
20483 C Sharp

Contents

 What people <i>still</i> build in C# today	2
 VS vs VS Code — What You Really Get.....	4
 Sample .NET Interactive Notebook (C#).....	5
A Real API-Connected Dashboard.....	7
Mini-dashboard that pulls multiple APIs	10
 Crypto Dashboard — Bitcoin vs Ethereum	11
Real historical API version.....	14
Reference	14

You said:

sure a sample is good - does using vs studio vs vs code really offer more or less options to dev folks other than downloading and installing workloads vs extensions?

ChatGPT said:

Excellent question again — that distinction between **Visual Studio (VS)** and **VS Code** is one of the most practical things to understand as a modern C# developer. Let's break it down clearly, then I'll show your **sample .NET Interactive notebook**.

What people *still* build in C# today

Even though there are templates for nearly everything, C# is alive and thriving, especially because of **.NET 8 and Azure integration**. Common types of C# apps built today include:

1. Cloud & API Backends

- **ASP.NET Core Web APIs** — power everything from mobile apps to microservices.
- **Azure Functions (serverless)** — for event-driven logic (e.g., IoT device data, form submissions, scheduled jobs).
- **gRPC services** — for high-performance service-to-service communication.

 Modern pattern: CRUD over a REST or GraphQL API backed by EF Core + Azure SQL/CosmosDB. That's the new "hello world" in enterprise C#.

2. Cross-platform desktop or mobile

- **.NET MAUI (Multi-platform App UI)** — lets you build Windows, macOS, Android, and iOS apps from a single codebase.

Think: "modern Xamarin" with XAML and C# front and back.

3. Enterprise & internal tools

- Line-of-business dashboards, HR tools, data-entry apps, and reporting systems.
- Often deployed via **Blazor Server** or **Blazor WebAssembly** (C# running in the browser via WebAssembly).

4. Game dev

- **Unity** uses C# for nearly everything.

Common for simulations, training, or gamified dashboards, not just entertainment.

5. Automation & scripting

- C# console apps, background services, or Windows Services to:
 - Sync data between systems
 - Process files or generate reports
 - Automate DevOps steps with GitHub Actions or Azure Pipelines

"Is it down to CRUD ops in the cloud?"

In a way — yes, **CRUD APIs** are the "entry point" for most C# apps today, but:

- They're often wrapped with **domain logic, AI integration, or IoT data ingestion**.
 - "CRUD" becomes "CLOUD CRUD": RESTful APIs with built-in telemetry, authentication (Entra ID), and scaling via Azure App Service or Kubernetes.
-

Does C# have a Jupyter Notebook equivalent?

Absolutely! It's called **.NET Interactive Notebooks**.

◆ You can:

- Run **C# code cells** (just like Python in Jupyter).
- Mix in **Markdown, charts, and data visualizations**.
- Even run **F#, PowerShell, or SQL** in the same notebook.

◆ How to try:

- Install the [.NET Interactive Notebooks VS Code extension](#)
- Create a file with the extension:
- MyNotebook.dib

or

MyNotebook.ipynb

- Then choose **C# (.NET Interactive)** as the kernel.

Example cell:

```
using System;  
  
Console.WriteLine("Hello from C# Notebook!");
```

```
var nums = new[] {1, 2, 3, 4};  
  
nums.Sum();
```

It runs right inline, and you can graph data, query APIs, or connect to databases — similar to Python's pandas or matplotlib, but C#-style.

VS vs VS Code — What You Really Get

Feature / Focus	Visual Studio (VS)	Visual Studio Code (VS Code)
 Primary purpose	Full IDE for enterprise & large .NET projects	Lightweight editor with extensions for any language
 Workloads vs Extensions	“Workloads” (e.g. ASP.NET, MAUI, Azure, Unity) install big toolchains & templates	“Extensions” (small add-ons) add lightweight support for languages & features
 Project templates	Rich built-in templates (WinForms, WPF, ASP.NET Core, MAUI, Azure Functions, etc.)	Minimal templates — mostly use CLI (dotnet new)
 Debugging	Deep integrated debugger (memory, threads, breakpoints, attach to process, etc.)	Debugger works via extensions but simpler — good for APIs, web, notebooks
 Performance	Heavy but powerful — best for enterprise solutions	Fast, modular — ideal for scripting, cloud, or container work
 Azure integration	1-click publish, remote debugging, Connected Services	Uses Azure CLI + extensions (manual deployment or scripts)
 AI/Testing/Profiling	Built-in test runner, profiler, code analysis, IntelliTest	Requires separate extensions; less visual tooling
 .NET Interactive / Notebooks	 Not supported directly	 Supported (via “.NET Interactive Notebooks” extension)
 Platforms	Windows only (macOS has limited version)	Cross-platform (Windows, macOS, Linux)

TL;DR

-  **Visual Studio** = full enterprise IDE (like flying a 747)
-  **VS Code** = modular, modern, cross-platform dev tool (like piloting a drone)
- Most **modern, cloud, AI, and notebook-based C# work** is trending toward **VS Code** because it's:
 - Cross-platform
 - Works with GitHub Copilot
 - Integrates with .NET CLI, containers, and notebooks better

Sample .NET Interactive Notebook (C#)

1. Save this as SampleDashboard.dib or SampleDashboard.ipynb
2. Open it in VS Code
3. With the **.NET Interactive Notebooks** extension installed.

```
#lcsharp
```

```
// Sample C# Notebook — Mini Data Dashboard
```

```
// Run each cell with Shift+Enter
```

```
using System;  
  
using System.Linq;  
  
using System.Collections.Generic;
```

```
// 1 Generate sample data
```

```
var students = new List<(string Name, int Score)>  
{  
    ("Ava", 85),  
    ("Ben", 90),  
    ("Cara", 78),  
    ("Dylan", 92),  
    ("Ella", 88)  
};
```

```
students
```

```
#lcsharp
```

```
// 2 Calculate simple stats  
  
var avg = students.Average(s => s.Score);  
  
var top = students.OrderByDescending(s => s.Score).First();
```

```
Console.WriteLine($"Average score: {avg}");  
Console.WriteLine($"Top student: {top.Name} ({top.Score})");
```

```
#!lcsharp  
// 3 Visualize with XPlot.Plotly  
#r "nuget:XPlot.Plotly"
```

```
using XPlot.Plotly;  
  
var chart = Chart.Bar(  
    students.Select(s => s.Name),  
    students.Select(s => s.Score)  
)
```

```
chart.WithTitle("Class Score Distribution");  
chart
```

```
#!markdown  
##  Summary  
We just:  
- Created in-memory data in C#  
- Performed LINQ-based calculations  
- Visualized it inline (like a Jupyter notebook)  
- All without a project file or build step
```

Perfect for demos, quick data tasks, or teaching.

A Real API-Connected Dashboard

1. Pull live JSON data from an endpoint and chart it
2. Python-like modern C# notebooks with the Type safety and LINQ power from .NET.

Here's a complete **.NET Interactive notebook** that:

1. Calls a **live JSON API**,
2. Parses it into a typed C# model,
3. Performs some quick analytics, and
4. Visualizes results with a bar chart — all inline.

You can save this as LiveDataDashboard.dib or LiveDataDashboard.ipynb and open it in **VS Code** with the **.NET Interactive Notebooks** extension.

Example — Fetch and Chart Live JSON Data

```
#lcsharp

// Live Data Dashboard: Fetch JSON and chart it

// Run each cell with Shift+Enter


using System;
using System.Net.Http;
using System.Text.Json;
using System.Linq;
using System.Collections.Generic;

// [1] Fetch JSON data from a public API (CoinDesk Bitcoin prices)
var http = new HttpClient();

var json = await http.GetStringAsync("https://api.coindesk.com/v1/bpi/currentprice.json");

Console.WriteLine("Raw JSON:");
Console.WriteLine(json.Substring(0, 150) + "...");
```

```
#!csharp
```

```
// [2] Define simple models to parse the response
```

```
public class CurrencyInfo
{
    public string Code { get; set; }
    public string Symbol { get; set; }
    public string Description { get; set; }
    public decimal RateFloat { get; set; }
}
```

```
public class BpiResponse
{
    public Dictionary<string, CurrencyInfo> Bpi { get; set; }
}
```

```
// Parse JSON to object
```

```
var options = new JsonSerializerOptions { PropertyNameCaseInsensitive = true };
var data = JsonSerializer.Deserialize<BpiResponse>(json, options);
```

```
data.Bpi.Select(kvp => $"{{kvp.Key}}: {{kvp.Value.RateFloat}}").ToList()
```

```
#!csharp
```

```
// [3] Visualize the data
```

```
#r "nuget:XPlot.Plotly"
```

```
using XPlot.Plotly;
```

```
var chart = Chart.Bar(
    data.Bpi.Keys,
    data.Bpi.Values.Select(v => v.RateFloat)
```

);

```
chart.WithTitle("Bitcoin Price Index by Currency");
chart.WithLabels(data.Bpi.Keys);
chart
```

#!markdown

Summary

You just:

- Fetched **live JSON** from a public REST API
- Deserialized it into C# objects
- Used **LINQ** to explore it
- Rendered an **interactive Plotly chart** inline

This same pattern works for any API — weather, finance, IoT sensors, etc.

Tip

To try a different data source, replace the `GetStringAsync` URL with any REST endpoint that returns JSON (for example, <https://api.open-meteo.com/v1/forecast?...> for weather).

Mini-dashboard that pulls multiple APIs

Bitcoin + Ethereum prices and simple trend comparison

Awesome 😎 — this is where .NET Interactive really starts to feel like a data-science dashboard, but in **C#**.

Below is a ready-to-run notebook you can save as

👉 `CryptoDashboard.dib` or `CryptoDashboard.ipynb`
and open in **VS Code** with the **.NET Interactive Notebooks** extension.

It will:

1. Pull live price data for **Bitcoin (BTC)** and **Ethereum (ETH)** from a public API.
 2. Parse the JSON into typed C# objects.
 3. Plot an interactive chart comparing both.
 4. Show quick stats inline.
-

Crypto Dashboard — Bitcoin vs Ethereum

#!csharp

```
// [1] Fetch live price data for BTC and ETH from CoinGecko
using System;
using System.Net.Http;
using System.Text.Json;
using System.Threading.Tasks;
using System.Collections.Generic;

var http = new HttpClient();
http.DefaultRequestHeaders.Add("User-Agent", "dotnet-interactive-demo");

// CoinGecko API for current price (USD)
var url = "https://api.coingecko.com/api/v3/simple/price?ids=bitcoin,ethereum&vs_currencies=usd";
var json = await http.GetStringAsync(url);

Console.WriteLine("Raw JSON preview:");
Console.WriteLine(json);
```

#!csharp

```
// [2] Parse into C# objects
public class CryptoPrice
{
```

```
    public decimal usd { get; set; }

    var options = new JsonSerializerOptions { PropertyNameCaseInsensitive = true };
    var prices = JsonSerializer.Deserialize<Dictionary<string, CryptoPrice>>(json, options)!
```

```
foreach (var kvp in prices)
{
    Console.WriteLine($"{kvp.Key,-10} : ${kvp.Value.usd}");
}
```

```
#!csharp
// 3 Simulate recent trend data (for demo purposes)
// You could replace this with historical API calls.

var rand = new Random();

var timestamps = Enumerable.Range(0, 10).Select(i => DateTime.UtcNow.AddMinutes(-i * 5)).Reverse().ToList();

decimal btcBase = prices["bitcoin"].usd;
decimal ethBase = prices["ethereum"].usd;

var btcTrend = timestamps.Select(t => (t, btcBase + rand.Next(-2000, 2000) / 50m)).ToList();
var ethTrend = timestamps.Select(t => (t, ethBase + rand.Next(-2000, 2000) / 50m)).ToList();
```

```
#!csharp
// 4 Chart BTC vs ETH
#r "nuget:XPlot.Plotly"
using XPlot.Plotly;

var btcLine = new Scatter
{
    x = btcTrend.Select(p => p.t),
    y = btcTrend.Select(p => p.item2),
    name = "Bitcoin (BTC)",
    mode = "lines+markers"
};
```

```
var ethLine = new Scatter
{
    x = ethTrend.Select(p => p.t),
    y = ethTrend.Select(p => p.item2),
    name = "Ethereum (ETH)",
    mode = "lines+markers"
};

var layout = new Layout.Layout
{
    title = "BTC vs ETH – Live Snapshot & Simulated 45-Minute Trend",
    xaxis = new Xaxis { title = "Time (UTC)" },
    yaxis = new Yaxis { title = "Price (USD)" }
};

var chart = Chart.Plot(new[] { btcLine, ethLine }, layout);
chart
```

```
#!csharp
// 5 Quick stats
var diff = prices["bitcoin"].usd - prices["ethereum"].usd;
Console.WriteLine($"BTC: ${prices["bitcoin"].usd:N2}");
Console.WriteLine($"ETH: ${prices["ethereum"].usd:N2}");
Console.WriteLine($"BTC-ETH difference: ${diff:N2}");
```

```
#!markdown
```

Summary

- Pulled **live JSON** for BTC & ETH from CoinGecko
- Parsed it into C# dictionaries

- Generated mock short-term trends
- Rendered an ****interactive Plotly chart**** inline
- Displayed quick summary stats

You can extend this easily to include:

- Historical data endpoints (real trend lines)
 - Volume or market cap overlays
 - Alerts when price deltas exceed thresholds
 - Azure Function or Power BI connectors for automation
-

Real historical API version

Actual 24-hour trend data instead of simulated values

Reference

1. [Getting Started with C# in VS Code](#)
2. [Collections | Microsoft Learn](#)
3. [Using .NET in Visual Studio Code](#)
4. [My Courseware | Courseware Store](#)
- 5.