

# THE PERFORMANCE LOOP

A PRACTICAL GUIDE TO PROFILING AND BENCHMARKING

 danielmarbach |  daniel.marbach@particular.net |  Daniel Marbach



```
[SimpleJob]
[MemoryDiagnoser]
public class StringJoinBenchmarks {

    [Benchmark]
    public string StringJoin() {
        return string.Join(", ", Enumerable.Range(0, 10).Select(i => i.ToString()));
    }

    [Benchmark]
    public string StringBuilder() {
        var sb = new StringBuilder();
        for (int i = 0; i < 10; i++) {
            sb.Append(i);
            sb.Append(", ");
        }

        return sb.ToString(0, sb.Length - 2);
    }

    [Benchmark]
    public string ValueStringBuilder() {
        var separator = new ReadOnlySpan<char>(new char[] { ',', ',' });
        using var sb = new ValueStringBuilder(stackalloc char[30]);
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i);
            sb.Append(separator);
        }

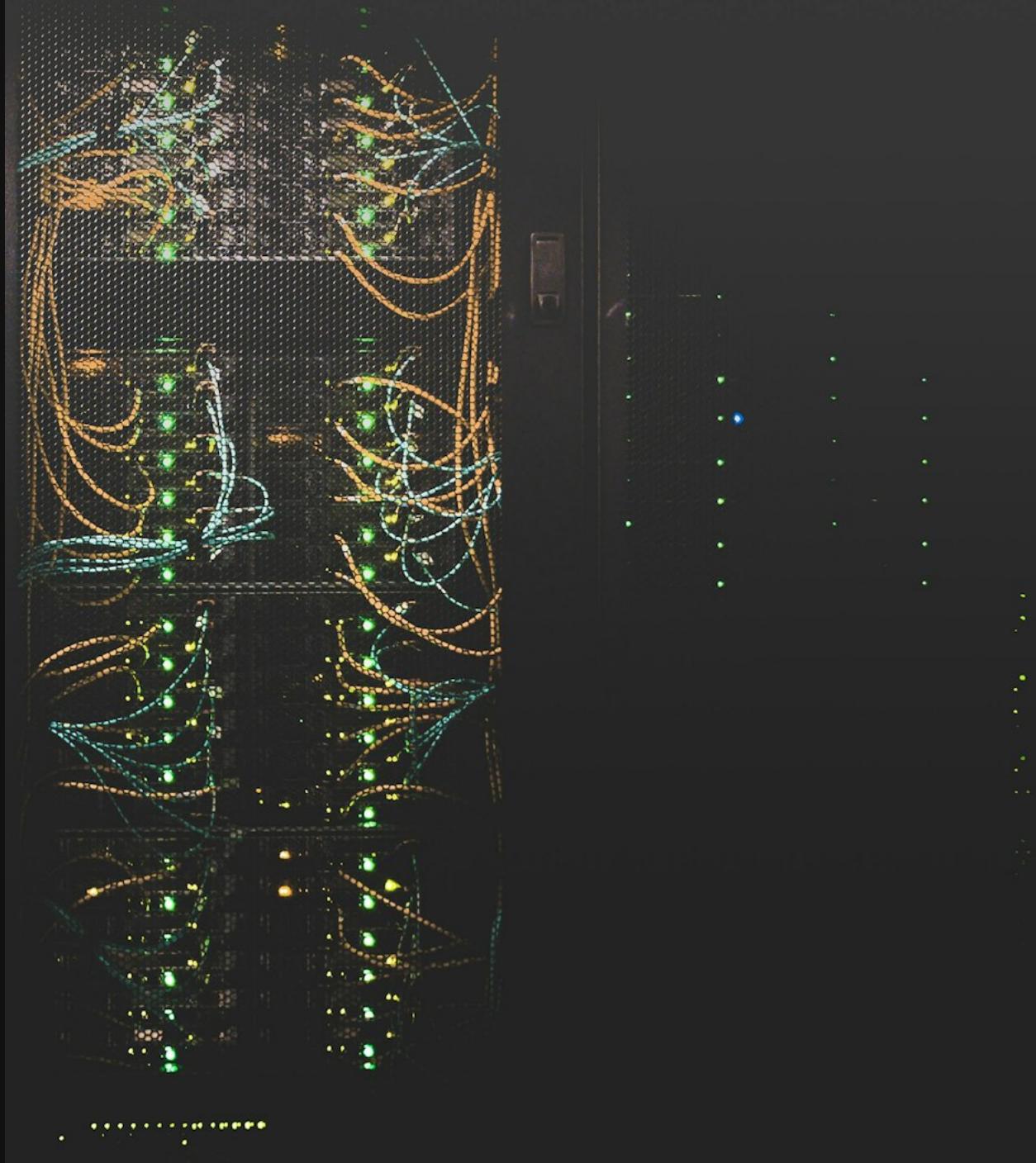
        return sb.AsSpan(0, sb.Length - 2).ToString();
    }
}
```

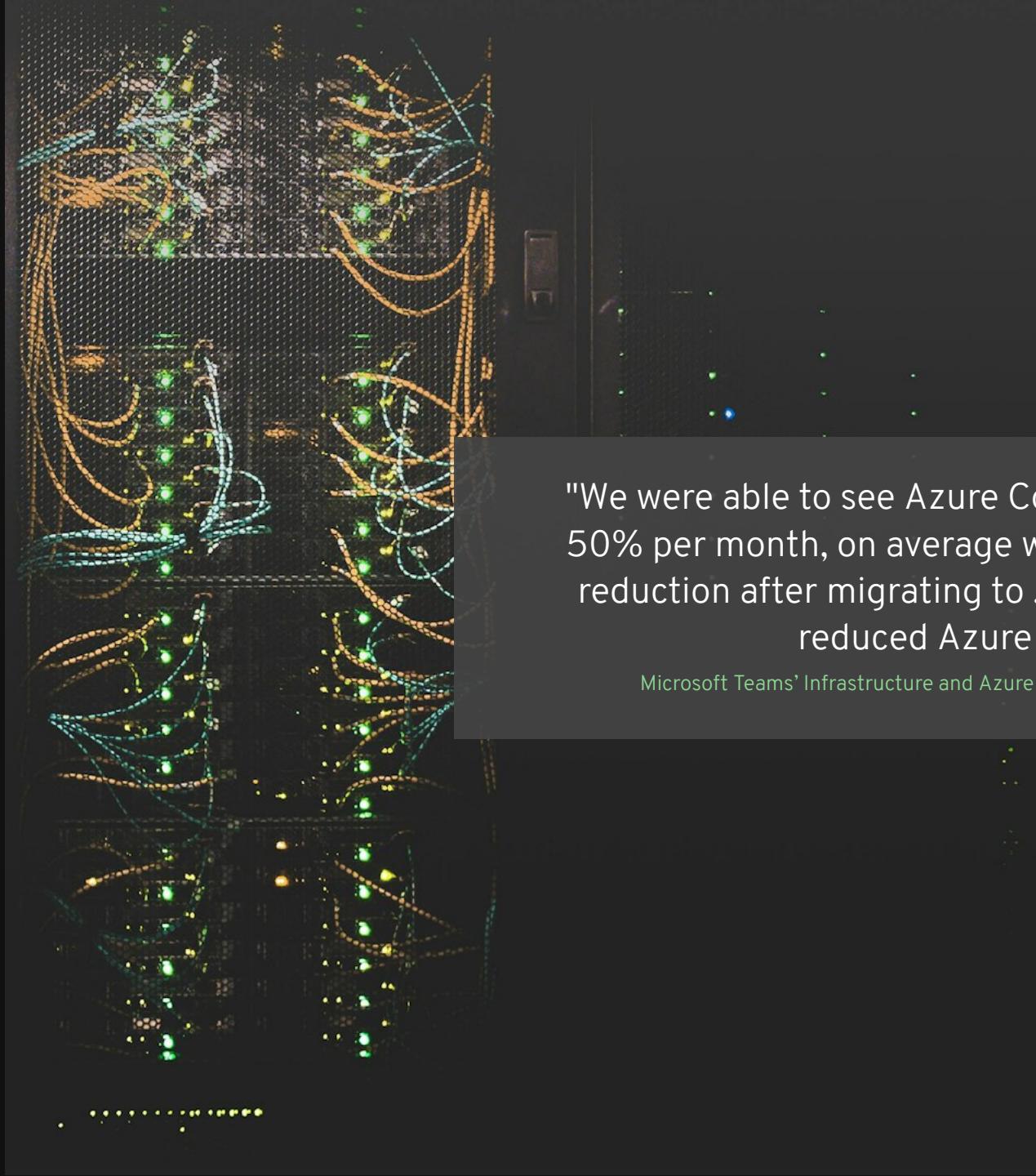
# "SIMPLE"



Wrong!

$$2 + 2 = 5$$





"We were able to see Azure Compute cost reduction of up to 50% per month, on average we observed 24% monthly cost reduction after migrating to .NET 6. The reduction in cores reduced Azure spend by 24%."

Microsoft Teams' Infrastructure and Azure Communication Services' Journey to .NET 6



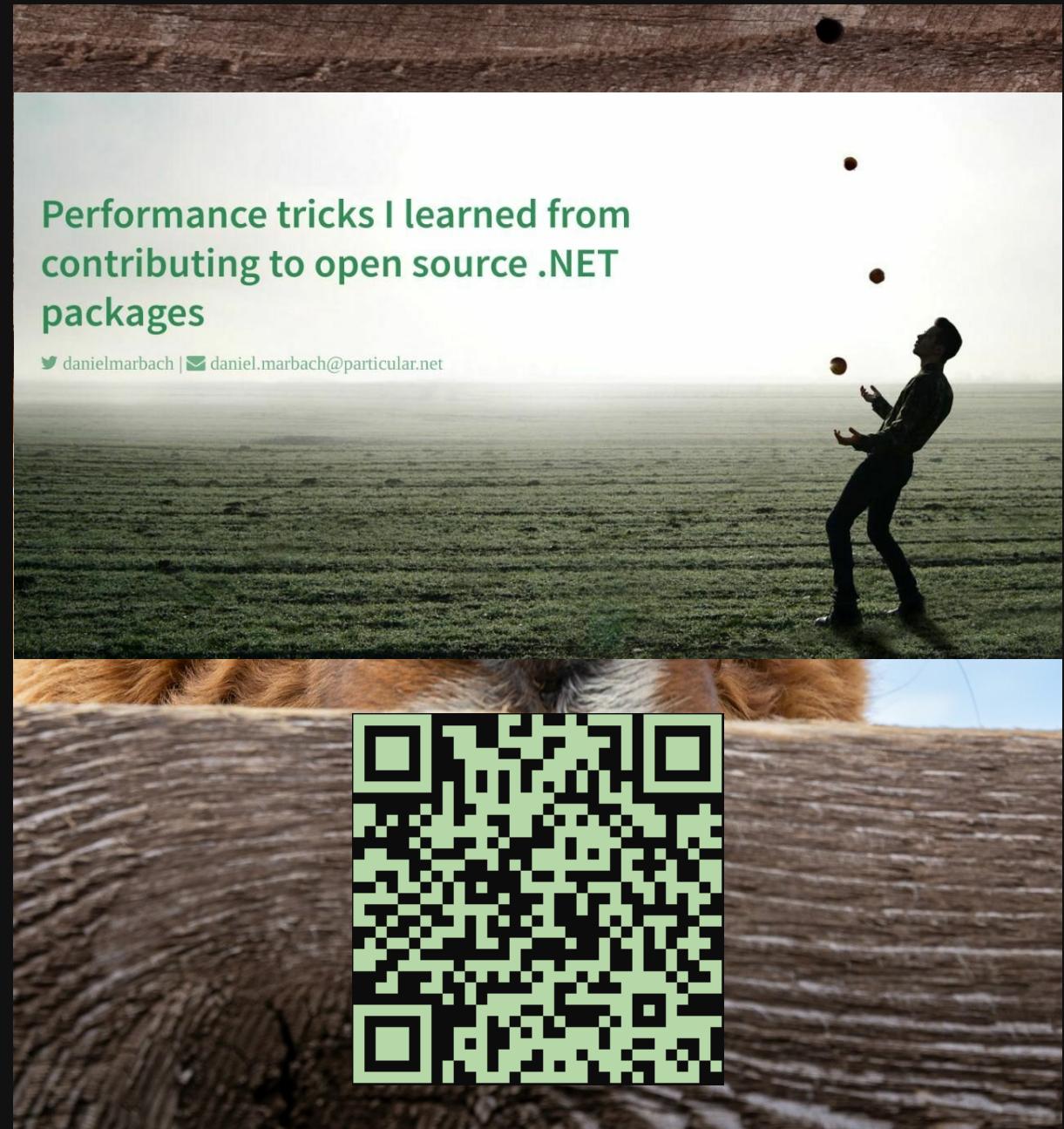
# PERFORMANCE AWARE



BEAR AWARE

# BE CURIOUS.... UNDERSTAND THE CONTEXT

- How is this code going to be executed at scale, and what would the memory characteristics be (gut feeling)
- Are there simple low-hanging fruits I can apply to accelerate this code?
- Are there things I can move away from the hot path by simply restructuring a bit my code?
- What part is under my control and what isn't really?
- What optimizations can I apply, and when should I stop?



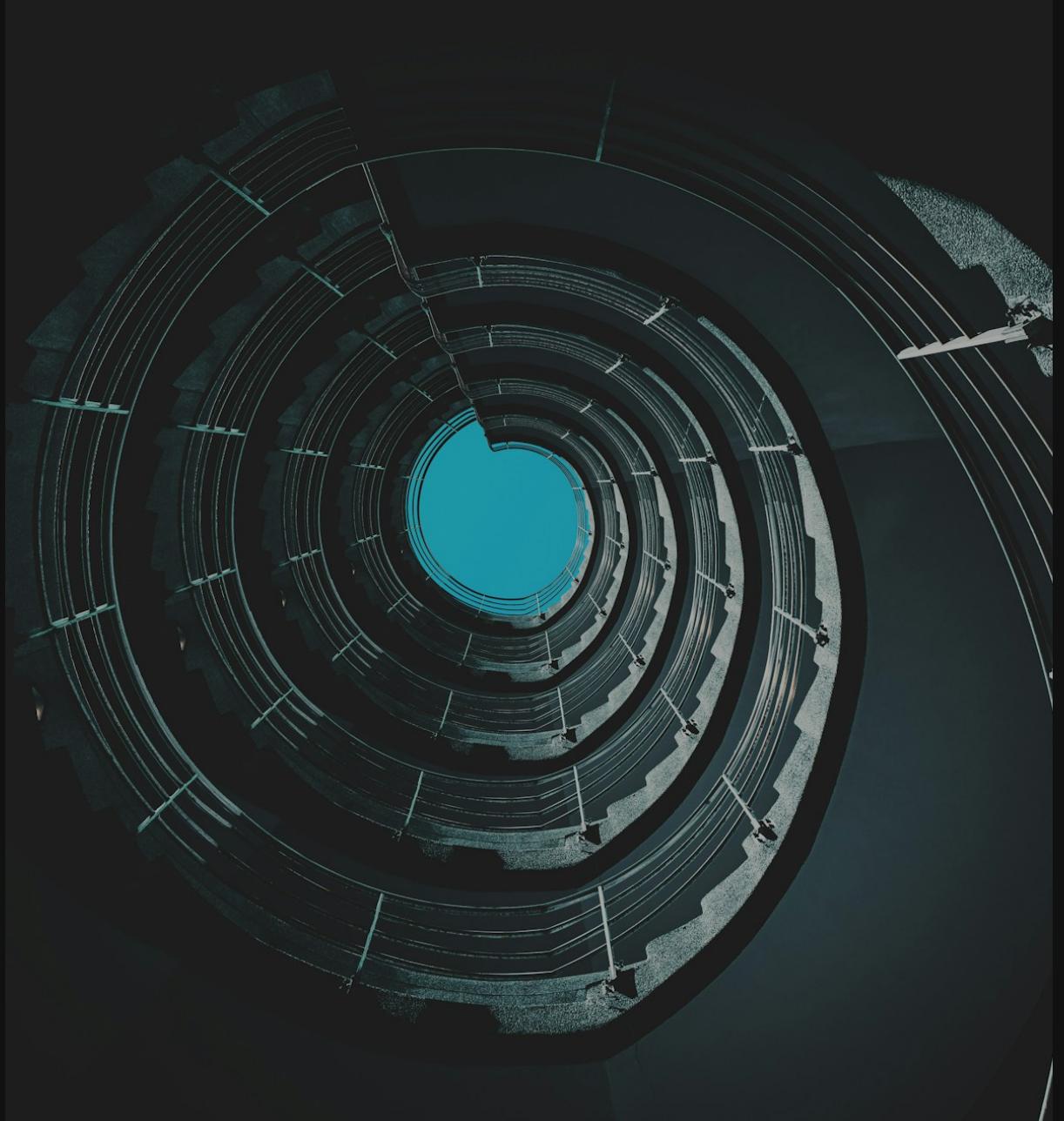
# THE PERFORMANCE LOOP

- Profile at least CPU and memory using a profiling harness
- Improve parts of the hot path

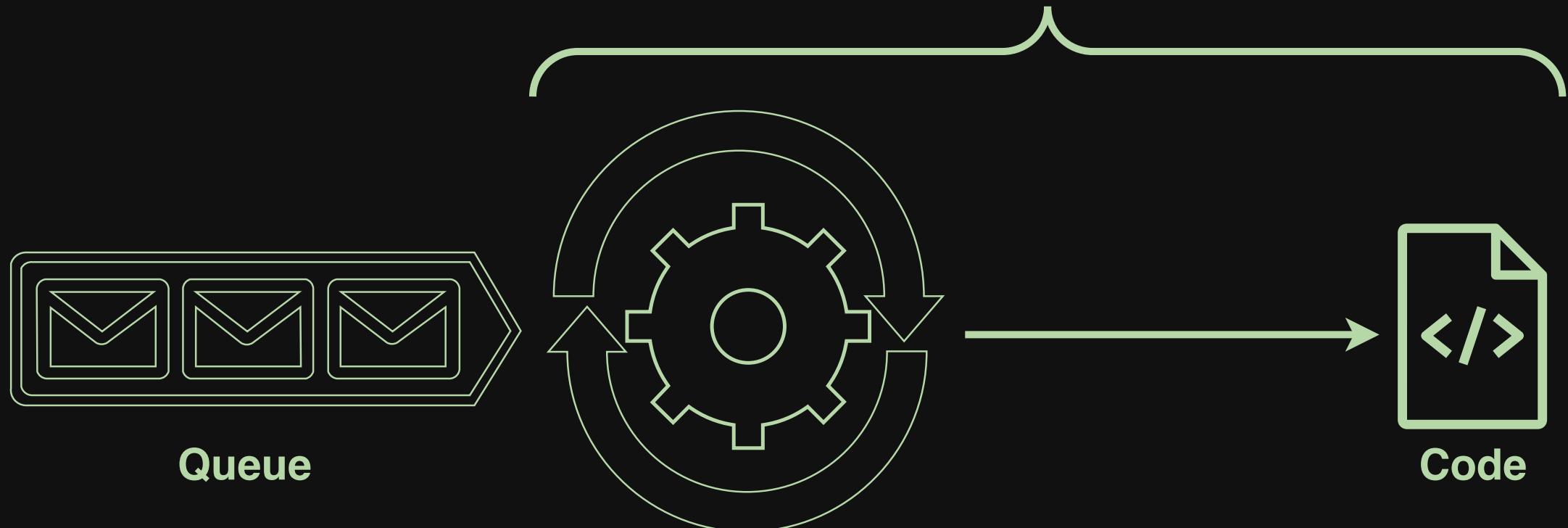


# THE PERFORMANCE LOOP

- Profile at least CPU and memory using a profiling harness
- Improve parts of the hot path
- Benchmark and compare
- Profile improvements again with the harness and make adjustments where necessary
- Ship and focus your attention to other parts

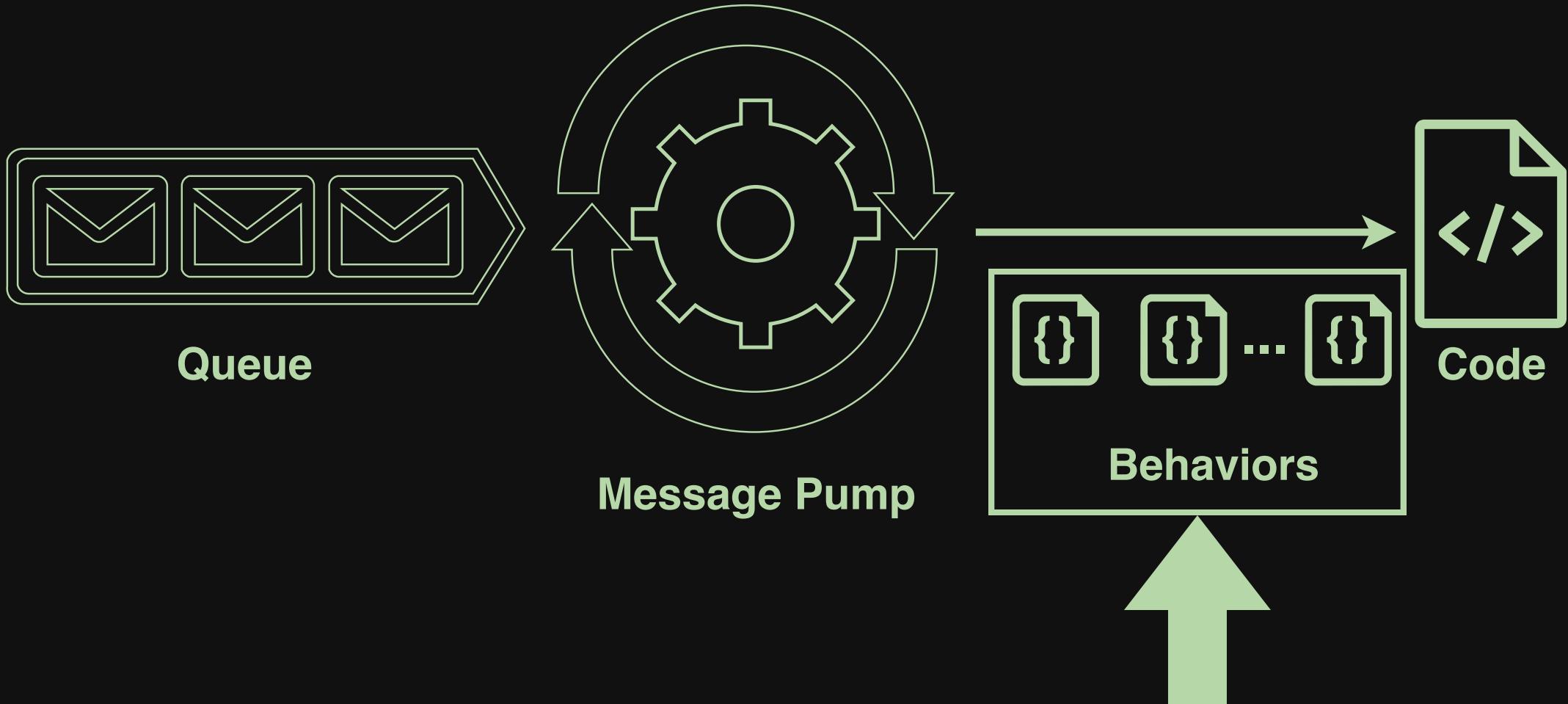


# NSERVICEBUS



[go.particular.net/ndc-oslo-2024-quickstart](http://go.particular.net/ndc-oslo-2024-quickstart)

# NSERVICEBUS PIPELINE



# ASP.NET CORE MIDDLEWARE

```
  ● ● ●  
1 public class RequestCultureMiddleware {  
2     private readonly RequestDelegate _next;  
3  
4     public RequestCultureMiddleware(RequestDelegate next) {  
5         _next = next;  
6     }  
7  
8     public async Task InvokeAsync(HttpContext context) {  
9         // Do work that does something before  
10        await _next(context);  
11        // Do work that does something after  
12    }  
13 }
```

# BEHAVIORS

```
  ● ● ●  
1 public class Behavior : Behavior<IIncomingLogicalMessageContext> {  
2     public override Task  
3         Invoke(IIncomingLogicalMessageContext context, Func<Task> next) {  
4             // Do work that does something before  
5             await next();  
6             // Do work that does something after  
7         }  
8     }
```

# PROFILING THE PIPELINE



# THE HARNESS

•••

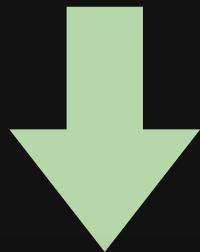
```
1 var endpointConfiguration = new EndpointConfiguration("Harness");
2 endpointConfiguration.UseSerialization<JsonSerializer>();
3 var transport = endpointConfiguration.UseTransport<MsmqTransport>();
4 endpointConfiguration.UsePersistence<InMemoryPersistence>();
5
6 var endpointInstance = await Endpoint.Start(endpointConfiguration);
7
8 Console.WriteLine("Attach the profiler and hit <enter>.");
9 Console.ReadLine();
10
11 var tasks = new List<Task>(1000);
12 for (int i = 0; i < 1000; i++)
13 {
14     tasks.Add(endpointInstance.Publish(new MyEvent()));
15 }
16 await Task.WhenAll(tasks);
17
18 Console.WriteLine("Publish 1000 done. Get a snapshot");
19 Console.ReadLine();
```

Publish Pipeline



Profiling the pipeline > Improving > Benchmarking > Profiling

# THE HARNESS



Receive Pipeline

```
...  
  
public class MyEventHandler : IHandleMessages<MyEvent> {  
    public Task Handle(MyEvent message, IMessageHandlerContext context)  
    {  
        return Task.CompletedTask;  
    }  
}
```



Profiling the pipeline > Improving > Benchmarking > Profiling

# THE HARNESS

- Compiled and executed in Release mode
- Runs a few seconds and keeps overhead minimal
- Disabled Tiered JIT  
`<TieredCompilation>false</TieredCompilation>`
- Emits full symbols  
`<DebugType>pdbonly</DebugType>`  
`<DebugSymbols>true</DebugSymbols>`



```
var endpointConfiguration = new EndpointConfiguration("Harness");
endpointConfiguration.UseSerialization<JsonSerializer>();
var transport = endpointConfiguration.UseTransport<MsmqTransport>();
endpointConfiguration.UsePersistence<InMemoryPersistence>();

var endpointInstance = await Endpoint.Start(endpointConfiguration);

Console.WriteLine("Attach the profiler and hit <enter>.");
Console.ReadLine();

var tasks = new List<Task>(1000);
for (int i = 0; i < 1000; i++)
{
    tasks.Add(endpointInstance.Publish(new MyEvent()));
}
await Task.WhenAll(tasks);

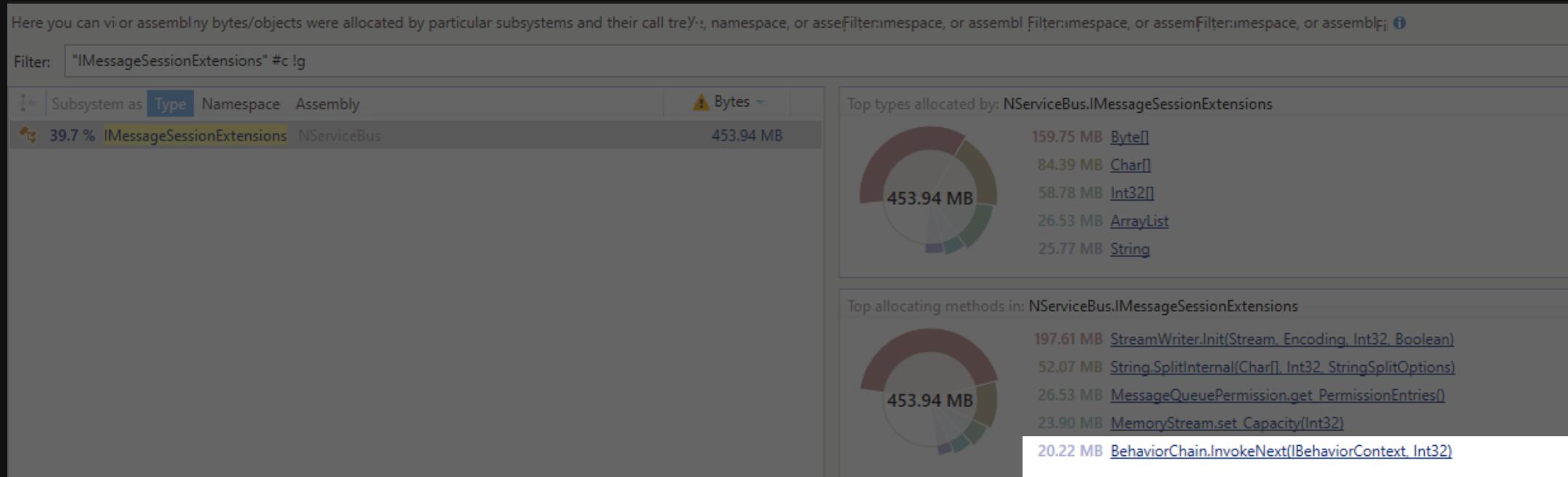
Console.WriteLine("Publish 1000 done. Get a snapshot");
Console.ReadLine();
```



```
public class MyEventHandler : IHandleMessages<MyEvent> {
    public Task Handle(MyEvent message, IMessageHandlerContext context)
    {
        return Task.CompletedTask;
    }
}
```



# MEMORY CHARACTERISTICS

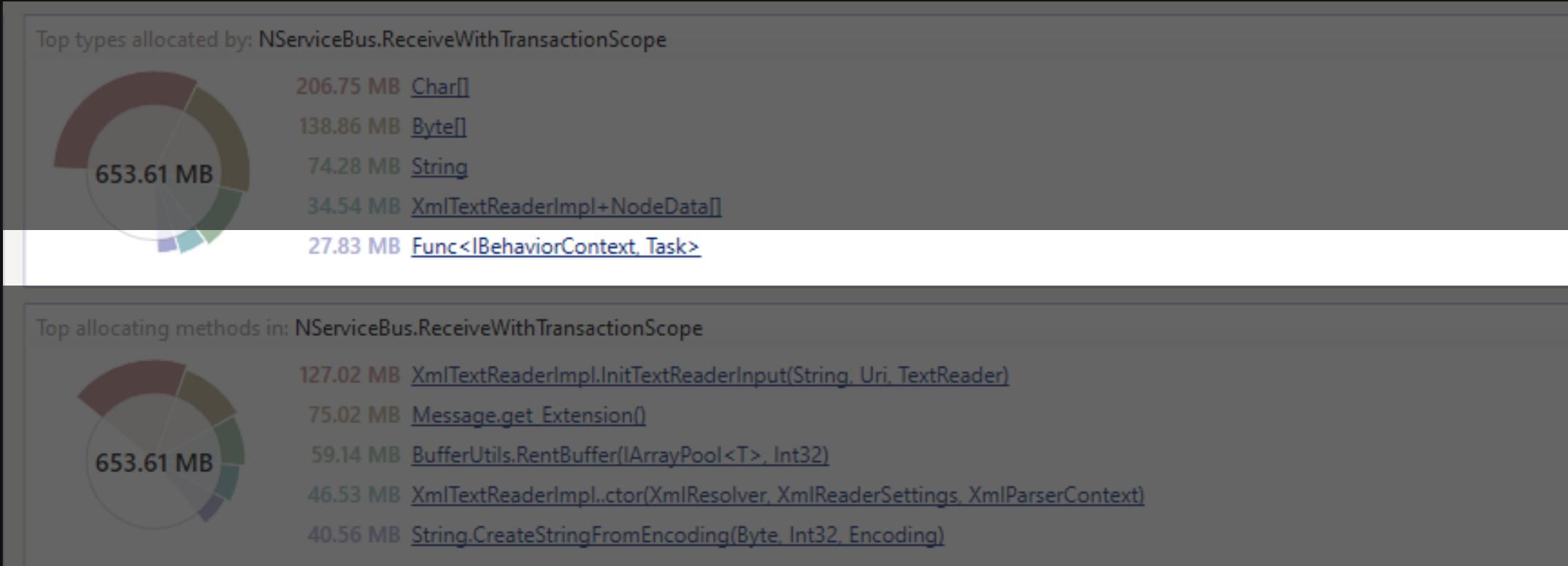


PUBLISH



Profiling the pipeline > Improving > Benchmarking > Profiling

# MEMORY CHARACTERISTICS



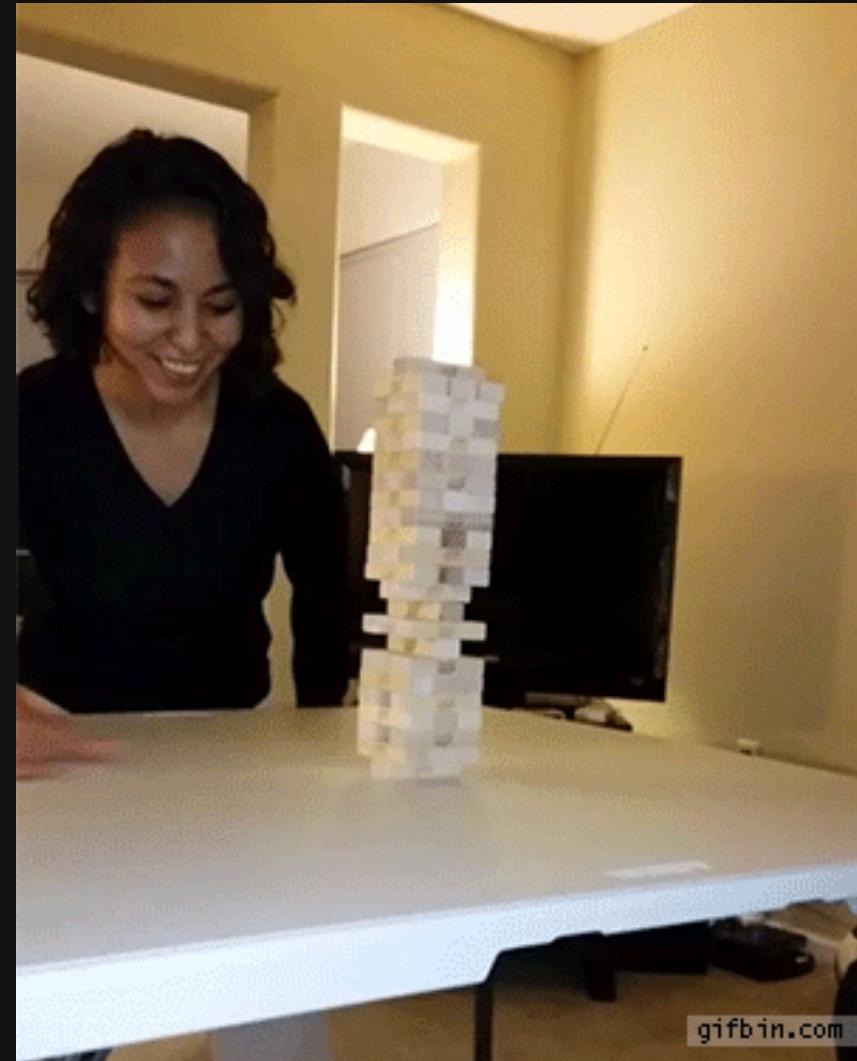
## RECEIVE



# MEMORY CHARACTERISTICS

- MSMQ has a diminishing user-base
- Ramping up knowledge may not be feasible
- Iterative gains on the hot path will lead to overall improvements
- Pipeline optimizations benefits all users

CONTEXT MATTERS  
YOU ARE THE EXPERT



# MEMORY CHARACTERISTICS

Top types allocated by: Behavior<TContext>.Invoke(...)



15.15 MB [Func<Task>](#)

225.3 KB [Behavior+ <>c DisplayClass0 0<|OutgoingLogicalMessageContext>](#)

110.3 KB [Behavior+ <>c DisplayClass0 0<|OutgoingPublishContext>](#)

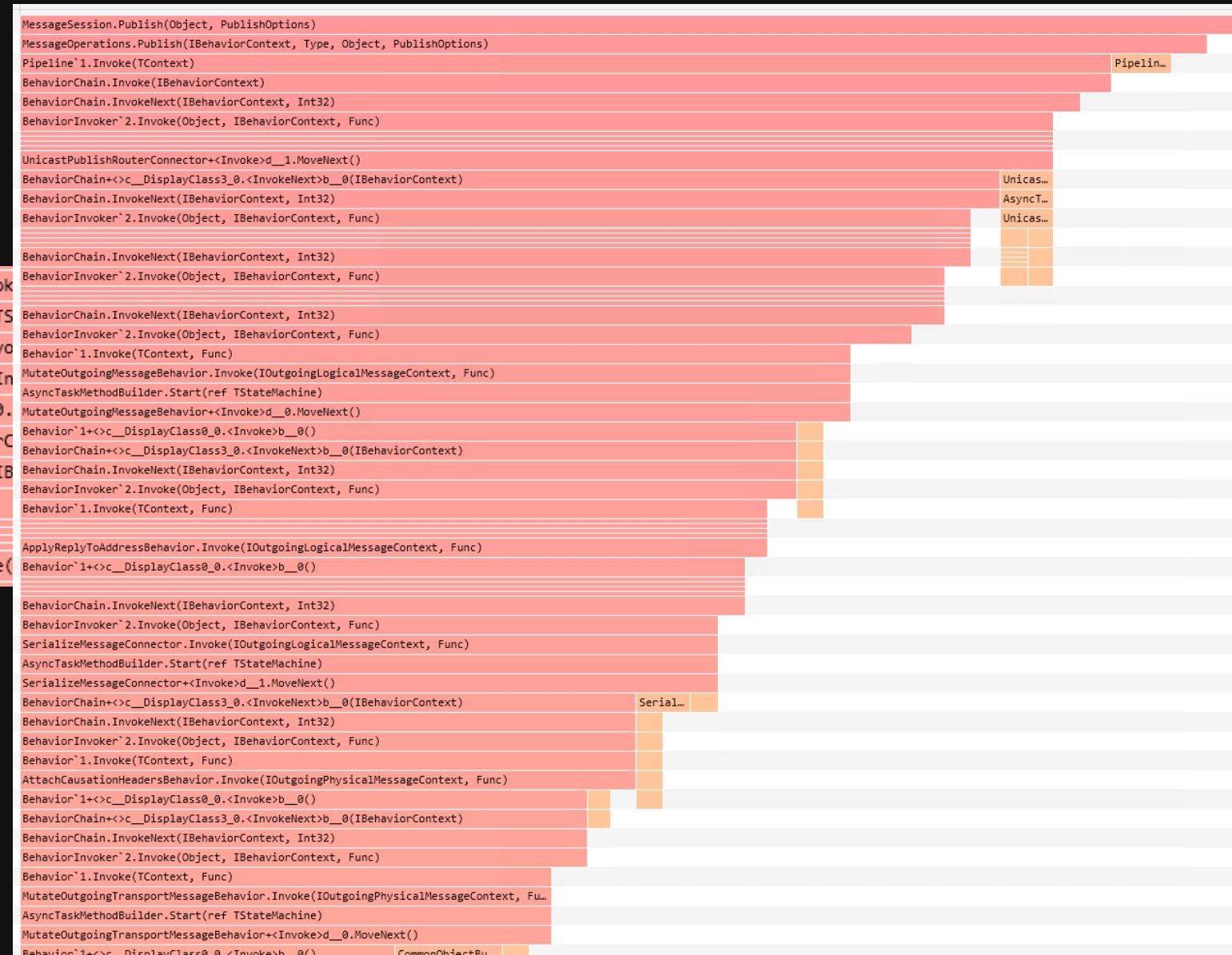


# MEMORY CHARACTERISTICS

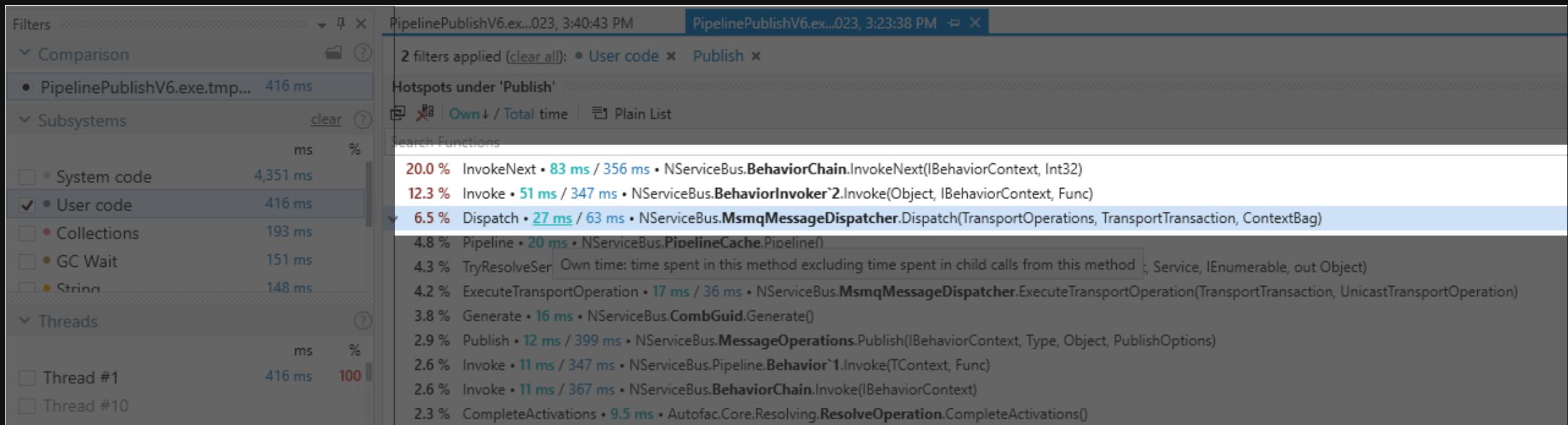
Type	Bytes
StageForkConnector+<>c_DisplayClass0_0<ITransportReceiveContext, IIncomingPhysicalMessageContext>	15.74 MB
Behavior+<>c_DisplayClass0_0<IOutgoingLogicalMessageContext>	NServiceBus.Pipeline
Behavior+<>c_DisplayClass0_0<IRoutingContext>	NServiceBus.Pipeline
Behavior+<>c_DisplayClass0_0<IOutgoingPhysicalMessageContext>	NServiceBus.Pipeline

# CPU CHARACTERISTICS

# CPU CHARACTERISTICS



# CPU CHARACTERISTICS

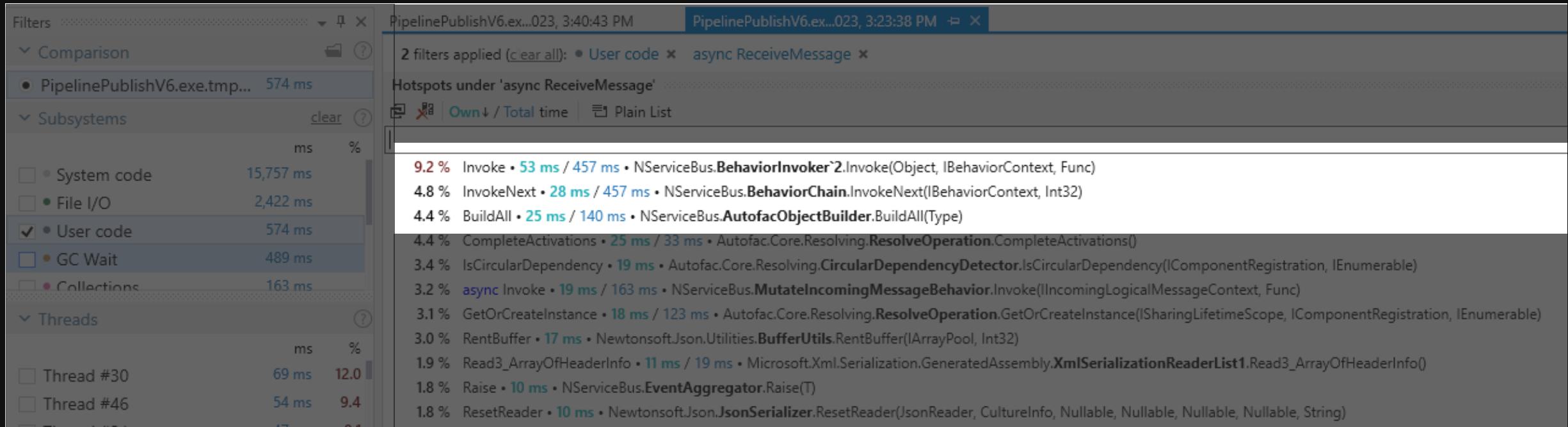


PUBLISH



Profiling the pipeline > Improving > Benchmarking > Profiling

# CPU CHARACTERISTICS



## RECEIVE



Profiling the pipeline > Improving > Benchmarking > Profiling

# IMPROVING

```
✓ NServiceBus.Core.Tests (3 tests) Success
  > ? RedirectHelper (net472) (1 test)
  ✓ RedirectHelper (net7.0) (2 tests) Success
    ✓ () NServiceBus.Core.Tests (2 tests) Success
      ✓ GlobalTestSetup (2 tests) Success
      ✓ () Pipeline (2 tests) Success
        ✓ PipelineTests (2 tests) Success
          ✓ ShouldExecutePipeline Success
          ✓ ShouldNotCacheContext Success
```

# IMPROVING

💡 10X faster execution with compiled expression trees

💡 How we achieved 5X faster pipeline execution by removing closure allocations

go.particular.net/ndc-oslo-2024-pipeline

Profiling >  Improving > Benchmarking > Profiling

DOCUMENTATION BLOG DISTRIBUTED SYSTEMS DESIGN FUNDAMENTALS

Particular Software

Platform Solutions Pricing Support Resources Community Company GET STARTED

How we achieved 5X faster pipeline execution by removing closure allocations

Written by Daniel Marbach on September 27, 2022



The NServiceBus messaging pipeline strives to achieve the right balance of flexibility, maintainability, and wicked fast...umm...ability. It needs to be wicked fast because it is executed at scale. For our purposes, "at scale" means that throughout the lifetime of an NServiceBus endpoint, the message pipeline will be executed hundreds, even thousands of times per second under high load scenarios.

Previously, we were able to achieve 10X faster pipeline execution and a 94% reduction in Gen 0 garbage creation by building expression trees at startup and then dynamically compiling them. One of the key learnings of those expression tree adventures is that reducing Gen 0 allocation makes a big difference. The less Gen 0 allocation used, the more speed can be squeezed out of the message handling pipeline, which ultimately means more speed for our users.

BLOG HOME RSS FEED

Webinar: A fireside chat with live Q&A with Udi Dahan An open discussion on managing complexity using orchestration and choreography LEARN MORE →

NServiceBus Quick Start Learn why software systems built on asynchronous messaging using NServiceBus are superior to traditional synchronous HTTP-based web services. CHECK IT OUT →

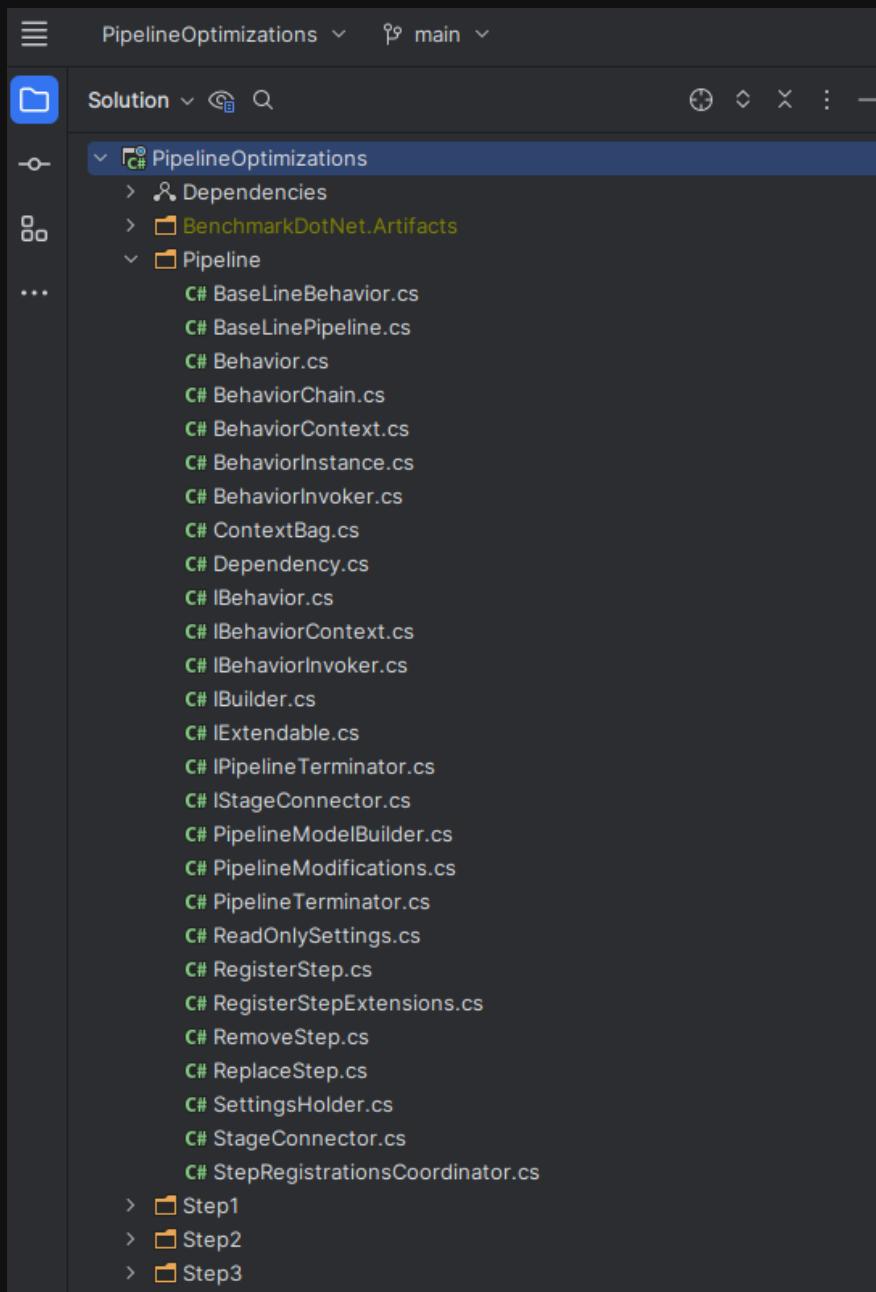
# BENCHMARKING THE PIPELINE



*A disgusting, festering mess.*

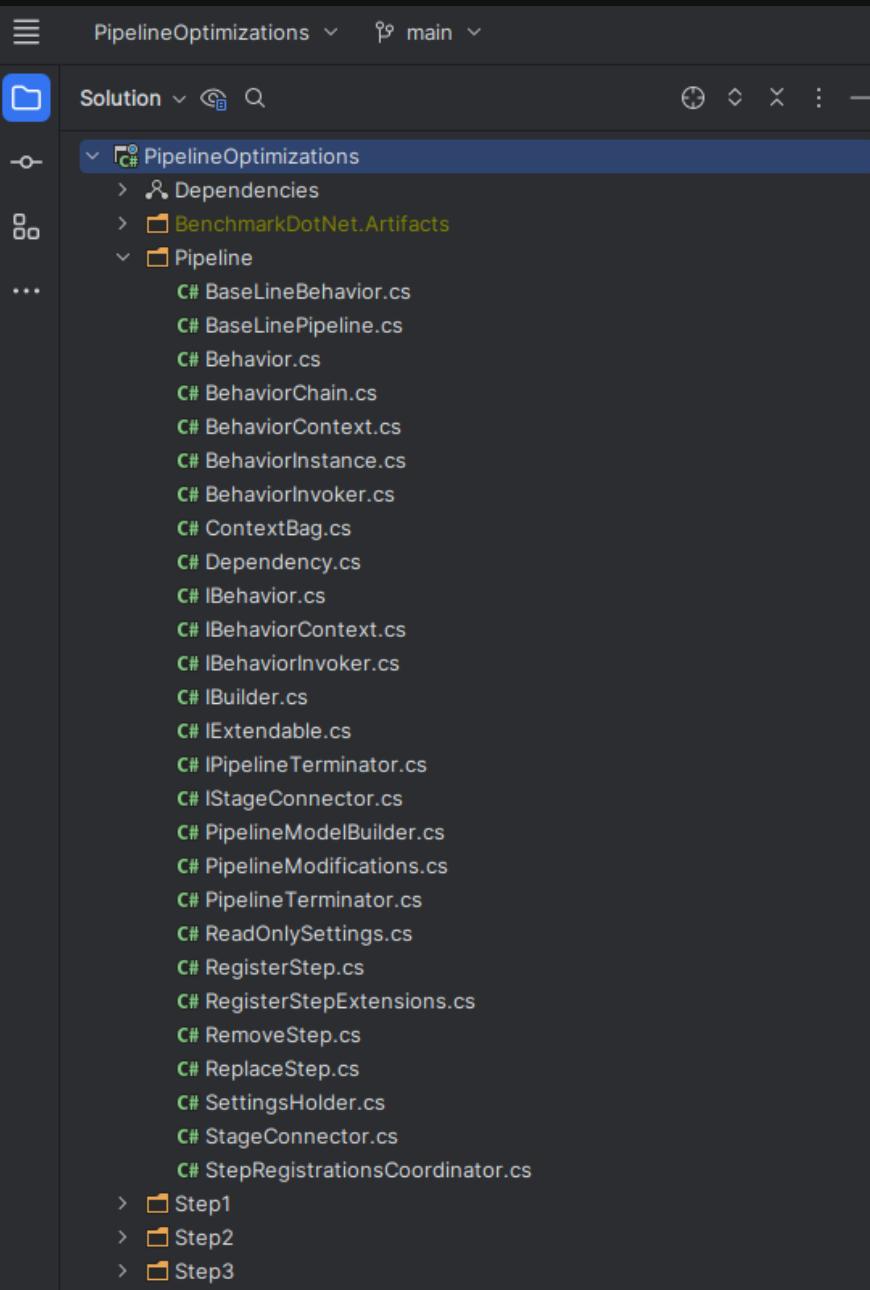
# EXTRACT CODE

- Copy and paste relevant code
- Adjust it to the bare essentials to create a controllable environment



# EXTRACT CODE

- Trim down to relevant behaviors
- Replaced dependency injection container with creating relevant classes
- Replaced IO-operations with completed tasks



# PERFORMANCE CULTURE

- Get started with small steps
- Culture change takes time
- Make changes gradually



# UNLIKE A UNIT TEST

- Measures a distribution of values

● ● ●

Method	Mean	Error	StdDev	Ratio	Param
Before	51.57 us	0.311 us	0.291 us	1.00	Value
After	21.91 us	0.138 us	0.129 us	0.42	Value

- Executed until results are stable potentially hundreds or thousands of times
- Takes minutes or hours
- Focuses on most common cases of frequently used code (hot path) with the required amount of permutations
- Cases should be derived from production usage

● ● ●

```
1 // Special nobs and dials
2 [Job]
3 [XYZDiagnoser]
4 public class Benchmark {
5
6     // Permutations that influence your scenario
7
8     [Params(...)]
9     public int Parameter1 { get; set; }
10
11    [Params(...)]
12    public int Parameter2 { get; set; }
13
14
15    [GlobalSetup]
16    public void SetUp() {
17        // Stuff that you don't want to measure
18    }
19
20    [Benchmark(Baseline = true)]
21    public void Before() {
22        // Your code before the changes
23    }
24
25    [Benchmark]
26    public void After() {
27        // Your code after the changes
28    }
29 }
```

```
● ● ●
1 [ShortRunJob]
2 [MemoryDiagnoser]
3 public class PipelineExecution {
4
5     [Params(10, 20, 40)]
6     public int PipelineDepth { get; set; }
7
8
9     [GlobalSetup]
10    public void SetUp() {
11        behaviorContext = new BehaviorContext();
12
13        pipelineModificationsBeforeOptimizations = new PipelineModifications();
14        for (int i = 0; i < PipelineDepth; i++)
15        {
16            pipelineModificationsBeforeOptimizations.Additions.Add(RegisterStep.Create(i.ToString(),
17                typeof(BaseLineBehavior), i.ToString(), b => new BaseLineBehavior()));
18        }
19
20        pipelineModificationsAfterOptimizations = new PipelineModifications();
21        for (int i = 0; i < PipelineDepth; i++)
22        {
23            pipelineModificationsAfterOptimizations.Additions.Add(RegisterStep.Create(i.ToString(),
24                typeof(BehaviorOptimization), i.ToString(), b => new BehaviorOptimization()));
25        }
26
27        pipelineBeforeOptimizations = new BaseLinePipeline<IBehaviorContext>(null, new SettingsHolder(),
28            pipelineModificationsBeforeOptimizations);
29        pipelineAfterOptimizations = new PipelineOptimization<IBehaviorContext>(null, new SettingsHolder(),
30            pipelineModificationsAfterOptimizations);
31    }
32
33    [Benchmark(Baseline = true)]
34    public async Task Before() {
35        await pipelineBeforeOptimizations.Invoke(behaviorContext);
36    }
37
38    [Benchmark]
39    public async Task After() {
40        await pipelineAfterOptimizations.Invoke(behaviorContext);
41    }
42 }
```

```
1 [ShortRunJob]
2 [MemoryDiagnoser]
3 public class PipelineExecution {
4
5     [Params(10, 20, 40)]
6     public int PipelineDepth { get; set; }
7
8
9     [GlobalSetup]
10    public void SetUp() {
11        behaviorContext = new BehaviorContext();
12
13        pipelineModificationsBeforeOptimizations = new PipelineModifications();
14        for (int i = 0; i < PipelineDepth; i++)
15        {
16            pipelineModificationsBeforeOptimizations.Additions.Add(RegisterStep.Create(i.ToString(),
17                typeof(BaseLineBehavior), i.ToString(), b => new BaseLineBehavior()));
18        }
19
20        pipelineModificationsAfterOptimizations = new PipelineModifications();
21        for (int i = 0; i < PipelineDepth; i++)
22        {
23            pipelineModificationsAfterOptimizations.Additions.Add(RegisterStep.Create(i.ToString(),
24                typeof(BehaviorOptimization), i.ToString(), b => new BehaviorOptimization()));
25        }
26
27        pipelineBeforeOptimizations = new BaseLinePipeline<IBehaviorContext>(null, new SettingsHolder(),
28            pipelineModificationsBeforeOptimizations);
29        pipelineAfterOptimizations = new PipelineOptimization<IBehaviorContext>(null, new SettingsHolder(),
30            pipelineModificationsAfterOptimizations);
31    }
32
33    [Benchmark(Baseline = true)]
34    public async Task Before() {
35        await pipelineBeforeOptimizations.Invoke(behaviorContext);
36    }
37
38    [Benchmark]
39    public async Task After() {
40        await pipelineAfterOptimizations.Invoke(behaviorContext);
41    }
42 }
```

```
● ● ●
1 [ShortRunJob]
2 [MemoryDiagnoser]
3 public class PipelineExecution {
4
5     [Params(10, 20, 40)]
6     public int PipelineDepth { get; set; }
7
8
9     [GlobalSetup]
10    public void SetUp() {
11        behaviorContext = new BehaviorContext();
12
13        pipelineModificationsBeforeOptimizations = new PipelineModifications();
14        for (int i = 0; i < PipelineDepth; i++)
15        {
16            pipelineModificationsBeforeOptimizations.Additions.Add(RegisterStep.Create(i.ToString(),
17                typeof(BaseLineBehavior), i.ToString(), b => new BaseLineBehavior()));
18        }
19
20        pipelineModificationsAfterOptimizations = new PipelineModifications();
21        for (int i = 0; i < PipelineDepth; i++)
22        {
23            pipelineModificationsAfterOptimizations.Additions.Add(RegisterStep.Create(i.ToString(),
24                typeof(BehaviorOptimization), i.ToString(), b => new BehaviorOptimization()));
25        }
26
27        pipelineBeforeOptimizations = new BaseLinePipeline<IBehaviorContext>(null, new SettingsHolder(),
28            pipelineModificationsBeforeOptimizations);
29        pipelineAfterOptimizations = new PipelineOptimization<IBehaviorContext>(null, new SettingsHolder(),
30            pipelineModificationsAfterOptimizations);
31    }
32
33    [Benchmark(Baseline = true)]
34    public async Task Before() {
35        await pipelineBeforeOptimizations.Invoke(behaviorContext);
36    }
37
38    [Benchmark]
39    public async Task After() {
40        await pipelineAfterOptimizations.Invoke(behaviorContext);
41    }
42 }
```

# PRACTICES

- Single Responsibility Principle
- No side effects
- Prevents dead code elimination
- Delegates heavy lifting to the framework
- Is explicit
  - No implicit casting
  - Explicit types were necessary
- Avoid running any other resource-heavy processes while benchmarking





# BenchmarkDotNet

Powerful .NET library for benchmarking

**Benchmarking is really hard**

BenchmarkDotNet will protect you from the common pitfalls  
because it does all the dirty work for you

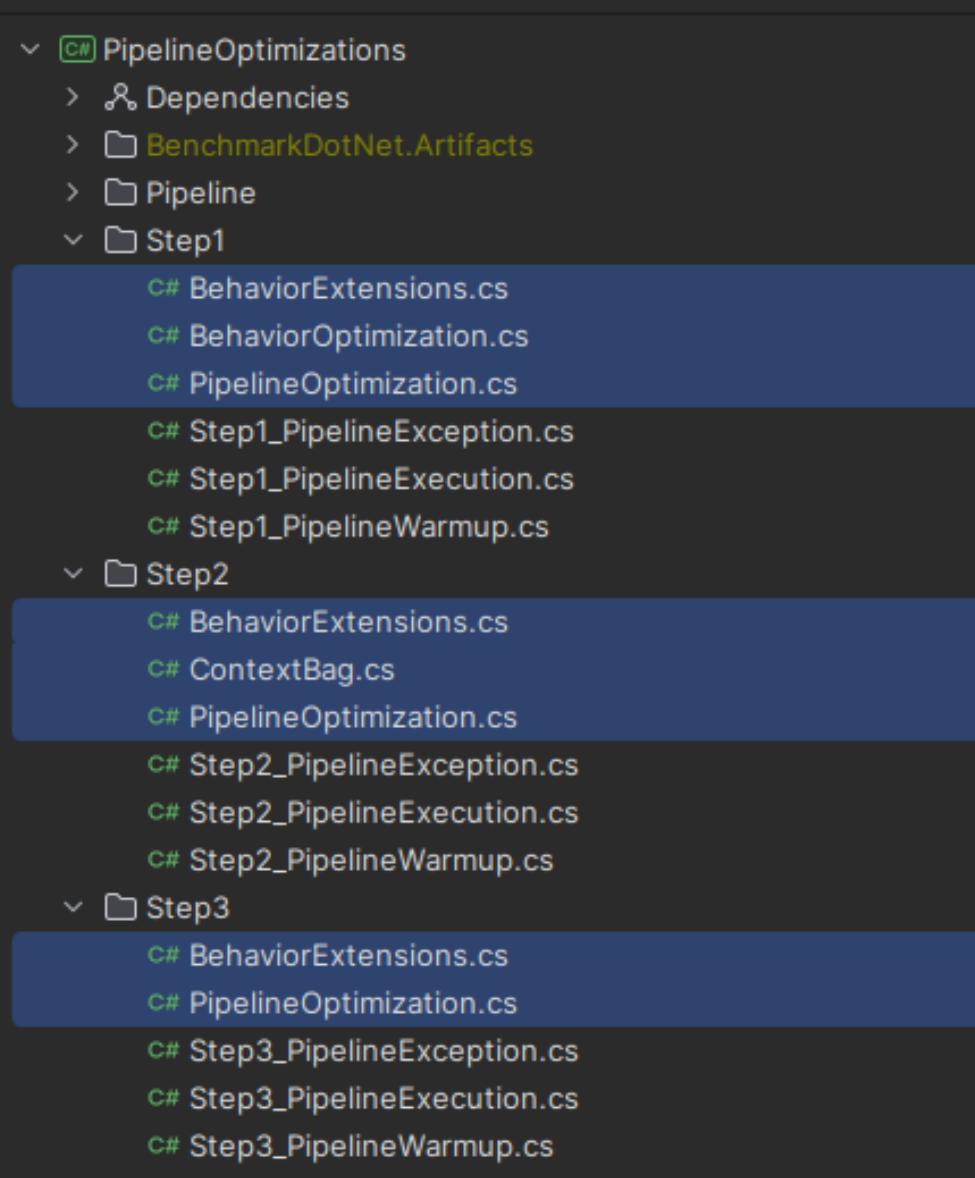
```
● ● ●
```

```
1 [ShortRunJob]
2 [MemoryDiagnoser]
3 public class Step2_PipelineException {
4     [GlobalSetup]
5     public void SetUp() {
6         ...
7         var stepId = PipelineDepth + 1;
8         pipelineModificationsBeforeOptimizations.Additions.Add(RegisterStep.Create(stepId.ToString(), typeof(Throwing), "1", b
=> new Throwing()));
9
10        ...
11        pipelineModificationsAfterOptimizations.Additions.Add(RegisterStep.Create(stepId.ToString(), typeof(Throwing), "1", b
=> new Throwing()));
12
13        pipelineBeforeOptimizations = new Step1.PipelineOptimization<IBehaviorContext>(null, new SettingsHolder(),
14            pipelineModificationsBeforeOptimizations);
15        pipelineAfterOptimizations = new PipelineOptimization<IBehaviorContext>(null, new SettingsHolder(),
16            pipelineModificationsAfterOptimizations);
17    }
18
19    [Benchmark(Baseline = true)]
20    public async Task Before() {
21        try
22        {
23            await pipelineBeforeOptimizations.Invoke(context).ConfigureAwait(false);
24        }
25        catch (InvalidOperationException)
26        {
27        }
28    }
29
30    [Benchmark]
31    public async Task After() {
32        try
33        {
34            await pipelineAfterOptimizations.Invoke(context).ConfigureAwait(false);
35        }
36        catch (InvalidOperationException)
37        {
38        }
39    }
40
41    class Throwing : Behavior<IBehaviorContext> {
42        public override Task Invoke(IBehaviorContext context, Func<Task> next)
43        {
44            throw new InvalidOperationException();
45        }
46    }
47 }
```

```
1 [ ShortRunJob ]
2 [ MemoryDiagnoser ]
3 public class Step2_PipelineException {
4     [ GlobalSetup ]
5     public void SetUp() {
6         ...
7         var stepId = PipelineDepth + 1;
8
9         pipelineModificationsBeforeOptimizations.Additions.Add(RegisterStep.Create(stepId.ToString(), typeof(Throwing), "1", b => new Throwing()));
10
11         ...
12
13         pipelineBeforeOptimizations = new
14             Step1.PipelineOptimization<IBehaviorContext>(null, new SettingsHolder(),
15                 pipelineModificationsBeforeOptimizations);
16         pipelineAfterOptimizations = new PipelineOptimization<IBehaviorContext>
17             (null, new SettingsHolder(),
18                 pipelineModificationsAfterOptimizations);
19     }
20 }
```



```
1 [ShortRunJob]
2 [MemoryDiagnoser]
3 public class Step2_PipelineException {
4     [GlobalSetup]
5     public void SetUp() {
6         ...
7     }
8
9     [Benchmark(Baseline = true)]
10    public async Task Before() {
11        try
12        {
13            await pipelineBeforeOptimizations.Invoke(behaviorContext);
14        }
15        catch (InvalidOperationException)
16        {
17        }
18    }
19
20    [Benchmark]
21    public async Task After() {
22        try
23        {
24            await pipelineAfterOptimizations.Invoke(behaviorContext);
25        }
26        catch (InvalidOperationException)
27        {
28        }
29    }
30    ...
31 }
```

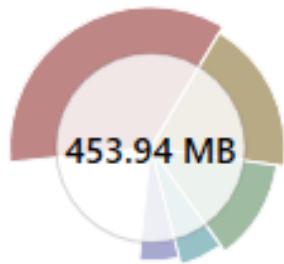


# PROFILING THE PIPELINE (AGAIN)



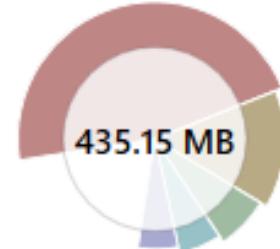
# MEMORY CHARACTERISTICS

Top types allocated by: NServiceBus.IMessageSessionExtensions



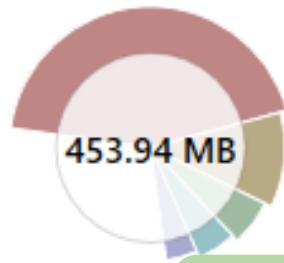
159.75 MB [Byte\[\]](#)  
84.39 MB [Char\[\]](#)  
58.78 MB [Int32\[\]](#)  
26.53 MB [ArrayList](#)  
25.77 MB [String](#)

Top types allocated by: NServiceBus.IMessageSessionExtensions



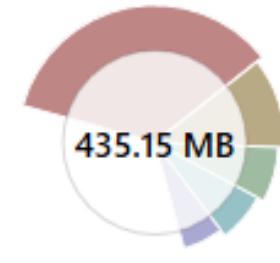
201.96 MB [Byte\[\]](#)  
63.77 MB [Char\[\]](#)  
30.59 MB [Int32\[\]](#)  
25.92 MB [Lookup+Grouping<String, Subscriber>\[\]](#)  
25.48 MB [String](#)

Top allocating methods in: NServiceBus.IMessageSessionExtensions



197.61 MB [StreamWriter.Init\(Stream, Encoding, Int32, Boolean\)](#)  
52.07 MB [String.SplitInternal\(Char\[\], Int32, StringSplitOptions\)](#)  
26.53 MB [MessageQueuePermission.get\\_PermissionEntries\(\)](#)  
23.90 MB [MemoryStream.set\\_Capacity\(Int32\)](#)  
20.22 MB [BehaviorChain.InvokeNext\(IHandlerContext, Int32\)](#)

Top allocating methods in: NServiceBus.IMessageSessionExtensions



152.55 MB [StreamWriter.Init\(Stream, Encoding, Int32, Boolean\)](#)  
49.01 MB [MemoryStream.set\\_Capacity\(Int32\)](#)  
31.30 MB [MemoryStream.ToArray\(\)](#)  
30.02 MB [Message.IdFromByteArray\(Byte\[\]\)](#)  
25.92 MB [Lookup< TKey, TElement >.ctor\(IEqualityComparer< T >\)](#)



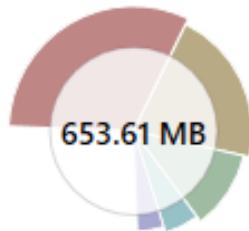
BEFORE

PUBLISH

AFTER

# MEMORY CHARACTERISTICS

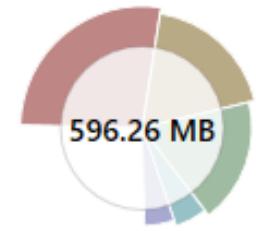
Top types allocated by: NServiceBus.ReceiveWithTransactionScope



206.75 MB [Char\[\]](#)  
138.86 MB [Byte\[\]](#)  
74.28 MB [String](#)  
34.54 MB [XmlTextReaderImpl+NodeData](#)  
27.83 MB [Func<IBehaviorContext, Task>](#)

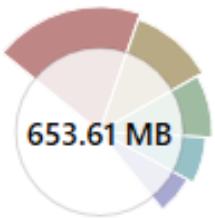


Top types allocated by: NServiceBus.ReceiveWithTransactionScope



161.42 MB [Char\[\]](#)  
111.67 MB [String](#)  
109.93 MB [Byte\[\]](#)  
29.97 MB [Int32\[\]](#)  
29.33 MB [Dictionary+Entry<String, Object>\[\]](#)

Top allocating methods in: NServiceBus.ReceiveWithTransactionScope

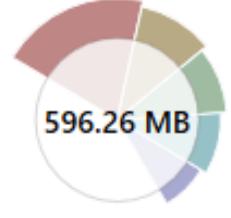


127.02 MB [XmlTextReaderImpl.InitTextReaderInput\(String, Uri, TextReader\)](#)  
75.02 MB [Message.get Extension\(\)](#)  
59.14 MB [BufferUtils.RentBuffer\(IArrayPool<T>, Int32\)](#)  
46.53 MB [XmlTextReaderImpl..ctor\(XmlResolver, XmlReaderSettings, XmlParserCor](#)  
40.56 MB [String.CreateStringFromEncoding\(Byte, Int32, Encoding\)](#)

BEFORE

RECEIVE

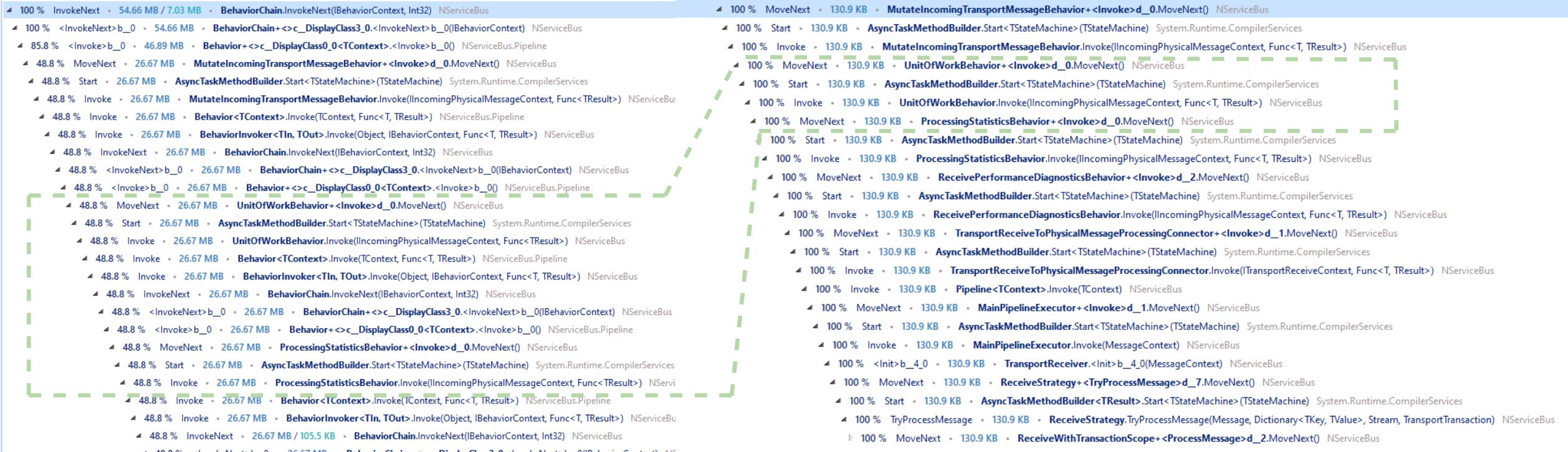
Top allocating methods in: NServiceBus.ReceiveWithTransactionScope



63.52 MB [XmlTextReaderImpl.InitTextReaderInput\(String, Uri, TextReader\)](#)  
58.75 MB [String.CreateStringFromEncoding\(Byte, Int32, Encoding\)](#)  
57.62 MB [Dictionary< TKey, TValue >.Initialize\(Int32\)](#)  
47.10 MB [Message.get Extension\(\)](#)  
47.10 MB [String.CtorCharArrayStartLength\(Char\[\], Int32, Int32\)](#)

AFTER

# MEMORY CHARACTERISTICS



## BEFORE

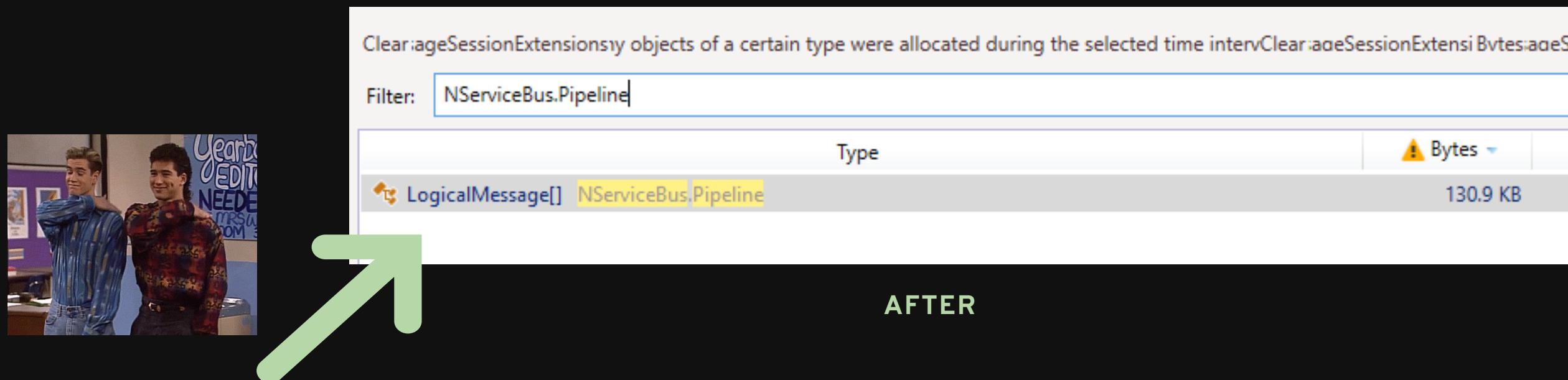
# RECEIVE

AFTER

# MEMORY CHARACTERISTICS

Type	Bytes
StageForkConnector+<>c_DisplayClass0_0<ITransportReceiveContext, IIIncomingPhysicalMessageContext>	15.74 MB
Behavior+<>c_DisplayClass0_0<IOutgoingLogicalMessageContext>	NServiceBus.Pipeline
Behavior+<>c_DisplayClass0_0<IRoutingContext>	NServiceBus.Pipeline
Behavior+<>c_DisplayClass0_0<IOutgoingPhysicalMessageContext>	NServiceBus.Pipeline

BEFORE

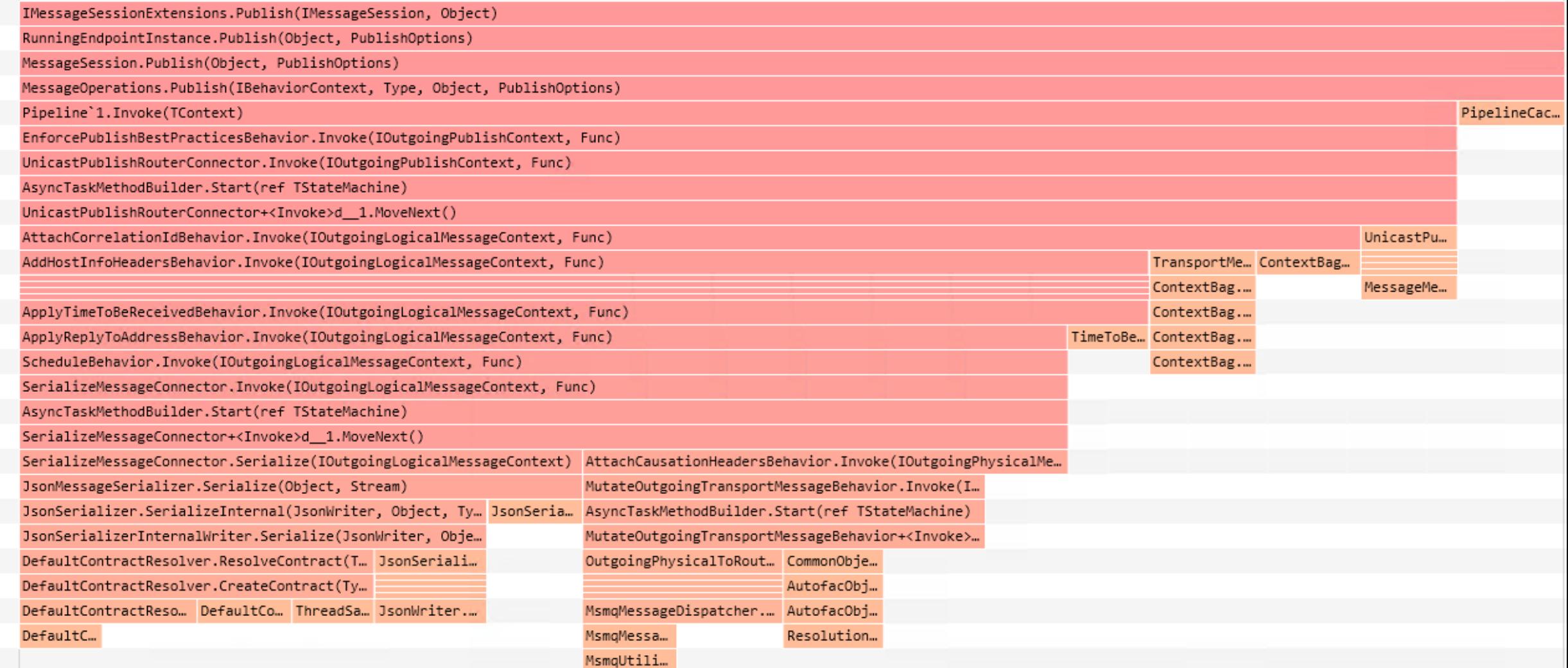


oh look, there is nothing 😊

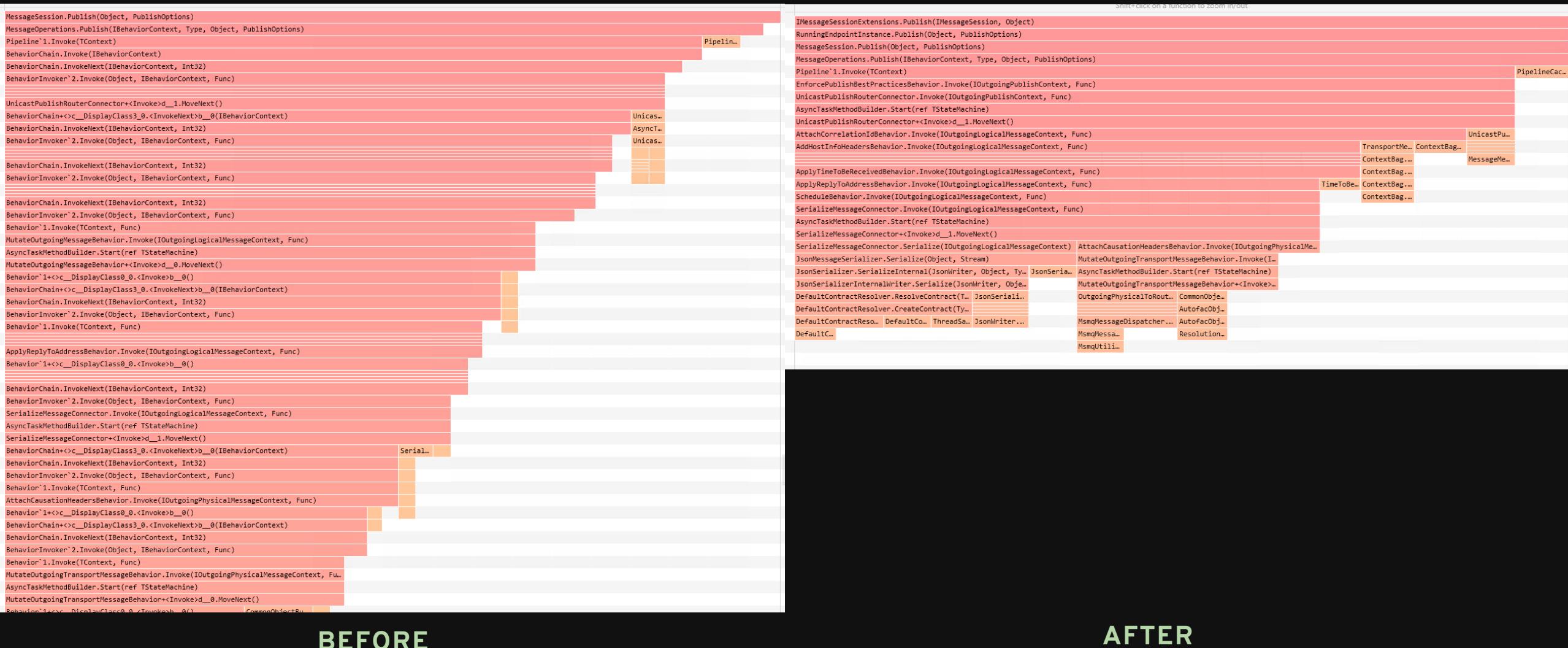
Profiling > Improving > Benchmarking > Profiling the pipeline (again)

# CPU CHARACTERISTICS

Shift+click on a function to zoom in/out



# CPU CHARACTERISTICS



BEFORE

PUBLISH

Profiling > Improving > Benchmarking > Profiling the pipeline (again)

# PREVENTING REGRESSIONS

- Guidance Preventing Regressions
- ResultComparer Tool

● ● ●

```
1 C:\Projects\performance\src\benchmarks\micro>
dotnet run -c Release -f net8.0 \
2      --artifacts "C:\results\before"
```

● ● ●

```
1 C:\Projects\performance\src\benchmarks\micro>
dotnet run -c Release -f net8.0 \
2      --artifacts "C:\results\after"
```

● ● ●

```
1 C:\Projects\performance\src\tools\ResultsComparer>
dotnet run --base "C:\results\before"
2 --diff "C:\results\after" --threshold 2%
```

"CPU-bound benchmarks are much more stable than Memory/Disk-bound benchmarks, but the *average* performance levels still can be up to **three times** different across builds."

Andrey Akinshin - Performance stability of GitHub Actions

# THE PERFORMANCE LOOP

A PRACTICAL GUIDE TO PROFILING AND BENCHMARKING

✉️ danielmarbach | 📩 daniel.marbach@particular.net | 💬 Daniel Marbach

- Use the performance loop to improve your code where it matters
- Combine it with profiling to observe how the small changes add up
- Optimize until you hit a diminishing point of return
- You'll learn a ton about potential improvements for a new design



[github.com/danielmarbach/BeyondSimpleBenchmarks](https://github.com/danielmarbach/BeyondSimpleBenchmarks)

