# Архитектурная парадигма System.IO.Pipelines: Глубокий анализ Zero-Allocation ввода-вывода и управления буферизацией в.NET Enterprise

## 1. Введение: Эволюция парадигмы обработки данных и смена контекста

Современная разработка высоконагруженных серверных систем (High-Load Backend) в экосистеме.NET претерпела фундаментальные изменения с выходом.NET Core 2.1 и последующих версий. Переход от монолитных приложений, базирующихся на синхронном вводе-выводе, к асинхронным событийно-ориентированным архитектурам потребовал пересмотра базовых примитивов работы с данными. Данный отчет, составленный в рамках теоретического блока третьего дня пятой недели обучения, посвящен детальному анализу библиотеки System.IO.Pipelines и концепции «Zero-Allocation Parsing» (парсинга без аллокаций).

Актуальность темы продиктована необходимостью перехода разработчиков, имеющих опыт в интерактивных системах реального времени (таких как Unity), к стандартам Enterprise-разработки.1 В среде Unity управление памятью часто подчинено жестким рамкам игрового цикла (Game Loop), где любая аллокация в кадре (frame) может привести к статтерам (stuttering). В серверной разработке на базе ASP.NET Core задачи схожи, но масштаб иной: вместо 60 кадров в секунду сервер должен обрабатывать десятки и сотни тысяч запросов в секунду (RPS). В таких условиях накладные расходы на копирование буферов памяти и работу сборщика мусора (Garbage Collector, GC) становятся критическим фактором, ограничивающим пропускную способность системы.1

Классическая модель, основанная на абстракции System.IO.Stream, несмотря на свою универсальность, архитектурно предрасполагает к избыточному копированию данных и фрагментации кучи. В противовес этому, модель Pipelines предлагает инверсию контроля над буфером: вместо того чтобы пользователь выделял память и передавал её потоку, поток сам предоставляет пользователю доступ к своим внутренним буферам. Это позволяет реализовать сценарии обработки данных, при которых байты считываются из сетевой карты, проходят через цепочку обработки и парсинга, и отбрасываются, ни разу не будучи скопированными в промежуточные массивы C#.2

В данном документе будет проведен исчерпывающий анализ:

1. Архитектурных ограничений классического ввода-вывода (Stream) и их влияния на производительность.
2. Внутреннего устройства примитивов Span<T>, Memory<T> и ReadOnlySequence<T>.
3. Механики работы Pipe, PipeReader и PipeWriter, включая управление обратным давлением (Backpressure).
4. Алгоритмов векторного поиска и парсинга данных без аллокаций, необходимых для выполнения практического задания по анализу логов.1

## 2. Кризис классической модели I/O: Почему Stream недостаточно

Чтобы понять ценность System.IO.Pipelines, необходимо детально разобрать анатомию работы System.IO.Stream и выявить те узкие места, которые делают его непригодным для сверхвысоких нагрузок.

### 2.1. Механика системных вызовов и буферизация ядра

Любая операция ввода-вывода (чтение файла, получение пакета из сети) начинается за пределами управляемого кода (Managed Code) — в пространстве ядра операционной системы (Kernel Space). Когда сетевая карта принимает пакет, он попадает в буфер драйвера, а затем в буфер сокета в пространстве ядра.

В классической модели.NET разработчик создает буфер в пространстве пользователя (User Space) — обычно это массив byte. Вызов stream.Read(buffer, 0, count) инициирует системный вызов, который копирует данные из буфера ядра в буфер приложения. Это первое копирование.

Однако абстракция FileStream или NetworkStream в.NET часто имеет собственный внутренний буфер для оптимизации мелких чтений. Таким образом, путь данных выглядит так:

1. **Network Card/Disk Controller** -> **Kernel Buffer** (DMA transfer).
2. **Kernel Buffer** -> **Internal Stream Buffer** (CPU copy, managed by Runtime).
3. **Internal Stream Buffer** -> **User Buffer** (CPU copy, managed by Developer).

Каждая операция копирования (memcpy) не только потребляет такты процессора, но, что более важно, "загрязняет" кэш процессора (L1/L2/L3 cache pollution), вытесняя полезные данные инструкций и переменных приложения. В высоконагруженных системах пропускная способность памяти (Memory Bandwidth) часто становится бутылочным горлышком раньше, чем загрузка CPU.3

### 2.2. Проблема закрепления памяти (Pinning) и фрагментация кучи

Для того чтобы операционная система могла скопировать данные в управляемый массив byte (который может быть перемещен сборщиком мусора в любой момент), этот массив должен быть "закреплен" (pinned) в физической памяти на время выполнения операции ввода-вывода.

В асинхронном программировании (ReadAsync) операция может длиться неопределенное время. Если массив закреплен, GC не может его переместить при уплотнении кучи (Compaction).

* Если массив мал, он находится в **SOH (Small Object Heap)**. Закрепленные объекты в SOH вызывают фрагментацию памяти, так как создают "дыры", которые нельзя устранить.
* Если массив велик (>= 85 000 байт), он попадает в **LOH (Large Object Heap)**, который до недавнего времени вообще не уплотнялся (до.NET Core 3.0), что приводило к постоянному росту потребления памяти процессом.

Таблица 1 иллюстрирует сравнение накладных расходов при различных подходах к буферизации.

| **Характеристика** | **Классический Stream (byte)** | **Оптимизированный Stream (ArrayPool)** | **System.IO.Pipelines** |
| --- | --- | --- | --- |
| **Аллокация буфера** | При каждом чтении (если new byte) | Аренда из пула, возврат вручную | Автоматически, управляется Pipe |
| **Копирование памяти** | Минимум 2 раза (Kernel->Stream->User) | Минимум 2 раза | 1 раз (Kernel->Pipe Memory) |
| **Давление на GC** | Высокое (создание/уничтожение массивов) | Среднее (управление арендой) | Нулевое (переиспользование сегментов) |
| **Управление потоком** | Нет (чтение до заполнения буфера) | Нет | Встроено (Backpressure) |
| **Парсинг** | Требует копирования в string/struct | Требует сложной логики границ буфера | Zero-copy через ReadOnlySequence |

### 2.3. Отсутствие семантики парсинга

Абстракция Stream представляет собой "трубу" с байтами, но не предоставляет удобных механизмов для работы с логическими единицами данных (сообщениями, строками, кадрами).

Рассмотрим чтение данных из TCP-сокета. Сообщение может быть больше размера буфера чтения. В модели Stream разработчик сталкивается с ситуацией:

1. Прочитали 1024 байта.
2. Нашли начало сообщения, но конец находится в следующем пакете.
3. Нужно скопировать "хвост" текущего буфера в новый, больший буфер.
4. Прочитать следующие байты в этот новый буфер.
5. Склеить (memcpy) и попытаться распарсить снова.

Этот код (boilerplate code) сложен, подвержен ошибкам (off-by-one errors) и крайне неэффективен по памяти.2 В отличие от этого, Pipelines предоставляет абстракцию, которая скрывает разрывы буферов, позволяя работать с данными так, как будто они лежат в непрерывной памяти.5

## 3. Новые примитивы памяти: Span, Memory и ReadOnlySequence

Переход к Pipelines невозможен без глубокого понимания типов, введенных в.NET Core 2.1 специально для высокопроизводительных вычислений. Эти типы обеспечивают безопасность типов (Type Safety) при работе с памятью без накладных расходов.

### 3.1. Span<T>: Безопасный указатель на стеке

Span<T> — это фундаментальный тип, представляющий собой окно в непрерывную область памяти. Это ref struct, что накладывает строгие ограничения: экземпляр Span может существовать только на стеке вызовов. Он не может быть полем класса (в куче), не может быть упакован (boxed) и не может использоваться в асинхронных методах (async/await) или итераторах (yield return), так как эти конструкции подразумевают сохранение состояния в куче.4

Структурно Span содержит два поля: указатель на память (ByReference) и длину. JIT-компилятор использует эту информацию для удаления проверок границ массива (Bounds Check Elimination) внутри циклов, что делает доступ к данным через Span столь же быстрым, как и через сырые указатели (pointers), но безопасным.4

Применение в парсинге:

При анализе логов мы будем использовать ReadOnlySpan<byte> для просмотра содержимого буферов. Поскольку парсинг — это синхронная операция (мы просто сканируем память), использование Span здесь идеально подходит и гарантирует отсутствие аллокаций.

### 3.2. Memory<T>: Владелец памяти в куче

Так как PipeReader работает асинхронно, он не может возвращать Span<T> напрямую в методе ReadAsync. Для передачи владения областью памяти используется структура Memory<T>. Это обычная структура (не ref struct), которая может храниться в куче (например, внутри Task<ReadResult>).

Memory<T> является фабрикой для Span<T>. Когда асинхронная операция завершается и мы переходим к синхронной обработке данных, мы вызываем свойство .Span у Memory<T>, получая быстрый доступ к данным.7 В архитектуре Pipelines сегменты данных хранятся именно как блоки Memory<byte>.

### 3.3. ReadOnlySequence<T>: Связный список сегментов

Это наиболее сложная и важная структура для понимания работы Pipelines. Поскольку Pipe может динамически выделять память кусками (сегментами) по мере поступления данных, логически непрерывное сообщение может физически располагаться в нескольких разрозненных блоках памяти.

ReadOnlySequence<T> — это абстракция, которая может представлять:

1. **Один сегмент:** (Fast Path) Обычная обертка над ReadOnlyMemory<T>.
2. **Множество сегментов:** (Slow Path) Связный список (Linked List) сегментов ReadOnlyMemory<T>.4

Для разработчика это означает, что нельзя просто взять buffer[i] для доступа к произвольному байту, так как индекс i может указывать на память в другом физическом блоке. Работа с ReadOnlySequence требует использования специальных итераторов (SequenceReader) или метода Slice(), который корректно обрабатывает границы сегментов.2

**Важный инсайт:** Тип ReadOnlySequence позволяет реализовать Zero-Copy Slicing. Когда мы берем подстроку из буфера (Slice), мы не копируем байты. Мы создаем новую легкую структуру ReadOnlySequence, которая содержит указатели на начало и конец нужного диапазона в уже существующих сегментах памяти.8

## 4. Архитектура System.IO.Pipelines

Библиотека System.IO.Pipelines реализует паттерн "Producer-Consumer" с разделением ответственности за управление памятью. Центральным элементом является класс Pipe.

### 4.1. Внутреннее устройство Pipe

Объект Pipe содержит два публичных свойства: PipeWriter и PipeReader. Внутри себя он управляет связным списком сегментов памяти.

* **Writer** добавляет данные в конец списка (Tail).
* **Reader** потребляет данные из начала списка (Head).

Такая структура превращает буфер в кольцевой (или бесконечно растущий/освобождающийся) список, где сегменты, освобожденные Reader'ом, возвращаются в пул памяти для повторного использования Writer'ом.2

### 4.2. PipeWriter: Инверсия контроля буфера

Ключевое отличие PipeWriter от Stream — метод GetMemory(int sizeHint). Вместо того чтобы просить пользователя "дай мне массив, я запишу туда данные", Writer говорит: "вот тебе свободная память (Memory<byte>), пиши прямо сюда".10

Цикл записи (Producer Loop) выглядит следующим образом:

1. **GetMemory(size)**: Writer запрашивает у Pipe свободный кусок памяти. Pipe либо возвращает часть текущего сегмента, либо арендует новый сегмент из MemoryPool<T>.2
2. **Запись данных**: Данные (например, из сокета) копируются непосредственно в эту память. Если используется Socket.ReceiveAsync, данные попадают туда прямиком из сетевого стека (Zero-Copy с точки зрения managed heap).
3. **Advance(bytesWritten)**: Пользователь сообщает Writer'у, сколько байт реально было записано. Это сдвигает внутренний курсор записи (WriteHead).
4. **FlushAsync()**: Делает записанные данные видимыми для Reader'а. Этот метод также реализует механизм **Backpressure** (обратного давления).9

Backpressure и управление потоком:

Если Reader не успевает разгребать данные, количество байт в пайпе растет. Когда оно достигает порога PauseWriterThreshold (задается в PipeOptions, по умолчанию 64 КБ для потоков), метод FlushAsync возвращает незавершенный ValueTask. Задача записи приостанавливается до тех пор, пока Reader не освободит память (опустит уровень ниже ResumeWriterThreshold).11 Это предотвращает переполнение памяти (OutOfMemory) при медленном потребителе — критически важное свойство для стабильности сервера.

### 4.3. PipeReader: Потребление и жизненный цикл данных

На стороне потребителя (Consumer) процесс построен вокруг метода ReadAsync() и концепции "Consumed/Examined".

1. **ReadAsync()**: Возвращает ReadResult, содержащий Buffer (ReadOnlySequence<byte>).
2. **Обработка**: Потребитель анализирует буфер. Он может найти одно или несколько полных сообщений (строк лога).
3. **AdvanceTo(consumed, examined)**: Это самый важный метод для корректности работы пайплайна.
   * **Consumed (Потреблено):** Позиция курсора, указывающая на данные, которые были полностью обработаны и больше не нужны. Память до этой позиции будет возвращена в пул.
   * **Examined (Изучено):** Позиция курсора, указывающая, до какого места мы просмотрели буфер. Если мы не нашли разделитель конца сообщения, examined должно быть равно buffer.End. Это сигнал для пайпа: "Я посмотрел всё, что есть, но мне мало. Разбуди меня в следующий раз только когда придут *новые* данные".10

Анти-паттерн бесконечного цикла:

Если вызвать AdvanceTo(buffer.Start, buffer.Start) (ничего не потреблено, ничего не изучено), пайплайн решит, что потребитель не смог обработать данные, и немедленно вернет те же самые данные в следующем вызове ReadAsync. Это приведет к 100% загрузке CPU в бесконечном цикле (Hot Loop).10

### 4.4. Модель потоков и PipeScheduler

По умолчанию System.IO.Pipelines использует PipeScheduler.ThreadPool для выполнения продолжений (callbacks) асинхронных операций. Это означает, что после завершения ввода-вывода обработка данных продолжится в потоке из пула потоков.NET.10

Однако архитектура позволяет подменить планировщик. Например, для высокоприоритетных операций можно использовать PipeScheduler.Inline (выполнение в том же потоке, что и I/O завершение), но это опасно возможными взаимоблокировками (deadlocks). В контексте задачи по анализу логов использование дефолтного ThreadPool является оптимальным, так как оно обеспечивает балансировку нагрузки между ядрами процессора.

## 5. Стратегии Zero-Allocation парсинга и анализ логов

Перейдем к прикладной теории, необходимой для выполнения задания третьего дня: реализации парсера логов, который ищет подстроку "ERROR" без создания строк.

### 5.1. Ментальная модель "Окно памяти"

При работе с StreamReader.ReadLine мы привыкли мыслить строками. В Pipelines мы должны мыслить "окнами" (Windows). Мы получаем окно ReadOnlySequence<byte>, в котором может быть:

* Часть строки.
* Одна полная строка.
* Полная строка и начало следующей.
* Много строк.

Задача алгоритма — найти разделитель (обычно \n), "вырезать" (Slice) сегмент до разделителя, обработать его, и повторить процесс, пока в окне есть полные строки. Оставшийся "хвост" оставляется в буфере до прихода следующей порции данных.14

### 5.2. SequenceReader<T>: Инструмент навигации

Для упрощения работы со сложной структурой ReadOnlySequence была введена ref struct SequenceReader<T>. Это высокопроизводительный курсор, который умеет прозрачно перешагивать через границы сегментов памяти.15

Ключевой метод — TryReadTo(out ReadOnlySequence<byte> sequence, byte delimiter, bool advancePastDelimiter).

Этот метод сканирует память в поисках байта-разделителя.

* Если данные лежат в одном сегменте (IsSingleSegment), он использует сверхбыстрый Span.IndexOf (с SIMD оптимизацией).
* Если данные пересекают границы сегментов, он переключается на "медленный путь" (Slow Path), копируя байты в локальный буфер или итерируясь посегментно.16

### 5.3. Векторный поиск подстроки "ERROR" (SIMD)

Задание требует найти количество вхождений строки "ERROR". Поскольку мы не можем конвертировать байты в строку, нам нужно искать последовательность байтов [0x45, 0x52, 0x52, 0x4F, 0x52] (ASCII/UTF-8 коды для "ERROR").

Наивный поиск вложенным циклом (for i.. for j..) будет крайне медленным. Современный.NET использует интринсики (Hardware Intrinsics) для векторизации таких операций.

**Алгоритм реализации:**

1. Получаем ReadOnlySequence<byte> line (одна строка лога).
2. Если line.IsSingleSegment (самый частый случай):
   * Получаем ReadOnlySpan<byte> span = line.First.Span.
   * Используем span.IndexOf(new ReadOnlySpan<byte>(new byte { 69, 82, 82, 79, 82 })).
   * Компилятор и JIT превратят это в инструкции AVX2/SSE, которые сравнивают по 16 или 32 байта за такт процессора.17
3. Если line многосегментный (редкий случай):
   * Можно использовать CopyTo в небольшой буфер на стеке (stackalloc byte[length]), если строка короткая.
   * Либо использовать SequenceReader.TryReadTo или побайтовое сравнение на границах.

Таблица 2 демонстрирует разницу в подходах к поиску.

| **Метод** | **Механизм** | **Аллокации (Heap)** | **Сложность** |
| --- | --- | --- | --- |
| **String.Contains** | Encoding.GetString -> string.IndexOf | Высокие (string per line) | O(N) + GC Overhead |
| **Naive Byte Scan** | Цикл for по байтам | 0 | O(N\*M), медленно |
| **Span.IndexOf** | SIMD (Vectorized CPU instructions) | 0 | O(N/VectorWidth), очень быстро |
| **SequenceReader** | Гибридный (Span + Manual) | 0 | Оптимально для Pipelines |

### 5.4. Паттерн реализации цикла обработки

Ниже приведен эталонный паттерн цикла обработки для PipeReader, учитывающий все нюансы управления памятью.

C#

async Task ProcessLogs(PipeReader reader)  
{  
 while (true)  
 {  
 ReadResult result = await reader.ReadAsync();  
 ReadOnlySequence<byte> buffer = result.Buffer;  
  
 while (TryReadLogLine(ref buffer, out ReadOnlySequence<byte> line))  
 {  
 ProcessLine(line); // Поиск "ERROR" внутри line  
 }  
  
 // К этому моменту buffer содержит только неполный остаток данных.  
 // Мы говорим пайпу:  
 // 1. Consumed: buffer.Start (всё до текущего начала потреблено)  
 // 2. Examined: buffer.End (мы посмотрели всё до конца и не нашли \n)  
 reader.AdvanceTo(buffer.Start, buffer.End);  
  
 if (result.IsCompleted) break;  
 }  
 await reader.CompleteAsync();  
}  
  
bool TryReadLogLine(ref ReadOnlySequence<byte> buffer, out ReadOnlySequence<byte> line)  
{  
 // Ищем \n. Если нашли - возвращаем true и модифицируем buffer (срез)  
 SequencePosition? position = buffer.PositionOf((byte)'\n');  
 if (position == null)  
 {  
 line = default;  
 return false;  
 }  
  
 // Включаем \n в строку или исключаем - зависит от логики,   
 // здесь сдвигаемся СРАЗУ ПОСЛЕ \n  
 var nextLineStart = buffer.GetPosition(1, position.Value);  
 line = buffer.Slice(0, position.Value);  
 buffer = buffer.Slice(nextLineStart);  
 return true;  
}

.2

## 6. Управление памятью: ArrayPool и MemoryPool

Важным аспектом, скрытым "под капотом" Pipelines, является то, откуда берется память. По умолчанию Pipe использует MemoryPool<byte>.Shared.

### 6.1. Slab Allocation и Thread-Local Buckets

MemoryPool<byte>.Shared в.NET Core реализован на базе ArrayPool<T>.Shared. Это высокооптимизированный пул массивов. Он использует разделение на "корзины" (buckets) по размеру массивов (2^n).

Для минимизации конкуренции потоков (thread contention) при доступе к пулу, ArrayPool использует Thread-Local кэши. Это означает, что если поток взял массив и быстро его вернул, он работает со своим локальным кэшем без блокировок. Если же массивы "кочуют" между потоками (Writer в одном, Reader в другом), эффективность пула может снижаться, и включаются механизмы спин-локов (spin-locks).18

В архитектуре Pipelines это учтено: сегменты памяти удерживаются пайпом достаточно долго и переиспользуются циклически, что минимизирует частоту обращений к глобальному пулу.

### 6.2. Влияние на LOH (Large Object Heap)

Стандартные сегменты пайпа (обычно 4096 байт) попадают в SOH. Однако при передаче огромных объемов данных пайп может запрашивать большие блоки. Использование пулинг-механизма гарантирует, что эти большие массивы не будут создаваться и уничтожаться постоянно, что спасает LOH от фрагментации. Это критическое преимущество перед new byte, которое часто встречается в коде новичков при работе с файлами.4

## 7. Сравнение производительности и заключение

Синтетические бенчмарки и опыт внедрения в Kestrel показывают следующую картину производительности 19:

1. **Throughput (Пропускная способность):** Pipelines демонстрируют прирост на 30-50% по сравнению с Stream + StreamReader в сценариях текстового парсинга.
2. **Latency (Задержка):** Значительное снижение "хвоста" задержек (p99 latency) за счет устранения пауз GC. В системах на базе Pipelines график потребления памяти (Memory Working Set) выглядит как плоская линия, тогда как на Streams он имеет форму "пилы" (sawtooth) из-за работы GC.
3. **Код:** Код на Pipelines более вербозный (verbose) и сложный в написании, чем StreamReader.ReadLine(), но он дает детерминированное поведение ресурсов.

### Резюме для практического задания

Для успешного выполнения задания третьего дня вам необходимо:

1. Отказаться от парадигмы "строка = string". Строка — это ReadOnlySequence<byte> или ReadOnlySpan<byte>.
2. Использовать PipeReader.Create(FileStream) для эффективного чтения файла.
3. Реализовать цикл ReadAsync -> PositionOf(\n) -> Process -> AdvanceTo.
4. В методе Process использовать span.IndexOf(errorBytes) для поиска вхождения подстроки.
5. Внимательно следить за аргументами AdvanceTo, чтобы избежать бесконечных циклов.

Освоение System.IO.Pipelines — это не просто изучение нового API. Это шаг к пониманию того, как работает современный высокопроизводительный ввод-вывод на уровне "железа", что является отличительной чертой Senior-разработчика платформы.NET.

#### Источники

1. План обучения работе с файлами и конфигурацией
2. System.IO.Pipelines: High performance IO in .NET - Microsoft Developer Blogs, дата последнего обращения: декабря 4, 2025, <https://devblogs.microsoft.com/dotnet/system-io-pipelines-high-performance-io-in-net/>
3. Read SSL via PipeReader in .NET - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/57018791/read-ssl-via-pipereader-in-net>
4. Writing High-Performance Code Using Span
5. Code, code and more code.: Pipe Dreams, part 1 - Marc Gravell, дата последнего обращения: декабря 4, 2025, <https://blog.marcgravell.com/2018/07/pipe-dreams-part-1.html>
6. Span / Memory / ReadOnlySequence in C# - Steven Giesel, дата последнего обращения: декабря 4, 2025, <https://steven-giesel.com/blogPost/8bb02704-a113-40a1-9414-fe7b8cb3dda5>
7. My mental model of Span, Memory and ReadOnlySequence in .NET, дата последнего обращения: декабря 4, 2025, <https://blog.scooletz.com/2021/02/03/spans-memory-mental-model>
8. Data Structure Parsing with System.IO.Pipelines - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/53573198/data-structure-parsing-with-system-io-pipelines>
9. System.IO.Pipelines — a little-known tool for lovers of high performance - Habr, дата последнего обращения: декабря 4, 2025, <https://habr.com/en/articles/466137/>
10. System.IO.Pipelines - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/io/pipelines>
11. Catch up with .NET Core - System.IO.Pipelines | N+1 Blog, дата последнего обращения: декабря 4, 2025, <https://nikiforovall.blog/dotnet-core/2020/04/24/catch-dotnetcore-piepelines.html>
12. PipeReader.AdvanceTo Method (System.IO.Pipelines) - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.pipelines.pipereader.advanceto?view=net-10.0>
13. DocsStaging/Pipelines.md at master - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/davidfowl/DocsStaging/blob/master/Pipelines.md>
14. An Introduction to SequenceReader - Steve Gordon - Code with Steve, дата последнего обращения: декабря 4, 2025, <https://www.stevejgordon.co.uk/an-introduction-to-sequencereader>
15. System.Buffers - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/io/buffers>
16. Why is this System.IO.Pipelines code much slower than Stream-based code?, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/64283938/why-is-this-system-io-pipelines-code-much-slower-than-stream-based-code>
17. What the fastest way to find all positions of a target byte in a span of bytes? - Reddit, дата последнего обращения: декабря 4, 2025, <https://www.reddit.com/r/csharp/comments/1fn0d2i/what_the_fastest_way_to_find_all_positions_of_a/>
18. System.IO.Pipelines: The default MemoryPool is suboptimal · Issue #27748 · dotnet/runtime, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/27748>
19. Experimenting with System.IO.Pipelines for high performance audio, дата последнего обращения: декабря 4, 2025, <https://markheath.net/post/system-io-pipelines-high-perf-audio>
20. High performance IO with System.IO.Pipelines - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/shows/on-dotnet/high-performance-io-with-systemiopipelines>