Coding & Development / JavaScript

Write JavaScript functions from natural language descriptions, including JSDoc comments and error handling.
Difficulty: Intermediate
Model: GPT-4 / Claude / Gemini
Use Case: Code Generation, JavaScript Development, Utility Functions
Updated: May 2026
Why This Prompt Exists
Most developers waste time writing boilerplate functions from scratch when they could describe intent.

You get:

- functions missing JSDoc comments (hard to maintain)
- no type checking (TypeScript would catch errors)
- poor error handling (crashes on edge cases)
- inconsistent naming conventions
- functions that work but aren’t reusable

But a function is not just code.

It is a reusable unit with documentation, type safety, and error handling.

- JSDoc: what it does, params, returns, throws
- Parameter validation: check types and presence
- Error handling: try/catch for edge cases
- Default parameters: sensible defaults where appropriate

Without documentation and error handling, functions are fragile.

This framework forces AI to generate production-ready JavaScript functions.

The Prompt

```
Assume the role of a senior JavaScript developer who writes clean, documented, production-ready functions.

Your task is to generate a JavaScript function from a natural language description.

Generate:

1. FUNCTION SIGNATURE
   - Function name (descriptive, camelCase)
   - Parameters with default values
   - Return type (in JSDoc)

2. JSDOC COMMENT
   - Brief description
   - @param for each parameter (type and description)
   - @returns description
   - @throws for error conditions

3. FUNCTION BODY
   - Parameter validation
   - Main logic
   - Error handling (try/catch)

4. EXAMPLE USAGE
```

   - How to call the function
   - Expected output

5. UNIT TEST (Jest or Mocha)
   - Test normal case
   - Test edge case
   - Test error case

INPUTS:

Function Description:
[WHAT SHOULD THE FUNCTION DO?]

Input Parameters (names and types):
[LIST]

Return Value (type and description):
[DESCRIBE]

Edge Cases to Handle (if any):
[LIST]

Error Conditions to Handle:
[E.G., "Invalid input," "Empty array," "Network error"]

Environment:
[NODE.JS / BROWSER / BOTH]

RULES:

```
- Use camelCase for function names
- Include JSDoc comments for all public functions
- Validate parameters (check types, presence)
- Handle edge cases gracefully (don't crash)
- Use const/let (not var)
- Use arrow functions where appropriate
- Add try/catch for async operations
```

How To Use It

- Be specific about parameter types (string, number, array of objects).
- List edge cases you know will happen (null, undefined, empty arrays).
- The generated unit test is a starting point — expand it with your own cases.
- Run the function through ESLint after generation.
- Add TypeScript types if your project uses TypeScript.

Example Input

**Function Description:** Calculate the average of an array of numbers. Return 0 if the array is empty.

**Input Parameters:** numbers (array of numbers)

**Return Value:** number (the average)

**Edge Cases to Handle:** Empty array, array with one item, array with negative numbers

**Error Conditions to Handle:** If input is not an array, throw TypeError

**Environment:** NODE.JS

Why It Works
Most generated functions lack documentation and error handling.

This framework improves outcomes by forcing:

- JSDoc comments (maintainability)
- parameter validation (robustness)
- error handling (resilience)
- unit tests (correctness)
- example usage (usability)

Great JavaScript functions don’t just work — they’re documented, tested, and handle errors gracefully.

## Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, JavaScript frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

### Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also The JavaScript Async/Await Converter