Roslyn syntax tree visualizer in Rider

Supervisor: Evgeniy Stepanov

Igor Manninen

April 2026

Declaration

I, Igor Manninen, being fully aware of the consequences of plagiarism, declare responsibly that this thesis entitled “Roslyn Syntax Tree Visualizer in Rider” is strictly a product of my own personal work and all sources used have been duly stated in the bibliographic citations and references. Where I have used ideas, text, and/or sources of other authors, they are clearly mentioned in the text with the appropriate citation and the relevant reference is included in the bibliographic references section with a full description.

The Declarant

Igor Manninen

Abstract

This thesis presents the design and implementation of a Roslyn Syntax Tree Visualizer for JetBrains Rider — a tool window that displays an interactive abstract syntax tree of C# source files inside the IDE, targeting developers who write Roslyn analysers and source generators.

A competitive analysis of five existing tools, a developer survey, and a hands-on development session identified feature gaps and workflow pain points. The findings were formalised into prioritised user stories and system requirements that guided the implementation and design of the tool.

The visualizer is built on Rider's architecture and uses a custom incremental tree update algorithm to overcome the absence of built-in tree diffs in the Roslyn API.

Performance benchmarks show that the Rider visualizer is on average 74% faster than Visual Studio's Syntax Visualizer. Correctness is verified by integration tests validated against gold files. The tool has been merged and released as a production feature in Rider 2025.1.

Keywords: Roslyn, abstract syntax tree, JetBrains Rider, syntax visualizer, source generators, static analysis, IDE tooling

Word count: 8,764

Contents

List of Figures

List of Tables

1. Introduction

The compiler has evolved from a simple translation tool into an interactive platform. In the .NET ecosystem, this shift is driven by the **Microsoft Roslyn Compiler Platform**, which allows developers to build custom static analysers and source generators on top of the compiler itself. Both tools rely on the **Abstract Syntax Tree (AST)** — a structured representation of source code that the compiler builds during parsing.

This introduction covers the key concepts behind this thesis: static analysis, the Roslyn platform, Roslyn Analysers, Source Generators, and why visualising the syntax tree matters for developers.

1.1 Abstract Syntax Tree

How an AST Is Built

To understand how code analysis works, it helps to first examine how a compiler processes source code. A traditional compiler transforms source text into an executable through a multi-phase pipeline [1] (Figure 1.1).

1. **Lexical analysis (tokenisation).** The raw character stream is broken into a flat sequence of tokens: keywords (if, class, return), identifiers (myVariable), literals (42, "hello"), operators (+, ==), and punctuation (;, $\{$). Comments and whitespace may be discarded at this stage or retained as metadata.
2. **Syntax analysis (parsing).** A parser consumes the token stream and checks whether it conforms to the language grammar, producing a hierarchical tree that captures the grammatical structure of the program. This tree is then simplified into an **Abstract Syntax Tree (AST)** by removing nodes that carry no semantic weight — grouping parentheses, statement-terminating semicolons, and similar punctuation whose role is already captured by the shape of the tree itself.
3. **Semantic analysis.** The AST is decorated with information that requires understanding the broader program context: type resolution, symbol binding (connecting each identifier to its declaration), scope analysis, and control- and data-flow computation. This phase produces the semantic model — a queryable object that can answer questions such as “what is the inferred type of this expression?” or “is this variable guaranteed to be assigned before its first use?”

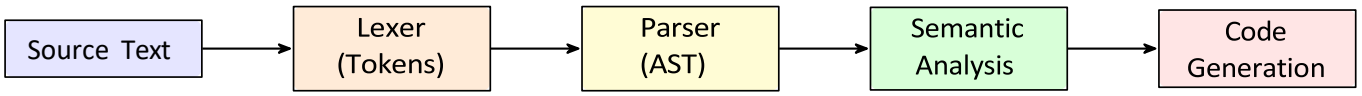

Figure 1.1: The compiler front-end pipeline. (Simplified)

Now consider a different goal: instead of compiling code, we want to *analyse* it — find potential bugs, enforce coding rules, or generate additional source files. Such analysis requires the same structured understanding of the source code that the compiler already builds: the token stream, the syntax tree, and the semantic model. Rather than re-implementing these phases from scratch, it is far more practical to reuse the compiler's own pipeline.

This is precisely the approach taken by the **Roslyn** compiler platform (Section 1.2). Roslyn exposes its internal compilation phases through a public API, allowing third-party analysers and generators to hook into the pipeline at well-defined points. The main advantages of this design are:

- **Correctness.** Analysers operate on the same data structures the compiler uses, so their view of the code is guaranteed to match what the compiler sees.
- **No duplication.** There is no need to write or maintain a separate parser, the compiler's battle-tested front end is reused directly.

The trade-off is that the developer must use raw syntax node API, how the compiler sees it, and that is why additional visualization tools are build to simplify development workflow.

Abstract Syntax Tree

An AST is a tree-shaped data structure in which each **node** represents a syntactic construct of the language. The root of the tree represents the entire compilation unit; each node's children represent its syntactic sub-components. Consider the following C# statement:

Listing 1.1: A simple C# variable declaration used to illustrate AST structure.

```
1 int x = a + b * 2;
```

The AST for this statement is shown in Figure 1.2. The top-level node is a LocalDeclarationStatement. It contains a VariableDeclarator whose initialiser is an AddExpression. The right operand of the addition is a MultiplyExpression of the identifier b and the integer literal 2. Notice how the operator precedence — the fact that multiplication binds more tightly than addition — is captured entirely by the tree structure: there is no need for parentheses.

1.2 The Roslyn Compiler Platform

The .NET Compiler Platform known by its codename **Roslyn**, is an open-source compiler and code-analysis API for C# and VB.NET, developed by Microsoft. It was released under the Apache 2.0 licence on 3 April 2014 and is hosted at

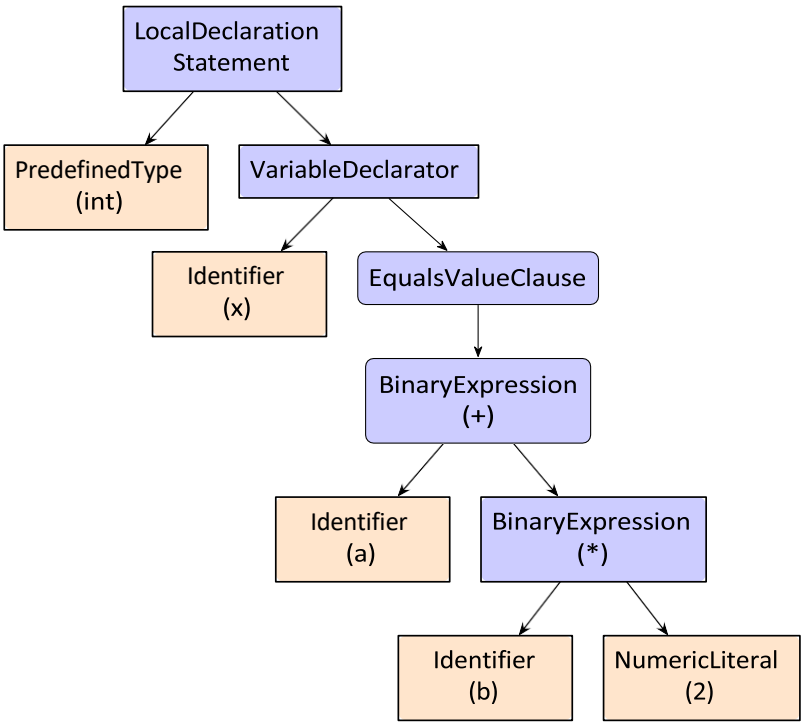

Figure 1.2: Abstract Syntax Tree for the statement int x = a + b * 2;. Blue nodes are non-terminal constructs (statements and expressions); orange nodes are terminal leaf nodes (tokens)

github.com/dotnet/roslyn [8]. Roslyn is self-hosting: the C# compiler is written in C#.

Before Roslyn, compilers were black boxes. Source code went in and an executable came out, but all the internal data the compiler built along the way — the syntax tree, the symbol table, the binding results — was discarded.

Roslyn changes this by exposing every phase of the compiler through a public API [10]. The same object model the compiler uses internally is available to IDE features and third-party tools.

The Roslyn API is split into four layers, each building on the one below.

- **Compiler APIs** - The base layer. Provides the Compilation object (a snapshot of all source files and references), the SyntaxTree (the parsed representation of one file), and the SemanticModel (type and symbol information). This layer has no dependency on any IDE and runs on all platforms.
- **Diagnostic APIs** - Built on top of the Compiler layer. This is where custom DiagnosticAnalyzer classes plug into the compilation. Their output appears in the IDE error list and in MSBuild output, just like compiler errors.
- **Scripting APIs** - Allow C# code to be executed as a script, one line at a time.
- **Workspaces APIs** The highest-level layer. Represents an entire solution — all projects, documents, and references — as a single queryable Solution object. Used for features like code formatting, code generation, and Find All References.

Properties of Roslyn Syntax Trees

Roslyn syntax trees have three important properties [14]:

- **Full fidelity.** Nothing from the source file is lost — every token, whitespace, comment, and directive is preserved. The original source text can be reproduced exactly from the tree.
- **Immutability.** A syntax tree never changes after it is created. Edits produce a new tree that reuses unchanged parts of the old one. This makes trees safe to use from multiple threads at the same time.
- **Three-tier element model.** A tree contains three kinds of elements: SyntaxNode (statements, expressions, declarations), SyntaxToken (keywords, identifiers, literals, operators), and SyntaxTrivia (whitespace, comments, directives — attached to tokens rather than existing as separate nodes).

1.3 Roslyn Analysers and Source Generators

Roslyn Analysers

A **Roslyn analyser** is a class that extends DiagnosticAnalyzer and plugs into the compiler pipeline to detect code issues [16]. Its output appears in the IDE as inline squiggles and Error List entries, identical to compiler errors. Analysers are distributed as NuGet packages or parts of SDK or as a project reference and activate automatically when added to a project.

An analyser works by registering callbacks inside the `Initialize` method. The compiler invokes them at the right point during compilation. Callbacks can target different levels: RegisterSyntaxNodeAction fires for a specific node type, RegisterSymbolAction fires for declared symbols, and RegisterCompilationAction fires once per build. **Syntax-level** callbacks work on the tree shape alone (fast, no binding needed); **semantic** callbacks query the SemanticModel to check types, symbols, and data flow.

An analyser can also include a **Code fix provider** (CodeFixProvider) that automatically corrects the detected issue.

Source Generators

Source generators are Roslyn components that run during compilation and add new C# files to the build. The files are never stored in version control — they are generated fresh on every build. This enables **compile-time code generation** with no runtime overhead, and unlike reflection-based approaches it is fully compatible with ahead-of-time (AOT) compilation.

Current best practice is to implement IIncrementalGenerator (introduced in .NET 6) [9]. A generator declares a pipeline of transformation steps inside its Initialize method. Roslyn caches each step's output and only re-runs steps whose inputs have changed.

Source generators are already used in popular .NET libraries:

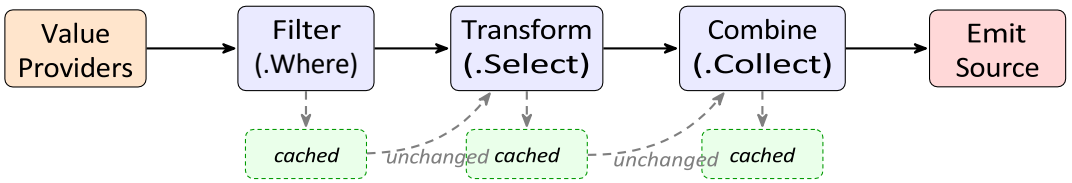

Figure 1.3: The incremental source generator pipeline. Each stage caches its output and is skipped when its inputs have not changed.

- **System.Text.Json** generates AOT-compatible serialisers [12];
- **CommunityToolkit.Mvvm** generates [INotifyPropertyChanged] properties from a single [ObservableProperty] attribute [13];
- **Regex** generates pre-compiled, allocation-free expressions via [GeneratedRegex] [15].

In each case the developer adds an attribute and the generator writes the implementation at compile time.

1.4 The Necessity of Syntax Tree Visualisation

The Core Challenge: Writing Code About Code

Writing an analyser or source generator means working with the syntax tree, not with plain text. This requires knowing exact node types: Is a method call an InvocationExpressionSyntax? Is async a child node or a token in the Modifiers list? The C# grammar has hundreds of node kinds, and many constructs nest in non-obvious ways. Without a way to inspect the tree for a real code sample, developers have to either memorise the hierarchy or find it through bruteforce.

The **Roslyn Syntax Visualiser** solves this problem. It shows a live, interactive tree of the file currently open in the editor and is described in the official Microsoft documentation as *"an essential tool to understand the models for code you want to analyze"* [11].

Rider is a widely used cross-platform IDE for .NET, but until recently it had no built-in syntax visualiser. Developers were forced to rely on external tools, which interrupted the workflow and reduced productivity. This thesis addresses that gap by designing and implementing a Roslyn Syntax Visualiser for Rider. The following chapters describe the research process — including a competitive analysis of existing tools and a definition of the required features — as well as the implementation and integration of the solution into the Rider platform.

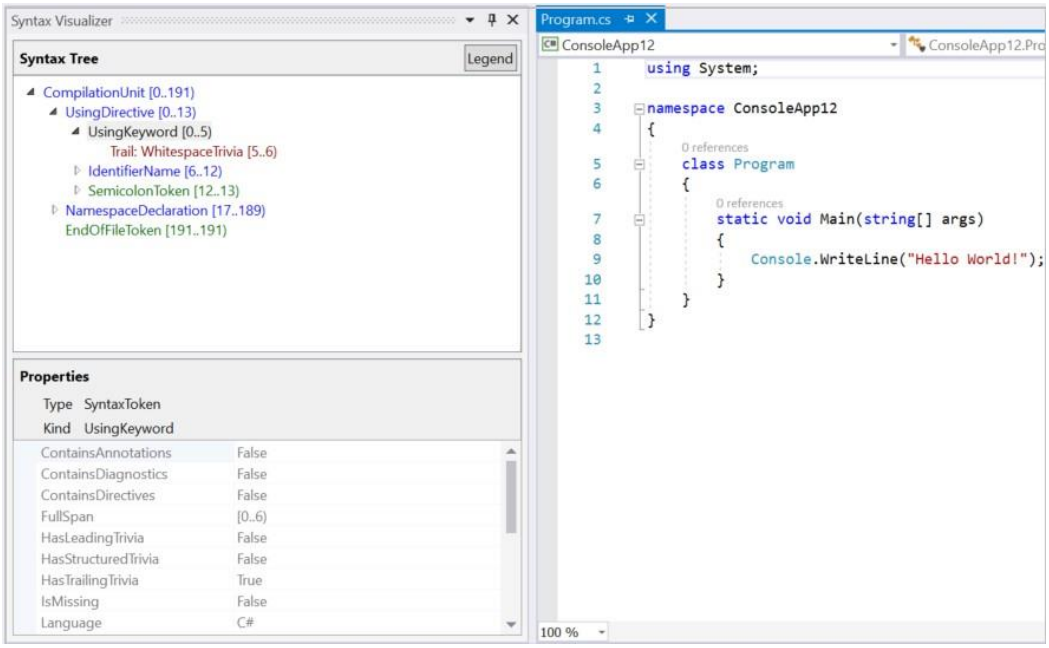

Figure 1.4: The Roslyn Syntax Visualiser in Visual Studio. Adapted from Microsoft documentation [11].

2. Market Analysis

2.1 Existing Solutions Analysis

Before designing the tool, I analyzed existing tools to identify what features are already available, where the gaps are, and what the new tool must improve upon. Five tools were evaluated:

Visual Studio Roslyn Syntax Visualizer

The Roslyn Syntax Visualizer is the official Microsoft tool, shipped as part of the .NET Compiler Platform SDK and available in Visual Studio as an optional component [11]. It provides a tool window that displays a live-updating syntax tree for the active file in the editor, showing each node's .NET type, SyntaxKind, and character span. The tool supports bidirectional navigation between the tree and the editor and offers semantic inspection through right-click actions.

As the first-party reference implementation, the Visual Studio Syntax Visualizer sets the baseline for what a syntax tree viewer should offer. However, it is limited to Windows and requires Visual Studio — it is unavailable on macOS, Linux. The bidirectional navigation also has a notable bug: navigating from a tree node to the editor highlights all occurrences of the same text (for example, every int keyword) rather than the specific node instance. Despite these shortcomings, this tool is very popular among developers, because of the high popularity of the Visual Studio.

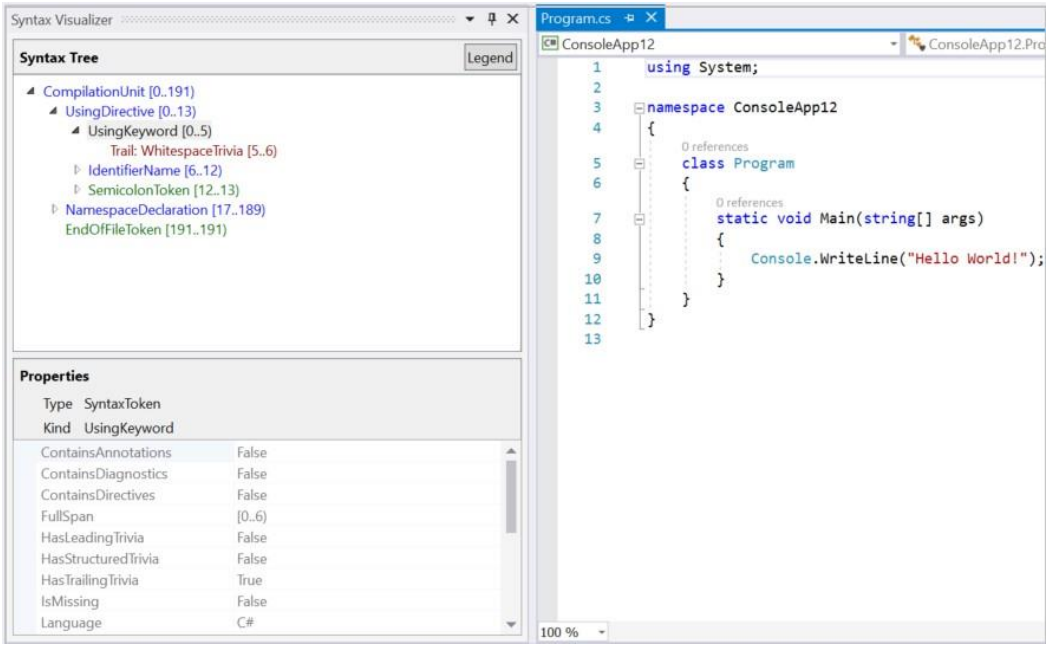

Figure 2.1: The Roslyn Syntax Visualizer in Visual Studio. Adapted from Microsoft documentation [11].

SharpLab.io

SharpLab.io is a web-based .NET playground created by Andrei Shchekin [17] that supports multiple output modes including C# decompilation, IL, JIT assembly, and AST visualisation. In AST mode it displays the Roslyn syntax tree with IOperation data and provides bidirectional cursor synchronisation between the tree and the code editor. It requires no installation, runs in any browser on any operating system, and supports C#, Visual Basic, and F#, with shareable permalinks for every snippet.

The main drawback of SharpLab is the absence of IDE integration: the developer must copy code out of the development environment, losing all project context. The tool provides no information about node properties, fields, or methods, limiting its usefulness for analyzers and source generators development. It also requires an internet connection and sends code to an external server, which raises privacy concerns for private projects. Also, in this case your workflow is dependent on the availability of this service During testing I found that it fails to process large files. SharpLab remains a convenient quick-inspection tool for small snippets, but its limitations make it unsuitable as a primary development solution.

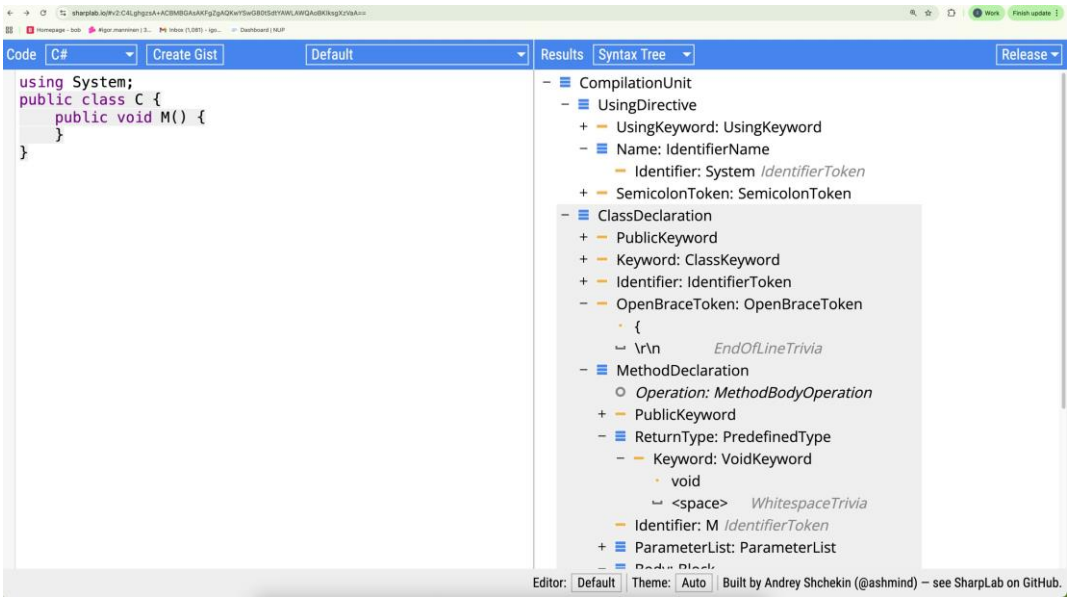

Figure 2.2: SharpLab.io in AST mode. The collapsible tree on the right shows Roslyn SyntaxNode and SyntaxToken types for the code entered on the left. Source: SharpLab repository [17].

Syndiesis

Syndiesis [3] is a cross-platform desktop application built with Avalonia UI and is the most feature-complete standalone tool for Roslyn syntax tree exploration. It offers four analysis layers — syntax tree, semantic model, IOperation tree, and attributes — with an embedded code editor, syntax highlighting, and real-time cursor-to-node synchronisation. It runs on Windows, macOS, and Linux and supports both C# and Visual Basic. Overall performance is good.

Despite its rich feature set, Syndiesis is not integrated into any IDE: the developer must launch a separate application and type or paste code into its own editor, with no connection to an open project. The installation process can also be challenging, since users must compile the binaries themselves and ensure a compatible .NET runtime version is installed and available. During my evaluation I noticed that navigating from a tree node to the corresponding code does not scroll the editor, which makes the tool difficult to use on large files where the target code may be off-screen. Syndiesis - is good tool, but due to it being a small pet project on GitHub, and nontrivial installation process, it makes Syndiesis a very popular solution on the market.

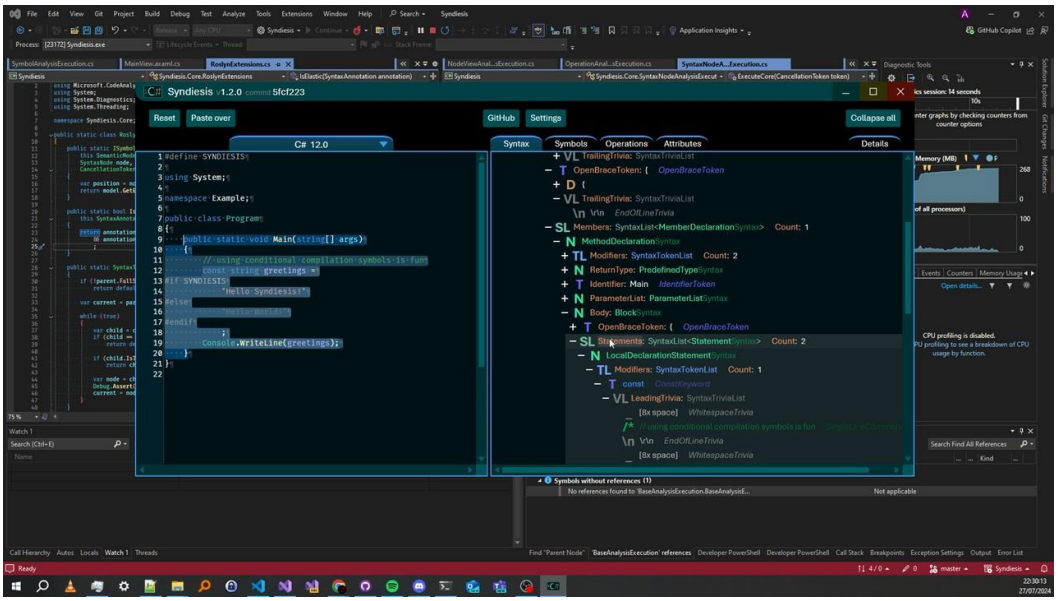

Figure 2.3: Syndiesis showing the syntax tree alongside a code editor. The tab bar provides quick switching between Syntax, Semantic, Operations, and Attributes views. Source: Syndiesis GitHub repository [3].

LINQPad

LINQPad [2] is a widely used .NET scripting tool primarily designed for running C# code snippets interactively. It includes a dedicated syntax tree visualizer that is available in the results panel after executing a code snippet: the developer writes or pastes code for the analysis, runs it, and the results menu offers an interactive, collapsible tree view of the syntax structure. LINQPad supports .NET 6 through 9 and runs on Windows and macOS.

However, LINQPad is not a tree visualisation tool by design — it is a general-purpose scripting environment tool, and syntax tree inspection is a side effect of its post-compilation capabilities. It provides no bidirectional navigation between the tree and the source code, and navigating the dumped tree becomes impractical on large code samples. After the test review, I noticed that the performance on the large code files is very poor, and even on 3000 lines code snipped it already hase visual freezes. Linux is not supported. While useful for quick inspections, LINQPad cannot replace a purpose-built visualizer for prolonged analyzer or source generator development.

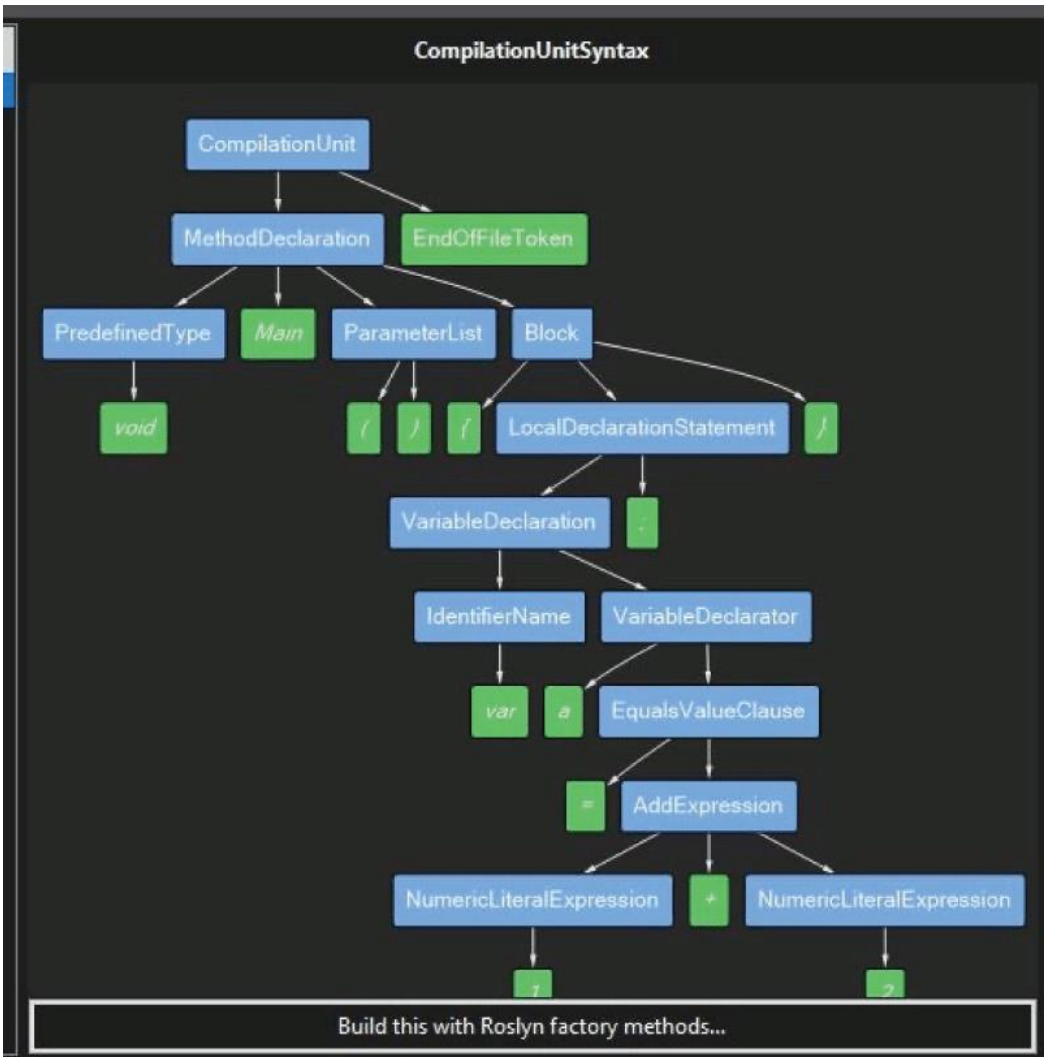

Figure 2.4: LINQPad's .Dump() output panel rendering a structured object tree. The same mechanism is used to explore Roslyn syntax tree nodes interactively. Source: linqpad.net.

Rossynt (JetBrains Rider Plugin)

Rossynt [4] was the only Roslyn syntax tree visualizer plugin for JetBrains Rider, created by GitHub user GitHubPang. It provided a native tool window inside the IDE that showed the syntax tree for the active file, including scratch files, with node property inspection, code highlighting, and bidirectional navigation. Because it ran wherever Rider ran, it was cross-platform by default.

Rossynt is the closest predecessor to the tool developed in this thesis and demonstrates that a Rider-integrated syntax tree viewer is both feasible and valuable. However, the project has been abandoned: its last release dates to February 2023 and it is incompatible with Rider 2023.2 and all later versions. Its architecture required a separately running .NET backend process that communicated with the plugin over HTTP, which introduced noticeable latency and made the performance worse than a fully integrated solution. The abandonment of Rossynt is the direct motivation for this thesis: Rider users currently have no maintained tool for Roslyn syntax tree visualization.

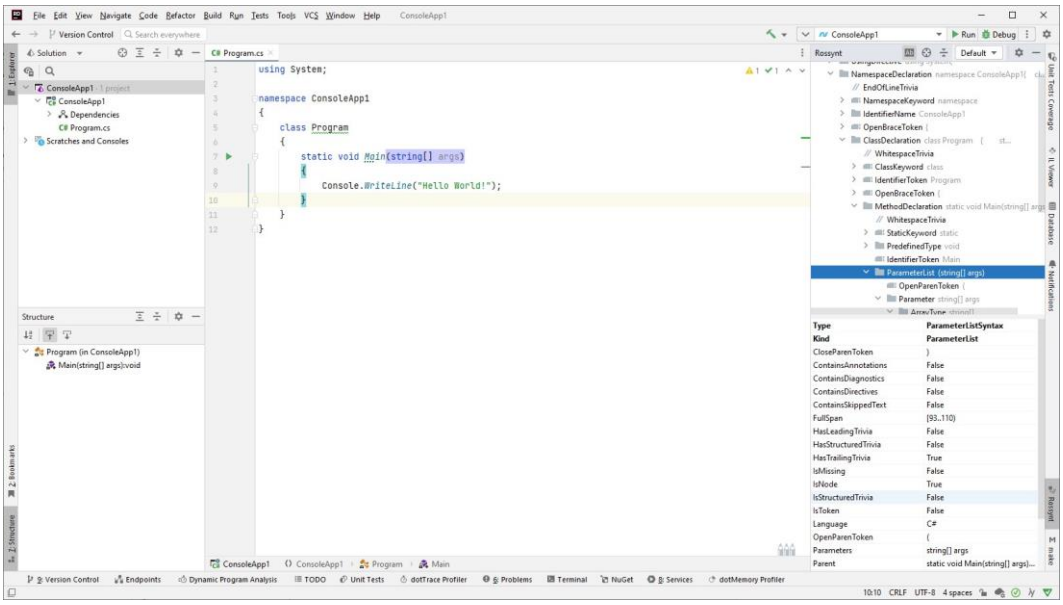

Figure 2.5: Rossynt plugin in JetBrains Rider. The tool window shows the Roslyn syntax tree for the active C# file. Source: Rossynt GitHub repository [4].

Summary

Table 2.1 summarises the key properties of the tools surveyed.

Tool	IDE	Cross-platform	Node data	Navigation	Performance
VS Syntax Visualizer	VS only	No	Yes	Limited	Good
SharpLab.io	None	Yes	No	Yes	Crashes
Syndiesis	None	Yes	Yes	Limited	Good
LINQPad	None	No	Yes	No	Slow
Rossynt	Rider	Yes	Yes	Yes	Ok
This thesis	**Rider**	**Yes**	**Yes**	**Yes**	**Good**

Table 2.1: Comparison of existing Roslyn syntax tree visualisation tools.

The analysis reveals a clear gap: no currently working and maintained tool provides native IDE integration in JetBrains Rider with real-time tree updates, node data inspection, and cross-platform support. Rossynt previously addressed this gap but is no longer functional.

2.2 Feature Analysis

After surveying the existing tools, the next step was to determine which features the new tool should provide and in what order of priority they should be implemented. Two following methods were used: a hands-on development session and a targeted survey of Rider developers with relevant experience.

Hands-on Development Session

To gain understanding of the workflow friction, one full day was spent writing Roslyn analyzers and Source generators from scratch. Throughout the session, every point at which the lack of a good tree visualizer caused a slowdown was noted. For example, having to guess node type, to which I can cast the current node, or switch to an external tool to verify the tree shape. Or scrolling through completion popup in IDE to determine if this node provides required properties or methods or not. This session produced a list of pain points that were used later, to shape and write down user stories.

Developer Survey

A short survey was conducted among JetBrains Rider developers who had practical experience writing source generators or Roslyn analyzers. Participants were asked to describe frequent pain points in their workflow and to rate which visualizer capabilities they considered most valuable. The responses confirmed the pain points identified during the hands-on session and provided additional perspective on relative importance of the visualizer features.

User Stories and Prioritisation

The findings from both sources were consolidated into a set of user stories. Each story follows the standard format: *As a <role>, I want <feature>, so that <benefit>.* Stories were ranked by the combined demand, which was determined after the survey and from personal experience.

1. **As a source generator developer, I want to see the syntax tree alongside the code, so that I can quickly attach the required subtree to the correct node.**

   This story was rated the highest and directly addresses a core feature that every visualizer must have. When navigating an unfamiliar code pattern, the developer needs to see exactly which SyntaxNode subtype covers a given piece of code. This feature is a non-negotiable requirement for the tool.

2. **As an analyzer developer, I want to see all node properties and their values in a list, so that I can easily filter the node by a specific condition.**

   Together with story №1, this addresses the second core task: knowing what properties a node exposes for matching or transformation. Both stories received the highest combined demand score from the survey and the hands-on session and are therefore non-negotiable requirements.

3. **As a source generator developer, I want to search within the properties table, so that I can quickly find a desired property.**

   This addresses discoverability on nodes with many properties. Without a search capability, scrolling through a large list of properties becomes impractical. This feature was rated important but slightly lower than navigation, because it primarily affects productivity on large inputs rather than correct-

ness of the workflow.

4. **As an analyzer developer, I want to click a tree node and jump to the matching code in the editor, so that I can quickly identify the correct code pattern and attach an inspection.**

   This story describes tree-to-editor navigation. Both the hands-on session and the survey confirmed that the absence of this feature in external tools is the single greatest workflow interruption: the developer loses context every time they must manually correlate a tree node with its position in the source file. Bidirectional navigation is therefore treated as a high-priority feature.

5. **As an analyzer developer, I want to place the caret in the editor and auto-expand the tree to that node, so that I can quickly locate the node on which to register an inspection.**

   This is the reverse direction of story 4 — editor-to-tree navigation. Together, stories 4 and 5 form the bidirectional navigation capability that the survey identified as the most impactful missing feature across existing tools.

6. **As a source generator developer, I want to search across all tree nodes, so that I can quickly find the target subtree.**

   Similar to story 3, this addresses discoverability but at the tree level rather than the property level. On large files with hundreds of tree nodes, manual scrolling is impractical. This feature was rated important but slightly lower than navigation.

7. **As an analyzer developer, I want to see the most-used node properties at the top of the table, so that I can quickly determine the node kind without scrolling.**

   The survey showed that developers query Kind on almost every node inspection, so surfacing it immediately avoids repeated scrolling. This feature was classified as desirable but not blocking.

3. Requirements and Design Goals

User stories from the Section 2.2 formed capabilities and features that our tool must provide. This chapter formalises these findings into functional requirements, non-functional requirements, and software design goals, that guide and support the implementation process.

3.1 Functional Requirements

1. The tool should display the Roslyn syntax tree of the currently open C# or VB .NET file in a dedicated Rider tool window, allowing developers to see source code and the syntax tree simultaneously.
2. The displayed tree shall update automatically when the source file is modified.
3. Selecting a tree node should display all fields, properties, and method signatures of the corresponding syntax tree node or token in a dedicated properties panel.
4. The properties panel should provide a search field that filters the displayed entries by name.
5. Double-clicking a tree node should highlight and scroll to the corresponding source code range in the editor.
6. Moving the caret in the editor shall expand and select the matching tree node, available both as a manual one-shot action and as a continuous automatic tracking mode.
7. Declaration nodes (classes, methods, properties, fields, etc.) should be rendered with context-appropriate icons and visibility overlays (public, private, protected, internal).

3.2 Non-Functional Requirements

1. The tool should not consume any frontend or backend resources when it is hidden.
2. Tree nodes should be loaded lazily on demand, so that no resource consumption will be efficient.
3. The tool shall run on all platforms (Windows, macOS, Linux) without platform-specific code parts.

4. User interactions (keystrokes, caret movement, tree clicks) should be debounced to prevent backend overload.
5. All backend requests should be sent through a single mutex on the frontend and executed on an exclusive single-threaded scheduler on the worker, ensuring thread-safe, and cancellable access to shared state.

3.3 Software Design Goals

After defining functional and non-functional requirements, the following design goals were created to drive the architectural decisions described in the subsequent chapters:

1. **Event-driven incremental updates.** The system should transmit only the delta between tree versions rather than full tree snapshots, keeping the IPC payload and resource usage as low as possible.
2. **Visibility-bounded processing.** All diffing, snapshot building, and update generation should be bounded to nodes that the frontend has actually expanded, ensuring that cost scales with what the user inspects rather than with total file size.
3. **Snapshot caching and replay.** The backend should cache versioned snapshots so that when multiple edits arrive before the frontend catches up, precomputed diffs can be replayed without redundant Roslyn API calls.
4. **Zero-allocation fast paths.** Unchanged subtrees should be reused by reference across snapshot versions, ensuring that memory allocations are proportional to the change size rather than to the total tree size.
5. **Self-correcting desynchronisation recovery.** If the frontend and backend fall out of sync, the system should recover using known to backend frontend snapshots, preventing corruption of the UI.

4. System Architecture and Design

Firstly, lets look how the architecture of the Rider is structured and how I can integrate my solution into existing code infrastructure.

4.1 Rider's Architecture

Unlike most IntelliJ-based IDEs where a single JVM process handles everything, Rider consists of several processes. In this work I will not go through a whole Rider process model, instead I will explain processes that were necessary to complete my goal

1. **Frontend process** (Kotlin/JVM) — the IntelliJ Platform. Handles UI rendering, the editor, and all user interaction.
2. **Backend process** (C#/.NET) — the ReSharper engine. Handles code analysis, refactoring, code completion, and all .NET-specific intelligence, provided by Rider.
3. **Roslyn Worker** (C#/.NET) — a separate C# process that hosts the Roslyn compiler platform and maintains a Roslyn workspace.

The visualizer spans all three processes, as illustrated in Figure 4.1.

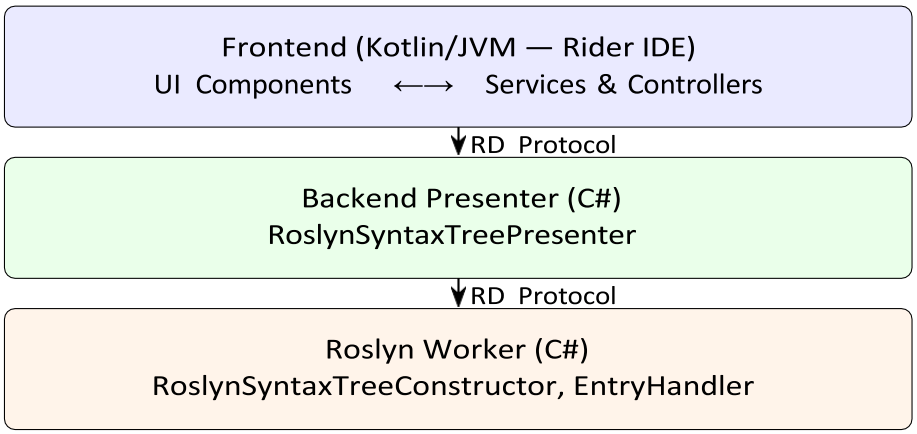

Figure 4.1: Three-layer architecture of the Roslyn Syntax Tree Visualizer.

Frontend

The frontend runs inside the Rider IDE JVM process and is responsible for all user-facing functionality. It owns all UI and rendering — the editor, the Solution

Explorer, tool windows, dialogs, popups, context menus, toolbars.

The frontend has zero knowledge of Roslyn APIs — it receives pre-serialised nodes and their data from the backend and renders them as a tree. The frontend is responsible for managing different states of the tree visualizer that I want to show to the user. (Displaying loading icons, hint messages when the roslyn worker process is not up yet, etc.)

Backend

The backend is the pure computational engine — it understands .NET code deeply (semantics, types, references, errors) and performs all heavy operations like code analysis, refactoring, building, debugging. It communicates results exclusively through RD Protocol. This separation lets Rider use ReSharper's engine while running inside the IntelliJ Platform shell.

For my project I don't need any of the backend features, but since a Roslyn Worker process is not connected to frontend directly, I have to use this backend as a bridge to the frontend. So, I'll have one backend component and its sole responsibility is receiving RPC calls from the frontend session, delegating them to the Roslyn Worker, and converting between frontend and backend RD protocol types.

Roslyn Worker

The Roslyn Worker is the compiler infrastructure layer — it holds the Roslyn workspace, parses code into syntax trees, runs analyzers and source generators, handles compilation, and supports hot reload. By running in a separate process, it isolates heavy compiler work from both the UI (frontend) and the analysis engine (backend), making the overall IDE more responsive and crash-resilient. The worker contains the whole Roslyn syntax tree retrieval and updating logic. It sync frontend state, and contracting tree updates, according to this state.

Also, a worker monitors open documents: whenever a tracked document receives a text change, the handler fires a signal that propagates through the backend presenter to the frontend.

Communication Protocol

All inter-process communication uses the **RD (Reactive Distributed) Protocol** [6], JetBrains' open-source IPC framework. RD uses compact binary serialisation, is type-safe, bidirectional (both sides can initiate calls and signals), and lifetime-aware (all subscriptions and resources are tied to Lifetime objects and automatically cleaned up). Protocol models are defined in Kotlin DSL and generated for both Kotlin and C# parts. The protocol model acts as a formal interface contract between layers — changes to either side must be reflected in the model, which is checked at compile time for both Kotlin and C#.

I use protocol to send updating events from the roslyn worker to the frontend, get node data for properties panel and notify frontend if some document had changed, so maybe we want to update the tree.

4.2 Roslyn Visualizer architecture

After investigating Rider architecture, I proposed session-based lifecycle of the visualizer feature. Instead of synchronizing state between backend and frontend, risking losing some data, I decided to make architecture as clear and simple as possible by using the snapshot approach.

At the start, when a tool window is opened, the frontend initiates handshake with the backend to create a session. All RPC calls and signal subscriptions are scoped to this session. When the tool window is hidden, the session lifetime terminates, which automatically disposes all RD subscriptions on both sides and cleans up backend resources.

The frontend has zero knowledge of how the backend gets, tracks and calculates updates. It just send the request for the update with the current known tree version. The backend, on the other hand, receives an update request, then gets a new tree, and then using known to backend frontend tree snapshots, construct the update to the tree.

This design ensures that no backend resources are held while the tool window is closed. If the Roslyn backend process restarts, the session lifetime terminates and a new session is established automatically.

5. Implementation Details

This chapter describes the key components, design decisions, and algorithms of the visualizer implementation. GitHub repository: https://github.com/JetBrains/rider-roslyn-visualizer

5.1 Frontend Implementation

The frontend is built in Kotlin and runs inside the Rider IDE process. It consists of services that manage the lifecycle, controllers that coordinate state, UI components that render the tree and properties, listeners that connect IDE events to the controllers, and actions that expose functionality through the IDE menus and toolbar.

5.1.1 Lifecycle and state management.

The RoslynVisualizerActivator is a project-level service that monitors tool window visibility using nested Kotlin coroutine lifetimes. When the tool window becomes visible, it waits for both the Roslyn process and the project model to become ready, then initiates the session handshake. Once a session is established, it creates the RoslynStateController — the central coordinator that owns four collaborators:

- RoslynUpdateController (a mutex-guarded wrapper around the RPC calls for events linearization)
- RoslynTreeMutator (applies update events to the Swing tree model)
- RoslynNavigationController (handles caret-based tree navigation and double-click text highlighting)

Event debouncing.

User events: keystrokes, caret movement, tree clicks can fire at very high rates. All event streams use MutableSharedFlow with a 200 ms debounce and collectLatest semantics: the debounce waits for the user to pause sending events, collectLatest cancels requests when a newer event arrives, and only the latest event is kept. Using this approach, we will keep the load to the protocol manageable. The state controller maintains four such flows: filePathFlow for file change and switch file events, nodeDataFlow for single-click property requests, expandTreeFlow for tree expansion events, and caretPositionFlow for caret tracking. A Mutex serialises all outgoing protocol calls to prevent race conditions where overlapping responses

could corrupt the tree state.

Tree rendering.

The syntax tree is rendered as a Swing JTree (RoslynSyntaxTree) with single selection, built-in speed search, and a custom ColoredTreeCellRenderer. Each tree node stores a RoslynSyntaxTreeEntry containing the node type, kind, span, and optional metadata. The renderer displays the node kind as primary text and the token text or identifier as secondary text in grey for faster navigation across the tree. Declaration nodes receive layered icons: a main type icon with visibility and modifier overlays (public/private/static/abstract), matching the JetBrains ReSharper icon convention. This allows user to identify method/property visibility without looking into the node properties panel. Almost all node types have their own icons.

Tool Window

The tool window is registered in the plugin descriptor (intellij.rider.xml) with the factory class RoslynVisualizerWindowFactory, anchored to the right side of the IDE. The doNotActivateOnStart flag ensures that the tool window does not open automatically when the IDE launches, avoiding unnecessary backend work for users who do not need it.

Select In

The RoslynVisualizerSelectInTarget class implements the SelectInTarget interface, which allows users to right-click any element in the editor and choose *Select In → Syntax Tree Visualizer*. This opens the tool window (if not already open) and expands the tree to the selected code element.

Actions

Three actions are registered:

- **Open Roslyn Visualizer** — opens and activates the tool window. Available from the Tools menu.
- **Select node under caret** — a one-shot action that expands the tree to the node matching the current caret position.
- **Always select node under caret** — a toggle action that enables continuous caret tracking. When enabled, the tree automatically follows the cursor as the user navigates the source code. The toggle state is persisted across IDE restarts.

5.2 Backend Implementation

The RoslynSyntaxTreePresenter is registered as a [SolutionComponent] in the backend host. This is needed for integration to the Rider backend architecture. It connects frontend session with the worker session and wires each frontend RPC call to its worker counterpart, converting between Frontend and Backend RD data

types [6] in both directions. The presenter is stateless — all tree state lives in the worker.

5.3 Roslyn Worker Implementation

Entry Handler

The RoslynSyntaxTreeEntryHandler runs inside the Roslyn Worker process and implements IWorkspaceService to automatically be integrated inside Roslyn workspace. For each session, it creates a new RoslynSyntaxTreeConstructor and routes the three RPC methods to it.

Syntax Tree Constructor

The RoslynSyntaxTreeConstructor is the stateful orchestrator of the worker. For each tracked file it maintains a TreeSnapshotStore - an ordered dictionary mapping version numbers to immutable TreeSnapshot objects - and a monotonically incrementing version counter. The frontend sends its last known version with every request, and after applying an update it fires the versionApplied signal, which lets the backend discard snapshots older than the acknowledged version. The constructor provides three operations:

- **Incremental tree update.** The RoslynSyntaxService resolves the Roslyn Document from the current workspace via GetSyntaxRootAsync(), or falls back to parsing from disk via CSharpSyntaxTree.ParseText() for scratch files. When a previous root exists, it uses WithChangedText() for an incremental reparse that reuses unchanged subtrees. For a full update (after switching files), the constructor builds a fresh snapshot with SnapshotBuilder and diffs it against an empty baseline, producing add events for every visible node. For an incremental update, it first checks whether a cached snapshot for the next version already exists in the store; if so, the pre-computed diff is replayed immediately. Otherwise it builds a new snapshot, stores it under the next version, and diffs it against the old snapshot.
- **Lazy children loading.** Invokes SnapshotBuilder on the target subtree to expand it, producing add events for the newly visible children and grandchildren and constructing new snapshot. This **two-level prefetch** means expanding a node loads both its children and grandchildren, so the frontend can display expansion arrows on the children correctly.
- **Node data retrieval.** Validates the version, locates the target node in the snapshot, and delegates to SyntaxNodePropertiesFinder.GetNodeData(), which uses System.Reflection to extract public fields, properties, and method signatures from the actual Roslyn object. This approach is deliberately dynamic — it works for any node type and automatically supports new types added in future Roslyn versions.

Snapshot Store and Visibility State

The backend maintains a **snapshot store** per file — a versioned history of immutable TreeSnapshot objects, each pairing a Roslyn SyntaxNode root with a parallel VisibilityNodeState tree. This is needed to track what is sent to the frontend, to construct correct update events.

Each VisibilityNodeState node tracks four pieces of metadata:

- IsOnFrontend — whether the node has been sent to the frontend, controlling lazy loading.
- IsShown — whether incremental updates should emit events at this depth.
- LastKnownSpan — the node's last known source location, used to suppress redundant update events.
- Children — list of children states.

Nodes are identified on the wire by pathToNode — a list of child indices from the root.

The SnapshotBuilder constructs a VisibilityNodeState tree from a Roslyn syntax tree. When an old snapshot exists, it reuses visibility state for nodes whose kind and token/node classification match, preserving the frontend's expansion state across incremental updates.

Declaration Metadata Extraction

For declaration nodes (classes, methods, properties, etc.), the DataForIconsVisitor extracts the identifier name, walks the modifier token list to set visibility flags (public, private, protected, internal) and modifier flags (abstract, static, virtual, readonly), and for properties checks the accessor list for getter/setter presence. Non-declaration nodes return no additional metadata. The frontend uses this data to select the base icon and overlay the appropriate modifier badges.

5.4 Tree Update Algorithm

This section describes the complete tree updating pipeline: how a code change in the editor becomes a set of minimal UI updates on the frontend tree. The algorithm utilizes all three processes and combines Roslyn's built-in incremental parsing with a custom snapshot-based diffing layer.

The RoslynSyntaxTreeConstructor.StartIncrementalTreeUpdate() method has three execution paths depending on the request type and cache state:

1. **Full update (new file or files switch).** Obtains a fresh Roslyn root (full parse, no old root to reuse). The SnapshotBuilder creates an initial VisibilityNodeState with the root expanded and first-level children visible. The DiffBuilder diffs against an empty baseline, producing add events for every visible node. The resulting snapshot is stored under the new version number.
2. **Cached replay.** Both the old snapshot (at the frontend's known version) and

a next snapshot already exist in the store. The Roslyn API is skipped entirely — the two cached snapshots are diffed directly. This path is hit when the worker has already pre-computed a snapshot in response to a fileHasChanged event but the frontend has cancelled the update call and didn't apply changes.

3. **Incremental recompute.** The old snapshot exists at the frontend's known version. A new Roslyn root is obtained via WithChangedText() (incremental reparse). The SnapshotBuilder rebuilds the visibility state, reusing old structure where possible. The DiffBuilder generates minimal events, and the new snapshot is stored.

SnapshotBuilder — Rebuilding the Visibility State

The SnapshotBuilder walks the new Roslyn tree and the old VisibilityNodeState simultaneously using a two-pointer technique:

1. If the old node is collapsed, the builder updates only the span if it changed and keeps the old children untouched. This is a key optimisation: collapsed subtrees are never walked, regardless of their size.
2. For expanded nodes, the builder advances an old-index and a new-index pointer through the children:
   - If the kind and token/node classification match at both indices, the builder recurses into the child and advances both pointers.
   - If they do not match, a lookahead heuristic scans the remaining new Roslyn children for a node matching the old child's kind. If found, the current new child is treated as an insertion; otherwise the old child is treated as a deletion.
3. When a new node is inserted among expanded siblings, the builder creates it with two-level expansion depth to match the surrounding context. This prevents a collapsed stub from appearing among expanded siblings.
4. If all children were reused by reference and the child count is unchanged, the builder returns the identical old state object. Combined with immutable records and ImmutableArray, unchanged subtrees share memory with the previous snapshot — allocations are proportional to the change size, not the file size.

DiffBuilder — Generating Minimal Events

The DiffBuilder compares old and new VisibilityNodeState trees and produces a list of add, remove, and update events. It operates at two levels:

Node-level comparison. If both old and new nodes are on the frontend and their kind or token classification differs, the builder emits a remove event for the old node followed by add events for the new node (replace semantics). If only the span or text changed, it emits a single update event. If only the old node was on the frontend, it emits a remove; if only the new node, it emits adds for the node and all its visible descendants.

Children-level comparison. The builder uses the same two-pointer technique

with lookahead as the SnapshotBuilder, but generates events instead of building state. A pathOffset counter accumulates +1 for each insertion and −1 for each deletion, so that remove events reference the correct frontend path index (old index plus path offset) despite position shifts.

Event ordering. Remove events are emitted in reverse child order before removing the parent. This is necessary because the frontend has a mutable state, and we have to make all modifications to the tree at the right order. Add events are emitted parent-first, then children in forward order, because the parent must exist before children can be inserted.

Version Acknowledgement and Memory Management

The version acknowledgement mechanism bounds memory usage regardless of how many edits occur. When the frontend fires versionApplied, the worker drops all snapshots with version numbers lower than the acknowledged version. Additionally, when a file is acknowledged, the worker drops all snapshots for other files, since only one file is visualised at a time. This ensures that at most two to three snapshots per file exist simultaneously — the current snapshot and any in-flight updates.

Advantages of the Approach

The event-driven update pipeline provides several performance and correctness properties:

- **Two-layer incrementality.** Roslyn's WithChangedText() reuses unchanged syntax subtrees at the parsing level.
- **Lazy expansion bounds complexity.** By only tracking and diffing visible nodes, the system scales to arbitrarily large files. A 10 000-line file with only the root expanded has approximately 20-50 visible nodes. The diff algorithm walks only those nodes, not the full AST.
- **Zero-allocation fast paths.** The SnapshotBuilder returns the old state object by reference when nothing changed. For a typical edit that modifies one method, over 95% of the visibility tree is reused by reference.
- **Correct ordering guarantees.** Remove events are emitted in reverse child order and add events in forward order, matching the Swing tree model's index semantics. The frontend mutex and the worker's exclusive scheduler prevent any concurrent modification.

5.5 Testing

The implementation includes an integration test suite based on the Rider test framework. Tests use .NET 8 SDK and a dedicated test solution to verify end-to-end behaviour. Each test follows a setup–action–teardown pattern: it waits for the Roslyn backend to become ready, creates the tool window, establishes a protocol session, performs the test action, and verifies that the session cleans up correctly on deactivation.

Twelve test cases cover the core functionality:

- Incremental tree updates after code edits.
- Tree node expansion and lazy child loading.
- Double-click-to-select: verifying that the editor selection matches the node's source span.
- Expand-to-caret: setting the caret and verifying that the tree expands to the correct node.
- Toggle-to-caret: enabling continuous tracking and moving the caret to multiple positions.
- Select In integration: invoking the "Select In" action and verifying tree expansion.

Tree state is validated using gold file comparison: the tree is serialised to a text format and compared against a expected reference file.

5.6 Correctness

The data management design guarantees that the frontend tree always moves to a correct representation of the source code, regardless of timing or ordering of events. Several mechanisms work together to achieve this.

First, all protocol calls from the frontend are serialised through a single Mutex, and all worker operations run on a ConcurrentExclusiveSchedulerPair.ExclusiveScheduler exclusive scheduler with limited to 1 concurrency level. This means that at any point in time, at most one update is being processed on each side. There is no window in which two concurrent modifications could interleave and leave the tree in an inconsistent state.

Second, the versioning scheme prevents stale updates from being applied. Every tree snapshot receives a monotonically increasing version number. The frontend sends its last known version with every request, and the worker uses that version to locate the correct baseline snapshot for diffing. If the baseline no longer exists — for example, because multiple edits arrived before the frontend could catch up — the worker falls back to a full rebuild rather than attempting to diff against wrong data. This forced-full path is self-correcting: any accumulated drift is discarded and the tree is rebuilt from scratch.

Third, the versionApplied signal closes the feedback loop. After the frontend applies an update, it acknowledges the new version, and the worker discards all older snapshots.

Fourth, event debouncing with collectLatest semantics ensures that only the most recent user action is processed. If the user types several characters in quick succession, intermediate requests are cancelled and only the final state triggers an update. This eliminates a class of bugs where outdated responses arrive after newer ones and overwrite correct state.

Finally, the session lifecycle provides a hard reset boundary. When the tool window is closed or the Roslyn process restarts, the session lifetime terminates, all subscrip-

tions are disposed, and all backend state is cleaned up. Reopening the tool window starts a fresh session with a new handshake, guaranteeing a clean initial state with no residual data from a previous session.

Together, these mechanisms — single-threaded execution on both sides, versioned snapshots with fallback rebuilds, acknowledgement-based cleanup, debounced event processing, and session-scoped lifetimes — ensure that the visualizer cannot enter an incorrect state under normal operation.

5.7 Challenges and Solutions

Several non-trivial engineering challenges arose during implementation.

Cross-Process State Synchronisation

The multi-process architecture means that the tree state is authoritative in the Roslyn Worker, but the UI lives in the frontend JVM process. Any desynchronisation — caused by dropped messages, process restarts, or version mismatches — could leave the UI in an inconsistent state. The solution is the version-tracked snapshot store with a forced-full fallback path: if the frontend's known version does not match any stored snapshot, the system automatically performs a complete rebuild rather than attempting a partial recovery.

Incremental Tree Update Algorithm

Designing a correct incremental update algorithm was the most challenging part of the project. The core difficulty is that Roslyn syntax nodes have no stable identity across re-parses — every call to WithChangedText() may produce an entirely new object graph, even for nodes whose source text did not change. This means the algorithm cannot simply compare object references; it must match nodes by their structural position, kind, and classification.

The first working version used a naive approach: rebuild the entire visibility state on every edit and diff it fully against the previous snapshot. This was correct but too slow for large files, because it walked the entire tree on every keystroke regardless of how small the edit was.

The next iteration introduced the two-pointer matching with lookahead heuristic. Getting the lookahead logic right required careful handling of edge cases: a single inserted node shifts all subsequent sibling indices, and the pathToNode addressing scheme means that every emitted event must account for these shifts via the pathOffset counter. An error in the offset would cause the frontend to insert or remove the wrong node, corrupting the tree silently.

The visibility state reuse added further complexity. When the SnapshotBuilder encounters a collapsed subtree, it must skip it entirely and carry over the old state. But when a node transitions from collapsed to expanded, the builder must construct fresh state for the newly visible children while preserving the parent's existing state.

The final design went through several iterations of writing, testing against edge cases, and rewriting before reaching a stable form.

Different lifetimes

In Rider many components have their own lifetime. For example user IO listeners have frontend lifetime, backend components have their own lifetime, roslyn componetns thier own. Even in one process different components have different lifetimes. So, every time i want to perform a call to backend, I have to check multiple conditions to verify that every component is in the correct state. This approach leads to synchronisation issues, race conditions and other hard to reproduce bugs. So, I come up with a session solution. I will bound every component lifetime to the session lifetime. And I will use a nested coroutine scope structure to initiate every layer of lifetimes one by one, and if parent scope will be cahcelled — it will cancel every child scope. There is another benefit of nested coroutine scope structure. It allows removing a lot of checked conditions in before calling to the other processes, since if the current scope is not cancelled, that means that corresponding conditions are satisfied.

6. Results Discussion

6.1 Ensuring Performance and User Experience

The visualizer is designed to remain responsive even on large files, using several techniques that work together to minimise latency and keep the UI smooth.

Performance

The most important performance mechanism is lazy loading. The tree is never fully expanded on first display — only the root and its immediate children are sent to the frontend. Deeper nodes are loaded on demand when the user expands a parent. This means that regardless of file size, the initial load transfers only a small number of nodes.

Incremental updates further reduce the cost of edits. Roslyn's WithChangedText() reuses unchanged subtrees at the parsing level, so the worker never re-parses the entire file on a keystroke. On top of that, the visibility-state diff only emits events for nodes that are both visible on the frontend and actually changed.

Snapshot caching avoids redundant computation. When multiple edits arrive in quick succession, the worker may pre-compute a snapshot before the frontend requests it. The cached replay path detects this and skips Roslyn API calls entirely.

The SnapshotBuilder returns old state objects by reference when nothing changed, so unchanged subtrees share memory with the previous snapshot.

User Experience

Event debouncing with 200 ms delay and collectLatest semantics ensures that rapid user actions (typing, caret movement) do not flood the backend with requests. Only the final state after the user pauses is processed, keeping the UI responsive.

The tree preserves the user's expansion state across edits. When the source code changes, nodes that were expanded before the edit remain expanded after the incremental update, so the user does not lose context.

Two-level prefetch during node expansion loads both children and grandchildren, so the frontend can display expansion arrows immediately without an additional round trip. This eliminates the visual twitching of nodes appearing without expand indicators.

Bidirectional navigation keeps the tree and editor synchronised. Double-clicking a

tree node selects the corresponding source span in the editor. The caret-tracking mode does the reverse — moving the cursor in the editor automatically expands and highlights the matching tree node. This two-way link allows the user to work from whichever view is more convenient.

Declaration nodes display layered icons with visibility and modifier overlays matching the ReSharper icon convention, allowing the user to identify method visibility at a glance without opening the properties panel.

The session lifecycle ensures a clean experience on edge cases. If the Roslyn process restarts or the project reloads, the session terminates and a new one is established automatically, with no stale state left behind.

6.2 Performance Analysis

To evaluate the visualizer's performance relative to existing tools, initial tree loading time and memory consumption were measured across five tools: the Rider visualizer, Syndiesis, Visual Studio's Syntax Visualizer, Rossynt, and LINQPad. Each tool was tested on C# files of increasing size: 100, 500, 1 000, 3 000, 6 000, and 68 000 lines. All measurements were taken using dotTrace profiler [5], measuring the time from requesting the tree to the UI being fully rendered.

Initial Load Time

Figure 6.1 compares the total initial load time across all five tools.

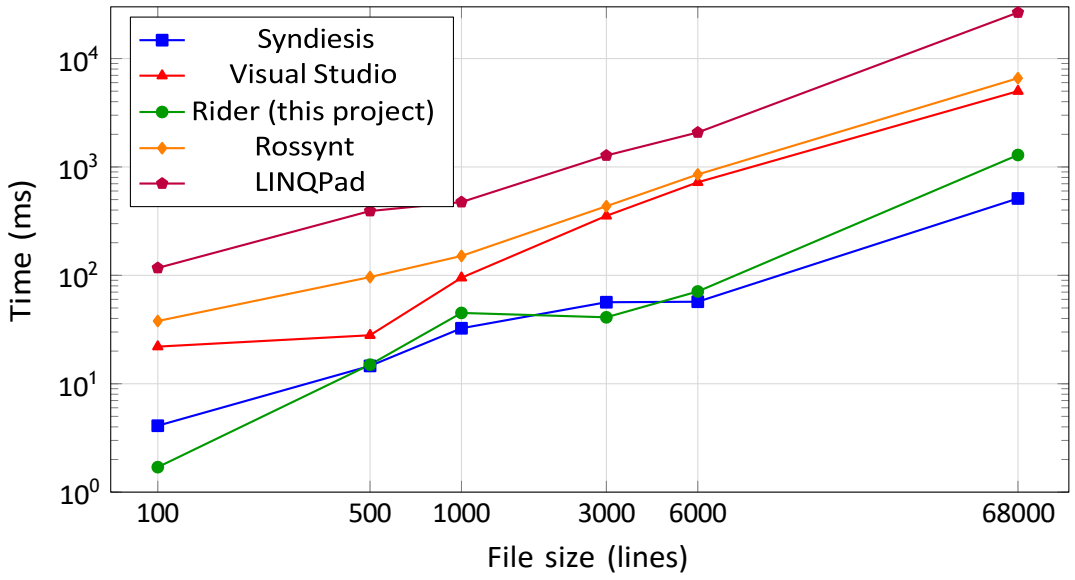

Figure 6.1: Initial tree load time comparison (log–log scale).

Syndiesis is the fastest tool across all file sizes, loading the 68 000-line file in 512 ms. The Rider visualizer is the second fastest, reaching 1 289 ms on the same file. Visual Studio and Rossynt are comparable at around 5–6.5 s, while LINQPad is the slowest at 26.6 s.

Memory Consumption

Figure 6.2 compares memory consumption for the initial tree load.

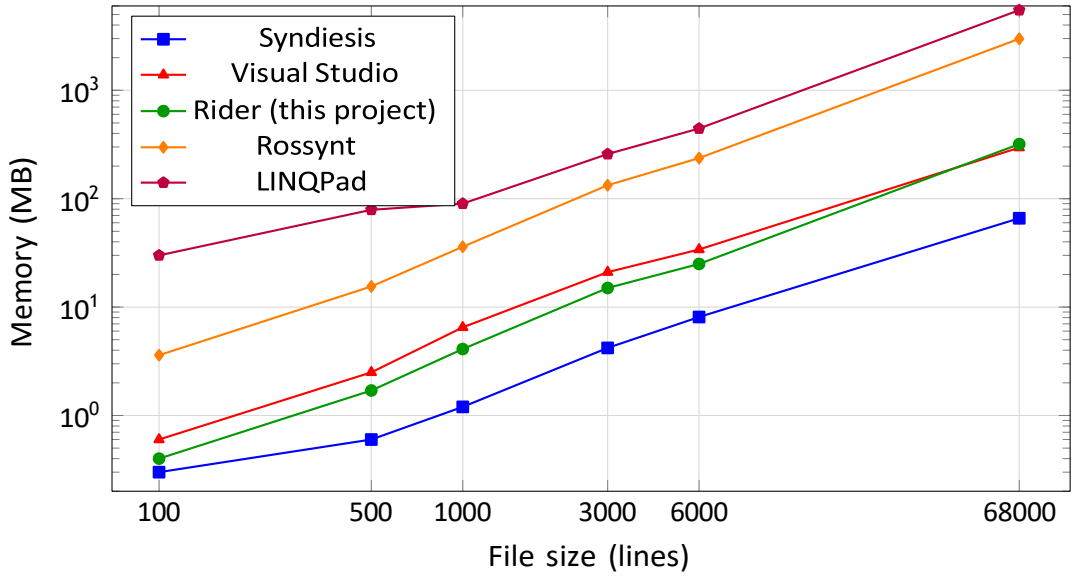

Figure 6.2: Memory consumption comparison (log–log scale).

Syndiesis uses the least memory across all file sizes, consuming only 66 MB for the 68 000-line file. The Rider visualizer and Visual Studio are comparable, using 319 MB and 296 MB respectively for the largest file. Rossynt and LINQPad consume significantly more memory — 2 985 MB and 5 485 MB — making them impractical for very large files.

The Rider visualizer's memory usage is higher than Syndiesis because Syndiesis renders a Roslyn synatx tree directly after receiving it from the compiler. But this approach is not applicable in my case, since I can display the UI only inside the frontend process and the syntax tree is obtained only on the backend process. Also, because of the multi-process architecture, I need to syncronize state across these processes. For that I use snapshots, which consume memory. This is a deliberate trade-off: the snapshot overhead is bounded (due to version acknowledgement call).

Summary

Table 6.1 summarises the results for the 68 000-line stress test.

Tool	Load time (ms)	Memory (MB)
Syndiesis	512	66
Rider (this project)	1 289	319
Visual Studio	5 002	296
Rossynt	6 592	2 985
LINQPad	26 595	5 485

Table 6.1: Performance comparison on a 68 000-line C# file.

The Rider visualizer is the second fastest tool in initial load time and comparable to Visual Studio in memory usage. Notably, the Rider visualizer outperforms Visual Studio's Syntax Visualizer — loading the 68 000-line file nearly four times faster (1.3 s vs. 5 s) while using a similar amount of memory. This is a particularly important result because Visual Studio is Rider's main competitor in the .NET development space, and its built-in Syntax Visualizer is the tool most Roslyn developers are familiar with. Delivering better performance than the established baseline validates the architectural choices made in this project.

7. Conclusion

This thesis set out to design and implement a Roslyn Syntax Tree Visualizer for JetBrains Rider — filling a gap that forced .NET developers to leave their primary IDE whenever they needed to inspect the syntax tree while building analysers or source generators.

The resulting tool is a fully integrated Rider toolwindow that utilizes three processes and satisfies all functional and non-functional requirements defined in Chapter 3. It displays a live, interactive syntax tree in a dedicated tool window, updates incrementally as the source code changes, bidirectional navigation between the tree and the editor, node property inspection via reflection, and declaration icons with visibility overlays.

The architecture is built around a session-based lifecycle with versioned immutable snapshots. This design allows cancellation of RPC calls and provides self-correcting recovery when processes fall out of sync. The correctness of the data management layer is guaranteed by single-threaded execution on both sides, monotonic version tracking with acknowledgement-based cleanup, event debouncing, and session-scoped lifetimes that prevent stale state.

Performance measurements show that the visualizer is the second fastest tool among five competitors in initial load time, handling a 68 000-line file in 1.3 s, while using memory comparable to Visual Studio's built-in visualizer.

Several engineering challenges were addressed during development. The incremental tree update algorithm required a two-pointer matching strategy with lookahead heuristics and careful event ordering. Cross-process state synchronisation was solved through the snapshot versioning scheme with a forced-full fallback path.

The tool is currently available as part of Rider's internal plugin infrastructure and has been released with Rider 2025.1 [7].

Future Work

The following feature could be implemented in the future:

- **Semantic model integration.** The current tool displays only the syntax tree. Adding access to Roslyn's SemanticModel would allow showing type information, symbol resolution, and data-flow analysis results directly in the properties panel.

GitHub repository: https://github.com/JetBrains/rider-roslyn-visualizer

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2nd edition, 2006.

[2] Joseph Albahari. Linqpad — the .net programmer's playground. https://www.linqpad.net, 2025. Accessed: April 2026.

[3] George Avgoustis. Syndiesis — cross-platform roslyn syntax tree visualizer. https://github.com/Rekkonnect/Syndiesis, 2025. Accessed: April 2026.

[4] GitHubPang. Rossynt — roslyn syntax tree viewer for jetbrains rider. https://github.com/GitHubPang/Rossynt, 2023. Accessed: April 2026.

[5] JetBrains. dottrace — .net performance profiler. https://www.jetbrains.com/profiler/, 2025. Accessed: May 2026.

[6] JetBrains. Rd — reactive distributed communication framework for .net, kotlin, c++. https://github.com/JetBrains/rd, 2025. Accessed: May 2026.

[7] JetBrains. Rider 2025.1 eap 5: Roslyn syntax tree visualizer, performance improvements, and more. https://blog.jetbrains.com/dotnet/2025/02/24/rider-2025-1-eap-5/, 2025. Accessed: April 2026.

[8] Microsoft Corporation. Roslyn — the .net compiler platform. https://github.com/dotnet/roslyn, 2014. Accessed: May 2026.

[9] Microsoft Corporation. Incremental generators. https://github.com/dotnet/roslyn/blob/main/docs/features/incremental-generators.md, 2023. Accessed: April 2026.

[10] Microsoft Corporation. Roslyn overview. https://github.com/dotnet/roslyn/blob/main/docs/wiki/Roslyn-Overview.md, 2023. Accessed: April 2026.

[11] Microsoft Corporation. Explore code with the roslyn syntax visualizer in visual studio. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/syntax-visualizer, 2024. Accessed: April 2026.

[12] Microsoft Corporation. How to use source generation in system.text.json. https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/source-generation, 2024. Accessed: May 2026.

[13] Microsoft Corporation. Mvvm toolkit — .net community toolkit. https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/, 2024. Accessed: May 2026.

[14] Microsoft Corporation. The .net compiler platform sdk (roslyn apis). https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/, 2024. Accessed: April 2026.

[15] Microsoft Corporation. .net regular expression source generators. https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-source-generators, 2024. Accessed: May 2026.

[16] Microsoft Corporation. Tutorial: Write your first analyzer and code fix. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix, 2024. Accessed: April 2026.

[17] Andrei Shchekin. Sharplab — .net code playground. https://sharplab.io, 2024. Accessed: April 2026.