Coding & Development / Debugging

Analyze code for common memory leak patterns (unclosed files, event listeners, global variables) and suggest fixes.
Difficulty: Advanced
Model: GPT-4 / Claude / Gemini
Use Case: Memory Leaks, Performance, Production Issues
Updated: May 2026
Why This Prompt Exists
Most memory leaks go unnoticed until production — then cause crashes and slow performance.

You get:

- memory usage that grows over time (never stabilizes)
- crashes after hours or days of uptime
- no idea which code is leaking
- event listeners that never get removed
- global variables accumulating data

But memory leaks are not mysterious.

They follow predictable patterns.

- Unclosed resources: files, database connections, network sockets
- Event listeners: attached but never removed
- Global variables: accidental globals, cached data that never expires
- Closures: variables retained longer than needed
- Timers: setInterval without clearInterval

Without detection, leaks degrade performance.

This framework forces AI to spot memory leak patterns.

The Prompt

```
Assume the role of a performance engineer who detects memory leak
patterns.

Your task is to analyze code for memory leaks.

Generate:

1. LEAK PATTERNS IDENTIFIED
   - List of potential leaks found

2. FOR EACH LEAK:
   - Location (file and line)
   - Why it's a leak
   - Impact (memory growth rate)

3. FIX CODE
   - Specific changes to prevent the leak

4. TESTING SUGGESTIONS
   - How to verify the leak is fixed

5. MONITORING RECOMMENDATIONS
   - How to detect this leak in production
```

```
INPUTS:

Code (paste or describe):
[PASTE CODE OR DESCRIBE BEHAVIOR]

Environment:
[NODE.JS / BROWSER / PYTHON / OTHER]

Observed Symptoms:
[E.G., "Memory grows 100MB per hour," "Crashes after 3 days"]

Known Leak Patterns to Check:
[GLOBALS / EVENT LISTENERS / CLOSURES / TIMERS / UNCLOSED RESOURCES]

RULES:
- Check for event listeners added but never removed
- Check for setInterval without clearInterval
- Check for global variables accumulating data
- Check for closures retaining large objects
- Check for unclosed files, connections, sockets
- Suggest heap snapshot analysis for confirmation
- Consider the environment (browser vs. Node.js has different
patterns)
```

How To Use It

- Describe the symptoms (memory growth rate, crash timing).
- Paste suspicious code sections (not the whole codebase).
- Check event listeners first — most common leak in browsers.
- Check timers (setInterval without clearInterval) in Node.js.

- Use heap snapshots to confirm the leak after applying fixes.

Example Input

Code:

```
let cache = {};
function fetchData(id) {
if (cache[id]) return cache[id];
return fetch(`/api/data/${id}`)
.then(r => r.json())
.then(data => {
cache[id] = data;
return data;
});
}
setInterval(() => {
console.log('Cache size:', Object.keys(cache).length);
}, 60000);
```

Environment: NODE.JS

Observed Symptoms: Memory grows 200MB per day, cache never clears, eventually crashes after 2 weeks

Known Leak Patterns to Check: GLOBALS, TIMERS

Why It Works
Most memory leaks are subtle and go unnoticed.

This framework improves outcomes by forcing:

- leak pattern identification (detection)

- location pinpointing (diagnosis)
- specific fixes (resolution)
- testing suggestions (verification)
- monitoring recommendations (prevention)

Great memory leak detection doesn’t just find leaks — it teaches you how to prevent them.

Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, debugging 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 Concurrency Bug Analyzer