Using Deepseek R1:
Prompt 1: I need help creating a Racket program that can read a CSV file of tasks with dependencies, validate them for circular and missing dependencies, and produce an execution order with timing information. The program should use functional programming concepts like the Either monad for error handling.

Prompt 2:
Can you show me how to implement topological sort in Racket for dependency resolution? It should handle task dependencies where each task has a name, duration, and list of prerequisite tasks.

Prompt 3:
How should I modify the program to properly validate task dependencies and detect both circular dependencies and missing dependencies? The error messages should clearly indicate which tasks have problems.

Prompt 4:
Make the program interactive so it repeatedly asks the user for input filenames until they choose to exit. It should handle both relative and absolute file paths.

Prompt 5:
The program fails when trying to open CSV files with error 'cannot open input file'. How can I make it handle file paths more robustly and show better error messages?

Prompt 6:
I need to modify my Racket dependency resolver's topological sort implementation to naturally produce output that matches the example's specific task ordering while maintaining correctness for all input files.
Generated Code:

racket
Copy
```
;; Either monad for error handling
(struct left (value) #:transparent)
(struct right (value) #:transparent)

(define (either-bind x f)
  (match x
    [(left err) (left err)]
    [(right val) (f val)]))

(define (either-return x)
  (right x))
```

```
(define (fail msg)
  (left msg))

(define (succeed val)
  (right val))


(define (topological-sort tasks)
  (define (visit task-name visited result graph)
    (if (member task-name visited)
        (values visited result)
        (let ([deps (task-dependencies (find-task-by-name task-name graph))])
          ;; Process dependencies in reverse order to get DFS-like behavior
          (let-values ([(new-visited new-result)
                        (let loop ([remaining (reverse deps)] [v visited] [r result])
                          (if (null? remaining)
                              (values v r)
                              (let-values ([(v1 r1) (visit (car remaining) v r graph)])
                                (loop (cdr remaining) v1 r1))))])
            (values (cons task-name new-visited)
                    (cons (find-task-by-name task-name graph) new-result))))))

  ;; Process tasks in reverse input order to match example behavior
  (let loop ([remaining (reverse tasks)] [visited '()] [result '()])
    (if (null? remaining)
        (reverse result)
        (let ([task-name (task-name (car remaining))])
          (if (member task-name visited)
              (loop (cdr remaining) visited result)
              (let-values ([(new-visited new-result) (visit task-name visited result tasks)])
                (loop (cdr remaining) new-visited new-result)))))))


(define (dependency-resolver filename)
  (if (file-exists? filename)
      (let* ([csv-data (parse-csv-file filename)]
             [tasks (build-task-graph csv-data)])
        (either-bind (validate-tasks tasks)
                     (lambda (valid-tasks)
                       (either-bind (check-circular-dependencies valid-tasks)
                                    (lambda (checked-tasks)
                                      (let ([schedule (schedule-tasks checked-tasks)])
                                        (let ([total-time (apply max (map (lambda (x) (+ (car x) (caddr x)))
schedule))])
```

```
                                (right (list schedule total-time))))))))
    (left (format "File not found: ~a" filename))))

(define (main-loop)
  (display "Enter filename (or 'quit' to exit): ")
  (let ([input (string-trim (read-line))])
    (cond
      [(equal? input "quit") (displayln "Goodbye!")]
      [(string=? input "") (begin (displayln "Please enter a filename")
                                  (main-loop))]
      [else
       (let ([full-path
              (if (absolute-path? input)
                  input
                  (build-path (current-directory) input))])
         (format-output (dependency-resolver full-path))
         (newline)
         (main-loop))])))

(define (validate-tasks tasks)
  (let ([task-names (map task-name tasks)])
    (let loop ([remaining tasks] [errors '()])
      (if (null? remaining)
          (if (null? errors)
              (succeed tasks)
              (fail (reverse errors)))
          (let* ([current (car remaining)]
                 [missing (filter (lambda (d) (not (member d task-names)))
                                  (task-dependencies current))])
            (if (null? missing)
                (loop (cdr remaining) errors)
                (loop (cdr remaining)
                      (cons (format "Task '~a' depends on missing tasks: ~a"
                                    (task-name current) missing)
                            errors)))))))))
```

Reflection:

The LLM I used was the Deepseek R1 model. Since this is for a class assignment, I would not use this if I were programming something I wanted to make money off of or something that required privacy. This LLM is very good at programming and very good at explaining things so I can understand the program. But the prompt engineering in Deepseek versus ChatGPT is much harder—you have to be specific and short. I don't think it hindered me because I didn't think I could program this on my own, as it was very complicated. I think prompt engineering would help, because based on the LLM, you're gonna need to use different prompts for the right

response you want. The big picture I am taking from using LLMs in this class is that you need to be able to understand the stuff that is coming out of the LLM. If you don’t understand, you're gonna get errors you can't describe or understand well enough to put back into the LLM. The one take is: pay attention to the parts in class, because those parts will come back to you when using the LLM—and then you will actually learn.