.NET Universe Unplugged 2025-08

AI Agent를 이용한 개발

Visual Sutdio 2022에서 GitHub Copilot Agent로 개발하기

디모이 (정세일)

@dimohy

dimohy@slogs.dev

프로필

디모이 입니다. 닷넷데브에서 활동이 활발했었고 지금은 잠시 휴식 중입니다.
슬로그램 사이트를 운영하고 있습니다. 앞으로 어떻게 발전할지 모르겠습니다.
.NET/C# 개발을 선호하고, 앞으로 .NET이 흥하면 좋겠습니다 ^^

디모이

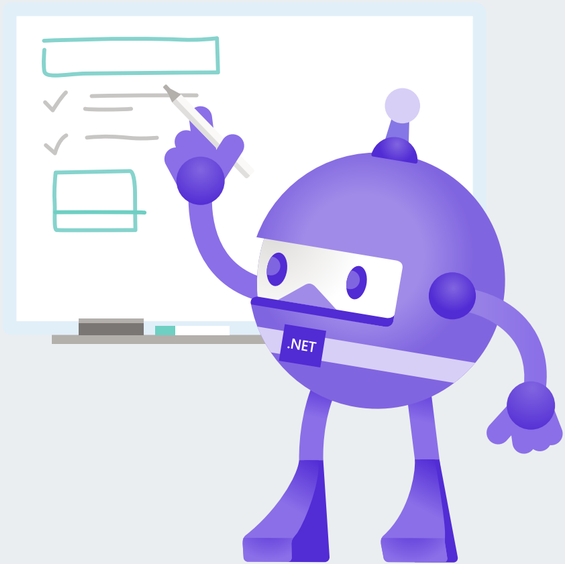
- **.NET 개발자**
- **(주)마음인 기술이사**
- **신구대학교 IT소프트웨어 겸임교수**
- **Microsoft MVP**
- **닷넷데브 운영진**
- **슬로그램 대표 운영진**

전문 분야

- 장비 연동 애플리케이션 개발
- 하지만 이것 저것 가리지 않고 개발함

주요 취미

- 요리 / 낚시 / 성경책 읽기



개요

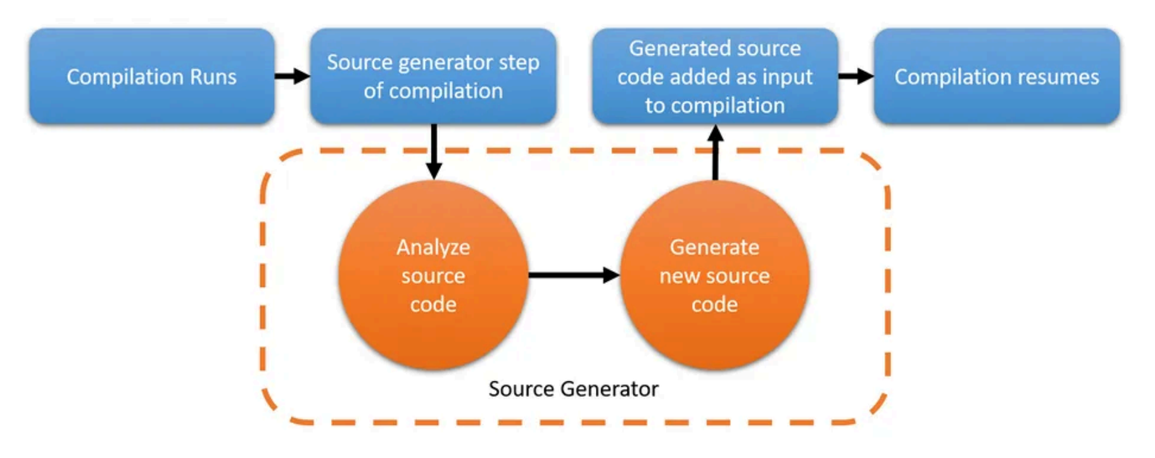
.NET/C# 개발자가 Visual Studio 2022에서 GitHub Copilot을 활용했을 때, AI를 통해 원하는 수준의 코드 품질을 얻을 수 있는지를 확인하고자 했습니다.

이를 위해 Structura라는 소스 생성기 기반의 라이브러리를 코딩 없이 AI만으로 구현해보았습니다.

그 결과, AI 기반 코딩의 가능성을 직접 확인할 수 있었고 동시에 그 한계 또한 분명히 인식할 수 있었습니다.

.NET 소스 생성기란?

>>> **컨테이너 동작 환경에서는 빠르게 시작해야 하기 때문에 JIT 보다는 AOT이 유리.**
Reflection을 사용하지 않고 Reflection의 동작성을 달성하기 위해 소스 생성기가 .NET 5부터 지원됨



동작 과정

- .NET Standard 2.0 기반의 코드 아날라이저 라이브러리로 동작
- 컴파일 도중 소스 생성기(Source Generator) 작동
- 기존 코드를 분석하여 새로운 코드를 만들지 결정함
- 새로운 소스 코드 생성 및 컴파일 대상에 포함
- 컴파일 재개

장점

- 런타임 시점이 아닌 컴파일 시점에서 필요 기능이 추가되므로 런타임 시 빠르게 동작함
- Reflection을 사용할 수 없는 환경에서 유사한 기능을 구현할 수 있음
  예) IoC(DI), PropertyChanged.SourceGenerator 등
- 반복적인 코드를 “특성”등으로 장식해서 소스 생성기를 통해 쉽게 추가할 수 있음

단점

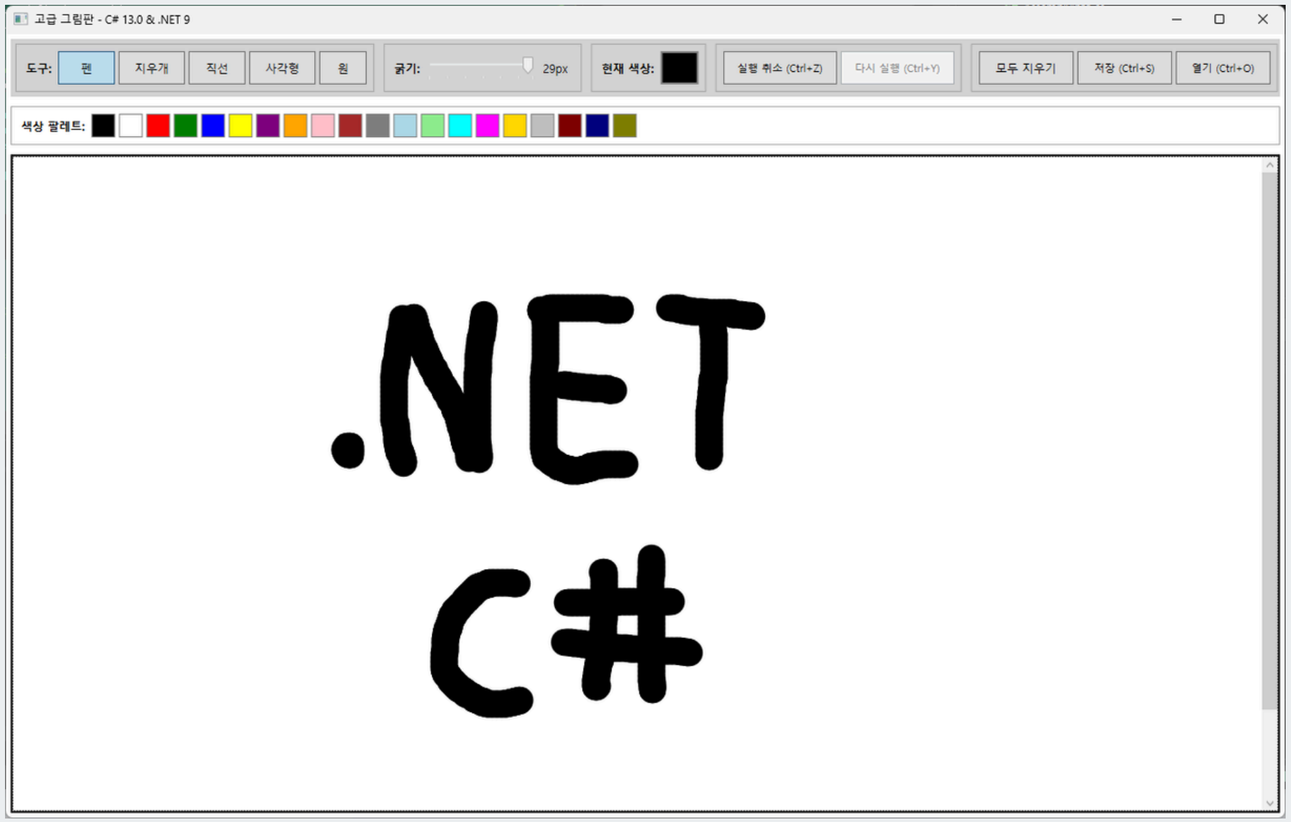
- C# 컴파일러인 Roslyn API를 잘 알아야 함
- 클래스, 특성, 속성, 메서드 등의 구조에 접근하기 위해서는 SyntaxNode및 SyntaxToken, SyntaxTree등 신택스 객체를 처리해야 해서 코드가 복잡해짐
- 컴파일 시점에서 동작하기 때문에 디버깅이 힘듬

>>> 코드가 복잡함. Roslyn API에 충분히 숙달되지 않으면 코딩하기 어려움

```
/// <summary>
/// Analyze projection variable - extract properties from List&lt;AnonymousType&gt;, arrays, etc.
/// </summary>
참조 1개
private static List<(string Name, string Type)>? AnalyzeProjectionVariable(IdentifierNameSyntax identifier, SemanticModel semanticModel)
{
    // Get type information for the variable
    var typeInfo = semanticModel.GetTypeInfo(identifier);

    // 배열 타입 처리
    if (typeInfo.Type is IArrayTypeSymbol arrayType)
    {
        var elementType = arrayType.ElementType;

        // Check if it's an anonymous type
        if (elementType.IsAnonymousType)
        {
            var properties = new List<(string Name, string Type)>();

            // Extract properties from the anonymous type
            foreach (var member in elementType.GetMembers())
            {
                if (member is IPropertySymbol property &&
                    property.DeclaredAccessibility == Accessibility.Public)
                {
                    var propertyName = property.Name;
                    var propertyType = FormatTypeName(property.Type);
                    properties.Add((propertyName, propertyType));
                }
            }

            return properties.Count > 0 ? properties : null;
        }
    }

    // 기존 Named Type 처리
    if (typeInfo.Type is not INamedTypeSymbol namedType)
        return null;

    // Check for collection types like IEnumerable<T>, List<T>, ICollection<T>
    var elementTypeFromCollection = GetCollectionElementType(namedType);
    if (elementTypeFromCollection == null)
        return null;

    // Check if it's an anonymous type
    if (!elementTypeFromCollection.IsAnonymousType)
        return null;

    var collectionProperties = new List<(string Name, string Type)>();

    // Extract properties from the anonymous type
    foreach (var member in elementTypeFromCollection.GetMembers())
```

AI는 .NET / C# 코드를 잘 생성할까?

>>> Visual Studio 2022 + GitHub Copilot



원래는 (작년?)

- 최신 C# 문법 및 WPF, Windows Forms 대응 코드 생성이 별로였음
  ※ 사용할 수 없는 수준

이제는

- GitHub Copilot (Claude Sonnet 4) 기준 잘 생성함
  - 최신 C# 문법
    - 파일 범위 namespace
    - record 타입, readonly 키워드
    - 튜플, 로컬 함수, switch 식, 패턴 매칭

예시

[프롬프트]
가장 최신의 C# 문법으로 그림판 같은 프로그램을 만들어줘!

```
case ToolType.Line when _currentShape is Line line:
    UpdateLine(line, _startPoint, currentPoint);
    break;

case ToolType.Rectangle when _currentShape is System.Windows.Shapes.Rectangle rect:
    UpdateRectangle(rect, _startPoint, currentPoint);
    break;

case ToolType.Ellipse when _currentShape is Ellipse ellipse:
    UpdateEllipse(ellipse, _startPoint, currentPoint);
    break;
```

2시간만?에 만든 Structura 라이브러리

>>> 쓸만한 정도의 코드품질이 되는데 2시간 소요되었고 패키징까지 하루 정도 소요됨

STRUCTURA

```
// EF Core projection query
var userProjection = dbContext.Users
    .Select(u => new { u.Name, u.Email, u.Department })
    .ToList();

// Strong type generation
TypeCombiner.Combine()
    .WithProjection(userProjection)
    .WithName("UserDto")
    .WithConverter()  // 🔥 Enable smart conversion
    .AsRecord()
    .Generate();

// Automatic conversion usage
List<Generated.UserDto> typedUsers = UserDto.FromCollection(userProjection);
Generated.UserDto singleUser = UserDto.FromSingle(userProjection.First());
```

Structura 라이브러리란?

- EF Core 질의 결과를 담는 DAO(Data Access Object) 클래스를 자동으로 생성해주는 소스 생성기 기반 라이브러리입니다.
- EF Core를 사용할 때 매번 DAO 클래스를 수동으로 작성하는 번거로움 때문에, DAO 대신 엔티티(Entity) 클래스를 직접 반환하는 안티패턴이 발생합니다.

AI로 확인하고 싶었던 것

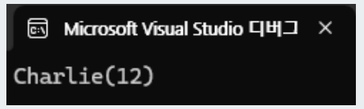
- AI가 학습하지 못했거나 제대로 학습하지 못했을 것이라 예상하는 Roslyn API도 잘 사용해서 코드를 생성하는지
- 코드를 구조화해서 잘 생성해주는지
- 결과가 최종적으로 쓸만한지

결과

- GitHub Copilot이 Microsoft Learn 정보를 활용해서인지 우수한 수준의 소스 생성기 코드를 생성함
- 프롬프트에 따라 매우 구조화된 코드를 생성함
- 결과적으로 매우 만족한 코드를 얻었음!

>>> 코드를 소스 생성기 코드가 컴파일 시점에서 실행해서 해석하고 다음의 코드로 만들어줌

```
using Structura;

var family = new[]
{
    new { Name = "Alice", Age = 35 },
    new { Name = "Bob", Age = 40 },
    new { Name = "Charlie", Age = 12 }
};

var result = family
    .Where(x => x.Age < 30)
    .Select(n => new { Desc = $"{n.Name}({n.Age})" });

TypeCombiner.Combine()
    .WithProjection(result)
    .WithName("CombinedNames")
    .WithConverter()
    .AsRecord()
    .Generate();

var combined = Generated.CombinedNames.FromCollection(result);
foreach (var item in combined)
{
    Console.WriteLine(item.Desc);
}
```



실행 결과 굿!

```
namespace Generated
{
    /// <summary>
    /// Generated record type: CombinedNam
    /// Properties from anonymous types, E
    /// Namespace: Generated
    /// Static converter methods: Combined
    /// </summary>
    - 참조
    public partial record CombinedNames(
        string Desc
    );
}
```

.WithProjection() 및
.AsRecord()에 의해
무명 클래스의 속성을 적절하게 레코드로
만들어짐

.WithConverter()에 의해
무명 클래스 인스턴스를 생성된 클래스 인스턴스로 변환
할 수 있음!

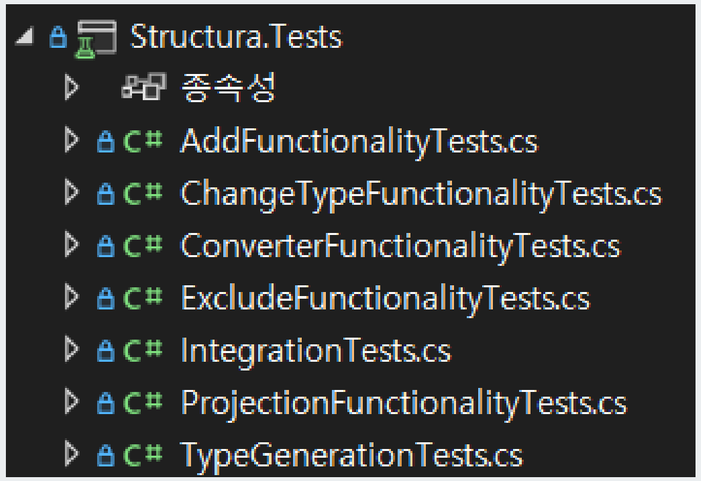
```
public partial record CombinedNames
{
    /// <summary>
    /// ?? **Converts** anonymous object collection to CombinedNames list
    /// </summary>
    - 참조
    public static List<CombinedNames> FromCollection(IEnumerable<object> source)
    {
        if (source == null)
            throw new ArgumentNullException(nameof(source));

        return source.Select(FromSingle).ToList();
    }
```

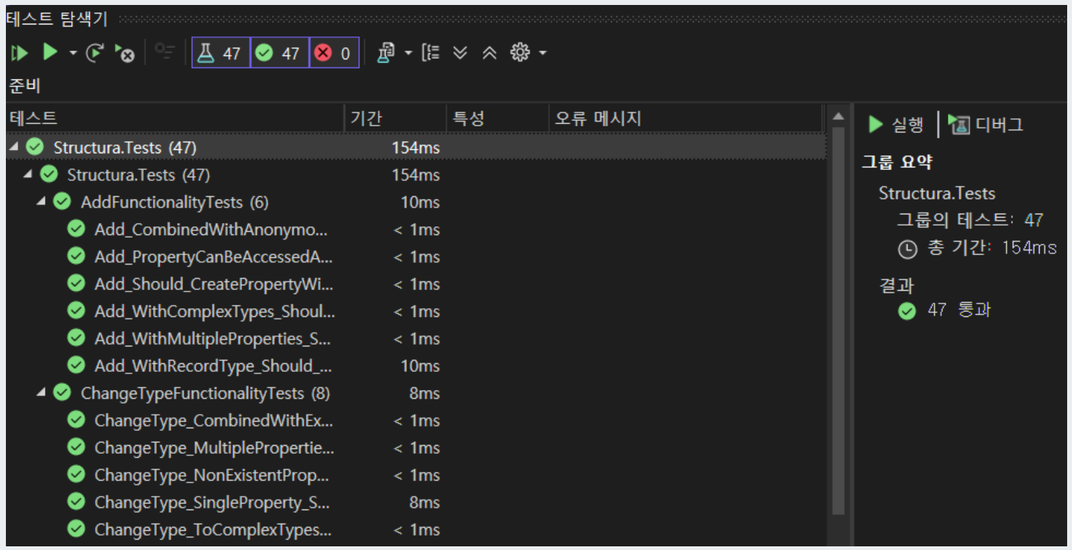
>>> AI가 생성해준 소스 생성기 코드

```
/// <summary>
/// Source generator for Structura type combination
/// </summary>
[Generator]
참조 0개
public class StructuraSourceGenerator : IIncrementalGenerator
{
    참조 0개
    public void Initialize(IncrementalGeneratorInitializationContext context)
    {
        // ?? **Generate() 호출 감지** - AnonymousTypeCombinerBuilder.Generate() 호출만 감지
        var generateCalls = context.SyntaxProvider
            .CreateSyntaxProvider(
                predicate: static (node, _) => IsGenerateMethodCall(node),
                transform: static (ctx, _) => GetGenerateCallInfo(ctx))
            .Where(static info => info != null);

        // Generate source for all found calls
        context.RegisterSourceOutput(
            generateCalls.Collect(),
            static (spc, generateCalls) => GenerateTypes(spc, generateCalls));
    }
```

- 상당히 깔끔하고 람다식 등을 적절하게 잘 활용하고 있음
- 생성한 메서드의 코드량이 적절했고 해석하기 용이했음
- 하지만 코드 생성시 약간의 문제도 있었음
  - 문자열 블럭(원시 문자열 리터럴)을 처음 시도했다가 에러가 발생해 결국에는 StringBuilder로 문자열을 생성하는 코드로 바꿈
  - 코드 블럭을 누락하는 초보적인 실수도 가끔 함. 이렇게 되면 코드를 지웠다 다시 생성했다 하는 오동작을 하게 됨
- 하지만 대체로 꽤 훌륭한 코드 품질을 보여줌

>>> 단위 테스트 프로젝트도 훌륭하게 생성하고 테스트를 수행함





- 단위 테스트를 요청하면 단위 테스트 프로젝트를 생성해서 훌륭하게 테스트 코드를 생성한 후 훌륭하게 테스트를 진행함
- 사람이 하기에 귀찮은 일을 시키면 좋겠다는 생각을 하게 됨
- 함정
  - 명령을 구체적으로 주지 않으면 구조적으로 테스트를 진행하지 못함
  - 쓸데없는 테스트 코드를 많이 생성하기도 함
- 결론
  - 하나씩 하나씩 좁은 범위로 끊어 가면서 명확하게 지시해야 함

(AI로 코딩을 하면) 어떤 장점이 있을까?

최초 구조를 잡을 때 유용함

- 명확한 요구사항과 구조화 지침이 있다면 구조화된 틀을 잘 만듦

경험하지 않은 코드를 학습할 수 있음

- 원하는 기능이 어떻게 동작하는지 아직 경험하지 못했을 때 훌륭한 학습 가이드라인이 될 수 있음

단순 작업을 위임함

- 데이터베이스 스키마를 이용해 엔티티 클래스를 생성하라는 등의 좁은 명령은 아주 잘 수행함

문제 해결의 속도가 빨라짐

- 문제를 해결하기 위한 리서치 시간이 단축되어 문제 해결 사이클이 빨라짐

SumTube (성공)

>>> 유튜브 영상의 스크립트를 추출해서 Ollama를 이용해 영상의 내용을 요약해주는 프로그램

SumTube는?

- Ollama를 이용해 로컬 LLM로 유튜브 영상을 요약해주는 프로그램으로
- 유뷰트 스크립트를 가져오는 yt-dlp 프로그램과
- 로컬 환경에서 LLM을 사용하게 해주는 Ollama를 이용하였음

AI로 확인하고 싶었던 것

- 외부 프로그램을 자동으로 포함하여 최종적으로 정상 동작하는가?
- 외부 프로그램과 상호 동작을 잘 하는가?
- AI가 내 의도를 이해해서 여러 상호작용을 잘 구현하는가?

결과

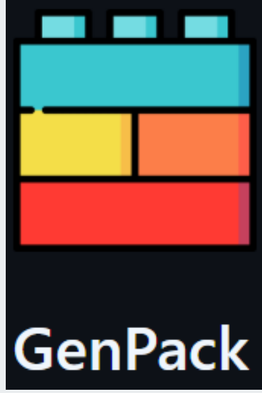
- 중간 중간 생성된 코드를 교정하는 지침을 내려야 했지만 최종적으로 잘 동작하는 코드를 생성함
- 코딩을 전혀 하지 않고 원하는 기능을 구현한 경험이 흥미로웠음

GenPack 고도화 (성공)

>>> 소스 생성기를 이용해 패킷을 만들어주는 라이브러리

```
[GenPackable]
public partial record PeoplePacket
{
    public readonly static PacketSchema Schema = PacketSchemaBuilder.Create()
        .@short("Age", "Age description")
        .@string("Name", "Name description")
        .Build();
}
```

```
public partial record PeoplePacket : GenPack.IGenPackable
{
    /// <summary>
    /// Age description
    /// </summary>
    public short Age { get; set; }
    /// <summary>
    /// Name description
    /// </summary>
    public string Name { get; set; } = string.Empty;
    public byte[] ToPacket()
    {
        using var ms = new System.IO.MemoryStream();
        ToPacket(ms);
        return ms.ToArray();
    }
    public void ToPacket(System.IO.Stream stream)
    {
        using var writer = new GenPack.IO.EndianAwareBinaryWriter(stream, GenPack.UnitEndian.Little, Ge
        writer.Write(Age);
        writer.Write(Name);
    }
    public static PeoplePacket FromPacket(byte[] data)
    {
        using var ms = new System.IO.MemoryStream(data);
        return FromPacket(ms);
    }
    public static PeoplePacket FromPacket(System.IO.Stream stream)
    {
        PeoplePacket result = new PeoplePacket();
        using var reader = new GenPack.IO.EndianAwareBinaryReader(stream, GenPack.UnitEndian.Little, Ge
        int size = 0;
        byte[] buffer = null;
        result.Age = reader.ReadInt16();
        result.Name = reader.ReadString();
        return result;
    }
}
```



GenPack는?

- 패킷에 대한 스키마를 Fluent API 형태로 정의하면 소스 생성기를 이용해 관련 패킷을 생성함
- 패킷을 클래스 형태로 유지하고 활용할 수 있는 구조 및 기능을 제공함
- Big/Little Endian, 문자열 인코딩 지원 (AI 활용)
- 체크섬 지원 (AI 활용)
- 목록 형태의 가변 길이 및 8비트, 16비트, 32비트 사이즈 지원 (AI 활용)

AI로 확인하고 싶었던 것

- 기존 사람이 생성한 코드를 손상 시키지 않고 원하는 기능을 잘 추가 구현해주는지
- AI가 생성한 코드가 성능에 큰 문제가 없는지
- 네이밍 규칙을 기존 코드를 잘 참고해서 구현해주는지

결과

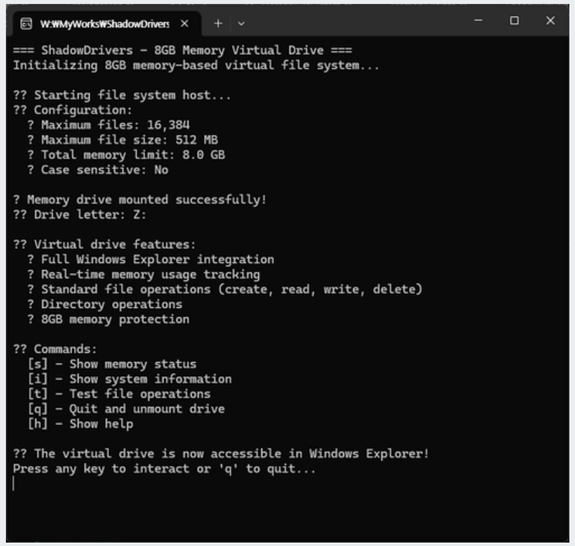
- 기존 코드를 활용해서 요청 기능을 잘 구현했으며 구현 품질도 만족

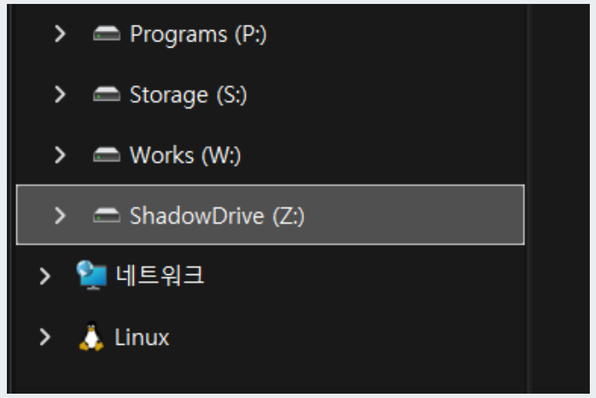
>>> 어라? 되네?

- 학습이 안되었을 것 같은 요구도 잘 진행하는 것을 보았을 때 LLM을 이용한 AI 코딩의 수준이 많이 높아졌다고 느꼈음
- Visual Studio 2022에서도 이제 어느 정도 사용 가능하겠다 느꼈음
- AI의 한계를 실험하고 싶은 욕구가 생겼음

ShadowDrivers (실패)

>>> WinFsp를 이용해서 원격의 디렉토리를 공유해 가상 드라이브로 사용하는 실험





ShadowDriver는?

- WinFsp(Windows File System Proxy - FUSE for Windows)를 이용
- 원격의 특정 디렉토리를 공유해 가상 드라이버로 매핑해주는 실험을 함
- 원격에 서버로 띄우고 내 컴퓨터에 클라이언트로 띄워 확인

AI로 확인하고 싶었던 것

- WinFsp의 .NET 라이브러리는 AI가 학습할 가능성이 매우 낮았음
- 이런 요구도 잘 수행하는지 궁금했음
- 가상 드라이브를 구현하는 과정은 작업량이 많고 다양한 예외 상황을 처리해야 해서 모든 것을 문제 없이 구현할지 궁금했음

결과

- AI가 WinFsp의 .NET 라이브러리 사용법을 몰랐음
- AI가 Microsoft Learn을 통해 확인 후 코딩했으나 실제 라이브러리의 사용법과 달랐음
- 샘플 코드를 추가해서 참고하라고 했더니 코드를 구현하고 컴파일도 됨!
- 하지만 다양한 문제점이 있어서 사용하기 어려운 수준이였음 (실패!)

GenPack

한계 & 지침

한계

- **AI가 학습하지 못한 내용은 구현하지 못하거나 제대로 구현하지 못함**
- **구현의 의도가 어느 수준에 도달하면 깨지는 문제가 발생함**
  - **코드 구현의 일관성 깨짐**
  - **잘 구현된 코드를 다시 고쳐서 엉터리로 오염시킴**
- **중복 코드 및 불필요한 방어 코드가 상당히 많아짐**
- **코드량이 많아질 수록 LLM 속도가 급격히 떨어짐**

=> 현재 **LLM은 개발자 능력 증폭기**로 훌륭하지만 자율적으로 모든 업무를 처리하는 수준에는 한참 도달하지 못한 상태임

지침

- **AI 지침을 적용하고 조금씩 고도화함**
  .github/copilot-instructions.md
- **개발자가 AI 사용을 통제해야 함**
  - **Agent 모드를 사용하지 않고 AI가 생성하는 코드를 일일이 검토해야 함**
  - **어디까지 AI가 맥락을 유지하고 이행하는지를 확인해야 함**
  - **넓게 해석하게 하지 말고 좁게 해석하고 코드를 생성하게 유도해야 함**
- **개발자가 AI 사용을 통제해야 함**
  - **Agent 모드를 지양하고 일일이 AI가 생성하는 코드를 검토해서 적용하는 것이 바람직해 보였음**
- **최종적으로 개발자가 AI가 생성하는 모든 코드에 대한 장악력을 지녀야 함**

※ Gemini 2.5 Pro와 Claude Opus 4를 적절히 겸해서 사용하면 좋다고 함
- **Gemini 2.5 Pro는 복잡한 버그 탐지와 문제 해결력 탁월**
- **Claude Opus 4는 새 코드 작성에 능함**

THANK YOU

CONTACT

닷넷데브/슬로그램 @dimohy

dimohy@slogs.dev