



by entwickler.de

Developing and Debugging Source Generators

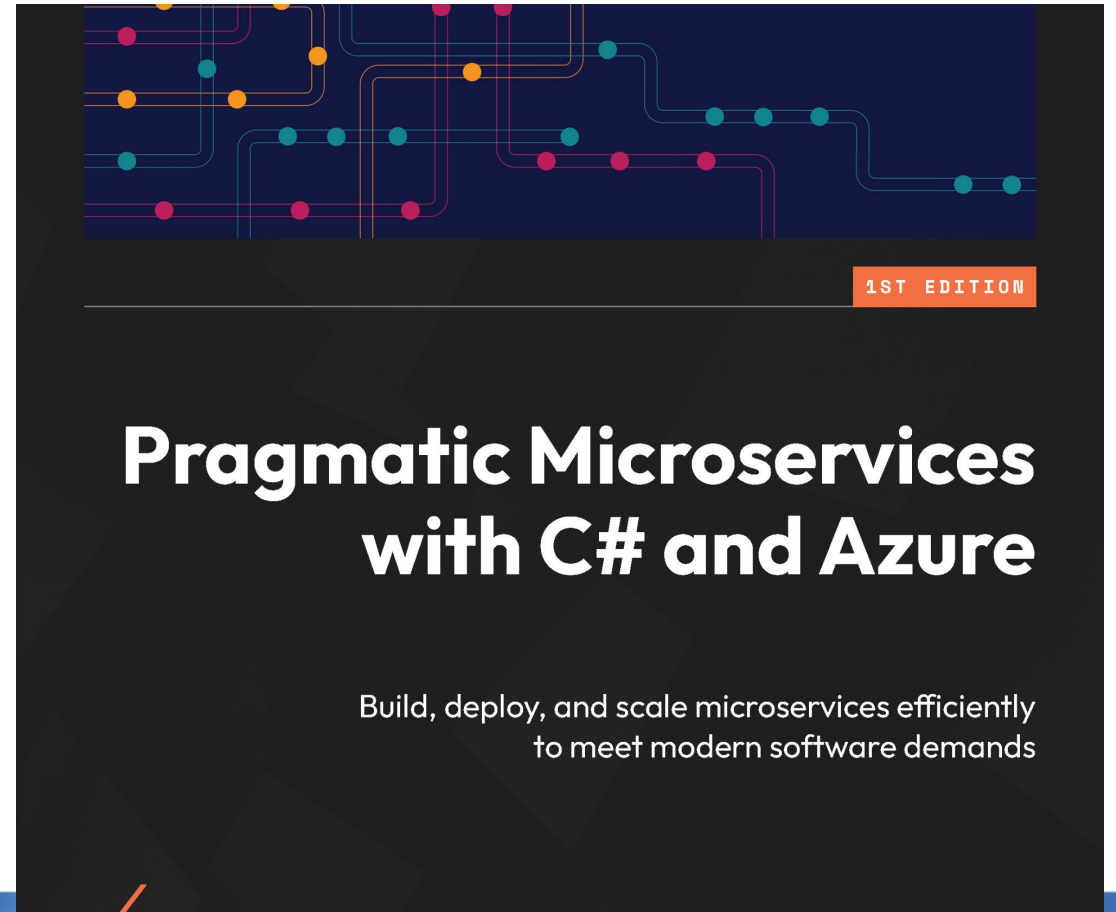
Christian Nagel

<https://www.cninnovation.com>

<https://csharp.christiannagel.com>

Christian Nagel

- Training
- Coaching
- Consulting
- Development
- New book: **Pragmatic Microservices**
- Microsoft MVP
- www.cninnovation.com
- csharp.christiannagel.com
- @christiannagel



A close-up, low-key photograph of a vintage movie projector. On the left, a large, white, spoked film reel is partially visible, slightly out of focus. The main body of the projector is dark, with various metallic components and a bright light source visible on the right side. The lighting creates strong highlights and deep shadows, emphasizing the mechanical details.

Agenda

- The .NET Compiler Platform
- Hello, generator!
- Practical source generators
- Specific features

What are source generators?

- Create source code during (pre) compilation
- Access syntax trees of the project's source code
- Remove barriers to linker-based and AOT compilation



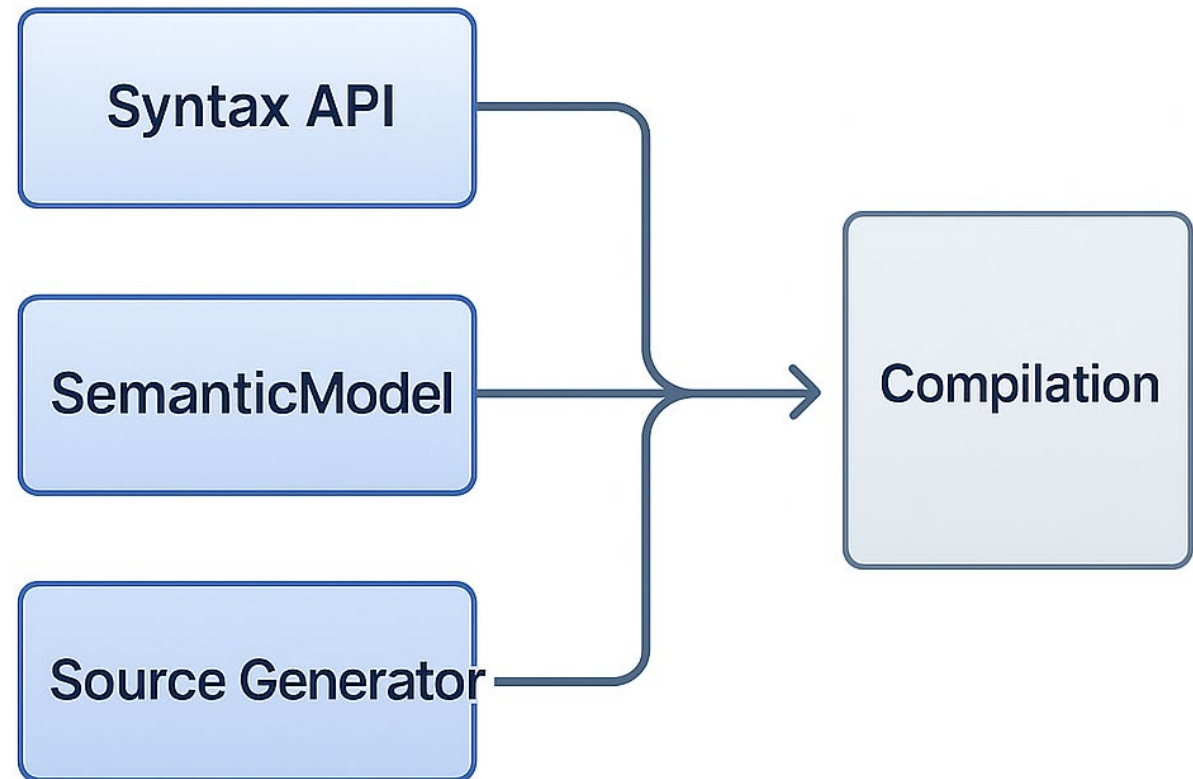


Source Generators History

- C# 6 introduced Roslyn - **The .NET Compiler Platform**
- Object Models for the compilation pipeline
- Analyzers: inspect code quality
- Source generators are based on analyzers



Compiler Pipeline

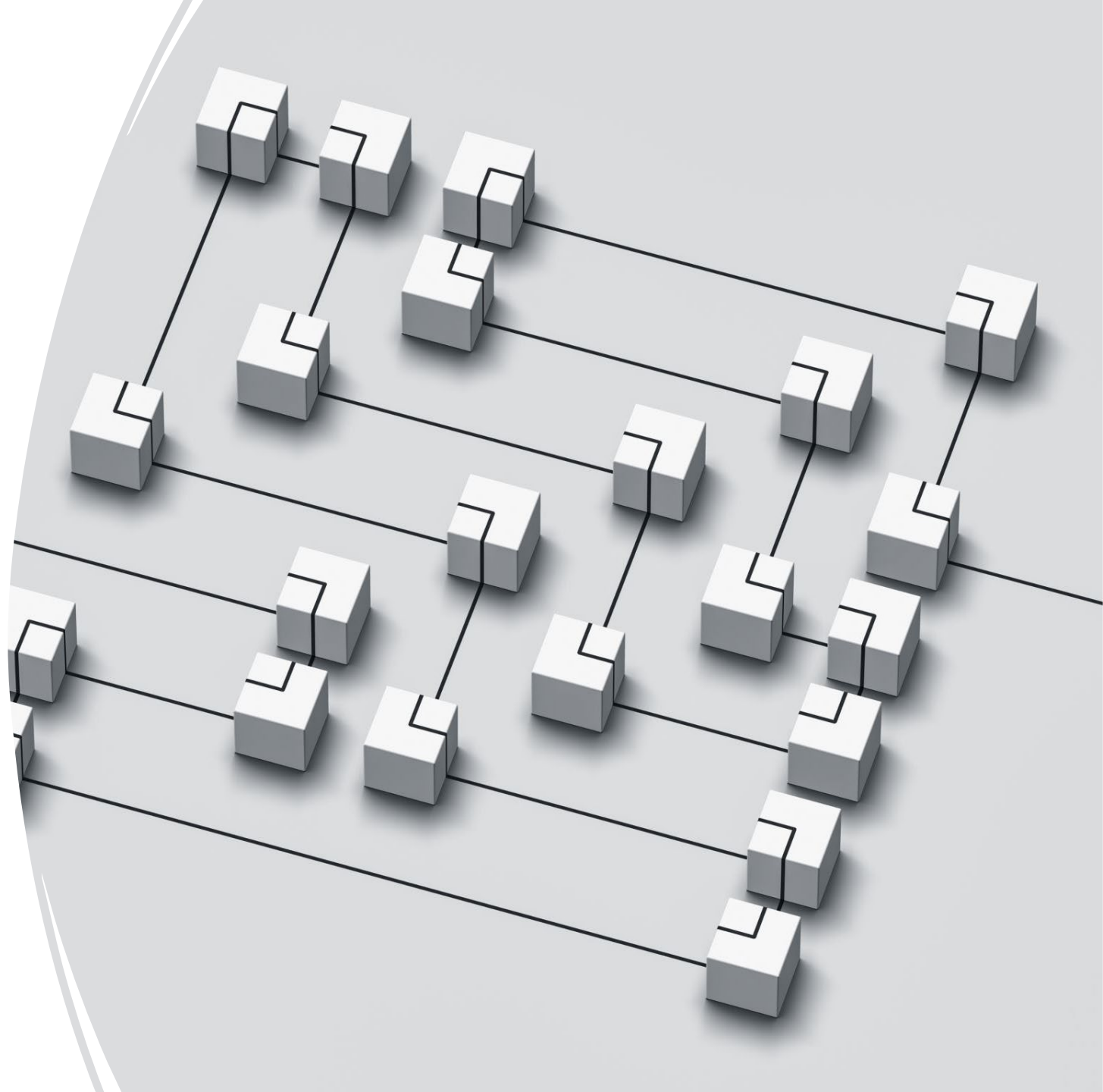


Syntax analysis

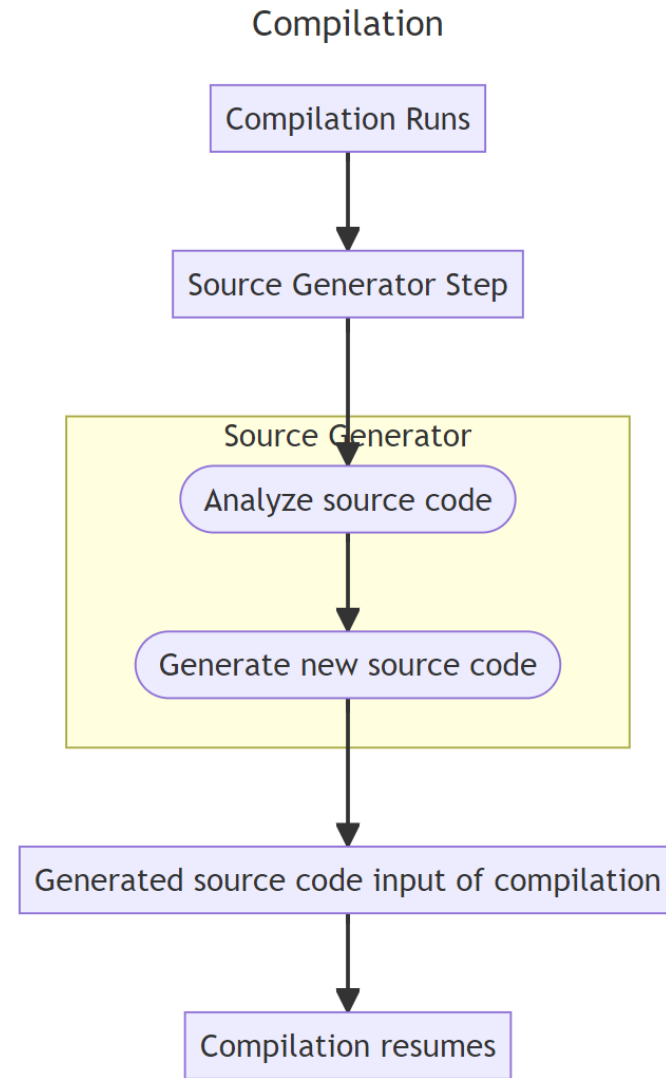
- Syntax API
- Tree structure of the source code
- Query for specific code
- Walk the tree

Semantic Analysis

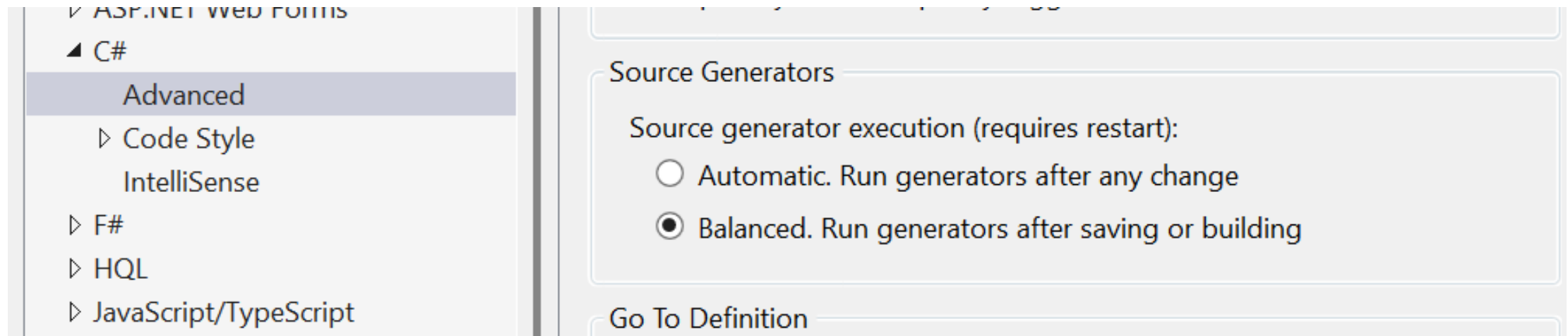
- Semantic meaning of a program
- Symbols, bindings
- Use the *syntax tree* and a **Compilation** to create a **SemanticModel**




Phases of compilation (with source generator)




Source generator triggers





Creating your Source Generator





2 Generations

- Source Generators (C# 9, .NET 5)
 - `ISourceGenerator` Interface, now deprecated
 - `Generator` attribute
- Incremental Generators (.NET 6+)
 - `IncrementalGenerator` Interface
 - `Generator` attribute

Design goals

- Produce C# code that's added to the compilation
- Additive only – cannot change existing code
- May access additional files (also non-C# files)
- Run un-ordered – a source generator doesn't have access to output of other generators!
- User specifies generators (NuGet packages)
- Generators create a pipeline (base input → output)

Non-design

- Language features (e.g. records)
- Code-rewriting
 - Optimization
 - Logging injection
 - IL weaving
 - ~~Call-site re-writing (Interceptors!)~~

Required generators

- There are exceptions (Roslyn repo)...

```
// Currently, only Razor is considered to be a required generator.  
/// </remarks>  
public static bool IsRequiredGenerator(this ISourceGenerator generator)  
{  
    // For now, we hard code the required generator list to Razor.  
    // In the future we might want to expand this to e.g. run any generators  
    // with open generated files  
    return generator.GetGeneratorType().FullName ==  
        "Microsoft.NET.Sdk.Razor.SourceGenerators.RazorSourceGenerator";  
}
```

The background of the slide features three wind turbines on a beach. The sun is setting behind the turbines, creating a warm, orange and yellow glow in the sky. The water in the foreground is dark blue with stylized white waves. The text "Hello, Generator!" is written in a large, white, outlined font across the middle of the image.

Hello, Generator!

Project File

- **TargetFramework**
 - netstandard2.0
- **LangVersion**
 - Can use a new C# version
- **IsRoslynComponent**
 - Not required, enhanced Analyzer rules, future compatibility
- **EnforceExtendedAnalyzerRules**
 - Performance analysis
 - Memory usage warnings
 - Best practices enforcements

Packages needed

- Microsoft.CodeAnalysis.Analyzers
 - Diagnostic analyzers
- Microsoft.CodeAnalysis.Csharp
 - Parsing, analyzing, generating C# code

Generator source code

- Implement **IIncrementalGenerator** interface
- **Generator** attribute

```
[Generator]
public class HelloWorldSourceGenerator : IIncrementalGenerator
{
    public void Initialize(IncrementalGeneratorInitializationContext context)
    {
    }
}
```

Generator source code

- **CompilationProvider** – get access to the complete compilation
- **RegisterSourceOutput** – map input to output

```
public void Initialize(IncrementalGeneratorInitializationContext context)
{
    // use this only if compilation is needed → more next
    var compilationProvider = context.CompilationProvider;

    context.RegisterSourceOutput(compilationProvider,
        static (spc, compilation) ⇒ Execute(spc, compilation));
}
```


Use the source generator

- Reference the project as analyzer
- Add to compiler analyzer pipeline
- Don't use library as compiler or runtime reference

```
<ItemGroup>  
  <ProjectReference  
    Include="..HelloWorldGenerator/HelloWorldGenerator.csproj"  
    OutputItemType="Analyzer"  
    ReferenceOutputAssembly="false" />  
</ItemGroup>
```

See generated files

- Dependencies → Analyzers
- Project file options

```
<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>  
<CompilerGeneratedFilesOutputPath>  
  Generated  
</CompilerGeneratedFilesOutputPath>
```

Package as NuGet

- **DevelopmentDependency**
- Specify the path analyzers/dotnet/cs – needed by analyzers!

```
<PropertyGroup>
  // PackageId, Version, ...
  <DevelopmentDependency>true</DevelopmentDependency>
</PropertyGroup>

<ItemGroup>
  <None
    Include="$(OutputPath)\$(AssemblyName).dll"
    Pack="true"
    PackagePath="analyzers/dotnet/cs"
    Visible="false" />
</ItemGroup>
```

Debugging source generators...



Visual Studio 2026 runs source generators in a separate process...



Restart Visual Studio 2022 with every new version



Better: create a unit test project!

Test helper invokes generator (1)

- Run generator
- receive input source
- return generated source and diagnostics

```
public static class TestHelper
{
    public static (string GeneratedSource, string[] Diagnostics) RunGenerator(
        string source)
    {
    }
}
```

Test helper invokes generator (2)

- Use **GeneratorDriver** to run generator
- Pass input code
- Return generated code and diagnostics

```
public static class TestHelper
{
    public static (string GeneratedSource, string[] Diagnostics) RunGenerator(
        string source)
    {
        // ...
        GeneratorDriver driver = CSharpGeneratorDriver.Create(generator);
        driver = driver.RunGeneratorsAndUpdateCompilation(
            compilation, out var outputCompilation, out var diagnostics);
    }
}
```

Snapshot Testing

- Great for checking and fixing results from source generation
- Captures generated output
- Readable and reviewable
- Fast feedback loop
- Supports complex scenarios

Summary Hello Generator

- IncrementalGenerator
- Debug with unit tests

Practical Source Generators



Scenario Stage 1

- Using the **DataSource** attribute, the source generator creates a factory of sample items

```
[DataSource(EntityName = "User", Count = 5)]
public class User
{
    public string Name { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;
    public int Age { get; set; }
    public bool IsActive { get; set; }
    public DateTime CreatedAt { get; set; }
    public UserRole Role { get; set; }
}
```

IncrementalGeneratorInitializationContext

Compilation provider

- Access to types, symbols, metadata
- Analyze codebase

Syntax provider

- Regenerates when specific syntax changes
- Targeted for specific syntax patterns

Register post initialization output

- Output after the initialization
- Create for attributes created by the source generator

Register source output

- Connects input with output

Create attribute (marker)

- **RegisterPostInitializationOutput**
 - Create attribute that can be used by the consumer
 - Helper types, base classes, attributes
 - Emit source code once before Execute, after initialization
 - Available after the initialization before the pipeline continues

Use Syntax Provider

- **CreateSyntaxProvider**
 - Preferred way to hook into the pipeline
- **Predicate**
 - Filter and transform efficiently
 - Runs on every syntax node → fast and shallow implementation
- **Transform**
 - Deep analysis
 - Convert into meaningful data (e.g. semantic models, symbols)
 - Return information that's required for code generation
- **Where**
 - Further filter transformed results, e.g. not null
- **Collect**
 - Aggregate the data into an ImmutableArray

IncrementalValueProvider

- Declarative pipeline stage
- Computes a value once per compilation
- Lazy – only runs when needed
- Can be cached – if inputs don't change, the output is reused
- Composable – Select/Combine/Where/...

Scenario Stage 2

- Add external data sources to be used by the source generator

```
[DataSource(  
    EntityName = "User",  
    Count = 5,  
    ConfigurationFile = "User.datasource.json")]
```

External files

- Files need to be specified by the consumer
 - **AdditionalFiles** in project file
- **AdditionalTextProvider**
 - Access non-code files
 - You can filter file name with **Where**
- **Combine**
 - Combine two **IncrementalValueProvider** or **IncrementalValuesProvider**
 - Result is a tuple (Left, Right)
 - Use **Select** on the combination to generate output

Stage 3: Improvement – cache file content

- Incremental value provider chain
- File change detection
- Per-file tracking
- Compares inputs between builds
- Caches outputs for unchanged inputs
- Re-executes transforms only with changed inputs

Stage 4-5: More cache improvements

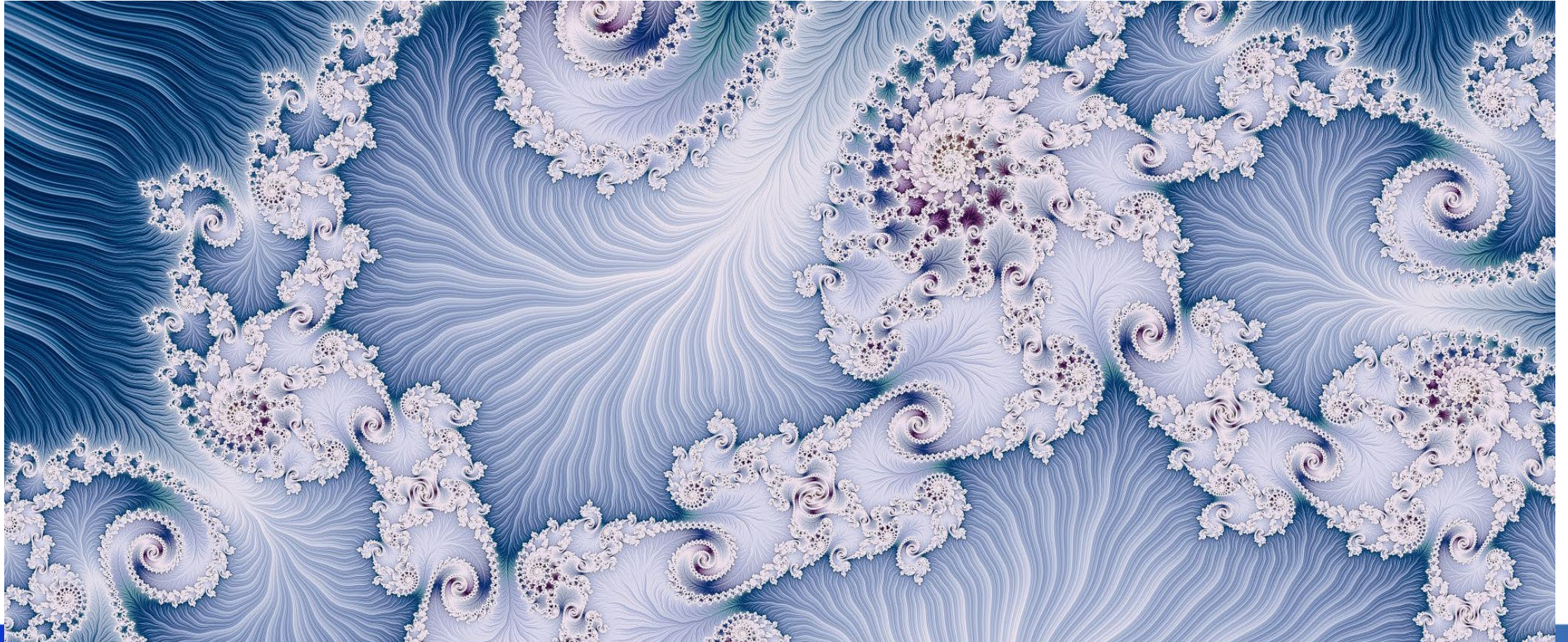
- Stage 4
 - Multi-level cache (L1, L2, L3)
- Stage 5
 - Optimized three-tier architecture
 - Lock-free in memory
- Just for reference
- Only useful for a big number and large external files
- Over-engineered in most scenarios

Stage 6 – Special for attributes (markers)

- Typical use case
- Eliminates manual interaction with attributes
- Better caching

```
var classDeclarations = context.SyntaxProvider  
    .ForAttributeWithMetadataNameC
```


Useful scenarios



Scenario: C# Version

- Use the C# language version to create different code
- **ParseOptionsProvider** with **CSharpParseOptions**

```
var languageVersionProvider = context.ParseOptionsProvider  
    .Select(static (parseOptions, ct)  
        ⇒ GetLanguageVersionInfo(parseOptions));
```


Scenario: Fill objects with private members

- **UnsafeAccessor** attribute

```
[UnsafeAccessor(UnsafeAccessorKind.Field, Name = "_title")]  
private static extern ref string GetBookTitleField(Book instance);
```

```
[UnsafeAccessor(UnsafeAccessorKind.Method, Name = "set_Publisher")]  
private static extern void SetBookWithPrivateSettersPublisher(  
    BookWithPrivateSetters instance, string value);
```

Scenario: Call-site rewriting

- **InterceptableLocation** class
- **SemanticModel.GetInterceptableLocation** method (preview)

```
[global::System.Runtime.CompilerServices.InterceptsLocationAttribute(1,  
"qPii2RCL4JQb4YSFXL7gkCUGAABQcm9ncmFtLmNz")]  
    public static async System.Threading.Tasks.Task<int>  
ProcessAsync(string data)
```

Guidelines



Source generator modes

Mode	Trigger frequency	Responsiveness	Performance impact	Use case
Full	On every keystroke	Immediate	High	Debugging, authoring generators
Balanced	File save, build, semantic changes	Timely, not constant	Moderate	Daily development
Minimal	On build only	Delayed	Low	Large solutions, CI builds

Performance guidelines

- Use **CreateSyntaxProvider** with a fast predicate
 - Avoid scanning all syntax trees
- Avoid **CompilationProvider** unless needed
 - Access full compilation is expensive
 - Only use if you need semantic model-wide analysis or type resolution
- Use **Collect** only when needed
 - Overusing can increase memory pressure
- Minimize use of **AdditionalTextProvider**
 - Filter them aggressively
- Use **RegisterPostInitializationOutput** for static code
 - Emit helper types once, avoid recomputing in Execute
- Use **Combine** instead of manual combining
 - Ensures caching and invalidation

Let's start creating source generators!



Thank you for coming!

Questions?

<https://github.com/cnilearn/bastamainz2025>

<https://www.cninnovation.com>

<https://csharp.christiannagel.com>

Wir bitten um
dein Feedback!

**BITTE
BEWERTEN
SIE UNS!**

