What's new in the SDK and tooling for .NET 10

This article describes new features and enhancements in the .NET SDK for .NET 10.

.NET tools enhancements

Platform-specific .NET tools

.NET tools can now be published with support for multiple RuntimeIdentifiers (RIDs) in a single package. Tool authors can bundle binaries for all supported platforms, and the .NET CLI will select the correct one at install or run time. This makes cross-platform tool authoring and distribution much easier.

These enhanced tools support various packaging variations:

- **Framework-dependent, platform-agnostic** (classic mode, runs anywhere with .NET 10 installed)
- **Framework-dependent, platform-specific** (smaller, optimized for each platform)
- **Self-contained, platform-specific** (includes the runtime, no .NET installation required)
- **Trimmed, platform-specific** (smaller, trims unused code)
- **AOT-compiled, platform-specific** (maximum performance and smallest deployment)

These new tools work much like normal published applications, so any publishing options you can use with applications (for example, self-contained, trimmed, or AOT) can apply to tools as well.

One-shot tool execution

You can now use the `dotnet tool exec` command to execute a .NET tool without installing it globally or locally. This is especially valuable for CI/CD or ephemeral usage.

Bash

```
dotnet tool exec --source ./artifacts/package/ dotnetsay  "Hello, World!"
Tool package dotnetsay@1.0.0 will be downloaded from source <source>.
Proceed? [y/n] (y): y
  _   _          _   _                 __        __                  _       _
 | | | |   ___  | | | |   ___          \ \      / /   ___    _ __  | |   __| |
| |
 | |_| |  / _ \ | | | |  / _ \          \ \ /\ / /   / _ \  | '__| | |  / _` |
| |
```

```
 |  _  | |  __/ | | | | | (_) |  _      \ V  V /   | (_) | | |    | | | (_| |
|_|
 |_| |_|  \___| |_| |_|  \___/  ( )      \_/\_/     \___/  |_|    |_|  \__,_|
(_)
                                |/
```

This downloads and runs the specified tool package in one command. By default, users are prompted to confirm the download if the tool doesn't already exist locally. The latest version of the chosen tool package is used unless an explicit version is specified (for example, `dotnetsay@0.1.0`).

One-shot tool execution works seamlessly with local tool manifests. If you run a tool from a location containing a `.config/dotnet-tools.json` nearby, the version of the tool in that configuration will be used instead of the latest version available.

The new `dnx` tool execution script

The `dnx` script provides a streamlined way to execute tools. It forwards all arguments to the `dotnet` CLI for processing, making tool usage as simple as possible:

Bash

```
dnx dotnetsay "Hello, World!"
```

The actual implementation of the `dnx` command is in the `dotnet` CLI itself, allowing its behavior to evolve over time.

For more information about managing .NET tools, see Manage .NET tools.

Use the `any` RuntimeIdentifier with platform-specific .NET tools

The platform-specific .NET tools feature is great for making sure tools are optimized for specific platforms that you target ahead-of-time. However, there are times where you won't know all of the platforms that you'd like to target, or sometimes .NET itself will learn how to support a new platform, and you'd like your tool to be runnable there too.

To make your tool work this way, add the `any` runtime identifier to your project file:

XML

```
<PropertyGroup>
  <RuntimeIdentifiers>
       linux-x64;
```

```
        linux-arm64;
        macos-arm64;
        win-x64;
        win-arm64;
        any
    </RuntimeIdentifiers>
</PropertyGroup>
```

This RuntimeIdentifier is at the 'root' of the platform-compatibility checking, and since it declares support for *any* platform, the tool that gets packaged will be the most compatible kind of tool - a framework-dependent, platform-agnostic .NET DLL, which requires a compatible .NET Runtime to execute. When you perform a `dotnet pack` to create your tool, you'll see a new package for the `any` RuntimeIdentifier appear alongside the other platform-specific packages and the top-level manifest package.

CLI introspection with `--cli-schema`

A new `--cli-schema` option is available on all CLI commands. When used, it outputs a JSON representation of the CLI command tree for the invoked command or subcommand. This is useful for tool authors, shell integration, and advanced scripting.

Bash

```
dotnet clean --cli-schema
```

The output provides a structured, machine-readable description of the command's arguments, options, and subcommands:

JSON

```
{
  "name": "clean",
  "version": "10.0.100-dev",
  "description": ".NET Clean Command",
  "arguments": {
    "PROJECT | SOLUTION": {
      "description": "The project or solution file to operate on. If a file
is not specified, the command will search the current directory for one.",
      "arity": { "minimum": 0, "maximum": null }
    }
  },
  "options": {
    "--artifacts-path": {
      "description": "The artifacts path. All output from the project, in-
cluding build, publish, and pack output, will go in subfolders under the
specified path.",
      "helpName": "ARTIFACTS_DIR"
    }
```

```
  },
  "subcommands": {}
}
```

Use .NET MSBuild tasks with .NET Framework MSBuild

MSBuild is the underlying build system for .NET, driving both build of projects (as seen in commands like `dotnet build` and `dotnet pack`), and acting as a general provider of information about projects (as seen in commands like `dotnet list package`, and implicitly used by commands like `dotnet run` to discover how a project wants to be executed).

When running `dotnet` CLI commands, the version of MSBuild that is used is the one that is shipped with the .NET SDK. However, when using Visual Studio or invoking MSBuild directly, the version of MSBuild that is used is the one that is installed with Visual Studio. This environment difference has a few important consequences. The most important is that MSBuild running in Visual Studio (or through `msbuild.exe`) is a .NET Framework application, while MSBuild running in the `dotnet` CLI is a .NET application. This means that any MSBuild tasks that are written to run on .NET can't be used when building in Visual Studio or when using `msbuild.exe`.

Starting with .NET 10, `msbuild.exe` and Visual Studio 2026 can run MSBuild tasks that are built for .NET. This means that you can now use the same MSBuild tasks when building in Visual Studio or using `msbuild.exe` as you do when building with the `dotnet` CLI. For most .NET users, this won't change anything. But for authors of custom MSBuild tasks, this means that you can now write your tasks to target .NET and have them work everywhere. The goal with this change is to make it easier to write and share MSBuild tasks, and to allow task authors to take advantage of the latest features in .NET. In addition, this change reduces the difficulties around multi-targeting tasks to support both .NET Framework and .NET, and dealing with versions of .NET Framework dependencies that are implicitly available in the MSBuild .NET Framework execution space.

Configure .NET tasks

For task authors, it's easy to opt into this new behavior. Just change your `UsingTask` declaration to tell MSBuild about your task.

XML

```
<UsingTask TaskName="MyTask"
    AssemblyFile="path\to\MyTask.dll"
```

```
    Runtime="NET"
    TaskFactory="TaskHostFactory"
/>
```

The `Runtime="NET"` and `TaskFactory="TaskHostFactory"` attributes tell the MSBuild engine how to run the Task:

- `Runtime="NET"` tells MSBuild that the Task is built for .NET (as opposed to .NET Framework).
- `TaskFactory="TaskHostFactory"` tells MSBuild to use the `TaskHostFactory` to run the Task, which is an existing capability of MSBuild that allows tasks to be run out-of-process.

Caveats and performance tuning

The preceding example is the simplest way to get started using .NET tasks in MSBuild, but it has some limitations. Because the `TaskHostFactory` always runs tasks out-of-process, the new .NET task always runs in a separate process from MSBuild. This means that there is some minor overhead to running the task because the MSBuild engine and the task communicate over inter-process communication (IPC) instead of in-process communication. For most tasks, this overhead is negligible, but for tasks that are run many times in a build, or that do a lot of logging, this overhead might be more significant.

With just a bit more work, you can configure the task to still run in-process when running via `dotnet`:

XML

```
<UsingTask TaskName="MyTask"
    AssemblyFile="path\to\MyTask.dll"
    Runtime="NET"
    TaskFactory="TaskHostFactory"
    Condition="$(MSBuildRuntimeType) == 'Full'"
/>
<UsingTask TaskName="MyTask"
    AssemblyFile="path\to\MyTask.dll"
    Runtime="NET"
    Condition="$(MSBuildRuntimeType) == 'Core'"
/>
```

Thanks to the `Condition` feature of MSBuild, you can load a Task differently depending on whether MSBuild is running in .NET Framework (Visual Studio or `msbuild.exe`) or .NET (the `dotnet` CLI). In this example, the Task runs out-of-process when running in Visual Studio or `msbuild.exe`, but runs in-process when running in the `dotnet` CLI. This gives the best

performance when running in the `dotnet` CLI, while still allowing the Task to be used in Visual Studio and `msbuild.exe`.

There are also small technical limitations to be aware of when using .NET Tasks in MSBuild—the most notable of which is that the `Host Object` feature of MSBuild Tasks isn't yet supported for .NET Tasks running out-of-process. This means that if your Task relies on a Host Object, it won't work when running in Visual Studio or `msbuild.exe`. Additional support for Host Objects is planned in future releases.

File-based apps enhancements

.NET 10 brings significant updates to the file-based apps experience, including publish support and native AOT capabilities. For an introduction to file-based apps, see File-based apps and Building and running C# programs.

Enhanced file-based apps with publish support and native AOT

File-based apps now support being published to native executables via the `dotnet publish app.cs` command, making it easier to create simple apps that you can redistribute as native executables. All file-based apps now target native AOT by default. If you need to use packages or features that are incompatible with native AOT, you can disable this using the `#:property PublishAot=false` directive in your .cs file.

File-based apps also include enhanced features:

- **Project referencing**: Support for referencing projects via the `#:project` directive.
- **Runtime path access**: App file and directory paths are available at runtime via `System.AppContext.GetData`.
- **Enhanced shebang support**: Direct shell execution with improved shebang handling, including support for extensionless files.

Project referencing example

C#

```
#:project ../ClassLib/ClassLib.csproj

var greeter = new ClassLib.Greeter();
var greeting = greeter.Greet(args.Length > 0 ? args[0] : "World");
Console.WriteLine(greeting);
```

Enhanced shebang support example

You can now create executable C# files that run directly from the shell:

C#

```
#!/usr/bin/env dotnet

Console.WriteLine("Hello shebang!");
```

For extensionless files:

Bash

```
# 1. Create a single-file C# app with a shebang
cat << 'EOF' > hello.cs
#!/usr/bin/env dotnet
Console.WriteLine("Hello!");
EOF

# 2. Copy it (extensionless) into ~/utils/hello (~/utils is on my PATH)
mkdir -p ~/utils
cp hello.cs ~/utils/hello

# 3. Mark it executable
chmod +x ~/utils/hello

# 4. Run it directly from anywhere
cd ~
hello
```

These enhancements make file-based apps more powerful while maintaining their simplicity for quick scripting and prototyping scenarios.

For more information about native AOT, see .NET native AOT.

Pruning of framework-provided package references

Starting in .NET 10, the NuGet Audit feature can prune framework-provided package references that aren't used by the project. This feature is enabled by default for all frameworks of a project that targets >= .NET 10.0 in the latest SDK. This change helps reduce the number of packages that are restored and analyzed during the build process, which can lead to faster build times and reduced disk space usage. It also can lead to a reduction in false positives from NuGet Audit and other dependency-scanning mechanisms.

When this feature is enabled, you might see a reduction in the contents of your applications' generated *.deps.json* files. Any package references supplied by the .NET runtime are automatically removed from the generated dependency file. When a direct package reference is within the pruning range, PrivateAssets="all" and IncludeAssets="none" are applied.

While this feature is enabled by default for the listed TFMs, you can disable it by setting the `RestoreEnablePackagePruning` property to `false` in your project file or *Directory.Build.props* file.

More consistent command order

Starting in .NET 10, the `dotnet` CLI tool includes new aliases for common commands to make them easier to remember and type. The new commands are shown in the following table.

Expand table

New noun-first form	Alias for
dotnet package add	`dotnet add package`
dotnet package list	`dotnet list package`
dotnet package remove	`dotnet remove package`
dotnet reference add	`dotnet add reference`
dotnet reference list	`dotnet list reference`
dotnet reference remove	`dotnet remove reference`

The new noun-first forms align with general CLI standards, making the `dotnet` CLI more consistent with other tools. While the verb-first forms continue to work, it's better to use the noun-first forms for improved readability and consistency in scripts and documentation.

CLI commands default to interactive mode in interactive terminals

The `--interactive` flag is now enabled by default for CLI commands in interactive terminals. This change allows commands to dynamically retrieve credentials or perform other interactive behaviors without requiring the flag to be explicitly set. For noninteractive scenarios, you can disable interactivity by specifying `--interactive false`.

Native shell tab-completion scripts

The `dotnet` CLI now supports generating native tab-completion scripts for popular shells using the `dotnet completions generate [SHELL]` command. Supported shells include `bash`, `fish`, `nushell`, `powershell`, and `zsh`. These scripts improve usability by providing faster and more integrated tab-completion features. For example, in PowerShell, you can enable completions by adding the following to your `$PROFILE`:

PowerShell

```
dotnet completions script pwsh | out-String | Invoke-Expression -ErrorAction SilentlyContinue
```

Console apps can natively create container images

Console apps can now create container images via `dotnet publish /t:PublishContainer` without requiring the `<EnableSdkContainerSupport>` property in the project file. This aligns console apps with the behavior of ASP.NET Core and Worker SDK apps.

Explicitly control the image format of containers

A new `<ContainerImageFormat>` property allows you to explicitly set the format of container images to either `Docker` or `OCI`. This property overrides the default behavior, which depends on the base image format and whether the container is multi-architecture.

Support for Microsoft Testing Platform in `dotnet test`

Starting in .NET 10, `dotnet test` natively supports Microsoft.Testing.Platform. To enable this feature, add the following configuration to your *global.json* file:

JSON

```
{
    "test": {
        "runner": "Microsoft.Testing.Platform"
    }
}
```

For more details, see Testing with dotnet test.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Last updated on 11/07/2025