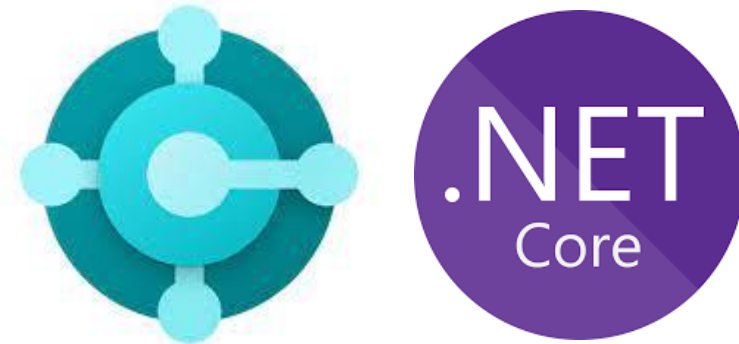


# Lessons learned from migrating to .NET Core

Wael AbuSeada  
Vladislav Nagorny



# Agenda

- Why .NET Core?
- Planning the migration
- Migration techniques
- Common challenges and solutions
- Handling non-migratable components
- Results and benefits post-migration
- Q&A

**Why .NET Core?**

# .NET Core

## Modular

- small framework assemblies, faster load and less RAM use

## Performance

- significant runtime performance improvements comparing to .NET Framework

## Cross-platform

- not just Windows, also supports Linux and macOS

## Flexible deployment

- exe can be self-contained, or use system-wide runtime

## Better/simpler build tools

- powerful CLI, simplified dev experience, faster compilation

## Open-source

- community contributions and transparent development process

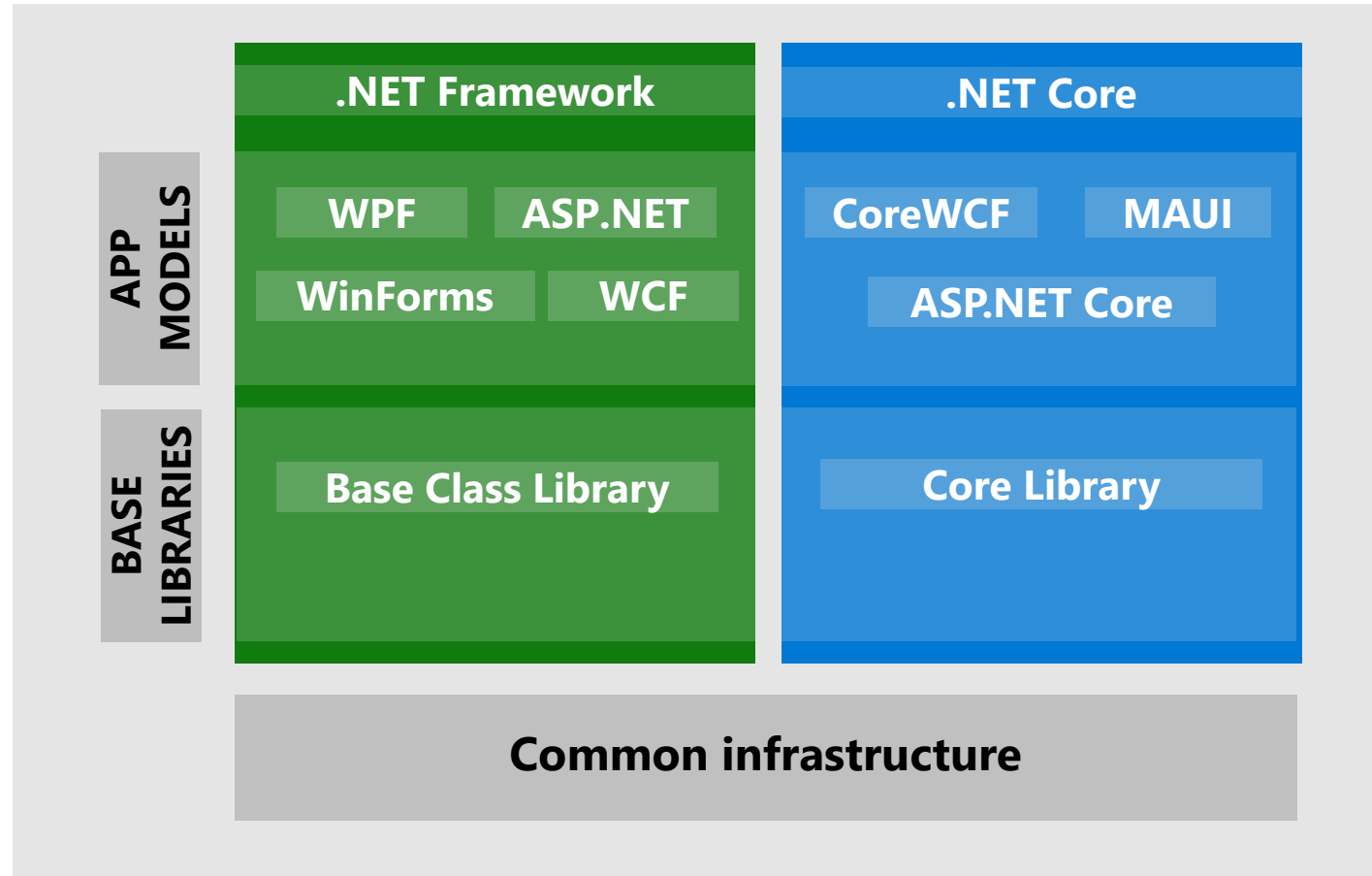
## Containers

- with small footprint and support for Linux and Windows Nano Server, great match for containers and microservices

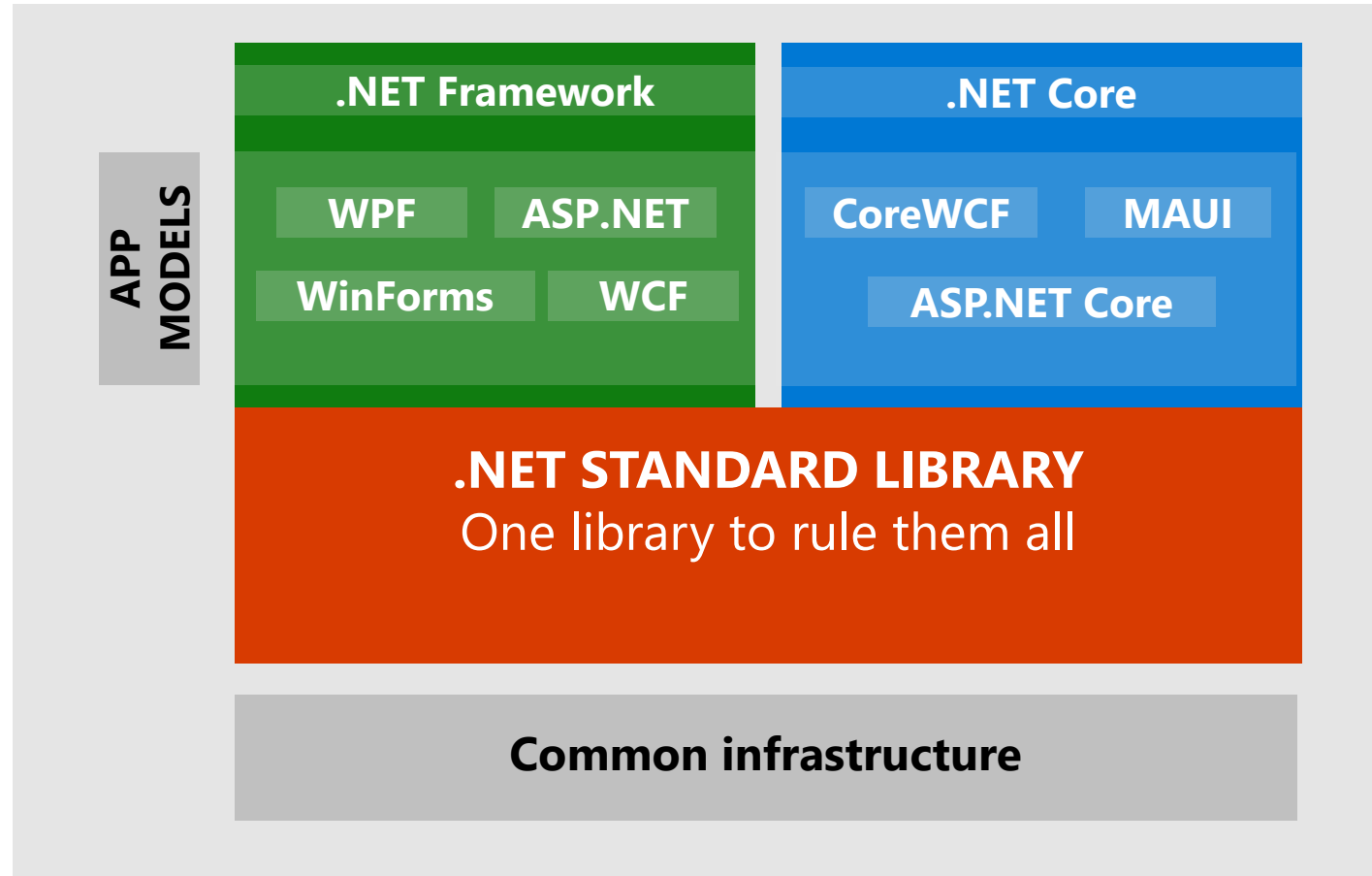
# Planning the migration



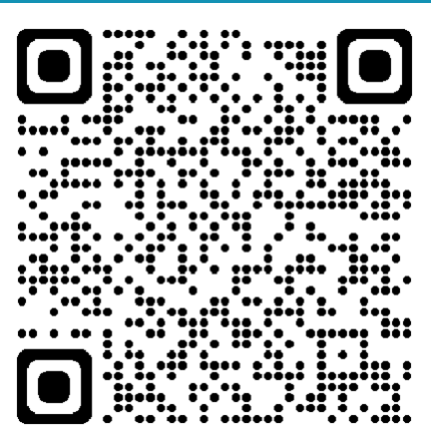
# .NET Framework vs .NET Core



# .NET Standard



# .NET migration guide



Microsoft | Learn Documentation Training Certifications Q&A Code Samples Assessments Shows Events

.NET Languages Features Workloads APIs Resources

Filter by title

.NET migration guide

Overview

- General information
  - About .NET
  - .NET Upgrade Assistant tool
  - Breaking changes
- Pre-migration
  - Assess the portability of your project
    - Needed changes before porting code
- Migration
  - Create a porting plan
  - Application porting guides
- Post-migration
  - Modernize

Download PDF

Learn / .NET / Migration guide /

## Overview of porting from .NET Framework to .NET

Article • 04/05/2023 • 29 contributors [Feedback](#)

### In this article

- Windows desktop technologies
- Windows-specific APIs
- .NET Framework compatibility mode
- Target framework changes in SDK-style projects
- Unavailable technologies
- Cross-platform
- The future of .NET Standard
- Tools to assist porting
- Considerations when porting
- See also
- Show less

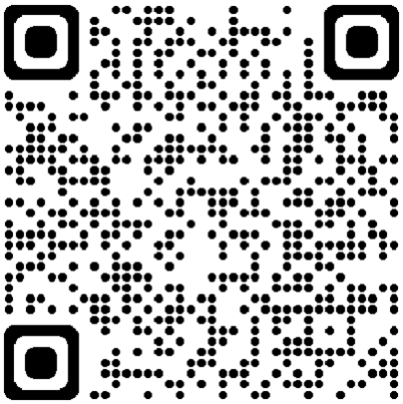
<https://learn.microsoft.com/en-us/dotnet/core/porting/>



# .NET Upgrade Assistant

Easiest way to start – possibly the only step needed

<https://learn.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-overview>



# .NET Upgrade Assistant



MyAddIn

C:\repos\Scratchpad\BCTechDays\MyAddIn

[Leave Feedback](#)

Welcome to the Upgrade Assistant!

This experience will guide you through the process of upgrading your project towards newer technologies.

Upgrade Assistant can help upgrade .NET Framework or .NET applications to newer .NET versions, bringing its cross-platform, high-performance capabilities to your product. [Learn more](#)

Ready for upgrade? Select how you want to upgrade MyAddIn



## In-place project upgrade

Upgrades project and its components in place using transformations applicable for the project.



## Side-by-side project upgrade

Upgrades project and its components in a copy project using transformations applicable for the project.



## Side-by-side incremental project upgrade

Selects or creates new target project and establishes a link between current and target projects to allow continuous move of project components one at a time.

# .NET Upgrade Assistant



MyAddIn

C:\repos\Scratchpad\BCTechDays\MyAddIn

[Leave Feedback](#)

← Target framework

What is your preferred target framework?

Next

☒ .NET 6.0 LTS

*Supported until november, 2024*

☐ .NET 7.0 STS

*Supported until maj, 2024*

☐ .NET 8.0 PREVIEW


*Try latest preview features*

☐ .NETStandard 2.0

☐ .NETStandard 2.1

[Which target framework is right for you?](#)

# .NET Upgrade Assistant


 MyAddIn  
C:\repos\Scratchpad\BCTechDays\MyAddIn


[Leave Feedback](#)


[←](#) [Target framework](#) > [Select components](#) > Upgrade


Upgrade selected components


Complete: 6 succeeded, 0 failed, 2 skipped.8/8


▸  MyAddIn.csproj ✓


▸  System ✓


▸  System.Data ✓

▸  System.Xml ✓

▸  Finalize project dependencies ✓

▸  Class1.cs ✓

▸  Class2.cs ✓

▸  AssemblyInfo.cs (Properties) ✓

Done?

# Manual migration

- List incompatible functionality and components
- Establish dependencies between components
- Define migration strategy for each component
- Start executing

# Migration techniques



# Target technology



Is there an alternative functionality in .NET Core?



Is there an alternative functionality in a third-party NuGet?



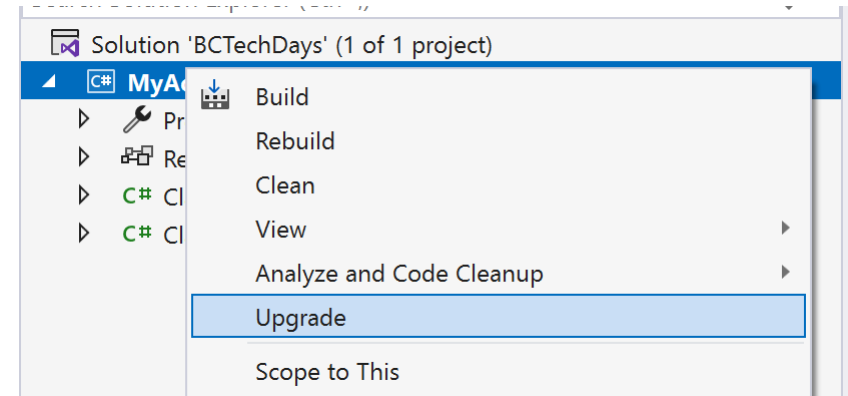
Has component been planned for refactoring or deprecation?



# In-place vs side-by-side project migrations

- **In-place**

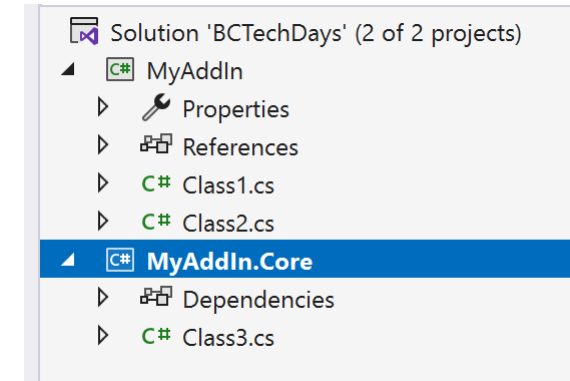
- Make changes directly in the project
- Suitable for small or relatively isolated components



# In-place vs side-by-side project migrations

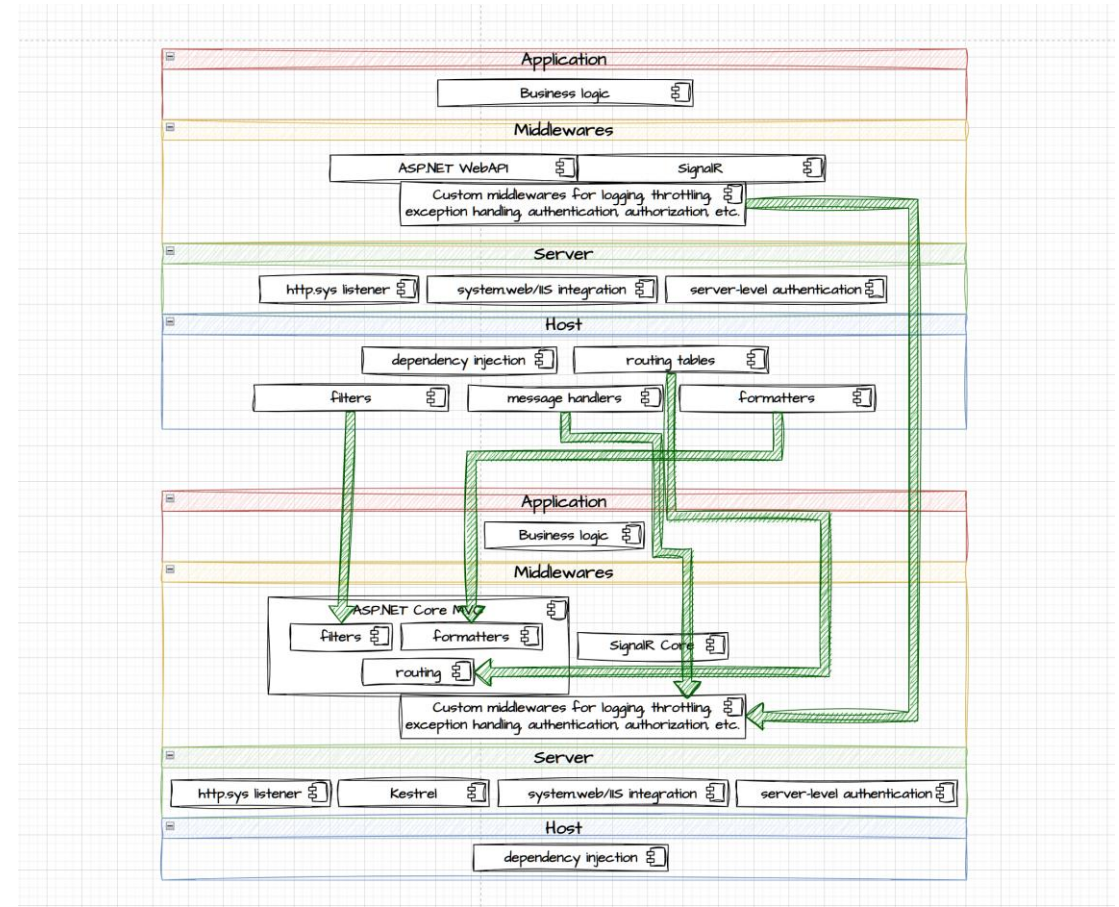
- **Side-by-side**

- Create another project
- Move functionality gradually
- Reduces complexity for migration of larger components,  
e.g. ASP.NET WebAPI layer



# OWIN/ASP.NET

- Was one of the biggest tasks during migration – Business Central Server had 11 endpoints based on OWIN
- OWIN and ASP.NET Core are very similar, yet there's a lot of nuances
- ASP.NET Core up until 2.2 is supported by .NET Framework, helps gradual migration
- SignalR vs SignalR Core, same name but different protocols



# System.Text.Json vs Newtonsoft.Json

- ASP.NET Core 3.0 uses System.Text.Json by default, Newtonsoft.Json is still available
- System.Text.Json is faster
- Keeping Newtonsoft.Json may be a better short-term solution for API compatibility

# WCF

- Evaluate alternatives
- Prefer replacement based on ASP.NET Core for future-proof solution
  - CoreWCF – ASP.NET Core based middleware that partially implements WCF
  - Plain controllers
  - gRPC
  - WebSocket
  - Custom middleware

# app.config/web.config vs appsettings.json

- app.config/web.config are still available via System.Configuration.ConfigurationManager
- Some settings are silently ignored
  - <gcServer enabled="true" />

```
App.config
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Setting1" value="Value1"/>
    <add key="Setting2" value="Value2"/>
  </appSettings>
  <connectionStrings>
    <add name="MyDatabase" connectionString="Data Source=(local);Initial Catalog=MyDB;Integrated Security=True" providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

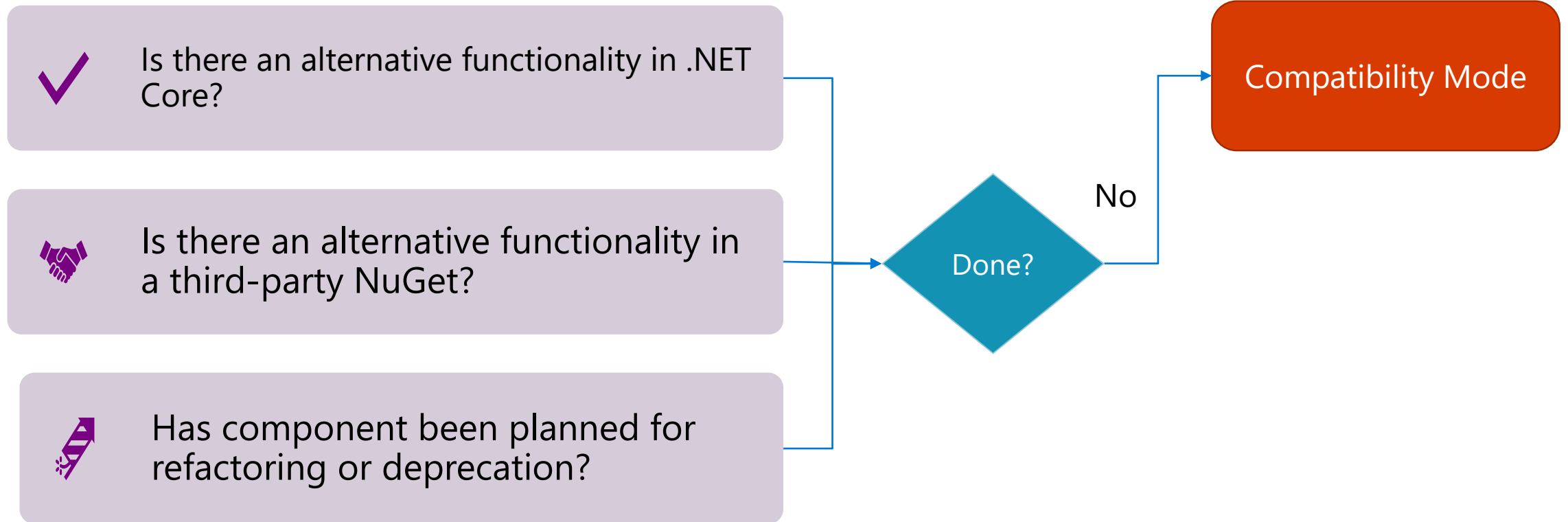
```
appsettings.json
Schema: https://json.schemastore.org/appsettings.json
1 {
2   "AppSettings": {
3     "Setting1": "Value1",
4     "Setting2": "Value2"
5   },
6   "ConnectionStrings": {
7     "MyDatabase": "Data Source=(local);Initial Catalog=MyDB;Integrated Security=True"
8   }
9 }
10
```

# Updating AL assembly probing paths

```
settings.json X
.vscode > {} settings.json > [ ] al.assemblyProbingPaths
1 {
2   "al.packageCachePath": "C:\\AllExtensions",
3   "al.assemblyProbingPaths": [
4     "C:\\Windows\\Microsoft.NET\\assembly",
5     "C:\\Program Files\\Microsoft Dynamics 365 Business Central\\220\\Service"
6   ]
7 }
8
```

```
launch.json {} settings.json X
.vscode > {} settings.json > [ ] al.assemblyProbingPaths
1 {
2   "al.packageCachePath": "C:\\AllExtensions",
3   "al.assemblyProbingPaths": [
4     "C:\\Program Files\\dotnet\\shared\\Microsoft.AspNetCore.App\\6.0.18",
5     "C:\\Program Files\\dotnet\\shared\\Microsoft.NETCore.App\\6.0.18",
6     "C:\\Program Files\\dotnet\\shared\\Microsoft.WindowsDesktop.App\\6.0.18",
7     "C:\\Program Files\\Microsoft Dynamics 365 Business Central\\220\\Service"
8   ]
9 }
10
```

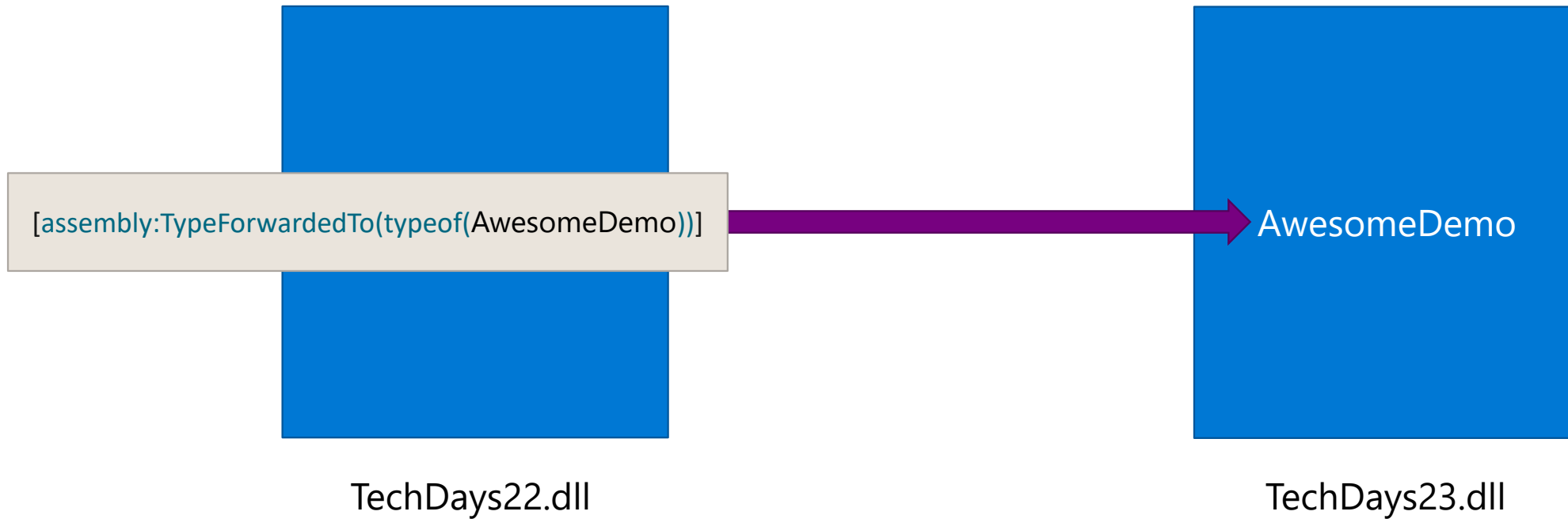
# Target technology



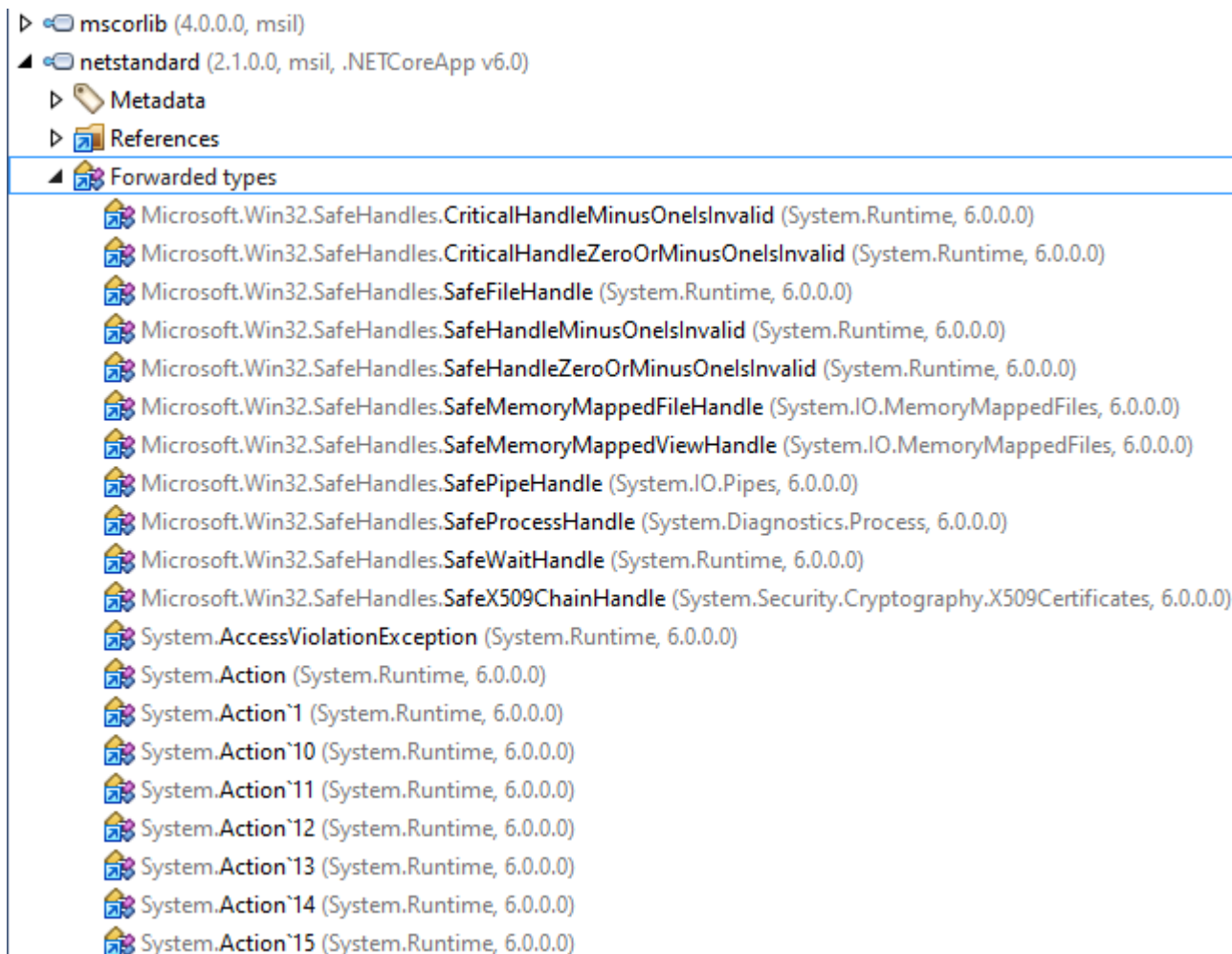


# Type forwarding

Supported in AL compiler

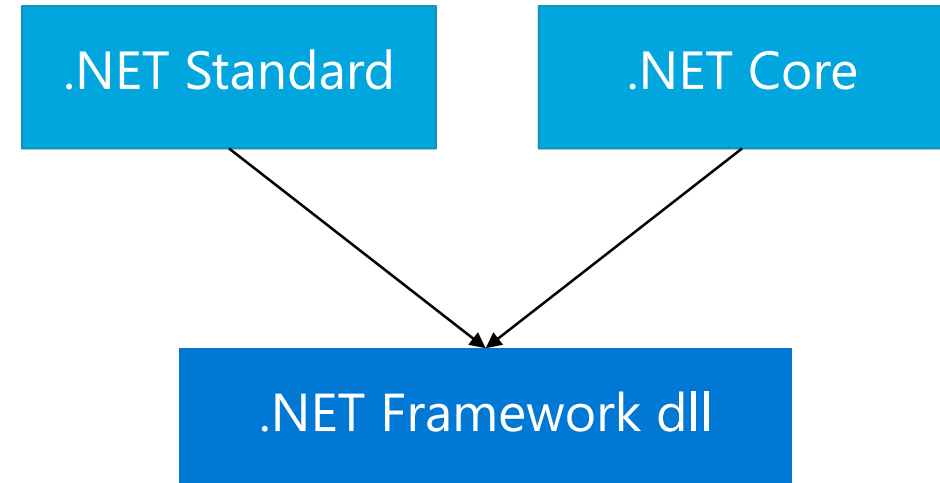


# Type forwarding in net standard



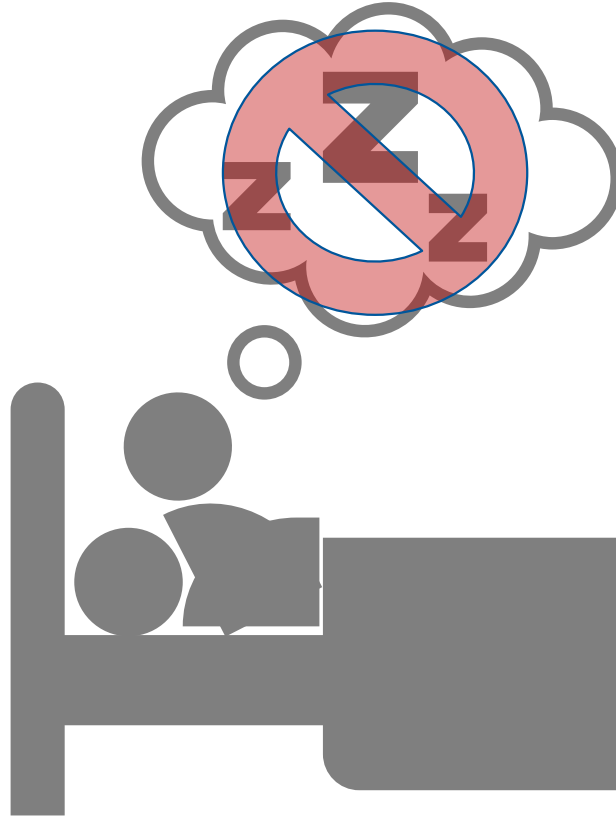
# .NET Framework Compatibility Mode

- It allows referencing .NET Framework libraries
- Possible thanks to type forwarding
- Why not just use compatibility mode?
  - Compile fine ☒
  - Doesn't cover all .NET Framework, but only .NET Standard subset of .NET APIs
  - Runtime error for unsupported APIs ☐
  - This happened to our OneDrive integration
  - Your .NET framework dlls will be loaded in compatibility mode, so you might get runtime errors

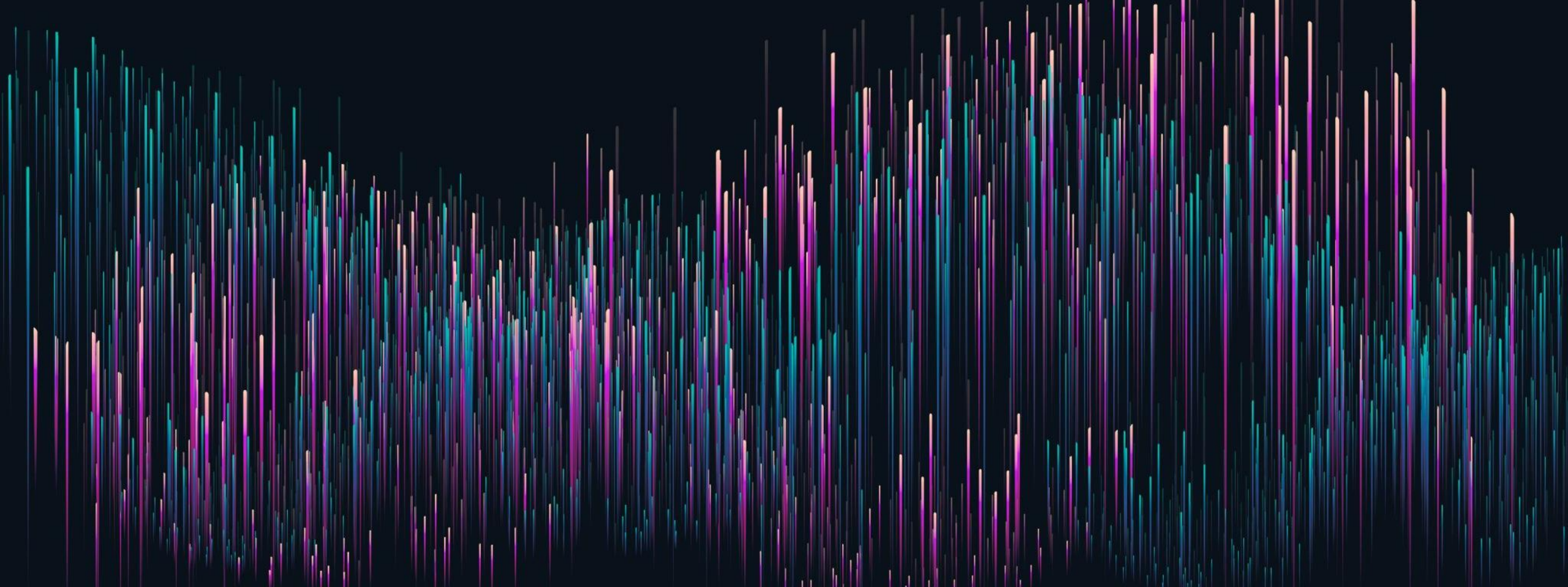


How to handle the  
stress of migrating  
huge project?

**Wake up scared**

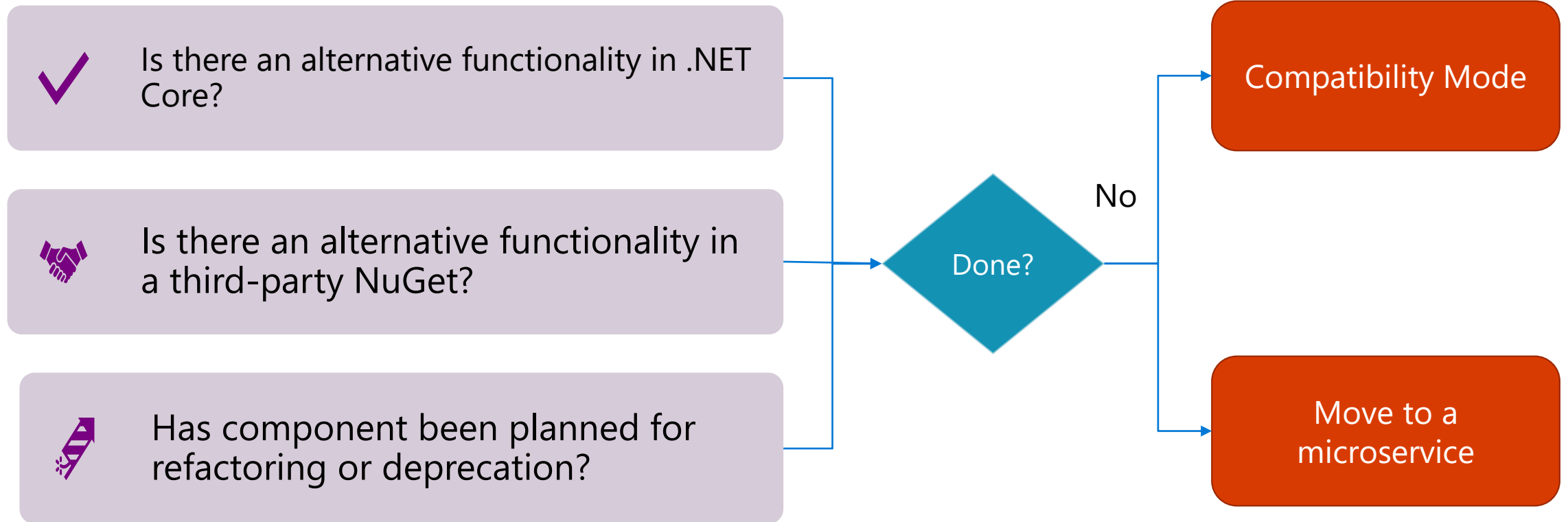


**Remember we have good test  
coverage on platform and application**



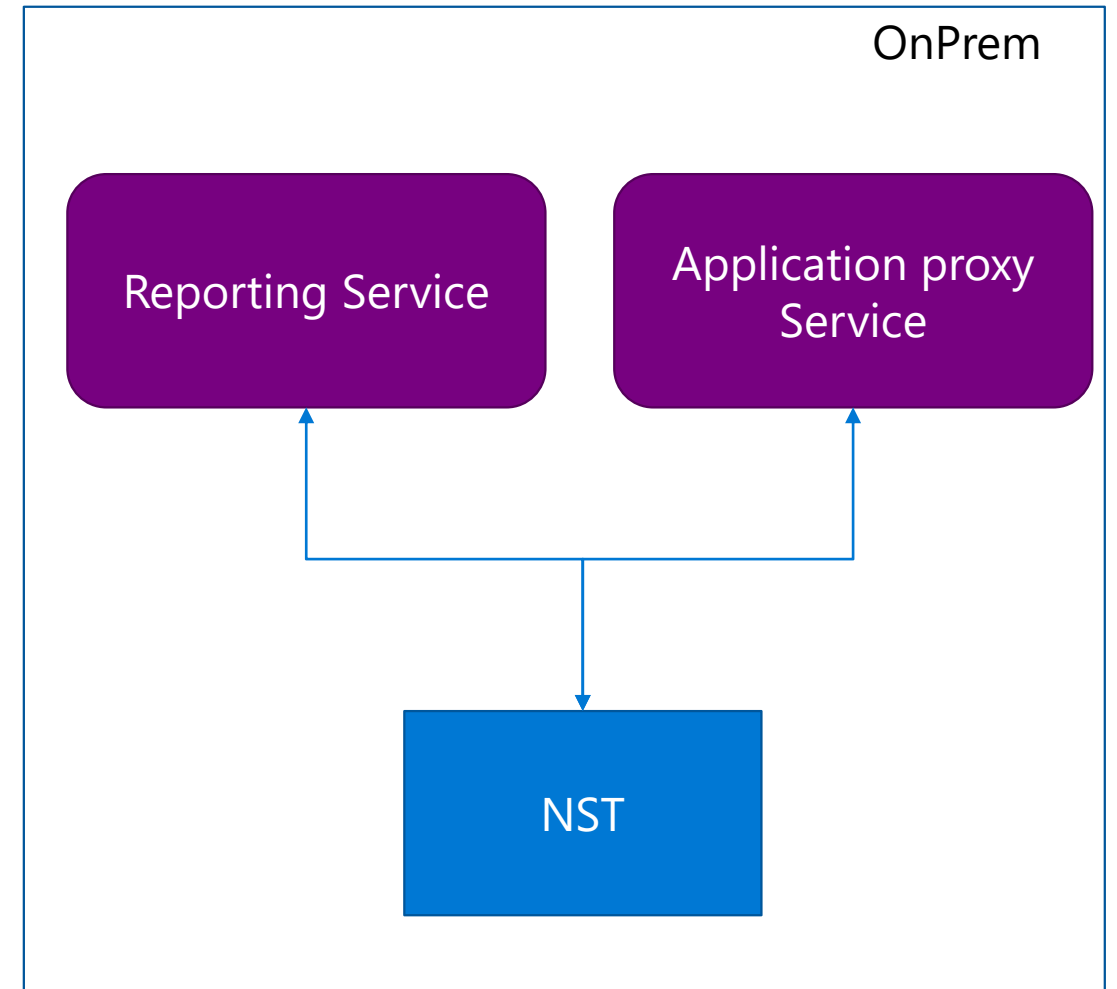
**Always invest in test automation**

# Target technology



# Moved to a microservice

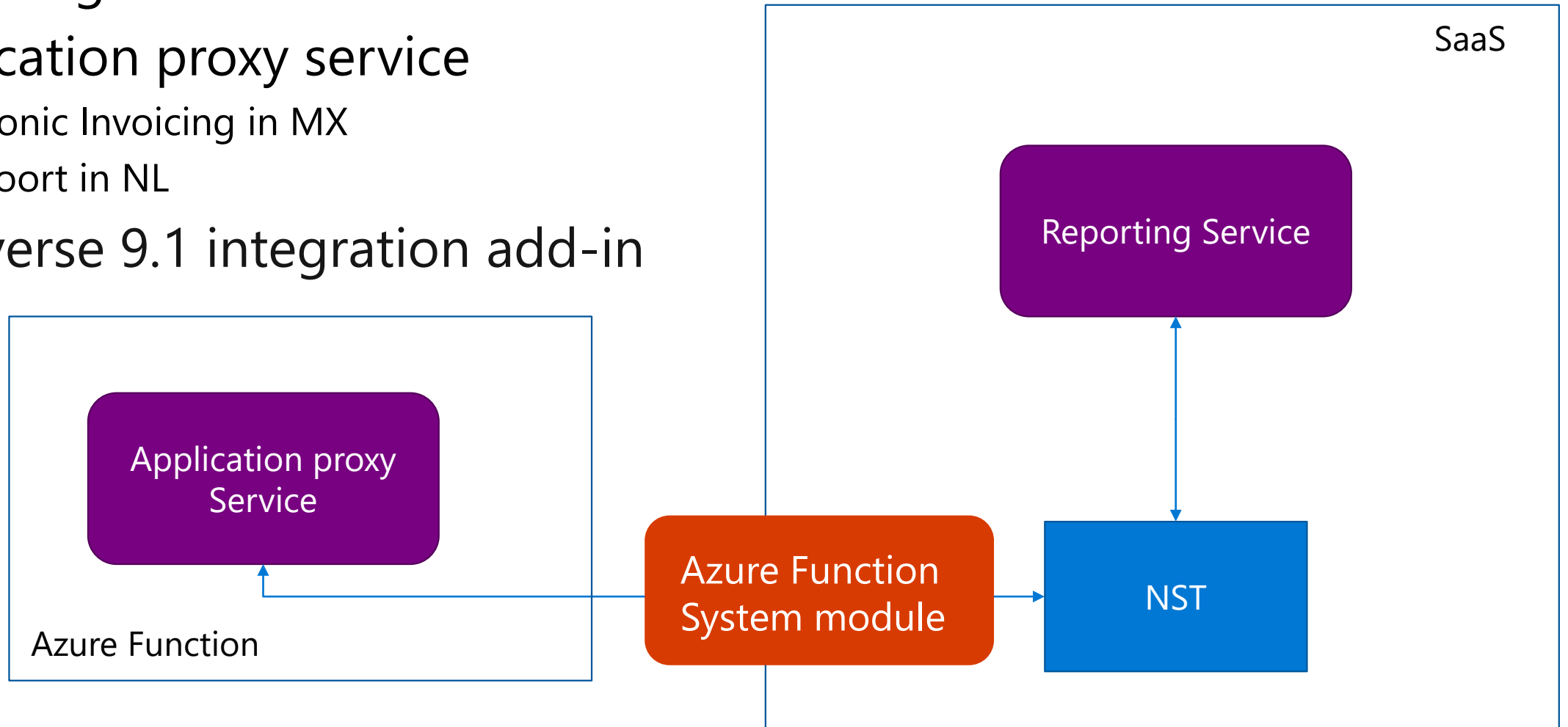
- Reporting service
- Application proxy service
  - Electronic Invoicing in MX
  - Digipoort in NL
- Dataverse 9.1 integration add-in





# Moved to a microservice

- Reporting service
- Application proxy service
  - Electronic Invoicing in MX
  - Digipoort in NL
- Dataverse 9.1 integration add-in



Done

# Other interesting changes in .NET Core

- `Encoding.Default` in .NET Core is always UTF8
  - In .NET Framework it was always the system's active code page
  - The documentation warns against using `Encoding.Default` due to the possible changes between .NET versions, from machine to machine, and within the same machine over time
- `Debug.Assert` in .NET Core will crash the process when assertion fails
  - In .NET Framework it would show interactive dialog, offering to ignore, to attach a debugger or to abort application execution
- Readonly fields in .NET Core cannot be modified using reflection
  - It was possible to write to a readonly field in .NET Framework using reflection

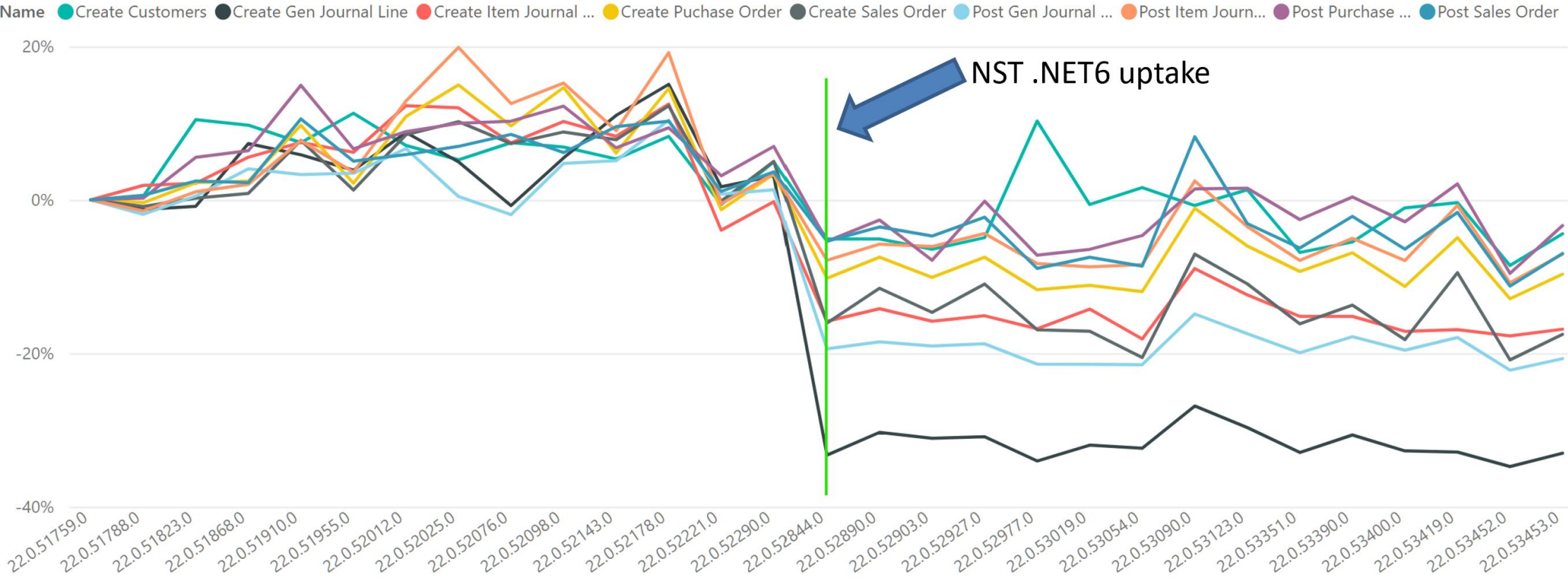
# Results and benefits of migration



# Performance

- In general, there are consistent computational performance improvements across all areas
- App scenarios up to 30% faster ↗
- Admin tasks (server startup, AL compilation, environment start-up, ...) up to 55% faster

# Performance



# Performance

Scenario	Time 21.x	Time 22.x	Improvement %
Build nav binaries (cold)	11:04.0	08:31.0	-23%
Build nav binaries (warm)	05:25.0	04:01.0	-25.8%

# Q&A

Any Questions?





Thank  
You!

