C# 13 und C# 14 – Was gibt es Neues?

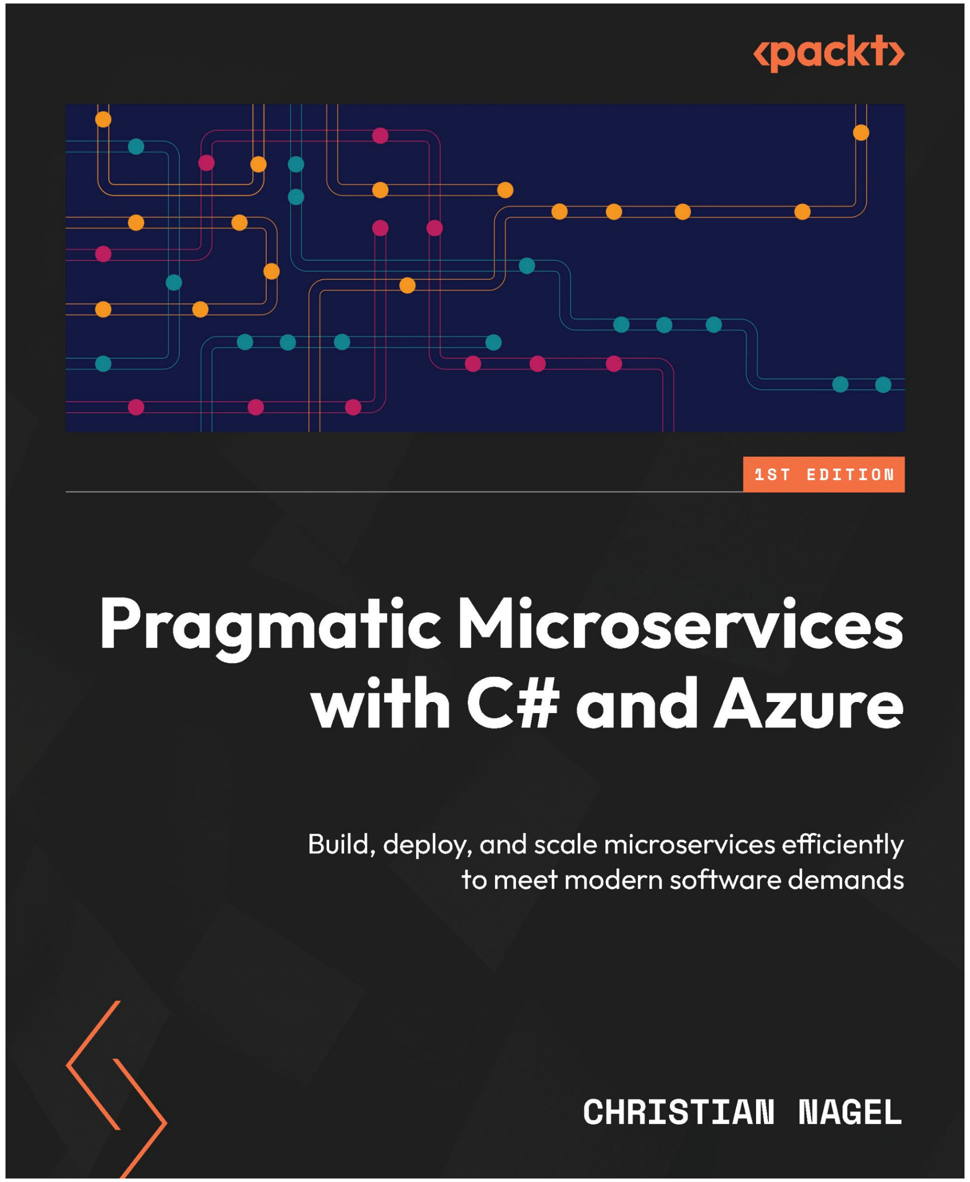
Christian Nagel

https://www.cninnovation.com

Christian Nagel

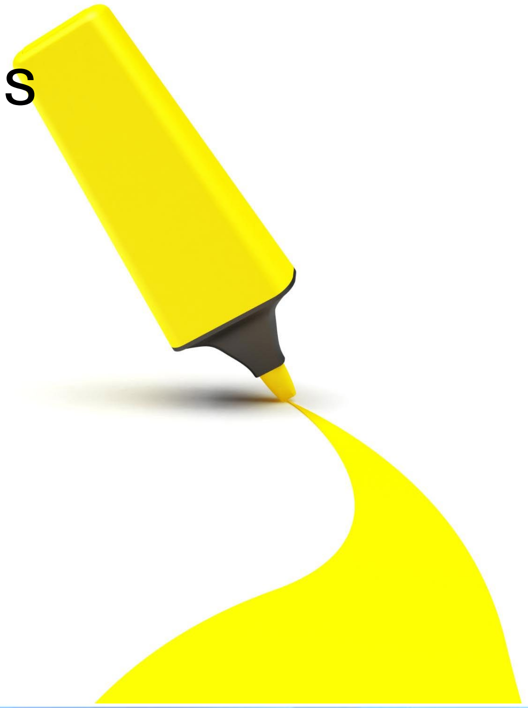
- Training
- Coaching
- Consulting
- Development
- New book: **Pragmatic Microservices**

- Microsoft MVP
- www.cninnovation.com
- csharp.christiannagel.com
- @christiannagel

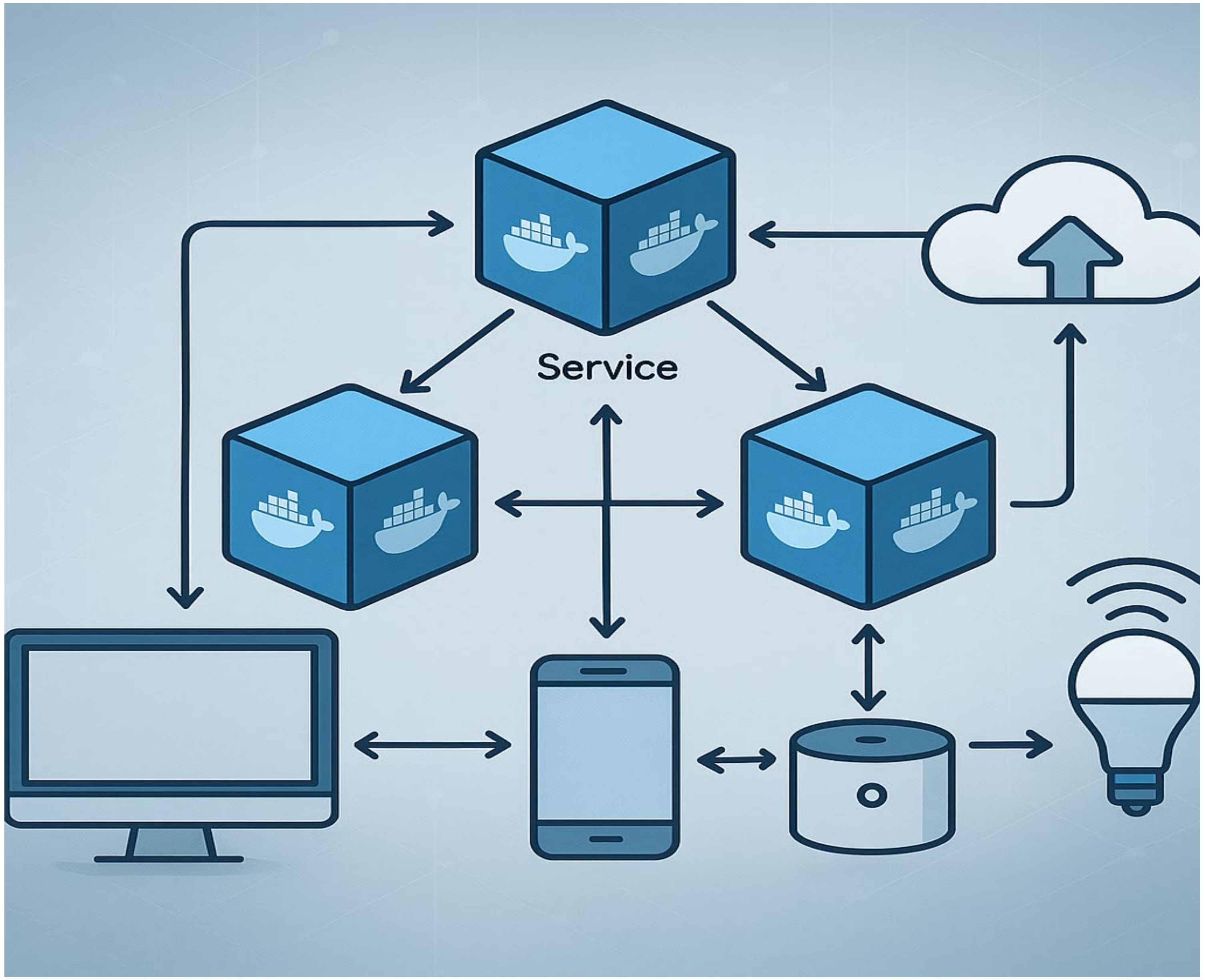

‹packt›
1ST EDITION
Pragmatic Microservices with C# and Azure
Build, deploy, and scale microservices efficiently to meet modern software demands
CHRISTIAN NAGEL

What did we do in the early C# days?

- Windows Forms
- ASP.NET Web Forms
- ASP.NET XML Web Services
- All running on Windows



What do we do today?



Explore the latest features with one of the most powerful programming language!

Simple things → Type enhancements → Span → Async

Simple, but practical…

Escape (C# 13)

Make escape codes easier

\e instead of \u001b

VT100 escape characters

Implicit Index access (C# 13)

- In object initializers

```
TimerRemaining countdown = new()
{
  Buffer =
  {
    [^1] = 0,
    [^2] = 1,
    [^3] = 2,
  }
};
```

Nameof with unbound generics (C# 14)

```
string name = nameof(List<>);
Console.WriteLine(name); // List
```

String literals in data sections (C# 14)

```
string s1 = "the quick brown fox jumped";
ReadOnlySpan<byte> s2 = "the quick brown fox jumped"u8;
```

- string literals in UserString heap (default)
- Limited to 2^24 bytes
- Enable with feature flag

```
<PropertyGroup>
  <Features>$(Features);
    experimental-data-section-string-literals=20
  </Features>
</PropertyGroup>
```

Type Members

Field-backed Properties

- field keyword
- C# 13 preview
- C# 14 release

```
// full property
private int _x;
public int X
{
  get => _x;
  set => _x = value;
}
```

```
// auto property
public int X { get; set; }
```

```
// semi-auto property
public int X
{
  get => field;
  set => field = value;
}
```

```
public MyType X => field ??= ComputeValue();
```

Partial properties and indexers (C# 13)

```
partial class Book
{
  public partial string Title { get; set; }
}
```

```
partial class Book
{
  private string _title;
  public partial string Title
  {
    get => _title;
    set => _title = value;
  }
}
```

Partial events and constructors (C# 14)

- Extensions are “transparent wrappers”
- Augmented with additional members and interfaces

```
public partial class Publisher
{
  [WeakEvent]
  public partial event Action<int, string> MyEvent;
}
```

```
public partial class Publisher
{
  public partial event Action<int, string> MyEvent
  {
    add { }
    remove { }
  }
}
```

Extensions (C# 14)

- Methods, properties, indexers, operators

```
public static class Enumerable
{
    // New extension declaration
    extension(IEnumerable source) { … }

    // Classic extension method
    public static IEnumerable<TResult> Cast<TResult>(this IEnumerable source) { … }

    // Non-extension member
    public static IEnumerable<int> Range(int start, int count) { … }
}
```

Extension blocks with methods

```
public static class Vector3DGeometry
{
    extension (Vector3D vector)
    {
        public Vector3D Normalize()
        {
            var magnitude = vector.Magnitude;
            if (Math.Abs(magnitude) < double.Epsilon)
                throw new InvalidOperationException("Cannot normalize a zero vector");

            return vector / magnitude;
        }
    }
}
```

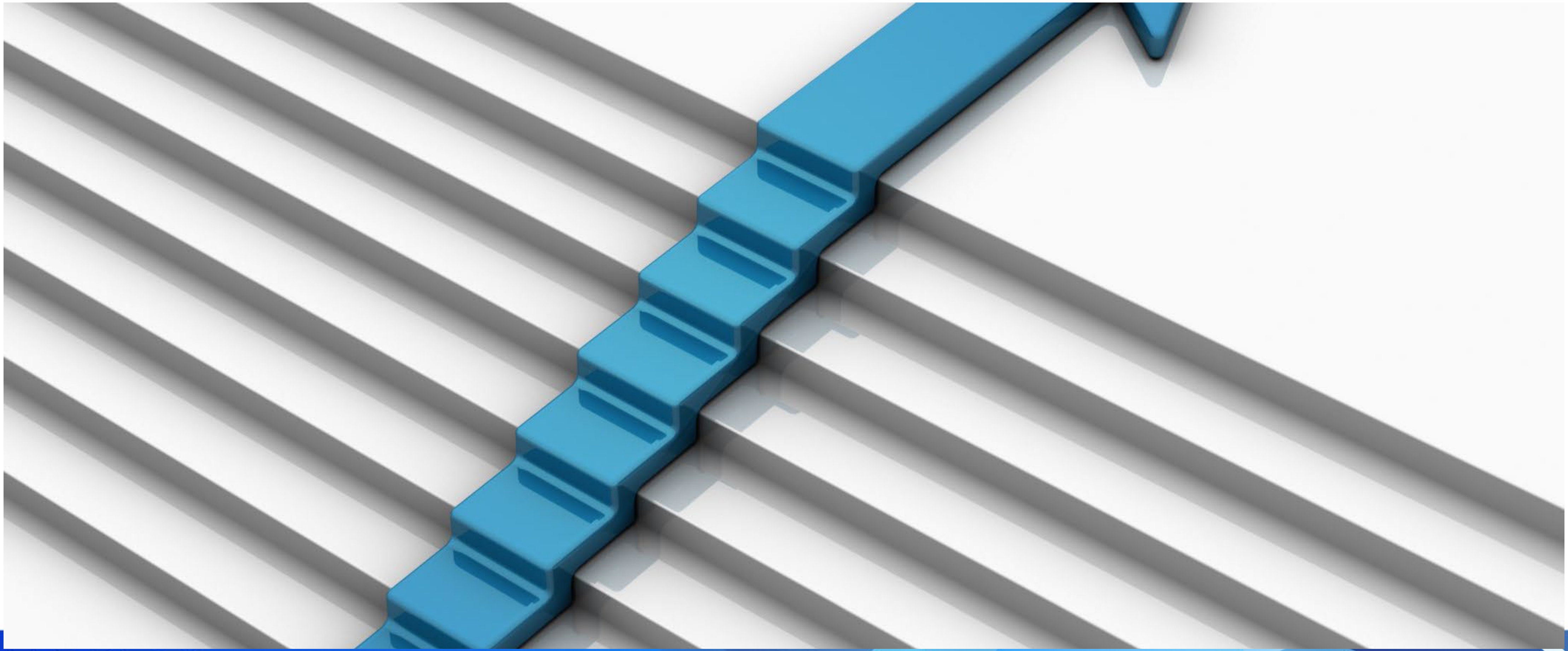
Extension blocks with properties

```
public static class Vector3DGeometry
{
    extension (Vector3D vector)
    {
        public double Magnitude =>
            Math.Sqrt(vector.X * vector.X +
                vector.Y * vector.Y + vector.Z * vector.Z);
    }
}
```

Extension blocks with operators

```
public static class Vector3DGeometry
{
    extension (Vector3D)
    {
        public static Vector3D operator +(Vector3D left, Vector3D right)
            => new(left.X + right.X, left.Y + right.Y, left.Z + right.Z);
    }
}
```

Ref Struct & Span Enhancements



Ref struct enhancements (C# 13, .NET 9)

- What is a ref struct?
- Compare struct .vs. class .vs. ref struct
- Before C# 13 – ref struct can’t implement interfaces
- C# 13
  - ref struct implement interfaces
  - Generic anti constraint: ***allows ref struct***

Params collections (C# 13)

- Params modifier not limited to arrays

```
void Foo(params IEnumerable<T> items)
{ }
```

```
void Foo(params Span<T> items)
{ }
```

```
void Foo(params MyCollection items)
{ }
```

First class Span<T> (C# 14)

- T[] → Span<T>
- T1[] → ReadOnlySpan<T2>
  - Covariance-compatible
- Span<T1> → ReadOnlySpan<T2>
- string → ReadOnlySpan<char>

Simple lambda parameters with modifiers (C# 14)

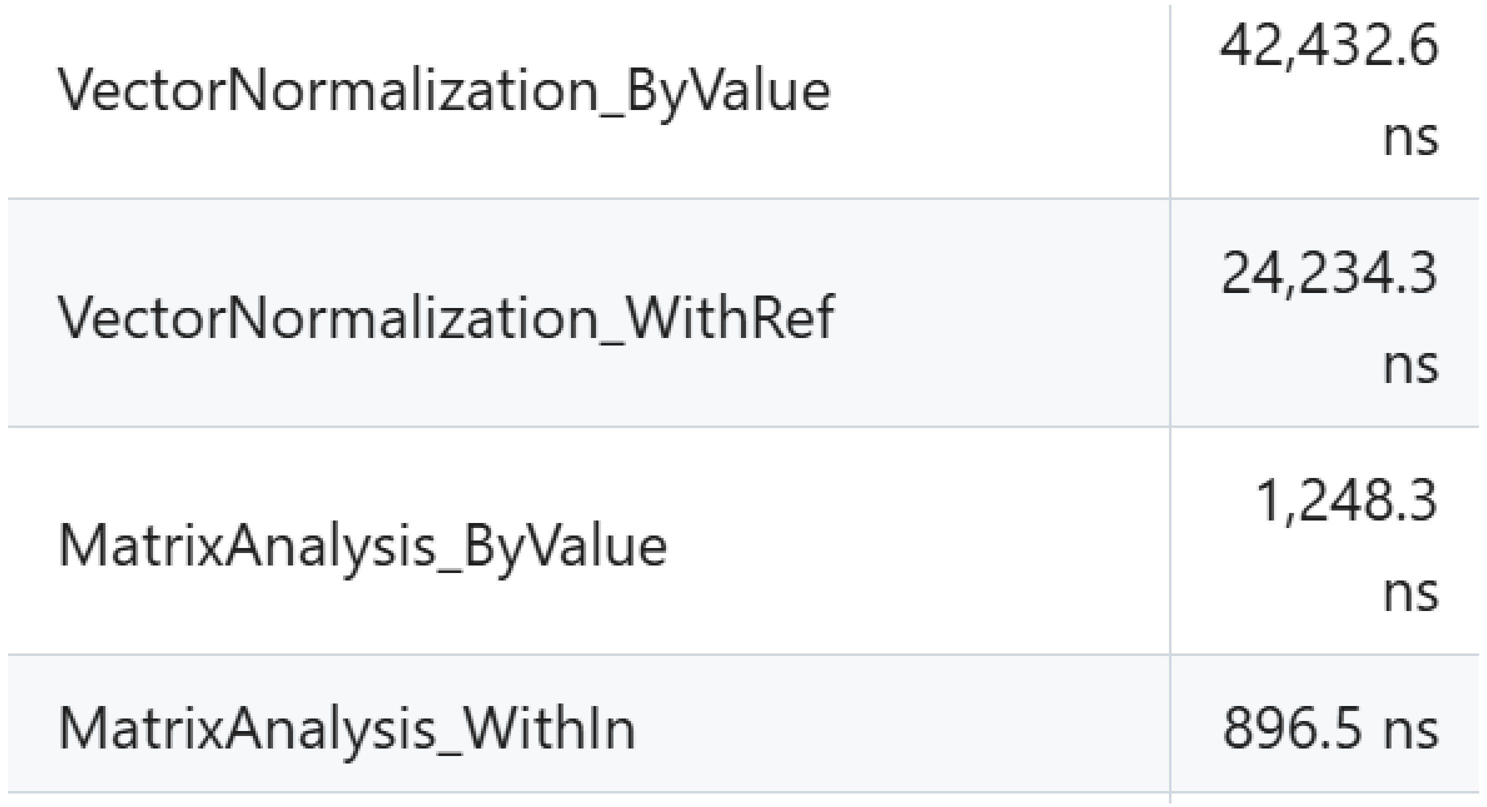
- Instead of:

```
TryParse<int> tryParseInt =
  (string text, out int result) => int.TryParse(text, out result);
```

- Do this:

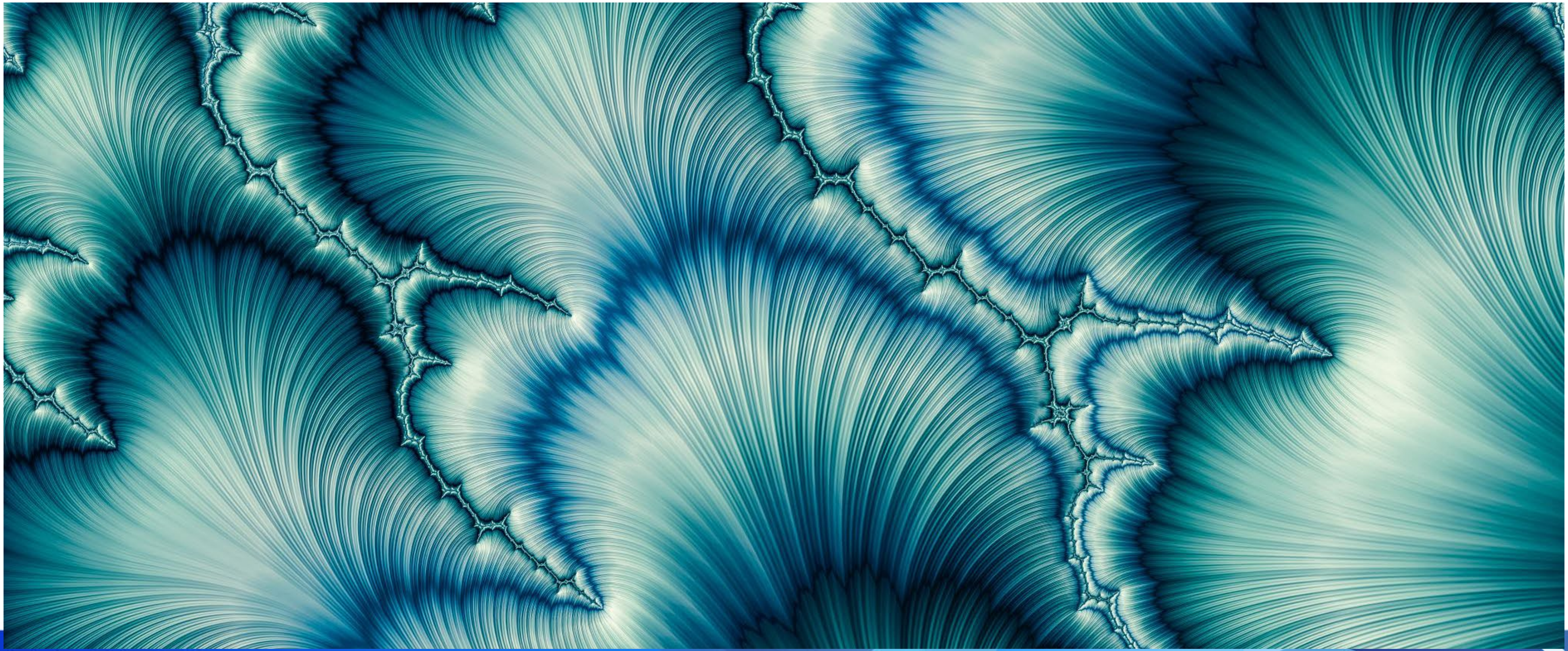
```
TryParse<int> tryParseInt =
  (text, out result) => int.TryParse(text, out result);
```

Why?



VectorNormalization_ByValue	42,432.6 ns
VectorNormalization_WithRef	24,234.3 ns
MatrixAnalysis_ByValue	1,248.3 ns
MatrixAnalysis_WithIn	896.5 ns

Async & Native



Lock Object

- .NET 9 includes *System.Threading.Lock* type
- First-class lock-type
- Simpler and faster

- The ***lock*** keyword is enhanced to not only support *Monitor*, but also *Lock*

Why is the Lock type faster?

Lower entry/exit overhead	• Lock.EnterScope is a ref struct with no heap allocation
Avoids kernel transitions	• Lock: leaner fast path for uncontented locks
Reduced instruction count	• Compiler emits direct calls to Lock.EnterScope
Stripped down design	• No conditions for recursion
Cache-friendly workload	• struct with predictable layout

Interceptors

- Replace implementation
- Usually used by source generators
- Pre-release with .NET 8
- Released with .NET 9 (with changes)
- Used from source generators
- *InterceptsLocation* Attribute
- .NET 9: Roslyn *GetInterceptableLocation*

Native AOT

- Compile .NET to native code
- Self-contained
- Quick startup, less memory usage
- Can run where JIT is not allowed
- Compilation to a single file

Native AOT Restrictions

- No dynamic loading
- No reflection emit
- No C++/CLI
- Trimming required
- Many libraries don’t support native AOT (yet)

Native AOT Updates

- Improved performance
- Trimming enhancements - smaller application size
- Enhanced platform support
- Microsoft.AspNetCore.OpenAPI
- SignalR, gRPC support, .NET MAUI
- WinUI in progress to fully support native AOT
- EF Core in progress

Native AOT For Action

- Make libraries AOT compatible
  - if possible
  - *IsAotCompatible* adds checks
- Create native AOT services
  - if useful and possible

Runtime async

- Preview with .NET 10, release with .NET 11
- After experiments with “green threads”
- Compiler rewrite
- Instead of IL code generation, runtime support
- Methods can “yield” control back to their caller
- Specific suspension points
- Improvements in performance

Performance improvements

Scenario	.NET 9 (Before)	.NET 10 w/ Runtime Async (After)	Improvement
1M `await Task.Yield()` calls	~1.25s, 35 MB allocated	~0.95s, 18 MB allocated	**~24% faster, ~49% fewer allocations**
500k `await ValueTask` completions	~0.82s, 12 MB allocated	~0.63s, 0 MB allocated	**~23% faster, allocations eliminated**
Async file I/O loop (100k ops)	~1.10s, 28 MB allocated	~0.88s, 15 MB allocated	**~20% faster, ~46% fewer allocations**

Summary

Productivity

- Natural type enhancements
- More partial members – source generator support!
- Span enhancements
- Extensions

Performance

- Span enhancements
- Native AOT
- Runtime async

Thank you for joining!

Questions?

- https://github.com/cnilearn/bastamainz2025
- https://csharp.christiannagel.com
- https://www.cninnovation.com

Wir bitten um
dein Feedback!

BITTE BEWERTEN SIE UNS!

★★★★★

entwickler.de