# Polarion License Management Automation Tool – Prompt for Cursor AI

## Overview

We need a robust Python program to automate and simplify the management of user licenses in **Polarion**. The program will interact with an administrator user from start to finish, guiding them through various operations related to Polarion license management. It should cover multiple use cases (querying licenses, identifying inactive users, adding/removing licenses) in a secure, foolproof manner. The solution will be used in production, so **data validation**, **error handling**, and **safety checks** are critical to prevent any mistakes. All possible checks and alerts must be included to ensure the integrity of license data.

## Data Sources and Inputs

1. **Polarion Users Database**: The program needs access to Polarion’s user database (e.g., via **pgAdmin4** or an API provided by our team). This database (table polarion.t\_user) contains user records with fields like *Polarion User ID*, *Full Name*, and *Email*. Using credentials or an API (to be provided separately by Nir), the program should fetch the list of all current users. This will help identify existing users and detect removed users.
2. **License Configuration Text**: The current Polarion license assignments are maintained in a text configuration (often copied from the Polarion administration interface). This text will be **provided by the user (pasted into the program or loaded from a file)** each time the tool runs. It contains the list of named and concurrent license slots across various Polarion license types. The program will parse and manipulate this text.

## License Configuration Format

The license configuration text is a plaintext list of license slots for different product configurations (ALM, QA, Requirements, Pro, Reviewer). Key points about this format:

* It is divided into sections by license type (ALM, QA, Requirements, Pro, Reviewer), each with **Named Users** and **Concurrent Users** subsections.
* Each license assignment line follows the pattern:
* <licenseType><User><Index>=<PolarionUserID>
* For example: namedALMUser1=admin or concurrentRequirementsUser10=eyal.
* Lines beginning with # are **comments** (indicating inactive license slots or placeholders) and should be ignored when parsing or counting licenses.
* Very high index numbers (for example, user entries numbered 5000 and above) are placeholders or inactive slots that should also be ignored for license counting purposes.
* **Example Snippet:** (for illustration of format)
* # ------------------------------- POLARION ALM --------------------------------  
  # NAMED USERS:  
  namedALMUser1=admin   
  namedALMUser2=alm   
  # namedALMUser3=john.doe (this line is commented out and ignored)   
  ...   
  # CONCURRENT USERS:  
  # concurrentALMUser1=admin (placeholder, commented out)   
  ...   
  # -------------------------------- POLARION QA --------------------------------  
  # NAMED USERS:  
  # namedQAUser1=admin (no active named QA users in this example)   
  # namedQAUser2= (placeholders)   
  ...   
  # CONCURRENT USERS:  
  concurrentQAUser1=s\_Koblenz\_CI   
  ...   
  # ----------------------------- POLARION REQUIREMENTS -----------------------------   
  # NAMED USERS:  
  namedRequirementsUser1=benjamins   
  namedRequirementsUser2=barakd   
  ...   
  namedRequirementsUser18=arielk   
  namedRequirementsUser19=evgenyra   
  # namedRequirementsUser20= (placeholder, commented)   
  ...   
  # CONCURRENT USERS:  
  concurrentRequirementsUser1=drorp   
  concurrentRequirementsUser2=zivk   
  ...   
  concurrentRequirementsUser10=eyal   
  ...   
  concurrentRequirementsUser33=ofirs   
  concurrentRequirementsUser5000=zohara (ignore index ≥ 5000)   
  concurrentRequirementsUser5001=yvilench (ignore)   
  ...   
  # -------------------------------- POLARION PRO --------------------------------  
  # NAMED USERS:  
  # namedProUser1=admin (no active named Pro users in example)   
  ...   
  # CONCURRENT USERS:  
  concurrentProUser1=dins   
  concurrentProUser2=orna   
  ...   
  concurrentProUser7=rivkak   
  # concurrentProUser8=maord (commented out)   
  concurrentProUser9=yochevedv   
  ...   
  concurrentProUser28=mustafaa   
  ...   
  # ----------------------------- POLARION REVIEWER -----------------------------   
  # NAMED USERS:  
  # namedReviewerUser1=admin (no active named Reviewer users)   
  ...   
  # CONCURRENT USERS:  
  concurrentReviewerUser1=netanelli   
  concurrentReviewerUser2=hilay   
  ...   
  concurrentReviewerUser10=libi   
  ...   
  concurrentReviewerUser28=davidb   
  concurrentReviewerUser5000=zurielt (ignore ≥ 5000)   
  concurrentReviewerUser5001=zurib (ignore)   
  ...
* **Interpretation:** In the above example, active license entries (non-comment lines with index < 5000) indicate which users (<PolarionUserID>) hold which license type and whether it’s a named or concurrent license. For instance, admin has a Named ALM license (User1), s\_Koblenz\_CI has a Concurrent QA license, benjamins has a Named Requirements license, eyal has a Concurrent Requirements license, dins has a Concurrent Pro license, netanelli has a Concurrent Reviewer license, etc.

The program should parse this entire text, **ignoring commented or placeholder lines**, to build an internal representation of license assignments. It may be useful to store this in a structured way, for example: a list or dict of license entries, or a mapping of user → license type.

## Functional Requirements

The tool should support the following primary use cases with an interactive, step-by-step approach. The user (an admin) will be prompted to choose an action and provide necessary input for that action. **All operations must be implemented with careful checks and confirmations.**

### 1. Retrieve and Load Data

**a. Fetch Active Users from Polarion:** Connect to the Polarion database (PostgreSQL) or use the provided API to retrieve all active user records. Gather at least the following details for each user:  
- **Polarion User ID** (this is the username/login, e.g., admin, benjamins, etc.)  
- **Full Name** (e.g., "John Doe")  
- **Email Address**

This will likely involve using credentials or API details provided by Nir. The program should handle connection errors or authentication issues gracefully, with clear error messages if the user data cannot be fetched.

**b. Parse License Configuration:** Take the input license configuration text (pasted by the user or loaded from a file) and parse it. The output of parsing should be a structured representation, for example:  
- A list of all active license entries, each containing: license category (ALM/QA/Requirements/Pro/Reviewer), license type (Named/Concurrent), index number, and the associated Polarion User ID.  
- Alternatively, a dictionary where keys are Polarion User IDs and values are one or more license assignments. (Since ideally each user should have only one active license type, each user would map to at most one license entry. However, the program should detect if any user appears in multiple license categories, which might indicate a configuration anomaly to flag.)

While parsing, the program should:  
- Ignore any line that is commented out (starts with #).  
- Ignore any license entries with an index of 5000 or above (placeholders).  
- Trim whitespace and handle empty or placeholder values (e.g., lines like namedQAUser2= with no user after = can be considered empty slots and ignored).

If the input text format is not exactly as expected, the program should handle it or alert the user if it’s unable to parse certain lines.

**c. Build Combined User License Table:** Using the data from (a) and (b), the program should create an internal table (or data structure) that combines user information with license information. Each active Polarion user should be represented with their full name, email, and any license they possess. Likewise, any user present in the license file but not found in the user list should also be noted (this likely indicates an ex-employee or deactivated account still holding a license slot).

* For each license entry from (b), find the corresponding user in the user list from (a). Compile a record with: **Polarion User ID**, **Full Name**, **Email**, and **License Type** (e.g., "Named ALM", "Concurrent QA", "Named Requirements", etc.).
* If a license entry’s user is **not** found in the active user list, record this as a “stale license” (user with a license but no longer with the company). These will be handled in use case 2.
* Conversely, it’s possible (though uncommon) that a user exists in the company with no license at all. Those users wouldn’t appear in the license text. We might not specifically need to list all such users, but the program could keep track if needed (perhaps for audit purposes).

### 2. **Use Case 1:** Query License Assignments for Specific Users

**Description:** The user may provide a list of one or more individuals (identified by full name, email, or Polarion user ID) and wants to know if those users currently have a Polarion license, and if so, what type. This corresponds to incoming Jira requests where someone asks to add certain people – first, we might want to check if they already have a license.

**Functionality:**  
- Prompt the admin to input one or multiple user identifiers. The program should accept identifiers in a flexible way (for example, the admin could input a comma-separated list of emails, or a list of full names, or usernames). The program should then attempt to match each identifier against the user list.  
- For each user in the input list, find the matching **Polarion User ID** (if the admin entered an email or full name, use the user database to find the corresponding user ID). If no match is found, report that the person is not found in the system (and thus has no license by definition).  
- If the user is found, check the combined license table to see if that User ID has an assigned license.  
- If yes, report **which license type** they have (e.g., "John Doe (jdoe@example.com) has a Named Requirements license").  
- If not, report that the user currently does not hold any Polarion license.  
- The program should be case-insensitive and robust in matching names or emails (e.g., handle minor differences or alert if ambiguous). For example, if the admin types "John Doe" but there are multiple "John Doe", or if a full name could match multiple users, the program should clarify by perhaps showing a list of matches or asking for more info (to avoid misidentification).  
- This query operation is read-only; it should **not** modify any data. It’s purely informational.

**Output for Use Case 1:** A clear report (could be printed to console or as formatted text) for each requested person, showing their license status. For example:

- John Doe (jdoe@company.com) – \*\*Has Named Requirements License\*\*   
- Jane Smith (jsmith@company.com) – \*\*No license assigned\*\* (not currently in any license category)   
- alice@example.com – \*\*User not found in Polarion\*\* (no such account)

This gives the admin quick insight into who already has access and who doesn’t.

### 3. **Use Case 2:** Identify and List Inactive Users Holding Licenses

**Description:** Over time, some users leave the company or are deactivated in Active Directory. When this happens, their Polarion account is typically removed (no longer present in the t\_user table), but they might still have an entry in the license configuration file. Such entries consume license slots unnecessarily. This use case finds all such instances.

**Functionality:**  
- Using the combined data from step 1c, filter for any license entries where the user was not found in the active users list. These represent **stale license assignments**.  
- For each such entry, gather details: the Polarion User ID, the license type/category they were assigned, and possibly the index number or line from the config for reference. (Full name and email would not be available via the user list, since they have no active account — unless we have historical data, which we likely do not, so just the username.)  
- Present a report to the admin listing all these inactive users with licenses. For example:

Inactive Users with Assigned Licenses:  
- alice.jones (Polarion user id `alice.jones`) – was assigned a Concurrent QA license   
- bob.smith (Polarion user id `bob.smith`) – was assigned a Named ALM license

(Above is just an example format; include whatever info is available. If full name is known from older data or could be parsed from the user ID, fine, but usually we’ll just have the username in this case.)  
- This list lets the admin know which license entries likely need removal. The program might also prompt: “Do you want to remove these inactive users from the license file?” as a follow-up action (since cleaning them up is often the next step). However, the actual removal would be handled via Use Case 3 (the admin might choose to remove them in that step).

**Output for Use Case 2:** A list of users who appear in the license file but not in the active user database, as shown above. This serves as an audit of licenses that can be freed up.

### 4. **Use Case 3:** Modify User Licenses (Add/Remove Licenses)

**Description:** This is the most critical and sensitive operation. It allows the admin to make changes to the license assignments — adding new users to licenses or removing users — and outputs an updated license configuration text. All changes must be validated to avoid mistakes, since an incorrect license configuration could lock users out or violate license limits. The program should interactively guide the admin through the changes and confirm the final outcome.

There are two sub-operations here: **Add license for user(s)** and **Remove license for user(s)** (potentially also "transfer license" which is effectively a combination of removal and addition for switching a user’s license type). We should handle each carefully:

#### 4.1 Adding a New License Assignment

**Scenario:** One or more users need to be given a Polarion license (for example, new employees or existing employees changing roles). The request might specify them by name or email, and possibly the desired license type (if known).

**Functionality:**  
- **Input:** Prompt the admin for the list of user(s) to add, and for each, the license type/category to assign. The program should make this as straightforward as possible, e.g.:  
- If only one user is being added, ask: “Enter the user's identifier (name/email/ID) to add, and the license type (ALM/QA/Requirements/Pro/Reviewer).”  
- If multiple users, perhaps allow inputting multiple entries. Alternatively, handle one user at a time in a loop: ask for user identifier, then ask for license type, then confirm, then next user, etc.  
- **User Identification:** For each provided identifier (name or email), find the corresponding Polarion user ID from the user list (similar to Use Case 1). If the user is not found in the user database:  
- **Alert:** The program should warn “User X not found in Polarion user list. They may not have an account. You cannot assign a license to a non-existent user.” and skip that user or prompt if the admin wants to proceed anyway. (In most cases, the admin should create the user account in Polarion first, but our tool will at least warn if the user doesn’t exist.)  
- **Check Existing Licenses:** If the user **already has a license** in the current config (found in our parsed data), we should **not add a duplicate**.  
- The program should inform the admin: “User X already has a license (Named QA, for example). No new license added.” Or, if the intention was to switch license type, the program should handle that as a removal + addition (see license switch below). But by default, prevent adding the same user twice.  
- **Determine License Slot:** Find the appropriate section and slot for the new license in the license text:  
- Identify whether it’s a Named or Concurrent license for the given category (the admin should specify which, or the program can ask). For example, the admin might specify "ALM named" or "Requirements concurrent" for the new user. If not explicitly specified, the program can clarify: e.g., “For user X, should this be a Named or Concurrent license for the <category> product?”  
- Within the license file structure, find the next available slot for that category and type. This could be: - The lowest-numbered unused index (or a commented-out placeholder) that can be reassigned. For example, if namedALMUser3 is commented out or unused, that slot can be reused for a new user. If no placeholder is available, use the next consecutive number after the highest existing one.  
- Ensure numbering continuity if required by Polarion (Polarion might not require continuous numbering, but keeping it tidy is good).  
- Example: If adding a Named Requirements user and currently namedRequirementsUser1...19 are used, with 20 commented out (available), assign the new user to namedRequirementsUser20. If none was free but license allows more, assign namedRequirementsUser20 as a new line.  
- **License Type Switch (if applicable):** If the admin intends to **change a user’s license type** (e.g., user currently has a Reviewer license, and now needs a Requirements license), handle it as follows:  
- **Comment out (deactivate) the old license entry** for that user. (Instead of outright deleting the line, prefix it with # to keep it as a record in the file but make it inactive.) For example, if concurrentReviewerUser15=johnsmith exists and we are giving John Smith a Requirements license, then change that line to # concurrentReviewerUser15=johnsmith in the output.  
- Then add the new license entry for the user in the target category (e.g., concurrentRequirementsUserX=johnsmith in the Requirements section).  
- The program should clearly communicate this to the admin (e.g., "Switching John Smith from Reviewer to Requirements license: the old Reviewer entry will be commented out, and a new Requirements entry will be added.").  
- **Confirmation:** Before applying the addition (especially if multiple users are being processed), the program should display the planned changes for review. For example:

Ready to add the following license assignments:  
- John Smith -> Named Requirements  
- Alice Brown -> Concurrent ALM  
- (etc.)

If any user had an existing license that will be switched, list that as well:

(John Smith’s previous license “Concurrent Reviewer” will be removed)

Prompt the admin to confirm (yes/no) before actually modifying the license data. This extra confirmation step helps prevent mistakes.  
- **Apply Changes:** Once confirmed, update the internal representation of the license assignments by adding the new entries (and commenting out any old entries in case of switches). The program should handle the text manipulation carefully to preserve formatting. Specifically:  
- Insert the new line in the correct place in the license text (under the right section and subsection). Ideally, maintain the sorted order by index. If reusing a commented slot, just replace the # at line start with nothing and put the user’s ID. If appending a new slot at end of section, ensure it's placed before the section’s concluding line of dashes or before the next section header.  
- For commenting out removals, prepend a # to the line. Maybe also add a note in the comment (like # concurrentReviewerUser15=johnsmith (removed) – optional, but could be useful for clarity).  
- After applying changes in memory, proceed to output (see Output section below).

#### 4.2 Removing an Existing License Assignment

**Scenario:** One or more users should be removed from the license file (e.g., when someone leaves the company or no longer needs access). In many cases, Use Case 2 will highlight such users.

**Functionality:**  
- **Input:** Prompt for the list of user(s) to remove. The admin can provide identifiers (usernames, full names, or emails). If multiple, handle iteratively or as a batch.  
- **User Identification:** Similar to adding, resolve each identifier to a Polarion user ID if possible. If a name/email is given and not found in the user list, still allow the removal if that username exists in the license file. (For example, if the person already left, they won’t be in the active user DB, but their username might still be in the license file – we still need to remove it. So the program should be able to handle removal of users not in the active DB by relying on the license data parsing.)  
- **Check License Presence:** For each user, verify that they indeed have a license entry in the current config.  
- If they do, prepare to remove it.  
- If they do not (no such username in any active license line), warn the admin: “User X has no license assignment, so it cannot be removed.” and skip that user.  
- **Removal Method:** To "remove" a license entry, the program should **comment out that line** in the license text (rather than deleting it outright). This is to maintain the structure and have a record of the removal. For example, namedALMUser5=johnsmith would be turned into # namedALMUser5=johnsmith in the output.  
- If the removed user was the only occupant of the highest index, we might not strictly need to keep a commented line, but it’s safer to just comment it for consistency.  
- **Confirmation:** Similar to adding, list the entries that will be removed and ask the admin to confirm. For example:

Ready to remove the following license assignments:  
- John Smith – Named ALM  
- Alice Brown – Concurrent QA

If any of these users are not found or have no license, mention those as well and exclude them from the action.  
- **Apply Changes:** Once confirmed, modify the internal license data (mark those entries as inactive by adding #). If desired, also append a note or timestamp in the comment for future reference (optional).

#### 4.3 Post-Modification Updates

After adding or removing, the program’s internal representation of license data is updated. It may be wise to re-generate the combined user license table (from step 1c) to reflect the new state, especially if multiple operations are done in one run. This allows subsequent actions in the same session to work off the updated data.

For example, if the admin is doing multiple changes in one go (like adding some users and removing others before finalizing), the program should keep track of all pending changes. Possibly implement a mechanism to do multiple operations and then output one final updated file (to avoid repeatedly pasting intermediate files).

However, to keep things simpler, the tool can handle one batch of additions/removals at a time, update the config, output it, and if further changes are needed, the admin can run it again or choose another menu option.

### 5. User Interaction Flow

The program should guide the admin through a clear interactive menu or prompt sequence. One possible flow:

1. **Main Menu:** When the program starts, present options such as:
2. Query user license status (Use Case 1)
3. List inactive (removed) users with licenses (Use Case 2)
4. Add user(s) to license (Use Case 3 - Add)
5. Remove user(s) from license (Use Case 3 - Remove)
6. Exit

The menu can be a simple text menu where the admin enters a number to choose an option. Always include an option to cancel or go back if needed.

1. **Option Execution:** Based on the choice:
2. If 1: Prompt for user identifiers (could allow multiple separated by commas or newlines). Perform the query, display results. After completion, perhaps return to main menu.
3. If 2: Immediately fetch and display the inactive users list (from Use Case 2). After showing the list, possibly prompt “Do you want to remove these users from the license file? (y/n)” as a convenience. If yes, proceed to remove them (confirm again if needed); if no, just go back to menu.
4. If 3: Prompt for details to add licenses (as described in 4.1). Possibly allow entering multiple users: the program could ask “How many users do you want to add?” and loop, or read a multiline input. Confirm and apply changes, then output the updated license text and a summary of changes.
5. If 4: Prompt for details to remove licenses (as in 4.2). Possibly offer a choice to remove the list from Use Case 2 (inactive users) or specify manually. Confirm and apply, output updated text and summary.
6. If 5: Exit the program.
7. **Continuous Interaction:** After completing an action (except exit), the program could return to the main menu to allow more operations in the same session. Alternatively, after a major change (like add/remove), it might exit after outputting results, to ensure the admin reviews the changes and uses the new license file. This design decision can be made based on user preference. (If it's equally fine to do multiple changes in one run, keep the menu loop; if not, possibly end after output so the user can take that output and apply it to Polarion, then run tool again for more changes later.)

Throughout the interaction, use **clear prompts and messages**. For example:  
- When asking for input, specify the format (e.g., "Enter user emails separated by commas:").  
- When something is not found or an action can’t be done, explain why.  
- Before making changes, summarize what will happen and ask for confirmation.  
- After changes, indicate success (e.g., "User X added to Requirements license.") or any issues.

## Output Requirements

After performing the requested operations (especially the modifications in Use Case 3), the program must output two important things:

1. **Updated License Configuration Text:** The program should print or save the entire license configuration content with the applied changes. This output should preserve the original format as much as possible (section headers, commented lines, etc.), just with the additions/removals done. The admin will take this text and replace the contents of the Polarion license configuration with it. So it needs to be correct and ready to paste.
2. Ensure the format exactly matches what Polarion expects. For instance, maintain the "namedALMUserX=" and "concurrent...UserX=" patterns, the section separators (the lines of dashes and headings like "POLARION ALM"), etc.
3. Comments should remain in place (except where we uncomment a slot or comment out a removed user).
4. It might be wise to keep a consistent ordering by index within each section. If we re-used a commented slot, it stays in place. If we added a new highest number, it should appear in sequence.
5. Double-check that for any category, the count of "namedXUser" lines does not exceed license limits if any (license limits might not be coded in the file, but just in case the number of slots matters). Probably not in scope to know license count limits from the file alone, but our program is mainly text editing, so trust the admin to know not to exceed their purchased licenses.
6. **Change Summary (Audit Report):** Additionally, the program should produce a summary of what changed for the admin’s reference, ideally in a clear, tabular or list format. For example:
7. **Added Users:** List any new license assignments added, with user name, license type, and the line added (e.g., "Added namedRequirementsUser20=johnsmith for John Smith (John Smith – Named Requirements license)").
8. **Removed Users:** List any licenses that were removed/commented out, with the user and the line affected (e.g., "Removed concurrentReviewerUser15=johnsmith (John Smith’s Reviewer license) – line now commented out").
9. If a user’s license was switched from one type to another, show both actions: the removal (commenting out old) and addition (new line) as part of the summary.
10. If any requested additions/removals were skipped due to issues (user not found, user already had license, etc.), mention those as well (e.g., "Skipped adding Alice Brown – user not found in database", or "Skipped adding Bob – already has a license").

This summary can be printed to the console for review or optionally saved to a log or CSV for record-keeping. The user mentioned an "Excel comparison line by line" at the end; we can interpret this as they want to easily see differences. We can’t directly create an Excel in this context, but we can present differences line-by-line. Another approach is to output a unified diff-like format of the license text before vs after. However, a simple human-readable report of changes as described should suffice, and they can copy that into Excel if needed.

**Example Change Summary Output:**

Changes applied to license configuration:  
- Added Named Requirements license for John Smith (added line: namedRequirementsUser20=johnsmith)  
- Removed Concurrent Reviewer license for John Smith (line concurrentReviewerUser15=johnsmith is now commented out)  
- Added Concurrent ALM license for Alice Brown (added line: concurrentALMUser2=aliceb)  
- Removed Named ALM license for Bob Green (line namedALMUser5=bobg is now commented out)

*(The above is an illustration; actual indices and categories would match the real scenario.)*

This gives a concise overview of what was changed, which the admin can cross-verify.

## Validation and Safety Checks

Because license management is sensitive, the program must include comprehensive validation and confirmation at each step. Here is a summary of key checks and safe practices to implement:

* **User Existence Check:** Whenever a user identifier is provided for addition or removal, verify if that user exists in the Polarion user list. If adding and the user doesn’t exist, do not proceed (or require an extra confirmation acknowledging the risk). If removing and the user is not found in the DB, that’s okay as long as they exist in the license file (since they might have left) – in this case just warn that the user is not in active list but will be removed from licenses.
* **Duplicate License Check:** Prevent adding a user who already has a license. Instead, alert the admin that the user already has license X. If the intention is to switch license types, handle via the switch process (remove old then add new) with proper messaging.
* **License Conflict Check:** Ensure a single user isn’t assigned to multiple license categories simultaneously in the final output. If our operations would result in a duplicate (e.g., if due to an oversight a user ends up in two places), detect and resolve it (this situation should be avoided by design, but double-check after modifications).
* **Index/Slot Management:** When adding, be careful to not accidentally overwrite an existing assignment. Use a free slot or next number. When removing (commenting out), ensure you don’t comment out a line that’s already commented (to avoid adding multiple # signs accidentally, perhaps strip any existing # first or check status).
* **Error Handling:** Wrap database calls in try/except and provide meaningful messages if the user data cannot be fetched. If the input license text is missing expected sections or otherwise malformed, handle gracefully (e.g., if a section is missing because no licenses of that type are present, that’s fine – the program can add the section if needed when adding a first user of that type, using the proper header format).
* **Security Considerations:** This tool is likely run by a trusted admin in a secure environment, but still:
* Do not log sensitive info like database credentials.
* If writing output to files, be mindful of file permissions.
* Validate inputs (if any special characters in user input that could break parsing, handle or sanitize them – e.g., newlines in names, etc., though not likely).
* **Confirmation Prompts:** Always ask for confirmation before making changes to the license data. Especially if multiple changes are queued, display all and confirm. Provide an option to cancel at that point, which should safely abort without making partial changes.
* **Backup:** (Optional but very useful) – before applying modifications, the program could save a backup of the original license text (in memory or to a file) so that if something goes wrong or the admin wants to revert, they have the original. This could even be part of the output (e.g., the diff or summary serves as a record to manually undo if needed).

By incorporating all these checks, the tool will minimize the risk of errors in the license file. The goal is that the admin can confidently use this automation in production, knowing that it won’t, for example, drop a user’s license unintentionally or assign a license incorrectly.

## Conclusion

This Python program will streamline Polarion license management by providing an interactive, safe interface for common tasks: checking user license status, detecting obsolete assignments, and updating license entries. It will integrate with the Polarion user database to ensure accuracy, and produce an updated license configuration text that can be directly applied. The prompt above should guide Cursor AI (or any code assistant) to generate a comprehensive solution with all the necessary functionality and safety measures. By following these specifications, the resulting program will significantly reduce manual effort and error rate in managing Polarion licenses, while being robust enough for production use.