Reflection on Aider-Assisted Development

Project: React Calculator Enhancement

Most Effective Techniques

1. Breaking Tasks into Smaller Steps
    - Example: Implementing calculation history required splitting into:
        - Adding **history** state in **App.js**
        - Updating **calculate.js** to store expressions/results
        - Styling the history panel in **Display.css**
    - Result: Aider provided accurate suggestions when tasks were isolated (e.g., **"Add a scrollable history div to Display.js"**).
2. Iterative Refinement
    - Initial history panel had poor readability. Refining with:

        "Style the history panel with a fixed sidebar."
    - Resulted in cleaner UI updates like:

        .history li:nth-child(even) { background-color: #f0f0f0; }

        .history { max-height: 150px; overflow-y: auto; }
3. Leveraging Keyboard Support
    - Mapping keys like **Enter** and **Escape** required explicit prompts:

        "Add keyboard event listeners for operators and equals key."
    - Aider generated functional code but needed manual adjustments for edge cases (e.g., preventing **Backspace** from clearing history).

Limitations Encountered

1. Context Loss
    - Aider forgot earlier changes (e.g., history persistence logic) when switching files.
    - Solution: Re-added files to context with **/add** before refining.

2. Complex State Logic
    - Dark mode implementation initially failed to persist via **localStorage**.
    - Manual fix required:

      componentDidMount() {

          const savedTheme = localStorage.getItem('darkMode') === 'true';

          this.setState({ darkMode: savedTheme });

      }
3. Babel Configuration Gaps
    - Error with **??** operator in **calculate.js** required manual Babel config updates:

          "plugins": ["@babel/plugin-proposal-nullish-coalescing-operator"]

Comparison to Traditional Workflow

Aspect	**Aider Workflow**	**Traditional Coding**
Speed	Accelerated boilerplate (e.g., tests, keyboard mappings).	Slower initial setup but more predictable flow.
Accuracy	Struggled with complex logic (e.g.,**localStorage**,**eval()**security fixes).	Full control over every line of code.
Learning Curve	Required precise prompts (e.g.,**"Fix history to use localStorage"**).	Familiar but time-consuming for repetitive tasks.

Suggestions for Aider

1. Better Context Retention : Automatically track dependencies between files (e.g., **App.js** ↔ **calculate.js**).
2. Linting Integration : Flag syntax errors (e.g., missing imports) before suggesting commits.
3. TDD Support : Generate tests *before* implementation code (e.g., **"Write Jest tests for history persistence"**).
4. Security Awareness : Warn against unsafe practices like **eval()** in **calculate.js**.

Conclusion

Aider significantly sped up repetitive tasks (e.g., test creation, keyboard support) but required manual intervention for complex logic (e.g., **localStorage**, Babel configs). Combining Aider's rapid prototyping with traditional debugging yielded the best results. Future projects will benefit from stricter prompt engineering and proactive use of **/diff** to validate changes.

Key Takeaways

- Prompt Precision : Clear, focused commands (e.g., **"Refactor history to use CSS classes"**) improved Aider's accuracy.
- Hybrid Approach : Use Aider for scaffolding, then refine manually for edge cases.
- Documentation : Keep notes on effective command patterns (e.g., **/add** → **/ask** → **/diff**).

This project demonstrated that Aider is a powerful tool for accelerating development when used strategically, but it still requires human oversight for security, maintainability, and complex state logic.