ChatGPT

Roadmap zum Branch-Aufräumen der Claire_de_Binare-Repos

Ausgangslage

Die Repositories **Claire_de_Binare** und **Claire_de_Binare_Docs** enthalten eine Vielzahl an Feature- und Experiment-Branches. Einige dieser Branches sind bereits vollständig in `main` integriert, andere tragen noch Änderungen oder Dokumentationen, die nicht verloren gehen dürfen. Der Nutzer hat eine Liste der Branches bereitgestellt und wünscht sich einen strukturierten, verlustfreien Bereinigungsplan. Die folgende Roadmap fasst die Analyse zusammen und schlägt ein strukturiertes Vorgehen zur Konsolidierung vor.

Ziele

- **Alles auf `main` bringen:** Sicherstellen, dass sämtliche relevanten Änderungen aus Nebenbranches auf den Branch `main` überführt werden.
- **Branches bereinigen:** Überflüssige, veraltete oder vollständig integrierte Branches löschen, sobald ihre Inhalte sicher auf `main` sind.
- **Struktur schaffen:** Branch-Namensgebung konsolidieren (z. B. Themen- oder Feature-Branches klar benennen), Dokumentationen an konsistente Orte verschieben und sinnvolle Ordnerstrukturen schaffen.
- **Keine Verluste:** Vor dem Löschen von Branches sicherstellen, dass keinerlei Änderungen verloren gehen.
- **Automatisierung nutzen:** Wo möglich, Skripte oder Workflows einsetzen, um die Branch-Analyse zu automatisieren (z. B. `git`-Kommandos, GitHub-APIs).

Phase 1 – Branch-Analyse

Vorgehensweise

1. **Vollständiges Fetch und Prune:** Führe in beiden Repositories `git fetch --all --prune` aus, um alle entfernten Branches zu laden und verwaiste Referenzen zu entfernen.
2. **Ablauf pro Branch:** Für jede vorhandene Remote-Branch:
3. **Letztes Commit bestimmen:** Ermittele das Datum des letzten Commits auf dem Branch.
4. **Vergleich mit `main`:** Ermittle mit `git rev-list` oder der GitHub API, wie viele Commits der Branch `main` voraus bzw. hinterher ist. Ein einfacher Vergleich (`git fetch` + `git branch -r --merged`) zeigt, ob ein Branch schon vollständig in `main` enthalten ist.
5. **Kategorisierung:** Einordnen in eine der drei Kategorien:
    - `old_merged` **(Alt und gemerged):** Branch ist älter als 30 Tage und enthält keine eigenen Commits mehr (ahead_by = 0). Beispiele aus dem Docs-Repo: `chore/update-codeowners` (0 Commits voraus, 75 hinterher【898592089302054†L5-L8】), `copilot/add-issue-translation-integration`【858101447218889†L5-L8】, `copilot/add-user-authentication`【769687149930781†L5-L8】, `copilot/improve-deep-`

`issues-pipeline`【394949332548931†L5-L8】 und `frosty-panini`【206925574739054†L5-L8】 sind bereits vollständig auf `main` und können nach der Freigabe gelöscht werden.
    - `old_unmerged` **(Alt aber nicht gemerged):** Branch ist älter als 30 Tage, hat aber noch eigene Commits (z. B. ahead_by ≥ 1). Hier muss entschieden werden, ob die Änderungen verworfen, manuell gemerged oder separat dokumentiert werden sollen. Ein Beispiel aus dem Docs-Repo ist `copilot/promote-selected-branch-to-main` mit einem Commit Vorsprung【851330085872460†L5-L8】.
    - `active` **(Aktiv):** Branches mit eigenem Entwicklungsfortschritt oder aktuellen Änderungen (ahead_by > 0, in der Regel jünger als 30 Tage). Diese sollten nach dem Testen per Pull Request in `main` integriert werden. Im Docs-Repo fallen `docs/all-in-20251225-1331`【542472469101577†L5-L8】 (3 Commits voraus) und `docs/baseline-scaffolding`【722421834678802†L5-L8】 (umfassende Umstrukturierung) in diese Kategorie.

6. **Dokumentation in einer Tabelle:** Halte die Ergebnisse in einer Markdown-Tabelle fest. Für das Docs-Repo ergibt sich folgende Übersicht:

Branch	ahead_by (Commits vor `main`)	behind_by	Kategorie	Erläuterung
`chore/update-codeowners`	0	75	**old_merged**	Keine eigenen Commits; Branch ist komplett hinter `main`【898592089302054†L5-L8】.
`copilot/add-issue-translation-integration`	0	71	**old_merged**	Vollständig auf `main`【858101447218889†L5-L8】.
`copilot/add-user-authentication`	0	53	**old_merged**	Vollständig auf `main`【769687149930781†L5-L8】.
`copilot/improve-deep-issues-pipeline`	0	16	**old_merged**	Enthält keine eigenen Commits【394949332548931†L5-L8】.
`frosty-panini`	0	59	**old_merged**	Vollständig integriert【206925574739054†L5-L8】.
`copilot/promote-selected-branch-to-main`	1	75	**old_unmerged**	Ein Commit ist nicht auf `main`【851330085872460†L5-L8】. Prüfen, ob er benötigt wird.

Branch	ahead_by (Commits vor `main`)	behind_by	Kategorie	Erläuterung
`docs/all-in-20251225-1331`	3	3	**active**	Drei Commits voraus 【542472469101577†L5-L8】, enthält zahlreiche neue Inhalte (neue Playbooks, Roadmaps, Umstrukturierung).
`docs/baseline-scaffolding`	1	1	**active**	Umfasst eine komplette Neu-Strukturierung der Agent-Dokumentation und viele neue Dateien 【722421834678802†L5-L8】.

Für das **Claire_de_Binare**-Repo ist die Branch-Landschaft deutlich umfangreicher (über 70 Branches). Da die GitHub-API die Commit-Zeitstempel der Branch-Heads nicht direkt liefert, empfiehlt es sich, das Repository lokal zu klonen und folgende Schritte zu automatisieren:

- **Liste aller Branches abrufen:** `git branch -r | grep -v main`.
- **Ahead/Behind ermitteln:** Für jeden Branch `BRANCH` kann `git rev-list --left-right --count main...BRANCH` die Anzahl der Commits vor und hinter `main` liefern. Alternativ kann über die GitHub API `compare_commits` genutzt werden (z. B. `feat/230-guards-integration-e2e` hat 4 Commits Vorsprung und 27 Commits Rückstand 【566831083044927†L5-L8】, `dependabot/pip/pip-81350e123e` hat 1 Commit Vorsprung 【708136259994778†L5-L8】). Branches mit `ahead_by = 0` (z. B. `chore/governance-canon-final` 【214882050595010†L5-L7】) sind bereits integriert.
- **Datum des letzten Commits:** `git log -1 --format=%ci origin/BRANCH` liefert das Datum, um zwischen „alt“ (älter als 30 Tage) und „aktiv“ zu unterscheiden.
- **Kategorisierung:** Wie oben beschrieben.

Da einige Branch-Namen auf automatisch generierte, thematische Gruppen hinweisen, kann bereits vor der detaillierten Analyse grob eingeordnet werden:

- `copilot/…` **-Branches:** Automation- oder Build-Themen. Viele davon (etwa `copilot/expanded-tool-ecosystem` mit 15 Commits Vorsprung 【147989063565707†L5-L8】) enthalten neue Features oder Skripte. Prüfen, ob sie in `main` übernommen oder obsolet sind.
- `claude/…` **-Branches:** Häufig experimentelle Sessions des Agents „Claude“. Diese Branches sollten auf nützliche Inhalte (z. B. Spezifikationen, Prototypen) überprüft werden. Nicht mehr benötigte Branches können gelöscht werden, wenn die Inhalte anderweitig dokumentiert sind.
- `dependabot/…` **-Branches:** Automatische Abhängigkeits-Updates. Wenn sie bereits in `main` integriert sind (ahead_by = 0), können sie gelöscht werden; sonst sollte das Update entweder auf `main` manuell übernommen oder die PR gemerged werden.
- `feat/…`, `fix/…`, `infra/…`, `hardening/…` **und ähnliche Branches:** Diese enthalten wahrscheinlich produktive Features oder Bugfixes. Sie sind vorrangig als **active** einzuordnen, bis eine Prüfung ergibt, dass sie obsolet sind.
- `revert-…`, `reset-…`, `cool-moser`, `festive-shamir`, **usw.:** Oft temporäre oder automatisch erzeugte Branches, die hinter `main` liegen. Wenn sie keine eigenen Commits mehr haben und älter als 30 Tage sind, gehören sie in `old_merged`.

Phase 2 – Integrations- und Löschprozess (nach Freigabe)

Sobald die Analyse abgeschlossen und alle Branches kategorisiert sind, folgt der eigentliche Bereinigungsprozess. ***Wichtig:*** Bevor Branches gelöscht oder force-gepusht werden, muss der Nutzer explizit die Freigabe mit `CONFIRM_CLEANUP` erteilen. Bis dahin darf nur analysiert werden.

1. **Löschen von `old_merged`-Branches:**
2. Für jeden Branch in dieser Kategorie kann der entfernte Branch (`origin/BRANCH`) gelöscht werden (`git push origin --delete BRANCH`). Halte den Löschvorgang in einer Logdatei (`branch_cleanup_log.md`) fest (Branch-Name, letzter Commit-Hash, Löschzeitpunkt).
3. **Bearbeiten von `old_unmerged`-Branches:**
4. Prüfe die Änderung im Branch (z. B. via `git diff main..BRANCH` oder GitHub-Vergleich). Entscheide, ob die Commits noch benötigt werden. Falls ja, erstelle einen Pull Request gegen `main` mit Titel `[CLEANUP][LEGACY] <branch-name> into main` und einem kurzen Diff-Summary. Falls die Änderungen obsolet sind, dokumentiere sie (z. B. in einer Wiki-Seite) und lösche den Branch.
5. **Aktive Branches mergen:**
6. Aktualisiere `main` (`git checkout main && git pull origin main`).
7. Für jeden **active**-Branch: `git checkout BRANCH`, `git merge origin/main` und lokale Tests ausführen (z. B. über `pytest`, `npm test`, `make test` oder vorhandene CI-Workflows). Falls alles sauber ist, erstelle einen PR `[CLEANUP] Merge <branch-name> into main` und merge diesen (falls CI erfolgreich). Lösche anschließend den Branch und notiere dies im Log.
8. Bei Merge-Konflikten oder fehlschlagenden Tests: Erstelle trotzdem einen PR, markiere ihn als „manual review required“ und dokumentiere die Probleme im Log.
9. **Ständige Dokumentation:** Führe während des gesamten Prozesses ein `branch_cleanup_log.md`, in dem alle Aktionen (Löschung, erstellte PR-Links, aufgetretene Konflikte) nachvollziehbar festgehalten werden.

Phase 3 – Repository-Strukturierung und Governance

Nach dem eigentlichen Branch-Aufräumen lohnt sich eine strukturelle Überarbeitung der beiden Repositories:

- **Branch-Namenskonventionen definieren:** Verwende klare Präfixe (z. B. `feat/…`, `fix/…`, `docs/…`, `chore/…`) und kennzeichne Ticket-IDs oder Themen konsistent. Automatisch generierte Branch-Namen von Copilot/Claude sollten künftig in strukturierte Tasks überführt werden, statt sie dauerhaft als Branches zu belassen.
- **Dokumentations-Ordnung:** Im `Docs`-Repo wurde mit `docs/all-in-20251225-1331` und `docs/baseline-scaffolding` bereits eine Umstrukturierung begonnen (z. B. Verschiebung von `docs`-Inhalten nach `knowledge/playbooks` 【542472469101577†L20-L40】 und Erstellung eines neuen Agent-Setup-Verzeichnisses【722421834678802†L21-L43】). Führe diese Struktur konsistent fort: zentrale Inhalte nach `knowledge/`, Vorlagen in `knowledge/templates/`, Playbooks und Runbooks in eigene Ordner.

- **Wiederholungen zusammenführen:** Mehrere Branches (z. B. `feat-unified-secret-loader` und `introduce-unified-secret-loader` bzw. `copilot/expanded-tool-ecosystem` und `test/emoji-filter-demo`) scheinen ähnliche Features zu implementieren. Vergleiche die Änderungen, wähle die aktuellste Variante und vereine sie in einem Branch, bevor du in `main` mergest.
- **Automatisierte CI/CD:** Integriere durchgehende Tests und Qualitätsprüfungen in den Workflows (`.github/workflows`). Dadurch lassen sich Merge-Konflikte und Regressionen früh erkennen. Die `copilot/expanded-tool-ecosystem`-Branch fügt z. B. umfangreiche Workflows und Tests hinzu【147989063565707†L20-L36】 – diese sollten nach erfolgreicher Integration in `main` beibehalten werden.
- **Versionierung und Releases:** Überlege, künftig Releases über Tags oder GitHub-Releases zu managen. Dadurch werden fertige Iterationen klar markiert und Nebenbranches entstehen mit sauberer Grundlage.

Zusammenfassung und nächste Schritte

1. **Analyse abschließen:** Vollständige Liste aller Branches lokal abrufen und per Skript das Datum des letzten Commits sowie Ahead/Behind-Status gegenüber `main` ermitteln.
2. **Kategorie-Tabelle erstellen:** Branches in `old_merged`, `old_unmerged` und `active` einordnen. Für `Claire_de_Binare_Docs` wurde dies bereits exemplarisch gezeigt (siehe Tabelle oben). Für das Haupt-Repo muss dieser Schritt noch umfassend durchgeführt werden.
3. **Freigabe abwarten:** Erst nach der ausdrücklichen Rückmeldung mit `CONFIRM_CLEANUP` sollte die Umsetzung beginnen.
4. **Durchführung:** Altes löschen, aktive Branches in `main` integrieren und dabei Tests ausführen. Alles in der Logdatei dokumentieren.
5. **Struktur und Governance etablieren:** Klare Namenskonventionen, feste Ordnerstrukturen und automatisierte Workflows verhindern künftig Wildwuchs und erleichtern die Wartung.

Diese Roadmap liefert einen strukturierten, sicheren Plan, um die vielen Branches beider Repositories ohne Datenverlust zu konsolidieren und langfristig die Code- und Dokumentationsqualität zu erhöhen.
