ChatGPT

Analyse des Projekts "checkzombies"

1. Allgemeine Bewertung des Projekts

Reifegrad und Vollständigkeit: Das Projekt **checkzombies** macht bereits einen fortgeschrittenen Eindruck, weist aber noch Unvollständigkeiten auf. Positiv hervorzuheben ist, dass wichtige Komponenten vorhanden sind: Ein **Shell-Skript** als Hauptprogramm, eine **Manpage** zur Dokumentation und sogar erste **Unit-Tests** sowie **CI/CD-Integration**. Die README listet ambitionierte Features wie systemd-Integration, interaktives Menü, Auto-Modus, vollständige Manpage, Tests und Debian/RPM-Pakete [1]. Viele dieser Funktionen sind konzeptionell vorhanden, jedoch sind einige noch nicht vollständig umgesetzt. Insbesondere existieren in der Manpage dokumentierte Optionen (`--list`, `--watch`, `--force`), die im Skript selbst (noch) keine Entsprechung finden – ein Hinweis auf fehlende Vollständigkeit in der Implementierung [2] [3]. Der **Reifegrad** ist somit als **mittelmäßig** einzustufen: die Grundlage ist gelegt und das Tool läuft für Standardfälle (Zombie-Suche und -Bereinigung interaktiv oder automatisch), doch es gibt funktionale Lücken und Raum für Verbesserungen.

Dokumentation: Die Dokumentation ist für ein junges Projekt relativ umfangreich. In der **README** wird der Zweck knapp erläutert („Linux-Tool zum Finden und sicheren Bereinigen von Zombie-Prozessen“ [4]) und ein **Schnellstart** zur Installation geboten (Download des Skripts nach `/usr/local/bin` und Aufruf der Manpage) [5]. Außerdem sind die Features in der README stichpunktartig aufgeführt [1]. Der Hauptteil der Dokumentation liegt in der **Manpage**, welche detailliert Befehlsaufruf, Optionen und Beispiele beschreibt. Die Manpage ist gut strukturiert mit den üblichen Sektionen (NAME, SYNOPSIS, DESCRIPTION etc.) und listet alle vorgesehenen Optionen mit Beschreibung [2]. Beispiele zeigen die Nutzung interaktiv, im Auto-Modus sowie das Pipen der Liste oder eine Live-Überwachung via externem `watch`-Kommando [6] [7]. Die Dokumentation ist verständlich geschrieben (in deutscher Sprache) und für Anwender hilfreich. **Kritisch** anzumerken ist jedoch die Diskrepanz zwischen Dokumentation und aktueller Funktionalität: Anwender könnten laut Manpage Optionen wie `-l/--list` erwarten, die im Tool keine Wirkung zeigen. Ebenso fehlt eine im Programm integrierte Hilfe (`-h`) und Versionsanzeige (`-v`), obwohl diese Optionen dokumentiert sind [8]. Hier sollte die Dokumentation entweder angepasst oder – besser – die Implementierung vervollständigt werden, um Benutzer nicht in die Irre zu führen.

Benutzerführung und Installation: Die Bedienung des Tools erfolgt entweder **interaktiv** (Standardaufruf ohne Optionen) oder automatisiert via `--auto` (sofern implementiert). Die interaktive **Benutzerführung** besteht aus einem einfachen Menü im Terminal, das Optionen anbietet, alle Zombie-Elternprozesse zu terminieren oder einzelne auszuwählen [9]. Dieses Menü ist grundsätzlich verständlich, nutzt sogar Symbole/Emojis zur Hervorhebung (z.B. „ Ungültig“ bei falscher Eingabe [10]), was die Erkennbarkeit von Status verbessert. Allerdings wird im Tool selbst kein Hilfetext angezeigt, wenn ein Nutzer eine falsche Option angibt oder `checkzombies -h` ausführt – statt einer Usage-Ausgabe landet man im interaktiven Menü oder erhält „Keine Zombie-Prozesse aktiv“ bei 0 Zombies, was verwirrend sein kann. Für die Installation bietet das Projekt mehrere Wege: ein manueller Download des Skripts (z.B. via `curl` in der README gezeigt) [5], oder den klassischen Build/Install-Weg über das bereitgestellte **Makefile**. Das Makefile ermöglicht das Kopieren des Skripts nach `/usr/local/bin` und der Manpage nach `/usr/local/man/man1` inklusive Komprimierung und `mandb`-Aufruf [11]. Ferner sind Ansätze für Paketierung

vorhanden: Eine **Debian-Paket**-Definition ist begonnen (debian/control vorhanden [12], `make deb` ruft dpkg-buildpackage auf [13]) und die README erwähnt sogar RPM-Pakete [14]. Das bedeutet, perspektivisch ist eine Installation über System-Pakete vorgesehen, was für Benutzer bequem und professionell wäre. Aktuell müssen Nutzer das Tool aber vermutlich noch manuell installieren oder die GitHub-Releases nutzen. Insgesamt ist die **Benutzerfreundlichkeit** schon ordentlich (Dank Manpage und Menü), kann aber durch vollständige Implementierung der vorgesehenen Optionen und verbesserte Hilfsanzeigen weiter gesteigert werden.

2. Code-Qualität des Shell-Skripts `checkzombies`

Stil und Lesbarkeit: Der Shell-Code in `bin/checkzombies` ist mit ~80 Zeilen kompakt und relativ verständlich strukturiert. Es gibt einen klar abgegrenzten Hauptteil (`main()`-Funktion) sowie Hilfsfunktionen wie `find_zombies()` zur Prozesssuche und `log()` zum Protokollieren [15]. Kommentare gliedern den Ablauf in Schritte (Zombie finden, anzeigen, interaktives Menü etc.) [16], was die Nachvollziehbarkeit erhöht. Die Wahl selbsterklärender Bezeichner (z.B. `zombie_list`, `count`) und die Formatierung (Einrückungen, Verwendung von `printf` für tabellarische Ausgabe [17]) tragen zur Lesbarkeit bei. Der Code nutzt moderne Bash-Möglichkeiten wie `$(...)`-Substitution, `[[ ]]` für Vergleiche und Herunterkonvertierung von Eingaben (`${choice,,}`) [18]. Allerdings gibt es kleinere Unstimmigkeiten: Eine Variable `zombies` wird deklariert aber nie verwendet [19], und die Kommando-Spalten aus `ps` werden mit `awk` recht starr ausgewertet (nur `$11` und `$12` werden erfasst [20]), was bei Prozessen mit langen oder vielen Argumenten unvollständige Kommandonamen liefern könnte. Insgesamt ist der Stil solide, könnte aber durch Konsistenz und Best Practices (siehe unten: `set -euo pipefail`, umfassendere Quoting-Regeln) weiter professionalisiert werden.

Robustheit und Sicherheitsaspekte: Hier bestehen noch **Verbesserungspotentiale**. Wichtige Sicherheitsmechanismen für Shell-Skripte wie `set -e` (Abbruch bei Fehlern), `set -u` (Ungenutzte/ungesetzte Variablen behandeln als Fehler) und `set -o pipefail` (Fehlererkennung in Pipelines) werden nicht verwendet [21]. Ohne diese kann das Skript trotz auftretender Fehler weiterlaufen, was im Falle unerwarteter Situationen (z.B. ein fehlgeschlagenes `ps` oder `systemctl`-Kommando) zu inkonsistentem Zustand führen könnte. So wird beispielsweise am Ende der Pipeline in `find_zombies()` mit `|| true` ein eventueller Fehler unterdrückt [20] – ein gängiges Muster, um bei fehlenden Zombies kein Skript-Abbruch zu provozieren, doch zusammen mit fehlendem `pipefail` könnte ein echtes Fehlerereignis unbemerkt bleiben. Auch eine explizite **Eingabevalidierung** findet nur punktuell statt: Bei der Verarbeitung der ps-Ausgabe wird mittels Regex geprüft, dass die gelesene PID wirklich numerisch ist [22], und im interaktiven Modus wird Benutzereingabe für Menüwahl und PID-Auswahl auf erwartetes Format validiert [10]. Das ist positiv, sollte aber ausgebaut werden – z.B. sollten ungültige Kommandozeilenargumente frühzeitig erkannt und mit einer Fehlermeldung abgefangen werden, anstatt sie ins Menü zu führen.

Sicherheitsrelevante Aspekte wie **Privilege Handling** sind pragmatisch, aber nicht optimal gelöst: Das Skript ruft zum Beenden von Prozessen und zum Restart von Diensten teils `sudo` auf [23]. Dies deckt zwar den Fall ab, dass ein Benutzer nicht als root läuft, kann aber in automatisierten Umgebungen (z.B. Cron oder CI) zu Problemen führen, falls keine sudo-Rechte oder -Passwörter verfügbar sind. Ein professionelles Tool würde entweder voraussetzen, als root ausgeführt zu werden (und dies prüfen) oder bewusst alle nötigen Rechte über `sudo` *außerhalb* des Skripts einholen. Die Verwendung von `sudo` im Skript selbst sollte konsistent sein – im Code wird im Auto-Modus `sudo` vor `systemctl restart` genutzt [24], aber

beim KILL im Einzelmodus nicht [25], was zu *"permission denied"*-Fehlern führen kann, falls der Nutzer dort nicht root ist. Hier wäre Einheitlichkeit wichtig.

Modularisierung und Erweiterbarkeit: Das Skript ist bislang monolithisch, was aufgrund der geringen Größe vertretbar ist. Einige Funktionen (z.B. `find_zombies`) kapseln bestimmte Aufgaben [20], was Wartung erleichtert. Allerdings werden aktuell alle Abläufe – Auflistung, Auto-Bereinigung, Interaktivmodus – in einer einzigen Funktion `main` abgebildet, verzweigt nur durch einfache `if` / `case`-Blöcke [18]. Neue Betriebsmodi oder Optionen einzufügen, erfordert Änderungen direkt an dieser Struktur. Beispielsweise ist die Integration der Option `--list` (nur auflisten und beenden) oder `--watch` (periodisch prüfen) momentan nicht vorgesehen und würde zusätzlichen Code im Hauptablauf erfordern. Besser wäre ein modularer Aufbau, wo die Optionen zunächst geparst werden und dann separate Funktionen aufgerufen werden, je nach Modus. Dies würde die **Erweiterbarkeit** erhöhen. Auch könnte man Logik, die mehrfach auftritt, auslagern: z.B. das Terminationsverfahren eines einzelnen Zombie-Parents (Signal senden, ggf. Service neu starten, Erfolg prüfen) als eigene Funktion. So würde auch die Anpassung (etwa Einführung einer Wartezeit und erneuten Prüfung, ob der Zombie entfernt wurde) an einer Stelle genügen. Kurzum: Aktuell ist die Modularisierung rudimentär; für künftige Erweiterungen (neue Optionen, komplexere Entscheidungslogik) sollte die Struktur noch verfeinert werden.

Logging und Fehlerbehandlung: Eine einfache Logging-Funktion existiert [15], welche mit Zeitstempel (allerdings fest zur Startzeit des Skripts generiert, nicht dynamisch pro Logeintrag) in eine Logdatei im Home-Verzeichnis schreibt [26]. So werden gefundene Zombies beispielsweise mit einem Header-Eintrag protokolliert [27]. Allerdings ist die Protokollierung uneinheitlich: Viele Aktionen (etwa das Senden von KILL-Signalen oder Neustarten von Diensten) werden nur per `echo` am Terminal ausgegeben [23], aber nicht ins Log geschrieben. Hier sollte konsequenter geloggt werden – idealerweise jede wichtige Aktion und ggf. Fehler. Zudem ist die Wahl der Logdatei `/home/dev/checkzombies.log` in der Manpage [28] bzw. `${HOME}/checkzombies.log` im Skript [26] zu hinterfragen. In einem professionellen Umfeld würde man Logs eher nach `/var/log` legen oder zumindest den Pfad konfigurierbar gestalten. Die Fehlerbehandlung im Skript beschränkt sich darauf, Fehler hauptsächlich durch Meldungen ans Terminal kenntlich zu machen. Beispielsweise wird ausgegeben, wenn ein Parent-Prozess bereits nicht mehr existiert, an den man ein SIGTERM schicken wollte [29]. Dieser Fall wird also erkannt und gemeldet, aber das Skript unternimmt keine weiteren Schritte (was ok ist, da der Zombie dann meist schon von `init` geerntet wurde). Bei ungültiger Benutzereingabe im Menü wird abgebrochen mit Fehlercode 1 [10], was korrekt ist. Nicht behandelt werden jedoch potentielle Fehler wie ein Scheitern des `systemctl restart` (z.B. falls der Service nicht ermittelt werden konnte) – das Skript ignoriert exit codes dieser Kommandos (führt sie zwar aus, aber prüft das Ergebnis nicht, abgesehen vom kurzen `&&` in Auto-Modus, welches Erfolg/Nicht-Existenz unterscheidet [29]). Hier könnte robustere Fehlerbehandlung helfen, etwa Rückgabewerte auswerten und im Misserfolgsfall entsprechende Warnungen oder andere Schritte einleiten. Insgesamt funktioniert das Skript in Fehlerfällen meist **weiter**, was gut ist für Resilienz, aber es meldet diese Fehler nur spärlich und verlässt sich auf manuelle Beobachtung im Terminal bzw. Log.

Signalverarbeitung und PID-Handling: Eine spezielle Behandlung von Signalen (etwa `SIGINT` für vorzeitiges Beenden) findet im Code nicht statt – `trap` wird nicht verwendet. Im interaktiven Normalbetrieb ist das nicht kritisch (Strg+C bricht das Skript einfach ab), jedoch wäre für einen geplanten Watch-/Daemon-Modus eine saubere Signalverarbeitung ratsam (z.B. um bei Strg+C einen letzten Logeintrag zu schreiben oder den Bildschirmcursorzustand zurückzusetzen, falls man irgendwann curses einsetzen würde). Das **PID-Handling** ist im Kontext des Tools vor allem das **Identifizieren von Zombies und ihrer Eltern** sowie das **gezielte Signalisieren dieser Elternprozesse**. Die Erkennung erfolgt über eine

Pipeline `ps aux | awk ...` die Prozesse mit Status `Z` filtert [20]. Dieses Vorgehen klappt grundsätzlich, hängt jedoch vom Output-Format von `ps` ab. Alternativ könnte man robustere Ansätze überlegen (z.B. `/proc/*/status` parsen nach `State: Z`), aber für eine erste Umsetzung ist `ps+awk` ausreichend und portabel. Die gefundenen **Parent-PIDs (PPID)** der Zombies werden im Skript weiterverarbeitet: Es wird versucht, anhand der PPID einen zugehörigen systemd-Service zu ermitteln (durch `systemctl status <PPID>` und Parsen der „Main PID“-Zeile) [30]. Gelingt dies, erhält man den Servicenamen, ansonsten wird `"none"` gesetzt. Diese Logik ist ein nützliches Feature, um z.B. anzuzeigen, dass ein Zombie zu einem bekannten Dienst gehört, den man statt des Prozesses ggf. neu starten sollte. In der Praxis könnte diese Erkennung jedoch unzuverlässig sein – nicht jeder Prozess mit Zombie ist ein Main-PID eines systemd-Services. Das Skript nutzt diese Info dann: Beim Bereinigen werden **Signale in Eskalationsstufen** geschickt. Im Auto-Modus versucht `checkzombies`, für jeden gefundenen Zombie-Parent zunächst einen sanften SIGTERM zu senden; sollte der Prozess schon weg sein, wird das gemeldet [29]. Im interaktiven "Alle terminieren" Modus wird ebenfalls SIGTERM genutzt und falls dieser fehlschlägt, sofort mit SIGKILL (`kill -KILL`) nachgesetzt [31] – eine sinnvolle Eskalation. Im Einzelwahl-Modus hingegen wird nach Benutzerbestätigung direkt SIGKILL gesendet [25], was inkonsistent zu den anderen Modi ist (hier könnte man überlegen, zunächst ebenfalls SIGTERM anzubieten). Insgesamt werden PIDs korrekt behandelt (zahlenmäßig validiert, in Ausgaben formatiert) und mit den nötigen Signalen versehen. Eine Ergänzung für Professionalität wäre eventuell, nach dem Kill zu überprüfen, ob der Zombie tatsächlich verschwunden ist und dies zu protokollieren – momentan verlässt man sich darauf, dass das Senden des Signals genügt. Auch eine Berücksichtigung besonderer PIDs (z.B. PPID 1 – init – als Sonderfall, den man natürlich nicht killen darf) fehlt im Skript, wobei Zombies mit PPID 1 äußerst unwahrscheinlich sind, da init solche Prozesse normalerweise sofort aufräumt.

3. Bewertung der Manpage `checkzombies.1`

Die Manpage ist bereits sehr ordentlich gestaltet und für ein Tool dieser Größe außergewöhnlich umfassend. **Struktur und Verständlichkeit:** Alle wichtigen Abschnitte einer Manpage sind vorhanden: NAME (mit Kurzbeschreibung) [32], SYNOPSIS (Aufruf mit Optionen) [33], DESCRIPTION (Erläuterung der Funktion) [34], OPTIONS (Beschreibung aller Optionen) [2], BEISPIELE [6], FILES (Logfile-Angabe) [28], AUTHOR und COPYRIGHT. Diese klare Gliederung erleichtert es dem Nutzer, schnell Informationen zu finden. Die Sprache ist Deutsch und zielgruppengerecht formuliert – präzise und knapp, aber verständlich. Besonders die Beispiele sind hilfreich, da sie typische Anwendungsfälle zeigen (interaktive Nutzung, automatisches Clean-up, reine Liste oder Verwendung des Unix-`watch` Kommandos zur Live-Überwachung) [35] [36]. Durch diese Beispiele können Anwender direkt erkennen, wie das Tool praktisch eingesetzt werden kann.

Vollständigkeit der Optionen: Die Manpage listet alle vorgesehenen Optionen samt kurzer Erklärung [2]. Dazu gehören `-h/--help` und `-v/--version` (obwohl diese derzeit keine Implementierung im Skript haben), die Standardoptionen darstellen – ihre Dokumentation zeugt von dem Anspruch, sich an gängige Konventionen zu halten. Die Kernfunktionen werden über `-a/--auto` (automatische Bereinigung aller Zombies), `-l/--list` (nur auflisten), `-w/--watch` (kontinuierliche Überwachung) und `-f/--force` (erzwingt KILL ohne Bestätigung) abgedeckt [2]. Inhaltlich deckt das alle sinnvollen Modi ab. Kritisch anzumerken ist jedoch, dass – wie bereits bei der Projektbewertung erwähnt – einige dieser Optionen im aktuellen Skript keine Wirkung haben. Insbesondere `--list` und `--watch` werden in der Implementierung ignoriert [3], und `--force` findet sich gar nicht im Code wieder. Das heißt, die Manpage beschreibt hier einen Soll-Zustand, dem der Ist-Zustand noch nicht entspricht. Für eine

professionelle Qualität muss die Dokumentation entweder dem Programm angepasst oder umgekehrt das Programm der Dokumentation entsprechend erweitert werden. Solche Abweichungen können Nutzer verwirren und sollten daher baldmöglichst behoben werden.

Integration mit dem System: Die Manpage ist für Section 1 (User Commands) vorgesehen [37] und wird im Rahmen der Installation in das übliche Verzeichnis `/usr/local/man/man1/` installiert und gzip-komprimiert [11]. Dies entspricht den Konventionen (lokale Installation unter `/usr/local`; in Distribution-Paketen würde man vermutlich `/usr/share/man/man1` nehmen, aber das lässt sich über PREFIX im Makefile steuern). Der Aufruf von `mandb` im Makefile sorgt dafür, dass die neue Manpage in die Datenbank aufgenommen wird [11] – ein guter Schritt, den weniger erfahrene Projekte oft vergessen. Ein kleiner Kritikpunkt ist der **FILES**-Abschnitt: Dort ist das Logfile mit festem Pfad `/home/dev/checkzombies.log` angegeben [28]. Dieser Pfad wirkt wie ein Platzhalter aus der Entwicklungsumgebung und sollte für den Endnutzer angepasst werden (z.B. `${HOME}/checkzombies.log` oder besser `/var/log/checkzombies.log`). Abgesehen von diesem Detail ist die Manpage aber auf einem **hohen Niveau**. Sie erhöht die Professionalität des Tools erheblich, da sie dem Nutzer eine sofort verfügbare Referenz bietet. Sobald die Implementation der Optionen nachgezogen ist, wird die Manpage das Tool komplettieren. Möglicherweise könnte man langfristig auch eine englischsprachige Manpage anbieten, um eine breitere Nutzerbasis anzusprechen, aber das ist optional.

4. Repository-Struktur

Die Repository-Struktur von **checkzombies** orientiert sich bereits an bewährten Standards für kleine Systemwerkzeuge und zeigt, dass an Professionalität gedacht wurde. Wichtige Verzeichnisse und Dateien sind vorhanden:

- `bin/`: Enthält das ausführbare Haupt-Skript `checkzombies`. Die Verwendung eines `bin`-Verzeichnisses für ausführbare Skripte ist üblich und sinnvoll, um Quellcode vom Rest zu trennen.
- `man/`: Beinhaltet die Manpage im Unterordner `man1`. Die Datei `checkzombies.1` liegt dort korrekt und entspricht dem Namensschema für Section-1-Manpages.
- `tests/`: Hier finden sich (bislang spärliche) Tests, in diesem Fall ein BATS-Testfile `test_checkzombies.bats`. Dass ein Test-Ordner existiert, ist positiv und zeigt den Willen, automatisiertes Testing zu integrieren, auch wenn derzeit nur ein trivialer Test („no zombies shows clean message“) implementiert ist [38].
- `debian/`: Dieses Verzeichnis deutet auf geplante Debian-Paketierung hin. Enthalten ist zumindest ein `control`-File mit Paketinformationen (Name, Maintainer, Abhängigkeiten etc.) [12]. Andere für ein vollständiges Debian-Paket nötige Dateien (z.B. `debian/rules`, `debian/changelog`) fehlen noch im Repo – möglich, dass hier noch Arbeit aussteht oder diese bei Paketbau dynamisch erzeugt werden. Die Präsenz des Verzeichnisses an sich signalisiert aber Professionalität, da so Drittnutzer oder Distributionen das Paket leicht übernehmen könnten.
- **CI-Konfiguration**: Unter `.github/workflows/ci.yml` ist eine GitHub Actions Pipeline definiert. Diese führt beim Push/Pull Request automatische Tests (BATS) und Linting (ShellCheck) aus und baut sogar ein Debian-Paket [39]. Diese Automatisierung ist für Open-Source-Projekte heute fast unverzichtbar und hier bereits vorbildlich eingerichtet. Man könnte sie noch erweitern (z.B. Releases automatisieren), doch das Grundgerüst steht.
- **Sonstige Dateien**: Eine `Makefile` existiert im Root, welche Installations- und Paket-Build-Schritte sowie Test/Lint erleichtert [40]. Eine `LICENSE` (MIT License) ist vorhanden [41] – wichtig für die

Rechtssicherheit. Die README haben wir schon betrachtet. Alle Dateinamen sind konsistent in Kleinschreibung und weitgehend selbsterklärend.

Insgesamt ist das **Verzeichnislayout logisch und vollständig** für ein solches Projekt. Es erinnert an die Struktur vieler Unix-Tools (Code unter `bin`, Doku unter `man`, Tests, Packaging-Metadaten). Dadurch können neue Beitragende sich schneller zurechtfinden, und Integrationen (z.B. in Debian oder CI-Systeme) greifen leicht auf die standardisierten Pfade zurück.

Fehlende Elemente für ein professionelles Tool: Einige Dinge könnten ergänzt werden, um das Tool noch "produktreifer" zu machen: - Ein **systemd-Service File** (z.B. unter `contrib/` oder direkt ein `checkzombies.service`) fehlt derzeit. Da das Tool eine Watch-Funktion andeutet, wäre ein passendes Service-Unit-File nützlich, um es als Dienst laufen zu lassen (z.B. alle X Sekunden via Timer oder als dauerhaft laufender Prozess mit `--watch`). Momentan müssten Nutzer sich ein eigenes Service-File schreiben, falls sie `checkzombies` als Hintergrunddienst einsetzen wollen. - Das **Testset** ist ausbaufähig. Bisher gibt es nur einen sehr einfachen Test [38]. Für mehr Professionalität wären zusätzliche Tests ratsam, etwa um die Argument-Parsing-Logik zu prüfen, oder um sicherzustellen, dass das Skript korrekt reagiert, wenn Zombie-Prozesse vorhanden sind (das letztere lässt sich automatisiert schwer simulieren, aber man könnte via Mocking oder kleine C-Programme Zombies erzeugen, um das Skript in Aktion zu testen). - Die **Debian-Paketierung** sollte vollständig gemacht werden. Neben `debian/control` wären `debian/rules` (mit `dh_installdirs`, `dh_installman` etc.), `debian/changelog` und ggf. `debian/install` wünschenswert, sodass ein `dpkg-buildpackage` ohne manuelle Intervention ein gültiges `.deb` erzeugt. Ähnlich, wenn RPM-Pakete versprochen sind, wäre ein Spezifikationsfile (`.spec`) oder wenigstens ein Verweis auf ein Build-System für RPM hilfreich. Gegebenenfalls könnten diese auch generiert werden, aber im Repo selbst sind sie noch nicht sichtbar. - **Continuous Integration** könnte erweitert werden: z.B. Matrix-Builds für verschiedene Distributionen oder Shell-Versionen. Derzeit wird nur auf Ubuntu-Latest getestet [42]. Allerdings ist das eher ein Optimierungspunkt. - Dokumentations-technisch könnte ein **CHANGELOG** oder eine `NEWS`-Datei hinzugefügt werden, um Änderungen zwischen Releases festzuhalten. In den GitHub Releases mag das ohnehin beschrieben sein, doch ein im Repo versionierter Changelog ist oft hilfreich für Nutzer und Maintainer. - Auch ein kurzer **Beitrag-Leitfaden** (CONTRIBUTING.md) wäre für externe Kollaborateure nützlich, falls das Projekt grössere Verbreitung finden soll. Dies ist aber optional.

Abschließend lässt sich sagen, dass die jetzige Repo-Struktur bereits viele Merkmale eines professionellen Tools erfüllt. Ein paar Ergänzungen (siehe oben) würden es abrunden und die Hürde senken, das Tool in produktive Umgebungen oder Distributionen zu übernehmen.

5. Konkrete Verbesserungsvorschläge

Abschließend folgen konkrete, priorisierte Empfehlungen, um **checkzombies** von seinem aktuellen Zustand zu einem wirklich professionellen Linux-Systemkommando weiterzuentwickeln:

- **Optionen-Parsing und CLI-Verhalten:** Implementieren Sie einen robusten Argument-Parser. Nutzen Sie dafür entweder die Bash-builtin `getopts` (für Kurzoptionen) in Kombination mit GNU `getopt` (für Langoptionen), oder parsen Sie `$@` manuell per `case`. Wichtig ist, dass alle in der Manpage dokumentierten Optionen auch tatsächlich ausgewertet werden. Beispielsweise könnte ein Schema so aussehen:

```
while :; do
    case "$1" in
        -h|--help) print_help; exit 0 ;;
        -v|--version) echo "checkzombies 2.0"; exit 0 ;;
        -l|--list) mode="list"; shift ;;
        -a|--auto) mode="auto"; shift ;;
        -w|--watch) mode="watch"; shift ;;
        -f|--force) force="yes"; shift ;;
        --) shift; break ;;  # end of options
        -*) echo "Unbekannte Option: $1"; print_usage; exit 1 ;;
        *) break ;;  # no more options
    esac
done
```

Damit würden **-h/--help** und **-v/--version** korrekt behandelt (Anzeige einer Hilfe bzw. Version und Beenden). Insbesondere `--list`, `--watch` und `--auto` sollten als **exklusive Modi** fungieren, die das Verhalten des Skripts steuern:

- `--list` **(-l):** Das Skript listet Zombies im gleichen Format wie bisher (Tabellarisch) auf und endet dann ohne ins interaktive Menü zu gehen. So kann ein Benutzer oder ein anderes Programm die Ausgabe weiterverwenden (Piping nach `grep` / `less` etc.), wie in der Manpage vorgesehen [7].
- `--auto` **(-a):** Führt die automatische Bereinigung durch (alle Zombie-Elternprozesse terminieren, Dienste neu starten) und gibt anschließend vielleicht eine Zusammenfassung aus (z.B. "X Zombies bereinigt") bevor es endet. Dies entspricht dem jetzigen Verhalten bei `--auto` [29], sollte aber auch für `-a` gelten und evtl. mit `--force` kombiniert werden können.
- `--watch` **(-w):** Implementieren Sie die kontinuierliche Überwachung. Dies könnte intern so gelöst sein, dass das Skript in einer Schleife alle z.B. 10 Sekunden (laut Manpage) `find_zombies` aufruft und entweder immer die Liste ausgibt (`--list`-Stil) oder neu gefundene Zombies meldet. Hier ist es sinnvoll, Signale zu fangen (trap SIGINT/SIGTERM), um die Schleife abbrechen zu können. Alternativ, falls man das Utility nicht permanent laufen lassen will, könnte man auch auf externe Tools verweisen – doch da `-w` in der Doku steht, sollte es entweder implementiert oder aus der Doku entfernt werden.
- `--force` **(-f):** Diese Option sollte die Sicherheitsabfragen reduzieren. Konkret könnte `--force` bewirken, dass im **Auto-Modus** sofort SIGKILL statt SIGTERM gesendet wird (oder zumindest nach sehr kurzer Wartezeit), und im **Interaktiv-Modus** die Rückfrage beim individuellen Kill entfällt bzw. automatisch mit "Yes" beantwortet wird. In jedem Fall ist es wichtig, im Output deutlich zu machen, dass `--force` genutzt wurde (evtl. Warnmeldung), da diese Option als gefährlich gekennzeichnet ist.

Darüber hinaus sollten unbekannte Optionen oder falsche Kombinationen eine sinnvolle **Fehlermeldung** und einen **Usage-Hinweis** produzieren, statt still ignoriert zu werden. Wenn z.B. jemand `checkzombies --list --auto` angibt, könnte das Skript melden, dass die Modi nicht kombinierbar sind. Eine Funktion `print_usage` oder `print_help` (die entweder kurz die Optionen aufzählt oder

direkt die Manpage referenziert) wäre hier hilfreich. Insgesamt steigert ein konsistentes CLI-Verhalten mit umfassendem Optionen-Parsing die Usability enorm und verhindert Fehleinsätze.

- **Verbesserung des Shell-Skripts (Robustheit, Wartbarkeit):** Wenden Sie bewährte Shell-Scripting-Praktiken an:
- Aktivieren Sie zu Beginn `set -euo pipefail`, um stilles Scheitern zu vermeiden. Kombinieren Sie dies mit gezielten Ausnahmebehandlungen (`|| true` wo gewünscht, oder besser spezifische Checks). So fällt z.B. ein Fehler in der Pipeline von `find_zombies` sofort auf, anstatt unbemerkt zu bleiben.
- Nutzen Sie umfangreicher **Variablen-Quoting**. In Bash-Skripten ist es ratsam, nahezu alle Variablen in Anführungszeichen zu setzen, außer wenn Worttrennung ausdrücklich gewünscht ist. Im aktuellen Skript ist das schon recht gut umgesetzt, aber stellenweise (z.B. bei der Verwendung von `awk` mit `$11 $12`) könnte man überlegen, ob nicht ein anderes Vorgehen (z.B. `ps` mit einem anderen Format) sicherer wäre, um Leerzeichen in Kommandonamen/Argumenten sauber zu handhaben.
- **Logging:** Erweitern Sie die Logging-Funktionalität. Jeder relevante Schritt (z.B. „Prozess X mit SIGTERM beendet“, „Prozess Y mit SIGKILL erzwungen“, „Service Z neu gestartet“) sollte ins Log geschrieben werden. Der statische Zeitstempel pro Lauf könnte dadurch ersetzt werden, dass `log()` bei jedem Aufruf die Zeit neu ermittelt (z.B. `date "+%Y-%m-%d %H:%M:%S"` innerhalb der Funktion). Außerdem sollte der Log-Pfad konfigurierbar oder zumindest an einen sinnvolleren Ort – etwa `/var/log/checkzombies.log` – verlegt werden. Für Systemintegration wäre es ideal, statt einer eigenen Logdatei den Syslog zu nutzen (z.B. via `logger` Befehl), damit Logs je nach System direkt in `/var/log/syslog` o.ä. auftauchen.
- **Fehler- und Erfolgscodes:** Überlegen Sie, die Exit-Codes des Skripts aussagekräftiger zu gestalten. Momentan endet das Skript fast immer mit `0` (außer bei Nutzereingabefehlern) [43]. Man könnte definieren: `0` = keine Zombies gefunden oder erfolgreich bereinigt, `1` = Fehler (z.B. falsche Nutzung), `2` = Zombies gefunden aber nicht bereinigt (z.B. Benutzer hat abgebrochen). Solche Konventionen sollte man dann auch in der Manpage (EXIT STATUS Sektion) dokumentieren. Dies wäre insbesondere nützlich, wenn das Tool in Scripte oder Monitoring eingebunden wird.
- **Interaktives Menü verbessern:** Der aktuelle interaktive Modus erlaubt nur eine einzige Aktion (alle oder einen töten) und beendet dann das Skript. Benutzerfreundlicher wäre es, wenn nach einer einzelnen Bereinigung (`i`-Modus) das Menü erneut angezeigt würde, solange noch Zombies vorhanden sind, bis der Nutzer `q` wählt. Dies erfordert eine Schleife um den Menüabschnitt. Zudem könnte man im Menü einen Eintrag „r) Reload“ oder automatisches Aktualisieren erwägen, falls während langer Betrachtung neue Zombies auftauchen – allerdings ist das ein Edge-Case. Wichtig ist vor allem konsistente Bedienung: derzeit führt `q` zu Abbruch mit Ausgabe der Zombie-Anzahl [44], was in Ordnung ist.
- **Signalverarbeitung:** Falls `--watch` implementiert wird oder das Tool generell längere Zeit läuft, fügen Sie `trap`-Handler für SIGINT/SIGTERM ein, um z.B. temporäre Zustände aufzuräumen oder eine geordnete Beendigung zu protokollieren. In Watch-Modus könnte ein Trap dafür sorgen, dass beim Abbruch eine letzte Meldung („Überwachung beendet“) erscheint, statt einfach mitten in einer Ausgabe abzubrechen.

- **Code Cleanup:** Entfernen Sie nicht mehr genutzte Variablen (`zombies` war überflüssig) und achten Sie auf konsistente Nutzung von `sudo`. Gegebenenfalls prüfen Sie am Start: Wenn nicht root, vielleicht Hinweis "Einige Funktionen erfordern Root-Rechte" ausgeben.

- **Erweiterung der Tests:** Schreiben Sie zusätzliche Unit-Tests, vor allem um die neuen Parser-Funktionen und Modi zu überprüfen. Für Shell-Skripte eignet sich BATS, was ja schon eingerichtet ist. Beispielsweise könnte man testen:

- Aufruf von `checkzombies -h` gibt einen bestimmten Usage-Text zurück und Exit-Code 0.
- `checkzombies --version` gibt die erwartete Versionsnummer aus.
- `checkzombies --list` mit einem simulierten Zombie-Prozess (dieser Teil ist tricky: man könnte ein Dummy-Zombie erzeugen, indem man in einem separaten Hintergrundprozess einen Zombie kreiert – z.B. in C oder durch forken in Bash). Hier könnte man notfalls den `find_zombies()` Output simulieren durch Stubbing, falls echte Zombies im Test schwer zu erzeugen sind.
- Prüfen, dass `--auto` bei keinem Zombie den richtigen Text ausgibt ("Keine Zombie-Prozesse aktiv" [43]) und bei vorhandenen Zombies bestimmte Aktionen meldet.
- Falls möglich, testen, dass im Auto-Modus ein `kill` Aufruf tatsächlich erfolgt (hier könnte man z.B. einen harmlosen Prozess forken, ihn hängen lassen und schauen, dass er vom Skript getötet wird).

Mehr Tests erhöhen das Vertrauen in Änderungen. Insbesondere wenn Sie die Option-Parsing einführen, kann ein falsches Handling dort schwerwiegende Folgen haben (z.B. unbeabsichtigtes Löschen falscher Prozesse, falls Parsing fehlschlägt). Automatisierte Tests fangen solche Probleme früh ab. Auch die ShellCheck-Linter-Ausführung, die bereits im CI integriert ist [45], sollte beibehalten werden – beheben Sie alle Warnungen, die ShellCheck ausgibt, um die Skriptqualität weiter zu verbessern.

- **Distribution und Deployment:** Vervollständigen Sie die Paketierungs-Schritte:
- **Debian-Paket:** Ergänzen Sie die nötigen Dateien unter `debian/` (eine minimale `debian/rules` mit `dh $@`, einen `debian/changelog` Eintrag für Version 2.0 etc.). Testen Sie das Bauen des Pakets lokal oder in CI. Prüfen Sie, ob die Manpage im Paket landet (ggf. via `debian/install` an den richtigen Ort kopieren lassen). Sobald das Debian-Paket sauber gebaut werden kann, könnten Sie es auf Plattformen wie **Launchpad** oder **OpenSUSE Build Service** hochladen, um PPAs bzw. Repositories für Nutzer bereitzustellen. In der README könnte dann alternativ zur manuellen Installation ein `apt install ./checkzombies_2.0_all.deb` vorgeschlagen werden.
- **RPM-Paket:** Ähnlich sollten Sie – falls anvisiert – eine RPM-Spec Datei erstellen oder via FPM ein RPM bauen. Der Verweis in der README auf RPM-Pakete [14] weckt Erwartungen; es wäre professionell, zumindest zum Release auch ein `.rpm` bereitzustellen. Vielleicht kann der GitHub Action Workflow erweitert werden, um in einem Fedora/CentOS Container ein RPM zu erzeugen und als Artifact zu speichern.
- **Homebrew / Cross-Plattform:** Falls das Skript grundsätzlich auf anderen Unix-Systemen läuft (macOS hat allerdings kein `systemd`, aber Zombies gibt es dort auch), könnten Sie perspektivisch an Homebrew-Rezept oder ähnliches denken. Das geht über Linux-Systemkommando hinaus, ist aber ein Weg, Verbreitung zu erhöhen.

- **CI/CD:** Der aktuelle CI-Workflow führt Tests, Linting und Paketbau durch – ein guter Start [39]. Sie könnten daran anschließend einen **Release-Workflow** einrichten, der bei Git-Tags automatisiert GitHub Releases erstellt und die gebauten Pakete anhängt. So wäre der Veröffentlichungsprozess effizient und weniger fehleranfällig. Außerdem könnte man in der CI Matrix verschiedene Shells (bash 4 vs. 5) oder Distros testen, um Kompatibilität sicherzustellen. Zwar ist `/bin/bash` überall ähnlich, aber Unterschiede können existieren.

- **Benutzerfreundlichkeit und Wartbarkeit:**

- **Konsistente Ausgabe & Sprache:** Überlegen Sie, ob das Tool in Deutsch bleiben soll oder für internationale Nutzer Englisch verwenden sollte. Aktuell sind alle Ausgaben und die Doku auf Deutsch, was für deutschsprachige Admins angenehm ist. Für ein breiteres Publikum wäre Englisch Standard. Ggf. könnten Sie auch zweisprachige Doku (README in Englisch, Manpage Deutsch) oder sogar Lokalisierung in Betracht ziehen. In jedem Fall: behalten Sie Konsistenz. Momentan gibt es z.B. in Ausgaben ein Mix aus deutschen Texten ("Keine Zombie-Prozesse aktiv" [43], "Ungültig" [10]) und technischen Begriffen. Das ist okay, aber in einer späteren Version könnte man entscheiden, alles auf Englisch umzustellen.
- **Ausgabeformat und Ästhetik:** Die Verwendung von Emojis (✔, , etc.) ist ein zweischneidiges Schwert. In interaktiven Terminals können sie die Verständlichkeit erhöhen (schneller Blickfang für Erfolg/Fehler). Allerdings sind sie in manchen Umgebungen nicht sichtbar oder können bei Logging zu Sonderzeichen führen. Eine professionelle Lösung könnte sein, eine **Option** anzubieten, diese Symbole zu deaktivieren (z.B. `--no-unicode` oder automatisch bei Nicht-TTY-Ausgabe weglassen). Zusätzlich könnten farbige Hervorhebungen (mittels ANSI Escape Codes) genutzt werden, um z.B. Warnungen rot und Erfolge grün zu markieren – aber auch hier sollte man Terminal-Detektion einbauen und abschaltbar halten. Als Beispiel: `echo -e "\e[31mERROR:\e[0m ..."` würde einen roten "ERROR:" ausgeben. Solche Details verbessern die Benutzererfahrung, sollten aber sparsam und konfigurierbar eingesetzt werden.
- **Manpage & Hilfe aktuell halten:** Stellen Sie sicher, dass bei allen Codeänderungen die Manpage aktualisiert wird. Nichts frustriert Nutzer mehr als veraltete Doku. Wenn Sie z.B. Exit-Codes definieren, fügen Sie eine **"EXIT STATUS"** Sektion hinzu. Wenn neue Optionen hinzukommen (oder entfallen), passen Sie SYNOPSIS und OPTIONS an. Eventuell lohnt sich auch ein kurzer **"SEE ALSO"** Abschnitt, falls es verwandte Tools gibt (z.B. `ps`, `top`, oder Monitoring-Plugins).
- **Wartbarkeit des Codes:** Sollte der Funktionsumfang deutlich wachsen (z.B. deutlich komplexere Interaktionen oder Zustandsverwaltung beim Watch-Modus), erwägen Sie ggf. die Grenze der Bash-Skript-Schiene. Manchmal lassen sich solche Tools ab einer gewissen Komplexität besser in einer Sprache wie Python umsetzen (bessere String-Verarbeitung, Systemzugriff, testbarer Code). Das muss hier nicht sofort sein – Bash reicht für den aktuellen Umfang aus –, aber im Hinterkopf behalten: Professionell heißt auch, die richtige Werkzeugwahl für die Problemgröße. Eine Zwischenlösung könnte sein, zumindest Konfigurationsvariablen (z.B. Intervall für Watch, Logfile-Pfad, evtl. Ausnahmeliste von Prozessen, die nicht gekillt werden sollen) nach außen hin einstellbar zu machen, etwa über Umgebungsvariablen oder eine kleine Config-Datei. So etwas erhöht die Flexibilität im Betrieb.
- **Systemd-Integration abrunden:** Wenn Systemd-Service-Dateien bereitgestellt werden, könnte man das Tool auch als langfristigen Dienst nutzen. Hierfür wäre dann evtl. ein begleitendes **Timer-Unit** sinnvoll, falls man periodisch scannen will ohne dauerhaften Prozess. Dokumentieren Sie solche Einsatzzwecke (z.B. in README: "So richtet man checkzombies als systemd-Service ein").
- **Feedback einholen:** Abschließend ein weicherer Tipp – holen Sie frühzeitig Feedback von Nutzern ein (evtl. Issues auf GitHub, oder Kollegen). Oft zeigen sich in der Praxis weitere Punkte, die verbessert werden können (z.B. Wunsch nach weiterer Filterung, oder Integration in Monitoring-Tools). Eine professionelle Weiterentwicklung orientiert sich an solchen Rückmeldungen, um das Tool praxisgerecht zu optimieren.

Zusammenfassend hat **checkzombies** bereits eine sehr gute Grundlage. Die genannten Verbesserungen – insbesondere vollständiges Optionen-Handling, Fehlerbehandlung, Logging und Dokumentation – würden das Tool in Qualität und Zuverlässigkeit auf das Niveau eines vollwertigen System-Kommandos heben. Mit sauberer Paketierung und durchdachter Benutzerführung kann checkzombies dann bedenkenlos sowohl von Administratoren im Alltag verwendet, als auch in Distributionen aufgenommen werden. Die Umsetzung

dieser Empfehlungen wird dazu führen, dass das Tool nicht nur funktional überzeugt, sondern auch in Bezug auf Wartbarkeit, Sicherheit und Benutzerzufriedenheit. Viel Erfolg bei der Weiterentwicklung!

---

1 4 5 14 README.md
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/README.md

2 6 7 8 28 32 33 34 35 36 37 checkzombies.1
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/man/man1/checkzombies.1

3 9 10 15 16 17 18 19 20 21 22 23 24 25 26 27 29 30 31 44 checkzombies
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/bin/checkzombies

11 13 40 Makefile
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/Makefile

12 control
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/debian/control

38 43 test_checkzombies.bats
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/tests/test_checkzombies.bats

39 42 45 ci.yml
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/.github/workflows/ci.yml

41 LICENSE
https://github.com/161sam/checkzombies/blob/cfc4c5ae925225de08822b867d44d6b3470afda9/LICENSE