt zu wissen, lässt ihn nicht verschwinden, also mergen wir jetzt `main` in `feature/locations`. In GitKraken ziehen Sie `main` auf Ihren Branch im Graphen, klicken mit rechts auf `main` und wählen Merge oder klicken einfach im Conflict-Prevention-Menü aus Schritt 1 auf „Merge `origin/main` into `feature/locations`“. Das Ergebnis ist dasselbe wie auf der Kommandozeile:

```
$ git merge main
Auto-merging components/site-header.tsx
CONFLICT (content): Merge conflict in components/site-header.tsx
Auto-merging package.json
CONFLICT (content): Merge conflict in package.json
Auto-merging pnpm-lock.yaml
CONFLICT (content): Merge conflict in pnpm-lock.yaml
Automatic merge failed; fix conflicts and then commit the result.
```

Drei Dateien mit Konflikten, und an dieser Stelle steigt bei den meisten der Puls. Bevor wir irgendetwas anfassen, lohnt es sich aber zu verstehen, was Git eigentlich getan hat. Git führt einen Drei-Wege-Merge durch: Es vergleicht jeden Bereich der Datei mit dem gemeinsamen Vorfahren. Hat nur eine Seite einen Bereich geändert, übernimmt Git diese Änderung ohne Rückfrage. Erst wenn *beide* Seiten denselben Bereich geändert haben, hält es an und überlässt Ihnen die Entscheidung.

Für unseren Header sieht dieser Vergleich Bereich für Bereich so aus.

BEREICH IN site-header.tsx	ICH	MAYA (main)	GITS URTEIL
`"use client";`	--	geändert	still gemergt
`Import-Zeilen`	geändert	geändert	**KONFLIKT**
`links-Array`	geändert	geändert	**KONFLIKT**
`<nav className>`	geändert	geändert	**KONFLIKT**
`<Link>-Props (aria-current, clsx)`	--	geändert	still gemergt
`<CartButton />`	geändert	--	still gemergt

Git fragt nur bei den Bereichen nach, die beide Seiten geändert haben -- alles andere wird stillschweigend gemergt

Drei Bereiche kollidieren, drei wurden ohne ein Wort gemergt. Das ist Git, das seine Arbeit gut macht, und meistens ist es genau das, was Sie wollen. Sehen Sie sich aber die erste Zeile an. Mayas `"use client"` wurde stillschweigend in meine Version der Datei gemergt. Mein Warenkorb-Button, den ich innerhalb einer Server Component geschrieben und getestet habe, rendert jetzt in einer Client Component und wird an den Browser ausgeliefert. Hier ist das harmlos. In einer Komponente, die ein Secret liest oder direkt die Datenbank abfragt, wäre es das nicht. In einer Next.js-Codebasis heißt „kein Konflikt“ also nicht „nichts anzusehen“. Ein Merge kann die Grenze zwischen Server und Client verschieben, ohne einen einzigen Konfliktmarker.

So sieht die Datei gerade auf der Festplatte aus:

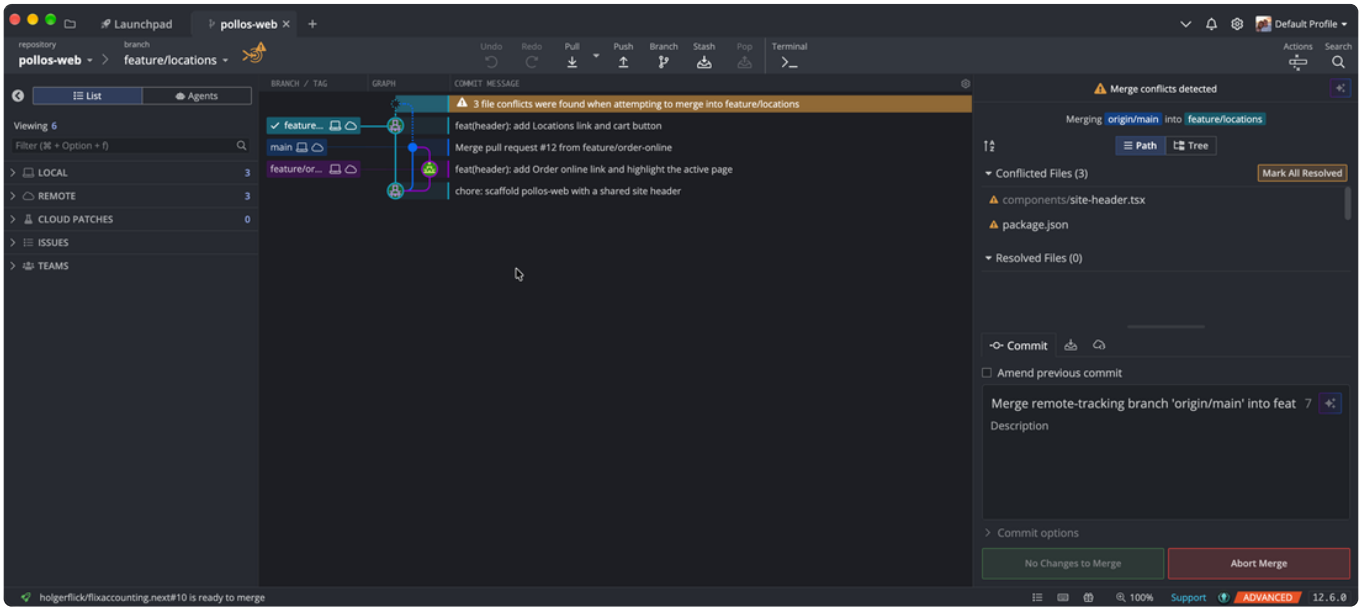
```
"use client";

import Link from "next/link";
<<<<<<< HEAD
import { CartButton } from "./cart-button";
=======
import { usePathname } from "next/navigation";
import clsx from "clsx";
>>>>>>> main

const links = [
  { href: "/", label: "Home" },
  { href: "/menu", label: "Menu" },
<<<<<<< HEAD
  { href: "/locations", label: "Locations" },
=======
  { href: "/order", label: "Order online" },
>>>>>>> main
];

export function SiteHeader() {
  const pathname = usePathname();

  return (
    <header className="flex items-center justify-between border-b px-6 py-4">
      <Link href="/" className="font-bold">
        Los Pollos Hermanos
      </Link>
<<<<<<< HEAD
      <nav className="flex items-center gap-6">
=======
      <nav className="hidden gap-4 md:flex">
>>>>>>> main
        {links.map((link) => (
          <Link
```

```
            key={link.href}
            href={link.href}
            aria-current={pathname === link.href ? "page" : undefined}
            className={clsx(pathname === link.href && "font-semibold underline")}
          >
            {link.label}
          </Link>
        ))}
        <CartButton count={0} />
      </nav>
    </header>
  );
}
```

`HEAD` ist mein Branch, `main` ist Mayas gemergte Arbeit. Lesbar? Sicher, bei 48 Zeilen. Stellen Sie sich das jetzt in einer 400-Zeilen-Seitenkomponente mit acht Hunks vor, freitags um 17:30 Uhr. Genau daher kommt die Panik.

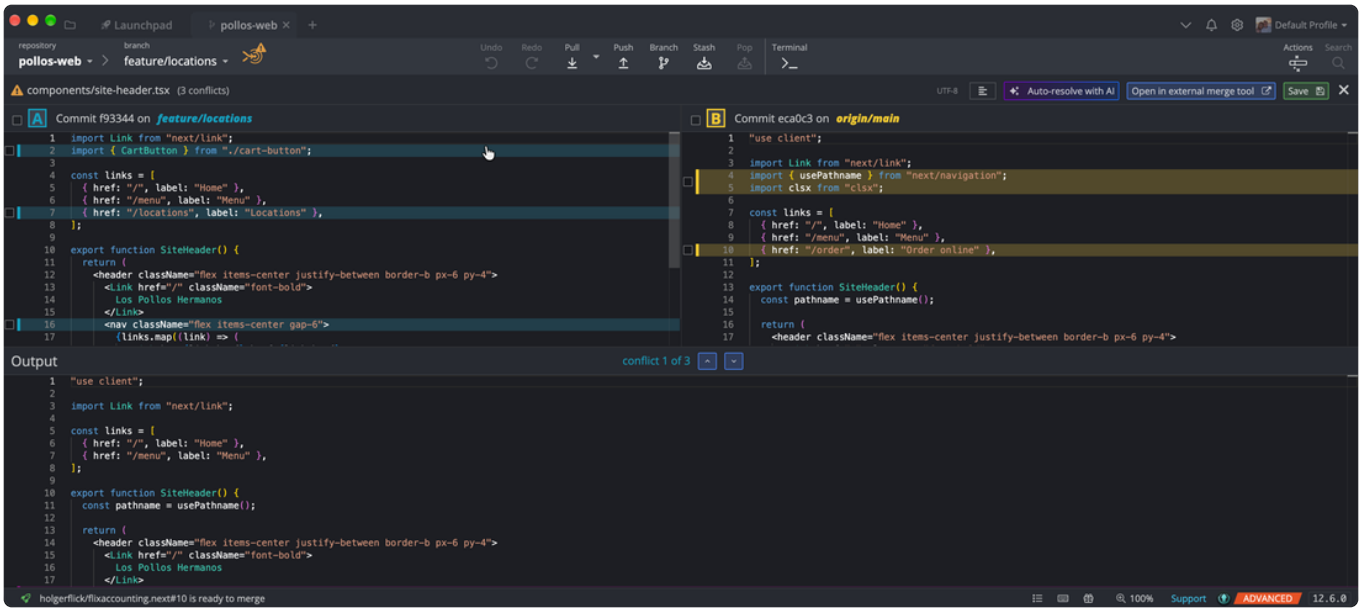
Schritt 3: Den Konflikt in GitKraken öffnen

Genau hier verdient sich ein visuelles Werkzeug seinen Platz, also wechseln wir von der Textdatei zu GitKraken. Nach dem fehlgeschlagenen Merge meldet ein Banner über dem Graphen „3 file conflicts were found when attempting to merge into feature/locations“, und das Commit Panel wechselt in den Merge-Modus: die Dateien mit Konflikten, eine noch leere Liste gelöster Dateien und ein roter Button **Abort Merge** genau dort, wo sonst der Commit-Button sitzt.



Der Merge hat mit drei Dateien voller Konflikte angehalten. Beachten Sie den Button Abort Merge unten rechts -- und das ausgegraute Undo in der Werkzeugleiste.

Ein Klick auf eine Datei öffnet das Merge Tool: mein Branch als **A** links, `origin/main` als **B** rechts, jeweils mit seinem Commit beschriftet, und unten der **Output**. Die Ausgabe enthält bereits alles, was Git selbst gemergt hat -- `"use client"` und `usePathname` stehen schon da -- und lässt die kollidierenden Bereiche für Sie offen. Ein Zähler zeigt „conflict 1 of 3“, mit Pfeilen, um zwischen den Konflikten zu springen.



Das Merge Tool für <code>site-header.tsx</code>: mein Branch (A) links, <code>origin/main</code> (B) rechts, unten die Ausgabe mit Syntax-Highlighting.

Sie lösen einen Hunk, indem Sie die Checkboxen bei den Zeilen anklicken, die in die Ausgabe sollen; eine Checkbox im Kopf jeder Seite übernimmt die ganze Seite auf einmal. Das Ausgabefeld hat Syntax-Highlighting und ist editierbar, was wichtiger ist, als es klingt, wie wir gleich sehen werden. Sobald Sie die Ausgabe speichern, stagt GitKraken die Datei, und Sie gehen zur nächsten über. Sie arbeiten einen Merge also Datei für Datei und

Hunk für Hunk ab, statt drei Dateien voller Marker auf einmal vor sich zu haben.

Mir gefällt dieser Ansatz, weil er die Frage verändert, die man sich stellt. Mit Markern in einer Textdatei lautet die Frage: „Welchen Block lösche ich?“ Mit zwei Seiten und einer Live-Ausgabe wird daraus: „Wie soll dieser Code aussehen?“ Das ist die bessere Frage.

Schritt 4: Drei Hunks, drei verschiedene Antworten

Die drei Hunks in unserem Header sehen sich ähnlich, aber jeder braucht eine andere Entscheidung, also gehen wir sie einzeln durch.

1. **Die Imports.** Ich habe `CartButton` hinzugefügt, Maya `usePathname` und `clsx`. Die Datei braucht alle drei, weil die still gemergten Teile der Datei alle drei verwenden. Beide Seiten anhaken.
2. **Das links-Array.** Ich habe „Locations“ hinzugefügt, Maya „Order online“. Auch hier gehören beide ins Ergebnis. Die einzige echte Entscheidung ist die Reihenfolge der Menüpunkte, und das ist eine Produktfrage, keine Git-Frage. Beide Seiten anhaken, in der gewünschten Reihenfolge.
3. **Die `<nav>`-Zeile.** Hier ist „beide nehmen“ falsch. Wir haben beide dasselbe Attribut desselben Elements geändert, und die Ausgabe darf nur ein öffnendes `<nav>`-Tag enthalten.

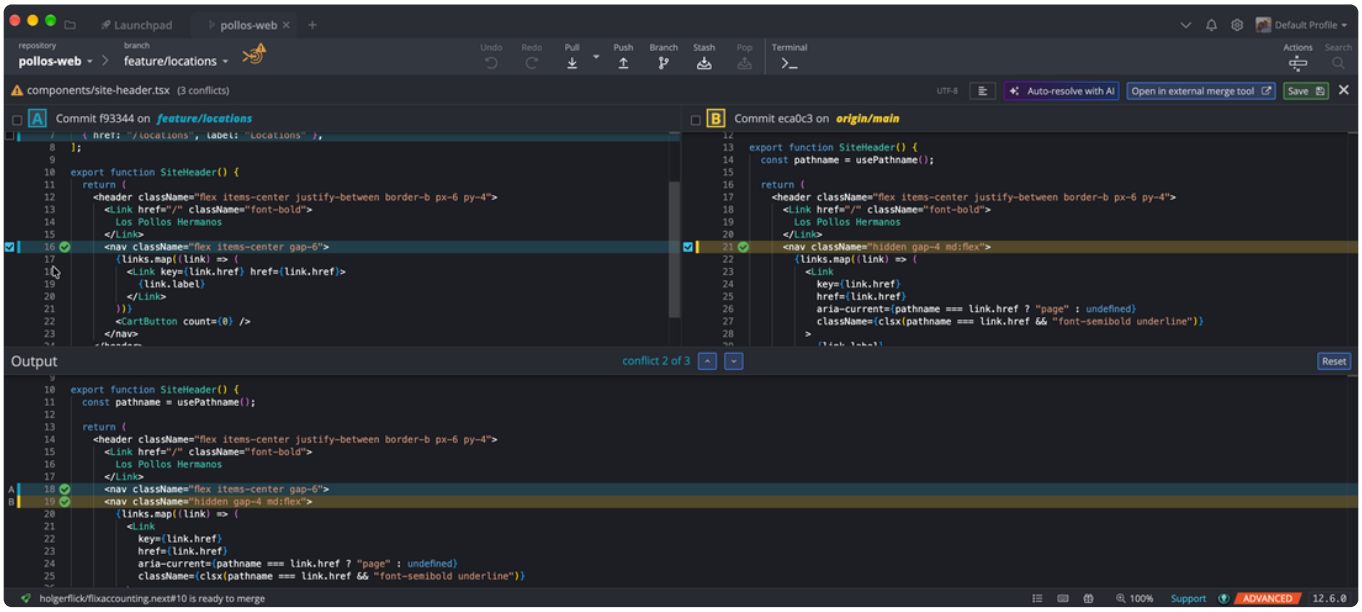
Sie denken vielleicht, der dritte Hunk sei zu offensichtlich, um ihn falsch zu machen. Ehrlich, Hand aufs Herz: Ich habe es absichtlich ausprobiert, um zu sehen, was passiert, und das Ergebnis ist lehrreich. Beide Seiten zu übernehmen erzeugt zwei öffnende `<nav>`-Tags. TypeScript bemerkt es:

```
$ pnpm typecheck
components/site-header.tsx(23,8): error TS17008: JSX element 'nav' has no corresponding closing
tag.
```

`next build` bemerkt es ebenfalls, aber Turbopack meldet den Parse-Fehler in Zeile 39 -- der schließenden Klammer der Komponente, sechzehn Zeilen unterhalb des eigentlichen Fehlers:

```
$ pnpm build
² Next.js 16.3.8 (Turbopack)
  Creating an optimized production build ...
> Build error occurred
Error: Turbopack build failed with 1 error:
./components/site-header.tsx:39:1
Error: Unexpected token. Did you mean `{'}'}` or `&rbrace;`?
```

Keine Überraschung: Ein Parser bemerkt ein fehlendes schließendes Tag erst, wenn ihm die Datei ausgeht. Genau deshalb sehe ich die Ausgabe lieber, während ich sie zusammenstelle, als hinterher eine Compiler-Meldung zu lesen. Im Merge Tool steht das doppelte `<nav>` in dem Moment im Ausgabefeld, in dem Sie die zweite Checkbox anklicken.



Beide Seiten des <code><nav></code>-Hunks angehakt: Die zwei öffnenden Tags stehen in der Ausgabe direkt untereinander, links mit A und B markiert -- kein Build nötig, um es zu bemerken.

Die richtige Antwort für diesen Hunk ist keine der beiden Seiten. Maya wollte die Navigation auf kleinen Bildschirmen ausblenden; ich wollte mehr Abstand und vertikale Zentrierung. Also wählen wir eine Zeile und bearbeiten sie direkt im Ausgabefeld, sodass beide Absichten erhalten bleiben:

```
      <nav className="hidden items-center gap-6 md:flex">
```

Das ist der Fall der manuellen Bearbeitung, und genau ihn erreicht man mit einer „Wähle eine Seite“-Denkweise nie. Nach dem Speichern ist dies die vollständig gelöste Datei. Sie besteht den Type-Check und baut.

```
"use client";

import Link from "next/link";
import { CartButton } from "./cart-button";
import { usePathname } from "next/navigation";
import clsx from "clsx";

const links = [
  { href: "/", label: "Home" },
  { href: "/menu", label: "Menu" },
  { href: "/locations", label: "Locations" },
  { href: "/order", label: "Order online" },
];

export function SiteHeader() {
  const pathname = usePathname();

  return (
    <header className="flex items-center justify-between border-b px-6 py-4">
      <Link href="/" className="font-bold">
        Los Pollos Hermanos
      </Link>
      <nav className="hidden items-center gap-6 md:flex">
        {links.map((link) => (
          <Link
            key={link.href}
            href={link.href}
            aria-current={pathname === link.href ? "page" : undefined}
            className={clsx(pathname === link.href && "font-semibold underline")}
          >
            {link.label}
          </Link>
        ))}
        <CartButton count={0} />
      </nav>
    </header>
  );
}
```

Die Lösung, die kompiliert und trotzdem falsch ist

Jetzt zu dem Szenario, das Teams tatsächlich Features kostet, und es hat nichts mit Syntaxfehlern zu tun. Unter Zeitdruck ist der schnellste Weg aus einem Konflikt, eine Seite der ganzen Datei zu nehmen. Auf der Kommandozeile ist das `git checkout --theirs`, das für einen ungemergten Pfad „stage #3 (theirs)“ auscheckt. Genau das machen wir jetzt für den Header, lösen die anderen beiden Dateien ordentlich und lassen die Prüfungen laufen:

```
$ git checkout --theirs components/site-header.tsx
Updated 1 path from the index
$ # package.json and pnpm-lock.yaml resolved as shown in the next section
$ pnpm typecheck
$ pnpm build
 Compiled successfully in 862ms
  Finished TypeScript in 560ms ...
 Generating static pages using 4 workers (3/3) in 159ms
```

Alles ist grün. Der Type-Check besteht, der Build besteht, die CI würde bestehen. Und mein Link „Locations“ und mein Warenkorb-Button sind weg -- nicht nur die kollidierenden Zeilen, sondern jede Änderung, die ich an der Datei gemacht habe, einschließlich der Bereiche, die Git korrekt gemergt hatte. `cart-button.tsx` existiert noch und kompiliert auch noch; sie wird nur nirgends mehr verwendet. Über eine ungenutzte Komponente beschwert sich niemand.

Das ist der klassische „Git hat meinen Code gefressen“-Moment, und um fair zu Git zu sein: Git hat nichts gefressen. Die Konfliktlösung war's. GitKraken bietet dieselbe Abkürzung für ganze Dateien bewusst an: Rechtsklick auf eine Datei mit Konflikt und „Take current“ oder „Take incoming“ wählen. Das ist das richtige Werkzeug, wenn eine Seite die andere tatsächlich ersetzt -- eine generierte Datei oder eine Komponente, die eine Kollegin komplett neu geschrieben hat. Für eine Datei, die Sie beide erweitert haben, ist es die Grüner-Build-Falle.

> **Ein grüner Build ist kein korrekter Merge**
> Type-Checker und Bundler prüfen, ob der gemergte Code *gültig* ist, nicht ob er die Arbeit beider Teammitglieder enthält. Sehen Sie sich nach dem Lösen eines Konflikts den Diff des Merge-Ergebnisses gegen **beide** Eltern an oder klicken Sie wenigstens durch das Feature, das die andere Seite gebaut hat. Zwei Minuten Hinschauen sind besser als ein Bug-Report mit dem Titel „Wo ist der Warenkorb hin?“.

Ich denke, jeder, der im Team entwickelt, kennt diese Situation. Sie bemerken einen Bug, beheben ihn und wollen mergen -- und genau zur selben Zeit hat jemand aus Ihrem Team etwas anderes in derselben Datei bemerkt und das behoben. Zwei völlig berechtigte Änderungen, eine Datei, und jetzt sind Sie derjenige, der beide zusammenführen muss, ohne eine davon zu verlieren. Für mich war dieser Moment immer ein Albtraum, egal ob ich auf der Kommandozeile saß oder eine der anderen Oberflächen nutzte, die ich über die Jahre ausprobiert habe. Die Warnung vor dem Merge, beide Seiten und die Ausgabe in einem Fenster und eine KI, die mir die mühsamen Hunks vorbereitet -- das hat mir den Schrecken endlich genommen.

Die Lockfile: Lassen Sie pnpm das machen

Die beiden verbleibenden Konflikte liegen in `package.json` und `pnpm-lock.yaml`, und für sie gilt eine andere Regel: Das Manifest von Hand reparieren, dann den Paketmanager die Lockfile neu aufbauen lassen. Der Konflikt in `package.json` ist trivial, da wir beide eine Abhängigkeit in benachbarten Zeilen hinzugefügt haben:

```
  "dependencies": {
<<<<<<< HEAD
    "lucide-react": "1.50.0",
=======
    "clsx": "2.1.1",
>>>>>>> main
    "next": "16.3.8",
```

Beide Abhängigkeiten bleiben. Im Merge Tool heißt das: beide Zeilen anhaken und alphabetisch sortieren. Die Lockfile dagegen sollten Sie nicht Zeile für Zeile lösen. Die pnpm-Dokumentation sagt es klar: „pnpm can automatically resolve merge conflicts in `pnpm-lock.yaml`. If you have conflicts, just run `pnpm install` and commit the changes.“ So sieht das in der Praxis aus:

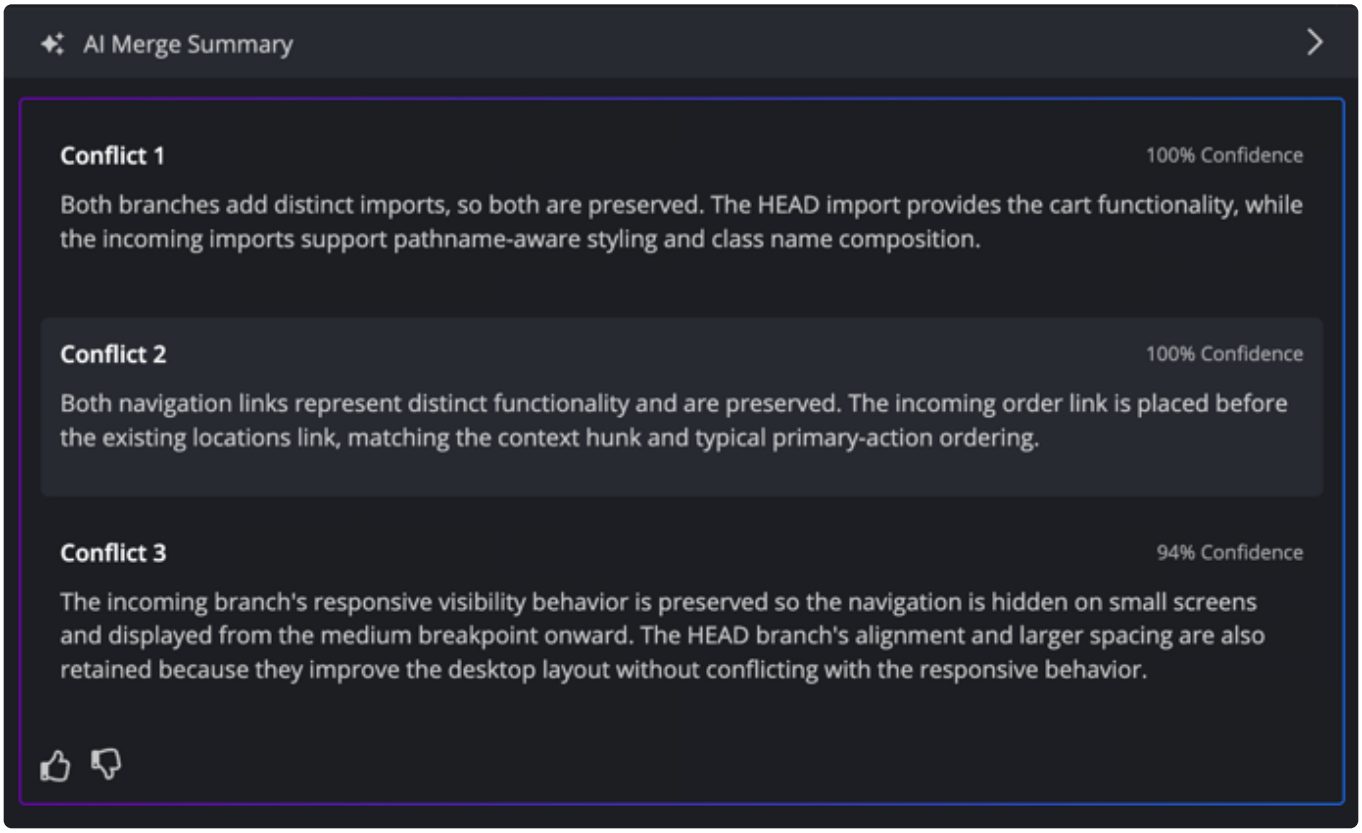
```
$ pnpm install
Merge conflict detected in pnpm-lock.yaml and successfully merged
Packages: +1
+
Progress: resolved 62, reused 32, downloaded 0, added 0, done

dependencies:
+ clsx 2.1.1

Done in 320ms using pnpm v10.33.2
```

Schön. Dieselbe Dokumentation fügt einen ehrlichen Vorbehalt hinzu: „we cannot guarantee that pnpm will choose the correct head“. Ein kurzer Blick auf den Lockfile-Diff in der Commit-Ansicht ist also trotzdem eine gute

Idee. Nutzt Ihr Team npm, beschreibt die ältere npm-Dokumentation dasselbe Muster: Konflikte in `package.json` von Hand lösen, dann `npm install` erneut ausführen.

Die KI den ersten Entwurf machen lassen

Für die Hunks, die eher mühsam als knifflig sind, kann GitKraken Ihnen eine Lösung vorschlagen. Seit Version 11.2 hat das Merge Tool einen Button **Auto-resolve with AI**. Laut der GitKraken-AI-Dokumentation füllt er das Ausgabefeld mit einer vorgeschlagenen Lösung, einer ausführlichen Erklärung und einer Konfidenzstufe für jeden Hunk mit Konflikt. Danach prüfen, bearbeiten, übernehmen oder verwerfen Sie das Ergebnis.



Die AI Merge Summary für unseren Header: eine Erklärung und eine Konfidenzstufe für jeden der drei Konflikte.

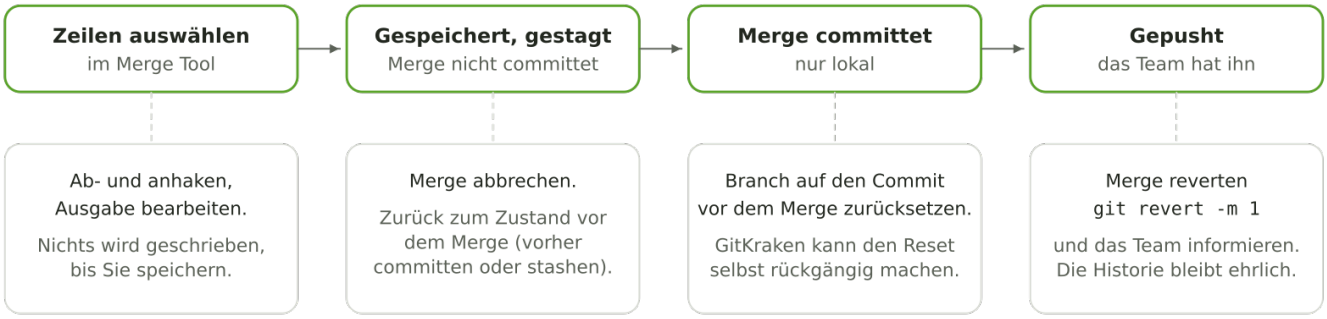
Bei unserem Header hat die KI alle drei Hunks inhaltlich richtig gelöst. Sie hat beide Import-Blöcke und beide Menü-Links behalten, jeweils mit 100 % Konfidenz. Bei der `<nav>`-Zeile hat sie genau das getan, was ich von Hand gemacht habe: Mayas responsive Sichtbarkeit beibehalten -- auf kleinen Bildschirmen ausgeblendet, ab dem mittleren Breakpoint sichtbar -- und meine Ausrichtung und den größeren Abstand übernommen, mit 94 % Konfidenz. Ehrlich gesagt hatte ich erwartet, dass sie genau an diesem Hunk scheitert. Die einzige Stelle, an der ich anders entscheiden würde, ist Geschmackssache: Sie hat „Order online“ vor „Locations“ gesetzt, mit der Begründung, das entspreche der „typical primary-action ordering“. In meiner Lösung oben steht Locations zuerst. Beide Reihenfolgen sind vertretbar, aber es ist eine Produktentscheidung, und die KI hat sie mit 100 % Konfidenz getroffen.

Meine Sicht auf KI-gestützte Konfliktlösung ist einfach: Sie ist ein schneller erster Entwurf, und die Konfidenzstufen sagen Ihnen, wo Sie zuerst lesen sollten, nicht wo Sie aufhören dürfen. Die Dokumentation selbst empfiehlt die Funktion, wenn Sie vorhaben, „to review the result carefully“, und dem stimme ich zu. Die Hunks für Imports und Links sind genau die Art Fleißarbeit, die ich gern abgebe. Trotzdem zeigt die Menü-Reihenfolge, wie ich diese Konfidenzzahl lese: Sie sagt, wie sicher sich das Modell beim Merge ist, nicht, ob Ihr Product Owner „Order online“ zuerst haben möchte. Alles, wo ein Merge Verhalten oder Absicht verändert, statt nur Zeilen hinzuzufügen, verdient einen Menschen, der weiß, was beide Teammitglieder gemeint haben. Beachten

Sie außerdem, dass die Funktion noch als Preview gekennzeichnet ist und ein kostenpflichtiges GitKraken-Abo voraussetzt. Wenn Sie lieber Ihr eigenes Modell verwenden, lässt GitKraken Sie einen eigenen Endpunkt für KI-Aufgaben konfigurieren, Konfliktlösung eingeschlossen.

Rückgängig machen: vier Sicherheitsnetze, je nachdem, wie weit Sie sind

Die Angst, „alles kaputt zu machen“, ist eigentlich eine Angst vor dem Unumkehrbaren, also hilft es, genau zu wissen, was sich an welcher Stelle rückgängig machen lässt. Die ehrliche Antwort lautet: Es gibt nicht den einen magischen Rückgängig-Button für einen Merge, sondern vier Sicherheitsnetze, und welches greift, hängt davon ab, wie weit Sie gekommen sind.



Jede Phase eines Merges hat ihren eigenen Rückweg -- je weiter Sie sind, desto bewusster wird er

Die ersten beiden Netze decken den gesamten Lösungsprozess ab. Solange der Merge nicht committet ist, lässt sich jede Checkbox zurücknehmen, und wenn Sie völlig den Überblick verlieren, können Sie abbrechen. Während ein Merge läuft, graut GitKraken den Undo-Button in der Werkzeugleiste aus; der Rückweg ist in dieser Phase der rote Button **Abort Merge**, den Sie in Schritt 3 im Commit Panel gesehen haben. Er entspricht `git merge --abort`, das laut Dokumentation den laufenden Lösungsprozess abbricht und *versucht*, den Zustand vor dem Merge wiederherzustellen („try to reconstruct the pre-merge state“). Achten Sie auf das Wort *versucht*.

Die Git-Dokumentation empfiehlt, Änderungen „always commit or stash“ -- also immer zu committen oder zu stashen --, bevor Sie `git merge` ausführen, weil nicht committete lokale Änderungen einen Abbruch womöglich nicht überleben. Beginnen Sie jeden Merge also mit einem sauberen Working Tree. Merken Sie sich nur das, und der Abbruch ist ein verlässlicher Notausgang.

Beim dritten Netz kommt GitKrakens Undo-Button ins Spiel. Seine dokumentierte Liste rückgängig machbarer Aktionen enthält das Mergen selbst nicht, wohl aber „Reset branch to a commit“. Haben Sie also einen schlechten Merge committet und noch nicht gepusht, setzen Sie Ihren Branch auf den Commit vor dem Merge zurück, und sollte sich der Reset als falsche Entscheidung herausstellen, drücken Sie Z (Strg+Z unter Windows und Linux), und er ist rückgängig gemacht. Von dort aus lässt sich entspannt experimentieren. Ist der Merge einmal gepusht, ist die respektvolle Option `git revert -m 1` auf den Merge-Commit plus eine Nachricht an Ihr Team. Gemeinsame Historie umzuschreiben ist der Weg, auf dem aus einem Merge-Problem ein Team-Problem wird.

Derselbe Konflikt in VS Code und auf der Kommandozeile

GitKraken ist nicht der einzige Weg, diesen Konflikt zu lösen, und die Alternativen verdienen ihre Anerkennung. VS Code zeigt CodeLens-Aktionen direkt über jedem Konflikt -- „Accept Current Change“, „Accept Incoming Change“, „Accept Both Changes“ und „Compare Changes“ -- und bietet einen vollwertigen Drei-Wege-Merge-Editor (zuerst als Opt-in in Version 1.69 verfügbar), mit Incoming und Current nebeneinander

und dem Ergebnis darunter. Eine Sache macht er besonders gut: Er zeigt den gemeinsamen Vorfahren, den die Dokumentation als „the common version against which Git compares the two sides“ beschreibt, erreichbar über das Menü More Actions des Editors. Lebt Ihr ganzes Team in VS Code, ist das ein leistungsfähiges Setup, und sein Button „Accept Both Changes“ erzeugt unser doppeltes `<nav>` ganz genauso bereitwillig. Das Werkzeug zeigt Ihnen das Problem; entscheiden müssen Sie trotzdem selbst.

Für die Kommandozeile habe ich eine einzige Empfehlung: Ändern Sie Gits Konfliktstil. Mit `zdiff3` (eingeführt in Git 2.35) enthalten die Marker auch die Basisversion, und plötzlich erklärt sich der `<nav>`-Hunk von selbst:

```
$ git config --global merge.conflictStyle zdiff3
$ git merge main
...
<<<<<<< HEAD
      <nav className="flex items-center gap-6">
||||||| 5ba0670
      <nav className="flex gap-4">
=======
      <nav className="hidden gap-4 md:flex">
>>>>>>> main
```

Mit sichtbarer Basis ist offensichtlich, dass beide Seiten von `flex gap-4` ausgegangen sind und Unterschiedliches geändert haben -- genau der Hinweis, dass Sie kombinieren statt wählen müssen. Wenn Sie außerhalb des Editors ein kostenloses Open-Source-Werkzeug mit grafischer Oberfläche möchten, bietet Meld (GPLv2) „three-way merge assistance with conflict handling and base version display“, und KDiff3 (GPL-2.0) vergleicht und mergt bis zu drei Dateien. KDiff3 gehört auch zu den externen Merge-Werkzeugen, die Sie in GitKraken unter *Preferences* > *General* einbinden können, neben Beyond Compare, P4Merge und anderen, und das Merge Tool hat direkt neben Save einen Button „Open in external merge tool“.

Was GitKraken für mich auszeichnet, ist die Kombination: die Warnung vor dem Merge, die Liste der Konflikte Datei für Datei, die editierbare Ausgabe, KI für die Fleißarbeit und der Graph, um zu sehen, was Sie gerade getan haben -- in einem Fenster, ohne die Teile selbst zusammenstecken zu müssen. Wenn Ihnen der Graph-Aspekt gefällt: Darüber habe ich in Visualizing Your Git History geschrieben, und darüber, wohin sich GitKraken entwickelt, in GitKraken ist jetzt die Code-Flow-Company.

Gewohnheiten, die Konflikte klein halten

Die beste Konfliktlösung ist ein kleinerer Konflikt, deshalb hier die Gewohnheiten, die ich in Next.js-Teams funktionieren sehe. Mergen Sie `main` früh und oft in Ihren Feature-Branch; der Konflikt in diesem Beitrag hat Minuten gedauert, weil er einen Tag alt war, nicht drei Wochen. Halten Sie gemeinsam genutzte Komponenten wie Header und Layouts klein, damit zwei Tickets seltener dieselben Zeilen anfassen. Behandeln Sie Lockfiles als generierte Artefakte und lassen Sie den Paketmanager sie auflösen. Und wenn Sie denselben Konflikt immer wieder lösen, etwa bei einem langlebigen Branch, aktivieren Sie `git rerere`, das Ihre manuellen Lösungen aufzeichnet und beim nächsten Auftreten desselben Konflikts wieder anwendet.

Das Fazit

Unser Konflikt hatte drei Dateien, drei Hunks in der Komponente und drei verschiedene richtige Antworten: beide nehmen, beide in der richtigen Reihenfolge nehmen und von Hand bearbeiten. Die Lockfile hat sich selbst gelöst, sobald `package.json` geklärt war. Die eigentliche Gefahr waren nie die Marker, sondern die Abkürzung über die ganze Datei, die jede automatische Prüfung bestanden und stillschweigend ein Feature gelöscht hat, und der Merge, der ohne zu fragen eine Server Component in eine Client Component verwandelt hat.

> Merge-Konflikte werden langweilig, sobald Sie sehen, was kollidiert, pro Hunk statt pro Datei entscheiden und von jedem Schritt aus den Rückweg kennen.

Genau das gibt mir GitKraken: die Warnung vor dem Merge, beide Seiten und die Ausgabe nebeneinander, einen KI-Entwurf für die mühsamen Hunks und einen klaren Rückweg. Wenn Sie es beim nächsten Konflikt Ihres Teams ausprobieren möchten, bekommen Sie mit meinem Empfehlungslink **50 % Rabatt auf GitKraken Pro**: GitKraken Pro ausprobieren. Der Transparenz halber: Es handelt sich um einen Empfehlungslink. Die Community Edition ist für lokale Repositories und öffentliche Remotes kostenlos, Sie können sie also zuerst an einem Open-Source-Projekt ausprobieren.

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 —