C to Python Conversion Agent

Project Overview

This project implements an intelligent agent that automatically converts C codebases to Python, preserving functionality while adapting to Python's idioms and best practices. The agent uses machine learning to understand the structure and intent of C code, enabling high-quality translations that go beyond simple syntax conversion.

Features

- **Full Directory Processing**: Scan and process entire C projects with multiple files
- **Intelligent Code Analysis**: Parse and understand C code structure and dependencies
- **LLM-Powered Translation**: Convert C constructs to Pythonic equivalents
- **Import Management**: Automatically handle library dependencies
- **Iterative Verification**: Ensure correctness through multi-stage verification
- **Documentation Generation**: Create comprehensive documentation for the translated project
- **Python Best Practices**: Follow PEP 8 style guidelines and modern Python conventions

Project Structure

Copy

```
c2py-agent/
├── core/
│   ├── __init__.py
│   ├── parser/                 # C code parsing functionality
│   │   ├── __init__.py
│   │   ├── ast_generator.py     # Abstract Syntax Tree generation
│   │   ├── c_preprocessor.py    # Handle C macros and includes
│   │   ├── symbol_table.py      # Track variables, functions, types
│   │   └── dependency_mapper.py # File relationships and dependencies
│   ├── llm/                    # LLM integration
│   │   ├── __init__.py
│   │   ├── api_client.py        # Interface with LLM service
│   │   ├── prompt_templates.py  # Structured prompts for different conversions
│   │   ├── code_analyzer.py     # Understand code intent and structure
│   │   └── translation_rules.py # Rules for C-to-Python mappings
│   ├── generator/              # Python code generation
│   │   ├── __init__.py
│   │   ├── code_generator.py    # Create Python code from intermediate rep
│   │   ├── import_manager.py    # Manage Python import statements
│   │   ├── type_annotations.py  # Add Python type hints
│   │   └── docstring_gen.py     # Generate docstrings for functions/classes
│   └── verification/           # Testing and verification
│       ├── __init__.py
│       ├── static_analyzer.py   # Lint and analyze generated code
│       ├── import_validator.py  # Verify imports resolve correctly
│       ├── syntax_checker.py    # Ensure valid Python syntax
│       └── test_runner.py       # Execute tests if available
├── utils/
│   ├── __init__.py
│   ├── config_manager.py       # Handle configuration options
│   ├── file_scanner.py         # Scan directories and files
│   ├── logger.py               # Logging functionality
│   └── documentation.py        # Generate project documentation
├── cli/
│   ├── __init__.py
│   ├── main.py                 # Command-line interface
│   └── interactive.py          # Interactive mode functions
├── tests/                      # Tests for the conversion agent itself
│   ├── test_parser.py
│   ├── test_generator.py
│   ├── test_verification.py
│   └── test_end_to_end.py
├── config/
│   ├── default_config.yml      # Default configuration
│   └── mapping_rules.yml        # C-to-Python mapping rules
├── README.md
├── requirements.txt
└── setup.py
```

Installation

bash

Copy

```bash
# Clone the repository
git clone https://github.com/yourusername/c2py-agent.git
cd c2py-agent

# Create and activate virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt

# Install the package in development mode
pip install -e .
```

Requirements

- Python 3.8+
- pycparser
- llm-client (or specific LLM SDK)
- pytest (for testing)
- typed_ast
- black (for code formatting)
- isort (for import sorting)
- pylint (for static analysis)

Usage

Basic Usage

bash

Copy

```bash
# Convert a C project to Python
c2py-agent convert /path/to/c/project /path/to/output/directory

# With custom config
c2py-agent convert /path/to/c/project /path/to/output --config my_config.yml

# Interactive mode for step-by-step conversion
c2py-agent interactive /path/to/c/project
```

Configuration Options

Create a custom configuration file to fine-tune the conversion process:

yaml

Copy

```yaml
# custom_config.yml
parser:
  ignore_headers: ["vendor/*", "test/*"]
  preprocess_macros: true

llm:
  model: "gpt-4"  # or "claude-3" or other
  temperature: 0.2
  max_tokens: 4096

generator:
  use_type_hints: true
  style: "pep8"
  doc_style: "google"  # or "numpy" or "sphinx"

verification:
  max_iterations: 5
  run_tests: true
  strict_imports: true
```

Addressing Conversion Challenges

1. Complex C Features

- **Pointers**: Automatically convert to appropriate Python alternatives:
  - Function pointers → First-class functions or lambdas
  - Data structure pointers → Python references or custom wrapper classes

    - Void pointers → Type-agnostic containers or dynamic typing
- **Memory Management**:
    - Replace malloc/free with Python's automatic memory management
    - Use context managers for resource cleanup
    - Convert manual buffer management to Python lists or bytes objects
- **Macros**:
    - Expand simple macros directly
    - Convert complex macros to functions or classes
    - Handle conditional compilation with runtime checks

2. Platform-Specific Code

- **Approach**: The agent identifies platform-specific sections and offers alternatives:
    - Create Python modules that use ctypes to interface with C libraries
    - Identify equivalent Python libraries for platform interaction
    - Flag sections requiring manual conversion with detailed notes
- **Assembly Code**:
    - Flag inline assembly for manual review
    - Suggest pure Python alternatives where possible
    - Provide guidance for using Python's C extension mechanisms

3. Performance Considerations

- **Performance Equivalence**:
    - Identify performance-critical sections and suggest optimizations:
        - NumPy/SciPy for numerical operations
        - PyPy for compatible code
        - Cython for critical sections
        - Optional C extension generation for bottlenecks
- **Parallelism Translation**:
    - Convert pthreads to Python's threading or multiprocessing
    - Map OpenMP constructs to concurrent.futures

4. External Dependencies

- **C Library Mapping**:
    - Maintain a mapping database of common C libraries to Python equivalents
    - Suggest pip packages for common C libraries

    - For unique libraries, suggest Python C extension approach
- **FFI Generation**:
    - Generate Python Foreign Function Interface code for C libraries without Python equivalents
    - Create ctypes or cffi wrappers automatically

Development Workflow

1. **Input Processing**:
    - Parse command-line arguments
    - Load configuration
    - Scan and index C project files
2. **C Code Analysis**:
    - Generate ASTs for each file
    - Create symbol tables
    - Map dependencies between files
    - Identify external libraries
3. **LLM-Based Translation**:
    - Feed preprocessed code to LLM with appropriate prompts
    - Analyze code intent and structure
    - Apply translation rules
4. **Python Code Generation**:
    - Create Python module structure
    - Generate code for each module
    - Manage imports and dependencies
    - Add type annotations and documentation
5. **Verification Loop**:
    - Verify syntax correctness
    - Check import resolution
    - Run static analysis
    - Execute tests if available
    - Iterate on issues until convergence
6. **Documentation Generation**:
    - Create README and setup instructions
    - Document project structure
    - Provide usage examples

- Note any manual intervention required

Contributing

Contributions are welcome! Please feel free to submit a Pull Request.

1. Fork the repository
2. Create your feature branch (`git checkout -b feature/amazing-feature`)
3. Commit your changes (`git commit -m 'Add some amazing feature'`)
4. Push to the branch (`git push origin feature/amazing-feature`)
5. Open a Pull Request

License

This project is licensed under the MIT License - see the LICENSE file for details.