Mai Waheed 202200556

Reflection – Enhancing a React Calculator with Aider AI

It was an empowering and eye-opening experience to work with Aider AI as a pair programming partner on this project to see how large language models can assist in real software development. What follows is a consideration of the best practices, limitations, and how this experience compares to traditional workflows.

Most Effective Aider Techniques Discovered

One of the strongest Aider features was its awareness of context: the ability to see structure and interdependence between React components when importing multiple related files using /add. The /ask command let me write natural-language requests to modify UI and logic together in multiple files, speeding development significantly. Using /diff before /commit also allowed me to check changes securely before committing them, cutting debugging time.

Another helpful method was breaking tasks down incrementally (e.g., "Add a dark mode toggle" then "Now persist the theme using local Storage"). The incremental, step-by-step asking allowed for cleaner and more maintainable code development.

Limitations Encountered

While it was good, Aider had some flaws. One of those was that Aider occasionally assumed something or added a logic flaw where input was not clear-cut. Scientific computations such as tan would occasionally provide edge-case bugs in their output by assuming the input. Once again, if more files were opened at the same time and were edited together, then Aider occasionally misplaced UI layout interactions (especially CSS and JSX form), for which the programmer had to clean manually.

Model switching was also a source of friction—Aider would switch to a lower-capacity model from time to time, reducing the quality of suggestions unless explicitly requested using --model. This made the process somewhat flaky when building more sophisticated UI capabilities.

Comparison of Traditional Coding Workflow

Relative to individual coding, Aider significantly accelerated boilerplate and minor refactoring tasks. Instead of executing boilerplate by changing numerous elements at risk of syntax or compatibility error, I was able to dictate the intention and receive production-level code. Yet debugging and fiddling remained a manual task—specifically for UI glamour and app functionality in corner cases.

In contrast, a traditional workflow gives more precise control and avoids unexpected behaviors but is slower for iterative prototyping or interface enhancements.

Suggestions for Improving Aider

- **Visual diff preview**: A visual preview of component render changes (like a mini JSX viewer) would help when modifying UI.
- **Error simulation/testing hook**: Integrating inline testing or a dry-run feature (/simulate) would let developers validate logic before actual commits.
- **Scoped context memory**: Better control over which files stay in active memory could reduce context confusion when working on large projects.

Overall, Aider proved to be a capable assistant, especially for frontend enhancements and repetitive code updates. With small improvements, it could become a staple tool in modern development workflows, especially for students and solo developers.