CS 264 HW - ReAct Agent for Software Engineering
Final Report

1. ACCURACY RESULTS

Final Accuracy: 8/20 (40%)

Evaluation Summary:
- Total Instances: 20
- Resolved Instances: 8
- Unresolved Instances: 12
- Empty Patches: 0
- Errors: 0

Resolved Instance IDs:
1. astropy__astropy-7166
2. django__django-11179
3. django__django-13810
4. django__django-14053
5. django__django-16662
6. django__django-7530
7. scikit-learn__scikit-learn-26323
8. sympy__sympy-24213

Unresolved Instance IDs:
1. django__django-10973
2. django__django-12406
3. django__django-13297
4. django__django-14011
5. django__django-16631
6. psf__requests-1921
7. psf__requests-2931
8. pytest-dev__pytest-7490
9. sphinx-doc__sphinx-7590
10. sphinx-doc__sphinx-9230
11. sphinx-doc__sphinx-9658
12. sympy__sympy-17655

2. OBSERVATIONS

The agent achieved a 40% success rate on the SWE-bench subset. Analysis of the report.json files for each instance revealed the following patterns:

Success Rates by Project:
- Django: 5/10 (50%) - django__django-11179, 13810, 14053, 16662, 7530
- Sphinx: 0/3 (0%) - All three instances failed
- Sympy: 1/2 (50%) - sympy__sympy-24213 resolved
- Requests: 0/2 (0%) - Both instances failed
- Others: astropy (1/1), scikit-learn (1/1), pytest (0/1)

Primary Failure Patterns (based on test_status in report.json):

1. INCOMPLETE FIXES (most common):
   Many failures show partial success in FAIL_TO_PASS - the agent fixed some
   target tests but not all. Examples:
   - django__django-10973: Fixed 4/5 target tests
   - django__django-12406: Fixed 1/3 target tests
   - pytest-dev__pytest-7490: Fixed 2/3 target tests

2. REGRESSIONS (2 instances):
   The psf__requests instances show PASS_TO_PASS failures (tests that passed in original code
   now fail due to the agent's changes):
   - psf__requests-1921: 23 regressions in PASS_TO_PASS.failure

- psf__requests-2931: 1 regression in PASS_TO_PASS.failure

3. NO PROGRESS (some instances):
   Some instances show 0 success in FAIL_TO_PASS, meaning the agent's changes
   did not fix any of the target tests:
   - django__django-14011: 0/17 target tests fixed
   - sphinx-doc__sphinx-7590: 0/1 target test fixed
   - sphinx-doc__sphinx-9230: 0/1 target test fixed

Observations:
- The agent performed best on Django issues, likely due to Django's well-
  documented codebase and clear code patterns.
- Sphinx issues had 0% success rate, suggesting difficulty with documentation
  tooling codebases or complex template/parsing logic.
- The requests library failures involved significant regressions.

3. CUSTOM TOOLS CREATED

I implemented 7 custom tools in envs.py to enhance the agent's ability to navigate, understand, and modify code. Below is a detailed description of each tool and the reasoning behind its design.

3.1 show_file(file_path, start_line, end_line)

Purpose: Display file contents with line numbers.

Description:
This tool reads a file and displays its contents with line numbers prefixed to each line (similar to `cat -n`). It supports optional start_line and end_line parameters to view specific portions of large files. By default, it shows 200 lines starting from the specified start_line.

Reasoning:
Line numbers are essential for the agent to make precise edits. When the agent needs to modify code, knowing exact line numbers helps it:
1. Reference specific locations in discussions
2. Understand the context around code to be modified
3. Provide more context for replace_in_file operations
4. Navigate large files efficiently by viewing relevant sections

This tool is marked as a prerequisite action: "ALWAYS use this before editing a file" to ensure the agent sees current file state before modifications.

3.2 replace_in_file(file_path, old_content, new_content)

Purpose: Perform precise, exact-match text replacements in files.

Description:
This tool finds an exact string match of old_content in the specified file and replaces it with new_content. It includes several safety features:
- Validates that old_content exists exactly in the file
- Checks for unique matches (fails if multiple matches found)
- Provides helpful error messages suggesting to use show_file first
- Uses heredoc with unique delimiters to handle special characters

Reasoning:
Precise code editing is critical for SWE-bench tasks. Unlike sed or regex-based replacements which can have unintended side effects:
1. Exact matching prevents accidental modifications to similar but different code
2. The uniqueness check forces the agent to provide enough context
3. Helpful error messages guide the agent to correct its approach
4. Safe file writing with heredocs handles quotes, newlines, and special chars

The requirement that "old_content must match EXACTLY, including all whitespace and indentation" encourages the agent to carefully examine files before editing.

3.3 search_code(pattern, path, file_pattern)

Purpose: Search for code patterns across the codebase.

Description:
This tool wraps `grep -rn` to search for regex patterns in code files. It:
- Searches recursively through directories
- Returns file paths, line numbers, and matching content
- Supports optional file pattern filtering (e.g., "*.py")
- Limits output to 50 matches to prevent overwhelming the context

Reasoning:
Finding relevant code is the first step in solving most issues. This tool helps:
1. Locate function/class definitions mentioned in issue descriptions
2. Find error messages or exception types
3. Discover related code patterns for understanding context
4. Identify all locations that might need modification

The 50-match limit balances comprehensiveness with context efficiency. The file_pattern option allows focusing on specific file types (essential in mixed-language projects).

3.4 find_files(name_pattern, path)

Purpose: Locate files by name pattern.

Description:
This tool uses `find` to locate files matching a name pattern. It:
- Supports glob patterns (e.g., "*.py", "test_*.py", "models.py")
- Filters out __pycache__ and .git directories automatically
- Limits results to 50 matches

Reasoning:
Before searching code content, the agent often needs to find specific files:
1. Test files to understand expected behavior (without modifying them)
2. Configuration files (setup.py, pyproject.toml, etc.)
3. Files with conventional names (models.py, views.py, urls.py)
4. Files mentioned in issue descriptions or error messages

The automatic filtering of __pycache__ and .git reduces noise and focuses on source files that matter for solving issues.

3.5 list_directory(path)

Purpose: Display directory contents and structure.

Description:
This tool runs `ls -la` on a directory to show:
- All files and subdirectories
- File permissions, sizes, and timestamps
- Hidden files (with the -a flag)

Reasoning:
Understanding project structure is essential before diving into code:
1. Helps the agent orient itself in unfamiliar codebases
2. Reveals project organization (src/, tests/, docs/, etc.)
3. Shows hidden configuration files (.gitignore, .flake8, etc.)
4. Provides quick overview without reading individual files

This tool is recommended as the first action: "Start with this to understand project structure" in the system prompt.

3.6 create_file(file_path, content)

Purpose: Create new files with specified content.

Description:
This tool creates a new file at the specified path with the given content. It:
- Automatically creates parent directories if they don't exist
- Uses heredoc for safe content handling
- Handles special characters properly

Reasoning:
While most SWE-bench tasks involve modifying existing files, some require:
1. Adding new test files (though we discourage modifying tests)
2. Creating new modules or helper files
3. Adding configuration files

The automatic parent directory creation (mkdir -p) prevents errors when creating
files in nested paths that don't yet exist.

3.7 view_around_line(file_path, line_number, context)

Purpose: View code context around a specific line.

Description:
This is a convenience wrapper around show_file that takes a center line number
and displays surrounding context. By default, it shows 10 lines before and after
the specified line number.

Reasoning:
This tool is particularly useful after search_code finds a match:
1. search_code returns line numbers of matches
2. view_around_line provides immediate context around those matches
3. The agent can quickly understand how the matching code fits into its surroundings
4. Helps decide if the found location is the right place to make changes

This creates an efficient workflow: search_code -> view_around_line -> replace_in_file

4. AGENT ARCHITECTURE

The agent follows the ReAct (Reasoning and Acting) paradigm with these components:

4.1 Response Parser (response_parser.py)
- Uses text-based markers (BEGIN_FUNCTION_CALL, END_FUNCTION_CALL, ARG, VALUE)
- rfind-based parsing to find the LAST occurrence of markers
- This prevents confusion when markers appear in reasoning/code examples
- Returns structured dict with thought, function name, and arguments

4.2 Agent Core (agent.py)
- Maintains message history with role, content, timestamp, unique_id
- Builds context for LLM including tool descriptions and response format
- Runs ReAct loop with error tracking (max 5 consecutive errors)
- Truncates tool output at 15000 characters to manage context
- Supports type conversion for function arguments (int, float, bool)

4.3 System Prompt
- Provides structured workflow: Understand -> Explore -> Locate -> Plan -> Fix -> Verify
- Includes tool-specific tips and common patterns
- Emphasizes minimal changes and verifying edits
- Explicitly warns against modifying test files

5. DESIGN DECISIONS AND TRADE-OFFS

5.1 Custom Tools vs. Raw Bash
Decision: Provide specialized tools rather than relying solely on run_bash_cmd.
Rationale: Specialized tools provide better error handling, output formatting,
and guardrails. They also make the system prompt clearer by documenting

available actions explicitly.

5.2 Output Truncation (15000 characters)
Decision: Truncate long tool outputs to preserve context window.
Rationale: Large files or search results can overwhelm the context. Truncation keeps the agent focused while still providing useful information.

5.3 Exact-Match Replacement
Decision: require_in_file requires exact matching with no regex.
Rationale: Regex replacements are error-prone and can modify unintended code. Exact matching is safer and forces the agent to be precise.

5.4 rfind-Based Parsing
Decision: Use rfind to locate the LAST function call markers.
Rationale: The agent might include examples of the format in its reasoning. Using rfind ensures we parse the actual function call, not an example.

6. CONCLUSION

The ReAct agent achieved 40% accuracy on the SWE-bench subset. The custom tools were designed to support a systematic workflow: explore the codebase, locate relevant code, understand context, make precise edits, and verify changes.

Key strengths:
- Specialized tools with clear semantics
- Robust parsing with rfind
- Helpful error messages guiding the agent
- Emphasis on minimal, verified changes

Areas for improvement:
- Better handling of multi-file changes
- Improved understanding of test requirements to avoid regressions
- More sophisticated search capabilities for complex codebases

End of Report