# Архитектурная трансформация и управление данными: Глубокий анализ подсистем ввода-вывода и конфигурации в среде.NET Enterprise

## 1. Введение: Смена парадигмы персистентности и управления состоянием

Переход от разработки интерактивных приложений реального времени (Unity) к проектированию высоконагруженных серверных систем (Enterprise.NET Backend) требует фундаментального пересмотра подходов к взаимодействию с файловой системой и управлению конфигурацией. В среде Unity разработчик оперирует в рамках детерминированного игрового цикла (Game Loop), где доступ к диску часто является блокирующей операцией, допустимой лишь в моменты загрузки сцен или сохранения прогресса, и где состояние приложения жестко привязано к жизненному циклу объектов в памяти.1 В противоположность этому, архитектура современного бэкенда на базе ASP.NET Core диктует требования абсолютной асинхронности, отсутствия состояния (statelessness) и способности обрабатывать тысячи конкурентных запросов ввода-вывода (I/O) без блокировки потоков пула.1

Настоящий отчет представляет собой исчерпывающее руководство и детальный учебный план для 5-й недели программы обучения, сфокусированной на "Работе с файловой системой и конфигурацией". Анализ, проведенный на основе предоставленных материалов и глубокого исследования экосистемы.NET, выявляет, что поверхностного изучения классов File или ConfigurationBuilder недостаточно для уровня Senior Backend Developer. Требуется глубокое понимание механики взаимодействия CLR (Common Language Runtime) с ядром операционной системы, принципов управления буферами памяти, алгоритмов ротации логов и паттернов надежного отслеживания изменений в файловой системе.

В данном документе детально рассматриваются теоретические основы и практические паттерны, необходимые для реализации надежных подсистем ввода-вывода, которые станут фундаментом для выполнения заданий недели, таких как разработка утилиты envdir и высокопроизводительного файлового копировщика.

## 2. Механика файлового ввода-вывода в.NET: От User Space к Kernel Space

Понимание производительности файловых операций начинается с осознания стоимости пересечения границы между пользовательским кодом (managed code) и ядром операционной системы. Каждая операция чтения или записи инициирует системный вызов, который влечет за собой переключение контекста, проверку прав доступа и работу драйверов файловой системы.

### 2.1. Асинхронность и модель ThreadPool

В контексте Unity разработчики часто используют корутины (Coroutines) для распределения нагрузки по кадрам. В.NET Core используется модель async/await, которая, на первый взгляд, кажется синтаксическим сахаром, но скрывает сложную машину состояний. Критически важным аспектом является различие между "истинным" асинхронным вводом-выводом и эмуляцией.

При использовании FileStream без явного указания флага FileOptions.Asynchronous (или useAsync: true в конструкторе), операционная система может выполнять операции синхронно, блокируя поток выполнения до завершения работы дискового контроллера, даже если в коде используется метод ReadAsync.2 В высоконагруженном веб-сервере это приводит к исчерпанию пула потоков (ThreadPool Starvation), так как рабочие потоки простаивают в ожидании ответа от диска, вместо того чтобы обслуживать новые HTTP-запросы.3

Правильная реализация требует использования перекрывающего ввода-вывода (Overlapped I/O в Windows) или механизмов epoll/io\_uring в Linux. Когда инициируется истинная асинхронная операция, поток возвращается в пул, а задача ввода-вывода передается драйверу устройства. По завершении операции аппаратное прерывание уведомляет ОС, которая ставит задачу продолжения (continuation) в очередь ThreadPool.

### 2.2. Оптимизация буферизации и пропускная способность

Стандартный размер буфера в FileStream составляет 4 КБ (4096 байт).4 Это значение исторически связано с размером страницы памяти и сектора диска. Однако для современных задач копирования больших файлов или обработки массивных логов этот размер является субоптимальным. Исследования показывают, что увеличение размера буфера может существенно снизить количество системных вызовов и накладных расходов на переключение контекста.

Анализ производительности различных размеров буфера при асинхронном копировании показывает следующую динамику:

| **Размер файла** | **Размер буфера (Best Practice)** | **Обоснование** |
| --- | --- | --- |
| < 128 КБ | 0 (Отсутствие буферизации) | Прямое чтение эффективнее накладных расходов на создание буфера.5 |
| 512 КБ - 1 МБ | 65 536 - 81 920 байт | Баланс между скоростью и потреблением памяти. Значение 81920 (чуть меньше 85 КБ) позволяет избежать попадания объекта буфера в Large Object Heap (LOH).5 |
| > 100 МБ | 1 МБ (с SetLength) | Минимизация системных вызовов. Критически важно предварительно задать размер целевого файла через SetLength, чтобы избежать фрагментации MFT (Master File Table).4 |

Важным нюансом является использование флага FileOptions.SequentialScan. Этот флаг сообщает менеджеру кэша операционной системы, что данные будут читаться последовательно, что позволяет ОС выполнять упреждающее чтение (read-ahead), загружая данные в кэш до того, как приложение их запросит.2

### 2.3. System.IO.Pipelines: Сдвиг парадигмы обработки данных

Традиционная работа со Stream в.NET сопряжена с избыточным копированием памяти. Данные считываются с диска в буфер ядра, затем копируются во внутренний буфер FileStream, и наконец — в массив byte, выделенный пользователем. Для Unity-разработчика, привыкшего избегать аллокаций в Update, это должно звучать тревожно.

Библиотека System.IO.Pipelines, рожденная из недр веб-сервера Kestrel, предлагает альтернативу. Она управляет памятью самостоятельно, используя пулы буферов (MemoryPool<T>), и позволяет парсить данные прямо в буфере без копирования.7 Это достигается за счет использования структур Span<T> и Memory<T>, которые представляют собой безопасные указатели на области памяти.

**Преимущества Pipelines перед классическими Stream:**

1. **Zero-copy parsing:** Возможность обрабатывать данные (например, искать символы новой строки в логах) без создания промежуточных строк или массивов.8
2. **Backpressure (Обратное давление):** Встроенные механизмы управления потоком данных, предотвращающие переполнение памяти, если скорость чтения превышает скорость обработки (writer vs reader).8
3. **Уменьшение GC Pressure:** Повторное использование буферов из пула снижает нагрузку на сборщик мусора, что критично для серверов, работающих 24/7.9

Тем не менее, внедрение Pipelines требует изменения мышления. Вместо линейного чтения байтов разработчик работает с PipeReader и циклом ReadAsync -> Examine -> AdvanceTo. Ошибка в вызове AdvanceTo (сообщение о том, сколько данных обработано и сколько нужно оставить) может привести к бесконечным циклам или потере данных.10

### 2.4. Memory-Mapped Files: Работа с гигантскими объемами данных

Для задач, связанных с файлами, превышающими объем оперативной памяти, или для обеспечения сверхбыстрого произвольного доступа,.NET предоставляет MemoryMappedFiles. Этот механизм проецирует часть файла в виртуальное адресное пространство процесса. Операционная система берет на себя управление страницами памяти, загружая их с диска по требованию (page fault) и выгружая при нехватке памяти.11

В контексте перехода от Unity это можно сравнить с тем, как движок стримит текстуры, но на более низком уровне. Использование MemoryMappedFile позволяет нескольким процессам разделять одну и ту же память (Shared Memory), что является самым быстрым способом межпроцессного взаимодействия (IPC) на локальной машине.13 Однако, стоит учитывать, что для последовательного чтения FileStream с правильно настроенным буфером часто оказывается быстрее за счет меньших накладных расходов на управление страницами виртуальной памяти.15

## 3. Архитектура конфигурации: От ScriptableObject к IOptions

В Unity конфигурация часто хранится в префабах или ScriptableObject, которые сериализуются в бинарные или текстовые форматы внутри ассетов. В Enterprise-разработке конфигурация должна быть строго отделена от кода и артефактов сборки, следуя методологии 12-Factor App.

### 3.1. Паттерн Options в ASP.NET Core

Платформа.NET Core предлагает мощную абстракцию IOptions<T>, которая позволяет связывать (bind) секции конфигурации (из JSON, переменных окружения, Azure Key Vault) с строго типизированными классами C#.16 Это решает проблему "магических строк" и позволяет валидировать конфигурацию на старте приложения.

Различают три основных интерфейса, выбор между которыми зависит от требуемого жизненного цикла конфигурации:

1. **IOptions<T>:** Регистрируется как Singleton. Значения считываются один раз при старте приложения. Изменения в файле конфигурации после старта игнорируются. Идеально для статических настроек, которые не должны меняться.17
2. **IOptionsSnapshot<T>:** Регистрируется как Scoped. Значения перечитываются при каждом новом HTTP-запросе. Это позволяет обновлять конфигурацию без перезапуска приложения, но создает накладные расходы на чтение при каждом запросе.
3. **IOptionsMonitor<T>:** Регистрируется как Singleton, но предоставляет механизм подписки на изменения (OnChange). Это гибридный подход, позволяющий реагировать на изменения конфигурации в реальном времени даже внутри синглтонов, что критично для микросервисов, требующих высокой доступности.17

### 3.2. Внедрение зависимостей и валидация

Критически важным аспектом, который часто упускают новички, является валидация конфигурации. Внедрение "сломанной" конфигурации может привести к падению сервиса в runtime. Использование ValidateDataAnnotations и ValidateOnStart позволяет предотвратить запуск приложения, если обязательные ключи конфигурации отсутствуют или имеют неверный формат (например, URL базы данных).18

Паттерн Options также позволяет решить проблему "Captive Dependency" (Захваченной зависимости), когда Singleton-сервис случайно захватывает Scoped-зависимость (например, IOptionsSnapshot), что приводит к использованию устаревших данных или утечкам памяти. Правильное использование IOptionsMonitor внутри Singleton-сервисов является стандартом индустрии.18

### 3.3. Управление переменными окружения и паттерн Envdir

В контейнеризированных средах (Docker, Kubernetes) конфигурация часто передается через переменные окружения. Утилита envdir, пришедшая из мира Unix (daemontools), представляет собой элегантный способ управления сложными конфигурациями процессов.19 Суть паттерна заключается в том, что директория представляет собой "окружение", каждый файл в ней — это имя переменной, а содержимое файла — её значение.

Реализация этого паттерна на C# требует работы с классом System.Diagnostics.Process и его свойством StartInfo.EnvironmentVariables. Важно понимать, что метод Environment.SetEnvironmentVariable изменяет переменные только *текущего* процесса. Чтобы передать измененное окружение дочернему процессу, необходимо модифицировать коллекцию StartInfo *до* вызова Process.Start.20 Кроме того, необходимо установить UseShellExecute = false, иначе управление переменными будет передано оболочке ОС, и программные изменения могут быть проигнорированы.21

## 4. Реактивные файловые системы и надежность мониторинга

Одной из самых сложных задач при переходе к бэкенду является надежное отслеживание изменений файлов. Класс FileSystemWatcher (FSW) в.NET предоставляет обертку над системными API (ReadDirectoryChangesW в Windows, inotify в Linux), но обладает рядом существенных недостатков, делающих его использование "в лоб" опасным для продакшена.

### 4.1. Проблемы FileSystemWatcher

1. **Переполнение буфера:** FSW использует внутренний буфер (по умолчанию 8 КБ) для хранения событий. При массовом изменении файлов (например, git checkout или распаковка архива) буфер переполняется, и события теряются без предупреждения (кроме события Error).22 Увеличение буфера до 64 КБ возможно, но он аллоцируется в неподкачиваемой памяти (non-paged pool), что является дорогим ресурсом.
2. **Дублирование событий:** Сохранение файла в текстовом редакторе может вызвать серию событий (создание временного файла, изменение атрибутов, запись содержимого, переименование). Наивная подписка на событие Changed приведет к многократному запуску логики обработки.22
3. **Блокировка файлов:** Событие Created или Changed может сработать *до* того, как процесс, пишущий файл, закроет дескриптор. Попытка открыть файл на чтение в обработчике события немедленно вызовет IOException (файл занят другим процессом).22

### 4.2. Стратегии повышения надежности: Debouncing и Retry

Для создания надежной системы мониторинга необходимо применять паттерны Debouncing (устранение дребезга) и Throttling (дросселирование). Использование Reactive Extensions (Rx.NET) позволяет элегантно группировать события. Например, оператор Throttle позволяет игнорировать серию событий, пока не наступит период тишины (например, 500 мс), гарантируя, что обработка начнется только после завершения записи файла.24

Кроме того, обязательным является внедрение полиси повторных попыток (Retry Policy). Библиотека **Polly** позволяет декларативно описать логику: "При возникновении IOException подождать экспоненциально возрастающее время и повторить попытку".25 Это единственный надежный способ справиться с гонкой за файловые дескрипторы в многопоточной среде.

## 5. Детальный учебный план: Неделя 5

Основываясь на проведенном анализе, ниже представлен переработанный и углубленный план обучения, нацеленный на формирование компетенций уровня Senior.

### День 1: Продвинутая работа с потоками и асинхронный I/O

**Цель:** Научиться выполнять операции ввода-вывода без блокировки потоков ThreadPool и понимать стоимость системных вызовов.

**Теория:** [**Продвинутая работа с потоками и I/O день 1**](https://docs.google.com/document/d/1I8BjqvQvSq7SpaDFMTNdw0ZN31DFYXh7TU1WMtf0hOs/edit?usp=drive_link)

* Архитектура FileStream: буферизация, позиционирование, взаимодействие с ОС.
* Различия FileOptions.Asynchronous vs синхронный I/O. Оверхед на контекстные переключения.
* Понятие "Lock Contention" в файловой системе и флаги FileShare.27

**Практическое задание: "Асинхронный копировщик с прогрессом"**

* **Задача:** Написать консольную утилиту для копирования больших файлов (более 1 ГБ).
* **Требования:**
  1. Использовать FileStream с явно заданным буфером 80 КБ (81920 байт) и флагом FileOptions.Asynchronous.5
  2. Реализовать паттерн IProgress<double> для отображения прогресса в консоли без блокировки основного потока.29
  3. Реализовать поддержку отмены операции через CancellationToken.30
  4. **Анализ:** Сравнить производительность (время и нагрузку на CPU) при использовании буфера 4 КБ и 80 КБ.

**Протокол рефлексии (AI Mentor):**

* "Проанализируй, почему использование File.Copy блокирует поток. Как ConfigureAwait(false) влияет на возврат в контекст синхронизации в консольном приложении по сравнению с UI-приложением?"

### День 2: Управление процессами и паттерн Envdir

**Цель:** Освоить управление дочерними процессами и конфигурацией через переменные окружения, реализуя инфраструктурный паттерн.

**Теория:** [**неделя 5 день 2 теория**](https://docs.google.com/document/d/1kUJTLNwI2rHMXCcED7Z1f-EtzMVx2bShlbRAxtPTv2Y/edit?usp=drive_link)

* Класс System.Diagnostics.Process. Наследование окружения дочерними процессами.31
* Паттерн envdir и его роль в Cloud-Native приложениях (12-Factor App).19
* Коллекция ProcessStartInfo.Environment и нюансы UseShellExecute.20

**Практическое задание: "Реализация утилиты Envdir на C#"**

* **Задача:** Написать клон утилиты envdir.
* **Алгоритм:**
  1. Принять путь к директории и команду для запуска.
  2. Прочитать все файлы в директории. Имя файла = Ключ, Содержимое = Значение переменной окружения.
  3. Модифицировать StartInfo.EnvironmentVariables для запускаемого процесса.21
  4. Запустить дочерний процесс и перенаправить его потоки вывода (StdOut, StdErr) в консоль родителя.
  5. Обработать коды выхода дочернего процесса.

**Протокол рефлексии:**

* "В чем разница между Environment.SetEnvironmentVariable и модификацией StartInfo? Почему изменения в родительском процессе не всегда видны в уже запущенных дочерних?"

### День 3: System.IO.Pipelines и Zero-Allocation парсинг

**Цель:** Перейти от парадигмы "копирования данных" к парадигме "работы с окном памяти".

**Теория:** [**теория недели 5 дня 3**](https://docs.google.com/document/d/1k8L33jiMCN9u-7YC_SvEwStuYy9mcJTHuMdlvvSt-FI/edit?usp=drive_link)

* Проблемы Stream: аллокации буферов и GC Pressure.
* Архитектура Pipe, PipeReader, PipeWriter.
* Типы Span<T>, Memory<T>, ReadOnlySequence<T>.32

**Практическое задание: "Высокопроизводительный анализатор логов"**

* **Задача:** Написать парсер, который читает огромный лог-файл (500 МБ+) и подсчитывает количество строк, содержащих слово "ERROR".
* **Требования:**
  1. Использовать FileStream для заполнения PipeWriter.
  2. В PipeReader использовать SequenceReader<byte> для поиска символов перевода строки \n.10
  3. **Строгое условие:** Не преобразовывать байты в строки (Encoding.GetString) для всей строки. Сравнивать байты напрямую или использовать Span-ориентированные методы поиска.
  4. Сравнить потребление памяти (Diagnostic Tools) с реализацией через StreamReader.ReadLine.

**Протокол рефлексии:**

* "Объясни роль методов AdvanceTo в управлении памятью пайплайна. Что произойдет, если не освободить просмотренный буфер?"

### День 4: Реактивный мониторинг файловой системы

**Цель:** Создать надежную систему горячей перезагрузки конфигурации, устойчивую к спаму событий.

**Теория:** [**Теория Четвёртого Дня Пятой Недели**](https://docs.google.com/document/d/1YGh_gZBJFy0l4OU0GKc5mrTOuyih4dhta4RVXm3qG74/edit?usp=drive_link)

* Внутреннее устройство FileSystemWatcher и буфер событий ядра.22
* Реактивное программирование (Rx.NET): Observable, Throttle, Buffer.24
* Стратегии обработки ошибок доступа к файлам (Retry patterns).

**Практическое задание: "Сервис Hot-Reload Конфигурации"**

* **Задача:** Реализовать сервис, который следит за файлом appsettings.json и обновляет конфигурацию в памяти.
* **Требования:**
  1. Обернуть события FileSystemWatcher в Observable.FromEventPattern.33
  2. Применить оператор Throttle (или Debounce) с окном 500 мс для фильтрации дубликатов.34
  3. Реализовать чтение файла с использованием политики Retry (через библиотеку Polly), чтобы пережить моменты блокировки файла текстовым редактором.25
  4. Интегрировать обновление через IOptionsMonitor (симуляция обновления).

**Протокол рефлексии:**

* "Почему простого Thread.Sleep недостаточно для надежного чтения файла? Как гарантировать, что мы читаем последнюю версию файла, а не промежуточную?"

### День 5: Ротация логов и атомарные операции

**Цель:** Реализовать потокобезопасную систему логирования с ротацией файлов, применимую в Enterprise.

**Теория:** [**Теория пятого дня пятой недели**](https://docs.google.com/document/d/1O0ByuVudIuD4v93hhCHA5R6DFKLl0ObOSvFSDdlG6VU/edit?usp=drive_link)

* Атомарность операций файловой системы (NTFS Rename/Move).35
* Конкурентный доступ к файлам: FileShare.ReadWrite.
* Паттерн Producer-Consumer для записи логов (вынос I/O из потока приложения).

**Практическое задание: "Потокобезопасный логгер с ротацией"**

* **Задача:** Написать класс RotationLogger.
* **Требования:**
  1. Логи пишутся во внутреннюю ConcurrentQueue или Channel<string>.
  2. Фоновый поток (Task) вычитывает очередь и пишет в файл.
  3. **Ротация:** Когда файл достигает 1 МБ, текущий файл закрывается, переименовывается в log.1 (сдвигая старые log.1 -> log.2), и создается новый.
  4. Использовать File.Move для атомарной подмены файлов.36
  5. Обеспечить корректное закрытие и сброс буферов (Flush) при остановке приложения (IDisposable и IAsyncDisposable).

**Протокол рефлексии:**

* "Как избежать потери логов при крахе приложения? В чем преимущество использования Channel<T> перед BlockingCollection для асинхронного логирования?"

## 6. Заключение

Пятая неделя обучения является переломным моментом, маркирующим отход от синхронных, однопоточных паттернов, характерных для клиентской разработки игр. Предложенный учебный план фокусируется не просто на изучении API (какие методы вызывать), а на понимании стоимости и последствий этих вызовов для всей системы.

Освоение асинхронного ввода-вывода с правильной буферизацией, внедрение System.IO.Pipelines для оптимизации работы с памятью и использование паттерна Options для конфигурации — это навыки, которые напрямую влияют на такие метрики, как пропускная способность (throughput) и задержка (latency) веб-сервисов. Реализация утилиты envdir и надежного FileSystemWatcher дает понимание того, как приложение взаимодействует с внешним миром и операционной системой, что является критической компетенцией для архитектора распределенных систем. Успешное выполнение заданий этой недели создаст надежный фундамент для дальнейшего изучения баз данных и сетевых протоколов.

#### Источники

1. Разработка бэкенда: план обучения C#
2. c# - FileStream.ReadAsync very slow compared to Read() - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/39353119/filestream-readasync-very-slow-compared-to-read>
3. Asynchronous file access (C#) - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/using-async-for-file-access>
4. C# FileStream : Optimal buffer size for writing large files? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/1862982/c-sharp-filestream-optimal-buffer-size-for-writing-large-files>
5. FileStream performance guidelines · dotnet runtime · Discussion #74405 - GitHub, дата последнего обращения: декабря 3, 2025, <https://github.com/dotnet/runtime/discussions/74405>
6. FileStream C# (How It Works For Developers) - IronPDF, дата последнего обращения: декабря 3, 2025, <https://ironpdf.com/blog/net-help/filestream-csharp/>
7. High performance IO with System.IO.Pipelines - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/shows/on-dotnet/high-performance-io-with-systemiopipelines>
8. System.IO.Pipelines - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/io/pipelines>
9. c# - Performance of System.IO.File vs System.IO.FileStream - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/15764433/performance-of-system-io-file-vs-system-io-filestream>
10. Why is this System.IO.Pipelines code much slower than Stream-based code?, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/64283938/why-is-this-system-io-pipelines-code-much-slower-than-stream-based-code>
11. When to use memory-mapped files? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/1859213/when-to-use-memory-mapped-files>
12. Memory-Mapped Files - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files>
13. WTF is memory-mapped files? - Medium, дата последнего обращения: декабря 3, 2025, <https://medium.com/@ZaradarTR/wtf-is-memory-mapped-files-9448c04078a3>
14. Memory Mapped Files, File I/O & Performance - Ayende @ Rahien, дата последнего обращения: декабря 3, 2025, <https://ayende.com/blog/163138/memory-mapped-files-file-i-o-performance>
15. c# - .NET 4.0 Memory Mapped Files Performance - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/2333075/net-4-0-memory-mapped-files-performance>
16. Options pattern - .NET - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/options>
17. Options pattern in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-10.0>
18. Best practices for Configuration : r/dotnet - Reddit, дата последнего обращения: декабря 3, 2025, <https://www.reddit.com/r/dotnet/comments/1174sag/best_practices_for_configuration/>
19. envdir (Python port) — envdir 0.7 documentation, дата последнего обращения: декабря 3, 2025, <https://envdir.readthedocs.io/>
20. c# - Set environment variables for a process - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/14553830/set-environment-variables-for-a-process>
21. ProcessStartInfo.EnvironmentVariables Property (System.Diagnostics) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.processstartinfo.environmentvariables?view=net-10.0>
22. c# - FileSystemWatcher pitfalls - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/21000831/filesystemwatcher-pitfalls>
23. Monitoring File System Changes with FileSystemWatcher in .NET - dotnet-guide.com, дата последнего обращения: декабря 3, 2025, <https://www.dotnet-guide.com/filesystemwatcher-class.html>
24. C# event debounce - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/28472205/c-sharp-event-debounce>
25. Building Robust C# Applications with Polly Resilience Framework - DEV Community, дата последнего обращения: декабря 3, 2025, <https://dev.to/chakewitz/building-robust-c-applications-with-polly-resilience-framework-294h>
26. Implement HTTP call retries with exponential backoff with Polly - .NET | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly>
27. FileStream.Lock(Int64, Int64) Method (System.IO) | Microsoft Learn, дата последнего обращения: декабря 3, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream.lock?view=net-10.0>
28. c# FileStream.Lock what happens? - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/61963826/c-sharp-filestream-lock-what-happens>
29. C# Asynchronous File Copy with Progress Bar - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/68314485/c-sharp-asynchronous-file-copy-with-progress-bar>
30. Asynchronous File I/O in C# | Keyur Ramoliya, дата последнего обращения: декабря 3, 2025, <https://keyurramoliya.com/posts/Asynchronous-File-IO-in-CSharp/>
31. A process inherits its environment from its parent, and the consequences of this simple statement - The Old New Thing - Microsoft Developer Blogs, дата последнего обращения: декабря 3, 2025, <https://devblogs.microsoft.com/oldnewthing/20150915-00/?p=91591>
32. Mastering Advanced C# Features for Peak Code Performance - Goat Review, дата последнего обращения: декабря 3, 2025, <https://goatreview.com/mastering-advanced-csharp-features-code-performance/>
33. FileSystemWatcher Changed event is raised twice - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/1764809/filesystemwatcher-changed-event-is-raised-twice>
34. Debouncing and Throttling Dispatcher Events - Rick Strahl - West Wind Technologies, дата последнего обращения: декабря 3, 2025, <https://weblog.west-wind.com/posts/2017/jul/02/debouncing-and-throttling-dispatcher-events>
35. c# - File.Move atomic operation - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/15274861/file-move-atomic-operation>
36. How to move a directory in C# .NET in a single atomic operation - Stack Overflow, дата последнего обращения: декабря 3, 2025, <https://stackoverflow.com/questions/683369/how-to-move-a-directory-in-c-sharp-net-in-a-single-atomic-operation>