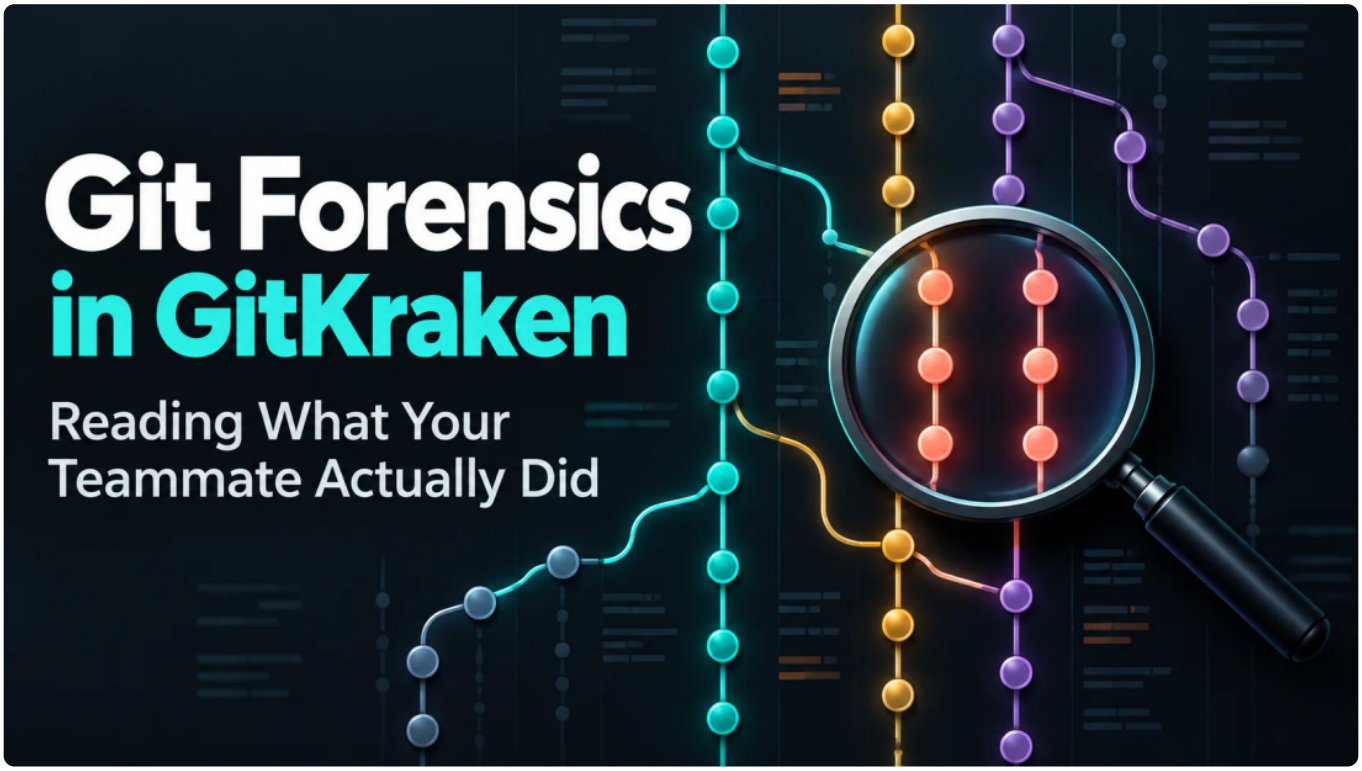
NEWS · OCTOBER 13, 2026

Git-Forensik in GitKraken: Lesen, was Ihr Teamkollege wirklich getan hat

Ein Teamkollege, der neu bei GitHub ist, hinterlässt vier klassische Git-Fehler auf dem Remote -- einen Commit direkt auf main, einen Branch vom falschen Branch, eine Leiter aus Merges und jeden Commit doppelt -- und wir lesen jeden davon im Commit-Graphen von GitKraken ab, bevor wir ihn beheben, einschließlich der Falle, die das Chaos nach dem Aufräumen zurückbringt.

By Dr. Holger Flick



Die Nachricht kommt meistens am späten Nachmittag: „Kannst du mal draufschauen? Ich glaube, Git ist kaputt.“ Sie stammt von einem Teamkollegen, der neu bei GitHub ist, und im Anhang steckt ein Pull Request, der drei Commits enthalten sollte und neun enthält. Zwei davon heißen „fix typo“. Einer heißt „stuff“. Irgendwo dazwischen stecken Merge-Commits, an die sich niemand erinnert.

Natürlich ist Git nicht kaputt. Git hat genau das getan, was man ihm gesagt hat, mehrfach, in einer Reihenfolge, die in dem Moment für jemanden Sinn ergab. Meiner Meinung nach ist das Schlimmste, was Sie jetzt tun können, sofort mit dem Reparieren anzufangen. Sie müssen zuerst wissen, was tatsächlich passiert ist, denn jede Reparatur in Git -- ein Reset, ein Rebase, ein Force Push -- setzt voraus, dass Sie wissen, welche Commits zählen und welche nur Kopien sind. Reparieren Sie das Falsche, haben Sie zwei Baustellen statt einer.

Für diesen Beitrag habe ich so ein Chaos von Grund auf nachgebaut. Dann lesen wir es wie einen Tatort im Commit-Graphen von GitKraken Desktop: vier klassische Fehler, wie jeder aussieht, wie er entstanden ist und wie Sie Branches und Pull Requests wieder auf Kurs bringen. Am Ende wartet eine Falle, die in meinem Test

mein eigenes Aufräumen rückgängig gemacht hat -- und sie ist der Grund, warum ich den Branch eines Kollegen nicht repariere, ohne vorher mit ihm zu sprechen.

Der Tatort: ein Repository zum Nachbauen

Um eine Historie zu lesen, brauchen wir eine realistische, und die aufgeräumten Demo-Repositories der meisten Tutorials taugen dafür nicht. Also habe ich ein kleines Shell-Skript geschrieben, das drei Entwickler an der Website einer fiktiven Restaurantkette spielt (Los Pollos Hermanos, was sonst). Jede Person arbeitet in ihrem eigenen Clone und pusht auf ein gemeinsames Remote, und jeder Fehler wird echt mit ganz normalen `git`-Befehlen gemacht -- nichts wird nachträglich bearbeitet oder gefälscht.

Die Besetzung ist überschaubar. **Holger** ist der Teamleiter und merged die Pull Requests. **Maya** ist eine

erfahrene Kollegin. **Jesse** ist neu bei Git und GitHub, motiviert und schnell. Über vier Tage rund um das

Monatsende im September liefert Maya zwei Pull Requests aus, und Jesse arbeitet an einer Speisekarte mit Preisen und an einem Catering-Formular. Das Repository beginnt mit einer Contribution-Anleitung, die in drei kurzen Zeilen sagt: von `main` abzweigen, einen Pull Request öffnen und niemals direkt auf `main` committen. Sie ahnen vermutlich, wie das ausgeht.

Das Skript hinterlässt drei Clones neben einem Bare-Repository, das die Rolle von GitHub spielt. Holgers Clone -- die Sicht des Teamleiters, mit allen Branches der Kollegen gefetcht -- habe ich in GitKraken Desktop geöffnet. Beginnen wir mit den Ermittlungen.

Was das Terminal zeigt

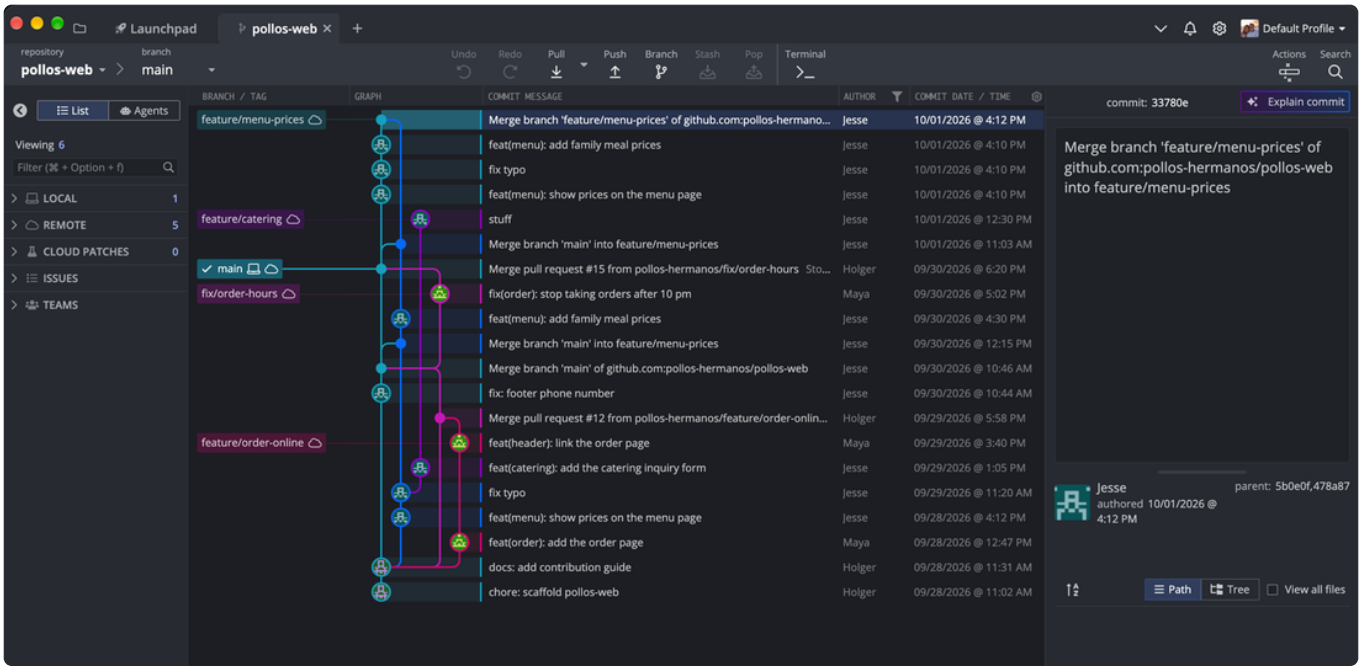
Schauen wir uns die Beweise zunächst so an, wie Git sie ausgibt, denn es ist alles da. Das ist die Ausgabe des klassischen Alias, den viele von uns in ihrer Shell-Konfiguration haben, mit dekorierten Branch-Namen:

```
$ git log --graph --oneline --all --decorate --date-order
*   33780e7 (origin/feature/menu-prices) Merge branch 'feature/menu-prices' of github.com:pol-
los-hermanos/pollos-web into feature/menu-prices
|\
* | 5b0e0f1 feat(menu): add family meal prices
* | 6d43567 fix typo
* | 2d16d8e feat(menu): show prices on the menu page
| | * b2498b3 (origin/feature/catering) stuff
| * |   478a876 Merge branch 'main' into feature/menu-prices
| |\ \
| |/ /
|/| |
* | |   c9c5402 (HEAD -> main, origin/main, origin/HEAD) Merge pull request #15 from
pollos-hermanos/fix/order-hours
|\ \ \
| * | | 34ec852 (origin/fix/order-hours) fix(order): stop taking orders after 10 pm
|/ / /
| * | 0ab974b feat(menu): add family meal prices
| * |   4159866 Merge branch 'main' into feature/menu-prices
| |\ \
| |/ /
|/| |
* | |   df1a71a Merge branch 'main' of github.com:pollos-hermanos/pollos-web
|\ \ \
* | | | dd72315 fix: footer phone number
| * | | c78ffdd Merge pull request #12 from pollos-hermanos/feature/order-online
|/| | |
| * | | 709ad73 (origin/feature/order-online) feat(header): link the order page
| | | * 1e0ea1a feat(catering): add the catering inquiry form
| | |/
| | * 5235256 fix typo
| | * 67c4443 feat(menu): show prices on the menu page
| |/
|/|
| * 351d218 feat(order): add the order page
|/
* f4ea835 docs: add contribution guide
* 8586694 chore: scaffold pollos-web
```

Ehrlich, Hand aufs Herz: Alle vier Fehler stecken in dieser Ausgabe. Die Information ist vollständig. Was fehlt, ist das Lesen. Die Spuren verschieben sich seitwärts, sobald ein Merge hereinkommt, dieselbe Commit-Nachricht taucht zweimal mit unterschiedlichen Hashes auf, und Sie müssen über dreißig Zeilen verfolgen, welcher `|` zu welchem Branch gehört. Ich kann das lesen. Ich kann es nicht schnell lesen, und ich kann es keinem Junior-Entwickler hinlegen und sagen: „Schau, genau das ist passiert.“

Dieselbe Historie in GitKraken

Öffnen wir nun dasselbe Repository in GitKraken Desktop. Jeder Branch bekommt seine eigene Spur und Farbe, Merges werden als Kurven in die Spur gezeichnet, in der sie landen, und jeder Commit trägt den Avatar seines Autors -- „wer hat was gemacht“ ist also sichtbar, bevor Sie eine einzige Nachricht lesen.



Das Demo-Repository direkt nach dem Fetch: <code>main</code>, zwei bereits gemergte Branches von Maya und Jesses zwei Feature-Branches.

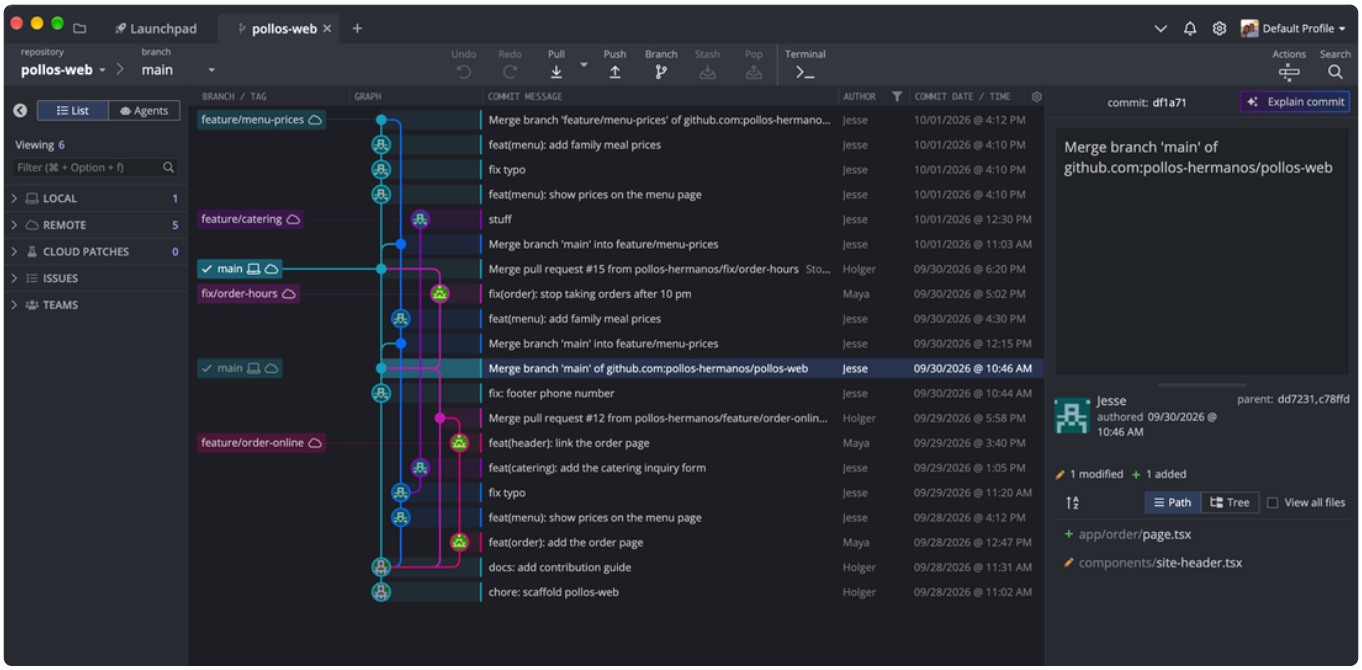
Zwei Funktionen leisten hier den Großteil der forensischen Arbeit. Erstens lassen sich die Spalten des Graphen konfigurieren: Standardmäßig sehen Sie Branch-/Tag-Labels, den Graphen und die Commit-Nachricht, und ein Rechtsklick auf den Spaltenkopf fügt Autor, Datum/Uhrzeit und SHA hinzu (Details in GitKrakens Dokumentation zur Oberfläche). Zweitens hat die Spalte **Author** ein Filtersymbol. Filtern Sie auf Jesse, und die Frage „Was hat er diese Woche eigentlich gemacht?“ ist ohne einen einzigen Befehl beantwortet. Die Suche mit Cmd/Strg+F funktioniert ebenfalls; sie durchsucht Commit-Nachrichten, SHAs und Autoren und hebt die Treffer im Graphen hervor, statt den Rest auszublenden.

Ein weiteres Detail lohnt sich gleich zu Beginn, weil es am Ende dieses Beitrags wichtig wird: die Symbole an den Branch-Labels. Das Label `main` trägt einen Laptop und eine Wolke -- der lokale Branch und sein Gegenstück auf dem Remote zeigen also auf denselben Commit. Die Branches von Jesse und Maya tragen nur die Wolke; sie existieren auf dem Remote, und in diesem Clone hat sie niemand ausgecheckt.

Mit diesem Überblick können wir uns die vier Spuren nacheinander vornehmen.

Spur 1: Ein Merge auf main, den kein Pull Request erzeugt hat

Auf `main` fällt zuerst ein Merge-Commit auf, dessen Nachricht nicht ins Muster der anderen passt. Jeder Pull Request hinterlässt einen Commit `Merge pull request #NN from ...`. Dieser hier lautet `Merge branch 'main' of github.com:pollos-hermanos/pollos-web`, und diese Nachricht ist der Fingerabdruck von `git pull`: Genau das schreibt Git, wenn es die Remote-Version eines Branches in Ihre lokale Version desselben Branches merged.



Jesses Commit liegt direkt auf <code>main</code>, mit PR #12 durch einen Merge verbunden, den nur <code>git pull</code> schreibt.

Liest man den Graphen rückwärts, erzählt er die ganze Geschichte. Jesse hat die Telefonnummer im Footer korrigiert und auf seinem lokalen `main` committet -- eine einzeilige Änderung, wozu also ein Branch. In der Zwischenzeit war PR #12 auf dem Server in `main` gelandet. Sein Push wurde abgelehnt, weil sein `main` und das `main` auf dem Server auseinandergelaufen waren. Also tat er, was die Fehlermeldung vorschlägt, und machte einen Pull. Der merged das `main` vom Server in sein eigenes, und danach ging der Push durch.

Wählen Sie den Merge aus, und das Commit-Panel bestätigt jeden Schritt dieser Geschichte. Die Parents sind `dd7231` und `c78ffd`: zuerst Jesses Footer-Korrektur, dann PR #12. Die aufgeführten Änderungen, ein neues `app/order/page.tsx` und ein geändertes `components/site-header.tsx`, sind genau Mayas Bestellseite -- die Arbeit, die Jesses Laptop noch nicht hatte. Das blasse Label `main` neben der ausgewählten Zeile ist GitKrakens Ghost-Branch-Label; es zeigt den Branch, der den betrachteten Commit enthält.

Sie fragen sich vielleicht, wie er überhaupt zu einem Merge kam, denn modernes Git weigert sich, auseinandergelaufene Branches zu pullen, bis Sie ihm sagen, wie. Das stimmt -- ich habe es mit Git 2.54 ausprobiert, und der Pull bricht mit einem Hinweis ab. Dieser Hinweis nennt allerdings drei Optionen, und die erste lautet so:

```
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
```

Raten Sie, welche Zeile jemand kopiert, der neu bei Git ist. In der Demo ist Jesses Clone genau mit dieser ersten Zeile konfiguriert.

Ist der Merge eine Katastrophe? Nein. Nichts ging verloren, und die Korrektur im Footer ist völlig in Ordnung. Die Nebenwirkung ist allerdings subtil. Ein Merge speichert den Branch, auf dem Sie gerade waren, als ersten Parent. Die First-Parent-Linie von `main` läuft deshalb jetzt über Jesses Laptop statt über PR #12:

```
$ git log --first-parent --oneline main
c9c5402 Merge pull request #15 from pollos-hermanos/fix/order-hours
df1a71a Merge branch 'main' of github.com:pollos-hermanos/pollos-web
dd72315 fix: footer phone number
f4ea835 docs: add contribution guide
8586694 chore: scaffold pollos-web
```

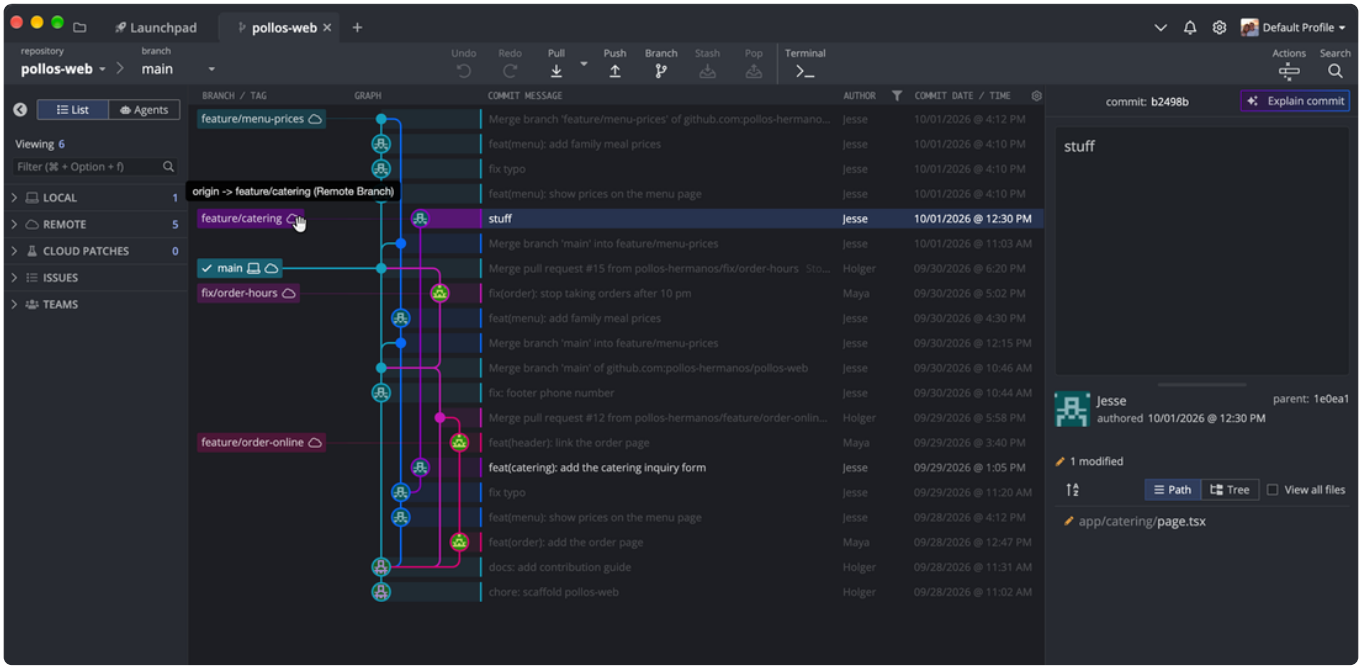
PR #12 ist aus dieser Liste verschwunden. Die Arbeit ist noch da; sie ist nur zum zweiten Parent gewandert. Wenn Ihr Team `--first-parent` als „was auf `main` gelandet ist, ein Pull Request pro Zeile“ liest, wird die Geschichte genau hier verwirrend.

Die Lösung: liegen lassen und main schützen

Die Reparatur für Spur 1 besteht meiner Meinung nach darin, an der Historie nichts zu tun. `main` ist geteilt; wer es umschreibt, zwingt jeden Clone im Team zur Wiederherstellung, und das ist ein weit größeres Chaos als ein einzelner seltsamer Merge-Commit. Die eigentliche Lösung liegt davor. Auf GitHub sorgt eine Branch-Protection-Regel oder ein Ruleset, das für `main` einen Pull Request verlangt, dafür, dass der Push endgültig scheitert -- mit einer Meldung, die Jesse auch sagt, warum. Und ein kurzes Gespräch über `git config --global pull.ff only` bringt Git dazu, beim nächsten auseinandergelaufenen Branch anzuhalten und zu fragen, statt still zu mergen.

Spur 2: Ein Branch, der aus dem falschen Branch wächst

Die zweite Spur ist der Grund, warum PR #14, das Catering-Formular, Commits über Speisekartenpreise enthält. Im Graphen beginnt die Spur von `feature/catering` nicht an `main`. Sie beginnt am Commit „fix typo“ von `feature/menu-prices`.



Beim Hovern über <code>feature/catering</code> leuchten seine eigenen zwei Commits auf -- und seine lila Spur mündet bei „fix typo“ in Jesses blaue Speisekarten-Spur, nicht in <code>main</code>.

Sie müssen die Spuren dafür nicht mit dem Auge verfolgen. Fahren Sie mit der Maus über das Label `feature/catering`, und GitKraken hebt die Commits des Branches hervor und blendet den Rest ab. Zwei Commits leuchten auf, „add the catering inquiry form“ und „stuff“. Aufschlussreich ist, wohin die lila Spur danach

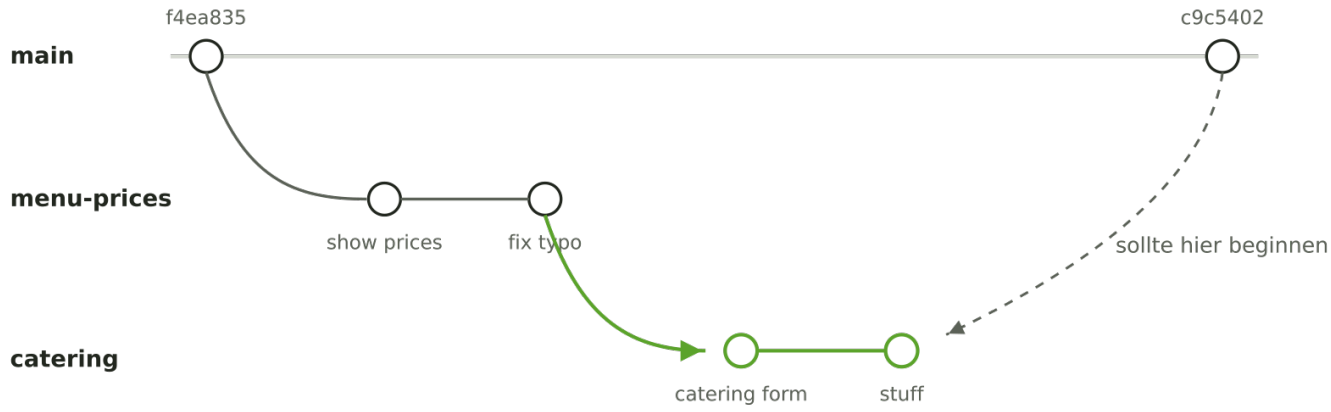
führt: Sie mündet nicht zurück in `main`, sondern bei „fix typo“ in Jesses blaue Speisekarten-Spur. Alles unterhalb dieser Abzweigung teilt sich der Branch mit dem Speisekarten-Branch -- und ein Pull Request gegen `main` enthält all das, wie wir gleich sehen. Stören die anderen Branches, klicken Sie mit der rechten Maustaste auf `main` und `feature/catering` und wählen **Solo**; der Solo-Modus blendet jeden Branch aus, den Sie nicht ausdrücklich auf Solo gesetzt haben.

Meiner Erfahrung nach ist das einer der häufigsten Fehler von Entwicklern, die neu bei Git sind, und er ist absolut verständlich. `git checkout -b` (oder der Button „New branch“ in jeder GUI) zweigt von dort ab, wo Sie gerade stehen, nicht von `main`. Jesse war noch auf seinem Speisekarten-Branch, als er mit dem Catering-Formular anfing. Der Pull Request gegen `main` listet dann pflichtbewusst alles auf, was auf dem Catering-Branch ist, aber nicht auf `main`:

```
$ git log --oneline main..origin/feature/catering
b2498b3 stuff
1e0ea1a feat(catering): add the catering inquiry form
5235256 fix typo
67c4443 feat(menu): show prices on the menu page
```

Beachten Sie den Unterschied zur Hervorhebung in GitKraken: Der Graph zeigt die zwei Commits, die zur Catering-Spur gehören, während der Pull Request aus allem berechnet wird, was noch nicht auf `main` ist. Zwei dieser vier Commits gehören zu einem anderen Pull Request. Wird PR #14 zuerst gemergt, landet die halbe Speisekarten-Funktion ohne Review auf `main`.

Das folgende Diagramm zeigt den Unterschied zwischen dem Punkt, an dem der Branch beginnt, und dem, an dem er beginnen sollte.

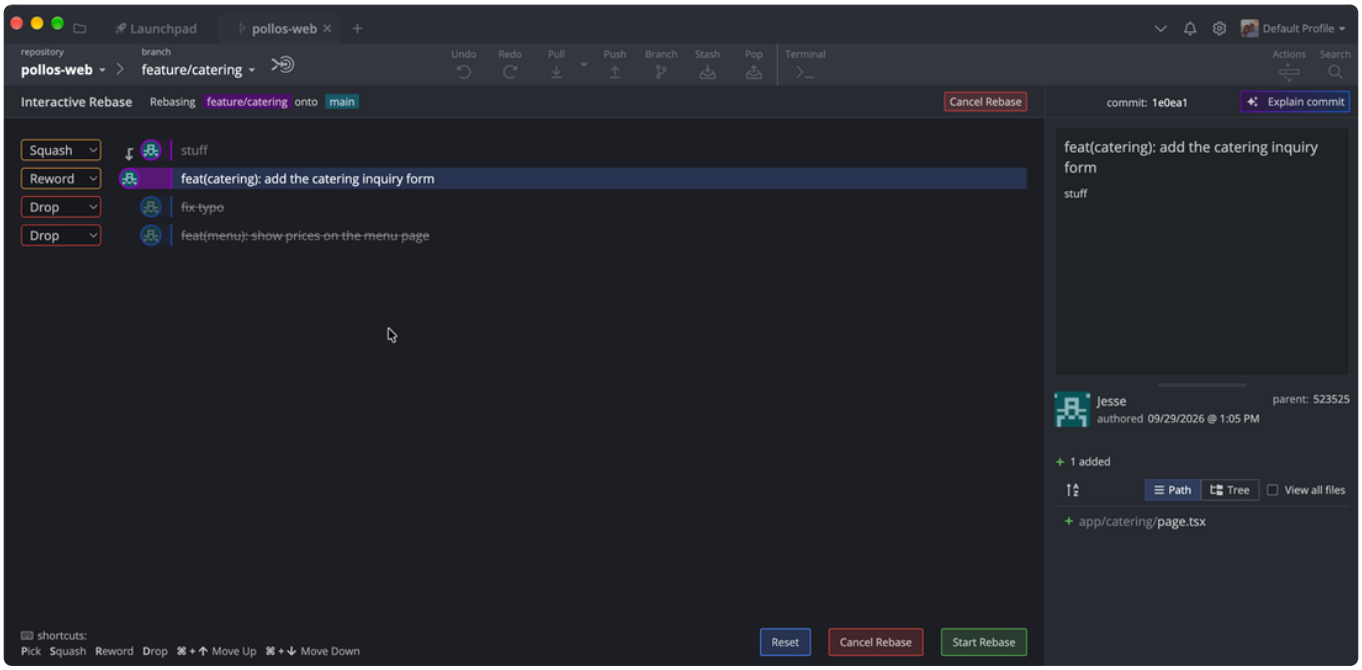


feature/catering beginnt auf Jesses Speisekarten-Branch statt auf main

Die durchgezogene farbige Linie ist das, was Jesse gebaut hat; die gestrichelte zeigt, wo der Catering-Branch hingehört. Alles zwischen den beiden Startpunkten landet im falschen Pull Request.

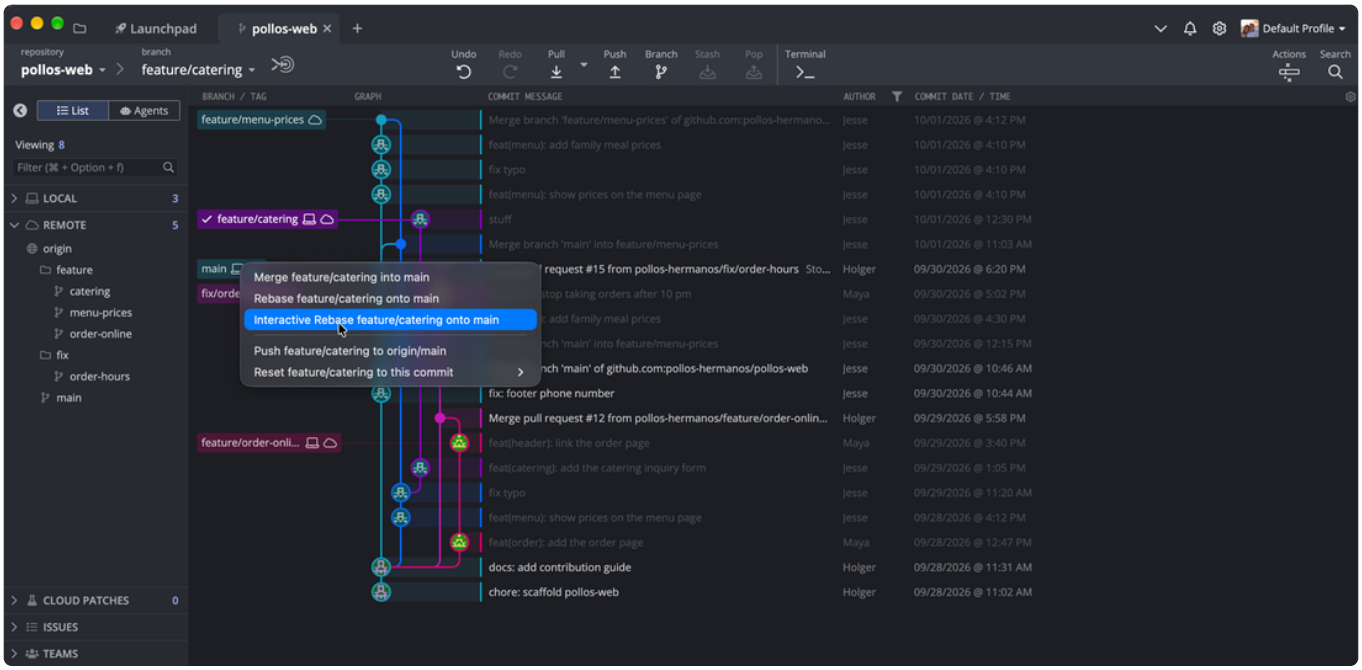
Die Lösung: den Branch verschieben oder die Abhängigkeit ehrlich machen

Es gibt zwei gute Antworten, und welche richtig ist, hängt von einer Frage ab, die nur Jesse beantworten kann: Braucht das Catering-Formular die Speisekartenpreise? Wenn nicht, verschieben wir die beiden Catering-Commits auf `main` und lassen die Speisekarten-Commits zurück. In GitKraken listet ein interaktiver Rebase von `feature/catering` auf `main` alle vier Commits, und wir verwerfen einfach die beiden Speisekarten-Commits. Wenn wir schon dabei sind, falten wir „stuff“ in den Commit davor, denn „stuff“ ist keine Commit-Nachricht, sondern ein Hilferuf. Das Traurige daran: Jesse hätte sich nicht einmal eine bessere ausdenken müssen. GitKraken Desktop kann mit seiner KI eine Commit-Nachricht aus den gestagten Änderungen vorschlagen, und ich habe genau darüber einen ganzen Liebesbrief an die schlechtesten Commit-Nachrichten der Geschichte geschrieben. „Fixed stuff“ hat es nicht ohne Grund in den Titel geschafft.



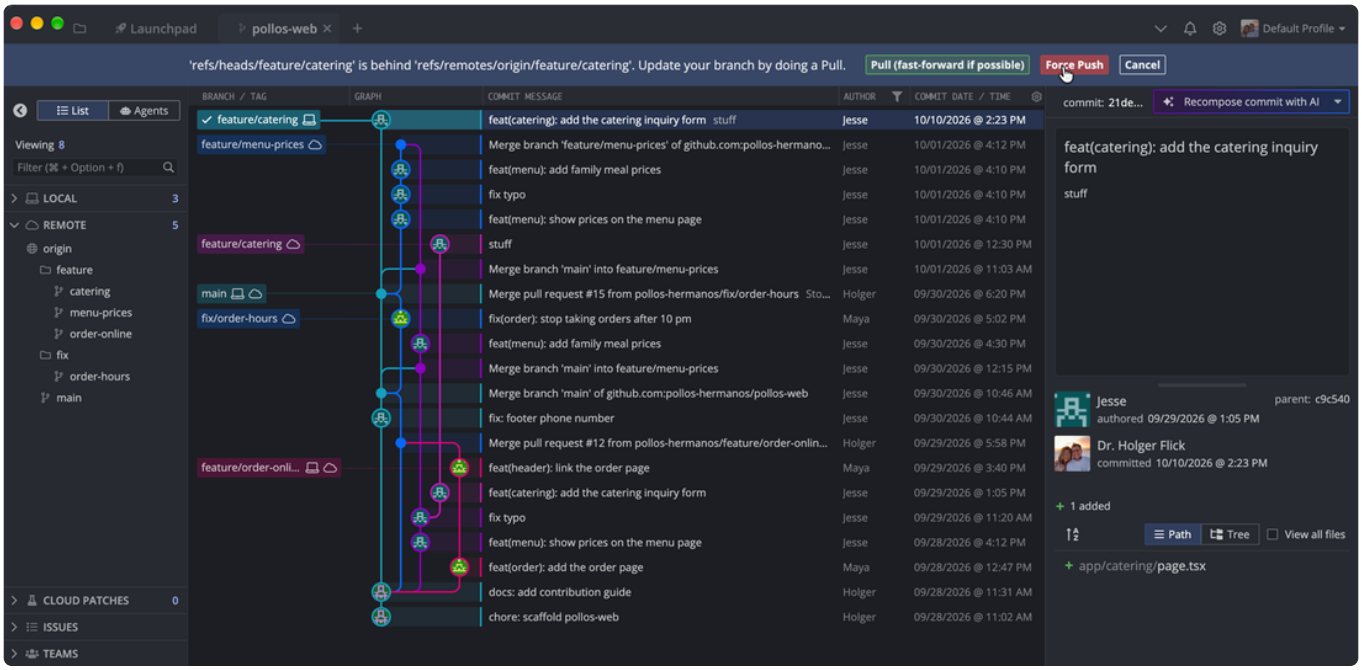
Interaktiver Rebase von <code>feature/catering</code> auf <code>main</code>: zwei Speisekarten-Commits verworfen, „stuff“ in das Formular gesquasht, das GitKraken von selbst auf Reword umgestellt hat.

Dafür doppelklicken Sie auf `origin/feature/catering`, um einen lokalen Branch anzulegen, und ziehen dann das Label `feature/catering` im Graphen auf `main`. GitKraken öffnet ein kleines Menü mit allem, was Sie mit den beiden Branches tun können.



<code>feature/catering</code> auf <code>main</code> fallen lassen: Merge, Rebase, interaktiver Rebase, Push oder Reset.

Wir wählen **Interactive Rebase feature/catering onto main**. Beachten Sie allerdings den Eintrag darunter: **Push feature/catering to origin/main** ist nur einen Mausrutscher entfernt und würde den Catering-Branch direkt auf das `main` des Servers legen -- ein Grund mehr für den Branch-Schutz aus Spur 1. GitKraken listet die Commits mit den Aktionen Pick, Reword, Squash und Drop, und nichts passiert, bevor Sie auf **Start Rebase** klicken. Verworfene Commits erscheinen durchgestrichen. Setzen Sie „stuff“ auf **Squash**, zeigt ein kleiner Pfeil, dass er in den Commit darunter gefaltet wird, und GitKraken stellt diesen Commit von selbst von Pick auf **Reword** um -- aus zwei Commits wird einer, und der braucht eine Nachricht. Die vorgeschlagene Nachricht ist die des Formulars mit „stuff“ als zweiter Zeile, und Sie bearbeiten sie direkt im Panel rechts, bevor Sie den Rebase starten. Löschen Sie diese Zeile; niemand wird sie vermissen. Beachten Sie, dass GitKrakens interaktiver Rebase keine Merge-Commits auf dem Branch akzeptiert, den Sie rebasen. Für den Catering-Branch funktioniert das also, für den Speisekarten-Branch mit seinen Merges nicht -- was in Ordnung ist, denn der braucht ohnehin eine andere Reparatur. Dann pushen wir, und hier wird es interessant.



Nach dem Rebase nennt GitKraken den Branch „behind“ und schlägt einen Pull vor. Direkt nach einem Rebase ist Force Push der richtige Button.

GitKraken verweigert den normalen Push, meldet, der lokale Branch liege hinter dem Remote zurück, und schlägt einen Pull vor. Genau genommen sind die beiden auseinandergelaufen: Der rebaste Commit existiert nur lokal, die vier alten nur auf dem Server. Lesen Sie diesen Banner zweimal, denn er gibt genau den Rat, dem Jesse in Spur 4 gefolgt ist. Nach einem bewusst ausgeführten Rebase merged **Pull** die alten Commits direkt wieder hinein, und **Force Push** ist der Button, den Sie wollen -- mehr zum sicheren Force Push in Spur 4. Beachten Sie auch das Commit-Panel: Jesse ist weiterhin der Autor, datiert auf den 29. September, aber Committer bin ich, heute. So sieht „ein Rebase schreibt neue Commits“ in der Praxis aus.

Auf der Kommandozeile ist derselbe Schritt ein `git rebase --onto`, der die Commits nach `5235256` nimmt und sie oben auf `main` neu anwendet:

```
$ git switch feature/catering
$ git rebase --onto origin/main 5235256
Successfully rebased and updated refs/heads/feature/catering.
$ git log --oneline origin/main..HEAD
5407852 stuff
009b1d8 feat(catering): add the catering inquiry form
```

Beide Hashes sind brandneu. Ein Rebase schreibt neue Commits und versieht sie mit der aktuellen Uhrzeit -- und genau darum geht es, wie sich zeigen wird, in Spur 4. Ein anschließendes `git rebase -i origin/main` macht aus „stuff“ dann ein `fixup` des Formular-Commits.

Hängt das Catering-Formular dagegen tatsächlich von der Speisekarte ab, besteht die ehrliche Lösung nicht darin, etwas zu verschieben, sondern es auszusprechen: Ändern Sie die Basis von PR #14 von `main` auf `feature/menu-prices`, wie es GitHub bei jedem offenen Pull Request erlaubt. Jetzt zeigt der Pull Request nur noch die Catering-Commits, und es ist offensichtlich, dass er auf die Speisekarte warten muss.

Spur 3: Die Leiter

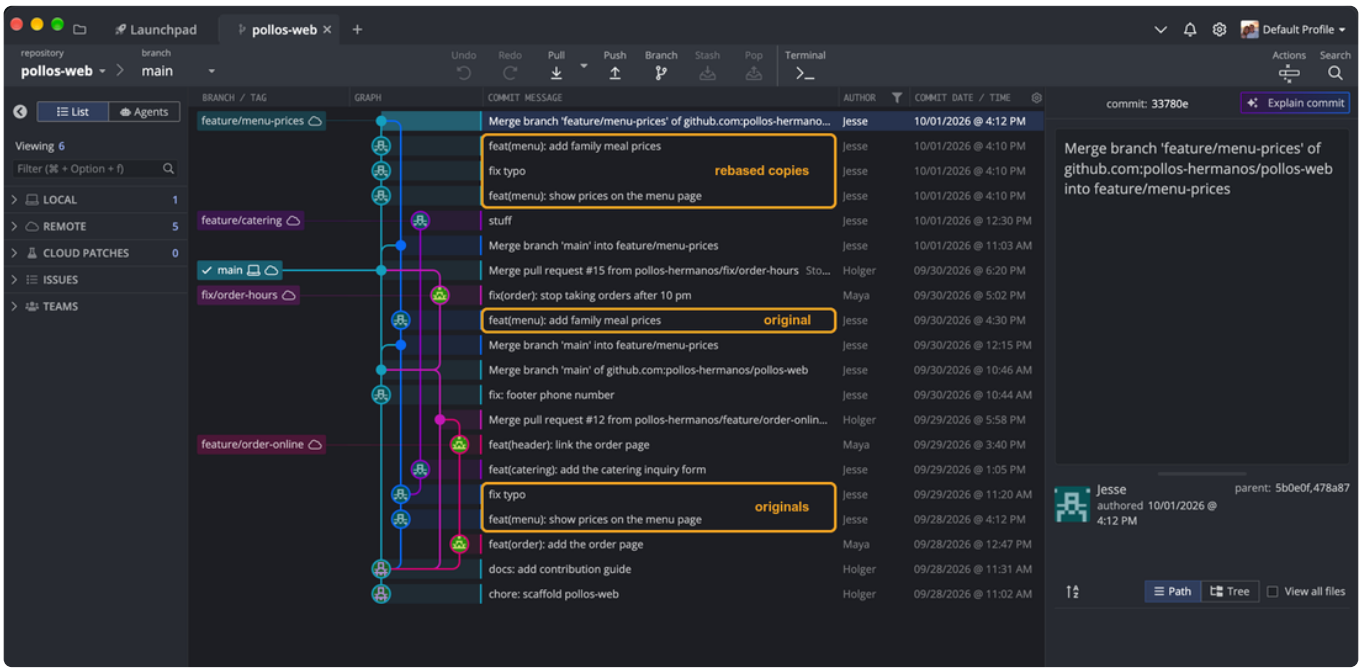
Die dritte Spur sieht im Graphen dramatisch aus und ist die harmloseste der vier. `feature/menu-prices` hat zwei Commits namens `Merge branch 'main' into feature/menu-prices`, und jeder zieht eine Sprosse von `main` hinüber zu Jesses Spur.

In der Demo stehen diese beiden Merges für den Button **Update branch**, den GitHub bei einem Pull Request anzeigt, der hinter seiner Basis zurückliegt. Ein Klick darauf führt „a traditional merge“ aus, also einen klassischen Merge des Basis-Branches in den Branch des Pull Requests; das Dropdown daneben bietet außerdem **Update with rebase** an. Jesse hat ihn zweimal gedrückt, und genau dafür ist der Button da. Ein Fehler ist das nicht.

Ich gebe zu, dass ich einen Feature-Branch ohne Sprossen bevorzuge; ein Rebase auf `main` oder ein Squash-Merge am Ende hält die Historie von `main` gut lesbar. Viele erfahrene Teams sehen das anders, und sie haben einen Punkt: Ein Merge hält fest, was tatsächlich passiert ist, und schreibt nie einen Commit um, den jemand anderes vielleicht schon hat -- ein Rebase schon. Wie wir gleich sehen, ist es genau dieser Unterschied, der aus dieser Leiter Spur 4 gemacht hat. Ist Ihr Team neu bei Git, sehe ich deshalb lieber ein paar Sprossen als einen umgeschriebenen Branch, den niemand versteht. Ein Squash-Merge des Pull Requests am Ende (eine der drei Merge-Methoden von GitHub) räumt sie auf, ohne dass jemand rebasen muss.

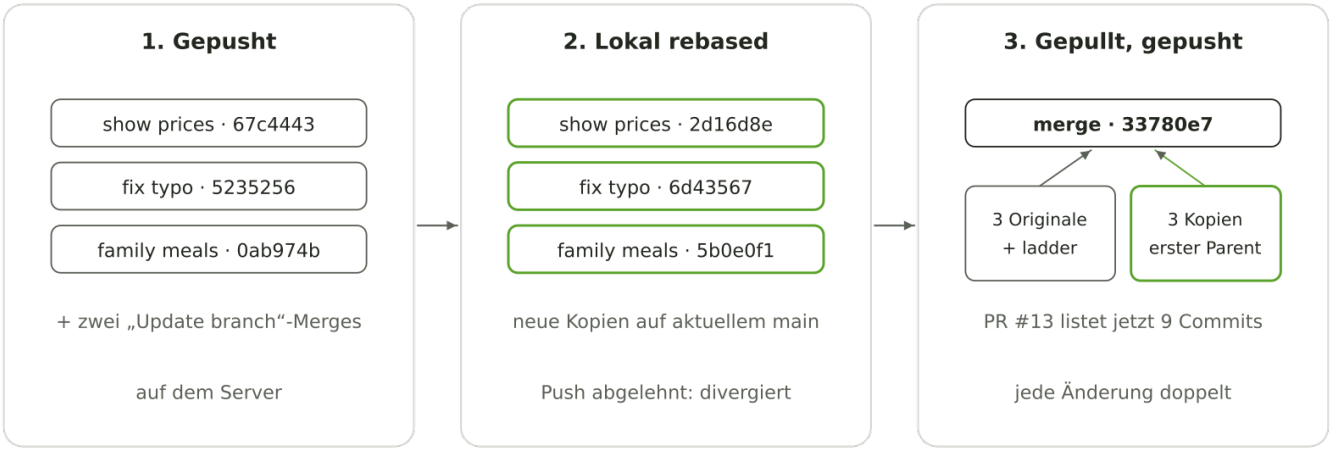
Spur 4: Jeder Commit, zweimal

Die vierte Spur ist diejenige, die „Ich glaube, Git ist kaputt“ ausgelöst hat. Oben im Graphen teilt sich `feature/menu-prices` in zwei parallele Spuren, die dieselben drei Commit-Nachrichten tragen -- „show prices“, „fix typo“, „family meal prices“ -- mit unterschiedlichen Hashes, oben durch einen Merge verbunden.



Dieselben drei Commits zweimal (Rahmen nachträglich eingezeichnet): oben die rebasten Kopien, weiter unten die Originale zwischen den Leiter-Merges. Der ausgewählte Merge nennt die Kopie <code>5b0e0f</code> als ersten Parent und ändert keine Dateien.

Folgendes ist passiert. Nach dem zweiten **Update branch** hat jemand Jesse gesagt, er solle rebasen statt mergen. Im Prinzip ein guter Rat. Er führte `git rebase origin/main` aus, und das verschiebt keine Commits -- es erzeugt neue Kopien davon oben auf `main`, mit neuen Hashes, und lässt die Merge-Commits weg. Sein lokaler Branch hatte jetzt die Kopien; der Server hatte noch die Originale. Der Push wurde abgelehnt, weil beide auseinandergelaufen waren, und Jesse tat, was am Tag zuvor auf `main` funktioniert hatte: Er machte einen Pull. Der Pull merged die Originale direkt wieder hinein.



Ein Rebase erzeugt Kopien; ein Pull des alten Branches merged die Originale zurück

Das entscheidende Detail steckt im dritten Feld. Die rebasten Kopien sind der erste Parent des Merges, weil Jesse beim Pull auf diesem Branch stand. GitKraken zeigt das ohne jeden Befehl: Wählen Sie `33780e7` aus, und das Commit-Panel listet die Parents als `5b0e0f, 478a87` -- zuerst die rebaste Kopie, dann die alte Leiter. Alles, was wir für die Reparatur brauchen, ist also schon im Graphen: Die Kopien liegen auf einer sauberen Spur oben auf dem aktuellen `main`, und der Merge an der Spitze trägt nichts Neues bei.

Sie müssen sich dabei auch nicht auf das Wort des Graphen verlassen. Git kann beweisen, dass beide Spuren dieselbe Arbeit sind. `--cherry-mark` vergleicht die beiden Seiten des Merges inhaltlich und markiert gleichwertige Commits mit `=`, und der Merge selbst ändert gegenüber seinem ersten Parent überhaupt nichts:

```
$ git log --oneline --cherry-mark --left-right --no-merges 33780e7^1...33780e7^2
= 5b0e0f1 feat(menu): add family meal prices
= 6d43567 fix typo
= 2d16d8e feat(menu): show prices on the menu page
> 34ec852 fix(order): stop taking orders after 10 pm
= 0ab974b feat(menu): add family meal prices
= 5235256 fix typo
= 67c4443 feat(menu): show prices on the menu page
$ git diff --stat 33780e7^1 33780e7
$
```

Drei Paare mit `=` und ein leerer Diff. Der einzige Commit ohne Zwilling ist Mayas 22-Uhr-Fix, den die Originale schlicht nicht enthalten, weil sie älter sind.

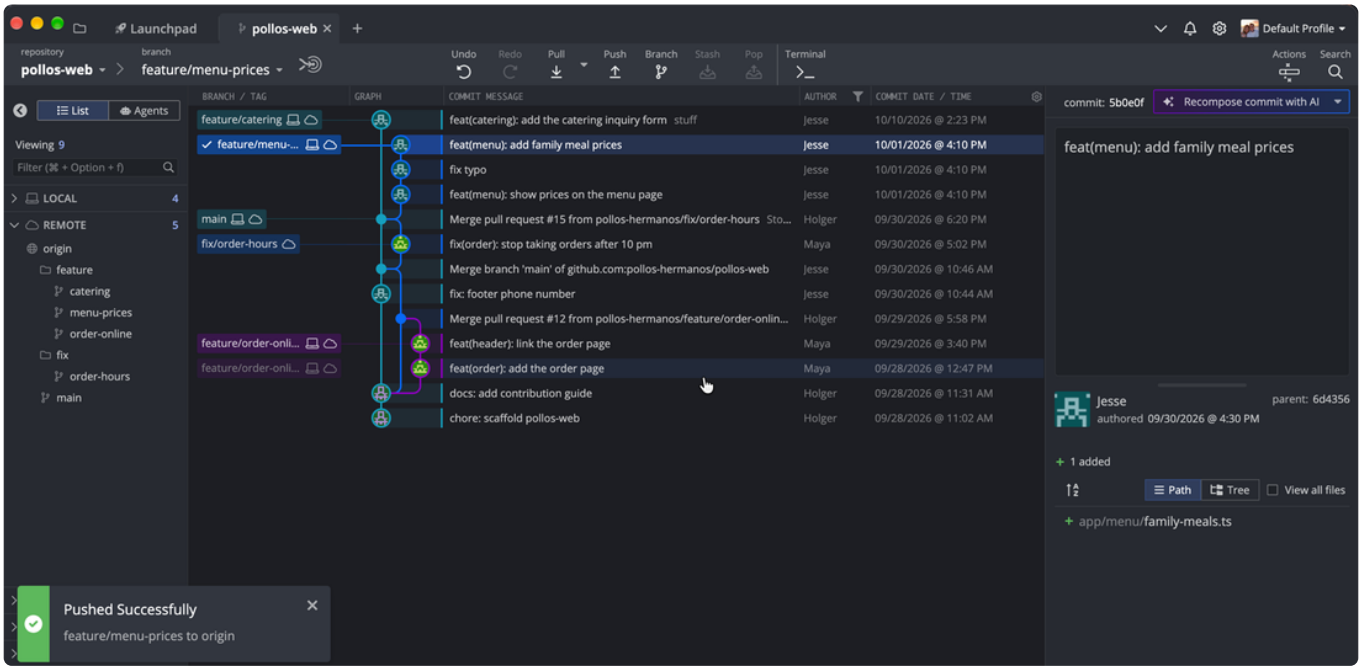
Die Lösung: auf die Kopien zurücksetzen, dann vorsichtig force-pushen

Da die saubere Version direkt vor uns liegt, ist die Reparatur ein Reset. Wir checken `feature/menu-prices` aus (ein Doppelklick auf den Remote-Branch legt den lokalen an), setzen ihn auf `5b0e0f1` zurück -- den rebasten Commit „family meal prices“, den ersten Parent des Merges -- und pushen mit Force. In GitKraken ist das ein Rechtsklick auf `5b0e0f1`, **Reset feature/menu-prices to this commit** und **Hard**. Der folgende Push wird wieder abgelehnt, und GitKraken zeigt denselben Banner wie beim Catering-Branch. Diesmal wissen wir es besser, als zu pullen, und wählen Force Push.

Auf der Kommandozeile verwende ich immer `--force-with-lease` statt eines einfachen `--force`. Es überschreibt den Branch auf dem Server nur, wenn er noch dort steht, wo Sie ihn zuletzt gesehen haben, sodass Sie nicht still einen Commit vernichten, den jemand vor einer Minute gepusht hat. Laut den Release Notes zu 9.6.1 verwendet auch GitKrakens Force Push standardmäßig `--force-with-lease`, wenn es die Git-Executable nutzt, und die ist standardmäßig aktiviert:

```
$ git switch feature/menu-prices
$ git reset --hard 33780e7^1
$ git push --force-with-lease
 + 33780e7...5b0e0f1 feature/menu-prices -> feature/menu-prices (forced update)
```

Zusammen mit dem rebasten `feature/catering` sieht der Graph jetzt so aus, wie er von Anfang an hätte aussehen sollen: zwei kurze Branches oben auf `main`, drei Commits für die Speisekarte, einer für das Catering.



Nach dem Aufräumen: Beide Branches von Jesse beginnen auf dem aktuellen <code>main</code>, ohne Duplikate und ohne Leiter. Der <code>git pull</code>-Merge aus Spur 1 bleibt bewusst auf <code>main</code>.

Die Falle: Jesses Laptop erinnert sich noch

Hier dachte ich, wir wären fertig, und genau hier hat mich mein Testlauf eines Besseren belehrt. Alles, was ich repariert habe, habe ich auf dem Server repariert. Jesses Laptop hat noch sein altes `feature/menu-prices`, das mit den Duplikaten, und ahnt nichts davon, dass irgendetwas passiert ist.

Also habe ich Jesse gespielt und getan, was er am nächsten Morgen tun würde: pullen, dann pushen.

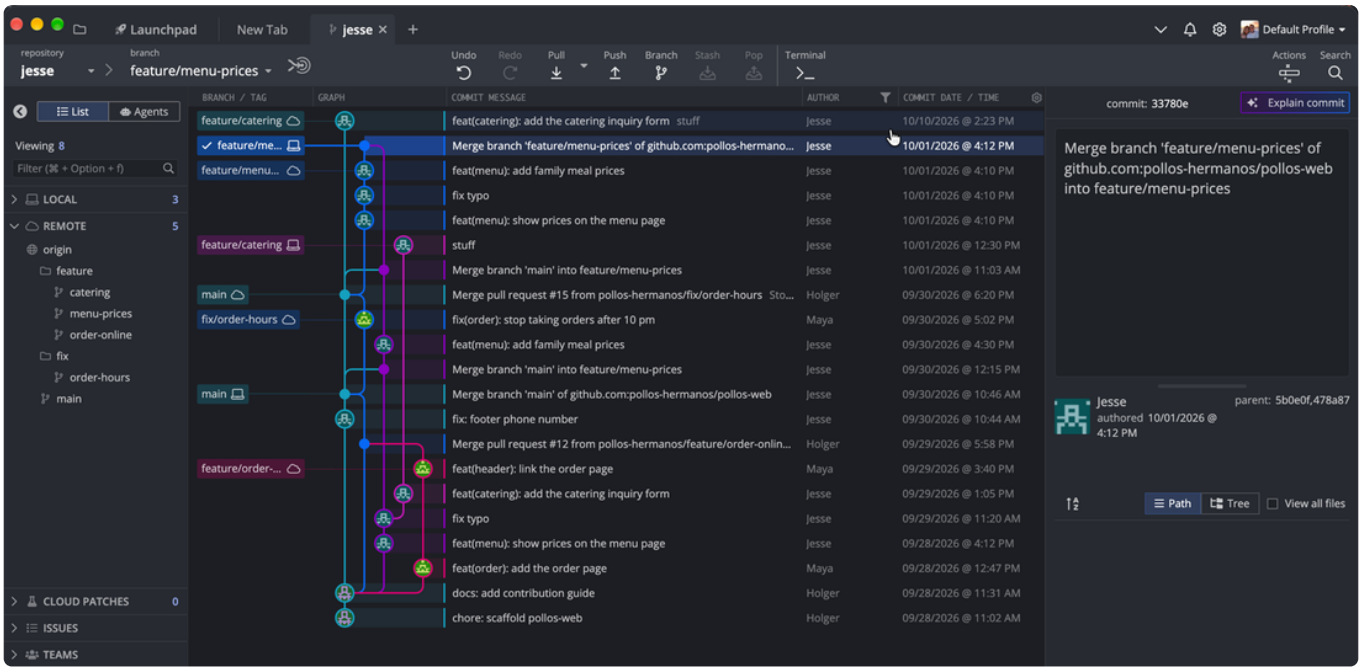
```
$ git status -sb
## feature/menu-prices...origin/feature/menu-prices
$ git pull
Already up to date.
$ git status -sb
## feature/menu-prices...origin/feature/menu-prices [ahead 6]
$ git push
   5b0e0f1..33780e7  feature/menu-prices -> feature/menu-prices
```

„Already up to date“ ist technisch korrekt und völlig irreführend. Aus Sicht von Git enthält Jesses Branch den aufgeräumten Branch plus sechs weitere Commits -- die Originale, die beiden Leiter-Merges und den Duplikat-Merge --, er ist also schlicht voraus. Und ein Branch, der voraus ist, lässt sich ganz ohne Force pushen. Ein einfaches `git push`, und jedes Duplikat ist zurück auf dem Server. Kein Fehler, keine Warnung.

In GitKraken ist diese Situation leicht zu erkennen, wenn man weiß, wohin man schauen muss. Erinnern Sie sich an den Laptop und die Wolke am Label `main` aus dem Überblick? Öffnen Sie Jesses Clone und

machen Sie einen Fetch, und bei seinen Branches haben sich die beiden getrennt. Das Laptop-Label von `feature/menu-prices` sitzt noch auf dem Duplikat-Merge `33780e7`; das Wolken-Label eine Zeile tiefer auf dem sauberen `5b0e0f`. Nach einem Fetch kann das linke Panel außerdem anzeigen, wie weit jeder Branch seinem Remote voraus ist oder hinterherhinkt.

Und es ist nicht nur der Speisekarten-Branch. Jesses lokales `feature/catering` zeigt noch auf den alten Commit „stuff“ -- 4 Commits voraus und 8 hinter seinem aufgeräumten Remote --, und sein lokales `main` sitzt noch auf seinem eigenen `git pull`-Merge aus Spur 1, zwei Commits zurück. Drei veraltete Branches, ein Laptop.



Jesses Clone nach einem Fetch: Die Laptop-Labels (seine lokalen Branches) und die Wolken-Labels (der Server) sitzen nicht mehr auf denselben Commits -- <code>feature/menu-prices</code> ist dem aufgeräumten Remote sechs Commits voraus.

Die Lösung auf Jesses Seite ist, seinen lokalen Branch an den Server anzugleichen -- nachdem sichergestellt ist, dass darauf keine neue Arbeit liegt, die nur auf seinem Laptop existiert:

```
$ git fetch
$ git reset --hard origin/feature/menu-prices
$ git status -sb
## feature/menu-prices...origin/feature/menu-prices
```

In GitKraken ist das ein Rechtsklick auf den Commit, auf den `origin/feature/menu-prices` zeigt, dann **Reset feature/menu-prices to this commit** und **Hard**. Liegt neue Arbeit auf dem veralteten Branch, übernehmen Sie sie vorher per Cherry-Pick auf den sauberen Branch. Und falls doch etwas schiefgeht: `git reflog` weiß noch, wohin jeder Branch vor dem Reset gezeigt hat.

Meine Regel nach dem Aufräumen eines fremden Branches lautet deshalb: vor dem Force Push mit der Person sprechen und sie danach durch den Reset begleiten. Stilles Aufräumen ist keine Hilfe; es stellt eine Falle für den nächsten Morgen auf.

Was ich im Team ändere, nicht nur im Repository

Vier Fehler zu reparieren dauert ein paar Minuten, sobald man sie lesen kann. Sie zu verhindern, hängt vor allem an zwei Einstellungen und einer Gewohnheit.

Die beiden Einstellungen kennen Sie aus Spur 1: `main` auf GitHub schützen, sodass niemand direkt darauf pushen kann, und `pull.ff only` setzen, sodass Git bei jedem auseinandergelaufenen Branch anhält und fragt, statt still zu mergen. Beides macht aus stillen Fehlern laute, und laute Fehler kommen mit einer Meldung, nach der auch ein Einsteiger suchen kann.

Die Gewohnheit ist, vor dem Pushen auf den Graphen zu schauen, und ganz besonders vor jedem Force Push. Für Entwickler, die neu bei GitHub sind, empfehle ich diese Routine: Vor dem Push öffnen Sie gemeinsam den Graphen und beantworten drei Fragen. Wo beginnt mein Branch? Sitzt mein lokales Branch-Label auf demselben Commit wie das Remote-Label, und wenn nicht, warum? Enthält dieser Pull Request nur meine Commits? Das dauert dreißig Sekunden, und meiner Meinung nach lehrt es mehr Git als jeder Spickzettel, weil der Entwickler seine eigene Arbeit als Bild sieht, statt Befehle auswendig zu lernen.

> **Alternativen**
> GitKraken ist nicht der einzige Weg, einen Commit-Graphen zu sehen. Git selbst liefert `gitk` mit, einen einfachen grafischen Historien-Browser, und `git log --graph` funktioniert überall, wie wir oben gesehen haben. In Visual Studio Code zeigt der eingebaute Source Control Graph die Historie seit Version 1.93 direkt neben Ihrem Code. Weitere Desktop-Clients mit Graph-Ansicht sind Fork, Tower und Sourcetree. GitHub Desktop konzentriert sich auf einen geradlinigen Commit-und-Sync-Workflow und zeigt die Historie als Liste; eine Graph-Ansicht ist ein langjähriger Feature-Wunsch, den das Team derzeit als nicht geplant markiert. Für die forensische Arbeit in diesem Beitrag zählt jede Ansicht, die die Spuren zeichnet und lokale und Remote-Labels nebeneinander zeigt. Ich verwende GitKraken, weil der Graph, der interaktive Rebase und die Pull-Request-Integration in einem Fenster zusammenleben.

Unterm Strich

Vier Fehler, und keiner davon war ein kaputtes Git. Ein Commit direkt auf `main`, mit `git pull` angeklebt. Ein Branch, der vom falschen Branch gestartet wurde. Eine Leiter aus völlig legitimen Update-Merges. Und ein Rebase eines bereits gepushten Branches, gefolgt von einem Pull, der jeden Commit in zwei Kopien hinterlassen hat. Jeder Fehler hat eine Signatur, die Sie in Sekunden im Graphen ablesen können, sobald Sie sie kennen, und für jeden gibt es eine sichere Reparatur, sobald Sie wissen, welche Commits die Kopien sind.

> Erst den Graphen lesen, dann reparieren -- und die Person, der der Branch gehört, sollte bei beidem dabei sein.

Wenn Sie mit Entwicklern arbeiten, die neu bei Git sind, möchte ich Sie um genau eine Sache bitten: Öffnen Sie deren Branch gemeinsam in einem Graphen, bevor Sie irgendetwas reparieren. Wenn Sie dafür GitKraken verwenden möchten, bekommen Sie mit meinem Empfehlungslink **50 % Rabatt auf GitKraken Pro**: GitKraken Pro ausprobieren. Zur Transparenz: Es ist ein Empfehlungslink, und ich bin GitKraken Ambassador. Die Community Edition ist für lokale Repositories und öffentliche Remotes kostenlos, und für die kostenpflichtigen Pläne gibt es eine kostenlose Testphase.

Wenn Sie tiefer einsteigen möchten: Mein früherer Beitrag über GitKrakens Graph-Ansicht ist eine Tour durch die alltäglichen Funktionen, und GitKraken Is Now the Code Flow Company erklärt, wohin sich das Unternehmen entwickelt. Und wenn sich zwei Branches nicht nur verheddern, sondern kollidieren, setzt mein Durchgang durch einen echten Merge-Konflikt in GitKraken genau dort an, wo dieser Beitrag endet.

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 —