# Архитектурный анализ реактивных файловых систем и динамического управления конфигурацией в среде.NET Enterprise

## 1. Введение: Эволюция парадигм ввода-вывода в распределенных системах

Современная разработка программного обеспечения корпоративного уровня (Enterprise) переживает фундаментальный сдвиг в подходах к управлению данными и состоянием приложений. Если традиционные модели, характерные для разработки десктопных приложений или игровых движков, таких как Unity, опирались на детерминированные циклы обновлений и синхронный доступ к ресурсам, то архитектура высоконагруженных бэкенд-систем на платформе.NET диктует совершенно иные требования. В центре этого архитектурного сдвига находится концепция реактивности — способности системы асинхронно реагировать на внешние стимулы, сохраняя при этом отзывчивость, устойчивость и эластичность.

Четвертый день пятой недели учебного плана, посвященный "Работе с файловой системой и конфигурацией", представляет собой критически важный этап в трансформации мышления разработчика.1 Переход от императивной модели "запрос-ответ" к событийно-ориентированной модели взаимодействия с файловой системой требует глубокого понимания низкоуровневых механизмов операционной системы, ограничений управляемого кода (managed code) и паттернов проектирования, обеспечивающих надежность в условиях неопределенности. Файловая система в контексте серверной разработки перестает быть просто статичным хранилищем данных ("data at rest"); она превращается в активный источник потоковых событий ("data in motion"), генерирующий сигналы об изменениях конфигурации, ротации журналов событий или поступлении новых пакетов данных для обработки.

Данный отчет представляет собой исчерпывающее теоретическое исследование, призванное обеспечить необходимую базу для выполнения практических задач по созданию сервиса горячей перезагрузки конфигурации ("Hot-Reload Configuration Service") и утилиты мониторинга файловой системы. Мы проведем детальную деконструкцию механики взаимодействия.NET Runtime (CLR) с ядром операционных систем Windows и Linux, проанализируем математические модели реактивного программирования (Rx.NET) для фильтрации событийного шума и рассмотрим стратегии обеспечения устойчивости (Resiliency) при конкурентном доступе к ресурсам. Особое внимание будет уделено проблемам, возникающим в контейнеризированных средах (Docker/Kubernetes), где абстракции файловых систем часто протекают, создавая нетривиальные вызовы для архитекторов систем.

### 1.1. От Game Loop к Event Loop: Смена ментальных моделей

Для разработчика, имеющего бэкграунд в Unity, привычной является модель Game Loop, где состояние мира обновляется дискретно, кадр за кадром. В этой модели доступ к файловой системе часто рассматривается как блокирующая операция, допустимая лишь в строго определенных фазах жизненного цикла приложения (например, при загрузке уровня). Состояние конфигурации обычно иммутабельно (неизменяемо) в рамках сессии игры и загружается один раз при старте из префабов или ScriptableObject.1

В противовес этому, серверное приложение на базе ASP.NET Core функционирует в непрерывном цикле обработки запросов, где блокировка потока выполнения для ожидания ввода-вывода (I/O) является архитектурным преступлением, ведущим к исчерпанию пула потоков (ThreadPool Starvation) и деградации производительности всей системы. Более того, требования бизнеса к доступности сервисов (SLA) часто исключают возможность перезапуска приложения для применения новых настроек. Это порождает потребность в системах "горячей" перезагрузки (Hot Reload), способных на лету подхватывать изменения из файлов конфигурации (appsettings.json), переменных окружения или секретов, смонтированных в файловую систему контейнера.

Реализация такой функциональности требует выхода за рамки стандартных классов System.IO.File и погружения в асинхронную, событийно-ориентированную природу файловых систем. Мы должны научиться воспринимать файловую систему не как библиотеку, где книги стоят на полках, а как конвейер, по которому непрерывно движутся данные, и где попытка взять книгу может закончиться неудачей, если кто-то другой читает её в данный момент.

## 2. Физика файлового мониторинга: Глубокий анализ взаимодействия Kernel Space и User Space

Понимание того, как работают механизмы отслеживания изменений файлов "под капотом", является абсолютно необходимым для предотвращения критических сбоев в продакшене. Класс FileSystemWatcher в.NET — это не магический черный ящик, а тонкая обертка над нативными API операционных систем. Различия в реализации этих API между Windows и Linux определяют границы применимости и потенциальные точки отказа наших приложений.

### 2.1. Windows: Архитектура ReadDirectoryChangesW и управление буферами

В операционной системе Windows механизм мониторинга файловой системы базируется на функции Win32 API ReadDirectoryChangesW.2 Эта функция позволяет приложению запросить у ядра уведомления об изменениях в определенной директории. Принципиально важно понимать, что этот процесс является асинхронным и опирается на модель перекрывающего ввода-вывода (Overlapped I/O), интегрированную с портами завершения ввода-вывода (I/O Completion Ports - IOCP).

Когда мы создаем экземпляр FileSystemWatcher и устанавливаем свойство EnableRaisingEvents = true, CLR инициирует системный вызов, который регистрирует запрос в драйвере файловой системы (обычно NTFS). Драйвер выделяет специальный буфер в неподкачиваемой памяти ядра (Non-paged kernel memory) для хранения записей о происходящих событиях.

#### Проблема переполнения буфера (Buffer Overflow)

Центральной проблемой надежности FileSystemWatcher является ограниченный размер этого внутреннего буфера. По умолчанию в.NET он составляет всего 8 КБ (8192 байта).4 Каждое событие файловой системы (создание, удаление, изменение) занимает место в этом буфере. Структура записи уведомления (FILE\_NOTIFY\_INFORMATION) содержит:

* Смещение до следующей записи (4 байта).
* Код действия (Action) — например, FILE\_ACTION\_ADDED или FILE\_ACTION\_MODIFIED (4 байта).
* Длину имени файла в байтах (4 байта).
* Само имя файла в формате Unicode (переменная длина).

При массовых операциях с файловой системой, таких как выполнение команды git checkout, распаковка большого архива, или интенсивная ротация логов, драйвер файловой системы генерирует тысячи событий в секунду. Если управляемое приложение (Consumer) не успевает вычитывать и обрабатывать эти события из буфера с той же скоростью, с которой драйвер (Producer) их туда пишет, происходит переполнение буфера.6

В момент переполнения драйвер вынужден отбросить самые старые события, чтобы освободить место для новых, или, в худшем случае, прекратить запись. В управляемом коде это проявляется как событие Error с исключением InternalBufferOverflowException. Критичность ситуации заключается в том, что **система теряет информацию о том, какие именно файлы были изменены**. Приложение получает лишь сигнал о том, что "что-то произошло", но не знает, что именно, что делает состояние внутреннего кэша приложения (например, конфигурации) неконсистентным по отношению к реальному состоянию диска.2

#### Ограничения масштабируемости и сетевые протоколы

Казалось бы, решение лежит на поверхности: увеличить размер буфера через свойство InternalBufferSize. Однако здесь вступают в силу жесткие ограничения архитектуры ОС и сетевых протоколов.

Для локальных дисков увеличение буфера возможно, но следует помнить, что эта память выделяется из неподкачиваемого пула ядра. Если в системе запущены тысячи экземпляров FileSystemWatcher (например, в микросервисной архитектуре на одном хосте), чрезмерное увеличение буфера может привести к истощению ресурсов ядра и дестабилизации всей операционной системы.3

Для сетевых файловых систем (SMB/CIFS shares) ситуация еще более строгая. Протокол SMB накладывает ограничение на размер пакета уведомлений, который не может превышать 64 КБ.4 Попытка установить InternalBufferSize больше 64 КБ при наблюдении за сетевой папкой приведет к немедленному исключению ERROR\_INVALID\_PARAMETER. Это фундаментальное ограничение распределенных файловых систем, которое невозможно обойти на уровне прикладного кода, и оно требует архитектурных решений, таких как переход на поллинг или использование специализированных механизмов синхронизации.

### 2.2. Linux: Inotify и проблемы многопоточности

В среде Linux, которая является основной целевой платформой для развертывания.NET Core приложений в Docker и Kubernetes, реализация FileSystemWatcher базируется на подсистеме ядра inotify.2 Несмотря на то, что цель та же — мониторинг изменений, — внутренняя механика и связанные с ней проблемы кардинально отличаются от Windows.

#### Дефект потоковой модели.NET в Linux

Одной из самых обсуждаемых проблем реализации FileSystemWatcher в.NET Core для Linux является модель потоков. Текущая реализация создает отдельный поток и отдельный экземпляр inotify для каждой наблюдаемой директории при использовании рекурсивного мониторинга (IncludeSubdirectories = true).9

Представьте, что ваше приложение следит за корневой папкой проекта, содержащей глубокую вложенность директорий (например, node\_modules). Это приведет к созданию тысяч потоков операционной системы, каждый из которых будет блокировать ресурсы стека и планировщика CPU. Это "дорогое" удовольствие с точки зрения ресурсов ядра, которое может парализовать работу контейнера, ограниченного по CPU и памяти (CPU throttling/OOM Kill).

#### Лимиты fs.inotify.max\_user\_watches

Ядро Linux имеет встроенные механизмы защиты от истощения ресурсов, одним из которых является параметр fs.inotify.max\_user\_watches. Этот параметр определяет максимальное количество дескрипторов наблюдения ("watches"), которые может создать один пользователь (User ID).

Значение по умолчанию в многих дистрибутивах Linux и образах контейнеров часто бывает консервативным — например, 8192 или даже 128.10

Если.NET приложение пытается зарегистрировать наблюдение за количеством директорий, превышающим этот лимит, будет выброшено исключение IOException: The configured user limit (...) on the number of inotify instances has been reached.

В отличие от Windows, где проблема чаще всего заключается в пропускной способности (переполнение буфера), в Linux мы сталкиваемся с жестким лимитом на количество наблюдаемых объектов.

Решение этой проблемы требует вмешательства в настройки хост-системы через изменение конфигурации sysctl (например, fs.inotify.max\_user\_watches=524288).13 Однако в облачных средах (Azure App Service, AWS Fargate, Google Cloud Run) разработчик часто не имеет привилегий root для изменения параметров ядра хоста, что делает использование FileSystemWatcher для глубоких деревьев директорий невозможным.

### 2.3. Контейнеризация и проблема Docker Volumes

Самый сложный пласт проблем возникает на стыке технологий виртуализации и файловых систем. В современной разработке стандартным является сценарий, когда код редактируется на хост-системе (например, Windows или macOS), а выполняется внутри Docker-контейнера (Linux), куда исходный код проброшен через Volume (Bind Mount).

В этом сценарии события файловой системы должны пересечь границу между хост-ОС и гостевой ОС контейнера. Однако протоколы, используемые для шаринга файлов (часто это вариации CIFS/SMB или gRPC-FUSE в Docker Desktop), не всегда корректно транслируют уведомления inotify.15

В результате, разработчик сохраняет файл appsettings.json в IDE на Windows, но процесс.NET внутри контейнера "молчит", так как ядро Linux внутри контейнера не получило сигнала об изменении от гипервизора. Документация Microsoft прямо указывает, что FileSystemWatcher неэффективен для смонтированных дисков и сетевых шаров.18

Это ограничение приводит нас к необходимости использования гибридных подходов, комбинирующих событийную модель с периодическим опросом (поллингом), о чем будет подробно рассказано в разделе, посвященном PhysicalFileProvider.

### Таблица 1: Сравнительный анализ механизмов мониторинга файловых систем

| **Характеристика** | **FileSystemWatcher (Windows)** | **FileSystemWatcher (Linux)** | **Polling Strategy (Cross-Platform)** |
| --- | --- | --- | --- |
| **Базовый API ОС** | ReadDirectoryChangesW (интеграция с IOCP) | inotify | Периодический вызов stat / GetFileAttributes |
| **Модель памяти** | Non-paged kernel pool (Буфер) | Kernel memory (Descriptors) | User space memory (хранение состояния) |
| **Основной лимит** | Размер буфера (8KB - 64KB) | fs.inotify.max\_user\_watches | CPU usage (частота опроса) |
| **Потоковая модель** | Асинхронный I/O (ThreadPool) | Поток на директорию (Thread-per-directory) | Таймер в ThreadPool |
| **Латентность** | Близкая к реальному времени (<1 мс) | Низкая (<10 мс) | Высокая (зависит от интервала, по умолч. 4 сек) |
| **Надежность в Docker** | Работает (Windows Containers) | Проблемы с Bind Mounts | **Высокая (работает везде)** |
| **Типичная ошибка** | InternalBufferOverflowException | IOException (inotify limit reached) | Отсутствует (только задержка детекции) |

## 3. Реактивное программирование (Rx.NET): Укрощение хаоса событий

Сырой поток событий, генерируемый FileSystemWatcher (Changed, Created, Renamed), по своей природе является "шумным", дублирующимся и недетерминированным. Попытка обрабатывать эти события напрямую через стандартные C# события (+=) в высоконагруженной системе обречена на провал из-за состояний гонки и сложности синхронизации.

Здесь на сцену выходит Reactive Extensions (Rx.NET) — библиотека, позволяющая обрабатывать асинхронные потоки данных с использованием декларативных операторов LINQ.

### 3.1. От событий к потокам: Observable.FromEventPattern

Первый шаг в построении надежной системы мониторинга — отказ от императивной обработки событий в пользу функциональной реактивной парадигмы. Метод Observable.FromEventPattern позволяет преобразовать классическую пару методов add/remove обработчиков событий в объект IObservable<EventPattern<T>>.19

C#

var watcher = new FileSystemWatcher(path);  
var changes = Observable.FromEventPattern<FileSystemEventHandler, FileSystemEventArgs>(  
 h => watcher.Changed += h,  
 h => watcher.Changed -= h  
);

Это преобразование фундаментально меняет подход: мы перестаем реагировать на события "здесь и сейчас" и начинаем строить конвейер (pipeline) обработки, в котором события могут буферизироваться, фильтроваться, объединяться и трансформироваться во времени. Это позволяет отделить источник событий (Source) от бизнес-логики их обработки (Sink).

### 3.2. Throttle vs Debounce: Терминологическая ловушка Rx.NET

В теории реактивного программирования существуют два основных паттерна ограничения частоты событий: Throttling (дросселирование) и Debouncing (устранение дребезга). Критически важно понимать разницу между ними и специфику их именования в библиотеке Rx.NET, которая отличается от общепринятой в индустрии (например, в RxJS).

**Семантическая инверсия:**

* **В индустрии (RxJS, Lodash):**
  + Debounce: Игнорировать события, пока они идут непрерывным потоком. Срабатывать только после того, как наступила "тишина" в течение заданного времени. Это аналог того, как работает человеческий глаз при чтении текста — мы не осознаем каждую букву отдельно, а воспринимаем слово целиком после завершения его написания.
  + Throttle: Гарантировать срабатывание события не чаще, чем раз в заданный интервал времени (например, раз в 100 мс), даже если поток событий непрерывен. Это аналог пулемета с ограничением скорострельности.21
* **В Rx.NET:**
  + Оператор, реализующий логику **Debounce**, называется **Throttle**.23 Это историческое наследие, которое часто сбивает с толку разработчиков.
  + Для реализации логики Throttling (Sample) используется оператор Sample.

Применение к файловой системе:

Для задачи отслеживания конфигурации нам необходим именно паттерн Debounce (то есть оператор Throttle в Rx.NET).

Когда текстовый редактор (например, VS Code) сохраняет файл, он делает это не атомарно. Он может выполнить серию записей: создать временный файл, записать контент, сбросить буферы, изменить атрибуты времени, переименовать файл. Это генерирует пачку событий за короткий промежуток времени (10-50 мс).

Если мы начнем читать файл при первом событии, мы рискуем прочитать частично записанный файл или столкнуться с блокировкой. Нам нужна последняя версия файла, когда активность полностью прекратилась.

Использование Throttle(TimeSpan.FromMilliseconds(500)) создает "скользящее окно": таймер сбрасывается с каждым новым событием. Сигнал пройдет дальше только тогда, когда в течение 500 мс не будет новых событий. Это значительно повышает вероятность того, что файловый дескриптор уже освобожден пишущим процессом.25

### 3.3. Группировка и изоляция контекстов: GroupBy

В Enterprise-системах мы часто наблюдаем не за одним файлом, а за целой директорией конфигураций. Наивное применение Throttle к общему потоку событий от FileSystemWatcher приведет к логической ошибке: активное редактирование файла logging.json будет постоянно сбрасывать таймер Throttle и блокировать обработку изменений в файле database.json, если они происходят параллельно.

Для решения этой проблемы необходимо применять оператор GroupBy, который разделяет единый поток событий на множество независимых подпотоков, ключом для которых является полный путь к файлу (e.FullPath).27

Архитектурный паттерн выглядит следующим образом:

1. **Merge:** Объединяем потоки Created, Changed и Renamed в единый поток IObservable<FileSystemEventArgs>.
2. **GroupBy:** Группируем события по имени файла. Результатом является поток потоков (IObservable<IGroupedObservable<string, FileSystemEventArgs>>).
3. **SelectMany + Throttle:** Для каждой группы применяем индивидуальный Throttle. Это гарантирует, что таймеры "тишины" для разных файлов независимы друг от друга.
4. **Subscribe:** Подписываемся на итоговый поток уже "чистых", дебаунсированных событий.

Такой подход обеспечивает изоляцию отказов и задержек: проблема с одним файлом не влияет на мониторинг других.

### Таблица 2: Влияние операторов Rx.NET на поток файловых событий

| **Сценарий (сохранение файла)** | **Без Rx (Raw Events)** | **Rx Sample (Interval)** | **Rx Throttle (Debounce)** |
| --- | --- | --- | --- |
| **0ms:** Событие 1 (Start Write) | Вызов обработчика | Запуск таймера | Запуск таймера (сброс) |
| **10ms:** Событие 2 (Write Content) | Вызов обработчика | Игнорирование | Сброс таймера (restart) |
| **20ms:** Событие 3 (Attributes) | Вызов обработчика | Игнорирование | Сброс таймера (restart) |
| **500ms:** (Тишина) | - | Вызов (если интервал истек) | **Вызов обработчика** |
| **Результат** | 3 чтения, 2 ошибки блокировки | 1 чтение (промежуточное) | **1 чтение (финальное)** |

## 4. Паттерны устойчивости (Resiliency): Работа с блокировками и гонками

Даже идеальная настройка реактивного конвейера с использованием Rx.NET не дает 100% гарантии, что в момент попытки чтения файла (File.ReadAllText) он будет доступен. Файловая система — это разделяемый ресурс, и другие процессы (антивирусы, агенты резервного копирования, системы индексации) могут заблокировать файл в любой момент.

В мире Enterprise-разработки исключение IOException при работе с файлами следует рассматривать не как "ошибку" (Error), а как ожидаемое "состояние" (Condition), которое система должна уметь обрабатывать штатно.

### 4.1. Анатомия Sharing Violations и HRESULT

В экосистеме.NET класс IOException является общим контейнером для широкого спектра ошибок ввода-вывода, от физических сбоев диска до логических конфликтов доступа. Для построения надежной системы автоматического восстановления (retry logic) критически важно уметь отличать фатальные ошибки (файл не найден, диск переполнен) от временных (файл заблокирован).

Идентификация блокировки файла (Sharing Violation) производится через анализ свойства HResult исключения. Ключевые коды ошибок, пришедшие из мира Win32 API:

* **0x80070020 (ERROR\_SHARING\_VIOLATION):** "The process cannot access the file because it is being used by another process". Это классическая ошибка блокировки.
* **0x80070021 (ERROR\_LOCK\_VIOLATION):** "The process cannot access the file because another process has locked a portion of the file". Возникает при блокировке диапазонов байтов.28

В современных версиях C# (начиная с 6.0) мы можем использовать фильтры исключений (when) для точечного перехвата именно этих состояний, не затрагивая другие типы IOException:

C#

catch (IOException ex) when ((ex.HResult & 0x0000FFFF) == 32)  
{  
 // Логика обработки блокировки (retry)  
}

Использование битовой маски 0x0000FFFF необходимо для извлечения кода ошибки из структуры HRESULT.29

### 4.2. Политики Retry с использованием Polly: Математика надежности

Для реализации логики повторных попыток в.NET стандартом де-факто является библиотека **Polly**. Однако простого "попробуй 3 раза" недостаточно для высоконагруженной системы. Необходим научно обоснованный подход к стратегии повторов.

Экспоненциальная задержка (Exponential Backoff):

Если файл заблокирован, немедленная повторная попытка скорее всего тоже провалится и лишь увеличит нагрузку на систему (конфликт за блокировки, CPU spin). Эффективная стратегия заключается в увеличении интервала ожидания после каждой неудачи.

Формула задержки обычно выглядит как $Delay = BaseDelay \times 2^{Attempt}$.

Это дает "передышку" системе и позволяет конкурирующему процессу завершить свою работу.31

Jitter (Дрожание):

В системах с высокой конкурентностью (например, 10 экземпляров микросервиса пытаются прочитать один и тот же файл на сетевой шаре) синхронные ретраи могут привести к проблеме "синхронизированного шторма" (Thundering Herd). Если все 10 процессов получат отказ одновременно и подождут ровно 2 секунды, через 2 секунды они снова столкнутся лбами.

Добавление случайной компоненты (Jitter) к времени ожидания позволяет рассинхронизировать попытки процессов, статистически уменьшая вероятность коллизий.31

Пример интеграции реактивного потока и Polly: поток Rx инициирует событие изменения, подписчик вызывает метод загрузки конфигурации, который внутри себя обернут в Policy.Execute. Если Polly исчерпывает лимит попыток (например, файл заблокирован "мертво"), система должна корректно залогировать инцидент, но не должна падать с необработанным исключением (Fail-safe design).32

## 5. Архитектура управления конфигурацией: Паттерн Options и Hot Reload

Четвертый день обучения также охватывает вопрос того, как правильно интегрировать динамически считываемую конфигурацию в систему внедрения зависимостей (Dependency Injection) приложения. Просто прочитать JSON-файл недостаточно — необходимо безопасно обновить состояние приложения без перезапуска.

### 5.1. Иерархия интерфейсов Options: IOptions, IOptionsSnapshot, IOptionsMonitor

Платформа.NET Core предлагает строгую иерархию интерфейсов для работы с конфигурацией, выбор между которыми зависит от требуемого жизненного цикла и производительности.1

1. **IOptions<T> (Singleton):**
   * Считывает значение один раз при первом обращении.
   * Кэшируется на все время жизни приложения.
   * **Не поддерживает** обновление. Идеален для статических настроек, которые никогда не меняются.
2. **IOptionsSnapshot<T> (Scoped):**
   * Создается заново для каждого Scope (обычно — для каждого HTTP-запроса).
   * Считывает актуальное состояние конфигурации в момент создания.
   * Обеспечивает консистентность данных в рамках одного запроса: даже если файл изменится посередине обработки запроса, Snapshot будет хранить старую версию до конца запроса.
   * **Накладные расходы:** Создание снимка при каждом запросе может создавать нагрузку на GC и CPU, если конфигурация велика.
3. **IOptionsMonitor<T> (Singleton):**
   * Существует в единственном экземпляре, но предоставляет свойство CurrentValue, которое всегда возвращает последнюю версию.
   * Позволяет подписаться на изменения через OnChange(Action<T>).
   * Это ключевой интерфейс для реализации **Hot Reload** в синглтон-сервисах.36

### 5.2. Антипаттерн Captive Dependency (Захваченная зависимость)

Одной из самых коварных архитектурных ошибок при работе с динамической конфигурацией является "Captive Dependency". Это происходит, когда сервис с более длительным жизненным циклом (Singleton) зависит от сервиса с более коротким циклом (Scoped), или неправильно использует Scoped-зависимость.

Представим Singleton-сервис CacheService, который в конструкторе принимает IOptionsSnapshot<CacheConfig>.

В момент старта приложения DI-контейнер создаст экземпляр CacheService и внедрит в него текущий снимок настроек. Так как CacheService никогда не пересоздается, он будет вечно хранить ссылку на этот первый снимок. Даже если IOptionsSnapshot технически поддерживает обновление, внутри синглтона он "заморожен" навечно. Приложение теряет способность реагировать на изменения конфигурации, хотя разработчик уверен в обратном.

Для Singleton-сервисов единственно верным решением является использование IOptionsMonitor<T>. Он не хранит данные, а предоставляет доступ к ним через прокси или механизм подписки, что позволяет получать свежие данные при каждом обращении к CurrentValue.38

В.NET 6+ был введен механизм валидации областей видимости (Scope Validation), который выбрасывает исключение при попытке внедрить Scoped в Singleton, но понимание этой механики на концептуальном уровне обязательно для Senior-разработчика.38

### 5.3. Внутренняя механика reloadOnChange и PhysicalFileProvider

Когда в Program.cs мы конфигурируем приложение с помощью AddJsonFile("appsettings.json", reloadOnChange: true), мы запускаем сложную цепочку взаимодействий 41:

1. Создается экземпляр PhysicalFileProvider, ответственный за доступ к файлам.
2. PhysicalFileProvider создает FileSystemWatcher (по умолчанию) для наблюдения за указанным файлом.
3. При срабатывании события изменения PhysicalFileProvider генерирует сигнал через IChangeToken.
4. ConfigurationBuilder, подписанный на этот токен, инициирует полную перезагрузку дерева конфигурации.

#### Поллинг как стратегия выживания в Docker

Как мы выяснили в разделе 2, FileSystemWatcher может не работать в Docker-контейнерах при использовании bind mounts. Если полагаться только на стандартное поведение, Hot Reload работать не будет.

Разработчики.NET предусмотрели механизм "аварийного переключения" (fallback).

PhysicalFileProvider умеет считывать переменную окружения DOTNET\_USE\_POLLING\_FILE\_WATCHER.18

Если эта переменная установлена в 1 или true, провайдер отказывается от создания FileSystemWatcher и переключается на использование PollingFileChangeToken.

Этот механизм запускает таймер, который каждые 4 секунды (интервал по умолчанию) проверяет метаданные файла (обычно LastWriteTimeUtc).45

Да, это увеличивает латентность реакции на изменения до 4 секунд и создает постоянную, хоть и небольшую, нагрузку на CPU/Disk IO. Однако, в распределенных системах надежность (Reliability) важнее мгновенной реакции. Это гарантирует, что конфигурация обновится даже на самых экзотических файловых системах (NFS, SMB, Docker Volumes).

В учебном задании 4-го дня студентам предстоит реализовать или сконфигурировать именно такой гибридный подход: приоритет событийной модели с возможностью деградации до поллинга через конфигурацию среды.

## 6. Синтез и Заключение

Теоретический базис четвертого дня пятой недели объединяет глубокие знания системного программирования с паттернами проектирования высокого уровня. Мы прошли путь от байтовых буферов ядра Windows до абстракций Dependency Injection в.NET.

Итоговая архитектура надежного сервиса конфигурации, которую студент должен быть способен спроектировать и реализовать, выглядит как многослойная система защиты от неопределенности:

1. **Слой ввода-вывода (I/O Layer):** Использование PhysicalFileProvider с поддержкой переменной DOTNET\_USE\_POLLING\_FILE\_WATCHER для обеспечения кросс-платформенной совместимости и работы в Docker.
2. **Реактивный слой (Reactive Layer):** Применение Rx.NET с операторами FromEventPattern, Merge, GroupBy и Throttle (Debounce) для нормализации хаотичного потока событий файловой системы и устранения дубликатов.
3. **Слой устойчивости (Resiliency Layer):** Обертывание операций чтения файлов в политики Polly с экспоненциальной задержкой (Retry with Exponential Backoff) и фильтрацией по HRESULT для корректной обработки транзиентных блокировок файлов.
4. **Слой приложения (Application Layer):** Использование паттерна IOptionsMonitor для потокобезопасного распространения обновлений конфигурации в Singleton-сервисах и предотвращения проблемы Captive Dependency.

Освоение этого материала переводит разработчика на качественно новый уровень: от написания кода, который работает в "идеальных условиях", к проектированию систем, способных выживать и корректно функционировать в агрессивной, асинхронной и ненадежной среде реального Enterprise-продакшена. Это и есть суть инженерной зрелости, к которой ведет данный курс обучения.

#### Источники

1. План обучения работе с файлами и конфигурацией
2. FileSystemWatcher Class (System.IO) - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?view=net-10.0>
3. How does System.IO.FileSystemWatcher works under the hood? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/71295028/how-does-system-io-filesystemwatcher-works-under-the-hood>
4. System.IO.FileSystemWatcher class - .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-io-filesystemwatcher>
5. FileSystemWatcher.InternalBufferSize Property (System.IO) | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher.internalbuffersize?view=net-10.0>
6. c# - FileSystemWatcher InternalBufferOverflow - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/22116374/filesystemwatcher-internalbufferoverflow>
7. What are practical limits on the number of FileSystemWatcher instances a server can handle? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/10195317/what-are-practical-limits-on-the-number-of-filesystemwatcher-instances-a-server>
8. Changing InternalBufferSize on FileSystemWatcher for a network path - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/33119881/changing-internalbuffersize-on-filesystemwatcher-for-a-network-path>
9. FileSystemWatcher on Linux uses an excessive amount of resources · Issue #62869 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/62869>
10. FileSystemWatcher may cause problems in containers - inotify limits and incorrect error message · Issue #27272 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/27272>
11. How to resolve inotify watcher limits during installation? - ITRS Documentation Home Page, дата последнего обращения: декабря 4, 2025, <https://docs.itrsgroup.com/docs/itrs-analytics/latest/faqs/increase-fs-inotify-max-user-instances/index.html>
12. .NET Core: The configured user limit (128) on the number of inotify instances has been reached. | Travis J. Gosselin, дата последнего обращения: декабря 4, 2025, <https://travisgosselin.com/configured-user-limit-inotify-instances/>
13. Why do I get "The configured user limit (128) on the number of inotify instances has been reached" for MySQL queries from ASP.NET Core MVC [closed] - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/43469400/why-do-i-get-the-configured-user-limit-128-on-the-number-of-inotify-instances>
14. FileSystemWatcher intermittently throws an exception on Linux - Meziantou's blog, дата последнего обращения: декабря 4, 2025, <https://www.meziantou.net/filesystemwatcher-intermittently-throws-an-exception-on-linux.htm>
15. FileSystemWatcher doesn't work with CIFS on Linux · Issue #76397 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/76397>
16. FileSystemWatcher and network shares · Issue #16924 · dotnet/runtime - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/issues/16924>
17. File system watch does not work with mounted volumes - Docker Community Forums, дата последнего обращения: декабря 4, 2025, <https://forums.docker.com/t/file-system-watch-does-not-work-with-mounted-volumes/12038>
18. PhysicalFileProvider.UsePollingFileWatcher Property (Microsoft.Extensions.FileProviders), дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.fileproviders.physicalfileprovider.usepollingfilewatcher?view=net-10.0-pp>
19. Observable.FromEventPattern discussed - My Memory, дата последнего обращения: декабря 4, 2025, <https://putridparrot.com/blog/observable-fromeventpattern-discussed/>
20. Observable.FromEvent(TDelegate, TEventArgs) Method (Func(Action(TEventArgs), TDelegate), Action(TDelegate), Action(TDelegate)) (System.Reactive.Linq) | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/previous-versions/dotnet/reactive-extensions/hh229241(v=vs.103)>
21. Difference between throttling and debouncing a function - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/25991367/difference-between-throttling-and-debouncing-a-function>
22. What's the difference between throttle and debounce in Rxswift3.0? - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/43888296/whats-the-difference-between-throttle-and-debounce-in-rxswift3-0>
23. Debounce incorrectly named throttle #1471 - dotnet/reactive - GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/reactive/issues/1471>
24. Debouncing and Throttling Dispatcher Events - Rick Strahl - West Wind Technologies, дата последнего обращения: декабря 4, 2025, <https://weblog.west-wind.com/posts/2017/jul/02/debouncing-and-throttling-dispatcher-events>
25. C# event debounce - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/28472205/c-sharp-event-debounce>
26. Observe File System Changes with Reactive Extensions for .NET - Endjin, дата последнего обращения: декабря 4, 2025, <https://endjin.com/blog/2024/05/observe-file-system-changes-with-rx-dotnet>
27. How to improve this Rx FolderWatcher? [closed] - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/28996172/how-to-improve-this-rx-folderwatcher>
28. How to tell if a caught IOException is caused by the file being used by another process, without resorting to parsing the exception's Message property - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/2568875/how-to-tell-if-a-caught-ioexception-is-caused-by-the-file-being-used-by-another>
29. How do I determine if an IOException is thrown because of a sharing violation?, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/425956/how-do-i-determine-if-an-ioexception-is-thrown-because-of-a-sharing-violation>
30. Handling I/O errors in .NET - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/io/handling-io-errors>
31. Implement HTTP call retries with exponential backoff with Polly - .NET | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly>
32. Deleting a file throws exception when is use, despite of using Polly - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/71257113/deleting-a-file-throws-exception-when-is-use-despite-of-using-polly>
33. Exception handling policies with Polly | My Memory, дата последнего обращения: декабря 4, 2025, <https://putridparrot.com/blog/exception-handling-policies-with-polly/>
34. Polly policy to log exception and rethrow - Stack Overflow, дата последнего обращения: декабря 4, 2025, <https://stackoverflow.com/questions/42952057/polly-policy-to-log-exception-and-rethrow>
35. IOptions vs IOptionsSnapshot vs IOptionsMonitor in .NET — What's the Difference?, дата последнего обращения: декабря 4, 2025, <https://medium.com/c-sharp-programming/ioptions-vs-ioptionssnapshot-vs-ioptionsmonitor-dotnet-6f2305b09770>
36. NET Core Options Pattern For Live Reload Settings - Ingo Payments, дата последнего обращения: декабря 4, 2025, <https://ingomoney.com/developer-blog/net-core-options-pattern-for-live-reload-settings/>
37. Real-Time Config Updates with IOptionsMonitor .NET | CodeNx - Medium, дата последнего обращения: декабря 4, 2025, <https://medium.com/codenx/real-time-configuration-updates-in-asp-net-core-with-live-loading-of-appsettings-json-d63eac388d28>
38. Dependency injection guidelines - .NET | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines>
39. I don't understand why shouldn't every service be a singleton when using the generic host : r/dotnet - Reddit, дата последнего обращения: декабря 4, 2025, <https://www.reddit.com/r/dotnet/comments/1epf2ro/i_dont_understand_why_shouldnt_every_service_be_a/>
40. .NET Core and DI – Beware of Captive Dependency – Hi, I'm Ankit!, дата последнего обращения: декабря 4, 2025, <https://ankitvijay.net/2020/03/17/net-core-and-di-beware-of-captive-dependency/>
41. Detect changes with change tokens in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/change-tokens?view=aspnetcore-10.0>
42. Configuration in ASP.NET Core - Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-10.0>
43. dotnet watch command - .NET CLI | Microsoft Learn, дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-watch>
44. runtime/src/libraries/Microsoft.Extensions.FileProviders.Physical/src/PhysicalFileProvider.cs at main · dotnet/runtime · GitHub, дата последнего обращения: декабря 4, 2025, <https://github.com/dotnet/runtime/blob/main/src/libraries/Microsoft.Extensions.FileProviders.Physical/src/PhysicalFileProvider.cs>
45. PollingFileChangeToken Class (Microsoft.Extensions.FileProviders.Physical), дата последнего обращения: декабря 4, 2025, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.fileproviders.physical.pollingfilechangetoken?view=net-10.0-pp>