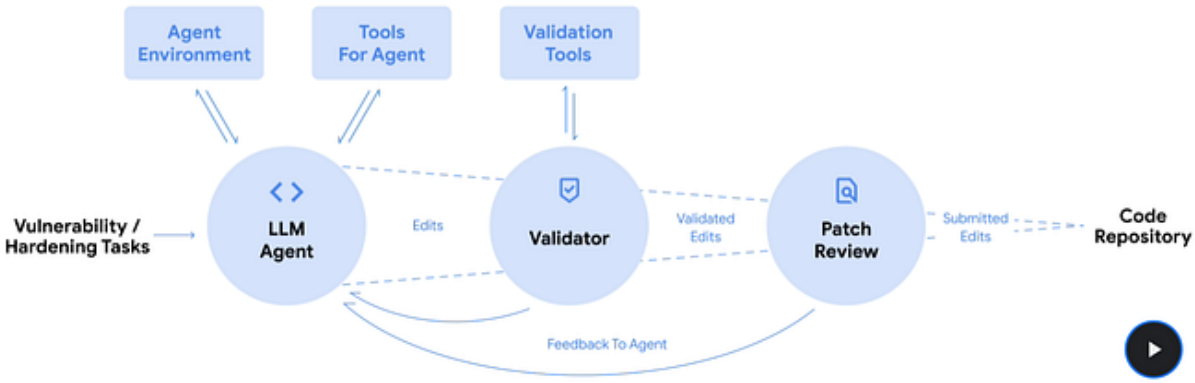
Google’s DeepMind CodeMender: The AI Agent That Finds and Fixes Software Bugs Automatically

Why Google Built CodeMender

Code Coup • 5 min read • 2025-10-10

um.com/coding-nexus/googles-deepmind-codemender-the-ai-agent-that-finds-and-fixes-software-bugs-automatically-a3d5



If you’ve ever spent a weekend chasing down a single bug - the kind that makes you question your life choices - you’ll likely appreciate what Google just released.

It’s called **CodeMender**, and it’s essentially an AI agent that can *autonomously identify and fix software vulnerabilities*.

It doesn’t just highlight issues like “hey, this could be unsafe.” It genuinely understands what’s wrong and automatically patches the problem.

Why Google Built CodeMender

Finding security bugs in large projects is challenging. Even with fuzzers, scanners, and static analysis tools, some vulnerabilities remain hidden for years. Google’s own tools, such as **OSS-Fuzz,** have identified many issues early; however, there is always more code to review, and humans can only do so much.

So, DeepMind built CodeMender - an AI that doesn’t get tired or distracted and can review millions of lines of code without blinking. In just six months, it’s already submitted **72 real security fixes** to open-source projects. That’s not a small feat; some of those projects have codebases over four million lines long.

How It Actually Works

At the core of CodeMender is a family of models called **Gemini Deep Think**. These models are designed for reasoning - not just autocomplete code, but to understand what the code is *doing*.

Here’s roughly how CodeMender works:

1. It scans your code and looks for weak spots.
2. It figures out what's actually causing the vulnerability.
3. It proposes a patch and reruns the code.
4. It ensures that everything still functions correctly and that no other components have malfunctioned.
5. A human engineer reviews the final patch before it goes upstream.

It's like having a developer who reads your code, reasons through it, writes the fix, and then double-checks their own work - only this one's made of math.

The agent traces the pointer as it's passed across multiple locations to build a complete picture of its scope.

Now let's find the callers of `GenerateOptimalTree` to see how the actual `bit_depths` array is allocated and passed.

<TOOL: source_code_browser>

```
// src/utils/huffman_encode_utils.c
 410: void VP8LCreateHuffmanTree(uint32_t* const histogram, int tree_depth_limit,
 411:                             uint8_t* const buf_rle, HuffmanTree* const huff_tree,
 412:                             HuffmanTreeCode* const huff_code) {
 413:   const int num_symbols = huff_code->num_symbols;
 ...
 416:   GenerateOptimalTree(histogram, num_symbols, huff_tree, tree_depth_limit,
 417:                        huff_code->code_lengths);
 ...
 420: }
```

<AGENT>

`GenerateOptimalTree` is called by `VP8LCreateHuffmanTree` at line 416. The `bit_depths` argument passed is `huff_code->code_lengths`.

Let's examine `VP8LCreateHuffmanTree` and the `HuffmanTreeCode` struct.

<TOOL: source_code_browser>

```
// src/utils/huffman_encode_utils.h
 32: typedef struct {
 33:   int       num_symbols;   // Number of symbols.
 34:   uint8_t*  code_lengths;  // Code lengths of the symbols.
 35:   uint16_t* codes;         // Symbol Codes.
 36: } HuffmanTreeCode;
```

<AGENT>
In VP8LCreateHuffmanTree, the histogram_size passed to GenerateOptimalTree is num_symbols, which comes from huff_code->num_symbols (line 413). The bit_depths argument is huff_code->code_lengths. This confirms that huff_code->code_lengths is intended to be an array of size huff_code->num_symbols. The HuffmanTreeCode struct has num_symbols and code_lengths. This fits the pattern for __counted_by. The code_lengths pointer points to an array of uint8_t elements, and the number of elements is stored in num_symbols.

... [SIMILAR REASONING FOR OTHER ACCESSES OF bit_depths]

The agent is asked to fix an -fbounds-safety error on a pointer

```
src/utils/huffman_encode_utils.c:146:15: error: array subscript on single pointer
'bit_depths' must use a constant index of 0 to be in bounds
  146 |     bit_depths[tree->value] = level;
      |     ~~~~~~~~~~~^~~~~~~~~~~~
```

The agent begins reasoning about the faulty pointer by examining where it's defined.

<AGENT>
... Examine the definition and usage of `bit_depths` in `src/utils/huffman_encode_utils.c` to understand its size and how it's accessed. ...

<TOOL: source_code_browser>

```
// src/utils/huffman_encode_utils.c
139: static void SetBitDepths(const HuffmanTree* const tree,
140:                          const HuffmanTree* const pool,
141:                          uint8_t* const bit_depths, int level) {
142:   if (tree->pool_index_left >= 0) {
143:     SetBitDepths(&pool[tree->pool_index_left], pool, bit_depths, level + 1);
144:     SetBitDepths(&pool[tree->pool_index_right], pool, bit_depths, level + 1);
145:   } else {
146:     bit_depths[tree->value] = level;
147:   }
148: }
```

<AGENT>
The error occurs within the `SetBitDepths` function. The `bit_depths` parameter is declared as `uint8_t*` `uint8_t* const bit_depths`. It's indexed using `tree->value`.

It then navigates the call stack to find the pointer's allocation site, determining its initial bounds.

Let's find the callers of `SetBitDepths` to understand how `bit_depths` is allocated and what its size should be.

<TOOL: source_code_browser>

```
// src/utils/huffman_encode_utils.c
169: static void GenerateOptimalTree(const uint32_t* const histogram,
170:                                 int histogram_size, HuffmanTree* tree,
171:                                 int tree_depth_limit,
172:                                 uint8_t* const bit_depths) {
...
242:       SetBitDepths(&tree[0], tree_pool, bit_depths, 0);
243:     } else if (tree_size == 1) {  // Trivial case: only one element.
244:       bit_depths[tree[0].value] = 1;
245:     }
...
249:       int max_depth = bit_depths[0];
250:       for (j = 1; j < histogram_size; ++j) {
251:         if (max_depth < bit_depths[j]) {
252:           max_depth = bit_depths[j];
253:         }
254:       }
```

<AGENT>
The function `SetBitDepths` is called from `GenerateOptimalTree` at line 242. In `GenerateOptimalTree`, `bit_depths` is a parameter declared as `uint8_t* const bit_depths`. It's accessed directly at line 244 and also passed to `SetBitDepths` at line 242. It's also read in the loop starting at line 249. The index `j` goes up to `histogram_size`. This strongly suggests that `bit_depths` points to an array of size `histogram_size`.

The agent traces the pointer as it's passed across multiple locations to build a complete picture of its scope.

```
410: void VP8LCreateHuffmanTree(uint32_t* const histogram, int tree_depth_limit,
411:                            uint8_t* const buf_rle, HuffmanTree* const huff_tree,
412:                            HuffmanTreeCode* const huff_code) {
413:   const int num_symbols = huff_code->num_symbols;
...
416:   GenerateOptimalTree(histogram, num_symbols, huff_tree, tree_depth_limit,
417:                       huff_code->code_lengths);
...
420: }
```

<AGENT>
`GenerateOptimalTree` is called by `VP8LCreateHuffmanTree` at line 416. The `bit_depths` argument passed is `huff_code->code_lengths`.

Let's examine `VP8LCreateHuffmanTree` and the `HuffmanTreeCode` struct.

<AGENT>
`GenerateOptimalTree` is called by `VP8LCreateHuffmanTree` at line 416. The `bit_depths` argument passed is `huff_code->code_lengths`.

Let's examine `VP8LCreateHuffmanTree` and the `HuffmanTreeCode` struct.

<TOOL: source_code_browser>

```
// src/utils/huffman_encode_utils.h
32: typedef struct {
33:   int       num_symbols;   // Number of symbols.
34:   uint8_t*  code_lengths;  // Code lengths of the symbols.
35:   uint16_t* codes;         // Symbol Codes.
36: } HuffmanTreeCode;
```

<AGENT>
In `VP8LCreateHuffmanTree`, the `histogram_size` passed to `GenerateOptimalTree` is `num_symbols`, which comes from `huff_code->num_symbols` (line 413). The `bit_depths` argument is `huff_code->code_lengths`. This confirms that `huff_code->code_lengths` is intended to be an array of size `huff_code->num_symbols`. The `HuffmanTreeCode` struct has `num_symbols` and `code_lengths`. This fits the pattern for `__counted_by`. The `code_lengths` pointer points to an array of `uint8_t` elements, and the number of elements is stored in `num_symbols`.

Example

Here's a real case: CodeMender discovered a **heap buffer overflow** in a program. Typically, that points to a single line of code where memory is written out of bounds. But when the AI started investigating, it realised the *actual* issue wasn't the buffer - it was poor stack handling during XML tag parsing.

That’s the kind of insight that takes a human developer hours, sometimes days, to identify. CodeMender just did it automatically, generated a small patch, ran it through its tests, and confirmed the bug was gone.

Handling the Hard Stuff

In another example, CodeMender addressed a complex **object lifetime issue** in C. This wasn’t a simple "change one line" fix - it required understanding and modifying part of a *custom code generator*. Nonetheless, it figured out the logic, applied the patch, and confirmed that the fix didn’t break anything else.

Making Code Safer Before Bugs Even Exist

What’s really clever is that CodeMender doesn’t just fix bugs - it can **rewrite code to make it safer**.

Take the **libwebp** library, for example. You may recall it from the iOS zero-click exploit that occurred a while back (CVE-2023-4863). CodeMender added compiler annotations `-fbounds-safety` that automatically prevent buffer overflows.

Basically, it tells the compiler, “Hey, keep an eye on this array’s size and don’t let anyone write past the end." Here’s an example:

```
// Before
bit_depths[tree->value] = level;

// After
__attribute__((counted_by(num_symbols)))
bit_depths[tree->value] = level;
```

A few words of annotation, and suddenly a whole category of memory bugs vanishes.

The Cool Part - It Fixes Its Own Mistakes

If CodeMender introduces an error while patching, such as a new compiler warning or test failure, it doesn’t just stop. Instead, it reads the compiler output, figures out what went wrong, and applies its own fix.

It even runs a system called an **LLM judge**, which verifies whether the new code behaves exactly like the original. If anything's wrong, CodeMender fine-tunes its patch until the behavior matches.

You could picture it like this:

```
while not tests_pass():
    patch = code_mender.make_patch()
    if build_fails(patch):
        code_mender.analyze_and_fix()
```

Pretty wild.

Still Reviewed by Humans (For Now)

Currently, every CodeMender patch undergoes human review before being merged. Google's team remains cautious - they aim to ensure the system is reliable and trusted before delegation to developers.

So far, the open-source community has been supportive of it. Some of CodeMender's patches have already been accepted and integrated into popular projects.

The Bigger Picture

DeepMind says it will soon publish research papers that delve into how CodeMender works under the hood - from its program analysis tools to its multi-agent architecture. Eventually, they aim to release CodeMender as a tool that any developer can use.

If that happens, we could be looking at the start of a new era where **software can help fix itself** - not just bugs, but deep, structural vulnerabilities that would typically take human experts weeks to identify.

Final Thoughts

We've had AI that can write code for a while now. But one that can *understand*, *reason about*, and *secure* code? That's a whole new level.

CodeMender won't replace developers - but it might finally replace those endless late-night debugging sessions that make us all question our career choices.

Author: [Your Name] - writing about the messy, fascinating intersection of AI and software for over a decade.

Other mentions by Author

• medium.com | Written by Code Coup