PubMed Paper Fetcher: Development Approach Documentation

Project Overview

This document outlines the systematic approach taken to develop a Python program that fetches research papers from PubMed API, specifically targeting papers with at least one author affiliated with pharmaceutical or biotech companies.

Problem Analysis & Requirements

The task required building a command-line tool with the following key requirements:

- Fetch papers using PubMed API with flexible query syntax
- Identify and filter papers with non-academic (company) authors
- Extract specific metadata and output to CSV format
- Support command-line arguments for query, debug mode, and file output
- Use Poetry for dependency management and create an executable command

Development Methodology

Phase 1: API Research & Understanding

Approach: Started with comprehensive API documentation analysis using Claude AI to create a structured reference guide.

Key Insights Discovered:

- PubMed E-utilities consists of multiple endpoints (ESearch, EFetch, ESummary)
- Two-step process required: ESearch for PMIDs → EFetch for detailed data
- XML parsing needed for extracting author affiliations
- Rate limiting considerations (10 requests/second with API key)
- Batch processing capabilities for efficiency

Reference Document Created:

```
# NCBI PubMed E-utilities API Guide
- Base URL: https://eutils.ncbi.nlm.nih.gov/entrez/eutils/
- Essential endpoints: esearch.fcgi, efetch.fcgi
- Key parameters: db=pubmed, retmode=xml, api_key
- Response structure analysis for XML parsing
```

Phase 2: Iterative Development with Basic Testing

Strategy: Build incrementally, testing each component before adding complexity.

Initial Implementation Steps:

1. **Basic main() function** without argparse
2. **PMID fetching** using ESearch API
3. **XML parsing** for paper details using EFetch API
4. **Step-by-step validation** of each API response

Testing Approach:

- Started with simple queries to verify API connectivity
- Tested XML parsing with known PMIDs
- Validated data extraction for individual papers
- Ensured proper handling of API responses

Phase 3: Skeleton Generation & Core Logic

Approach: Used ChatGPT to generate a structured skeleton while maintaining control over the core logic.

Key Components Developed:

- `PubMedFetcher` class structure
- Method signatures for data extraction
- XML parsing logic for author affiliations
- Basic error handling framework

Personal Contributions:

- Defined return types and data structures
- Implemented business logic for company identification
- Designed the filtering algorithm for non-academic authors

Phase 4: Command-Line Interface Integration

Implementation: Added argparse for professional command-line tool functionality.

Arguments Implemented:

```
parser.add_argument("query", nargs="?", help="Search query for PubMed.")
parser.add_argument("-d", "--debug", action="store_true", help="Enable debug logging.")
parser.add_argument("-f", "--file", type=str, help="Output filename (CSV).")
parser.add_argument("-m", "--max", type=int, default=20, help="Max number of results")
```

Phase 5: Critical Bug Fix - Filtering Logic Error

Problem Identified: Initial implementation was incorrectly filtering out company-affiliated authors and retaining only academic ones.

Root Cause: Logical error in the filtering condition - the boolean logic was inverted.

Solution Process:

1. **Debugging**: Added logging to trace filtering decisions
2. **ChatGPT Consultation**: Explained the expected behavior vs actual behavior
3. **Logic Correction**: Fixed the conditional statements in `_filter_company_authors()`
4. **Validation**: Tested with known company-affiliated papers

Corrected Logic:

```
# Before (incorrect): Excluded company affiliations
if not any(self._is_company_affiliation(aff) for aff in author['affiliations']):
    company_authors.append(author['name'])

# After (correct): Included company affiliations
if any(self._is_company_affiliation(aff) for aff in author['affiliations']):
    company_authors.append(author['name'])
```

Phase 6: Code Quality & Documentation Enhancement

Approach: Used ChatGPT for code review and documentation generation while maintaining code ownership.

Improvements Made:

- **Comprehensive docstrings** for all methods
- **Inline comments** explaining complex logic
- **Type hints** throughout the codebase
- **Error handling** improvements
- **Code optimization** suggestions

Key Optimization - Session Retention:

```
# Before: New connection for each request
response = requests.get(url, params=params)

# After: Session reuse for efficiency
self.session = requests.Session()
response = self.session.get(url, params=params)
```

Phase 7: Environment Setup & Packaging

Challenge: Setting up Poetry and executable command integration.

Environment Issues Encountered:

1. **Python Installation**: Initial Python installation from website caused PATH issues
2. **VSCode Terminal**: Unable to access Poetry commands in VSCode terminal
3. **Solution**: Reinstalled Python from Microsoft Store for proper PATH configuration

Poetry Configuration:

```
[tool.poetry.scripts]
get-papers-list = "pubmed_fetcher.main:main"
```

Success Metrics:

- `poetry install` successfully set up dependencies
- `poetry run get-papers-list` command worked correctly
- All requirements met for packaging and distribution

Technical Architecture Decisions

Class-Based Design

Rationale: Chose `PubMedFetcher` class to encapsulate:

- API configuration and session management
- State management for PMIDs and paper data
- Reusable methods for different query types
- Clean separation of concerns

Company Identification Strategy

Multi-layered Approach:

1. **Known Company List**: Curated list of major pharma/biotech companies
2. **Legal Entity Suffixes**: Corp, Inc, Ltd, GmbH, etc.
3. **Industry Keywords**: Biotech, therapeutics, pharmaceutical, etc.
4. **Academic Exclusions**: University, hospital, research center, etc.

Batch Processing Strategy

Performance Considerations:

- Fetch extra PMIDs (5x target) to account for filtering
- Process in batches of 100 to respect API limits
- Early termination when target results achieved
- Session reuse for network efficiency

Development Tools & Resources Used

AI/LLM Tools

- **Claude AI**: API documentation analysis and reference guide creation
- **ChatGPT**: Code skeleton generation, debugging assistance, documentation enhancement, and optimization suggestions

Development Environment

- **Python**: Core programming language
- **Poetry**: Dependency management and packaging
- **Git/GitHub**: Version control and code hosting
- **VSCode**: Development environment (with environment setup challenges)

Key Libraries

- **requests**: HTTP client for API calls
- **xml.etree.ElementTree**: XML parsing for PubMed responses
- **argparse**: Command-line argument parsing
- **csv**: Output file generation
- **logging**: Debug and error tracking

Lessons Learned

Technical Insights

1. **API Integration**: Two-step process (search → fetch) is common pattern
2. **XML Parsing**: Robust handling of missing fields is crucial

3. **Filtering Logic**: Boolean logic errors can be subtle and hard to debug
4. **Environment Setup**: Python installation source matters for PATH configuration

Development Process

1. **Incremental Development**: Testing each component before integration prevents compound errors
2. **AI Assistance**: Effective when combined with domain knowledge and testing
3. **Documentation**: Creating API reference upfront saved significant development time
4. **Error Handling**: Comprehensive logging is essential for debugging API integrations

Problem-Solving Approach

1. **Research First**: Understanding the API thoroughly before coding
2. **Test Early**: Validating assumptions with real data
3. **Iterate Quickly**: Building incrementally with frequent testing
4. **Seek Help**: Using AI tools for specific challenges while maintaining code ownership
5. **Document Everything**: Creating references and documentation for future maintenance

Final Implementation Quality

Code Organization

- **Modular design** with clear separation of concerns
- **Comprehensive error handling** for API failures and data issues
- **Efficient batch processing** with session reuse
- **Flexible filtering** with multiple identification strategies

Performance Characteristics

- **Batch API calls** for efficiency
- **Session reuse** for network optimization
- **Early termination** when target results achieved
- **Configurable limits** for different use cases

Robustness Features

- **Rate limit compliance** with API key usage
- **XML parsing resilience** for missing fields
- **Comprehensive logging** for debugging
- **Graceful failure handling** for network issues

This development approach successfully delivered a production-ready tool that meets all specified requirements while maintaining code quality and performance standards.