Lecture 07: Build Code Reviewer

🎯 What Are We Building?

We're building an **AI-powered Code Review Agent** that can:

1. **Scan** entire folders of code (HTML, CSS, JavaScript)
2. **Read** all files automatically
3. **Find** bugs, security issues, and bad code
4. **Fix** the problems by rewriting the files
5. **Report** what was fixed

Think of it as your personal code assistant that reviews your projects and fixes issues automatically!

🧠 Why This Project?

Real-World Problem:

- Developers make mistakes (bugs, security holes, bad practices)
- Manually reviewing code is slow and boring
- We need an automated assistant

What You'll Learn:

1. How to work with files and folders programmatically
2. How AI can interact with your computer safely
3. Function calling - giving AI "tools" to use
4. Building practical CLI applications

📂 Understanding File Systems (First Principles)

What is a File System?

Imagine your computer is a huge library:

- **Files** = Books (index.html, app.js, styles.css)
- **Folders** = Shelves/Sections (src/, components/, utils/)
- **Paths** = Addresses to find books

Example structure:

```
my-website/ ├── index.html ├── styles.css ├── js/ │ ├── app.js │ └── utils.js
└── images/ └── logo.png
```

🗺️ Understanding Paths

What is a Path?

A path is the **address** to reach a file or folder.

Real-World Analogy:

```
Your home address: Country → State → City → Street → House Number Computer pa
th: Drive → Folder → Subfolder → File
```

Two Types of Paths:

1. Absolute Path (Full Address)

Starting from the root of your computer:

```
/home/rohit/projects/my-website/index.html C:\Users\Rohit\projects\my-website
\index.html
```

Like saying: "Start from your country, go to your state, then city..."

2. Relative Path (From Current Location)

Starting from where you are now:

```
js/app.js (go into js folder, find app.js) ../images/logo.png (go up one leve
l, then into images)
```

Like saying: "From where you're standing, go to the next room..."

🛠️ Node.js Modules: `fs` vs `path`

The Key Difference:

`path` Module	`fs` Module
Works with **ADDRESSES** (strings)	Works with **ACTUAL FILES** (disk)
Like using a **GPS**	Like being a **delivery person**
Just text manipulation	Actually touches files
Fast (no disk access)	Slower (reads/writes disk)
Doesn't need file to exist	File must exist

📍 The `path` Module

Purpose:

Works with file/folder paths as **text strings**.

Think of it as:

A GPS that helps you build and understand addresses, but never actually goes to those locations.

When to Use `path`:

- Building file paths
- Getting file extensions
- Extracting filenames
- Converting between path formats

Why It's Needed:

Problem: Different operating systems use different separators:

```
Windows: C:\Users\Rohit\project\src\index.js Linux: /home/rohit/project/src/index.js Mac: /Users/rohit/project/src/index.js
```

Solution: `path` handles this automatically!

Mental Model:

```
Question: Do I need to build/analyze an ADDRESS? Answer: Use `path` Examples: - Combine "src" + "components" + "Button.js" → Use path - Get ".js" from "Button.js" → Use path - Get filename from full path → Use path - Build full path from pieces → Use path
```

📁 The `fs` Module

Purpose:

Interacts with **actual files and folders** on your disk.

Think of it as:

A delivery person who actually goes to addresses, opens mailboxes, delivers packages, and checks what's there.

When to Use `fs`:

- Reading file contents

- Writing to files
- Checking if file exists
- Listing files in a folder
- Creating/deleting files

Mental Model:

```
Question: Do I need to ACCESS actual files on disk? Answer: Use `fs` Example
s: - Read what's inside a file → Use fs - Check if file exists → Use fs - Lis
t all files in a folder → Use fs - Write data to a file → Use fs - Check if s
omething is a file or folder → Use fs
```

🔄 How They Work Together

Common Pattern:

Step 1: Use `path` to build the address

```
Build: "src" + "components" + "Button.js" Result: "src/components/Button.js"
```

Step 2: Use `fs` to actually access the file

```
Read: Read the content of "src/components/Button.js" Result: "import React fr
om 'react'..."
```

Analogy:

Planning a Trip:

- **Using `path`** = Looking at a map, planning your route
  - "Which roads connect these cities?"
  - "What's the distance?"
  - Works even if you haven't traveled yet!
- **Using `fs`** = Actually driving the route
  - "Is there traffic right now?"
  - "Is this road closed?"
  - You MUST be on the road to know!

🎯 Quick Decision Guide

Use `path` when:

- ✅ Working with filenames/paths as text
- ✅ Building paths from pieces
- ✅ Extracting parts of paths (extension, filename, directory)
- ✅ Converting between path formats
- ✅ File doesn't need to exist

Use `fs` when:

- ✅ Reading file contents
- ✅ Writing to files
- ✅ Checking if files exist
- ✅ Listing directory contents
- ✅ Creating/deleting files
- ✅ Checking file properties (size, type, permissions)

🌲 Understanding Nested File Structures

Visual Example:

```
my-website/ ├── index.html ├── css/ │ ├── main.css │ └── responsive.css ├── j
s/ │ ├── app.js │ ├── utils/ │ │ ├── api.js │ │ └── helpers.js │ └── componen
ts/ │ ├── Header.js │ └── Footer.js └── images/ └── logo.png
```

How to Read All Files:

Concept: Recursion (going deeper and deeper)

Process:

1. Start at `my-website/`
2. Look inside: find `index.html`, `css/`, `js/`, `images/`
3. For each **folder**, go inside and repeat
4. For each **file**, save its path

Like exploring a building:

- See a door (folder)? Go through it
- See an object (file)? Note it down
- Keep exploring until no more doors

🤖 AI Function Calling Concept

The Problem:

We want AI to interact with files, but we can't let it run random commands on our computer (dangerous!).

The Solution:

Give AI a **menu of specific actions** it can request.

How It Works:

Step 1: We define "tools" (functions AI can use)

```
Available Tools: - list_files: Get all files in a folder - read_file: Read a
file's content - write_file: Write fixed content to a file
```

Step 2: AI requests to use a tool

```
AI: "I want to call list_files with directory='./src'"
```

Step 3: WE execute the tool safely

```
We: Execute listFiles('./src') We: Return results to AI
```

Step 4: AI uses the result and continues

```
AI: "Now I want to call read_file with file_path='./src/app.js'"
```

Restaurant Analogy:

Menu = Tools we define

```
Available Dishes: 1. list_files 2. read_file 3. write_file
```

Customer = AI

```
"I'd like list_files with directory=./src" "Now I'd like read_file with file_
path=app.js"
```

Kitchen = Our code

```
We cook each order safely We control what actually happens
```

🔒 Why Function Calling is Safe

What AI CAN'T Do:

- ❌ Run any command it wants
- ❌ Delete your entire system
- ❌ Access files we didn't allow
- ❌ Do anything outside the tools we gave it

What AI CAN Do:

- ✅ Only call functions we defined
- ✅ Only with parameters we specified
- ✅ We control the implementation
- ✅ We can validate every action

🎯 Project Flow

How Our Agent Works:

1. User Input:

```
User: "Review the code in ./my-project"
```

2. AI Planning:

```
AI thinks: "I need to: 1. List all files 2. Read each file 3. Find issues 4.
Fix them 5. Report what I did"
```

3. AI Action Loop:

```
AI: Call list_files("./my-project") We: Execute → Return [app.js, utils.js, i
ndex.html] AI: Call read_file("app.js") We: Execute → Return file content AI:
Analyzes content, finds bugs AI: Call write_file("app.js", fixed_content) We:
Execute → File updated ... repeats for all files ... AI: Returns final report
as text We: Display report to user
```

📝 Key Concepts Summary

File System:

- Files and folders organized in a tree structure
- Paths are addresses to locate files
- Can be absolute (full address) or relative (from current location)

`path` Module:

- GPS for file addresses
- Works with text/strings
- Fast, doesn't touch disk
- Cross-platform path handling

`fs` Module:

- Delivery person for files
- Actually reads/writes disk
- Slower, needs files to exist
- Real file operations

AI Function Calling:

- Give AI a menu of tools
- AI requests actions
- We execute safely
- Controlled, secure interaction

Recursion:

- Going deeper into nested structures
- Process folders → go inside → repeat
- Like exploring a building room by room

💡 Why This Matters

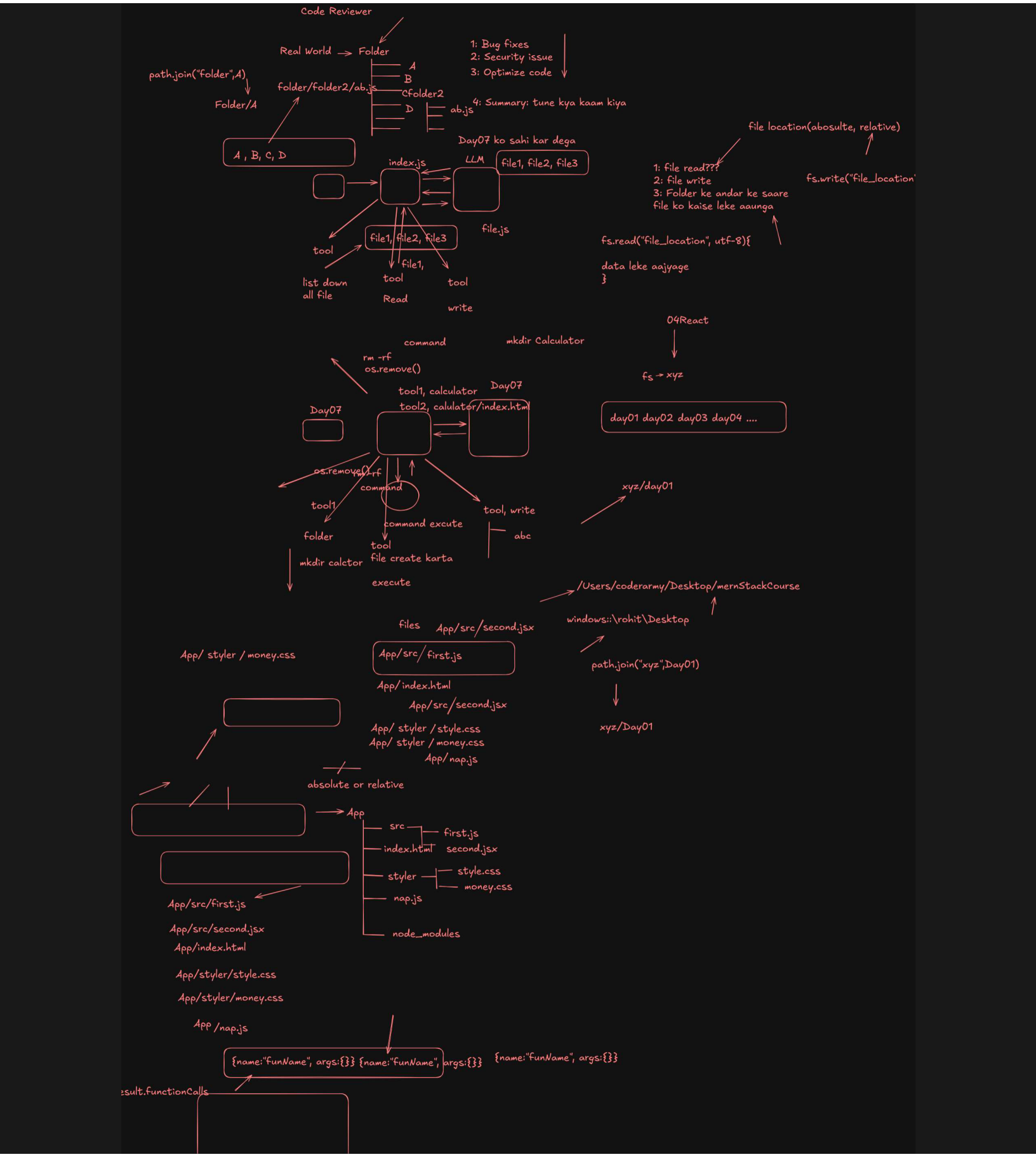
For Teaching at Coder Army:

This project teaches:

1. **First Principles:** Understanding WHY before HOW
2. **Real Problem Solving:** Automating tedious tasks
3. **AI Integration:** Practical use of LLMs
4. **System Design:** Building safe, scalable tools
5. **File Operations:** Essential skill for any developer

Career Applications:

- Build automation tools
- Create developer productivity tools
- Understand how AI coding assistants work
- Learn safe AI integration patterns
- Practice with real-world file operations


Code Reviewer
Real World → Folder
A
B
Cfolder2
D
ab.js
path.join("folder",A)
Folder/A
folder/folder2/ab.js
1: Bug fixes
2: Security issue
3: Optimize code
4: Summary: tune kya kaam kiya
A , B, C, D
Day07 ko sahi kar dega
index.js
LLM
file1, file2, file3
file.js
tool
list down all file
file1, file2, file3
file1,
tool
Read
tool
write
file location(abosulte, relative)
1: file read???
2: file write
3: Folder ke andar ke saare file ko kaise leke aaunga
fs.write("file_location"
fs.read("file_location", utf-8){
data leke aajyage
}
04React
fs → xyz
day01 day02 day03 day04 ....
command
mkdir Calculator
rm -rf
os.remove()
tool1, calculator
tool2, calulator/index.html
Day07
Day07
os.remove() -rf
command
tool1
folder
mkdir calctor
command excute
tool
file create karta
execute
tool, write
abc
xyz/day01
/Users/coderarmy/Desktop/mernStackCourse
windows::\rohit\Desktop
files
App/src/second.jsx
App/ styler / money.css
App/src/ first.js
App/index.html
App/src/second.jsx
App/ styler / style.css
App/ styler / money.css
App/ nap.js
path.join("xyz",Day01)
xyz/Day01
absolute or relative
App
src
first.js
second.jsx
index.html
styler
style.css
money.css
nap.js
node_modules
App/src/first.js
App/src/second.jsx
App/index.html
App/styler/style.css
App/styler/money.css
App /nap.js
{name:"funName", args:{}} {name:"funName", args:{}}
{name:"funName", args:{}}
esult.functionCalls


1: 2500
extension vs code