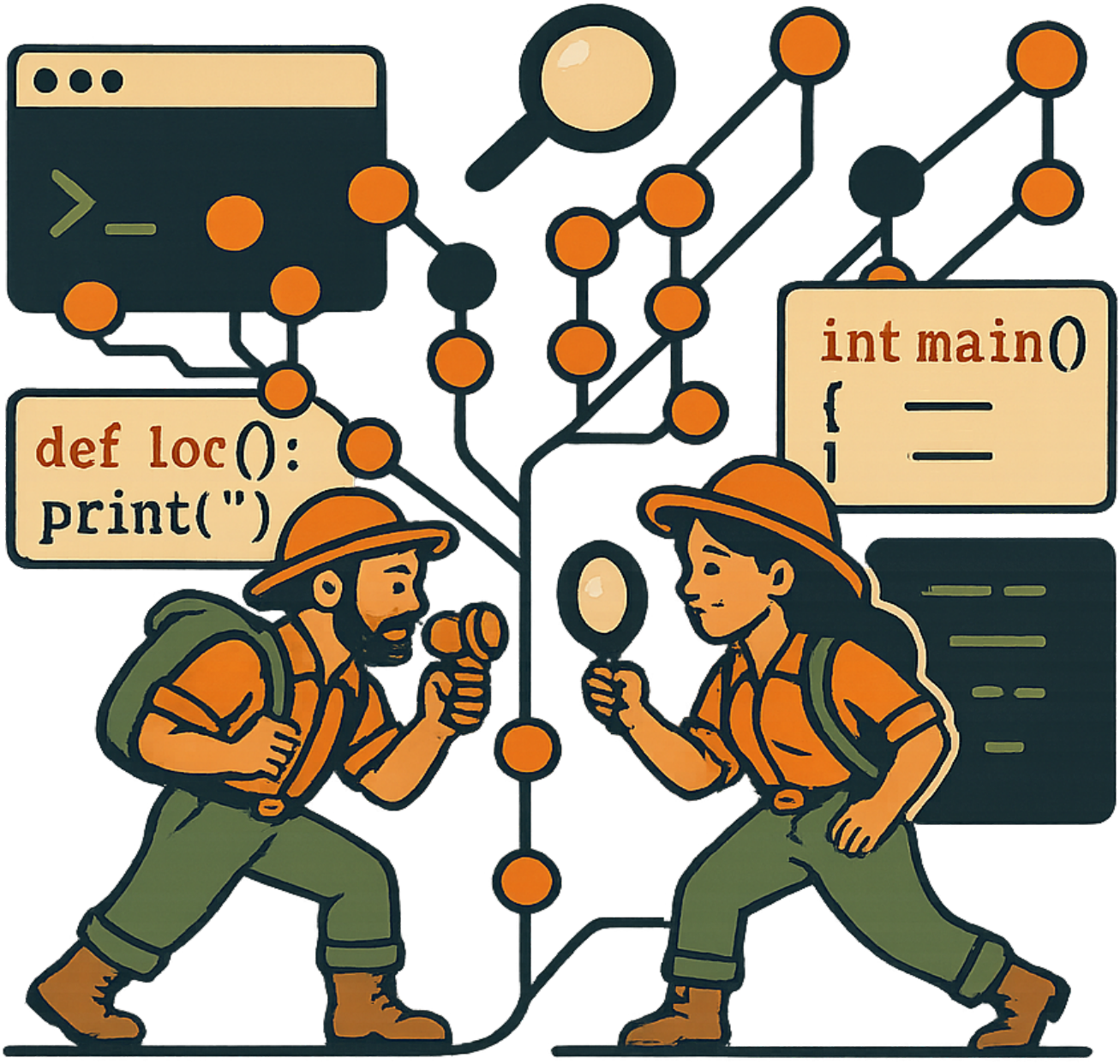
ADVANCED SYSTEMS PROGRAMMING

INTRODUCTION ASSIGNMENT

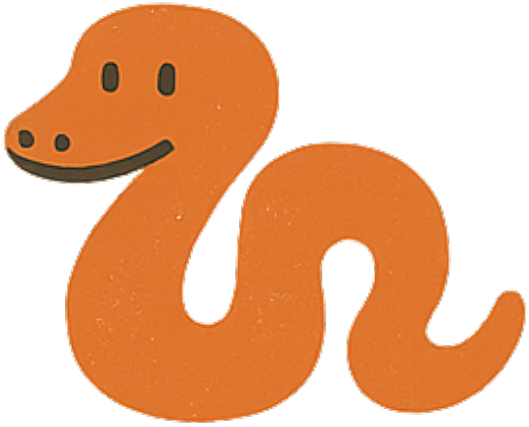


IMPORTANT POLICIES

Academic integrity and AI: This assignment requires your own analysis and understanding. While you may consult external resources, all written work must be in your own words. You are encouraged to work with the latest AI tools, but submissions containing unedited AI output or lack original analysis will be **disqualified**. You may discuss the assignment with other students, but each submission must be your own.

Collaboration: This assignment is submitted in pairs. Collaboration is to be done via a single GitHub repository forked from the original course repository. Your repositories should be private. Do not share them with other students. Do not open pull requests to the original repository unless asked to do so.

Getting help: If you encounter technical issues you cannot resolve after reasonable effort, contact course staff. Learning when to seek help (and when not to) is part of the assignment.



INTRODUCTION

Welcome to Advanced Systems Programming 2026! This assignment introduces you to a real-world codebase called CAF (Content Addressable Filesystem), a lightweight version control system similar to git, written in Python and C++. CAF was thoughtfully developed by previous students (Meshi, Bar, and Omer) and serves as your gateway to understanding how complex systems are architected, implemented, and maintained. Though it may appear modest in scale, CAF contains the same fundamental challenges found in production systems: command line interfaces, state management, file operations, data integrity, and more. Through this project, you'll develop essential skills in code analysis, system design, debugging, and feature implementation. You'll learn not just how real-world systems work, but why they're designed the way they are.

Prerequisites:

- Basic familiarity with Python and C++
- Docker installed and functional
- git basics (clone, commit, push)
- Command-line comfort

Learning objectives:

- Analyze and understand complex, multi-language codebases
- Map software architecture and component relationships
- Debug and fix issues in unfamiliar code
- Implement new features following existing patterns
- Communicate technical findings clearly

GETTING STARTED: SETTING UP THE PROJECT

Repository setup:

1. Fork the repository to your own GitHub account. Students working together should collaborate using **one fork**. Do not open any pull requests to the original repository
2. Clone your forked repository to your local machine
3. Refer to the `README.md` file in the root directory of the repository for detailed setup instructions, usage examples, and project overview. It contains comprehensive information about the project structure, commands, and development workflow

Next, we will delve into the codebase and its accompanying files, charting the layout and tracing the flow of information throughout the system. In the sections ahead, we will uncover the remnants of past decisions, examine the directory structure, and outline the boundaries of each module. Picture it as exploring an ancient ruin, uncovering its secrets one directory at a time.

TASK I: EXPLORING THE CODEBASE

In this section, you will develop your ability to analyze, synthesize, and communicate technical information by exploring the project. Create a comprehensive list of all significant files and directories, and for each, provide a thoughtful analysis of its purpose, role, and relationships within the project. This will lay the foundation for deeper exploration in the sections ahead.

Objective: Create a comprehensive catalog of the project's structure and understand each component's role.

Required analysis: For each significant file or logical group of files, provide:

1. **Functionality:** What does this file do?
2. **Purpose:** Why is it needed in the project?
3. **Relationships:** How does it connect to other components, if any?
4. **Your insight:** Your own understanding or observation, if applicable.

What constitutes "significant" files:

- Core source files (Python/C++ modules with >20 lines)
- Configuration files (Makefile, Dockerfile, setup files)
- Test files and test infrastructure
- You may exclude: temporary files, `__pycache__`, `.gitignore`, and other files that do not contribute to the project's functionality or structure.

Deliverable: A structured list with your analysis.

Example of quality analysis: Your analysis should be clear, concise, and insightful. It does not have to be this detailed, but it should show that you have thought about the purpose and role of each file or group of files.

1. `caf/caf/__main__.py`: This file serves as the application entry point, enabling CAF to run as both a module (`python -m caf`) and installed command (`caf`). It leverages Python's `__name__` mechanism to detect script execution and delegates to the CLI module. This separation keeps the main logic in cli.py while providing multiple execution pathways.

2. `caf/caf/cli.py`: This file implements the command-line interface (CLI) for the CAF application. It defines the commands and options available to users, such as `init`, `commit`, and others. The CLI is built using the `argparse` library, which simplifies the creation of command-line interfaces in Python. The `cli()` function is called from `__main__.py`. It uses a data structure that describes each command, its options, and its arguments to dispatch the appropriate function when a command is invoked. Actual command implementations live in `cli_commands.py`.

Tips:

- Look for patterns in file names and directory structure.
- Try to infer the flow of information and control between files.
- Consider how each file contributes to the overall goals of the project.
- If you use external help (like Google or an AI tool), always process and rephrase the information in your own words.

This exercise is not just about cataloging files. It's about developing your ability to analyze, synthesize, and communicate technical information. Your insights here will lay the foundation for deeper exploration in the sections ahead.

TASK 2: MAPPING THE CODEBASE

In this section, you will build a mental model of the project as a system. You will identify the main concepts, modules, classes, and functions, and map how they interact. This will help you see how the pieces fit together and make you more effective in extending and maintaining the codebase.

Objective: Create a visual representation of how the system components interact and information flows through the application.

Required diagram elements:

- Main modules and their primary responsibilities
- Key classes and their relationships
- Entry points and execution flow
- Python/C++ boundaries and interfaces
- Data flow between components
- File system interactions

Special focus areas:

- **Language Integration:** Where is the Python/C++ boundary? How do they communicate? What information or data is passed between them?
- **Data Persistence:** How is repository state stored and retrieved?
- **Command Processing:** Path from CLI input to execution
- **Error Handling:** How errors propagate through the system

Diagram requirements:

- Use clear shapes and colors to distinguish component types
- Include directional arrows showing information/control flow
- Label connections with the type of interaction
- Acceptable formats: PDF, PNG, PPTX or JPG (scanned hand-drawn diagrams must be clearly legible, well-organized, and visually appealing)
- Recommended tools: draw.io, Lucidchart, PowerPoint or similar

Deliverable: A single, comprehensive diagram with accompanying brief explanation (1-2 paragraphs) of your architectural insights. Your map should reflect your own understanding. Do not simply copy output from an AI tool or documentation generation tool!

The best diagrams will be those that clearly and accurately represent the system architecture, show deep understanding of the codebase, and are visually appealing. It might be included in future course materials, so make it your best work!

This section is about building a mental model of the project as a system rather than flat textual files. This is a crucial skill in programming, as it allows you to see how different parts of the codebase interact and depend on each other. The more clearly you can see how the pieces fit together, the more effective you will be in extending and maintaining the codebase.

TASK 3: BRINGING THE PROJECT TO LIFE

Now that you have explored the codebase and mapped its structure, your next step is to get the project running on your own machine. We use Docker, which simplifies the setup of a consistent development environment. This ensures you can bring the project to life and begin experimenting with it easily.

Objective: Set up the development environment and run the test suite.

Setup steps:

1. Build the Docker image using the provided Makefile
2. Start and enter the Docker container
3. Deploy the library (libcaf) and the CLI (caf) in the container
4. Verify the environment by running basic CAF commands
5. Execute the complete test suite using `pytest`

See the project `Makefile` for commands to execute outside and inside the container. If you encounter issues you cannot resolve, contact the course staff for help. Part of the assignment is learning to how to solve problems yourself, and also when to seek assistance.

Testing the application:

1. Run `pytest` to see detailed test results
2. Record which tests pass/fail with exact counts
3. For failing tests, note the error messages
4. Test basic CLI functionality manually (e.g., `caf --help`)

Deliverable: None.

TASK 4: DEBUGGING AND FIXING FAILING TESTS

In this section, you will develop your debugging and problem-solving skills. Once you have run the test suite, you will find that some tests don't pass. Investigate these failing tests, understand the reasons behind their failure, and fix the underlying issues. Focus on understanding the intent of the code and tests, and make thoughtful, minimal changes.

Objective: Develop systematic debugging skills and fix code issues that cause test failures.

Debugging methodology:

1. **Identify:** Which specific tests are failing?
2. **Analyze:** Read test code to understand expected behavior
3. **Investigate:** Trace through application code to find the issue, perhaps using print statements or a debugger
4. **Hypothesize:** Form a theory about the root cause
5. **Fix:** Make minimal, targeted changes to the application or test code as needed
6. **Verify:** Re-run tests to confirm the fix

Understanding the failing tests:

When you run the test suite, you will encounter two kinds of failing tests. Understanding why each test fails is crucial to developing effective debugging skills:

- Some tests fail because they correctly verify behavior that the current implementation does not properly handle. The test is valid, but the code has a bug. In cases like these, your task is to **fix the underlying code** to make the invariants tested by the tests hold true.
- Some tests fail due to relying on details that they should not be relying on. The test simply isn't robust in the face of changes. While you could fix the code to match the expected outcome, the proper fix is to **make the test less reliant on irrelevant details** by modifying the test itself.

Important constraints:

- Make minimal changes that address root causes
- Preserve existing functionality
- Document your reasoning for each change
- All existing tests should pass after your fixes
- Do not disable or modify **correct** tests to make them pass

Debugging tips and techniques:

- Add print statements to trace execution
- Use the Python debugger in PyCharm, VS Code, or the command line (`pdb`)
- Verify assumptions about data structures and types

Deliverable: For each fix you made:

- Test name(s) and original failure reason(s)
- Your analysis of the root cause
- Brief description of the specific code changes you made
- Verification that the fix works and the tests now pass (for example, a screenshot of the passing test suite)
- Any remaining issues, partial solutions or challenges you faced
- Include a link to a GitHub branch or pull request that shows your changes, and provide a written explanation in your submission

TASK 5: IMPLEMENTING A TAGGING FEATURE

Your next challenge is to extend the CAF project by implementing a feature inspired by git's tagging system. This will require you to research what a git "tag" is, understand its purpose and implementation, and then design and build a similar feature for the CAF project. Focus on writing robust, documented code that is consistent with the rest of the project.

Objective: Research, design, and implement a git-style tagging system for the CAF project.

Research phase: Research git's tagging system and document:

- What are git tags and how are they different from branches and commits?
- Common use cases (releases, milestones, etc)
- Storage mechanism and relationship to commits

Minimum implementation requirements:

- **Data model:** Objects/data structures to represent tags
- **Storage:** File system representation (where/how tags are stored)
- **Operations:**
  - List tags: `caf tags`
  - Create tag: `caf create_tag <tag name> <commit-hash>`
  - Delete tag: `caf delete_tag <tag name>`

- **CLI integration:** Add tag commands to existing CLI structure
- **Tests:** Unit tests for tag operations and CLI functionality (separate logic and CLI tests)
- **Error handling:** Appropriate error messages for invalid operations

Design considerations:

- Follow existing code patterns and conventions
- Consider how tags interact with CAF's commit model
- Think about error cases and edge cases (duplicate tags, invalid commits, etc.)
- Not every edge case needs to be handled, but you should document any limitations or assumptions in your design
- Ensure your code is readable, maintainable, documented and properly formatted

Implementation guide:

1. Start with data structures and storage format
2. Implement core tag operations (create, list, delete)
3. Write tests for the core operations before integrating with the CLI
4. Add CLI command implementations
5. Write tests for the CLI commands to ensure they work as expected
6. Integrate your code with the existing CLI system in a way that feels natural and consistent, and test the CLI commands manually
7. Cover your implementation by writing tests for edge cases and error conditions that you can think of
8. Document your code
9. There is no need to test that your core objects work (e.g., tags), since if you followed the existing patterns, they should not be doing much

Deliverable:

- Research summary (one paragraph)
- Summary of your design and implementation approach (one-two paragraphs)
- If you encounter challenges, describe how you addressed them or what help you needed
- Complete implementation with tests: link to a branch or pull request in your forked repository
- Your branch or pull request should be clearly organized and easy to navigate

GRADING

Your assignment will be evaluated on both absolute quality and relative performance. We assess the technical accuracy of your work and the clarity and depth of your explanations. Additionally, your submission will be compared against other students' work to ensure fair and consistent grading standards.

Grading is based on your code and written submission with the following criteria:

Task	**Points**	**Key Criteria**
Task 1: File analysis	20	Completeness, insight, clarity
Task 2: Architecture diagram	20	Accuracy, clarity, completeness
Task 3: Environment setup	-	-
Task 4: Debugging fixes	20	Correctness, clear explanations, logical changes
Task 5: Tag implementation	30	Working feature, thoughtful design, comprehensive tests
Code quality and style	10	Follows patterns, readable, well-documented
Total	**100**	

SUBMISSION CHECKLIST

Required deliverables:

- ☐ **File analysis:** Table or list summarizing the purpose, role, and relationships of each significant file or group of files
- ☐ **Architecture diagram:** Clear visual map with brief explanation
- ☐ **Bug fixes:** Detailed explanation of each fix with reasoning and link to your forked repository
- ☐ **Tag feature:** Research, design considerations, link to your implementation branch
- ☐ **Reflection (optional):** 1–2 paragraphs on what you learned, challenges faced, and how you overcame them. You can also include any additional insights or thoughts on the project and your experience with it, and what you would like to explore further in this class.

Submission format:

- ☐ Single PDF document with all written work, except for diagrams which can be in another PNG, JPG, or PPTX file, or part of the PDF
- ☐ PDF file should link to your GitHub repository for code changes where applicable in a clear way
- ☐ Clear section headers matching assignment structure
- ☐ Readable fonts, appropriate formatting and general presentation
- ☐ Include your names and ID numbers at the top of the document
- ☐ Submissions should be in English or Hebrew. The level of your English will not affect your grade, so please give it your best effort because technical writing is usually done in English

CONCLUSION

You've now journeyed through the complete lifecycle of working with a real-world codebase: from initial exploration and understanding, through debugging and problem-solving, to extending the system with new functionality.

These skills—reading unfamiliar code, mapping system architecture, debugging methodically, and implementing features thoughtfully—are fundamental to professional software development. The experience of working with CAF mirrors what you'll encounter in industry: inheriting existing systems, understanding their design decisions, and evolving them to meet new requirements.

Remember that the best engineers are not those who never encounter problems, but those who approach problems systematically, communicate their findings clearly, and persist through challenges with creativity and determination.

The foundation you've built here will serve you well throughout this course and in your future work with complex systems. We hope you are proud of what you have accomplished.

Well done! You've not just explored the temple—you've added to its architecture.