Multi-Agent Coding Assistant

Enter a software requirement below to see how the multi-agent system analyzes, implements, reviews, documents, tests, and prepares deployment artifacts.

Software Requirement

Describe the software you want to build

Create a Python program that manages a simple in-memory library system.
The system should allow adding books, listing all books, searching for a book by title, and removing a book by title.
Each book should have a title and an author.
Book titles must be unique.
Invalid operations should return clear error messages.

Run Multi-Agent System

Workflow completed successfully.

Refined Requirement

System purpose:

- Provide an in-memory Python library manager that stores books (title + author) and exposes operations to add, list, search, and remove books by title.

Supported operations (each implemented as a Python method of a single Library class):

- Library.add_book(title: str, author: str) → None

- Library.list_books() → list[dict]
- Library.find_book_by_title(title: str) → dict
- Library.remove_book_by_title(title: str) → dict

For all operations, common invariants and internal representation:

- The Library is an in-memory object that holds all books only for the process lifetime.
- Each book is represented externally as a dict with exactly two keys: "title" (str) and "author" (str).
- Titles are unique in the library using a case-insensitive uniqueness rule: two titles that are equal when normalized by stripping whitespace and lowercasing are considered identical.
- All string inputs are normalized by trimming leading and trailing whitespace before validation and comparison.

Operation details (inputs, outputs, and error handling):

1. Library.add_book(title: str, author: str) → None

- Inputs:
  - title: str; required; after trimming whitespace it must be a non-empty string.
  - author: str; required; after trimming whitespace it must be a non-empty string.
- Output:
  - Returns None on success.
- Error conditions and handling:
  - If title or author is not of type str, raise TypeError with message: "title and author must be strings".
  - If trimmed title is empty, raise ValueError with message: "title must be a non-empty string".
  - If trimmed author is empty, raise ValueError with message: "author must be a non-empty string".
  - If a book with the same normalized title already exists, raise ValueError with message: "book with title '<original_input_trimmed>' already exists" (use the trimmed title from the input in the message).
- Side effects:
  - On success, store the book in the library so subsequent operations see it.

2. Library.list_books() → list[dict]

- Inputs: none.
- Output:
  - Returns a list of book dicts, each with keys "title" and "author" (both strings) representing the stored books.

- The returned list must be sorted by title in ascending order using case-insensitive comparison of the trimmed titles to provide deterministic output.
- Error conditions and handling:
  - This operation does not raise errors under normal in-memory operation.

3. Library.find_book_by_title(title: str) → dict

- Inputs:
  - title: str; required; after trimming whitespace it must be a non-empty string.
- Output:
  - Returns a book dict with keys "title" and "author" for the matching book. The returned "title" and "author" values must match the stored values (preserve original casing as provided at add time, except for trimmed whitespace removed at input time).
- Error conditions and handling:
  - If title is not of type str, raise TypeError with message: "title must be a string".
  - If trimmed title is empty, raise ValueError with message: "title must be a non-empty string".
  - If no book with the same normalized title exists, raise LookupError with message: "book with title '<original_input_trimmed>' not found".
- Matching behavior:
  - Matching uses the same case-insensitive normalized-title rule as add_book (trim + lowercase) and requires an exact normalized match (no substring or fuzzy matching).

4. Library.remove_book_by_title(title: str) → dict

- Inputs:
  - title: str; required; after trimming whitespace it must be a non-empty string.
- Output:
  - Returns the removed book dict with keys "title" and "author" (values as stored, with original casing except for trimmed whitespace removed at add time).
- Error conditions and handling:
  - If title is not of type str, raise TypeError with message: "title must be a string".
  - If trimmed title is empty, raise ValueError with message: "title must be a non-empty string".
  - If no book with the same normalized title exists, raise LookupError with message: "book with title '<original_input_trimmed>' not found".
- Side effects:
  - On success, remove the book from the in-memory store so subsequent operations do not see it.

Assumptions and clarifications (explicit):

- Title uniqueness and lookups use case-insensitive matching after trimming whitespace. This means titles that differ only by case or leading/trailing whitespace are considered identical.
- The library stores only one record per title (no support for multiple copies or editions).
- The system preserves the original title and author character casing as provided when the book was added, except that leading/trailing whitespace is removed at input time.
- All errors are signaled by raising the specified Python exception types with the exact messages described above. Caller code may catch these exceptions to obtain the clear error messages.

Non-goals (explicit):

- No persistent storage (no file, database, or external storage). Data is retained only in memory while the Library instance exists.
- No concurrency control or thread-safety guarantees are provided.
- No partial title matching, fuzzy search, filtering, pagination, sorting options, or metadata beyond title and author.
- No user interface, input parsing, or command-line behavior; the API is intended for direct use from Python code.

Minimal reproducible API summary (to implement exactly):

- class Library:
  - def add_book(self, title: str, author: str) → None
  - def list_books(self) → list[dict]
  - def find_book_by_title(self, title: str) → dict
  - def remove_book_by_title(self, title: str) → dict

All operations above are mandatory and must behave exactly as specified.

Code Review Result

Code approved after 0 review attempt(s).

Generated Python Code

```python
#!/usr/bin/env python3
"""
In-memory Library manager module.

Provides a Library class with methods to add, list, find, and remove books by title
Each book is represented as a dict with exactly two keys: "title" and "author".
"""

from typing import Dict, List


class Library:
    """
    In-memory library that stores books (title and author) for the process lifetime

    Titles are unique using a case-insensitive rule after trimming whitespace.
    """

    def __init__(self) -> None:
        """Create a new, empty Library instance."""
        # Internal mapping from normalized title -> book dict with keys "title" and
        self._books_by_norm: Dict[str, Dict[str, str]] = {}

    def _normalize_title(self, title: str) -> str:
        """Normalize a title by trimming whitespace and lowercasing."""
        return title.strip().lower()

    def add_book(self, title: str, author: str) -> None:
        """
        Add a book to the library.

        Args:
            title: Title string (required). Trimmed; must be non-empty.
            author: Author string (required). Trimmed; must be non-empty.

        Raises:
            TypeError: if title or author is not a str (message: "title and author
            ValueError: if trimmed title is empty (message: "title must be a non-em
            ValueError: if trimmed author is empty (message: "author must be a non-
            ValueError: if a book with the same normalized title already exists
                        (message: "book with title '<original_input_trimmed>' alrea
        """
        if not isinstance(title, str) or not isinstance(author, str):
```

```
            raise TypeError("title and author must be strings")

        trimmed_title = title.strip()
        trimmed_author = author.strip()

        if trimmed_title == "":
            raise ValueError("title must be a non-empty string")
        if trimmed_author == "":
            raise ValueError("author must be a non-empty string")

        norm = trimmed_title.lower()
        if norm in self._books_by_norm:
            raise ValueError(f"book with title '{trimmed_title}' already exists")

        # Store the book with trimmed title and author, preserving original casing
        self._books_by_norm[norm] = {"title": trimmed_title, "author": trimmed_auth

    def list_books(self) -> List[Dict[str, str]]:
        """
        List all books stored in the library.

        Returns:
            A list of book dicts (each with keys "title" and "author"), sorted by t
            ascending using case-insensitive comparison of trimmed titles.
        """
        books = list(self._books_by_norm.values())
        # Sort deterministically by trimmed, case-insensitive title
        sorted_books = sorted(books, key=lambda b: b["title"].strip().lower())
        # Return shallow copies to avoid accidental external mutation
        return [dict(b) for b in sorted_books]

    def find_book_by_title(self, title: str) -> Dict[str, str]:
        """
        Find and return a book by title.

        Args:
            title: Title string to search for (required). Trimmed; must be non-empt

        Returns:
            The matching book dict with keys "title" and "author" (values as stored

        Raises:
            TypeError: if title is not a str (message: "title must be a string").
```

```
            ValueError: if trimmed title is empty (message: "title must be a non-er
            LookupError: if no matching book is found
                        (message: "book with title '<original_input_trimmed>' not
        """
        if not isinstance(title, str):
            raise TypeError("title must be a string")

        trimmed_title = title.strip()
        if trimmed_title == "":
            raise ValueError("title must be a non-empty string")

        norm = trimmed_title.lower()
        if norm not in self._books_by_norm:
            raise LookupError(f"book with title '{trimmed_title}' not found")

        return dict(self._books_by_norm[norm])

    def remove_book_by_title(self, title: str) -> Dict[str, str]:
        """
        Remove a book by title and return the removed book.

        Args:
            title: Title string of the book to remove (required). Trimmed; must be

        Returns:
            The removed book dict with keys "title" and "author" (values as stored)

        Raises:
            TypeError: if title is not a str (message: "title must be a string").
            ValueError: if trimmed title is empty (message: "title must be a non-er
            LookupError: if no matching book is found
                        (message: "book with title '<original_input_trimmed>' not
        """
        if not isinstance(title, str):
            raise TypeError("title must be a string")

        trimmed_title = title.strip()
        if trimmed_title == "":
            raise ValueError("title must be a non-empty string")

        norm = trimmed_title.lower()
        if norm not in self._books_by_norm:
            raise LookupError(f"book with title '{trimmed_title}' not found")
```

```
        removed = self._books_by_norm.pop(norm)
        return dict(removed)


if __name__ == "__main__":
    # Minimal safe execution entry point; no actions performed.
    pass
```

Documentation

Library — in-memory Python library manager

Overview

Library is a simple in-memory manager for books (title + author). It provides methods to add, list, find, and remove books by title. Books are stored only for the process lifetime and each book is represented externally as a dict with keys "title" and "author".

Architecture / Design

- Main class: `Library`
  - Responsibility: validate inputs, enforce title uniqueness, store books in memory, and provide retrieval/removal operations.
- Internal representation:
  - `self._books_by_norm: Dict[str, Dict[str, str]]` — maps a normalized title (trimmed + lowercased) to the stored book dict `{"title": <trimmed title>, "author": <trimmed author>}`.
  - Titles are normalized for keying and comparisons; stored values preserve original casing except that input strings are trimmed of leading/trailing whitespace before storage.
- Data flow:
  1. Caller supplies inputs (strings).
  2. Methods validate types and trim strings.
  3. Title is normalized for lookup/uniqueness (trim + lowercase).

4. For add: if unique, the book dict is stored at the normalized key.
5. For find/remove/list: normalized title is used to lookup; returned book dicts are shallow copies of stored dicts to avoid external mutation.

- Deterministic listing:
  - `list_books()` returns books sorted by title in ascending order using case-insensitive comparison of trimmed titles.

Public API

Class: `Library`

- Constructor:
  - `Library() -> Library`
    - Create an empty in-memory library.
- Methods:

1. `add_book(self, title: str, author: str) -> None`
   - Purpose: Add a new book to the library.
   - Inputs:
     - `title`: required `str`. Will be trimmed (leading/trailing whitespace removed) before validation and storage.
     - `author`: required `str`. Will be trimmed before validation and storage.
   - Output: `None` on success.
   - Errors:
     - `TypeError("title and author must be strings")`
       - Raised if `title` or `author` is not of type `str`.
     - `ValueError("title must be a non-empty string")`
       - Raised if trimmed `title` is empty.
     - `ValueError("author must be a non-empty string")`
       - Raised if trimmed `author` is empty.
     - `ValueError("book with title '<trimmed_title>' already exists")`
       - Raised if a book with the same normalized title already exists. The message uses the trimmed title from the input.

- Side effects:
  - Stores `{"title": <trimmed_title>, "author": <trimmed_author>}` under the normalized title key so subsequent calls see it.

2. `list_books(self) -> list[dict]`
   - Purpose: Return all stored books.
   - Inputs: none.
   - Output:
     - `list` of book dicts, each with exactly keys `"title"` and `"author"`, where values are the stored strings (trimmed at add time; original casing otherwise).
     - The list is sorted by title ascending using case-insensitive comparison of trimmed titles.
     - Each returned dict is a shallow copy of the stored dict.
   - Errors: None under normal in-memory operation.
3. `find_book_by_title(self, title: str) -> dict`
   - Purpose: Look up and return a stored book by title.
   - Inputs:
     - `title`: required `str`. Trimmed before validation and normalized (trim + lowercase) for lookup.
   - Output:
     - The matching book dict with keys `"title"` and `"author"` (values as stored).
     - Returned dict is a shallow copy of the stored dict.
   - Errors:
     - `TypeError("title must be a string")`
       - If `title` is not a `str`.
     - `ValueError("title must be a non-empty string")`
       - If trimmed `title` is empty.
     - `LookupError("book with title '<trimmed_title>' not found")`
       - If no book exists with the same normalized title.
4. `remove_book_by_title(self, title: str) -> dict`
   - Purpose: Remove a book by title and return the removed record.
   - Inputs:
     - `title`: required `str`. Trimmed before validation and normalized for lookup.
   - Output:

    - The removed book dict with keys `"title"` and `"author"` (values as stored). Returned as a shallow copy.
  - Errors:
    - `TypeError("title must be a string")`
      - If `title` is not a `str`.
    - `ValueError("title must be a non-empty string")`
      - If trimmed `title` is empty.
    - `LookupError("book with title '<trimmed_title>' not found")`
      - If no book exists with the same normalized title.
  - Side effects:
    - Removes the book from the in-memory store on success.

Notes on normalization and matching:

- Normalization rule used for uniqueness and lookup: trimmed title then lowercased (i.e., title.strip().lower()).
- Titles that differ only by case or by leading/trailing whitespace are considered identical.
- Matching requires exact normalized equality (no substring/fuzzy matching).

Usage Example

```
from your_module import Library  # replace with actual module name

lib = Library()

# Add books
lib.add_book("  The Hobbit  ", "J.R.R. Tolkien")
lib.add_book("1984", "George Orwell")

# List books (sorted case-insensitively by trimmed title)
books = lib.list_books()
# Example result:
# [
#   {"title": "1984", "author": "George Orwell"},
#   {"title": "The Hobbit", "author": "J.R.R. Tolkien"}
# ]

# Find a book (lookup is case-insensitive and trims input)
```

```
found = lib.find_book_by_title(" the hobbit ")
# found == {"title": "The Hobbit", "author": "J.R.R. Tolkien"}

# Remove a book
removed = lib.remove_book_by_title("1984")
# removed == {"title": "1984", "author": "George Orwell"}
```

Assumptions and Limitations

- In-memory only: no persistence; data exists only for the Library instance lifetime.
- Single record per title: no support for multiple copies or editions.
- No concurrency/thread-safety guarantees.
- Only exact normalized-title matching (trim + lowercase) is supported; no partial or fuzzy search.
- Only two fields per book: `"title"` and `"author"`.
- All error signalling uses Python exceptions with the exact types and messages specified in the API above. Caller code may catch these exceptions to examine error conditions.

Generated Test Cases

```
import pytest
from app import Library


def test_add_book_success_trims_input_and_preserves_casing():
    lib = Library()
    result = lib.add_book("  The Hobbit  ", " J.R.R. Tolkien ")
    assert result is None
    books = lib.list_books()
    assert isinstance(books, list)
    assert len(books) == 1
    assert books[0] == {"title": "The Hobbit", "author": "J.R.R. Tolkien"}


def test_add_book_type_errors_for_non_string_inputs():
    lib = Library()
    with pytest.raises(TypeError, match=r"^title and author must be strings$"):
        lib.add_book(123, "Author")
    with pytest.raises(TypeError, match=r"^title and author must be strings$"):
```

```
        lib.add_book("Title", None)


def test_add_book_empty_title_or_author_raises_value_error():
    lib = Library()
    with pytest.raises(ValueError, match=r"^title must be a non-empty string$"):
        lib.add_book("   ", "Author")
    with pytest.raises(ValueError, match=r"^author must be a non-empty string$"):
        lib.add_book("Title", "   ")


def test_add_book_duplicate_title_case_insensitive_raises_value_error():
    lib = Library()
    lib.add_book("My Book", "Author One")
    # Attempt to add same title with different casing/whitespace
    with pytest.raises(ValueError, match=r"^book with title 'my book' already exis
        lib.add_book(" my book ", "Author Two")


def test_list_books_sorted_case_insensitive_and_deterministic():
    lib = Library()
    lib.add_book("bTitle", "Author B")
    lib.add_book("Atitle", "Author A")
    lib.add_book(" cTitle ", "Author C")
    listed = lib.list_books()
    titles = [b["title"] for b in listed]
    # Titles should be trimmed and in case-insensitive ascending order: Atitle, bT
    assert titles == ["Atitle", "bTitle", "cTitle"]


def test_list_books_returns_shallow_copies_not_internal_references():
    lib = Library()
    lib.add_book("Mutable", "Author")
    books = lib.list_books()
    # Modify returned dict
    books[0]["title"] = "Tampered"
    # Original stored book should remain unchanged
    found = lib.find_book_by_title("Mutable")
    assert found["title"] == "Mutable"


def test_find_book_by_title_success_and_error_conditions():
    lib = Library()
```

```
    lib.add_book("Dune", "Frank Herbert")
    # Successful find with different casing/whitespace
    found = lib.find_book_by_title(" dune ")
    assert found == {"title": "Dune", "author": "Frank Herbert"}

    # Type error when non-string provided
    with pytest.raises(TypeError, match=r"^title must be a string$"):
        lib.find_book_by_title(99)

    # ValueError when trimmed title is empty
    with pytest.raises(ValueError, match=r"^title must be a non-empty string$"):
        lib.find_book_by_title("   ")

    # LookupError when not found; message must include trimmed input
    with pytest.raises(LookupError, match=r"^book with title 'NotHere' not found$"
        lib.find_book_by_title(" NotHere ")


def test_remove_book_by_title_success_and_error_conditions():
    lib = Library()
    lib.add_book("Frankenstein", "Mary Shelley")
    # Remove using different casing/whitespace; should return stored values
    removed = lib.remove_book_by_title(" FRANKENSTEIN ")
    assert removed == {"title": "Frankenstein", "author": "Mary Shelley"}
    # After removal, book should no longer be found
    with pytest.raises(LookupError, match=r"^book with title 'Frankenstein' not fou
        lib.find_book_by_title("Frankenstein")

    # TypeError for non-string input
    with pytest.raises(TypeError, match=r"^title must be a string$"):
        lib.remove_book_by_title(None)

    # ValueError for empty trimmed title
    with pytest.raises(ValueError, match=r"^title must be a non-empty string$"):
        lib.remove_book_by_title("   ")

    # LookupError for non-existent title with exact trimmed input in message
    with pytest.raises(LookupError, match=r"^book with title 'Gone' not found$"):
        lib.remove_book_by_title(" Gone ")


def test_integration_add_find_remove_and_list_together():
    lib = Library()
```

```
    # Add multiple books
    lib.add_book("1984", "George Orwell")
    lib.add_book("Brave New World", "Aldous Huxley")
    lib.add_book(" Fahrenheit 451 ", "Ray Bradbury")

    # Find one
    found = lib.find_book_by_title(" fahrenheit 451 ")
    assert found == {"title": "Fahrenheit 451", "author": "Ray Bradbury"}

    # Remove another
    removed = lib.remove_book_by_title("1984")
    assert removed == {"title": "1984", "author": "George Orwell"}

    # Remaining list should be sorted case-insensitively
    remaining = lib.list_books()
    remaining_titles = [b["title"] for b in remaining]
    assert remaining_titles == ["Brave New World", "Fahrenheit 451"]
```

Deployment Configuration

requirements.txt

run.sh

```
#!/usr/bin/env bash
set -euo pipefail

# Expected single-file Python application name
APP=library.py

if [ ! -f "$APP" ]; then
  echo "Error: $APP not found. Place the application single Python file named $
  exit 1
fi

# Install dependencies only if requirements.txt is non-empty
if [ -s requirements.txt ]; then
  echo "Installing Python dependencies from requirements.txt..."
  python3 -m pip install --user -r requirements.txt
fi
```

```
echo "Running $APP with python3"
exec python3 "$APP"
```