Markdown Remote Image Downloader

11 min read · Oct 22, 2025

Alain Airom (Ayrom) Following

Side project for myself 🫠

MarkDoun
)))tp/remote-site/image.png
()))tp/.remote-site/image.png
))))/.:images/image.png
-/images
unpdsh.com
Ratne Lotlen
python downloadimages.py /path-to docs
Dawnloded
go run . /path-to docs

Motivation on writing this code

As an avid blogger, I always strive to maintain local copies of my posts on my laptop. Much like countless other platforms, my content is often crafted in Markdown — a versatile and globally adopted format familiar to anyone who's encountered a `README.md` file.

A common scenario arises when publishing content online: uploaded images, like the one illustrated above, are typically hosted in remote repositories such as Amazon S3 or similar cloud storage. Consequently, when a blog post is downloaded for local safekeeping, these images are not truly localized; their references within the Markdown still point to external URLs, appearing something like `https://some-site/xxx/yyy/1*oqZ8EJKzNONj1FpeHIrebw.png`.

Simply downloading these images manually and placing them in a local folder, such as `/Users/theuser/folder/images/1*oqZ8EJKzNONj1FpeHIrebw.png`, introduces a new problem: these are absolute, hard-coded paths. This approach creates a fragile setup; should the parent folder be renamed or moved, these image links will inevitably break, rendering the images unviewable. The robust solution, therefore, is to transform these external URLs into **relative references**, such as `![](./images/image_name.png)`, ensuring the images are always found regardless of the document's location within its local directory structure.

The idea came to me that it might be nice to be able to do this in an automated way if there is a big number of files which we want to process in a batch way. That is the reason why I wrote a code in Python.

The 'Python' Code

Below is the code I wrote in Python which manages to copy embedded images in the "./image" folder under the parent folder and replaces the link to the image in the related Markdown file.

```
import os
import re
import requests
import argparse # New import for command-line arguments
from urllib.parse import urlparse, unquote

def process_markdown_files(root_dir):
    """
    Recursively scans a directory for Markdown files, finds remote images,
    downloads them to a central 'images' folder, and updates the links.
    """
    root_dir = os.path.abspath(root_dir)

    print(f"Starting image processing in folder: {root_dir}")

    images_dir = os.path.join(root_dir, "images")

    os.makedirs(images_dir, exist_ok=True)
    print(f"Ensuring target image directory exists: {images_dir}")

    IMAGE_REGEX = re.compile(r"!\[.*?\]\((https?://.*?)\)")

    MD_EXTENSIONS = ('.md', '.markdown')

    files_processed = 0
    images_downloaded = 0
    for dirpath, _, filenames in os.walk(root_dir):
        # Skip the 'images' folder itself to prevent endless recursion or error
        if dirpath == images_dir:
            continue

        for filename in filenames:
            if filename.lower().endswith(MD_EXTENSIONS):
                md_path = os.path.join(dirpath, filename)
                print(f"\nProcessing file: {md_path}")
                files_processed += 1

                try:
                    with open(md_path, 'r', encoding='utf-8') as f:
                        content = f.read()
                except Exception as e:
                    print(f"ERROR: Could not read file {md_path}. Skipping. Err
                    continue

                new_content = content
                matches = list(IMAGE_REGEX.finditer(content))

                if not matches:
                    print("  No remote images found.")
                    continue

                print(f"  Found {len(matches)} remote image(s).")

                for match in matches:
                    image_url = match.group(1)
                    print(f"  -> Found image URL: {image_url}")

                    try:
                        response = requests.get(image_url, stream=True)
                        response.raise_for_status()

                        parsed_url = urlparse(image_url)
                        original_filename = os.path.basename(unquote(parsed_url

                        if not original_filename or original_filename == '.':
                            import hashlib
                            url_hash = hashlib.sha256(image_url.encode('utf-8')
                            file_extension = '.' + response.headers.get('Conten
                            original_filename = url_hash + file_extension

                        image_filename = original_filename
                        image_dest_path = os.path.join(images_dir, image_filena

                        base, ext = os.path.splitext(image_filename)
                        counter = 1
                        while os.path.exists(image_dest_path):
                            image_filename = f"{base}_{counter}{ext}"
                            image_dest_path = os.path.join(images_dir, image_fi
                            counter += 1

                        with open(image_dest_path, 'wb') as image_file:
                            for chunk in response.iter_content(8192):
                                image_file.write(chunk)

                        print(f"    Downloaded successfully as: {image_filename
                        images_downloaded += 1

                        new_image_ref = f"./images/{image_filename}"

                        def replacement_fn(m):
                            if m.group(1) == image_url:
                                return m.group(0).replace(image_url, new_image_
                            return m.group(0)

                        new_content = IMAGE_REGEX.sub(replacement_fn, new_conte

                    except requests.exceptions.RequestException as req_err:
                        print(f"  -> ERROR downloading image {image_url}: {req_
                    except Exception as gen_err:
                        print(f"  -> An unexpected error occurred: {gen_err}")

                if new_content != content:
                    try:
                        with open(md_path, 'w', encoding='utf-8') as f:
                            f.write(new_content)
                        print(f"  File {filename} updated successfully with new
                    except Exception as e:
                        print(f"ERROR: Could not write to file {md_path}. Error
                else:
                     print("  No changes to file content were required.")

    print("\n" + "="*50)
    print("✨ Processing Complete! ✨")
    print(f"Total Markdown files checked: {files_processed}")
    print(f"Total images downloaded and links updated: {images_downloaded}")
    print(f"Images saved in: {images_dir}")
    print("="*50)

def main():
    parser = argparse.ArgumentParser(
        description="Recursively find remote images in Markdown files, download
        formatter_class=argparse.RawTextHelpFormatter
    )
    parser.add_argument(
        "root_directory",
        type=str,
        help="The path to the root folder containing the Markdown files.\nThe '
    )

    args = parser.parse_args()
    root_directory = args.root_directory

    if os.path.isdir(root_directory):
        process_markdown_files(root_directory)
    else:
        print(f"\nError: The directory '{root_directory}' does not exist or is
        print("Please provide a valid path as a command-line argument.")
        exit(1)

if __name__ == "__main__":
    main()
```

While the application operates flawlessly, its performance on a large number of files revealed a significant bottleneck, making it notably slow 🐌.

So I wrote a Go application doing the same thing!

The 'Go' Code

```
package main

import (
 "fmt"
 "io"
 "net/http"
 "net/url"
 "os"
 "path/filepath"
 "regexp"
 "strings"
)

const imagesFolderName = "images"

var imageRegex = regexp.MustCompile(`!\[.*?\]\((https?://.*?)\)`)

func isMarkdown(filename string) bool {
 ext := strings.ToLower(filepath.Ext(filename))
 return ext == ".md" || ext == ".markdown"
}

func downloadAndReplaceImage(match []string, imagesDir string) (string, error)
 fullMatch := match[0]
 imageURL := match[1]

 fmt.Printf("  -> Found image URL: %s\n", imageURL)
```

```
    parsedURL, err := url.Parse(imageURL)
    if err != nil {
     return "", fmt.Errorf("error parsing URL %s: %w", imageURL, err)
    }

    path := parsedURL.Path
    originalFilename := filepath.Base(path)
    originalFilename, _ = url.PathUnescape(originalFilename)

    if originalFilename == "." || originalFilename == "/" || originalFilename == "
     originalFilename = "downloaded_image"
    }

    imageFilename := originalFilename
    imageDestPath := filepath.Join(imagesDir, imageFilename)
    base := strings.TrimSuffix(imageFilename, filepath.Ext(imageFilename))
    ext := filepath.Ext(imageFilename)
    counter := 1

    for {
     _, err := os.Stat(imageDestPath)
     if os.IsNotExist(err) {
      break
     }
     imageFilename = fmt.Sprintf("%s_%d%s", base, counter, ext)
     imageDestPath = filepath.Join(imagesDir, imageFilename)
     counter++
    }

    resp, err := http.Get(imageURL)
    if err != nil {
     return "", fmt.Errorf("error downloading image %s: %w", imageURL, err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
     return "", fmt.Errorf("bad status code %d for URL %s", resp.StatusCode, image
    }

    if ext == "" {
     contentType := resp.Header.Get("Content-Type")
     if strings.HasPrefix(contentType, "image/") {
      fileExt := "." + strings.Split(contentType, "/")[1]
      imageFilename = imageFilename + fileExt
      imageDestPath = filepath.Join(imagesDir, imageFilename)
     }
    }

    outFile, err := os.Create(imageDestPath)
    if err != nil {
     return "", fmt.Errorf("error creating file %s: %w", imageDestPath, err)
    }
    defer outFile.Close()
    _, err = io.Copy(outFile, resp.Body)
    if err != nil {
     return "", fmt.Errorf("error saving image content to %s: %w", imageDestPath,
    }

    fmt.Printf("    Downloaded successfully as: %s\n", imageFilename)

    newImageRef := fmt.Sprintf("./%s/%s", imagesFolderName, imageFilename)
    newFullMatch := strings.Replace(fullMatch, imageURL, newImageRef, 1)

    return newFullMatch, nil
   }

   func processFile(mdPath string, rootDir string) (bool, int, error) {
    fmt.Printf("\nProcessing file: %s\n", mdPath)

    contentBytes, err := os.ReadFile(mdPath)
    if err != nil {
     return false, 0, fmt.Errorf("could not read file %s: %w", mdPath, err)
    }
    content := string(contentBytes)
    newContent := content
    imagesDownloaded := 0

    imagesDir := filepath.Join(rootDir, imagesFolderName)

    matches := imageRegex.FindAllStringSubmatch(content, -1)
    if len(matches) == 0 {
     fmt.Println("  No remote images found.")
     return false, 0, nil
    }

    fmt.Printf("  Found %d remote image(s).\n", len(matches))

    for _, match := range matches {
     newMatch, err := downloadAndReplaceImage(match, imagesDir)
     if err != nil {
      fmt.Printf("  -> ERROR: %v\n", err)
      continue
     }

     newContent = strings.Replace(newContent, match[0], newMatch, 1)
     imagesDownloaded++
    }

    if newContent != content {
     err = os.WriteFile(mdPath, []byte(newContent), 0644)
     if err != nil {
      return false, imagesDownloaded, fmt.Errorf("could not write to file %s: %w",
     }
     fmt.Printf("  File %s updated successfully with new image paths.\n", filepath
     return true, imagesDownloaded, nil
```

```go
    }

    fmt.Println("  No changes to file content were required.")
    return false, imagesDownloaded, nil
}

func processMarkdownFiles(rootDir string) error {
    fmt.Printf("Starting image processing in folder: %s\n", rootDir)

    imagesDir := filepath.Join(rootDir, imagesFolderName)

    if err := os.MkdirAll(imagesDir, 0755); err != nil {
        return fmt.Errorf("error creating images directory %s: %w", imagesDir, err)
    }
    fmt.Printf("Ensuring target image directory exists: %s\n", imagesDir)

    filesProcessed := 0
    imagesDownloadedTotal := 0
    filesUpdated := 0

    err := filepath.Walk(rootDir, func(path string, info os.FileInfo, err error) e
        if err != nil {
            fmt.Printf("Error accessing path %s: %v\n", path, err)
            return nil
        }

        if info.IsDir() {
            if path == imagesDir {
                return filepath.SkipDir
            }
            return nil
        }

        if isMarkdown(info.Name()) {
            filesProcessed++

            updated, downloadedCount, procErr := processFile(path, rootDir)
            if procErr != nil {
                fmt.Printf("  -> MAJOR ERROR processing file %s: %v\n", path, procErr)
            }
            imagesDownloadedTotal += downloadedCount
            if updated {
                filesUpdated++
            }
        }
        return nil
    })

    if err != nil {
        return err
    }

    fmt.Println("\n" + strings.Repeat("=", 50))
    fmt.Println("✨ Processing Complete! ✨")
    fmt.Printf("Total Markdown files checked: %d\n", filesProcessed)
    fmt.Printf("Total images downloaded and links updated: %d\n", imagesDownloaded
    fmt.Printf("Files updated: %d\n", filesUpdated)
    fmt.Printf("Images saved in: %s\n", imagesDir)
    fmt.Println(strings.Repeat("=", 50))

    return nil
}

// --- Main
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Usage: go run . <root_directory_path>")
        fmt.Println("Example: go run . /Users/alainairom/Devs/my-docs")
        os.Exit(1)
    }

    rootDirectory := os.Args[1]

    if _, err := os.Stat(rootDirectory); os.IsNotExist(err) {
        fmt.Printf("Error: The directory '%s' does not exist or is not accessible.\n"
        os.Exit(1)
    }

    if err := processMarkdownFiles(rootDirectory); err != nil {
        fmt.Printf("Application encountered a fatal error: %v\n", err)
        os.Exit(1)
    }
}
```

The performance of this iteration was outstanding; it blazed through the same document set with remarkable speed 💨. However, this entire project is also a deeply personal, recreational pursuit. After what felt like an eternity since my last dive into C development, the urge to revisit and challenge myself with the language became irresistible, culminating in the creation of the final application in C. 👀

The 'C' Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <regex.h>
```

```c
#include <curl/curl.h> // Requires libcurl, which is standard on macOS
#include <limits.h>    // For PATH_MAX

#define BUFFER_SIZE 4096
#define MAX_FILENAME_LEN 256
#define IMAGE_FOLDER "images"

struct MemoryStruct {
    char *memory;
    size_t size;
};

size_t write_memory_callback(void *contents, size_t size, size_t nmemb, void *u
    size_t realsize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;

    char *ptr = realloc(mem->memory, mem->size + realsize + 1);
    if (ptr == NULL) {
        printf("Out of memory (realloc failed)\n");
        return 0;
    }

    mem->memory = ptr;
    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0; // Null-terminate

    return realsize;
}

int download_and_save_image(const char *image_url, const char *images_dir, char
    CURL *curl_handle;
    CURLcode res;
    struct MemoryStruct chunk;

    chunk.memory = malloc(1);
    chunk.size = 0;

    curl_global_init(CURL_GLOBAL_ALL);
    curl_handle = curl_easy_init();

    if (!curl_handle) {
        fprintf(stderr, "Error: Could not initialize cURL.\n");
        curl_global_cleanup();
        return 0;
    }

    // Set cURL options
    curl_easy_setopt(curl_handle, CURLOPT_URL, image_url);
    curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, write_memory_callback)
    curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk);
    curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "C Markdown Image Processo
    curl_easy_setopt(curl_handle, CURLOPT_FOLLOWLOCATION, 1L); // Follow redire
```

```
        res = curl_easy_perform(curl_handle);

        if (res != CURLE_OK) {
            fprintf(stderr, "curl_easy_perform() failed for %s: %s\n", image_url, c
            free(chunk.memory);
            curl_easy_cleanup(curl_handle);
            curl_global_cleanup();
            return 0;
        }

        const char *last_slash = strrchr(image_url, '/');
        const char *url_basename = (last_slash != NULL) ? last_slash + 1 : image_ur

        char temp_name[MAX_FILENAME_LEN];
        strncpy(temp_name, url_basename, MAX_FILENAME_LEN - 1);
        temp_name[MAX_FILENAME_LEN - 1] = '\0';

        char *query_start = strchr(temp_name, '?');
        if (query_start) {
            *query_start = '\0';
        }

        if (strlen(temp_name) == 0 || strcmp(temp_name, ".") == 0 || strcmp(temp_na
           snprintf(temp_name, MAX_FILENAME_LEN, "downloaded_image");
        }

        char original_filename[MAX_FILENAME_LEN];
        strcpy(original_filename, temp_name);

        char final_filename[MAX_FILENAME_LEN];
        char filepath[PATH_MAX];
        int counter = 0;

        do {
            if (counter == 0) {
                strcpy(final_filename, original_filename);
            } else {
                char *dot = strrchr(original_filename, '.');
                if (dot) {
                    snprintf(final_filename, MAX_FILENAME_LEN, "%.*s_%d%s", (int)(d
                } else {

                    snprintf(final_filename, MAX_FILENAME_LEN, "%s_%d", original_fi
                }
            }
            snprintf(filepath, PATH_MAX, "%s/%s", images_dir, final_filename);
            counter++;
        } while (access(filepath, F_OK) == 0);


        FILE *fp = fopen(filepath, "wb");
        if (fp) {
```

```c
        fwrite(chunk.memory, 1, chunk.size, fp);
        fclose(fp);
        printf("    Downloaded and saved as: %s\n", final_filename);
        strncpy(new_filename_out, final_filename, MAX_FILENAME_LEN);
        new_filename_out[MAX_FILENAME_LEN - 1] = '\0';
    } else {
        fprintf(stderr, "Error: Could not open file for writing: %s\n", filepat
        free(chunk.memory);
        curl_easy_cleanup(curl_handle);
        curl_global_cleanup();
        return 0;
    }

    free(chunk.memory);
    curl_easy_cleanup(curl_handle);
    curl_global_cleanup();
    return 1;
}

int process_markdown_file(const char *filepath, const char *root_dir) {
    printf("\nProcessing file: %s\n", filepath);

    FILE *fp = fopen(filepath, "r");
    if (!fp) {
        perror("Error opening file");
        return 0;
    }

    fseek(fp, 0, SEEK_END);
    long fsize = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    char *content = malloc(fsize + 1);
    if (!content) {
        fclose(fp);
        perror("Memory allocation failed");
        return 0;
    }
    size_t bytes_read = fread(content, 1, fsize, fp);
    content[bytes_read] = 0; // Ensure null termination
    fclose(fp);

    regex_t regex;

    const char *pattern = "!\\[[^\\]]*\\]\\((https?:\\/\\/[^)]*)\\)";

    if (regcomp(&regex, pattern, REG_EXTENDED) != 0) {
        fprintf(stderr, "Could not compile regex. (Check pattern compatibility)
        free(content);
        return 0;
    }

    char *current_pos = content;
    char *result_buffer = NULL;
    size_t result_len = 0;
    int images_downloaded = 0;
    int changed = 0;

    while (*current_pos != '\0') {
        regmatch_t pmatch[2];
        int status = regexec(&regex, current_pos, 2, pmatch, 0);

        if (status == 0) {

            changed = 1;

            int full_match_len = pmatch[0].rm_eo - pmatch[0].rm_so;
            int url_len = pmatch[1].rm_eo - pmatch[1].rm_so;

            char full_match[full_match_len + 1];
            char image_url[url_len + 1];

            strncpy(full_match, current_pos + pmatch[0].rm_so, full_match_len);
            full_match[full_match_len] = '\0';

            strncpy(image_url, current_pos + pmatch[1].rm_so, url_len);
            image_url[url_len] = '\0';

           printf("  -> Found match: %s\n", full_match);

            char new_filename[MAX_FILENAME_LEN];
            char images_dir[PATH_MAX];
            snprintf(images_dir, PATH_MAX, "%s/%s", root_dir, IMAGE_FOLDER);

            if (download_and_save_image(image_url, images_dir, new_filename)) {
                images_downloaded++;

               char new_ref[MAX_FILENAME_LEN + 100];
                snprintf(new_ref, sizeof(new_ref), "./%s/%s", IMAGE_FOLDER, new

                char new_full_match[full_match_len + MAX_FILENAME_LEN + 100];

               char *url_start_in_match = strstr(full_match, image_url);

                if (url_start_in_match) {
                    int prefix_len = url_start_in_match - full_match;
                    int suffix_len = full_match_len - (prefix_len + url_len);

                    snprintf(new_full_match, sizeof(new_full_match),
                             "%.*s%s%.*s",
                             prefix_len, full_match,
                             new_ref,
                             suffix_len, url_start_in_match + url_len);
                } else {
                    strcpy(new_full_match, full_match);
                }
               int pre_match_len = pmatch[0].rm_so;
                int new_full_match_len = (int)strlen(new_full_match);

                size_t new_result_len = result_len + pre_match_len + new_full_m
                result_buffer = realloc(result_buffer, new_result_len);

                if (result_buffer) {
                   strncpy(result_buffer + result_len, current_pos, pre_match_l
                    result_len += pre_match_len;

                   strncpy(result_buffer + result_len, new_full_match, new_full
                    result_len += new_full_match_len;
                    result_buffer[new_result_len - 1] = '\0';
                } else {
                    perror("realloc failed during string replacement");
                    goto cleanup;
                }

               current_pos += pmatch[0].rm_eo;

            } else {
               int full_match_total_len = pmatch[0].rm_eo; // length of content
                size_t new_result_len = result_len + full_match_total_len + 1;

                result_buffer = realloc(result_buffer, new_result_len);

                if (result_buffer) {
                    strncpy(result_buffer + result_len, current_pos, full_match
                    result_len += full_match_total_len;
                    result_buffer[new_result_len - 1] = '\0';
                } else {
                    perror("realloc failed (error fallback)");
                    goto cleanup;
                }
                current_pos += pmatch[0].rm_eo;
            }

        } else if (status == REG_NOMATCH) {
            size_t remaining_len = strlen(current_pos);
            result_buffer = realloc(result_buffer, result_len + remaining_len +
            if (result_buffer) {
                strcpy(result_buffer + result_len, current_pos);
                result_len += remaining_len;
                result_buffer[result_len] = '\0';
            } else {
                perror("realloc failed (final copy)");
                goto cleanup;
            }
            break;
        } else {
           char errbuf[100];
            regerror(status, &regex, errbuf, sizeof(errbuf));
```

```
                fprintf(stderr, "Regex matching error: %s\n", errbuf);
                break;
            }
        }

        regfree(&regex);

        if (changed) {
            printf("  File updated successfully.\n");
            fp = fopen(filepath, "w");
            if (fp) {
                fputs(result_buffer, fp);
                fclose(fp);
            } else {
                perror("Error opening file for writing");
            }
        } else {
            printf("  No changes to file content were required.\n");
        }

    cleanup:
        free(content);
        if (result_buffer) free(result_buffer);
        return images_downloaded;
    }

    void traverse_directory(const char *path, const char *root_dir) {
        DIR *dir;
        struct dirent *entry;
        struct stat statbuf;
        char fullpath[PATH_MAX];

        if (!(dir = opendir(path))) {
            fprintf(stderr, "Cannot open directory: %s\n", path);
            return;
        }

        if (strcmp(path, root_dir) == 0) {
            char images_path[PATH_MAX];
            snprintf(images_path, PATH_MAX, "%s/%s", root_dir, IMAGE_FOLDER);
           if (mkdir(images_path, 0755) == 0) {
                printf("Created images directory: %s\n", images_path);
            } else if (access(images_path, F_OK) == 0) {
                printf("Target images directory already exists: %s\n", images_path)
            } else {
                perror("Error creating images directory");
                closedir(dir);
                return;
            }
        }

        while ((entry = readdir(dir)) != NULL) {
           if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
```

```c
                continue;
            }

            snprintf(fullpath, PATH_MAX, "%s/%s", path, entry->d_name);

            if (stat(fullpath, &statbuf) == -1) {
                perror("stat error");
                continue;
            }

            if (S_ISDIR(statbuf.st_mode)) {
               if (strcmp(entry->d_name, IMAGE_FOLDER) == 0 && strcmp(path, root_di
                    continue;
                }
                traverse_directory(fullpath, root_dir); // Recurse
            } else if (S_ISREG(statbuf.st_mode)) {
               const char *extension = strrchr(entry->d_name, '.');
                if (extension && (strcmp(extension, ".md") == 0 || strcmp(extension
                    process_markdown_file(fullpath, root_dir);
                }
            }
        }

        closedir(dir);
    }

    int main(int argc, char *argv[]) {
        if (argc != 2) {
            fprintf(stderr, "Usage: %s <root_directory_path>\n", argv[0]);
            fprintf(stderr, "The 'images' folder will be created in the root direct
            return 1;
        }

        char *root_directory = argv[1];

        struct stat st;
        if (stat(root_directory, &st) != 0 || !S_ISDIR(st.st_mode)) {
            fprintf(stderr, "Error: The directory '%s' does not exist or is not a d
            return 1;
        }

        char absolute_root[PATH_MAX];
        if (realpath(root_directory, absolute_root) == NULL) {
            perror("Error resolving path");
            return 1;
        }

        printf("Starting Markdown image processing for root: %s\n", absolute_root);
        traverse_directory(absolute_root, absolute_root);
        printf("\nProcessing Complete.\n");

        return 0;
    }
```

This one is rocket fast 🚀

That’s all folks 🌟

Conclusion

Again, this is something I’ve been working on for a while, and I just wanted to share it with other geeks who might do non useful stuff as I do sometimes… 😂

Thanks for reading!

Link

- GitHub Repository of the code: https://github.com/aairom/markdown-remote-image-downloader

Markdown Filesystem C Language Python Golang

Following

Written by Alain Airom (Ayrom)

723 followers · 968 following

IT guy, IBMer… sharing my hands-on experiences and technical subjects of my interest (IBM or not). A bit "touche à tout"!

No responses yet

Bgerby

What are your thoughts?