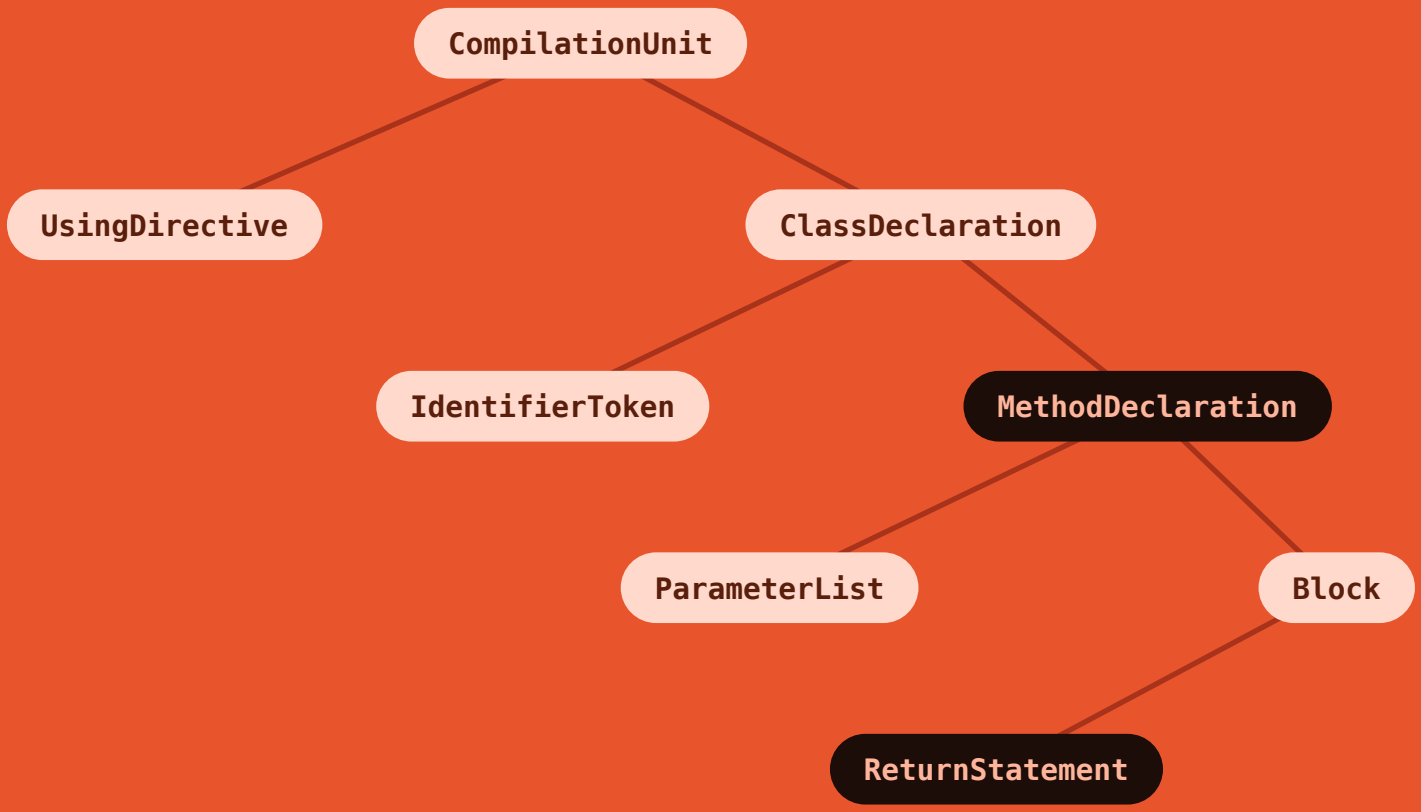
THE .NET COMPILER PLATFORM PREVIEW

Roslyn In Depth

Analyzers, Source Generators, and Compiler-Powered Tooling



Gérald Barré

MICROSOFT MVP · MEZIANTOU.NET

Copyright © 2026 Gérald Barré

All rights reserved. No part of this publication may be reproduced, distributed, transmitted, stored in a retrieval system, or otherwise used in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder, except as permitted by applicable copyright law.

Published by Gérald Barré
Montréal, Québec, Canada

First edition, 2026

Legal deposit, Bibliothèque et Archives nationales du Québec, 2026
Legal deposit, Library and Archives Canada, 2026

The information in this book is provided on an "as is" basis, without warranties of any kind, express or implied. Although the author has taken reasonable care in preparing this work, the author makes no representations or warranties regarding the accuracy, completeness, or suitability of its contents. To the fullest extent permitted by applicable law, the author shall not be liable for any loss or damage arising directly or indirectly from the use of, or reliance on, the information contained in this book.

Product names, company names, logos, brands, and other trademarks mentioned in this book are the property of their respective owners. Their use does not imply affiliation with, sponsorship by, or endorsement by the respective trademark owners.

Unless otherwise indicated, source-code examples included in this book are provided for educational purposes and may be used and adapted in your own projects. The source code is provided without warranty of any kind.

Contents

A free preview — the introduction and 2 chapters. The full edition has 38 chapters and 9 appendices.

Introduction in this preview
- What you will build
- Who this book is for
- A note on Visual Basic
- How the code in this book works
- Versions
- How to read this book
- Using the samples

Part I — The Compiler as a Library

Chapter 1 — What Is Roslyn? in this preview
- Compiler as a service
- A ten-minute tour
- The three API layers
- Where Roslyn runs
- Immutability, and why the API looks the way it does
- Roslyn, MSBuild, and the SDK
- What this book calls things
- Exercises

Chapter 2 — The Compilation Pipeline and the Object Model
- The stages
- The pipeline in one program
- Parse options: the language, not the world
- Compilation options: everything else
- References, and the reference problem
- Declaration versus binding
- `IOperation`, briefly
- Immutability in practice
- A harness you will reuse
- Exercises

Chapter 3 — The Syntax API
- Three kinds of thing
- Span and full span
- Navigating
- Walkers
- Rewriters
- Constructing new syntax
- Which syntax API do I reach for?
- Code that does not compile
- Tools that answer “what does this look like as a tree?”
- Exercises

Chapter 4 — The Semantic API
- The three questions
- Binding failure is information
- Types and conversions
- Symbols
- Displaying symbols
- Symbol equality: never use ==
- Finding well-known types
- Three string forms, and which one an API wants
- Documentation IDs
- Attributes as data
- Nullability
- Accessibility
- What this costs
- The semantic questions rules actually ask
- The four layers, in one sentence each
- Exercises

Chapter 5 — The IOperation API and Flow Analysis
- The problem `IOperation` solves
- The operation tree
- Operations you did not write
- Control flow graphs
- The cheaper flow-analysis APIs
- Choosing a level
- Walking operations
- Exercises

Chapter 6 — Diagnostics: The Common Currency
- Descriptor and diagnostic
- Anatomy of a descriptor
- Localization
- Locations

Severity, and what actually decides it
Hidden diagnostics
Tags that change behavior
`AD0001`, and why your analyzer must not throw
The descriptor, as a checklist
Exercises

Part II — Choosing the Right Extension Point

Chapter 7 — What You Can Build with Roslyn (and What You Cannot)
One question, six answers
The catalog
The decision table
Where things run, and what it costs you
What is not extensible
When Roslyn is the wrong answer
A note on ambition
Design checkpoint: choose your mechanism
Exercises

Part III — Production Analyzers

Chapter 8 — Your First Analyzer
The project
The analyzer
The tests
Running it for real
Debugging it
What is missing
Exercises

Chapter 9 — The Registration Model in Depth
The registrations
Cost, roughly
Syntax or operations: a worked comparison
Per-compilation state
Symbol-scoped state
Concurrency, and what it forbids
Cancellation
Generated code
Choosing, in one page
Exercises

Chapter 10 — Designing Diagnostics That People Keep Enabled
The failure mode you are designing against
False positives are fatal, and they are not symmetrical
Choosing a severity honestly
`IsEnabledByDefault`
Every rule needs an escape hatch
Writing the message
Titles, categories, and grouping
Numbering a rule family
Help links and documentation
The `Unnecessary` tag and fading
`NotConfigurable`
Adding a rule is a breaking change
One rule, designed twice
Where this chapter sits in a rule's life
A checklist before shipping a rule
Exercises

Chapter 11 — Configuring Compiler Components: EditorConfig, MSBuild, and AdditionalFiles
The three channels
Severity: the channel you get for free
Reading your own `.editorconfig` options
MSBuild properties
`AdditionalFiles`: data too big for a key
All three at once
Global analyzer config files
The SDK's own knobs
Four different things live in `.editorconfig`
Precedence, end to end
Exercises

Chapter 12 — Testing Analyzers
The packages
A verifier helper
The markup language
Controlling the environment
The tests every rule needs
Organizing a growing suite
What the tests do not cover
Exercises

Chapter 13 — Analyzer Performance
Measure first

One rule, three implementations
Reading the numbers
The rules
A rule of thumb for "is this fast enough?"
When a rule is inherently expensive
Exercises

Chapter 14 — Suppression: `#pragma`, Attributes, and `DiagnosticSuppressor`
What users already have
`DiagnosticSuppressor`
What suppression actually does
Limits
Designing a rule that suppresses well
When a suppressor is the wrong answer
Checkpoint: is this analyzer ready for people?
Exercises

Part IV — Fixes and Refactorings

Chapter 15 — Code Fixes
Two assemblies, not one
Passing data from the analyzer to the fix
The fix
`equivalenceKey` and Fix All
Convergence
Offering more than one fix
Fixes that touch more than one document
What runs where
Exercises

Chapter 16 — Editing Code Without Breaking It
The problem, concretely
`DocumentEditor`
`SyntaxGenerator`
Annotations: telling the workspace what you want
The three passes
Which editing tool, and when
Trivia, from the editing side
Editing text instead of trees
Respecting the user's style
Fixes that span documents
The safe-edit checklist
Exercises

Chapter 17 — Refactorings
Refactoring, fix, or analyzer?
The provider
Titles and ordering
Refactorings that span documents
A checklist for the offer itself
What refactorings cannot do
Exercises

Chapter 18 — Testing Fixes and Refactorings
The basic shape
Convergence and iterations
`FixedState` and `BatchFixedState`
Choosing among several actions
Testing refactorings
The assertions people forget
What still is not covered
Exercises

Part V — Source Generators

Chapter 19 — Source Generators: The Model
What a generator can and cannot do
Minimum viable generator
A complete generator
Running it in a real project
Testing it
`ISourceGenerator`: the legacy model you will read, not start with
Generation versus the alternatives
Exercises

Chapter 20 — Incremental Generators in Depth in this preview
The mental model
The sources
Configuration, in one page
A pipeline that uses most of it
The combinators
The outputs
Reading MSBuild from a generator
Testing a pipeline
A note on `CompilationProvider`
Review your pipeline before you read Chapter 21

Exercises
Chapter 21 — Caching and Equality
How incrementality works
The cardinal rule
The same generator, twice
Proving it
Building a good model
The `ImmutableArray<T>` trap
The diagnostics trap
The `CompilationProvider` trap
The `Collect()` trap
Memory
Why did this stage re-run?
A checklist
Exercises
Chapter 22 — Emitting Good Code
The generator
Hint names
Encoding, and why it is not optional
Line endings
The file header
Attributes that mark it as generated
`global::` on every single type reference
Names that cannot collide
Matching the user's declaration
Adapting to the consumer's language version
Trimming and ahead-of-time compilation
The result
What this generator does not handle
A checklist
Exercises
Chapter 23 — Generator Diagnostics and Failure Modes
What the user sees when a generator throws
Why the analyzer, and not the generator
The default design: the generator generates, the analyzer complains
What is left in the generator
If you do report: carry data, not diagnostics
Descriptors are still a public contract
Ordering and determinism
The failure modes, in one table
Exercises
Chapter 24 — Testing and Debugging Generators
Driving a generator
The one assertion that matters most
Snapshot testing
Expected output as a string
Running what was generated
The generated code as an ordinary reference
What to test
Seeing the output for real
Debugging
When it works in tests but not in the IDE
A generator's own build log
A generator's test plan
Checkpoint: is this generator ready for people?
Exercises
Chapter 25 — Beyond Generation: Interceptors
What an interceptor is
The library being intercepted
The generator
The consumer
Where interceptors are used
Fragility, and what it costs
When not to use them
Exercises
Part VI — Shipping and Adoption
Chapter 26 — Packaging Analyzers and Generators as NuGet Packages
What is in an analyzer package
The packaging project
Shipping build logic with the package
The dependency problem
Proving it works
Splitting the package
A checklist
Exercises
Chapter 27 — Versioning Against Roslyn
The matrix
Choosing a floor
When you need a newer API

One package, several variants
Building the variants
Which one actually loaded?
Run-time feature detection
Deprecating a host
One package, followed through three hosts
The `netstandard2.0` tax
Exercises

Chapter 28 — Release Tracking, Documentation, and Analyzer Hygiene
The analyzers for analyzer authors
Release tracking
Versioning a rule set
Documenting a rule
Public API tracking
A pre-release checklist
Exercises

Chapter 29 — Consuming Analyzers at Scale
What is already there
Layered `.editorconfig`
Precedence, and a trap in that output
The five severities, and the ones that cannot be measured
Adopting analyzers in a repository that already exists
`dotnet format`
Third-party rule sets worth knowing
Keeping CI honest without making it hostile
Adopting a rule set, in order
Checkpoint: the shipping path, end to end
Exercises

Part VII — Workspaces and Solution Tools

Chapter 30 — The Workspaces API
The object model
The tour
`AdhocWorkspace`
Fork, then apply
Computing what changed
Documents
Options
Threading
Exercises

Chapter 31 — MSBuildWorkspace: Loading a Real Solution
Where MSBuild runs
The project file
Opening things
`WorkspaceFailed` is not optional
The result
Multi-targeting
Performance
A preflight checklist
When `MSBuildWorkspace` will not cooperate
Exercises

Chapter 32 — Analyzing a Whole Solution
Analyzer or solution tool?
The shape of a solution-wide tool
`SymbolFinder`
Running your own analyzers over a solution
Finding a type across the solution
Reporting
Questions worth asking a solution
Exercises

Chapter 33 — Rewriting a Whole Solution
The tool
Dry run by default
`Renamer`
Editing many documents
Filtering what you touch
Applying
Making the change reviewable
Verification
The gate before `--apply`
Exercises

Chapter 34 — Formatting and Code Style, Programmatically
The tour
Where the options come from
`Formatter`
`Simplifier`
What `dotnet format` does
Preserving what you did not change
Exercises

Chapter 35 — Compiling and Running C# at Run Time
The sample
Scripting
Emit and load
Security
Trimming, AOT, and single-file
Cheaper alternatives to consider first
Exercises

Part VIII — Hosts, Recipes, and Troubleshooting

Chapter 36 — Roslyn Inside the Tools
The IDE is the same compiler model, in a different host
Live analysis versus build analysis
Visual Studio
VS Code and the language server
Rider and ReSharper
Choosing a vehicle
"It works in the build but not in the IDE"
Exercises

Chapter 37 — Recipes and Case Studies
Recipe: enforce an architectural boundary
Recipe: catch fire-and-forget async
Recipe: strongly typed IDs
Recipe: dependency-injection registration
More recipes, sketched
Exercises

Chapter 38 — Anti-Patterns and Pitfalls
Analyzers
Code fixes and refactorings
Source generators
Packaging and versioning
Workspaces and tools
Design mistakes that are not bugs
The three questions
Before you conclude it is a Roslyn bug

Appendices

Appendix A — Syntax Cheat Sheet
C# construct to node type
Members every node has
Patterns worth memorizing
Reading the kind lists
`SyntaxKind`: declarations
`SyntaxKind`: expressions
`SyntaxKind`: statements

Appendix B — Symbol API Cheat Sheet
Getting a symbol
Asking about a symbol
Documentation ID spelling
Displaying a symbol
Across a solution (Chapter 32)
`SymbolKind`
`TypeKind`
`MethodKind`
`SpecialType`

Appendix C — `IOperation` Quick Reference
Getting an operation
The operations you will use most
Members every operation has
Flow analysis
All `OperationKind` values

Appendix D — The Version Matrix
The rule
The matrix
Finding the versions on a machine
Package pairings
Which package for which component
Package folder layout
The `netstandard2.0` tax
Compiler-side properties worth knowing

Appendix E — Component Template Pieces
The analyzer project
The code fix project
The generator project
Release tracking files
The test project
The packaging project
The consumer test

The inner loop

Appendix F — Diagnostic ID Prefixes
Prefixes in wide use
Choosing your own
Categories

Appendix G — Tooling
Answering “what does this look like as a tree?”
Seeing what a generator produced
Measuring
Debugging
Packages
Command line
Repositories worth having open

Appendix H — Glossary

Appendix I — Further Resources
Where do I learn X?
The checklists and decision tables
Source
Documentation
Tools
Packages to read the source of
The companion volume
This book’s samples

Introduction

The C# compiler is a library.

That sentence is the whole reason this book exists. `csc` is not a sealed executable that swallows `.cs` files and spits out a DLL; it is a set of NuGet packages you can reference from a console application, and every intermediate result it computes on the way from text to IL — the token stream, the syntax tree, the symbol table, the bound operations, the diagnostics — is a public API you are allowed to call.

That design decision, made when the compiler was rewritten in managed code under the codename **Roslyn**, is why your editor can rename a symbol across a solution without regular expressions, why a NuGet package can add a compile error to your build, and why a library can write code for you at compile time instead of doing reflection at run time.

This book is about writing those things yourself.

What you will build

By the end you will have written and tested every kind of component Roslyn lets you plug in, and packaged the ones that ship together:

- **analyzers** that report a problem in the editor as the user types, on the right span, with no false positives, and that cost almost nothing on projects they do not apply to;
- a **code fix** that repairs the problem without destroying the surrounding formatting, and that survives "Fix all occurrences in solution";
- a **refactoring** offered at the cursor, and a **suppressor** that silences someone else's rule when you have context they do not;
- **incremental source generators** that add code to the compilation, keep their pipelines cached between keystrokes, and have tests proving it;
- a **NuGet package** carrying an analyzer, its code fix, and a generator, with a consumer test that restores the real package and proves it works; and a second package that ships one analyzer built against more than one Roslyn version;
- and two **solution-wide tools** built on `MSBuildWorkspace`: one that loads a real solution and answers a question about every project in it, and one that rewrites it and hands you a reviewable diff.

These are separate, self-contained samples rather than one product grown chapter by chapter: a chapter that needs a new idea gets a new rule or generator built for that idea. Appendix E gathers the project files into one set of proven pieces, including where a refactoring and a suppressor go when you package them alongside the rest.

Who this book is for

You are comfortable in C#. You have used analyzers and source generators as a consumer — you have seen a `CA` warning, you have used a generator that came with a library — and you now want to be on the other side of that line.

You do not need any compiler background. This book teaches the parts of the pipeline it needs, when it needs them, and nothing more: there is no chapter on parser theory, because writing an analyzer does not require one.

You do not need to know MSBuild either. Roslyn runs **inside** a build, so the two meet in a few places — how an analyzer is passed to the compiler, how a generator reads an MSBuild property, how a workspace loads a project. Where that happens, this book explains the minimum and points at its companion volume, *MSBuild In Depth*, for the rest.

Before you start, make sure the ordinary project mechanics are not the thing slowing you down:

- You can run `dotnet --version`, `dotnet build`, and `dotnet test` from a terminal.
- You can read a small SDK-style `.csproj`: `TargetFramework`, `PackageReference`, `ProjectReference`, and MSBuild properties.
- You are comfortable restoring a changed sample to a known state, either with Git or by re-extracting the sample archive.

- You have an editor or integrated development environment (IDE) where you can debug a test. Visual Studio gives analyzer authors a few extra conveniences, but the tests and command-line builds in this book do not require it.

That is the infrastructure floor, not a secret prerequisite. When a chapter needs more build machinery — package layout, binary logs, `global.json`, workspace loading — it introduces the piece it needs.

A note on Visual Basic

Roslyn compiles two languages, and the API is layered accordingly. `Microsoft.CodeAnalysis` is deliberately language-neutral — symbols, `IOperation`, diagnostics, the analyzer and generator registration model, workspaces — and above it sit two parallel language layers, `Microsoft.CodeAnalysis.CSharp` and `Microsoft.CodeAnalysis.VisualBasic`, each with its own syntax tree, its own `SyntaxKind`, and its own `SyntaxFactory`.

Every listing in this book is C#, and every component it builds declares `LanguageNames.CSharp`. That is a choice about examples, not a boundary in the API. Everything above the syntax layer transfers unchanged: the registration model, diagnostic design, configuration, testing, performance, the incremental generator pipeline, packaging and versioning, and the workspace APIs are the same types with the same rules whichever language the code being analyzed is written in. What does not transfer is anything that names a piece of syntax — a `SyntaxKind`, a cast to a C#-specific node type, a `SyntaxFactory` call, an extension method from `CSharpExtensions`. The VB counterparts have different names and often different shapes, because the two grammars are different.

That gives you a rule for the times you do want to cover both. An analyzer written entirely against `ISymbol` and `IOperation` — registering operation actions rather than syntax-node actions — can declare both languages in its `[DiagnosticAnalyzer]` attribute and work for both; one that pattern-matches on syntax nodes is C#-only, and saying so in the attribute is what keeps it from being loaded, and paying for itself, in a VB compilation. Packaging knows about the split too: the `analyzers/dotnet/cs/`, `analyzers/dotnet/vb/`, and language-agnostic `analyzers/dotnet/` folders in Chapter 26 are what decide which compilations a shipped component is loaded into at all. Where a language-neutral API has a C#-specific companion, this book says so at the point it matters — Chapter 5 does it for conversions — so the seams are visible even though only one side of them is exercised.

The reason to pick one language and stay there is that almost all the extension work being done today is C#: Visual Basic is supported and maintained, but it is no longer gaining language features, so the analyzers, fixes, and generators you are likely to write — or to be asked to write — target C# code. If you need the VB side, the map above tells you which chapters you can read as-is and which paragraphs you have to translate.

How the code in this book works

Every complete listing in this book is a real file, read in when the book is compiled rather than retyped into the text — so what you see on the page is what is in the companion archive, character for character. Short fragments and single commands are still written inline, where showing a whole file would obscure the point. Console output is captured from real runs of the samples rather than composed.

For a book about compilers, "it compiles" is not a high enough bar, so the guarantee goes further for the shipped components. Analyzer samples that represent real packages have tests that assert the exact diagnostic they report and the exact span they report it on. Code fixes have tests that assert the exact code they produce. Generators have tests that assert the source they emit compiles — and the generated files shown from consumer projects are the compiler's own output, written to disk by a real build, so the text cannot describe generated code that was never generated. Harness-only examples, such as the Chapter 13 performance benchmark, prove themselves by running and by checking their captured output.

All of it is rebuilt and re-run whenever the book changes, so a listing, its tests, and the output printed beside it cannot drift apart.

Versions

The samples target the **.NET 10 SDK**, pinned in `global.json`. Analyzer and generator projects compile against `Microsoft.CodeAnalysis` **4.14** rather than the newest release — deliberately, because a Roslyn component must reference the **oldest** host it intends to support, not the newest one available. Chapter 27 explains that rule in full

and builds a package that multi-targets several Roslyn versions. Tools that host their own workspace, which have no such constraint, use the newer workspace-host version pinned in the sample package versions instead.

Roslyn moves. Where a feature is new, experimental, or likely to change, the text says so and pins a version. Interceptors are the obvious case: stable only since the .NET 9.0.2xx SDK, opt-in namespace by namespace, and fragile by design.

How to read this book

The book is in eight parts, and each one opens with a page telling you where you are, what the next few chapters build, and what to watch for. The parts are not decoration: the object model changes at some of those seams, and so does the amount of damage a mistake can do.

Part	Chapters	What it is
I — The Compiler as a Library	1–6	The vocabulary: syntax, symbols, operations, diagnostics. Read in order, even if you came for generators.
II — Choosing the Right Extension Point	7	A map of everything Roslyn lets you build, and everything it does not. Read it early; come back to it whenever you are choosing between two mechanisms.
III — Production Analyzers	8–14	From a first rule to one people keep enabled: registration, design, configuration, tests, performance, suppression.
IV — Fixes and Refactorings	15–18	Repairing what an analyzer reports, without wrecking the file it repairs.
V — Source Generators	19–24	Adding code to a compilation, and keeping the pipeline that does it genuinely incremental. Chapter 25 is optional and is called out in its own chapter opener.
VI — Shipping and Adoption	26–29	Packaging, versioning against a moving compiler, hygiene, and the view from the consuming repository.
VII — Workspaces and Solution Tools	30–35	Outside the compiler process: loading, analyzing, and rewriting a whole solution.
VIII — Hosts, Recipes, and Troubleshooting	36–38	How the IDE differs from the build, a cookbook, and a troubleshooting reference.

Read in order and every path below is covered. Most readers arrive for one of them, though, and each path lists, in book order, the chapters that the later chapters on it depend on:

If you came to write…	Read
An **analyzer**	1–6, then 7, then 8–14. Ship it with 26–29, and read the IDE smoke-test checklist in Chapter 24 before you do: it is written for generators, but Chapters 26 and 29 hold every package to it. Read 36 the first time the editor and the build disagree about your rule.
A **code fix** or **refactoring**	The analyzer path, then 15–18. A fix needs the diagnostic it repairs, so 8–14 is not optional here. Fixes and refactorings run only in an editor, which makes Chapter 36 and that smoke-test checklist more important on this path, not less.
A **source generator**	1–6, then 7, then 8–12, then 19–24 — 25 only if its opening checklist describes your problem. Ship it with 26–29, and read 21 twice. Chapters 8 to 12 are on this path because a generator project is set up like Chapter 8's analyzer, reports diagnostics designed the Chapter 10 way, reads MSBuild properties and `AdditionalFiles` through Chapter 11's channels, and is tested with the habits of Chapter 12. Chapter 23 also pairs a generator with an analyzer and a code fix, so read 15 before it. Read 36 when the editor and the build disagree.
A **solution-wide tool** on `MSBuildWorkspace`	1–6, then 30–34. Chapter 35 is a different subject bound in the same part; 15–18 pays off if your tool writes code, and Chapter 34 builds on 16.
Nothing — something is **broken**	38 first, by symptom. Then 36 if the build and the IDE disagree, 27 if an API is missing at run time, and Appendix I to find the chapter that covers it.

Not every chapter is on every reader's path. Two are marked **optional** on their part opener, and again in a box on their own first page listing the conditions that make them worth your time: Chapter 25 (interceptors), because the feature is advanced, needs an explicit opt-in, and has changed shape between releases, and Chapter 35 (running C# at run time), because it is an application of the compiler APIs rather than a component you ship. Skip either on a first read and lose nothing that a later chapter depends on — the main path through Part V is Chapters 19 to 24, and through Part VII, Chapters 30 to 34.

Most teaching chapters follow the same shape — concept, a minimal working component, a realistic one, what the compiler or IDE actually does with it, how to test it, what it costs, and what goes wrong. Dense chapters close with a **Carry forward** box: a handful of bullets, no new material, holding only what the next chapters assume you now have. Then the exercises — and after every set of them, an **Expected observations** box.

That box exists because most of these exercises ask you to break something and predict what happens, and an exercise you can get wrong in private is not worth doing. Each entry pairs one exercise with the command to run where there is one, what should happen, and the sentence the exercise existed to produce. It sits **after** the questions rather than beside them, because predicting the answer is most of the value; read it once you have your own answer, including — especially — when the two disagree. Exercises marked **open** have no single right answer, and their entries give the criteria a good one meets instead.

Three recurring boxes carry more weight than the prose around them, and they are not the same thing. A **Carry forward** box is what you should be holding when you turn the page. An **Expected observations** box is the answer key for the exercises. A **checklist** is meant to be worked through with a project open — before a rule joins a rule set, before a package is pushed, before the run that writes to somebody's repository. The book accumulates about twenty of them, and Appendix I collects the major checklists in the order a component meets them, so that the working reader does not have to remember which chapter argued for which list.

The last chapter is deliberately different. Chapter 38 is a lookup table — symptom, cause, first thing to check — meant to be open beside a failing build rather than read through, so it has no sample and no exercises. The nine appendices after it are reference in the same spirit, and Appendix I is the index to all of it: which chapter answers which question, and every checklist in the book gathered in the order a component meets them.

Callouts flag things worth stopping for:

Gotcha. A trap that is easy to fall into and hard to debug.

Under the hood. What the compiler is really doing, when knowing changes how you write the code.

Version note. Behavior that depends on the Roslyn, SDK, or Visual Studio version.

IDE vs. build. Behavior that differs between live analysis in an editor and a command-line build. This one is unique to this book, and it accounts for a surprising share of the bugs you will hit.

Performance. Something that runs on every keystroke.

Try it. A short experiment worth doing yourself.

Using the samples

The companion archive — `roslyn-in-depth-samples.zip` — is the source for the sample listings, not a transcription. Everything in it was built and tested on the way to this page.

Download it from `https://goto.meziantou.net/roslyn-in-depth-samples.zip`. Unpack it and work from the **archive root**, the folder holding its `README.md`. Every path printed in this book is written relative to that root, so `samples/ch08-first-analyzer/Ch08.Analyzer/` means what it says only if that is where you are standing.

In the archive	What it is
`global.json`	The SDK pin — .NET 10. It ships with the samples so that what you build is what was validated. Run `dotnet --version` in the archive root first; if it does not match, nothing below will behave as printed.
`samples/chNN-.../`	One folder per chapter **that has sample code**: `ch` and the chapter number, then a short slug for the subject, which is not always the chapter's title. A few chapters are argument rather than code — Chapter 7 chooses between extension points, Chapter 10 designs diagnostics, Chapters 36 and 38 are host notes and a lookup table — and those have no folder here. The archive's `README.md` carries the full map.
`samples/appendix-reference/`	The program that generates the kind lists printed in the reference appendices, rather than a chapter's sample.
`samples/chNN-.../<Consumer>/generated/`	Generated sources written to disk by a real build and committed. When a chapter shows what a generator emitted, this is the file it was read from — not something retyped. Rebuilding a consumer regenerates them in place, so you can change a generator and diff the result.
`_nobuild/`	Deliberately not compiled: fragments, packaging inputs consumed by a different build, and code that is **meant** to fail so a chapter can show the error. A build failure in here is the point, not a problem.
`README.md`	The archive's own map: a quick start, what each folder convention means, and one row for every chapter that ships sample code. It is generated from the archive's actual contents, so it cannot drift from what is in there — which is also why the conceptual chapters above are absent from it rather than listed as empty.

To build one project, point `dotnet build` at it. To build everything except the folders that are supposed to fail:

```
find samples -name '*.csproj' -not -path '*/_nobuild/*' -exec dotnet build {} \;
```

In PowerShell:

```
Get-ChildItem samples -Recurse -Filter *.csproj |
  Where-Object FullName -notmatch '[/\\]_nobuild[/\\]' |
  ForEach-Object { dotnet build $_.FullName }
```

Each chapter's samples stand alone, and that includes their diagnostic IDs: every chapter uses its own block of `MZ` numbers, so a rule that comes back in a later chapter can come back under a new ID. Appendix F explains the scheme.

Several exercises ask you to break something on purpose. Undo the edit afterward — `git checkout` if you cloned the repository, or re-extract the archive — because the next exercise assumes the file it was handed.

One thing is in the book but not in the archive: the captured console output these pages print. The archive carries the source listings and the committed `generated/` sources; it does not carry the repository's `outputs/` directory, which holds the captured console text and is regenerated by CI on every build. Where a chapter shows what a command or a program printed, running the sample yourself is how you reproduce it — expect the same text apart from absolute paths, timings, SDK patch versions, and machine names.

Start by looking at what Roslyn actually is.

Chapter 1 — What Is Roslyn?

Roslyn is the C# and Visual Basic compiler, rewritten in managed code, with its internals exposed as a supported public API. The name was the project codename; the shipping products are the `csc` and `vbc` executables in the .NET SDK, the language service inside Visual Studio and Visual Studio Code, and a family of NuGet packages under `Microsoft.CodeAnalysis.*`.

The interesting part is not that the compiler was rewritten. It is what the rewrite made possible: **the compiler and the editor are the same code**. When Visual Studio underlines an error while you type, it is not running a lookalike parser — it is running the compiler, incrementally, over the file you are editing. That single fact explains most of what follows in this book, including several of its hardest constraints.

Compiler as a service

A traditional compiler is a batch process. You hand it files, it hands you an assembly and a list of errors, and everything it learned in between — which tokens it saw, which types it resolved, which overload it picked — dies with the process.

Roslyn keeps all of it and lets you ask:

- **What is the structure of this text?** Give me a tree of nodes, tokens, and whitespace that round-trips exactly to the original characters.
- **What does this identifier mean?** Not "the string `Greet` appears here", but "this is a call to `Greeter.Greet(string)`, declared in that file, returning `string`".
- **What does this code do?** A language-neutral tree of operations — an invocation, an assignment, a conversion — above the syntax.
- **What is wrong with it?** The same diagnostics the build produces, on demand, without a build.
- **And by the way, produce an assembly.**

Anything you can build with Roslyn is some combination of those questions plus the ability to hand back new source text.

A ten-minute tour

Here is a complete program that does all of it. It is longer than a first listing usually is, so read it looking for four specific things — they are the design principles the rest of the book is built on, and everything else in the program is scaffolding around them:

1. **parsing never fails**, whatever you hand it;
2. the tree **round-trips** to the original text, character for character;
3. nothing has any **meaning** until there is a compilation with references;
4. and **diagnostics are data**, not just error messages.

Each is picked apart after the output. One NuGet reference:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <RootNamespace>HelloRoslyn</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <!-- The C# compiler, as a library you can reference like any other. -->
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp" />
  </ItemGroup>

</Project>
```

And the program:

```
using System;
using System.Linq;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Emit;

// Ten minutes with Roslyn: parse C# text, look at its structure, ask the
// compiler what the code *means*, and finally emit an assembly -- all from a
// plain console application with one NuGet reference.

const string code = """
    using System;

    class Greeter
    {
        public string Greet(string name) => "Hello, " + name;
    }
    """;

// 1. Parse. This is pure syntax: no references, no type checking, no errors
//    about unknown names. It is fast and it always produces a tree.
SyntaxTree tree = CSharpSyntaxTree.ParseText(code);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();

Console.WriteLine("== Syntax ==");
foreach (var node in root.DescendantNodes().Take(6))
    Console.WriteLine($"{node.Kind(),-28} {Excerpt(node)}");

// The tree is lossless: it round-trips to the exact original text.
Console.WriteLine($"Round-trips exactly: {root.ToFullString() == code}");

// 2. Compile. A Compilation is a set of syntax trees plus references plus
//    options. Only now does the compiler know what 'string' means.
var compilation = CSharpCompilation.Create(
    assemblyName: "HelloRoslyn.Sample",
    syntaxTrees: [tree],
    references: [MetadataReference.CreateFromFile(typeof(object).Assembly.Location)],
    options: new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary));

// 3. Ask semantic questions through a SemanticModel, which binds syntax to
//    symbols. This is the step that turns text into meaning.
SemanticModel model = compilation.GetSemanticModel(tree);

MethodDeclarationSyntax greet = root.DescendantNodes()
    .OfType<MethodDeclarationSyntax>()
    .Single();

var method = (IMethodSymbol)model.GetDeclaredSymbol(greet)!;

Console.WriteLine();
Console.WriteLine("== Semantics ==");
Console.WriteLine($"Method       : {method.ToDisplayString()}");
Console.WriteLine($"Returns      : {method.ReturnType.ToDisplayString()}");
Console.WriteLine($"Declared in  : {method.ContainingType.ToDisplayString()}");
Console.WriteLine($"Parameter    : {method.Parameters.Single().Name}");

// The '+' in the method body is a symbol too: string concatenation.
BinaryExpressionSyntax plus = root.DescendantNodes().OfType<BinaryExpressionSyntax>().Single();
var op = (IMethodSymbol)model.GetSymbolInfo(plus).Symbol!;
Console.WriteLine($"'+' binds to : {op.ToDisplayString()}");

// 4. Emit. The same object model that answered those questions produces IL.
using var peStream = new System.IO.MemoryStream();
```

```
EmitResult result = compilation.Emit(peStream);

Console.WriteLine();
Console.WriteLine("== Emit ==");
Console.WriteLine($"Success      : {result.Success}");
Console.WriteLine($"Bytes emitted: {peStream.Length > 0}");
foreach (var diagnostic in result.Diagnostics)
    Console.WriteLine($"Diagnostic   : {diagnostic.Id} {diagnostic.Severity} -
        {diagnostic.GetMessage()}");

static string Excerpt(SyntaxNode node)
{
    var text = node.ToString().ReplaceLineEndings(" ");
    return text.Length <= 40 ? text : text[..37] + "...";
}
```

Running it prints:

```
== Syntax ==
UsingDirective              using System;
IdentifierName              System
ClassDeclaration            class Greeter {     public string Gre...
MethodDeclaration           public string Greet(string name) => "...
PredefinedType              string
ParameterList               (string name)
Round-trips exactly: True

== Semantics ==
Method       : Greeter.Greet(string)
Returns      : string
Declared in  : Greeter
Parameter    : name
'+' binds to : string.operator +(string, string)

== Emit ==
Success      : True
Bytes emitted: True
Diagnostic   : CS8019 Hidden - Unnecessary using directive.
```

Those are the four things, in the output. Each one is a design principle you will meet again in every chapter.

Parsing never fails. `ParseText` always returns a tree. Text that is not valid C# still parses — into a tree containing error nodes and missing tokens. Your analyzer will run on such trees constantly, because in an editor most of the code is half-typed most of the time.

The tree is lossless. `root.ToFullString() == code` is `True`. Every space, every comment, every `#region` is in the tree. This is what makes automated refactoring possible at all: a tool can rewrite one node and hand back a file that is byte-identical everywhere else.

Meaning requires a compilation. The parse step did not know what `string` is. Only after adding references and constructing a `Compilation` could the `SemanticModel` report that `+` binds to `string.operator +(string, string)`. Syntax is cheap and local; semantics are expensive and global. Almost every performance rule in this book is a consequence of that sentence.

Diagnostics are not just errors. `Emit` succeeded and still returned a diagnostic: `CS8019 Hidden`, "unnecessary using directive". Hidden diagnostics never appear in build output; they exist so an editor can gray out that `using` and offer to remove it. The same object your analyzer will produce — a `Diagnostic` — carries everything from an invisible editor hint to a build error.

The three API layers

`Microsoft.CodeAnalysis` is not one API. It is three, stacked, with very different stability and very different audiences.

Layer	Package	What it gives you
Compiler	`Microsoft.CodeAnalysis.CSharp`	Syntax trees, compilations, semantic models, symbols, operations, diagnostics, emit. Fully public and stable. Analyzers and source generators live here.
Workspaces	`Microsoft.CodeAnalysis.CSharp.Workspaces`	Solutions, projects, documents, and the ability to **apply** changes. Public and stable. Code fixes, refactorings, and standalone tools live here.
Features / IDE	`Microsoft.CodeAnalysis.Features`, `…EditorFeatures`	The actual IDE behaviors: completion, classification, navigation, the built-in refactorings. Mostly **internal**, explicitly unsupported, and free to change in any release.

The boundary between the second and third rows is a common source of disappointment for people new to this API. “Can I add a quick-info tooltip?” is a reasonable question with an unreasonable answer, and Chapter 7 gives it straight rather than making you discover it after a week of decompiling. (Completion is the rare feature with a public door; Chapter 36 shows it.)

The first two layers are what this book teaches. They are enough to build every component listed in the introduction. You will not need the workspaces layer for a while: Parts I to III stay entirely in the compiler layer, and the workspaces row only becomes yours in Chapter 15, at the moment a diagnostic stops describing a problem and starts editing the document.

Gotcha. The line between the second and third rows is the line between a component that keeps working and one that breaks on a Tuesday. The supported surface is what a package on nuget.org exposes as public API: the compiler layer, the workspaces layer, and the abstract classes you derive from. The tempting surface is everything you can reach once the features assemblies are loaded in the same process — the IDE’s own refactorings, its completion providers, its formatting engine internals. They are reachable, they are sometimes even public, and they carry no compatibility promise at all: the IDE ships them for itself and updates them with itself. Chapter 36 shows what that looks like from the other side, when the same component behaves differently in Visual Studio, in VS Code, and in a command-line build.

Version note. A component that only reports diagnostics or generates source needs the compiler layer alone. Adding a code fix pulls in the workspaces layer, which is why an analyzer package that ships fixes has a slightly larger dependency surface — and why the two are often split into separate assemblies. Chapter 26 packages both.

Where Roslyn runs

There is more than one Roslyn on your machine, and knowing which one is executing your component matters when behavior differs.

In the compiler. `dotnet build` runs `csc` from the .NET SDK — usually inside a long-lived server process, `VBCSCompiler`, kept warm between builds so the JIT and the reference metadata are not paid for every time. Your analyzer and your generator run here, in-process, once per compilation.

In Visual Studio. The IDE hosts its own copy of Roslyn and keeps a live model of the entire solution in memory. Part of it runs inside `devenv` on .NET Framework, and most analysis runs in a separate out-of-process service on modern .NET — so the same analyzer assembly can be loaded by both runtimes. Your analyzer runs here too — continuously, on partial code, on a background thread, while the user types.

In Visual Studio Code. The C# extension runs `Microsoft.CodeAnalysis.LanguageServer`, a Roslyn-based language server, on .NET. Same compiler, different host, communicating over the Language Server Protocol (LSP).

In your own process. Anything you write with `MSBuildWorkspace` or a hand-built `Compilation` hosts Roslyn itself, with none of the constraints above. Part VII lives here.

In Rider. JetBrains Rider and ReSharper run their own analysis engine, written independently of Roslyn — and they host Roslyn next to it. Your analyzer, your generator, your code fix, and your diagnostic suppressor all run there, and their results are merged into Rider's own inspections. What does not carry over is anything that depends on Visual Studio's internals. Chapter 36 draws the line precisely.

This matters because the same analyzer assembly is loaded by hosts running different runtimes and different Roslyn versions. It is why analyzer projects in this book target `netstandard2.0`, and why Chapter 27 is about versioning.

Immutability, and why the API looks the way it does

Everything in the Roslyn object model is immutable. A `SyntaxTree` cannot be edited; `WithX` methods return a new tree. A `Compilation` cannot be modified; `AddSyntaxTrees` returns a new compilation. `ISymbol` instances are snapshots that belong to exactly one compilation.

This is not aesthetic preference. Immutability is what allows the IDE to hand your analyzer a consistent snapshot of a solution while the user keeps typing into a newer one, to run dozens of analyzers over the same compilation on many threads without locks, and to keep several versions of a file alive at once for undo and preview.

Two practical consequences you should internalize now, because they cause real bugs later:

Never cache Roslyn objects across compilations. A symbol from yesterday's compilation is meaningless in today's, and holding one alive pins the entire compilation — every syntax tree, every metadata reference — in memory. If that feels theoretical now, it does not stay that way: Chapter 21 turns this one sentence into a measurable generator failure at multi-gigabyte scale.

Never compare symbols with ==. Two `ISymbol` instances can represent the same symbol without being the same object. Use `SymbolEqualityComparer`. Chapter 4 explains why, and the compiler ships an analyzer that flags the mistake.

Under the hood. Immutable trees would be prohibitively expensive if editing one character copied the whole file. Roslyn avoids that with a two-layer representation: an internal "green" tree of small, parent-less, shareable nodes, and the public "red" tree of `SyntaxNode` objects created lazily on demand as you navigate, each knowing its parent and absolute position. Typing a character reuses almost every green node from the previous version. You never touch the green layer directly, but it explains two things you will notice: navigating a tree allocates, and a `SyntaxNode` is meaningless without the tree it came from.

Roslyn, MSBuild, and the SDK

`dotnet build` is MSBuild. MSBuild resolves references, decides whether anything needs compiling, and then invokes the `Csc` task, which runs Roslyn with a long command line. Analyzers are passed on that command line as `/analyzer:` arguments; generators are passed exactly the same way, because to the compiler a generator **is** an analyzer assembly with a different interface inside.

That is the entire integration, and it is worth remembering because it makes several later problems obvious rather than mysterious:

- If your analyzer is not running, the question is whether it made it onto the `csc` command line — an MSBuild question, answered by a binary log.
- If your generator sees the wrong value for an MSBuild property, the property was not marked as compiler-visible — again MSBuild, covered in Chapter 11.
- If your package's analyzer does not load for consumers, it is in the wrong folder inside the `.nupkg` — Chapter 26.

MSBuild In Depth, the companion volume, covers the build side in detail. This book assumes only that `dotnet build` eventually runs `csc`.

What this book calls things

A few terms are used precisely from here on:

- **Syntax** is structure without meaning: the shape of the text.
- **Semantics** is meaning: what the names refer to, in the context of a compilation with references.
- **Binding** is the process that turns the first into the second.
- A **symbol** is a declared thing — a type, method, property, parameter.
- An **operation** is a semantic description of executable code, above syntax.
- A **diagnostic** is a message with an ID, a severity, and a location.
- A **component** means any of the things you can plug into Roslyn: an analyzer, a fix, a refactoring, a suppressor, or a generator.

Carry forward

- The compiler is a library, and every intermediate result it computes — tokens, trees, symbols, operations, diagnostics — is a public API.
- Parsing never fails and the tree is lossless; meaning requires a `Compilation` with references. Syntax is cheap and local, semantics expensive and global.
- `Microsoft.CodeAnalysis` is three stacked APIs: the compiler layer (stable, what components use), the workspaces layer (for tools that own a solution), and the features layer (not a supported extensibility surface).
- Everything is immutable, which is what makes concurrent analysis and IDE snapshots possible — and what makes a cached symbol a memory leak.
- The vocabulary is fixed from here: syntax, semantics, binding, symbol, operation, diagnostic, and **component** for anything you plug in.

Exercises

1. Change the sample's `code` string to something that does not compile — delete the closing brace — and print `tree.GetDiagnostics()`. Confirm that parsing still returns a tree, and look at what the error nodes contain.
2. Remove the `MetadataReference` from the `CSharpCompilation.Create` call and run the sample again. Which of the four sections still work? What does `model.GetSymbolInfo(plus).Symbol` return now, and what does `GetSymbolInfo(plus).CandidateReason` say?
3. Print `root.ToString()` and `root.ToFullString()` for a file that starts with a comment. Explain the difference in terms of leading trivia.
4. Add a second syntax tree to the compilation that declares a class using `Greeter`, and ask the semantic model of **that** tree about the type. Confirm that symbols cross file boundaries but belong to one compilation.

Expected observations

Predict first, then check. A different answer is worth understanding, not correcting away.

1. Run: `dotnet run --project samples/ch01-intro/HelloRoslyn`
   One diagnostic, `CS1513 "} expected"`, positioned at the very end of the text — and a tree all the same. `root.Kind()` is still `CompilationUnit`, `root.ContainsDiagnostics` is `True`, and `root.ToFullString()` still round-trips to your broken text exactly. The error is carried as a **missing token**: `DescendantTokens().Where(t => t.IsMissing)` finds a `CloseBraceToken` whose full span is zero characters wide. The parser invented the token the grammar required, marked it missing, and carried on.
   Takeaway: A missing token is a real token with no text. That is how "parsing never fails" is implemented, and it is why an analyzer running in an editor must never assume a node it navigated to has any text behind it.
2. The program never reaches `Emit`: it crashes with a `NullReferenceException` on the `'+' binds to` line. The syntax section and the round-trip still work — they are pure syntax and never needed a reference — and the semantics section starts well: `GetDeclaredSymbol` still gives you `Greeter.Greet(string)`, because **declaring** a method is a syntactic act, but its `ReturnType.TypeKind` is `Error`. Then `model.GetSymbolInfo(plus).Symbol` is `null`, `CandidateReason` is `None`, and `CandidateSymbols` is empty — not "ambiguous", not "inac-

cessible", simply nothing to bind to, because `System.String` does not exist in this compilation — and the sample's `!` promised otherwise. Replace the cast with `model.GetSymbolInfo(plus).Symbol as IMethodSymbol`, print `op?.ToDisplayString()`, and `Emit` fails with about a dozen errors, led by `CS8021` (no assembly containing `System.Object`) and `CS0518 "Predefined type 'System.String' is not defined or imported"`.

Takeaway: *`CandidateReason` is only interesting when binding **found** something and rejected it. A plain `null` symbol with `CandidateReason.None` means the question could not be asked, which in real life is usually a missing reference — and in an analyzer is the case you must handle by staying silent. The `!` that crashed the sample is the assumption an analyzer must never make.*

3. `ToString()` starts at `using System;`; `ToFullString()` starts at your comment. On the sample text with one comment line added, the two differ by exactly the length of that comment, and only `ToFullString()` equals the original. Printing `root.Span` against `root.FullSpan` shows the same thing numerically: the full span starts at 0 and the span does not.

   Takeaway: *Trivia belongs to a **token**, and the root's span begins at its first token's text. Every node has both spans; `ToString()` is for reading, `ToFullString()` is the only one that round-trips. Chapter 16 turns this into the rule that keeps code fixes from eating comments.*

4. First, the interesting failure. `AddSyntaxTrees` returns a **new** compilation, and asking the old one for a model over the new tree throws `ArgumentException: SyntaxTree is not part of the compilation`. Use the returned compilation and it works: the type of the `Greeter` field in the second file resolves, and `Locations[0].SourceTree` is the **first** tree — the symbol points back at the declaration in the other file. `SymbolEqualityComparer.Default.Equals` says it is the same symbol the first tree's model declared.

   Takeaway: *Symbols belong to a compilation, not a file. And the throw is the immutability rule from this chapter arriving one chapter early: every `AddX` returns a new object, and holding the old variable is the most common way to be confused by it.*

Chapter 20 — Incremental Generators in Depth

An incremental generator is not a function from compilation to source. It is a **pipeline**: a graph of providers, each one a stage whose output is cached and compared against the previous run. Understanding the graph is what turns a generator that works into one that is fast — and this chapter is about the graph, while Chapter 21 is about the comparison.

The mental model

`Initialize` is called once. It does not generate anything; it **describes** the pipeline. From then on, on every compilation — every keystroke, in the IDE — Roslyn re-runs the stages whose inputs changed and reuses the cached outputs of the ones that did not.

Three kinds of thing in the graph:

- `IncrementalValueProvider<T>` — one value. The compilation, the parse options, an MSBuild property.
- `IncrementalValuesProvider<T>` (note the plural) — a stream of values. One per matching syntax node, one per additional file.
- **Outputs** — terminal nodes registered with `RegisterSourceOutput` and friends, which call `AddSource`.

The pipeline is a graph, and prose is a poor way to hold a graph in your head. This is the one built later in this chapter — settings files in, constant classes out — with every kind of node the model offers:

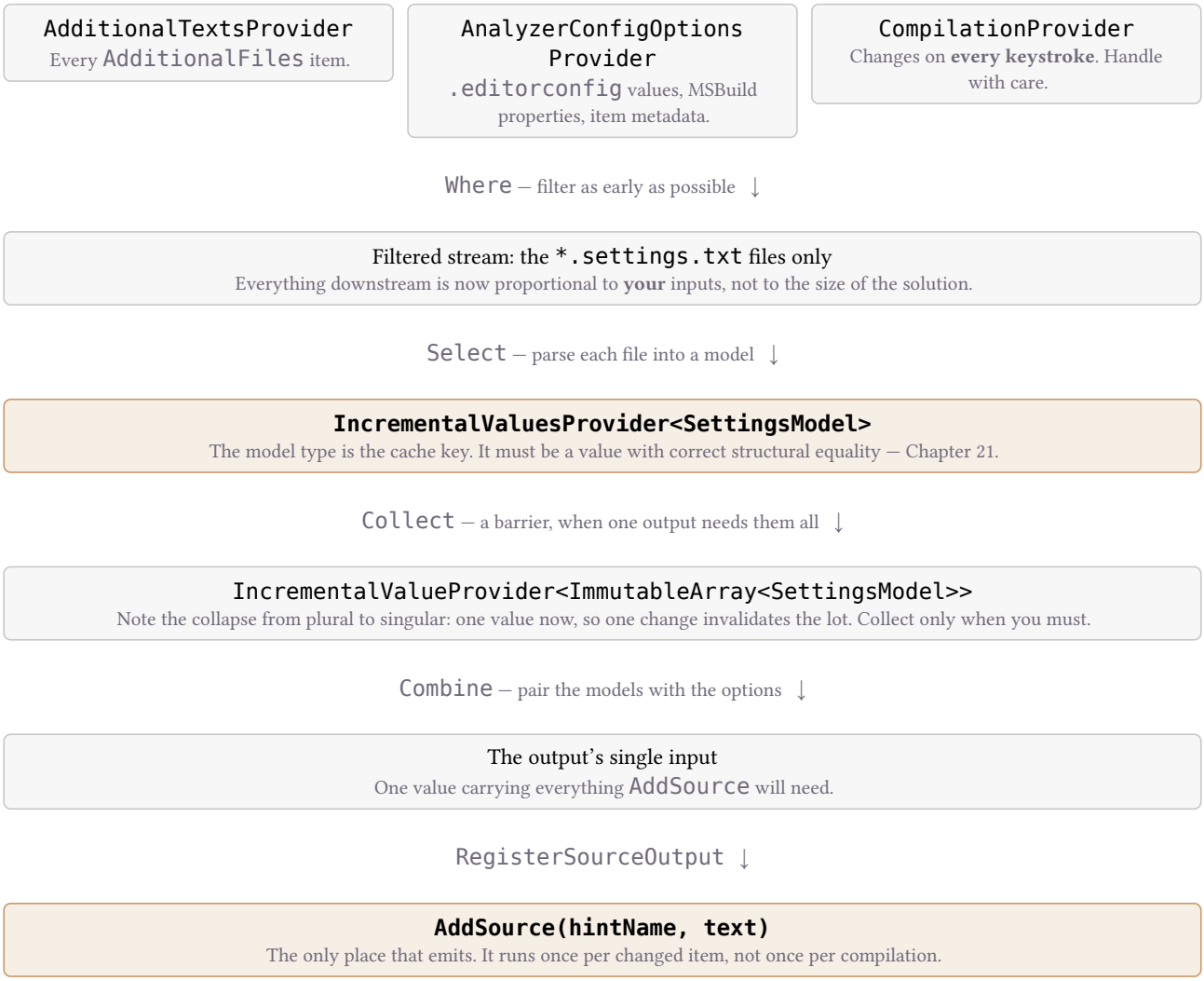


Figure 1: An incremental pipeline. Each box caches its own output; a box re-runs only when something it depends on produces a value that is **not equal** to last time's. The equality in that sentence is Chapter 21, and it is the part that goes wrong.

Two things in Figure 1 are worth naming now, because they are where generators go wrong. `Collect` turns a stream into one value, so anything downstream of it re-runs whenever **any** item changes — it is a barrier, and Chapter 21 calls it a trap for that reason. And `RegisterPostInitializationOutput`, which the marker attribute uses, is not in the figure at all: it has no inputs, so it is not part of the graph.

The sources

Provider	Gives you
`SyntaxProvider.ForAttributeWithMetadataName`	Every symbol carrying a given attribute. **The default choice** for attribute-driven generators.
`SyntaxProvider.CreateSyntaxProvider`	Every syntax node your predicate accepts. Powerful, and expensive — see below.
`CompilationProvider`	The `Compilation`. Use with extreme care (Chapter 21).
`ParseOptionsProvider`	Language version, preprocessor symbols.
`AdditionalTextsProvider`	The `AdditionalFiles` items.
`AnalyzerConfigOptionsProvider`	`.editorconfig` values, MSBuild properties, item metadata.
`MetadataReferencesProvider`	The compilation's references.

`ForAttributeWithMetadataName` versus `CreateSyntaxProvider`
This is the biggest performance decision in a generator.

`CreateSyntaxProvider(predicate, transform)` runs `predicate` over **every syntax node in every file** — millions of calls on a large solution — on the first run, and then over every node of each file as it changes: the file being edited, on every keystroke. `transform` runs on every match after every edit. It is the general mechanism, and it is what all pre-2022 generators used.

`ForAttributeWithMetadataName(name, predicate, transform)` uses an index the compiler already maintains: which files mention which attribute names. Files that do not mention the attribute are never visited. On a solution where three files out of two thousand carry your attribute, the difference is roughly three files' worth of work against two thousand files' worth.

Use `CreateSyntaxProvider` only when the trigger is not an attribute — a particular syntactic shape, a call to a specific method. And when you do, make the predicate as cheap as physically possible: syntax only, no semantic model, no allocations, ideally one type check and one token comparison.

Configuration, in one page

The pipeline below reads two MSBuild values. How MSBuild values reach a compiler component at all is the subject of Chapter 11. It sits in the analyzer part of the book because that is where the subject first comes up, but nothing in it is analyzer-specific except severity, and the generator route in the Introduction sends you through it for that reason. If you took that route, the page below is a recap and you can read it at speed; if you arrived here another way, it is the part of Chapter 11 that a generator cannot do without, and the rest of that chapter — `.editorconfig` severity, and the precedence rules between the three channels — is worth reading before you ship.

Nothing in MSBuild is visible to the compiler by default. A property the consumer sets is invisible to your generator until the project — or a `.props` file your package ships — declares it visible:

```
<PropertyGroup>
  <SettingsNamespace>Contoso.Config</SettingsNamespace>
</PropertyGroup>
<ItemGroup>
  <CompilerVisibleProperty Include="SettingsNamespace" />
</ItemGroup>
```

The build then writes the declared values into a generated `.editorconfig`-style file and passes it to the compiler on the `csc` command line, which is where `AnalyzerConfigOptionsProvider` reads them from. Two declarations, two prefixes, two places to look them up:

Declared with	Arrives as	Read from
`CompilerVisibleProperty Include="P"`	`build_property.P`	`GlobalOptions` — a property belongs to the whole compilation, so there is no per-file variant.
`CompilerVisibleItemMetadata Include="AdditionalFiles" MetadataName="M"`	`build_metadata.AdditionalFiles.M`	`GetOptions(additionalText)` — the metadata of **that** item, which is how one input file gets a different class name from the next.

Both arrive as strings, always, and both may be absent: the consumer may not have set the property, and the package may not have shipped the `.props` that declares it. Read defensively — `TryGetValue`, then a default you are willing to ship — because a generator that throws on a missing option fails as `CS8785` in somebody else's build, with your name on it (Chapter 23).

Performance. Take the **value** out of the options provider, not the provider itself. `Select`ing the whole `AnalyzerConfigOptionsProvider` into the one string you need means an unrelated `.editorconfig` edit no longer invalidates anything downstream — which is stage 1 of the pipeline below, and the reason it is written the way it is.

A pipeline that uses most of it

The generator: turn `*.settings.txt` files into constant classes.

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;
using Microsoft.CodeAnalysis.Text;

namespace Ch20.Generator;

/// <summary>
/// Turns '*.settings.txt' files into strongly typed constant classes, and emits
/// an index of everything it found.
///
/// The point of this generator is its *pipeline*: additional files, per-file
/// options, a global MSBuild property, a filter, a projection, and a Collect()
/// that fans everything back in.
/// </summary>
[Generator(LanguageNames.CSharp)]
public sealed class SettingsGenerator : IIncrementalGenerator
{
    private const string NamespaceProperty = "build_property.RootNamespace";
    private const string ClassNameMetadata = "build_metadata.AdditionalFiles.SettingsClassName";

    public void Initialize(IncrementalGeneratorInitializationContext context)
    {
        // 1. A global MSBuild property. CompilerVisibleProperty puts it here.
        IncrementalValueProvider<string> rootNamespace = context.AnalyzerConfigOptionsProvider
            .Select(static (provider, _) =>
                provider.GlobalOptions.TryGetValue(NamespaceProperty, out string? value) &&
                    value.Length > 0
                    ? value
                    : "Generated");

        // 2. Every AdditionalFiles entry, narrowed to the ones we care about.
```

```
        //    Where() before the expensive step: reading the file is not free.
        IncrementalValuesProvider<AdditionalText> settingsFiles =
            context.AdditionalTextsProvider
            .Where(static file => file.Path.EndsWith(".settings.txt",
                StringComparison.OrdinalIgnoreCase));

        // 3. Pair each file with its own per-file options, then project both
        //    into a value model. Everything downstream sees strings only.
        IncrementalValuesProvider<SettingsFile> models = settingsFiles
            .Combine(context.AnalyzerConfigOptionsProvider)
            .Select(static (pair, cancellationToken) =>
            {
                (AdditionalText file, AnalyzerConfigOptionsProvider optionsProvider) = pair;

                // Per-file options come from CompilerVisibleItemMetadata.
                AnalyzerConfigOptions options = optionsProvider.GetOptions(file);
                string className = options.TryGetValue(ClassNameMetadata, out string? name) &&
                    name.Length > 0
                    ? name
                    : DefaultClassName(file.Path);

                return Parse(className, file, cancellationToken);
            });

        // 4. One output per file, with the namespace combined in.
        context.RegisterSourceOutput(
            models.Combine(rootNamespace),
            static (ctx, pair) =>
            {
                (SettingsFile model, string ns) = pair;
                ctx.AddSource($"{model.ClassName}.g.cs", SourceText.From(EmitClass(ns, model),
                    Encoding.UTF8));
            });

        // 5. One output for all files together. Collect() turns a stream into a
        //    single value -- and is a barrier: nothing downstream of it runs
        //    until every upstream item is available.
        context.RegisterSourceOutput(
            models.Collect().Combine(rootNamespace),
            static (ctx, pair) =>
            {
                (ImmutableArray<SettingsFile> all, string ns) = pair;
                if (all.IsEmpty)
                    return;

                ctx.AddSource("SettingsIndex.g.cs", SourceText.From(EmitIndex(ns, all),
                    Encoding.UTF8));
            });
    }

    private static string DefaultClassName(string path)
    {
        string file = System.IO.Path.GetFileName(path);
        string stem = Sanitize(file.Substring(0, file.Length - ".settings.txt".Length));
        return char.ToUpperInvariant(stem[0]) + stem.Substring(1) + "Settings";
    }

    private static SettingsFile Parse(string className, AdditionalText file,
        System.Threading.CancellationToken cancellationToken)
    {
        var entries = ImmutableArray.CreateBuilder<SettingsEntry>();
        SourceText? text = file.GetText(cancellationToken);
```

```
        if (text is not null)
        {
            foreach (TextLine line in text.Lines)
            {
                string content = line.ToString().Trim();
                if (content.Length == 0 || content[0] == '#')
                    continue;

                int equals = content.IndexOf('=');
                if (equals <= 0)
                    continue;

                string key = content.Substring(0, equals).Trim();
                string value = content.Substring(equals + 1).Trim();
                if (key.Length > 0)
                    entries.Add(new SettingsEntry(Sanitize(key), value));
            }
        }

        return new SettingsFile(className, new
            EquatableArray<SettingsEntry>(entries.ToImmutable()));
    }

    private static string Sanitize(string raw)
    {
        var text = new StringBuilder(raw.Length);
        foreach (char c in raw)
            text.Append(char.IsLetterOrDigit(c) ? c : '_');

        if (text.Length == 0 || char.IsDigit(text[0]))
            text.Insert(0, '_');

        return text.ToString();
    }

    private static string EmitClass(string ns, SettingsFile model)
    {
        var text = new StringBuilder();
        text.AppendLine("// <auto-generated/>");
        text.AppendLine("#nullable enable");
        text.AppendLine();
        text.AppendLine($"namespace {ns}");
        text.AppendLine("{");
        text.AppendLine($"    internal static class {model.ClassName}");
        text.AppendLine("    {");

        foreach (SettingsEntry entry in model.Entries)
            text.AppendLine($"        public const string {entry.Key} =
                {Literal(entry.Value)};");

        text.AppendLine("    }");
        text.AppendLine("}");
        return text.ToString();
    }

    private static string EmitIndex(string ns, ImmutableArray<SettingsFile> files)
    {
        var text = new StringBuilder();
        text.AppendLine("// <auto-generated/>");
        text.AppendLine("#nullable enable");
        text.AppendLine();
        text.AppendLine($"namespace {ns}");
```

```
        text.AppendLine("{");
        text.AppendLine("    internal static class SettingsIndex");
        text.AppendLine("    {");
        text.AppendLine("        public static readonly string[] Files =");
        text.AppendLine("        {");

        foreach (SettingsFile file in files.OrderBy(f => f.ClassName, StringComparer.Ordinal))
            text.AppendLine($"            {Literal(file.ClassName)},");

        text.AppendLine("        };");
        text.AppendLine("    }");
        text.AppendLine("}");
        return text.ToString();
    }

    private static string Literal(string value) =>
        "\"" + value.Replace("\\", "\\\\").Replace("\"", "\\\"") + "\"";
}

internal readonly record struct SettingsEntry(string Key, string Value);

internal readonly record struct SettingsFile(string ClassName, EquatableArray<SettingsEntry>
    Entries);
```

Five stages, numbered in the source. Reading them in order:

1. A global MSBuild property via `AnalyzerConfigOptionsProvider.Select`. `GlobalOptions` holds MSBuild properties (prefixed `build_property.`) that the project marked `CompilerVisibleProperty`. The `Select` turns the whole options provider into a single `string`, so everything downstream depends on the **value** rather than on the provider — which means a change to an unrelated option does not invalidate anything.

2. `Where` before the expensive work. `AdditionalTextsProvider` yields every additional file in the project; the filter cuts it to the relevant ones before anything reads their contents.

3. `Combine` to pair a stream with a value. `settingsFiles.Combine(optionsProvider)` produces a stream of tuples. `Select` then does the parsing — and crucially, returns a **value model** made of strings, never an `AdditionalText` or a symbol.

Per-file options come from `GetOptions(additionalText)`, and they are populated by `CompilerVisibleItemMetadata` in the project file. That is how one file gets a custom class name.

4. `RegisterSourceOutput` per item. The models are combined with the namespace so the callback has everything it needs, and each one produces a file.

5. `Collect()` to fan back in. `models.Collect()` turns `IncrementalValuesProvider<T>` into `IncrementalValueProvider<ImmutableArray<T>>` — every model, in one value. That is what the index file needs.

Performance. `Collect()` is a barrier. Nothing downstream of it can run until every upstream item is available, and any change to any item invalidates the whole collected value. Use it when you need all the items together (an index, a registry, a dispatch table); never as a convenience.

The combinators

Operator	Effect
`Select`	One in, one out. The workhorse.
`SelectMany`	One in, many out. Flattening a file into its entries.
`Where`	Filter. Put it as early as possible.
`Collect`	Stream to single array. A barrier.
`Combine`	Pair a provider with another. Values with values, or a stream with a value.
`WithComparer`	Override how this stage's outputs are compared — Chapter 21.
`WithTrackingName`	Name a stage so tests can assert on it — Chapter 21.

Note what is **not** there: no `Zip`, no ordering guarantees, no way to make one stream depend on another item-wise. If you find yourself wanting those, the model is telling you the design is wrong.

The outputs

`RegisterSourceOutput(provider, action)` — add source. The normal one.

`RegisterImplementationSourceOutput(provider, action)` — add source that nothing else in the compilation can **reference**: method bodies for `partial` declarations rather than new API. It is a promise that a host may skip this output when it needs only the shape of the code. The Roslyn IDE does not take that option today — it runs implementation outputs like any other — so the registration documents intent rather than saving time. Use it only where the promise is true; a host that did skip it would leave any reference to the output unresolved.

`RegisterPostInitializationOutput(action)` — add source before anything is analyzed, with no inputs at all. For marker attributes and other fixed content, as Chapter 19 showed. Its context carries one extra member, `AddEmbeddedAttributeDefinition`: call it, mark the attribute you emit `[global::Microsoft.CodeAnalysis.Embedded]`, and that attribute becomes invisible to every other assembly — including the ones granted `InternalsVisibleTo`, which `internal` alone does not cover.

Sidebar: pre-compilation output

Version note. This sidebar is about an API newer than the book's 4.14 floor, and still experimental. Nothing later in the book depends on it.

`RegisterPreCompilationSourceOutput(provider, action)`, new in Roslyn 5.9, adds source **to the initial compilation**, from inputs. Using it raises `RSEXPERIMENTAL007`, which you must suppress deliberately.

It fills the one hole in the list above. Post-initialization writes into the initial compilation but takes no inputs; `RegisterSourceOutput` takes inputs but its output is never fed back, so nothing — not the generator itself, not another generator — can bind against it. Pre-compilation does both: it reads providers, and its source lands in the compilation that the standard phase then sees.

The cost of that is a hard restriction. The provider **must not** touch the compilation or syntax nodes, because neither exists yet. `AdditionalTextsProvider`, `ParseOptionsProvider`, and `AnalyzerConfigOptionsProvider` are legal; `CompilationProvider` and `SyntaxProvider` are not. Nothing stops you at compile time — the provider types are the same ones — so the failure is a run-time `InvalidOperationException` that puts the generator in an error state and skips its standard phase entirely. Other generators keep running.

The callback receives a `PreCompilationSourceProductionContext`, not a `SourceProductionContext`. It has `AddSource` and `CancellationToken` and nothing else: there is no `ReportDiagnostic`, by design. Diagnostics belong in an analyzer (Chapter 23).

```
var protoFiles = context.AdditionalTextsProvider
    .Where(static f => f.Path.EndsWith(".proto"));

context.RegisterPreCompilationSourceOutput(protoFiles, static (ctx, file) =>
    ctx.AddSource($"{Path.GetFileName(file.Path)}.g.cs", GenerateTypes(file)));
```

```
// The compilation now contains those types.
context.RegisterSourceOutput(context.CompilationProvider, static (ctx, compilation) =>
{
    var t = compilation.GetTypeByMetadataName("MyProto.MyMessage"); // non-null
    ctx.AddSource("Serializers.g.cs", GenerateSerializer(t!));
});
```

Note `Path.GetFileName`: `file.Path` is absolute, and an absolute path is not a legal hint name — `AddSource` throws on the leading `/`, or on the `:` of a Windows drive letter.

Razor is the reason it exists. It needed its own declarations back in a compilation and was faking it by cloning the compilation, binding against the copy, and throwing the copy away. A pre-compilation output deletes the clone — roughly a **50%** improvement in Razor generation in early measurements — and adds no compilation phase, because the trees go into the initial compilation via `AddSyntaxTrees`, where post-init trees already go. It also answers, sideways, the long-standing request for configurable marker attributes.

The version floor matters more than the novelty. A generator following Chapter 27 compiles against Roslyn 4.14, and this API does not exist there. Using it means either raising your floor to 5.9 — giving up every host older than that — or shipping it only in a `roslyn5.9` variant (Chapter 27). For most generators, in most packages, that is not yet a trade worth making. Watch it; do not reach for it.

Reading MSBuild from a generator

The project side, from the consumer:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <RootNamespace>Ch20.Consumer</RootNamespace>

    <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
    <CompilerGeneratedFilesOutputPath>generated</CompilerGeneratedFilesOutputPath>
  </PropertyGroup>

  <ItemGroup>
    <!-- The generator reads RootNamespace, which is only visible if the project
         says so. -->
    <CompilerVisibleProperty Include="RootNamespace" />

    <!-- …and per-file metadata, which needs its own declaration. -->
    <CompilerVisibleItemMetadata Include="AdditionalFiles" MetadataName="SettingsClassName" />
  </ItemGroup>

  <ItemGroup>
    <AdditionalFiles Include="app.settings.txt" />
    <AdditionalFiles Include="database.settings.txt" SettingsClassName="DbSettings" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="../Ch20.Generator/Ch20.Generator.csproj"
                      OutputItemType="Analyzer"
                      ReferenceOutputAssembly="false" />
  </ItemGroup>

  <ItemGroup>
    <Compile Remove="generated/**/*.cs" />
  </ItemGroup>

</Project>
```

These are the two declarations from "Configuration, in one page" as a consumer really writes them, with both halves visible at once: the property and the item metadata, next to the settings files the metadata applies to. Note that the consumer is doing the declaring here. That is the right thing for a sample and the wrong thing for a shipped package — expecting every user to write `CompilerVisibleProperty` for an option your generator invented is how a generator ends up silently doing nothing for most of the people who install it. Ship a `.props` file that declares it for them (Chapter 26).

The input files:

```
# Keys become constants; everything after the first '=' is the value.
ApplicationName = Ledger
SupportEmail    = support@example.com
max retries     = 3
```

and the code that uses the result:

```
namespace Ch20.Consumer;

public static class Usage
{
    // AppSettings comes from app.settings.txt (default class name).
    public static string Name => AppSettings.ApplicationName;

    // DbSettings comes from database.settings.txt, renamed by item metadata.
    public static string Connection => DbSettings.ConnectionName;

    // 'max retries' is not a valid identifier; the generator sanitized it.
    public static string Retries => AppSettings.max_retries;

    // And the index knows about both files.
    public static int FileCount => SettingsIndex.Files.Length;
}
```

What the compiler wrote:

```
// <auto-generated/>
#nullable enable

namespace Ch20.Consumer
{
    internal static class AppSettings
    {
        public const string ApplicationName = "Ledger";
        public const string SupportEmail = "support@example.com";
        public const string max_retries = "3";
    }
}
```

```
// <auto-generated/>
#nullable enable

namespace Ch20.Consumer
{
    internal static class SettingsIndex
    {
        public static readonly string[] Files =
        {
            "AppSettings",
            "DbSettings",
        };
```

```
    }
}
```

The second file came from the `Collect()` stage; the first from the per-item stage. `DbSettings.g.cs` — not shown — came from the same per-item stage with a class name supplied by item metadata.

Testing a pipeline

Driving a generator with additional files and options needs two small test doubles, and they are worth having:

```
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Threading;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;
using Microsoft.CodeAnalysis.Text;

namespace Ch20.Generator.Tests;

/// <summary>An AdditionalFiles entry backed by a string.</summary>
internal sealed class InMemoryAdditionalText(string path, string text) : AdditionalText
{
    private readonly SourceText _text = SourceText.From(text);

    public override string Path { get; } = path;

    public override SourceText GetText(CancellationToken cancellationToken = default) => _text;
}

/// <summary>
/// The options a generator sees: a global set (MSBuild properties) and a
/// per-file set (item metadata).
/// </summary>
internal sealed class InMemoryOptions(
    Dictionary<string, string> globals,
    Dictionary<string, Dictionary<string, string>>? perFile = null) :
        AnalyzerConfigOptionsProvider
{
    private readonly Dictionary<string, Dictionary<string, string>> _perFile = perFile ?? [];

    public override AnalyzerConfigOptions GlobalOptions { get; } = new Options(globals);

    public override AnalyzerConfigOptions GetOptions(SyntaxTree tree) => new Options([]);

    public override AnalyzerConfigOptions GetOptions(AdditionalText textFile) =>
        new Options(_perFile.TryGetValue(textFile.Path, out var options) ? options : []);

    private sealed class Options(Dictionary<string, string> values) : AnalyzerConfigOptions
    {
        public override bool TryGetValue(string key, out string value) =>
            values.TryGetValue(key, out value!);
    }
}
```

```
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Xunit;

namespace Ch20.Generator.Tests;
```

```
public sealed class SettingsGeneratorTests
{
    private static GeneratorDriverRunResult Run(
        IEnumerable<AdditionalText> additionalFiles,
        Dictionary<string, string>? globals = null,
        Dictionary<string, Dictionary<string, string>>? perFile = null)
    {
        var compilation = CSharpCompilation.Create(
            "Tests",
            [CSharpSyntaxTree.ParseText("class Placeholder { }")],
            Basic.Reference.Assemblies.Net100.References.All,
            new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary));

        GeneratorDriver driver = CSharpGeneratorDriver.Create(
            generators: [new SettingsGenerator().AsSourceGenerator()],
            additionalTexts: additionalFiles,
            optionsProvider: new InMemoryOptions(
                globals ?? new Dictionary<string, string> { ["build_property.RootNamespace"] =
                    "Demo" },
                perFile));

        return driver.RunGenerators(compilation).GetRunResult();
    }

    private static string SourceOf(GeneratorDriverRunResult result, string hintName) =>
        result.Results.Single().GeneratedSources.Single(s => s.HintName ==
            hintName).SourceText.ToString();

    private static readonly AdditionalText AppSettings =
        new InMemoryAdditionalText("/proj/app.settings.txt", """
            # a comment
            ApplicationName = Ledger
            max retries     = 3
            """);

    [Fact]
    public void A_settings_file_becomes_a_constants_class()
    {
        var result = Run([AppSettings]);

        string source = SourceOf(result, "AppSettings.g.cs");
        Assert.Contains("internal static class AppSettings", source);
        Assert.Contains("public const string ApplicationName = \"Ledger\";", source);
    }

    [Fact]
    public void Keys_that_are_not_identifiers_are_sanitized()
    {
        var result = Run([AppSettings]);
        Assert.Contains("public const string max_retries = \"3\";", SourceOf(result,
            "AppSettings.g.cs"));
    }

    [Fact]
    public void The_namespace_comes_from_the_MSBuild_property()
    {
        var result = Run([AppSettings], globals: new() { ["build_property.RootNamespace"] =
            "Acme.App" });
        Assert.Contains("namespace Acme.App", SourceOf(result, "AppSettings.g.cs"));
    }

    [Fact]
```

```
    public void Item_metadata_overrides_the_class_name()
    {
        var result = Run(
            [AppSettings],
            perFile: new()
            {
                ["/proj/app.settings.txt"] = new()
                    { ["build_metadata.AdditionalFiles.SettingsClassName"] = "Renamed" },
            });

        Assert.Contains("internal static class Renamed", SourceOf(result, "Renamed.g.cs"));
    }

    [Fact]
    public void Files_that_do_not_match_the_pattern_are_ignored()
    {
        var result = Run([new InMemoryAdditionalText("/proj/readme.txt", "Key = Value")]);
        Assert.Empty(result.Results.Single().GeneratedSources);
    }

    [Fact]
    public void Collect_produces_one_index_for_all_files()
    {
        var result = Run([
            AppSettings,
            new InMemoryAdditionalText("/proj/database.settings.txt", "ConnectionName =
                primary"),
        ]);

        string index = SourceOf(result, "SettingsIndex.g.cs");
        Assert.Contains("\"AppSettings\",", index);
        Assert.Contains("\"DatabaseSettings\",", index);
    }
}
```

`CSharpGeneratorDriver.Create` takes the generators, the additional texts, and the options provider, so every input the pipeline reads can be supplied from a test. `GetRunResult()` gives you the generated sources by hint name.

A note on `CompilationProvider`

It is in the table, and it is the most dangerous entry there.

`context.CompilationProvider` yields the `Compilation` — which changes on every keystroke, by definition. Anything combined with it re-runs every time, and anything derived from it that flows onward carries a reference to the whole compilation.

There are legitimate uses: reading `compilation.AssemblyName`, checking whether a well-known type exists, reading the language version. In every case, `Select` it down to a small value **immediately**:

```
IncrementalValueProvider<bool> hasType = context.CompilationProvider
    .Select(static (c, _) => c.GetTypeByMetadataName("Some.Type") is not null);
```

The provider now yields a `bool`. Keystrokes that do not change the answer change nothing downstream. Chapter 21 explains why that matters.

Review your pipeline before you read Chapter 21

This chapter has been about the **shape** of the graph: what the sources are, what the combinators do, where the outputs hang off it. The next chapter is about why that graph caches — and it will be much easier to read with your own pipeline drawn out first, because most of its warnings are about a shape you can already check for.

So before turning the page, take the generator you are writing (or the sample's) and walk this list. Every item is answerable from the `Initialize` method alone, with no knowledge of equality at all.

Pipeline review: shape only

- The **source** is `ForAttributeWithMetadataName` unless you can say why it cannot be. If it is `CreateSyntaxProvider`, its predicate is syntax-only and cheap, and rejects almost everything.
- Filtering happens **early** — `Where` before `Select`, and the predicate before the transform — so the expensive stages see as few items as possible.
- Each item is turned into a **model** of your own making, and no stage after that point holds an `ISymbol`, `SyntaxNode`, `Compilation`, or `Diagnostic`. You can check this by reading the generic argument of every provider in the chain.
- `Collect` appears only where an output needs every item at once — an index file, a registration table. Per-item outputs hang off the uncollected provider.
- Anything derived from `CompilationProvider` is `Select`-ed down to a primitive — a `bool`, a `string`, an enum — in the same expression that introduced it.
- MSBuild values arrive through `CompilerVisibleProperty` or `CompilerVisibleItemMetadata`, and are projected to the value you need rather than carried as an options provider.

A "no" on any of those is a shape problem, and it is cheaper to fix here than after Chapter 21 has told you what it costs. A row of "yes" does not mean your generator caches — that is precisely what the next chapter is for — but it means the reasons it might not are down to the **types** flowing through the graph rather than the graph itself.

Carry forward

- `Initialize` runs once and **describes** a pipeline. It does not generate anything; the pipeline does, on every compilation, re-running only the stages whose inputs changed.
- `ForAttributeWithMetadataName` is the default choice, and the biggest performance decision in a generator. `CreateSyntaxProvider` visits every node in every file on the first run, and every node of the edited file on each keystroke after that.
- `Where` early, `Select` to a model, `Collect` only when an output needs every item — `Collect` is a barrier, and everything after it re-runs when any one item changes.
- MSBuild values reach a generator only through `CompilerVisibleProperty` or `CompilerVisibleItemMetadata`, arriving as `build_property.*` and `build_metadata.*` on the options provider.
- `CompilationProvider` changes on every keystroke. Depending on it directly defeats the whole model — Chapter 21 shows what to do instead.

Exercises

1. Add a third settings file to the consumer and rebuild. Confirm the index file updates and the other two class files do not change.
2. Rewrite stage 3 so the model holds the `AdditionalText` instead of parsed strings, and read Chapter 21 immediately afterward.
3. Replace `ForAttributeWithMetadataName` in Chapter 19's generator with `CreateSyntaxProvider` and a predicate that checks for any attribute list. Count how many times the predicate runs on a large project.
4. Move the `Where` filter in stage 2 to **after** the `Select` that parses each file. Explain what work that wastes, and on which keystrokes.

Expected observations

Predict first, then check. A different answer is worth understanding, not correcting away.

1. Run: `dotnet build samples/ch20-incremental-generators/Ch20.Consumer`, then `git status` and `git diff` from the archive root
   Exactly two things changed. `SettingsIndex.g.cs` gains one line — `"CacheSettings",` between the other

two, because the index is sorted — and `CacheSettings.g.cs` is new. `AppSettings.g.cs` and `DbSettings.g.cs` are untouched, byte for byte; `git status` does not list them at all.
*Takeaway: The per-item output and the `Collect()` output have different blast radii, and that is a property of the **pipeline**, not of the build. Adding an input changes one file and the index. Anything that fans in through `Collect()` changes whenever anything changes — which is why the index is a separate output rather than something the per-file stage also produces.*

2. Everything still compiles and the generated text is identical, which is the trap. Run it twice with `track IncrementalGeneratorSteps` on — Chapter 21 shows the harness — handing the driver a fresh set of `AdditionalText` objects with unchanged content the second time, and the model step's reason changes from `Unchanged` to `Modified`. Nothing about the output moved; the cache simply stopped working.
Takeaway: `AdditionalText` compares by reference, so a model holding one is equal to last run's only while the host hands you the very same object. Whenever it builds new ones with the same content — a project reload, or a test like this one — every stage downstream re-runs while producing the same source. This is the failure Chapter 21 is about, and it is invisible to every test that only asserts output.
3. On a synthetic compilation of 502 files and 29,031 syntax nodes, with one file carrying the attribute, the first run: `ForAttributeWithMetadataName` calls its predicate **once**. `CreateSyntaxProvider` calls its predicate **29,031** times — once per node in the compilation, matching the node count exactly. Both produce the same one generated file. After that, an edit re-runs the predicate only over the edited file's nodes.
Takeaway: The ratio is not a constant factor; it is the size of the solution. The predicate is cheap; running it on every node when the solution loads, and on every node of the edited file on each keystroke after that, is not, and it is work that scales with somebody else's repository rather than with your inputs. This is the measurement behind "use `ForAttributeWithMetadataName` unless you cannot".
4. The parse runs on **every** `AdditionalFiles` entry rather than on the matching ones. In a project with five additional files of which two are settings files, the count goes from 2 to 5 per run, and the three extra results are computed, allocated, and then discarded by the filter.
*Takeaway: The waste is proportional to the additional files the **consumer** happens to have — a banned-symbols list, a JSON schema, a `.editorconfig` — none of which you control or can test against. It is paid on the first run and again whenever any additional file changes. `Where` before `Select` costs one line and bounds the work by your own inputs.*

End of preview

The full edition continues with the other 36 chapters and 9 appendices.

meziantou.net

Roslyn In Depth

Analyzers, Source Generators, and Compiler-Powered Tooling

The C# compiler is a library, and everything it knows about your code — every token, every symbol, every operation — is a public API. This book turns that API into working components: analyzers that catch bugs as you type, code fixes that repair them, source generators that write code at compile time, and tools that load an entire solution to analyze or rewrite it. It goes deep on the parts that decide whether a component survives contact with a real codebase: incremental caching, testing, performance, and packaging.

What you will learn

- Read code as syntax, symbols, and operations
- Ship code fixes that preserve formatting
- Test analyzers, fixes, and generators
- Package and version against Roslyn
- Write analyzers that do not misfire
- Build incremental generators that stay incremental
- Profile and cut analyzer overhead
- Rewrite a whole solution with MSBuildWorkspace

About the Author

Gérald Barré — known online as **meziantou** — is a software developer with more than ten years of industry experience and a recognized **Microsoft MVP**.

He works across the .NET stack: C#, ASP.NET Core, Blazor, Entity Framework Core, and the tooling around them. Roslyn is not an academic interest for him: he is the author of **Meziantou.Analyzer**, a widely used set of Roslyn analyzers and code fixes with well over a hundred rules, shipped as a NuGet package and consumed by thousands of projects. Most of the advice in this book — about false positives, about performance, about packaging and versioning — comes from maintaining that package across a decade of Roslyn releases.

Gérald writes regularly at **meziantou.net**, where he has published hundreds of articles on .NET and C#, many of them about the compiler APIs this book covers. He also maintains **Meziantou.Framework** and a number of other open-source projects.

Gérald Barré
Microsoft MVP · meziantou.net

MVP Microsoft Most Valuable Professional