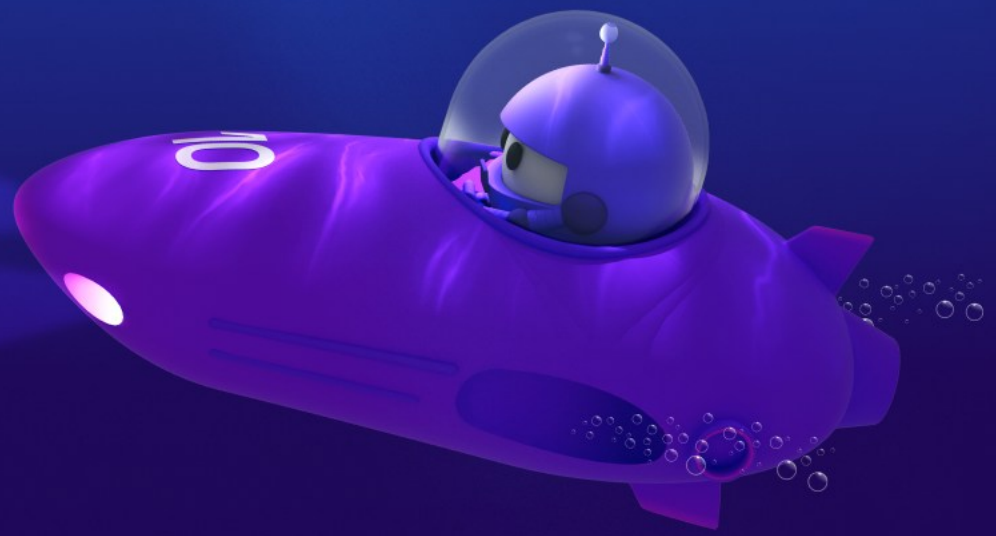
# Oltre HybridCache: FusionCache

Jody Donetti



**DotNetConf Liguria 2025**

# Jody Donetti

**Code + R&D**

Faccio cose (principalmente) sul web da circa 30 anni.

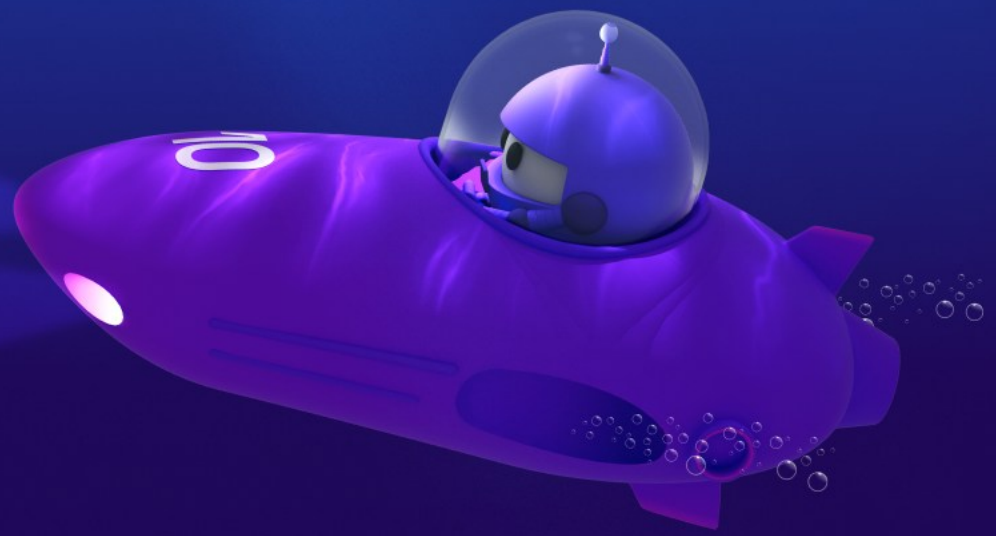
Ho avuto a che fare con la maggior parte dei tipi di cache: memory, distributed, hybrid, HTTP, offline e CDN.

Ho creato FusionCache, una hybrid cache .NET free + OSS.

🏆 Google OSS Award

🏆 Microsoft MVP Award

# Hybrid Caching: Dove Eravamo Rimasti



# 📌 Hybrid Caching: Dove Eravamo Rimasti

Quindi, ricapitolando.

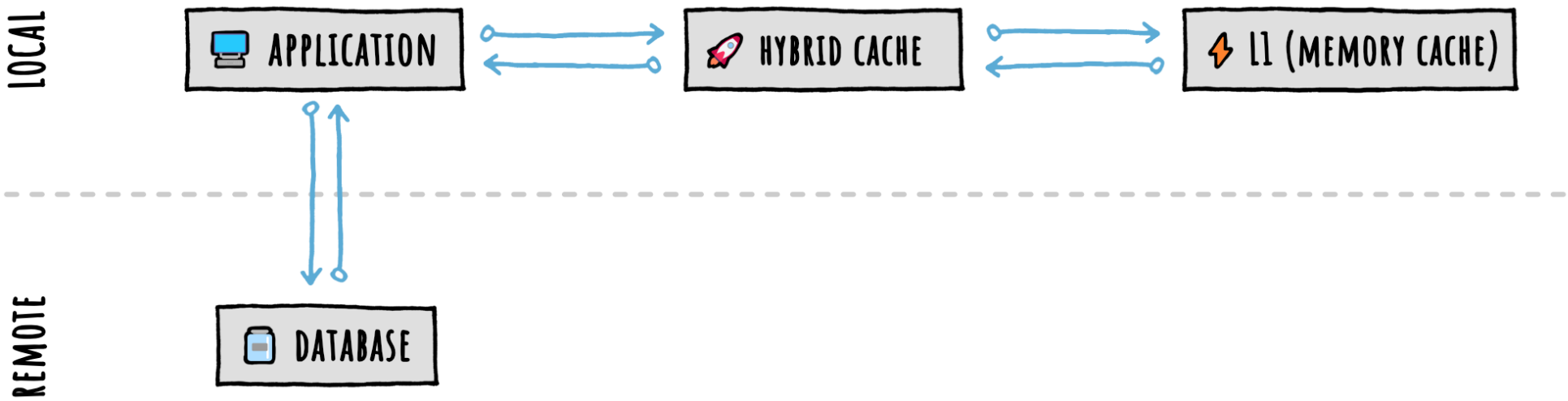
Tramite una cache ibrida possiamo ottenere il meglio di entrambi i mondi:

- caching in **memoria**
- caching **distribuito**

Per la parte in memoria abbiamo **L1** (primo livello).
Per la parte distribuita abbiamo **L2** (secondo livello).

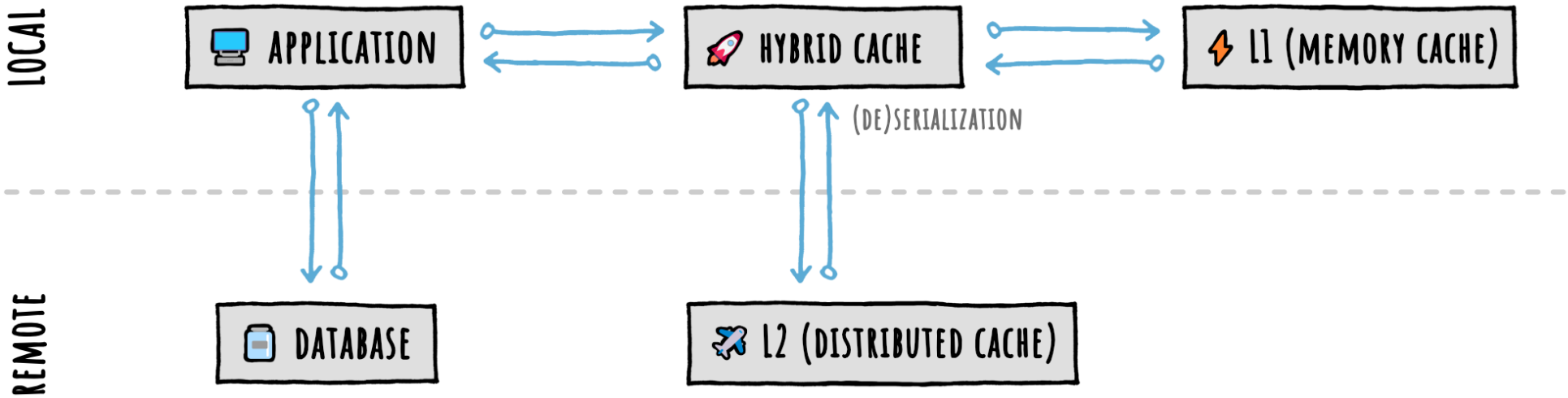
# Hybrid Caching: Dove Eravamo Rimasti

Non siamo obbligati ad usare **2 livelli**, possiamo usarne anche **solo uno** in **memoria** (**L1**):



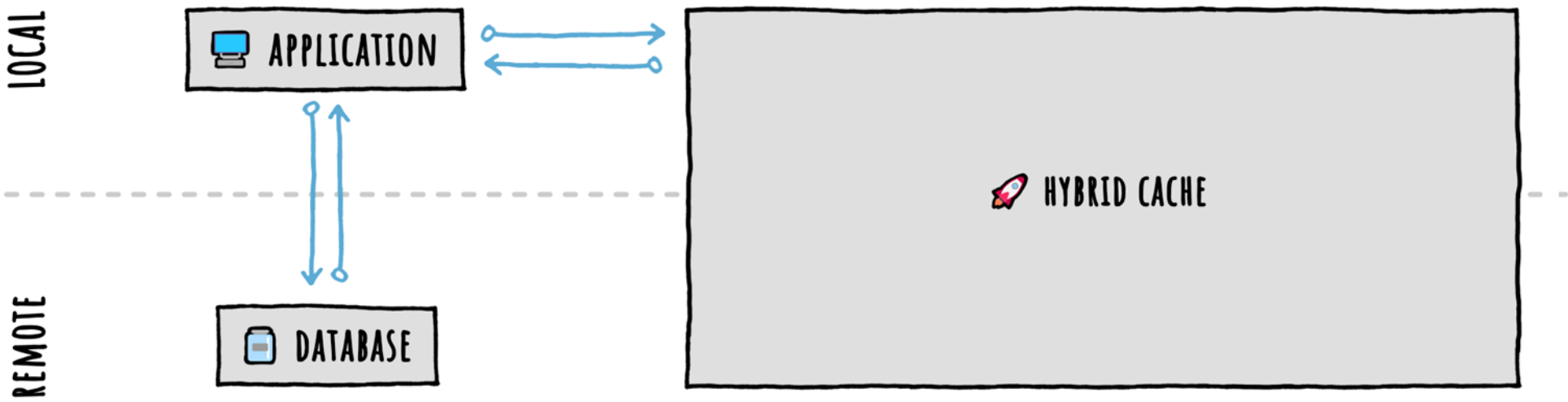
# Hybrid Caching: Dove Eravamo Rimasti

E in qualunque momento **abilitare**, anche **dinamicamente**, il livello **distribuito** (**L2**):



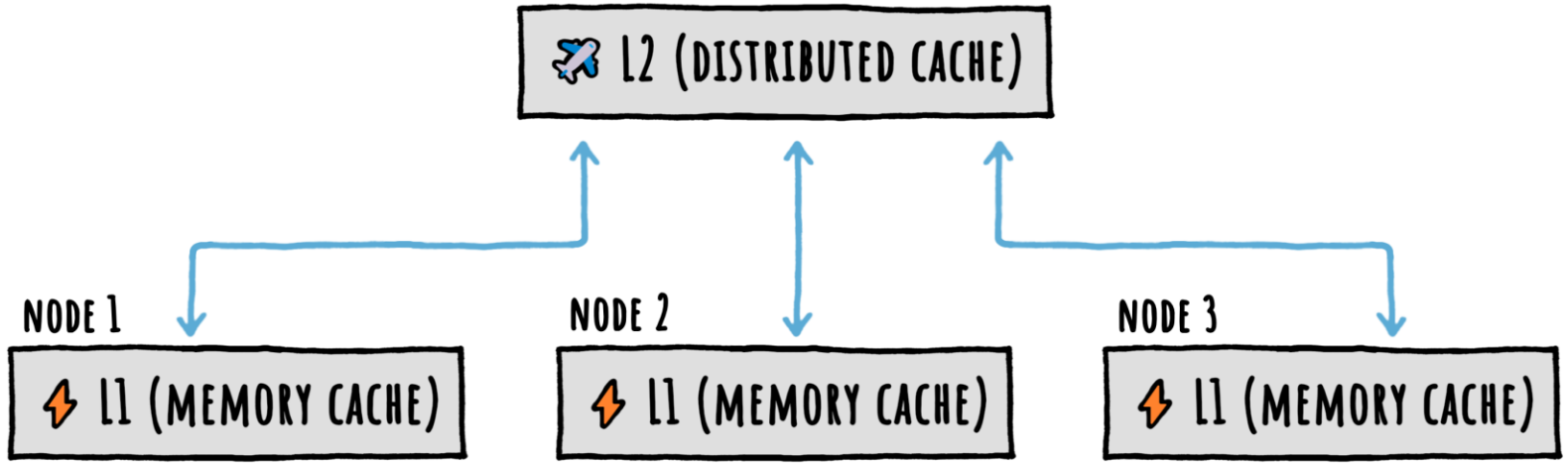
# Hybrid Caching: Dove Eravamo Rimasti

Il tutto **senza cambiare** il nostro codice dovunque, grazie ad una **API unificata**:



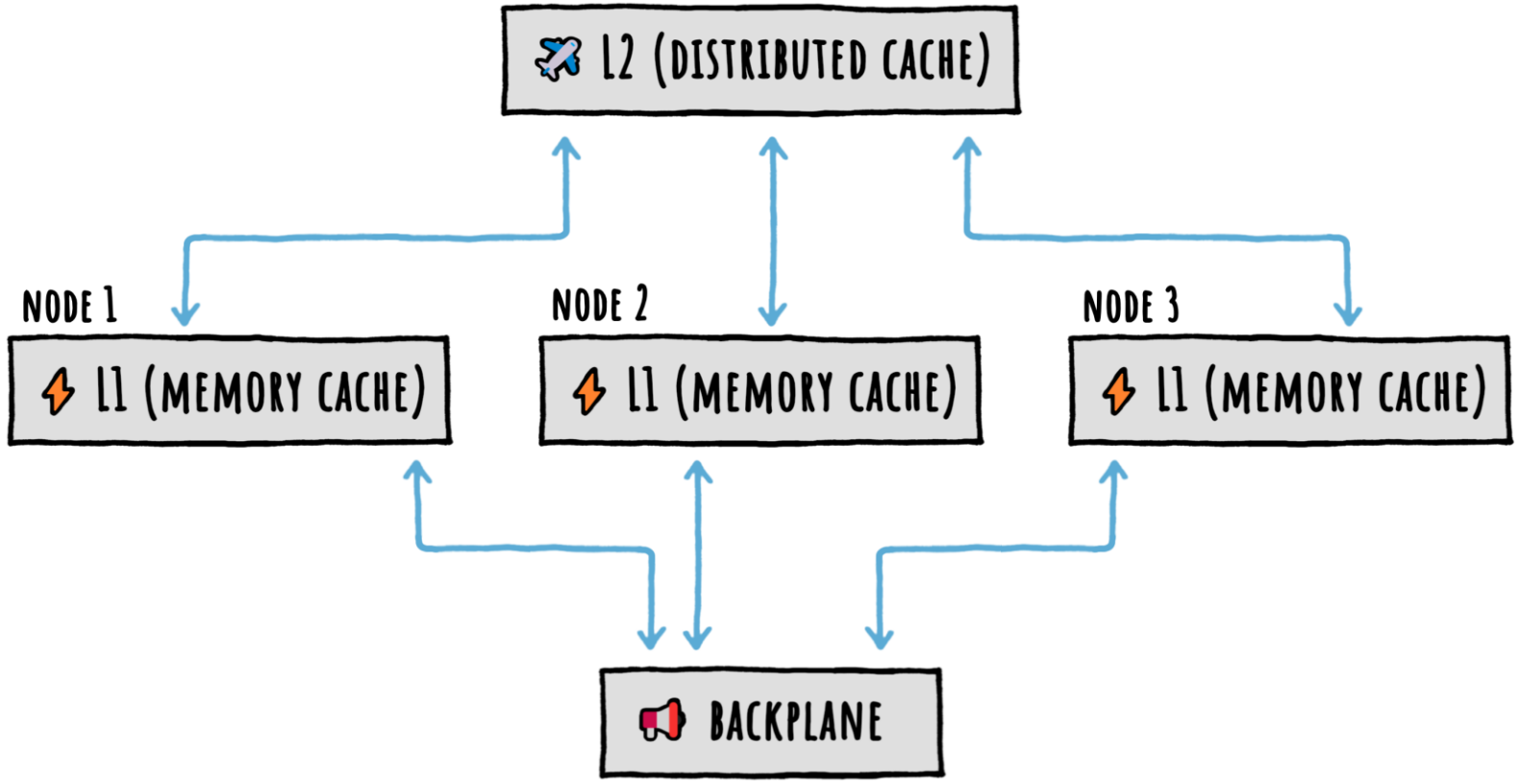
# Hybrid Caching: Dove Eravamo Rimasti

Potremmo poi dover **scalare orizzontalmente**, con scenari **multi-nodo** (L1 multiple):



# Hybrid Caching: Dove Eravamo Rimasti

In tal caso possiamo semplicemente aggiungere un **backplane**:



# 🔖 Hybrid Caching: Dove Eravamo Rimasti

Così facendo la **coerenza** verrà gestita automaticamente tramite **notifiche distribuite**.

E, di nuovo, non dovremmo cambiare il nostro codice dovunque.

Not bad.

# 📌 Hybrid Caching: Dove Eravamo Rimasti

Quali librerie possiamo usare?

📦 **CacheTower (multi-level)**
github.com/TurnerSoftware/CacheTower

📦 **CacheManager (multi-level)**
github.com/MichaCo/CacheManager

📦 **EasyCaching (multi-level)**
github.com/dotnetcore/EasyCaching

📦 **FusionCache (hybrid)**
github.com/ZiggyCreatures/FusionCache

Ⓜ️ **Microsoft HybridCache (hybrid)**

# 📌 Hybrid Caching: Dove Eravamo Rimasti

Quali librerie possiamo usare?

📦 **CacheTower (multi-level)**
github.com/TurnerSoftware/CacheTower

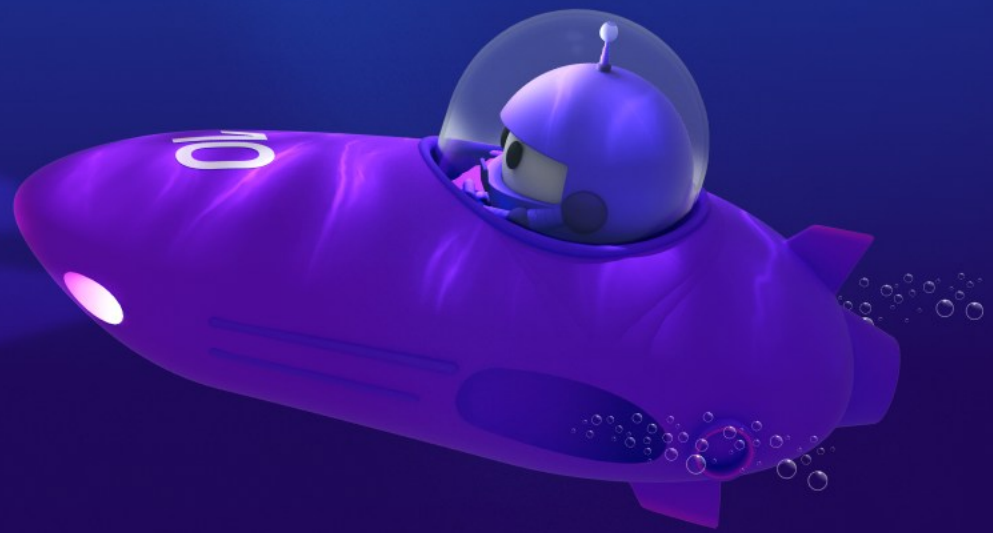
📦 **CacheManager (multi-level)**
github.com/MichaCo/CacheManager

📦 **EasyCaching (multi-level)**
github.com/dotnetcore/EasyCaching

📦 **FusionCache (hybrid)**
github.com/ZiggyCreatures/FusionCache

Ⓜ️ **Microsoft HybridCache (hybrid)**

# Tagging



# 🏷️ Tagging

Questa feature è **spettacolare**.

Immaginiamo di aver bisogno, dopo aver aggiornato un dato, di dover «allineare» la cache.

Normalmente possiamo fare una semplice `Remove(key)`, fine.

Questo però funziona **solo** se il dato modificato è riflesso in una **singola cache entry**.

# 🏷️ Tagging

Ma cosa succede se:

- le entry impattate sono molteplici, con **diverse cache key**?
- le entry impattate **non hanno** un **riferimento** nella loro **cache key**?

Fondamentalmente: come possiamo operare su più cache entry in contemporanea?

Possiamo usare i **tag**.

E come?

# 🏷️ Tagging

Quando **scriviamo** nella cache, **assegnamo** uno o più **tag**:

```
cache.Set<int>("foo", 1, tags: ["tag-1", "tag-2"]);
cache.Set<int>("bar", 2, tags: ["tag-2", "tag-3"]);
cache.GetOrSet<int>("baz", _ => 3, tags: ["tag-1", "tag-3"]);
```

# 🏷️ Tagging

Successivamente, eseguiamo una `RemoveByTag(tag)`:


Fine, fatto, non serve altro.

# 🏷️ Tagging

In ambito caching è una delle feature più potenti in assoluto.

E fra tutte le librerie di caching a disposizione, solo 2 supportano Tagging:

✅ **FusionCache**

✅ **HybridCache (*)**

Per entrambe le librerie il funzionamento è lo stesso, quello che abbiamo visto poco fa.

Il design interno e l'implementazione sono invece diverse fra di loro.

# 🏷️ Tagging

Come abbiamo visto, a livello **logico** e come **utilizzo** la feature è dannatamente **semplice**.

Corrisponderebbe in un database tradizionale ad una query di questo tipo (pseudo codice):

```
DELETE FROM table WHERE ‘tag-1’ IN tags
```

In realtà, sotto sotto, questa feature è **monumentale**.

Per comprendere appieno il **perchè**, dobbiamo considerare alcuni **aspetti**.

# 🏷️ Tagging

Le cache ibride usano le normali cache memory/distributed come **building blocks**:

Ossia esse sono i livelli sottostanti, su cui poter costruire:

- **L1:** memory cache
- **L2:** distributed cache

Se queste cache non supportano nativamente una feature, è difficile andare oltre.

# 🏷️ Tagging

Consideriamo che tipicamente le cache (sia memory che distributed):

- **non permettono** di lavorare in modo **massivo**, su più cache key
- **non permettono** di eseguire **query** come `DELETE FROM table WHERE 'tag-1' IN tags`
- permettono di lavorare **solo by-key**: la key è sempre il punto d'ingresso
- **nessuna cache** (memory o distributed) supporta il concetto di **tag**
- una cache ibrida può essere composta da **una L2** e **tante L1**, disperse su molte istanze/nodi
- operazioni **multiple** e **concorrenti**, applicate a una L2 + molte L1
- nei sistemi distribuiti esistono problematiche **fondamentali** a livello di **clock/sync**
- i sistemi distribuiti possono **fallire**, anche solo **temporaneamente**

Ok 🥵

# 🏷️ Tagging

Le **fondamenta** su cui poggiarsi per ottenere una feature come Tagging sono quelle.

E’ stata una sfida **incredibilmente complessa** e apparentemente impossibile.

Eppure, funziona tutto, e in FusionCache con delle performance **sorprendenti**.

ℹ️ **NOTE:** se volete approfondire il design sottostante, ci sono i docs di FusionCache!

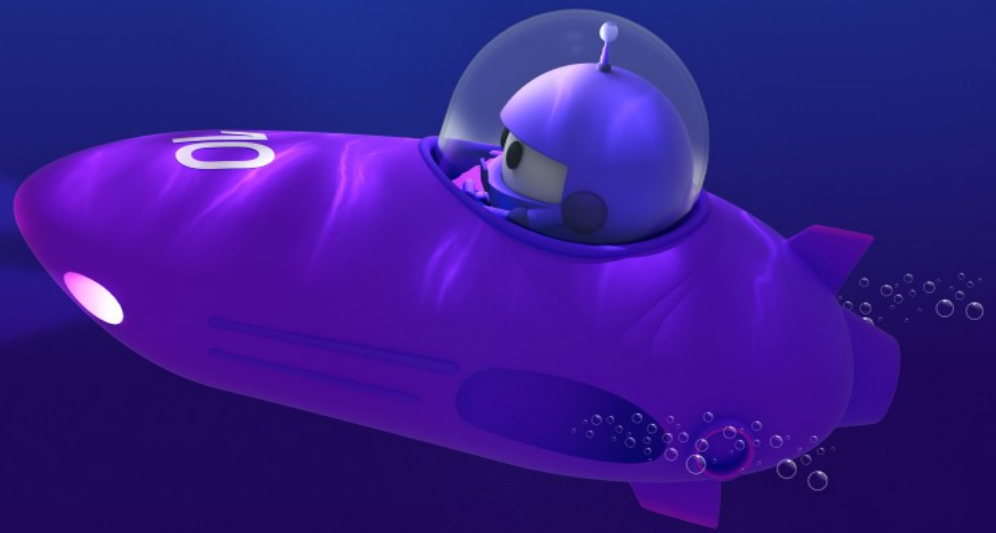
# 🏷️ Tagging

In conclusione:

- **l’utilità** di questa feature è **enorme**, e apre scenari altrimenti inaccessibili
- da una parte abbiamo una **complessità** interna dannatamente **elevata**
- dall’altra, come visto, un **utilizzo finale** incredibilmente **semplice**, quasi banale

Questo contrasto rende Tagging una delle mie feature **preferite** in assoluto.

# Named Caches (FusionCache)



# Named Caches

E’ bello poter cambiare dinamicamente il setup da **L1** a **L1+L2**.

Ma per quanto sia bello, dobbiamo fare **una** scelta: o L1, o L1+L2.

E se avessimo bisogno di **setup diversi** per **scopi diversi**?

FusionCache ci viene in soccorso.

E lo fa ispirandosi agli **HTTP Named Clients** di .NET stesso.

# Named Caches

Con HTTP Named Clients facciamo `AddHttpClient(name)`:

```
builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");
    // CONFIG ...
});
```

# Named Caches

E poi lo otteniamo tramite `IHttpClientFactory.CreateClient(name)`:

```
public class MyController : Controller
{
    private readonly HttpClient _client;

    public MyController(IHttpClientFactory clientFactory)
    {
        _client = clientFactory.CreateClient("GitHub");
    }

    // ...
}
```

# Named Caches

Allo stesso modo, con **Named Caches** facciamo `AddFusionCache(name)`:

```
services.AddFusionCache("Products");
services.AddFusionCache("Customers");
```

# Named Caches

E poi la otteniamo tramite `IFusionCacheProvider.GetCache(name)`:

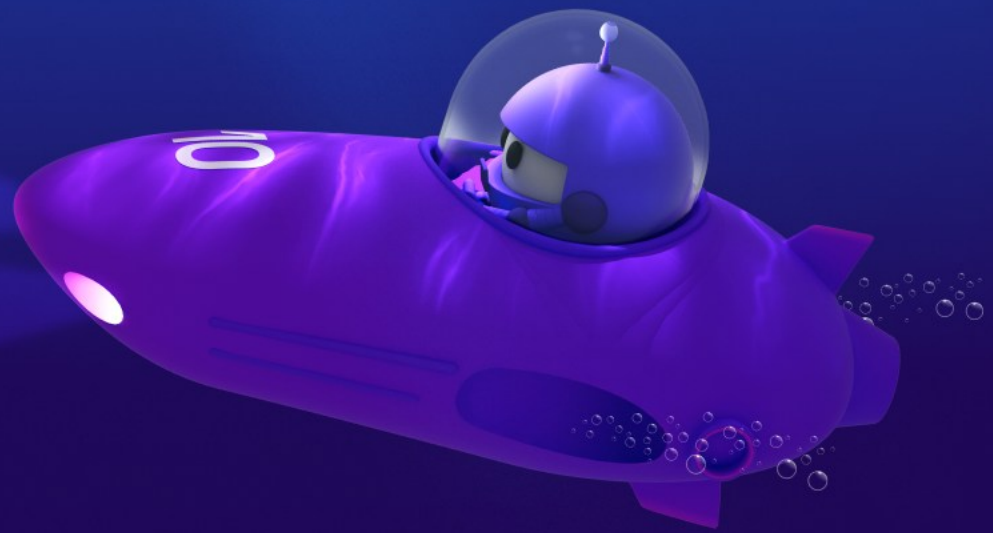
```
public class MyController : Controller
{
    private readonly IFusionCache _cache;

    public MyController(IFusionCacheProvider cacheProvider)
    {
        _cache = cacheProvider.GetCache("Products");
    }

    // ...
}
```

# Named Caches

E ovviamente **ogni cache** può avere **setup** e **opzioni** totalmente **diverse**:

```
// PRODUCTS CACHE (L1 + L2 + BACKPLANE)
services.AddFusionCache("Products")
    .WithDefaultEntryOptions(...)
    .WithSerializer(...)
    .WithDistributedCache(...)
    .WithBackplane(...);

// CUSTOMERS CACHE (ONLY L1)
services.AddFusionCache("Customers")
    .WithOptions(...);
```

# Problemi



# 🚨 Problemi

Fin qui abbiamo coperto svariati requisiti, da semplici a molto complessi.

Ma abbiamo trascurato un dettaglio decisamente importante.

**Le cose possono andare storte.**

E pretendere che **non sia così** o **non prepararsi** significa **perdere** in partenza.

# 🚨 Problemi

Vediamo quindi quali **problemi** possono verificarsi in **scenari reali**.

E, soprattutto, vediamo come:

- prevenirli
- affrontarli
- mitigarli
- risolverli

Ne varrà la pena.

# 🚨 Problemi

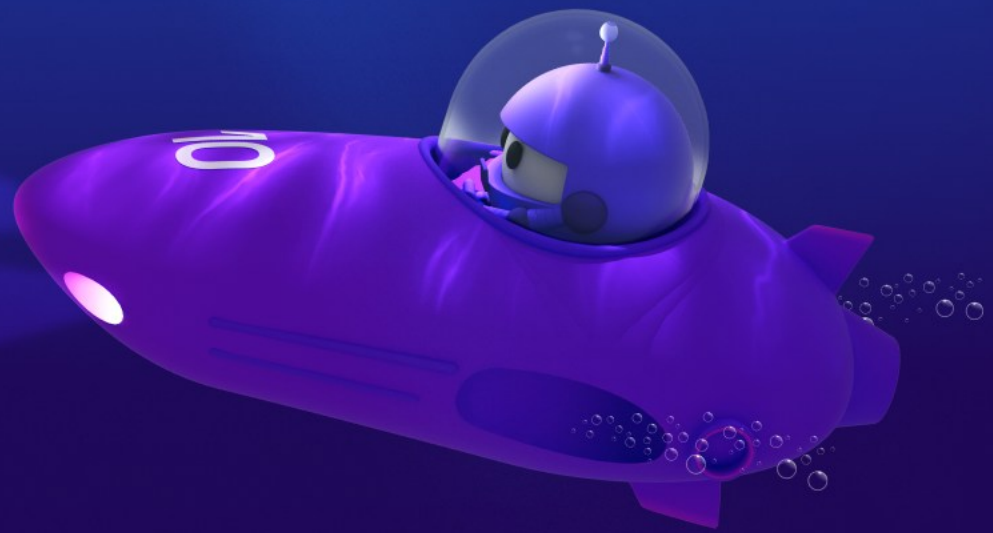
Serve però fare una piccola **premessa**.

Le **altre librerie** sono tutte **ottime**, e vale la pena **conoscerle** per poter **scegliere** al meglio.

Detto ciò, ad oggi **FusionCache** è l'unica libreria che possiede **feature** legate alla **resilienza**.

Quindi tutto ciò che vedremo da qui in avanti è **esclusivamente** relativo a FusionCache.

# Database: Errori



# Database: Errori

A volte, quando si parla con il database, possono verificarsi **errori temporanei**.

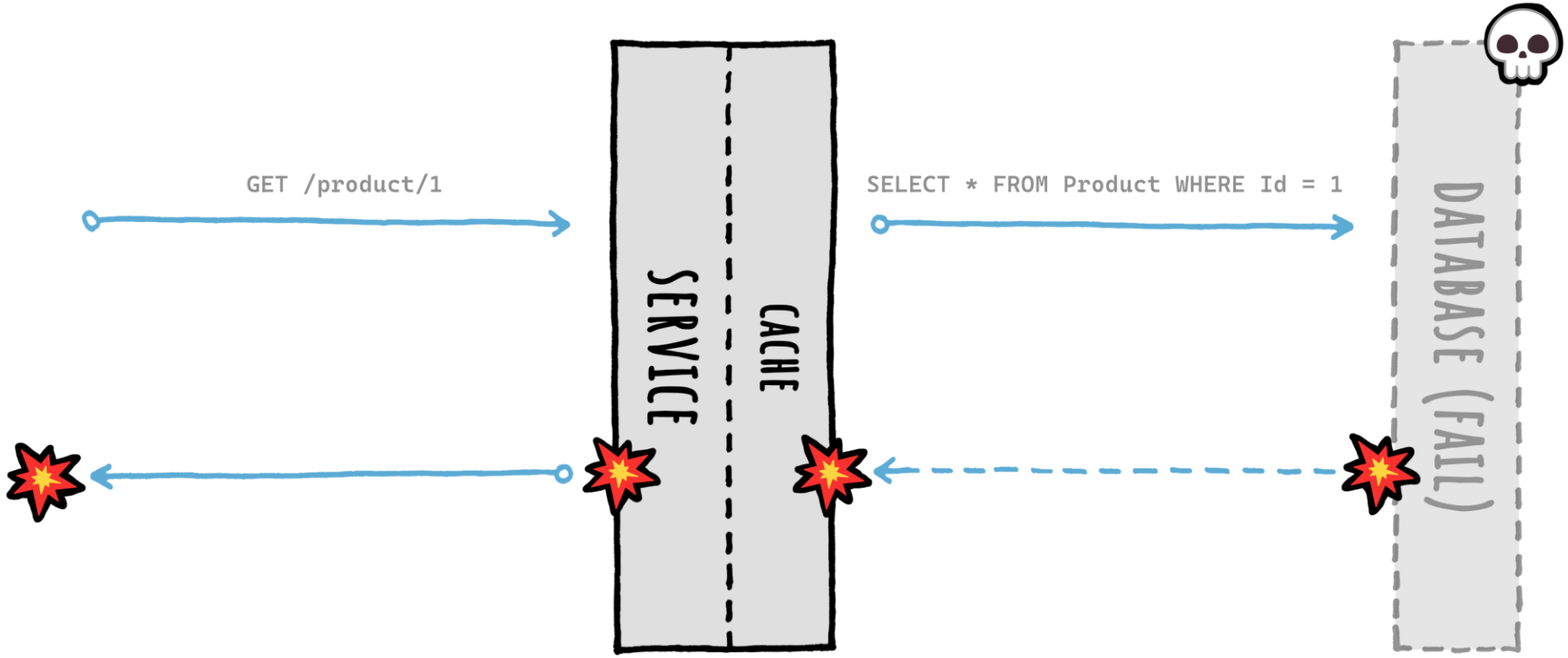
Questi possono accadere per vari motivi:

- **query timeout:** query scritta male, indice mancante
- **database restart:** aggiornamento dell'engine, crash, riavvio
- **problemi network:** perdita di connettività, cambiamento della topologia

Cosa succede allora?

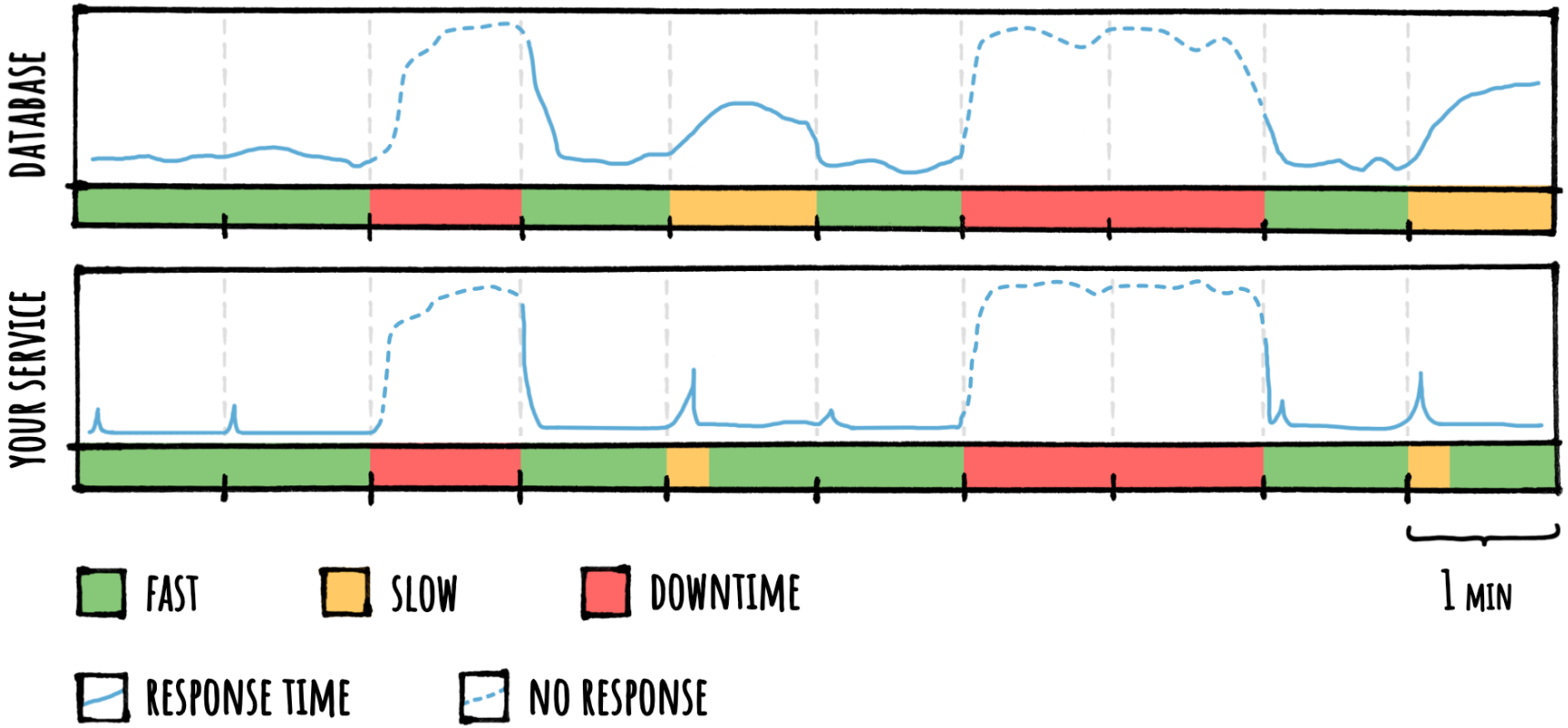
# Database: Errori

Questo:



# Database: Errori

Quando il **database** genera errori, il nostro **servizio** genera errori:

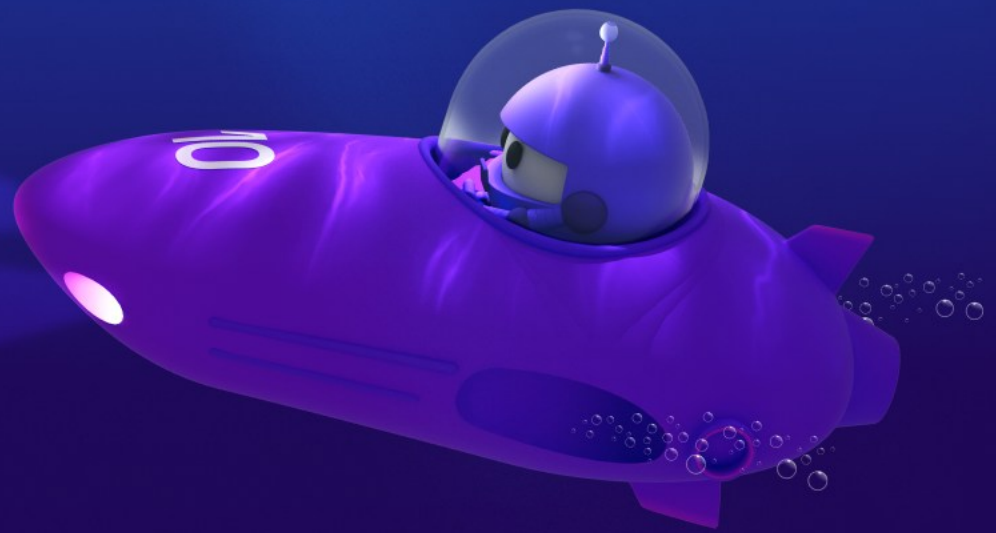


# Database: Errori

Ma se i dati nella cache sono già **scaduti** e il database non è **disponibile**, c'è davvero qualcosa che possiamo fare?

Entra in scena: **Fail-Safe**.

# Fail-Safe (FusionCache)



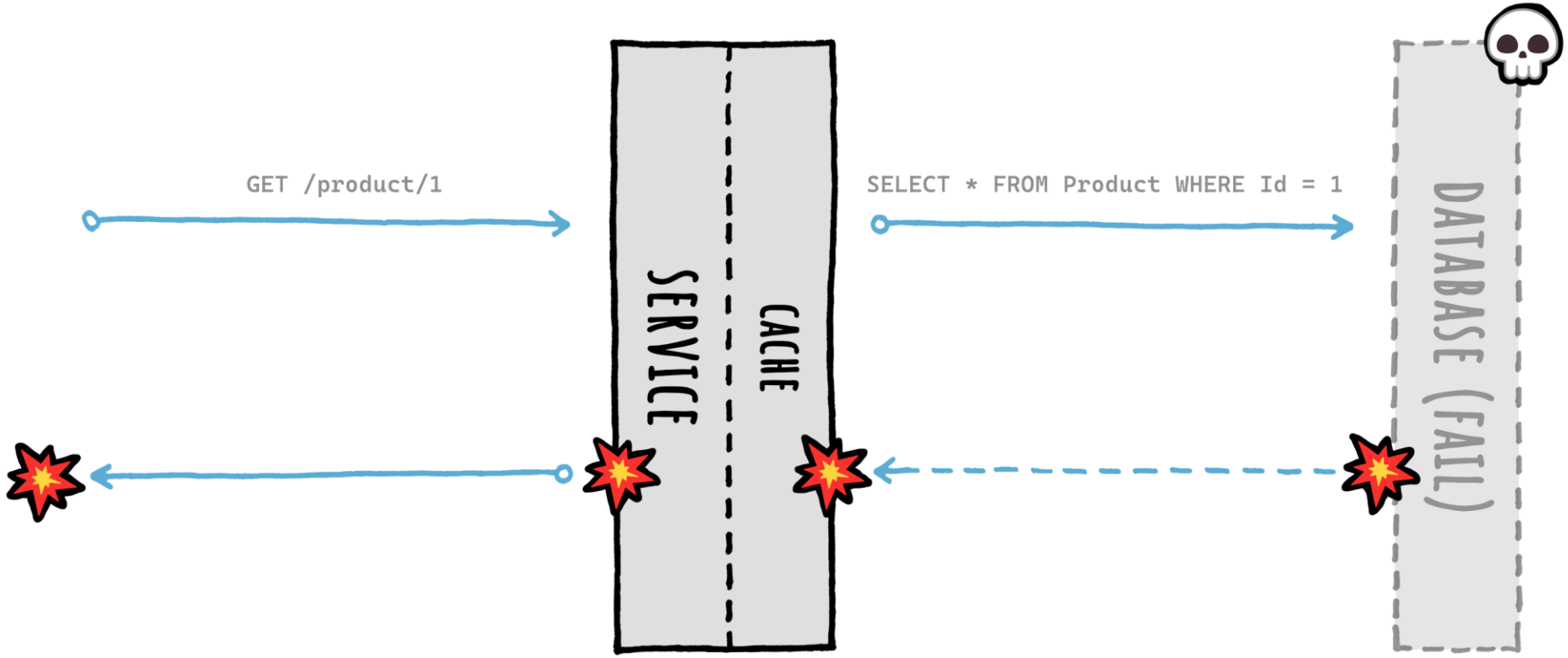
# Fail-Safe

Se memorizzassimo qualcosa in cache per, diciamo, **10 minuti**, sarebbe un problema usarlo **un po' di più** nel caso in cui il database **non sia disponibile**?

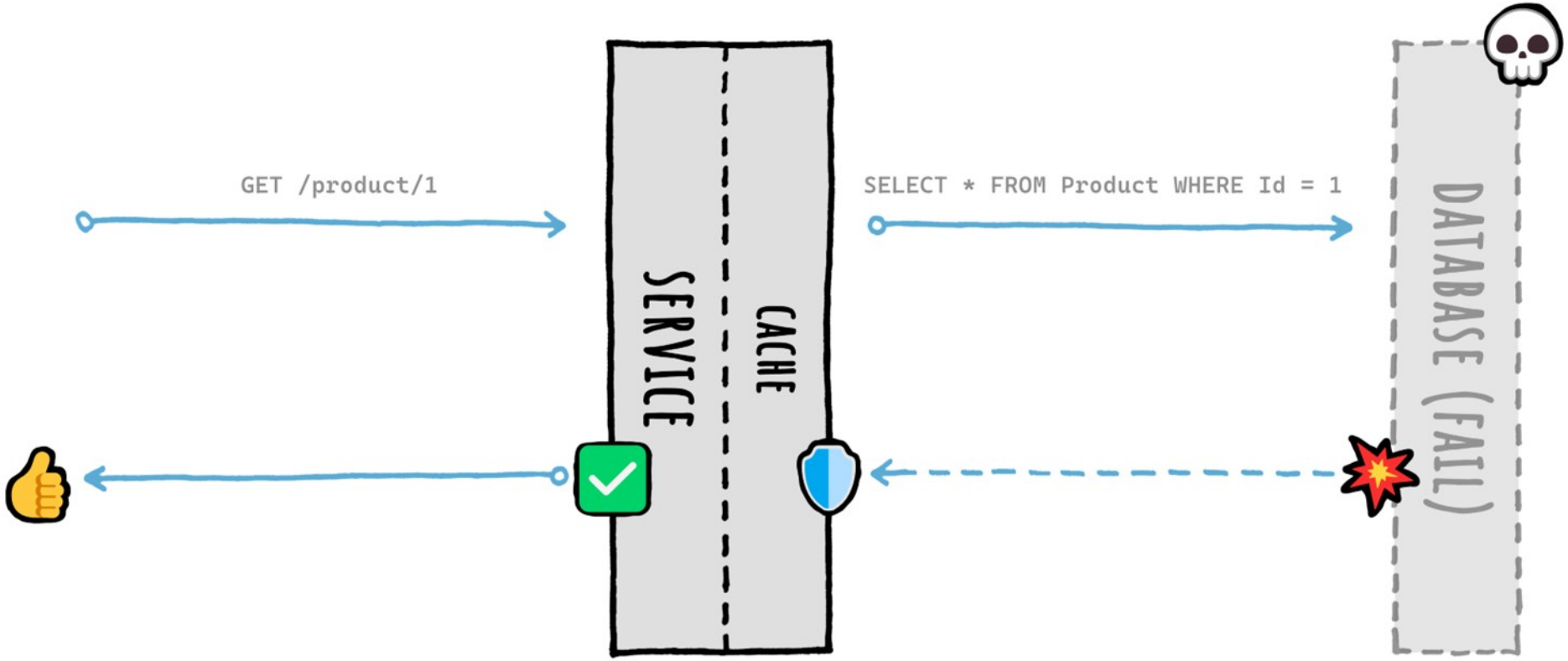
In FusionCache, il meccanismo di **Fail-Safe** ci permette di fare proprio questo.

È come una «seconda chance» per quando le cose «vanno storte».

# Fail-Safe: Senza



# Fail-Safe: Con



# Fail-Safe

Supponiamo di avere una `Duration` di **5 sec**:

- **senza Fail-Safe:** dopo 5 sec l'entry è **scaduta**, quindi **eliminata** dalla cache
- **con Fail-Safe:** dopo 5 sec l'entry è **considerata scaduta**, ma **non eliminata** dalla cache

Con Fail-Safe la `Duration` diventa virtuale/logica.

In entrambi i casi dopo 5 sec ci sarà un **refresh**, ma con Fail-Safe il dato resterà comunque **disponibile** come **fallback**.

E in caso di problemi, l'entry scaduta viene **temporaneamente** risalvata nella cache, per un po' di tempo extra (tutto configurabile).

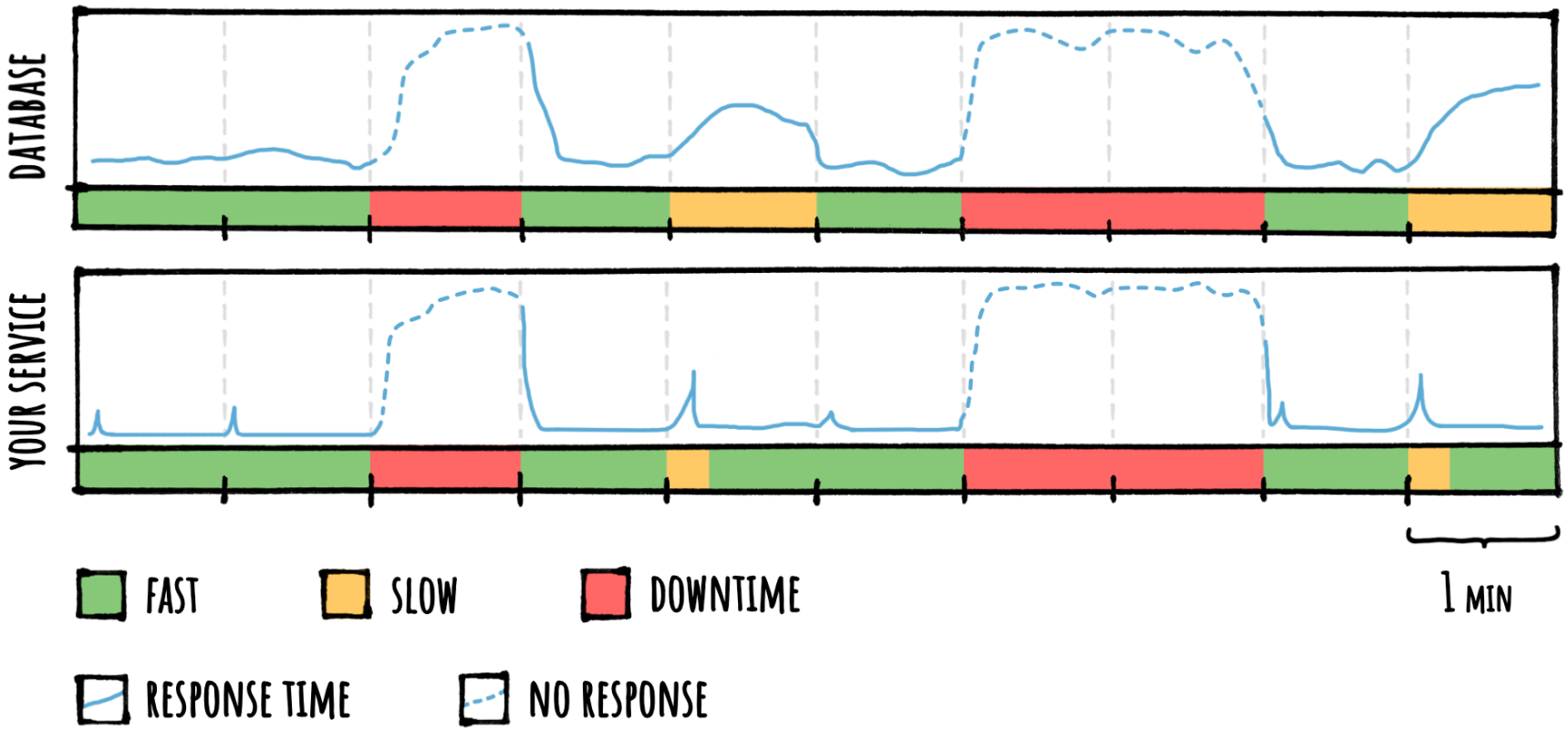
# Fail-Safe

Si usa così:

```
var id = 42;

var product = cache.GetOrSet<Product>(
  $"product:{id}",
  _ => GetProductFromDb(42),
  options => options
    .SetDuration(TimeSpan.FromMinutes(1))
    // FAIL-SAFE
    .SetFailSafe(true, TimeSpan.FromHours(24), TimeSpan.FromSeconds(30))
);
```

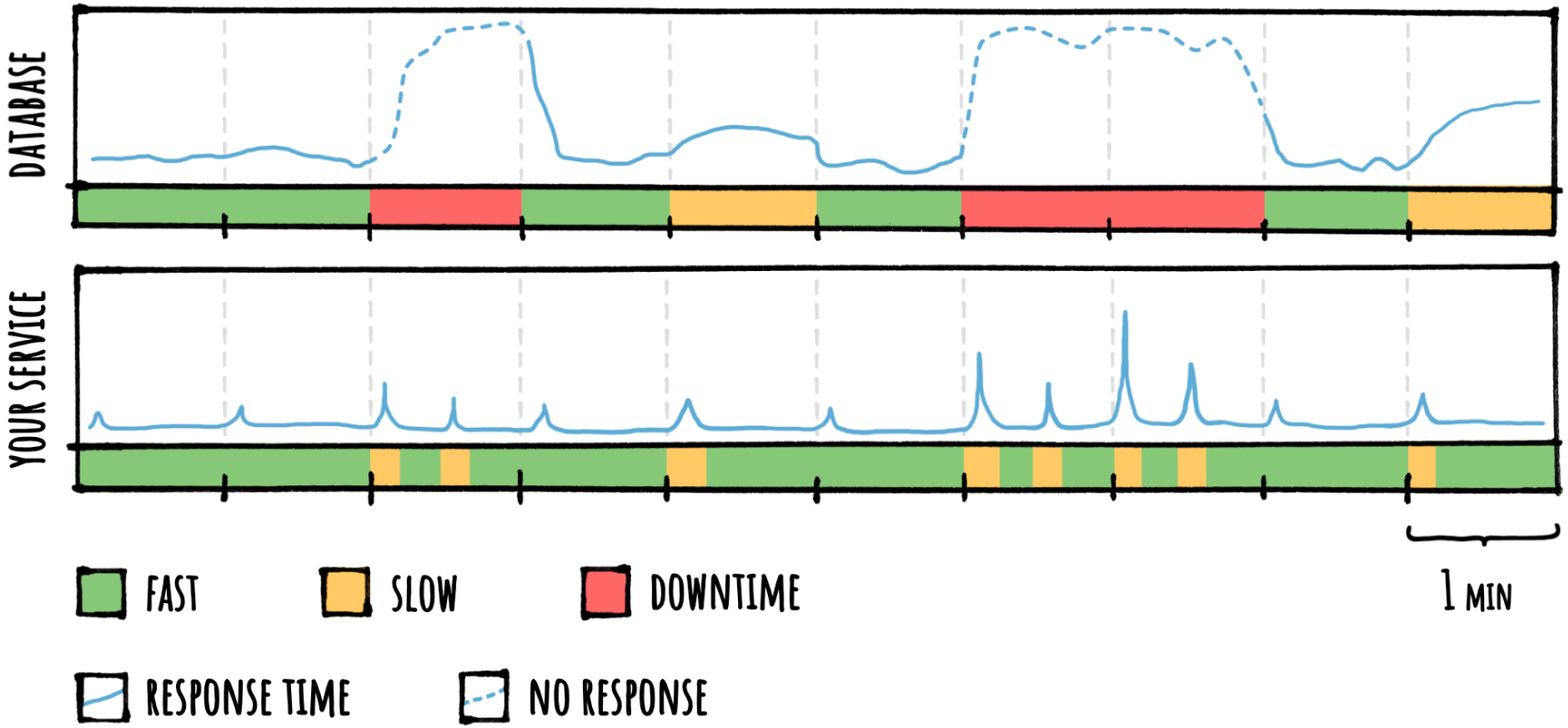
# Fail-Safe: Senza

Senza Fail-Safe il nostro servizio **riflette** i problemi del database:



# Fail-Safe: Con

Con Fail-Safe il nostro servizio sarà **schermato** dai problemi del database:



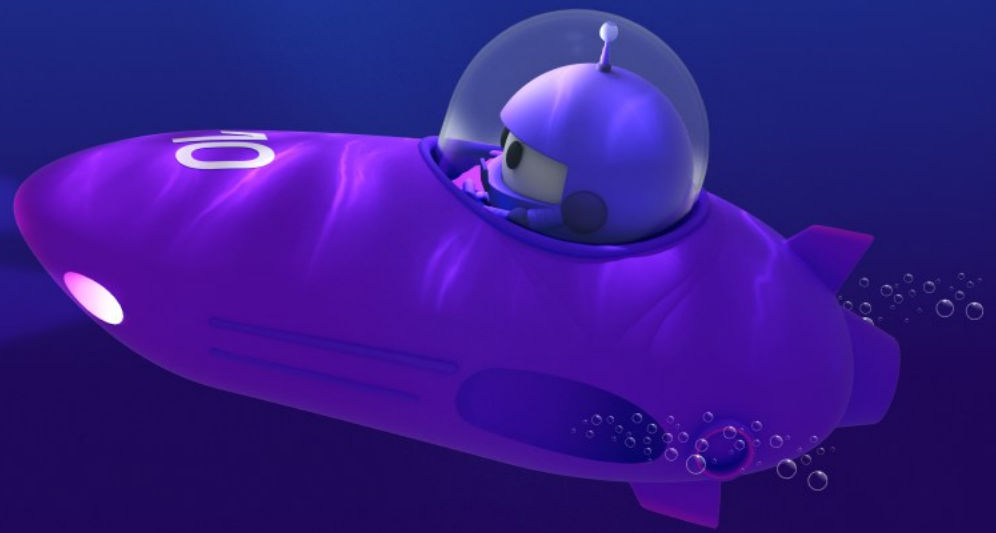
# ❤️ Fail-Safe: Sidney Lumet

**Anno:** 1964

**Regia:** Sidney Lumet
**Fotogragia:** Gerald Hirschfeld

**Cast:**
- Edward Binns
- Walter Matthau
- Henry Fonda
- Dan O'Herlihy
- Fritz Weaver
- Janet Ward
- Frank Overton
- Dana Elcar

# Database: Rallentamenti



# 🐌 Database: Rallentamenti

A volte il database non è completamente **offline** o **irrangingibile**: è solo **lento**.

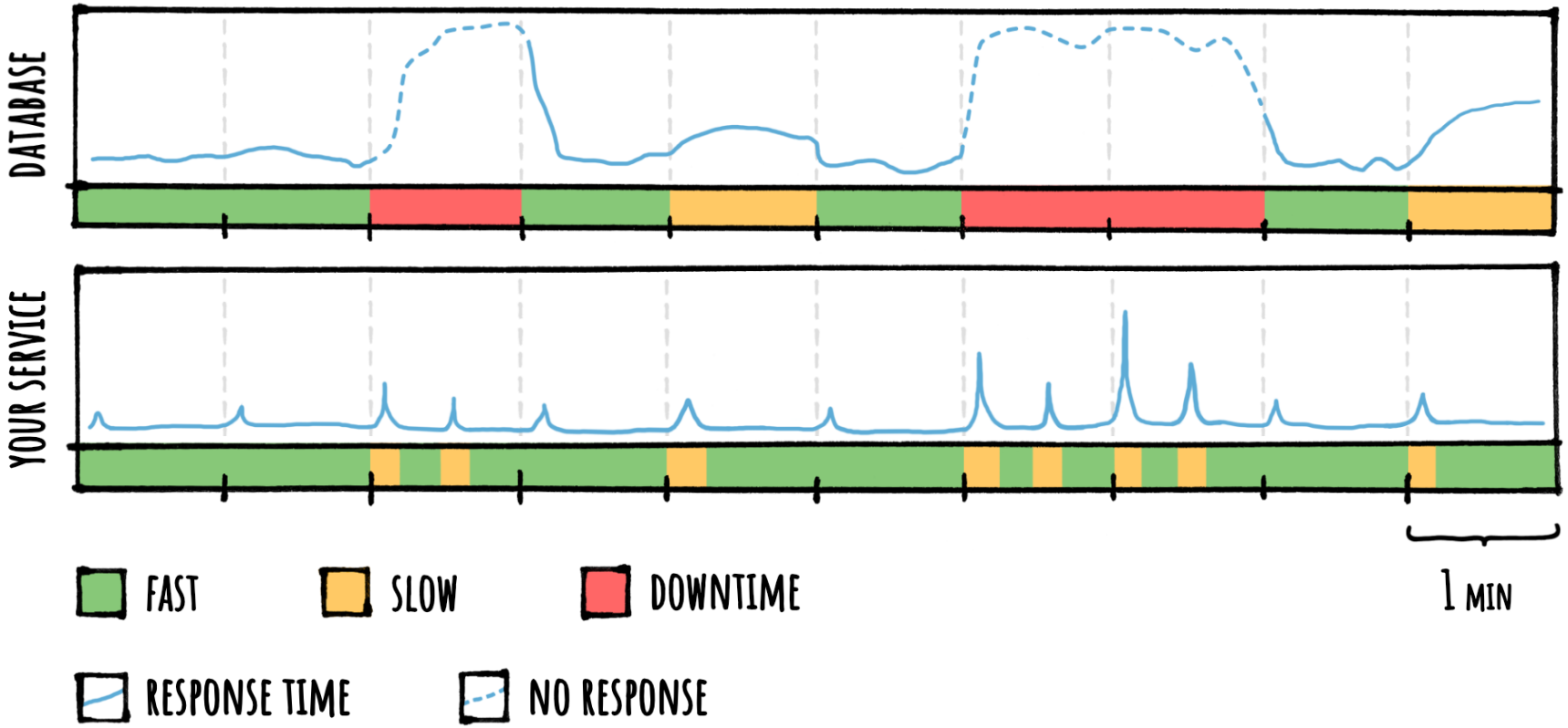
Questo può accadere per vari motivi:

- **overload:** il database è attualmente sovraccarico
- **query non ottima:** a volte non scriviamo le query più spettacolari
- **indice mancante:** abbiamo dimenticato un indice
- **problemi network:** congestione, cambiamento della topologia

Cosa succede allora?

# 🐌 Database: Rallentamenti

Quando il **database** è lento, **il nostro servizio** è lento:



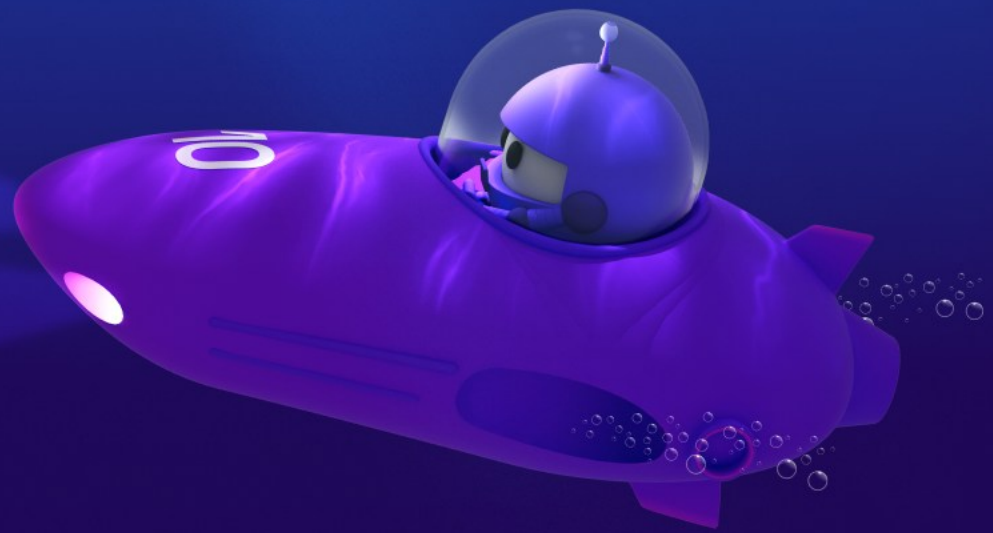
# 🐌 Database: Rallentamenti

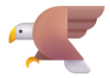
Ma se i dati sono già **scaduti** e il database è **lento**, cosa possiamo fare?

Forse... agire in modo **proattivo**?

Entra in scena: **Eager Refresh**.

# Eager Refresh (FusionCache)



#  Eager Refresh

Con **Eager Refresh** aggiorniamo un valore **prima** che scada, ma solo **dopo** una certa **soglia**.

La soglia è espressa come **percentuale** della `Duration` con un valore da 0,0 a 1,0 dove:

- 0,5 = 50%
- 0,75 = 75%
- etc

Quando viene effettuata una richiesta alla cache **dopo** la soglia, si attiva **Eager Refresh**.

Quando attivato:

- il valore nella cache viene restituito **immediatamente** (dato che è **ancora valido**)
- la factory viene eseguita in **background**, in modo **non bloccante**

# Eager Refresh

Si abilita così:

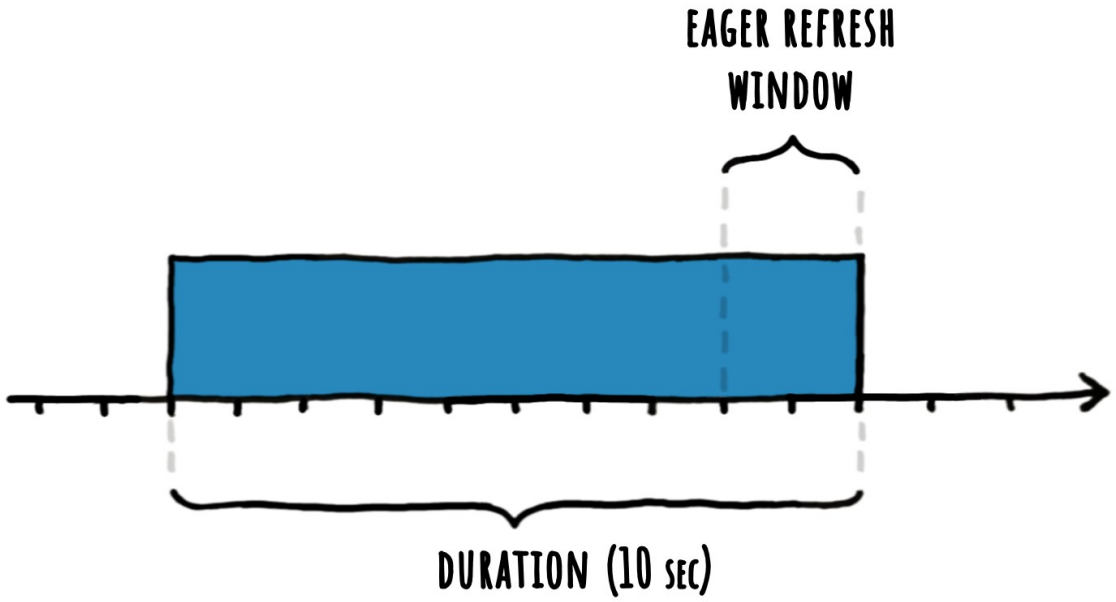
```
var id = 42;

var product = cache.GetOrSet<Product>(
  $"product:{id}",
  _ => GetProductFromDb(42),
  options => options
    .SetDuration(TimeSpan.FromMinutes(1))
    // EAGER REFRESH
    .SetEagerRefresh(0.9f)
);
```

# Eager Refresh

Così facendo Eager Refresh ci aiuterà se arriva una richiesta:

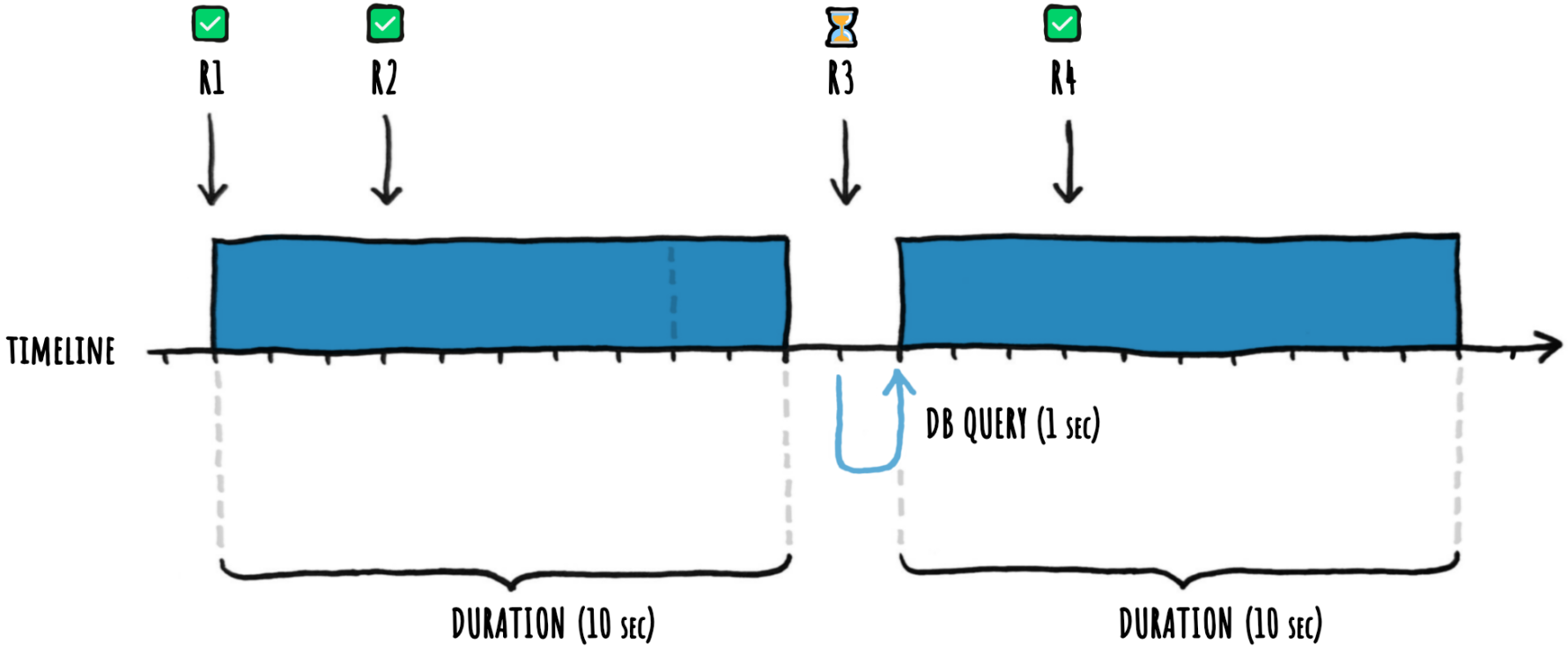
- **dopo** la **soglia**
- **prima** della scadenza

all'interno della **finestra di tempo** chiamata **Eager Refresh Window**.



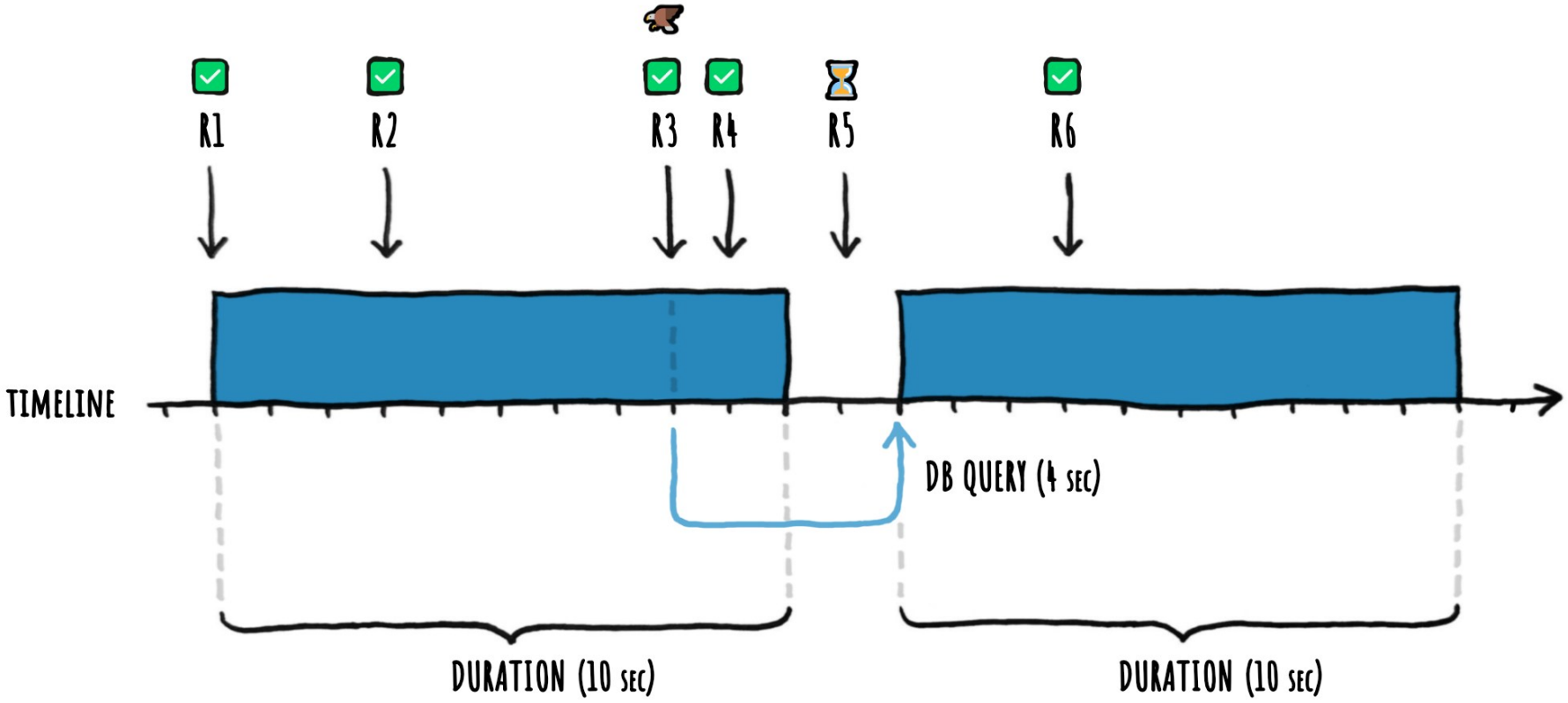
# Eager Refresh

Ma cosa succede se non ci sono richieste **all'interno** di quella finestra?



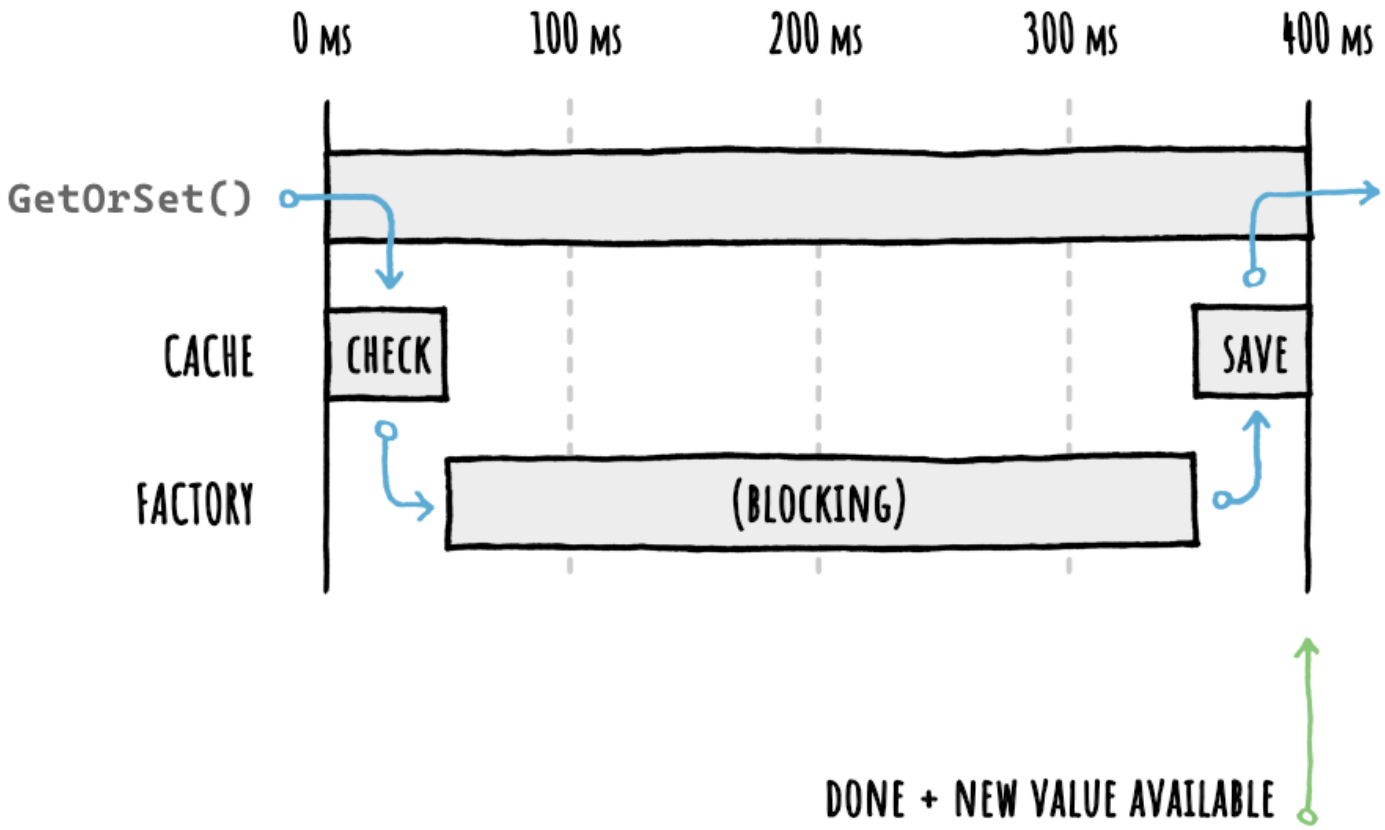
# Eager Refresh

Oppure se Eager Refresh si attivava puntualmente, ma la factory è **particolarmente lenta**?



# Eager Refresh

Il problema è che l'esecuzione della factory è un'**operazione bloccante**:



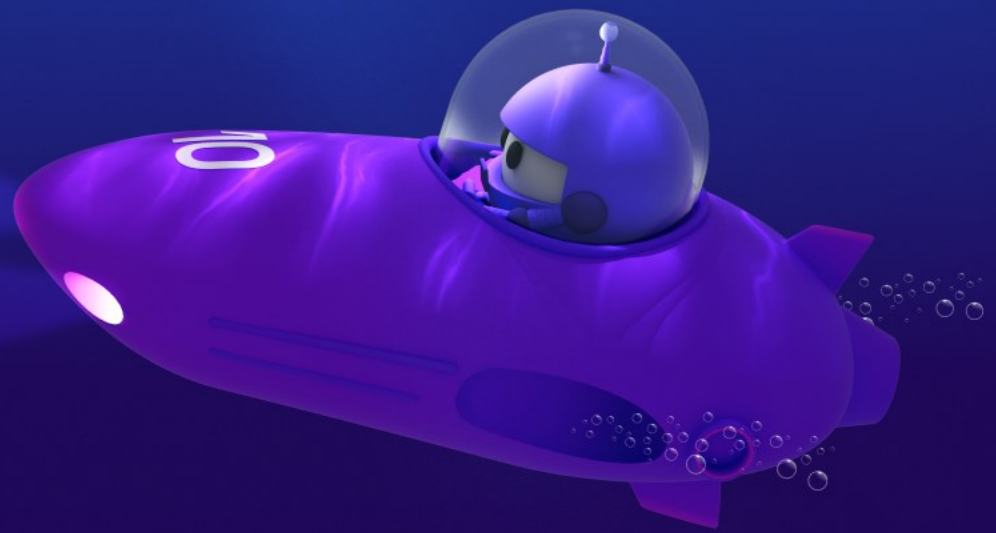
#  Eager Refresh

Ma ecco il punto: se i dati nella cache sono scaduti, l'esecuzione della factory è un'**operazione bloccante** e il database è **lento**, c'è davvero qualcosa che possiamo fare?

(ormai avrete capito dove voglio andare a parare)

Entra in scena: **Factory Timeouts**.

# Factory Timeouts (FusionCache)



# ⏱️ Factory Timeouts

Se memorizziamo qualcosa in cache, diciamo, per **5 min**, è un problema usarlo un po' di più nel caso il database sia **lento**?

Suona familiare, vero? Esatto, come Fail-Safe.

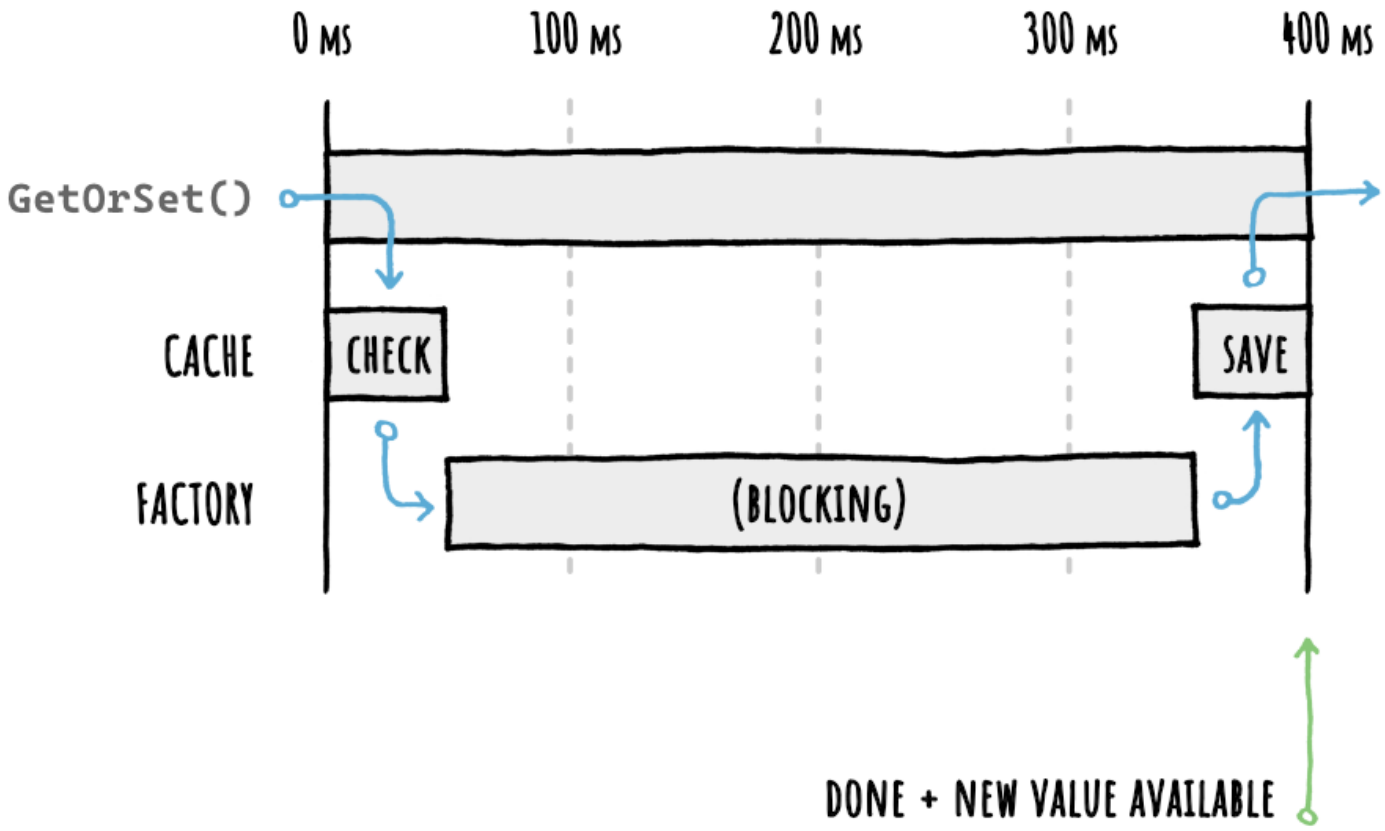
L'idea è che possiamo impostare un timeout speciale per l'esecuzione della factory.

Se scatta il timeout, verrà:

- **usato** il valore **scaduto**, disponibile grazie a Fail-Safe
- **completata** la factory in background

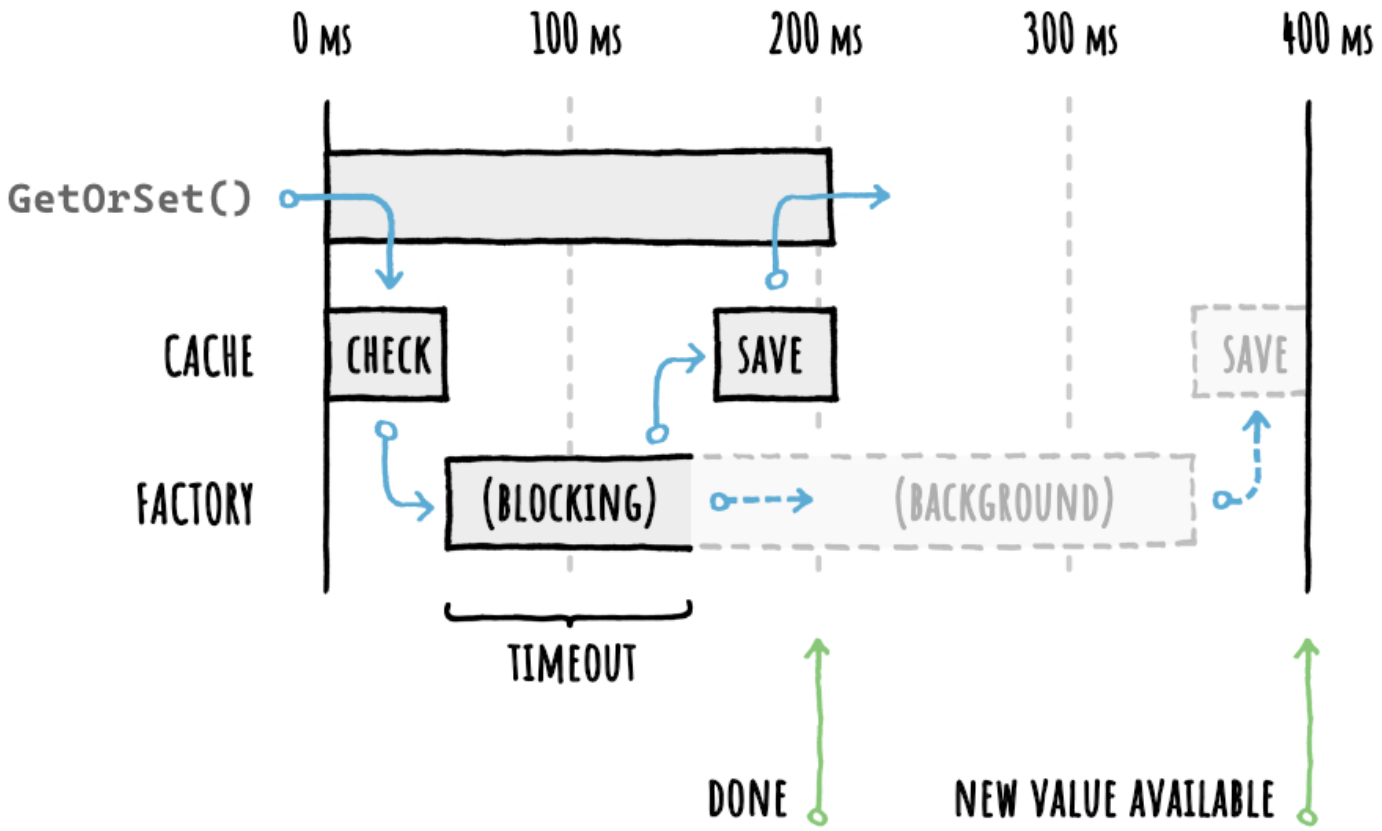
# ⏱️ Factory Timeouts

Prima:



# ⏱️ Factory Timeouts

Dopo:



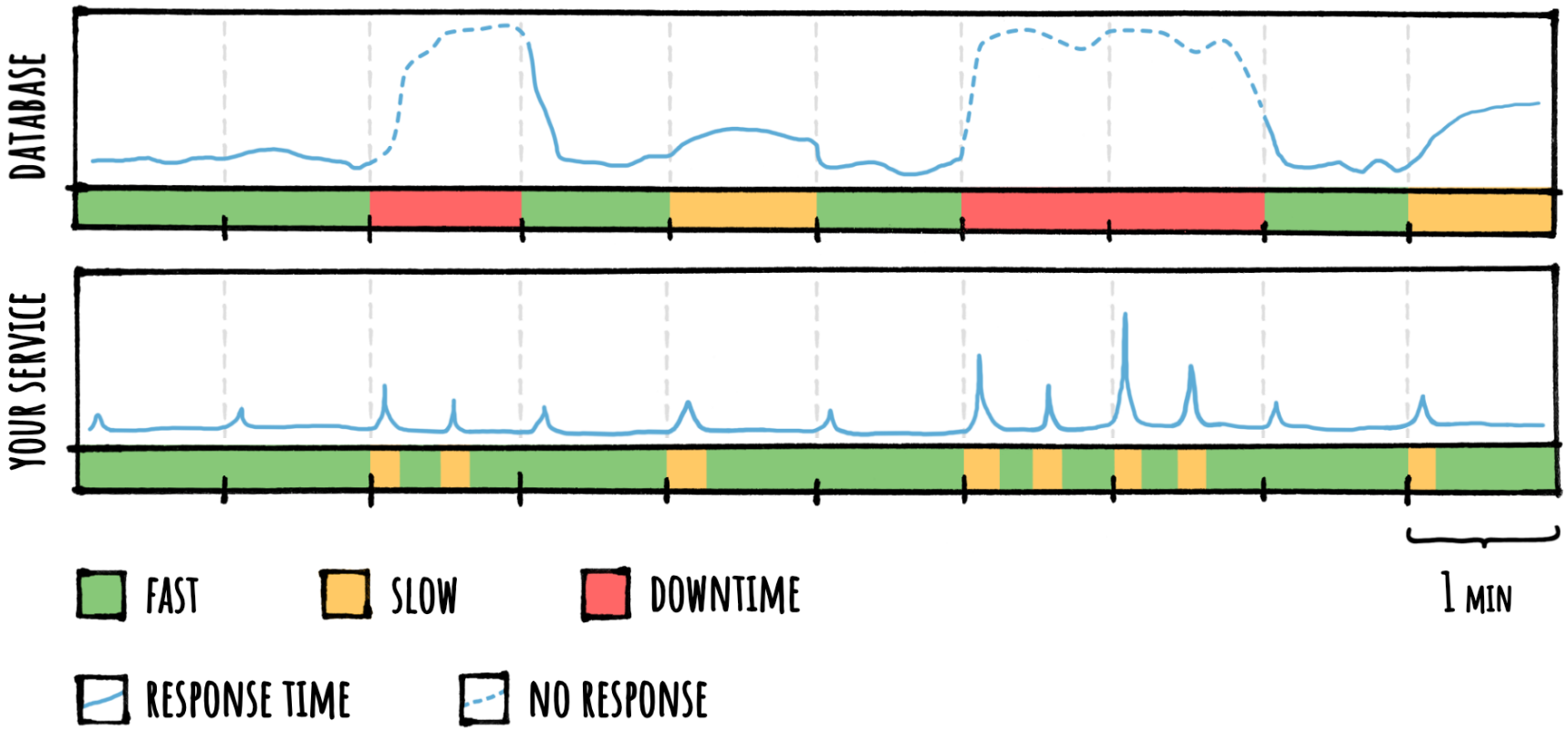
# ⏱️ Factory Timeouts

Si usa così:

```
var id = 42;

var product = cache.GetOrSet<Product>(
  $"product:{id}",
  _ => GetProductFromDb(42),
  options => options
    .SetDuration(TimeSpan.FromMinutes(1))
    // FACTORY TIMEOUTS
    .SetFactoryTimeouts(TimeSpan.FromMilliseconds(100), TimeSpan.FromSeconds(1))
);
```

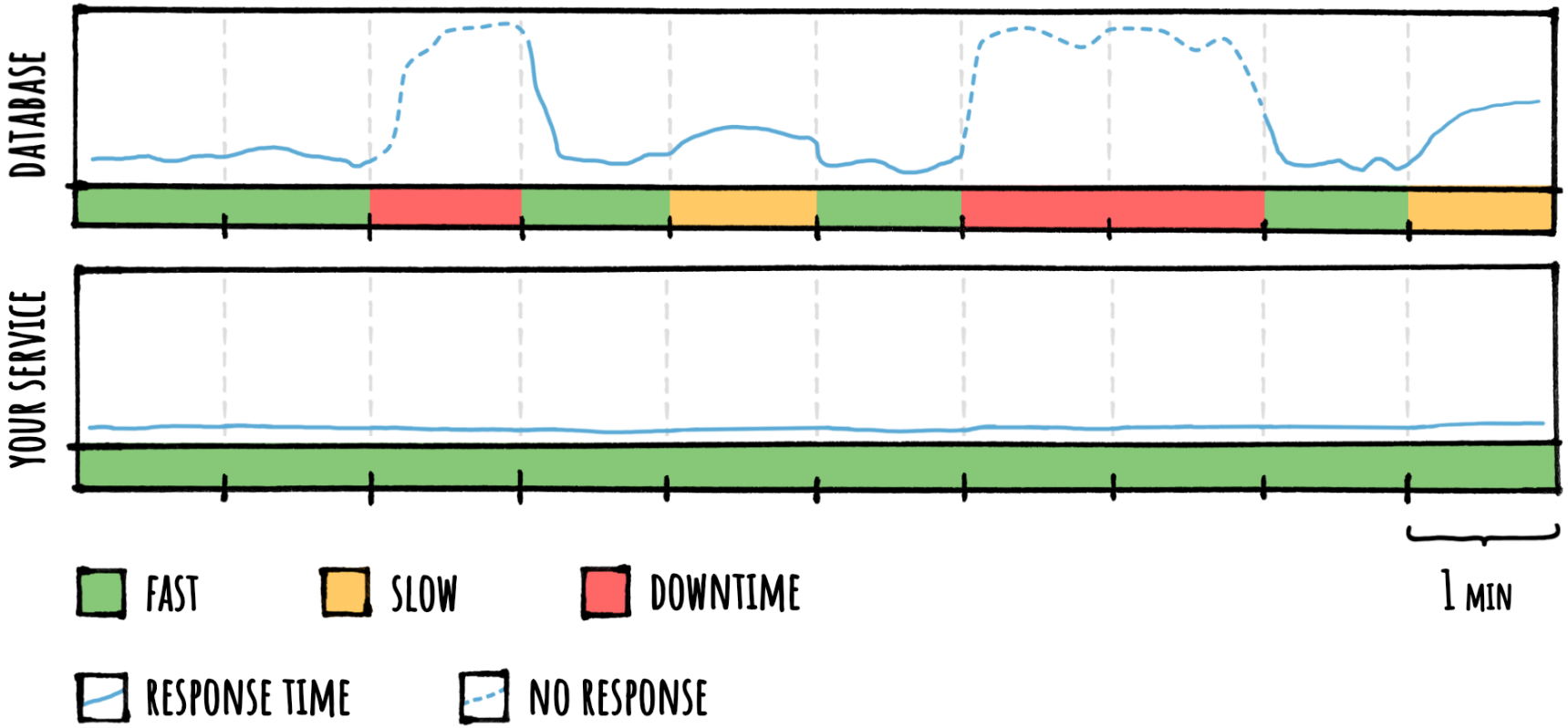
# ⏱️ Eager Refresh + Factory Timeouts: Senza

Senza **Eager Refresh** e **Factory Timeouts** eravamo così:

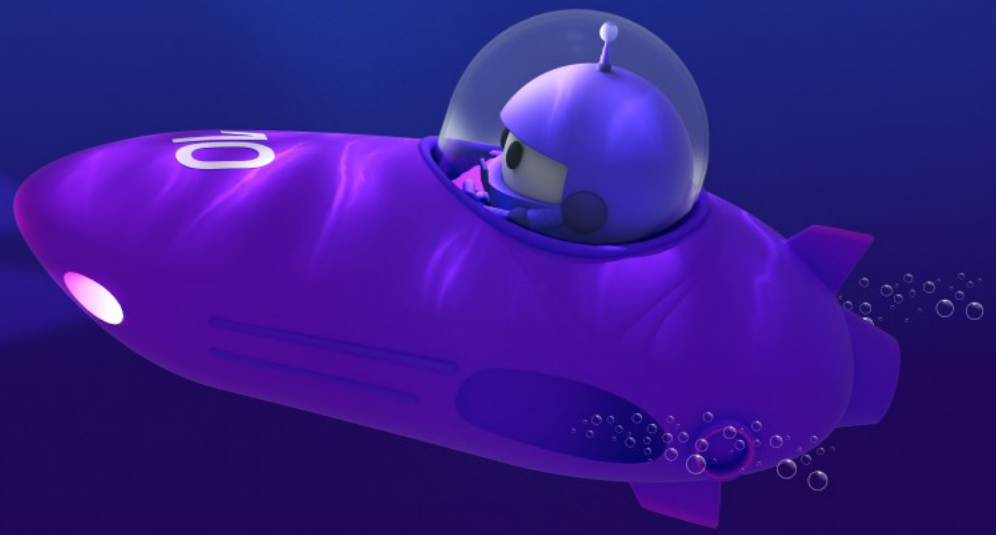


# ⏱️ Eager Refresh + Factory Timeouts: Con

Abilitandoli siamo così:



# Distributed Parts: Problemi



# Distributed Parts: Problemi

Abbiamo visto come poter affrontare problemi durante la comunicazione con il **database**.

Ma il database non è l'unica **componente distribuita**, ci sono anche:

- distributed cache (L2)
- backplane

E anche con loro possiamo avere problemi.

# Distributed Parts: Problemi

I motivi sono potenzialmente gli stessi:

- restart/crash
- overload
- problemi network
- etc

Quindi, cosa possiamo fare?

# Distributed Parts: Problemi

A livello applicativo possiamo fare ben poco.

Anzi **nulla**.

Pensiamoci un attimo.

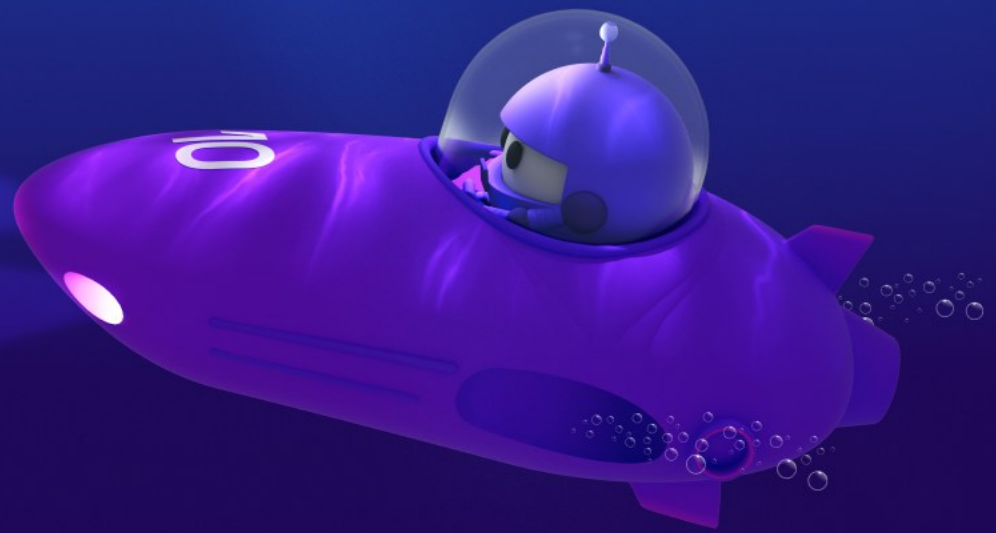
# Distributed Parts: Problemi

Ok, quindi?

Qualcuno può fare qualcosa?

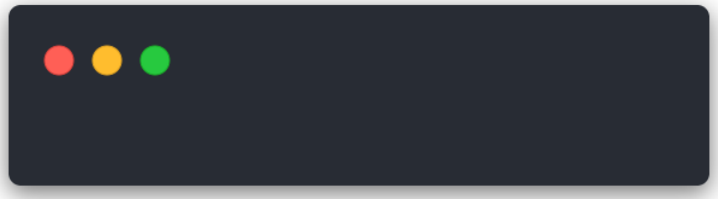
Si, la **cache stessa**.

# Auto-Recovery (FusionCache)



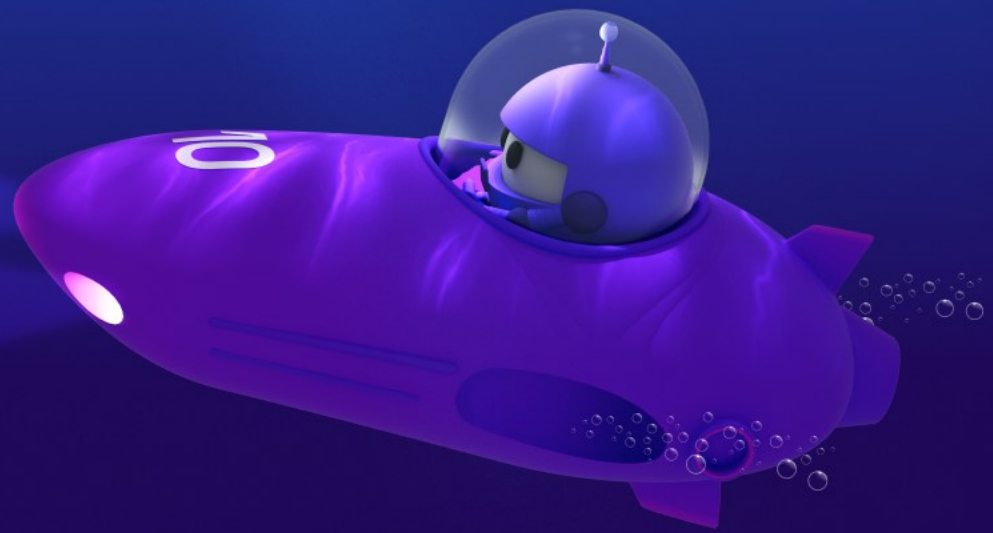
# Auto-Recovery

Facciamo così:



Vediamo una demo (se c’è tempo 😅).

# Observability



# 🔭 Observability

In un sistema **distribuito complesso** ci sono in ogni momento parecchie attività in corso.

E sarebbe bello poter capire meglio cosa stia succedendo.

Si può fare qualcosa?

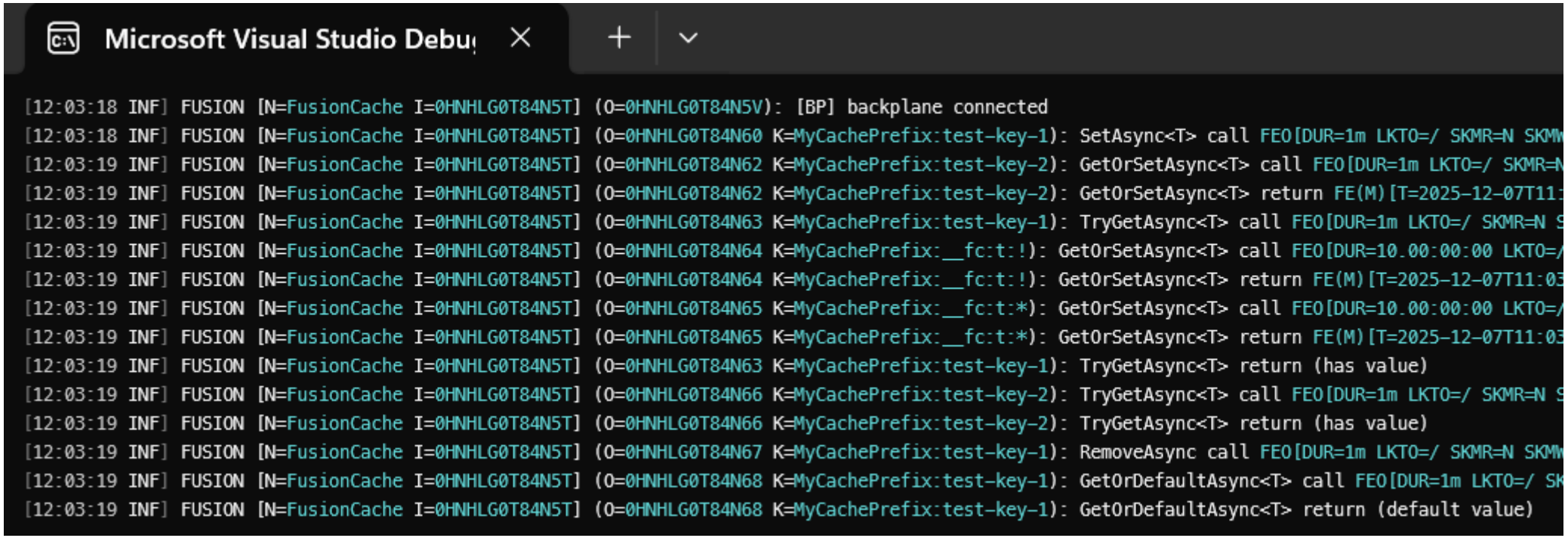
# 🔭 Observability

Si, abilitare **Logging**:

```
// LOGGING SETUP
services.AddLogging(b => b
    .SetMinimumLevel(LogLevel.Warning)
    .AddSimpleConsole(options => options.IncludeScopes = true)
);

// INTEGRATED LOGGING
services.AddFusionCache();
```

# Observability

Otteniamo un log, **se vogliamo**, molto dettagliato:



#  Observability

Possiamo anche customizzare alcuni **log level** utilizzati:

```
services.AddFusionCache()
    .WithOptions(options =>
    {
        // FACTORY SYNTHETIC TIMEOUTS: Debug (SO THEY WILL BE IGNORED)
        options.FactorySyntheticTimeoutsLogLevel = LogLevel.Debug;
        // ANY OTHER FACTORY ERRORS: Error (SO THEY WILL -NOT- BE IGNORED)
        options.FactoryErrorsLogLevel = LogLevel.Error;
    });
```

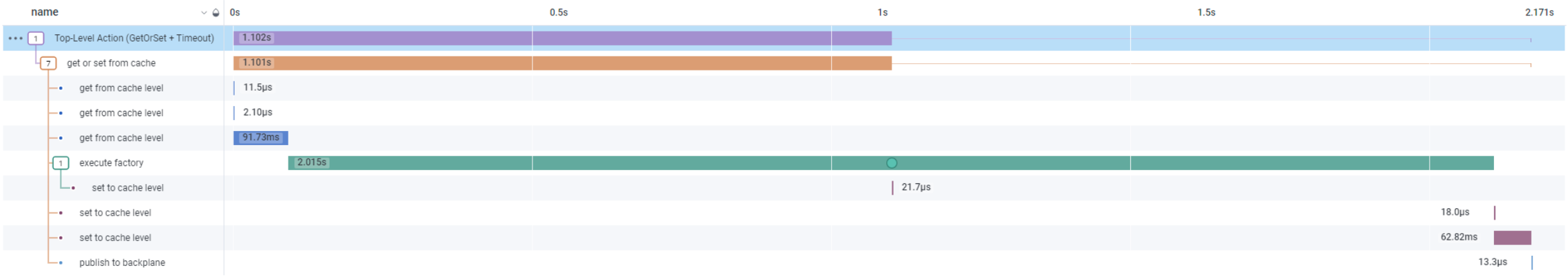
#  Observability

Ma, ancor di più, possiamo usare **OpenTelemetry**:

```
services.AddOpenTelemetry()
  // SETUP TRACES
  .WithTracing(tracing => tracing
    .AddFusionCacheInstrumentation()
    .AddConsoleExporter()
  )
  // SETUP METRICS
  .WithMetrics(metrics => metrics
    .AddFusionCacheInstrumentation()
    .AddConsoleExporter()
  );
```

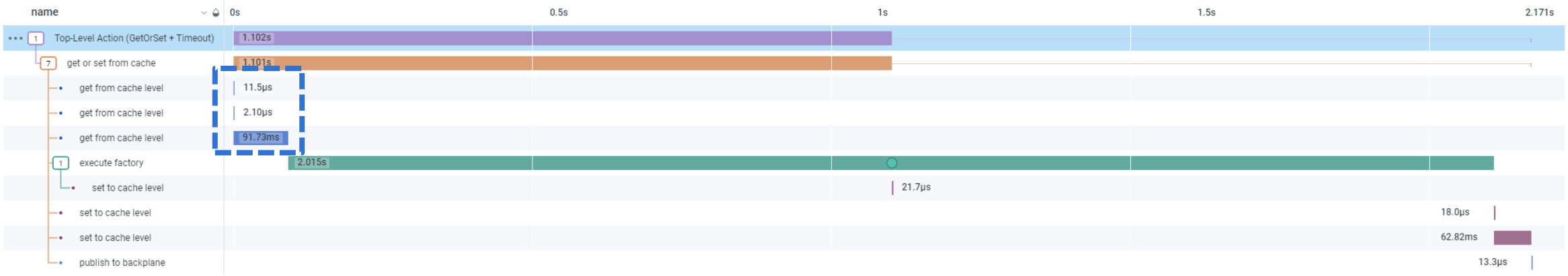
# Observability

E ottenere questa meraviglia:



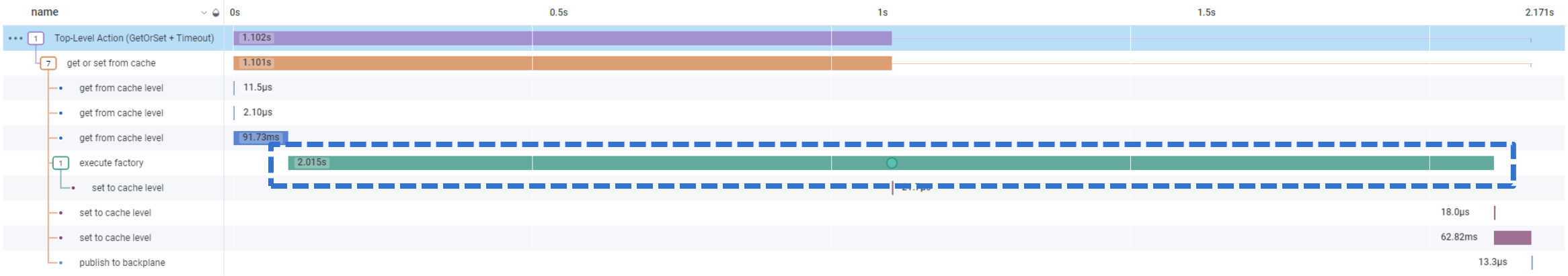
# Observability

Dove si possono osservare le varie operazioni sui livelli **L1** e **L2**:



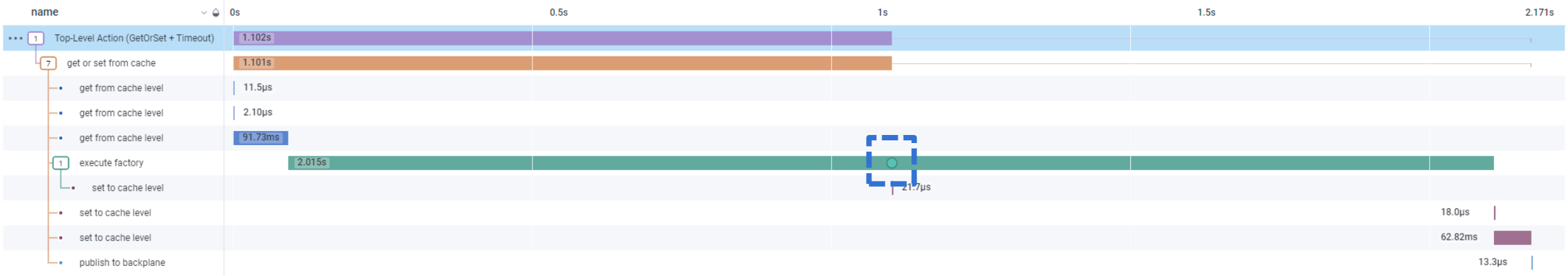
# Observability

L’esecuzione della **factory**:



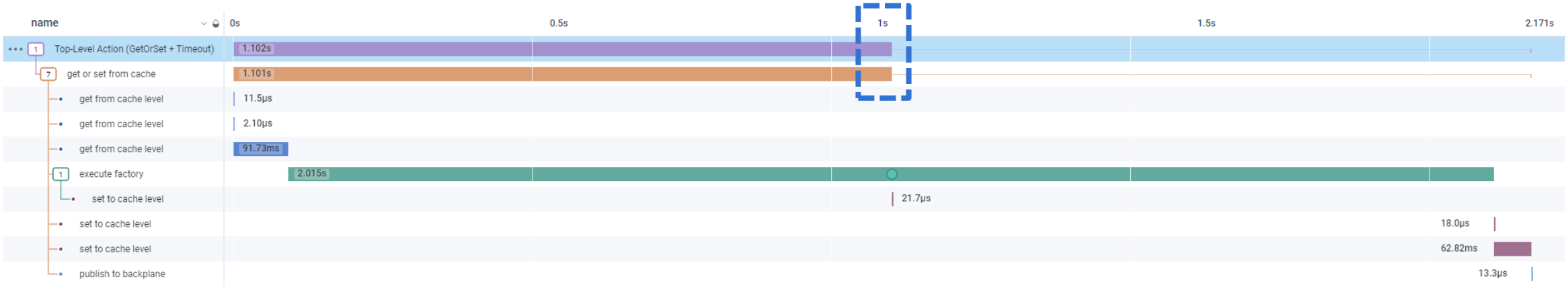
# Observability

Quando è scattato il **Factory Timeout** e la **factory** è stata spostata in **background**:



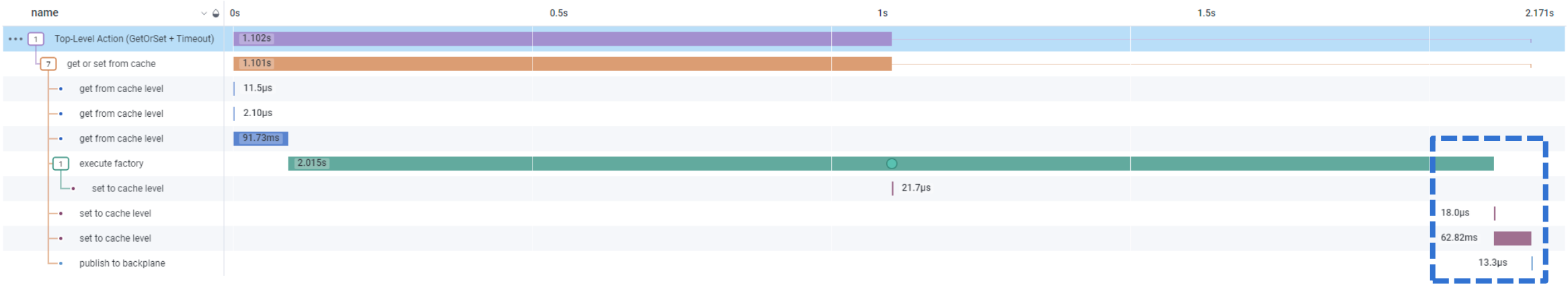
# Observability

E che la chiamata `GetOrSet()` iniziale è terminata **prima**, senza restare **bloccata** dalla factory:

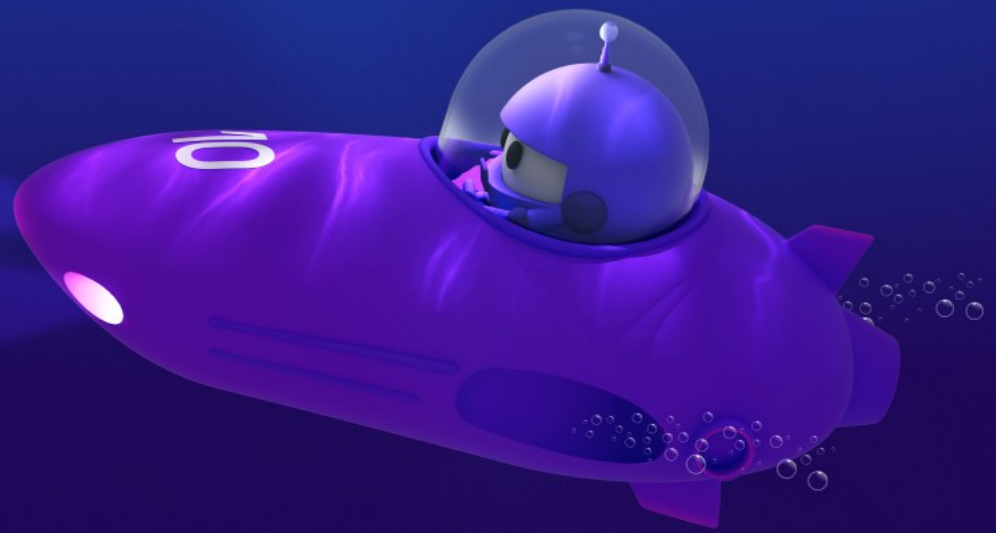


# Observability

E alla fine della factory in background **l'update** di **L1+L2** e la notifica sul **backplane**:



# HybridCache: Limitazioni E Problemi (fine 2025)



# HybridCache: Limitazioni E Problemi

Ricordate gli asterischi **(*)** che abbiamo visto prima?

Poiché l'attuale implementazione Microsoft è la primissima **versione**, presenta ancora alcune limitazioni e problemi.

La maggior parte non è legata all'**astrazione**, ma solo alla attuale **implementazione** di default.

Alcuni sono minori, altri più gravi, quindi è importante conoscerli.

Vediamo.

# HybridCache: no new()

La classe concreta è `internal`, quindi non è possibile istanziare direttamente tramite `new()`.

C'è un builder, ma anche quello è `internal`.

Questo significa affidarsi **solo** all'approccio **DI**.

Questo non è necessariamente un problema, ma è importante saperlo.

# HybridCache: no controllo su L1/L2

Per quanto riguarda l'approccio DI, non possiamo specificare cosa fare con:

- **L1**: memory level
- **L2:** distributed level

Entrambi arrivano automaticamente dal DI container, senza alcun controllo.

Ad esempio, per L2:

- se è registrato un servizio `IDistributedCache`, verrà utilizzato **per forza**
- se non è registrato un servizio `IDistributedCache`, **nulla** verrà utilizzato

Stessa cosa per L1.

# HybridCache: singola istanza

Di nuovo, riguardo DI: non possiamo avere più registrazioni di cache.

Questo significa che qualunque parte di codice userà la **stessa istanza**: dobbiamo fare attenzione alle possibili **collisioni** per le **cache key**.

E significa anche **no** a **configurazioni multiple**, dato che c'è un'unica istanza.

# HybridCache: solo async

L'API è **esclusivamente asincrona**.

Non possiamo usare HybridCache nei call site **sincroni**, ma solo **asincroni**.

Si, ok: potremmo usare `.Result` oppure `.GetAwaiter().GetResult()` ma... dai, su.

Anche no.

# HybridCache: no read-only

Non esistono metodi di sola lettura, quindi non è possibile semplicemente «leggere un valore».

Potrebbe sembrare possibile simularlo tramite un extension method custom che chiami `GetOrCreateAsync()` con alcune opzioni specifiche:

`github.com/dotnet/extensions/issues/5688#issuecomment-2692247434`

I problemi sono:

- in una **configurazione L1+L2** non funziona correttamente (nessuna copia da L2 a L1)
- la stampede protection è **non deterministica** (su CACHE MISS)

Riguardo al secondo punto, vediamo meglio.

# HybridCache: non deterministica

Questo è piuttosto subdolo.

Quando si chiama `GetOrCreateAsync(key, factory)` l'aspettativa è:

- se il valore **è** nella cache (CACHE HIT) viene **restituito**
- se il valore **non è** nella cache (CACHE MISS) viene **eseguita la factory**

Ma attualmente la factory **non sempre viene eseguita** su CACHE MISS.

E non c'è modo di saperlo o di controllarlo.

Vediamo un esempio concreto.

# HybridCache: non deterministica

Provando a creare un metodo read-only, potremmo creare un ext method `TryGetAsync()`:

```
public static async ValueTask<(bool Found, T? Value)> TryGetAsync<T>(this HybridCache cache, string key)
{
    var found = true;

    var value = await cache.GetOrCreateAsync(
        key,
        _ =>
        {
            found = false;
            return ValueTask.FromResult(default(T));
        },
        new HybridCacheEntryOptions()
        {
            Flags = HybridCacheEntryFlags.DisableLocalCacheWrite | HybridCacheEntryFlags.DisableDistributedCacheWrite
        }
    );

    return (found, value);
}
```

# HybridCache: non deterministica

Dichiariamo una variabile `found`, inizializzata a `true` (CACHE HIT):

```
public static async ValueTask<(bool Found, T? Value)> TryGetAsync<T>(this HybridCache cache, string key)
{
    var found = true;

    var value = await cache.GetOrCreateAsync(
        key,
        _ =>
        {
            found = false;
            return ValueTask.FromResult(default(T));
        },
        new HybridCacheEntryOptions()
        {
            Flags = HybridCacheEntryFlags.DisableLocalCacheWrite | HybridCacheEntryFlags.DisableDistributedCacheWrite
        }
    );

    return (found, value);
}
```

# HybridCache: non deterministica

Chiamiamo `GetOrCreateAsync()`:

```
public static async ValueTask<(bool Found, T? Value)> TryGetAsync<T>(this HybridCache cache, string key)
{
    var found = true;

    var value = await cache.GetOrCreateAsync(
        key,
        _ =>
        {
            found = false;
            return ValueTask.FromResult(default(T));
        },
        new HybridCacheEntryOptions()
        {
            Flags = HybridCacheEntryFlags.DisableLocalCacheWrite | HybridCacheEntryFlags.DisableDistributedCacheWrite
        }
    );

    return (found, value);
}
```

# HybridCache: non deterministica

In caso di CACHE MISS verrà eseguita la **factory**, che imposterà `found` a `false`:

```
public static async ValueTask<(bool Found, T? Value)> TryGetAsync<T>(this HybridCache cache, string key)
{
    var found = true;

    var value = await cache.GetOrCreateAsync(
        key,
        _ =>
        {
            found = false;
            return ValueTask.FromResult(default(T));
        },
        new HybridCacheEntryOptions()
        {
            Flags = HybridCacheEntryFlags.DisableLocalCacheWrite | HybridCacheEntryFlags.DisableDistributedCacheWrite
        }
    );

    return (found, value);
}
```

# HybridCache: non deterministica

E verrà ritornato un **valore di default**:

```
public static async ValueTask<(bool Found, T? Value)> TryGetAsync<T>(this HybridCache cache, string key)
{
    var found = true;

    var value = await cache.GetOrCreateAsync(
        key,
        _ =>
        {
            found = false;
            return ValueTask.FromResult(default(T));
        },
        new HybridCacheEntryOptions()
        {
            Flags = HybridCacheEntryFlags.DisableLocalCacheWrite | HybridCacheEntryFlags.DisableDistributedCacheWrite
        }
    );

    return (found, value);
}
```

# HybridCache: non deterministica

E, per evitare di «sporcare la cache», verrà **disabilitata** la **scrittura** nella cache:

```
public static async ValueTask<(bool Found, T? Value)> TryGetAsync<T>(this HybridCache cache, string key)
{
    var found = true;

    var value = await cache.GetOrCreateAsync(
        key,
        _ =>
        {
            found = false;
            return ValueTask.FromResult(default(T));
        },
        new HybridCacheEntryOptions()
        {
            Flags = HybridCacheEntryFlags.DisableLocalCacheWrite | HybridCacheEntryFlags.DisableDistributedCacheWrite
        }
    );

    return (found, value);
}
```

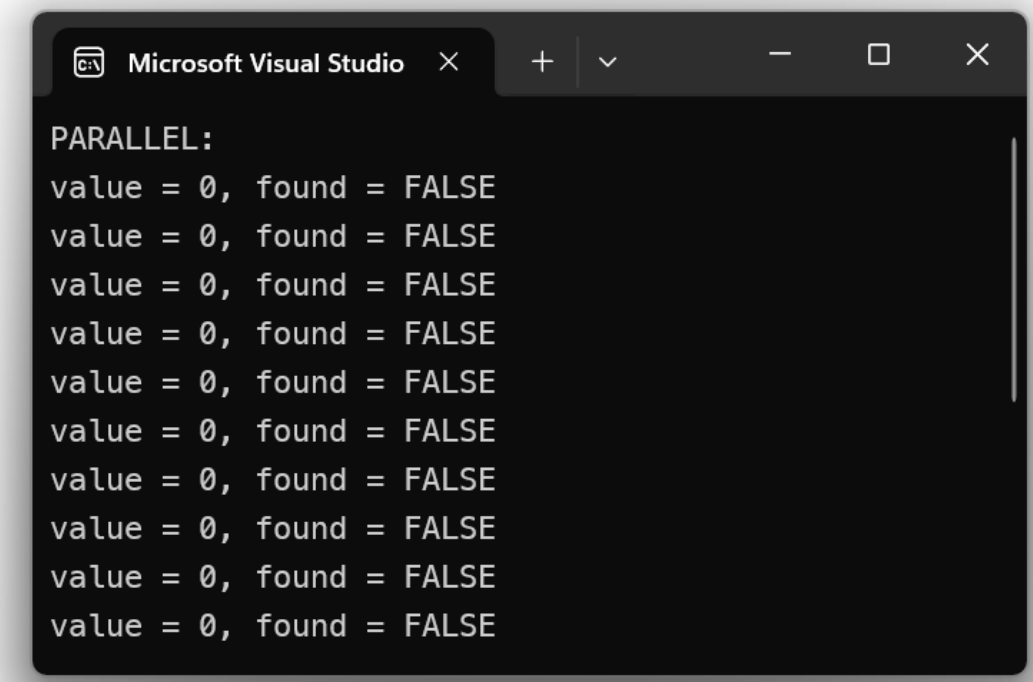
# HybridCache: non deterministica

Supponiamo di fare chiamate in **parallelo**, con `int` come tipo:

```
Console.WriteLine("PARALLEL:");
await Parallel.ForAsync(0, 10, async (_, _) =>
{
    var foo = await cache.TryGetAsync<int>("foo");
    Console.WriteLine($"value = {foo.Value}, found = {foo.Found.ToString().ToUpper()}");
});
```

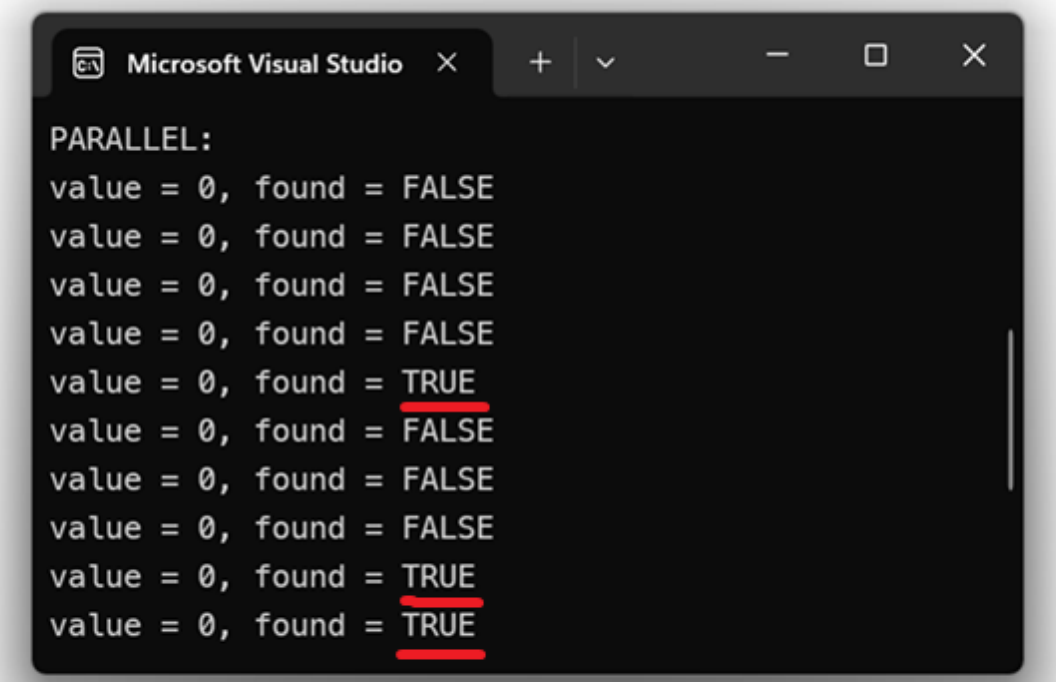
# HybridCache: non deterministica

Ci aspetteremmo **questo**:



# HybridCache: non deterministica

... e **invece** otterremmo questo (sequenza diversa ad ogni esecuzione):




Fondamentalmente, il comportamento è **non deterministico** (su cache miss).

# HybridCache: incoerente su più nodi

Attualmente HybridCache non ha qualcosa come il **Backplane** in FusionCache.

Quindi **non può notificare** gli altri nodi riguardo update o altro lasciando la cache, nel suo complesso, **incoerente** fino alla scadenza naturale del dato.

Come mitigazione possiamo diminuire `LocalCacheExpiration`, ma non è una vera soluzione.

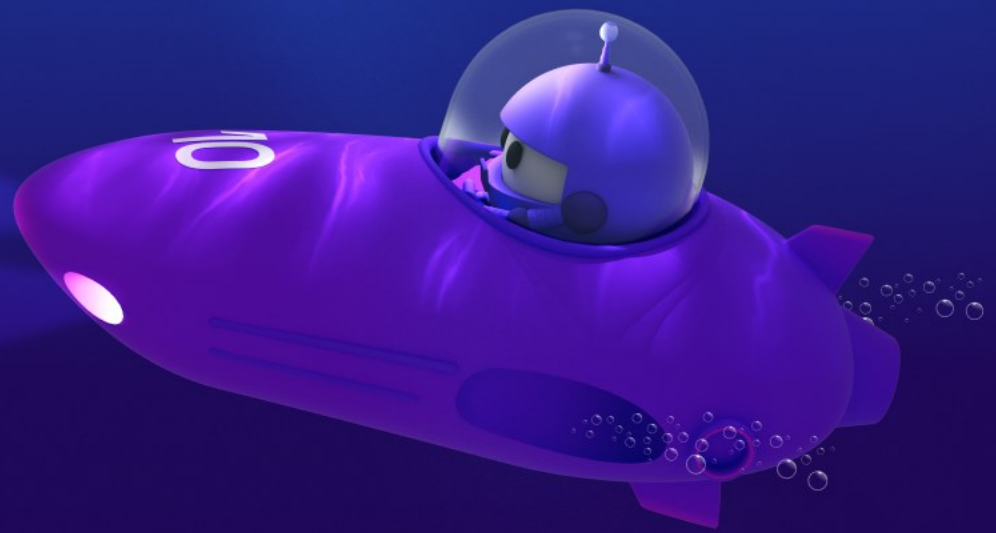
Imho: l'attuale implementazione di default di HybridCache non è **usabile** con più di 1 nodo.

# HybridCache: Limitazioni E Problemi

Dicevamo: quindi, c’è una soluzione?

Oh yeah.

# AsHybridCache()



#  AsHybridCache()

Dunque, abbiamo visto come `HybridCache` sia due cose:

- una **astrazione**
- una **implementazione di default**

E in quanto astrazione, **altre implementazioni** sono possibili.

#  AsHybridCache()

Dunque, abbiamo visto come `HybridCache` sia due cose:

- una **astrazione**
- una **implementazione di default**

E in quanto astrazione, **altre implementazioni** sono possibili.

E `FusionCache` è una cache **ibrida**.

# 🦥 AsHybridCache()

Dunque, abbiamo visto come `HybridCache` sia due cose:

- una **astrazione**
- una **implementazione di default**

E in quanto astrazione, **altre implementazioni** sono possibili.

E `FusionCache` è una cache **ibrida**.

Ooooh 😌

#  AsHybridCache()

FusionCache resta un progetto indipendente, a sé stante.

Questo non cambia.

Ma lavorare con un'astrazione **condivisa** e parte di .NET stesso può avere valore.

Quindi FusionCache può essere utilizzata **anche** come implementazione della nuova astrazione `HybridCache`, il tutto mantenendo le **funzionalità extra** di FusionCache.

E tutto grazie a un piccolo **adapter**.

Come?

# AsHybridCache()

Così:

```
services.AddFusionCache()
  .AsHybridCache(); // MAGIC
```

Nessuna modifica al codice necessaria, da nessuna parte.

#  AsHybridCache()

Ricordate tutti quei (*)?

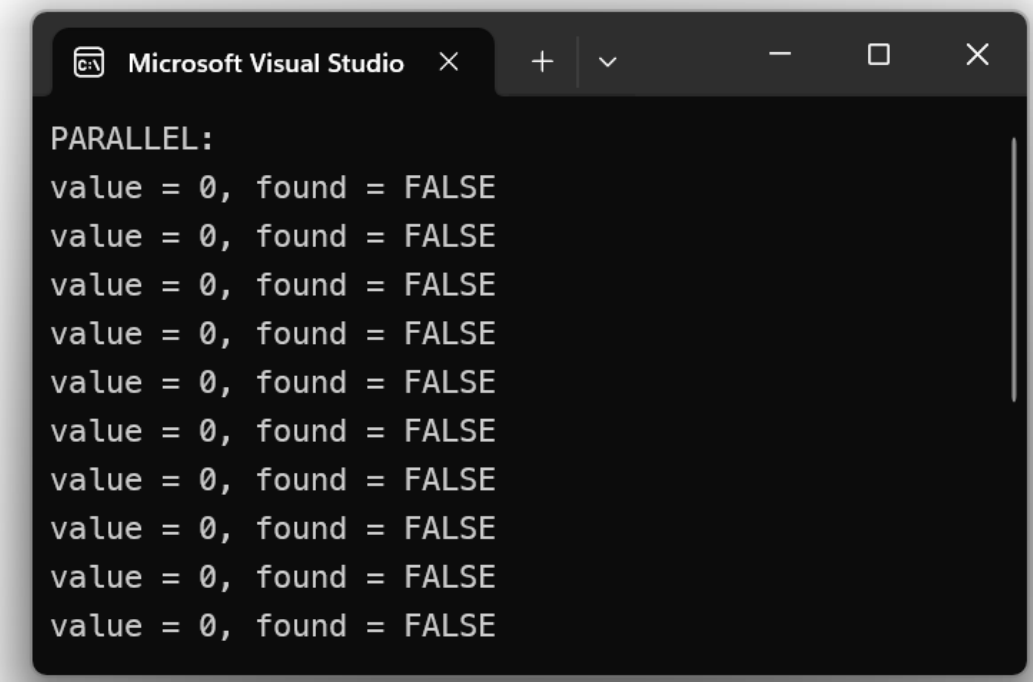
Spariti.

Utilizzando FusionCache come implementazione di HybridCache otteniamo:

- **cache stampede:** unificata + deterministica + sync/async
- **controllo:** controllo completo su L1/L2
- **backplane:** notifiche distribuite e istantanee multi-nodo
- **cache sempre coerente:** bye bye nodi fuori sync, incluso per Tagging
- **cache multiple:** possiamo usare Named Caches con Keyed Services
- **extra feature:** Fail-Safe, Eager Refresh, Factory Timeouts, Auto-Recovery, ecc

# AsHybridCache()

Riguardo al **non determinismo**, ora è così, sempre:

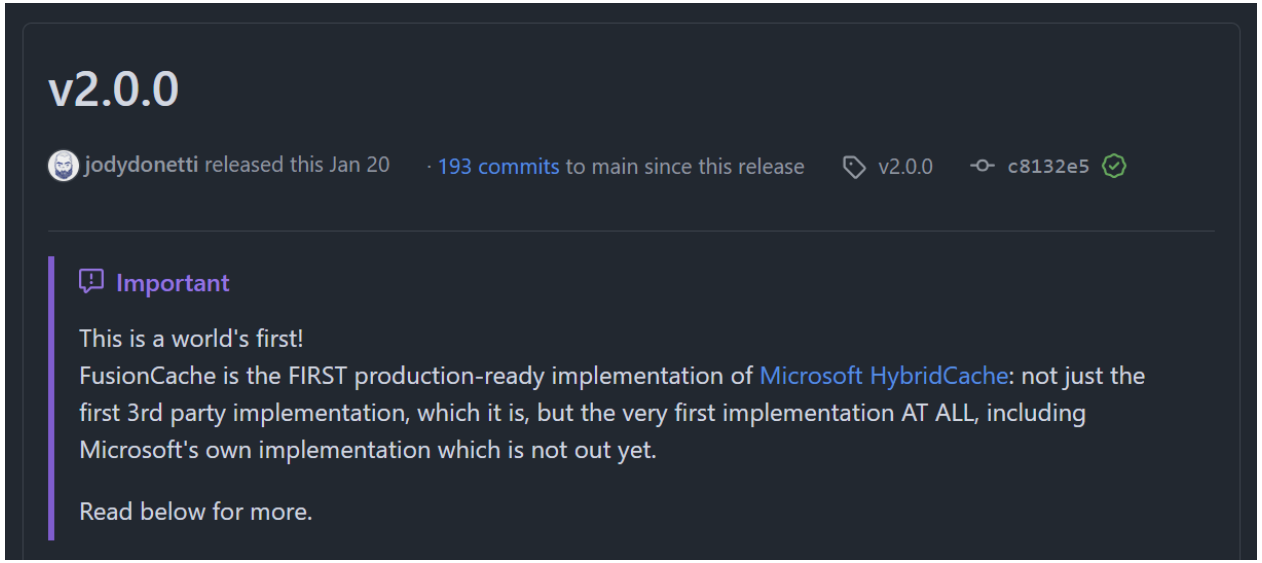


# AsHybridCache()

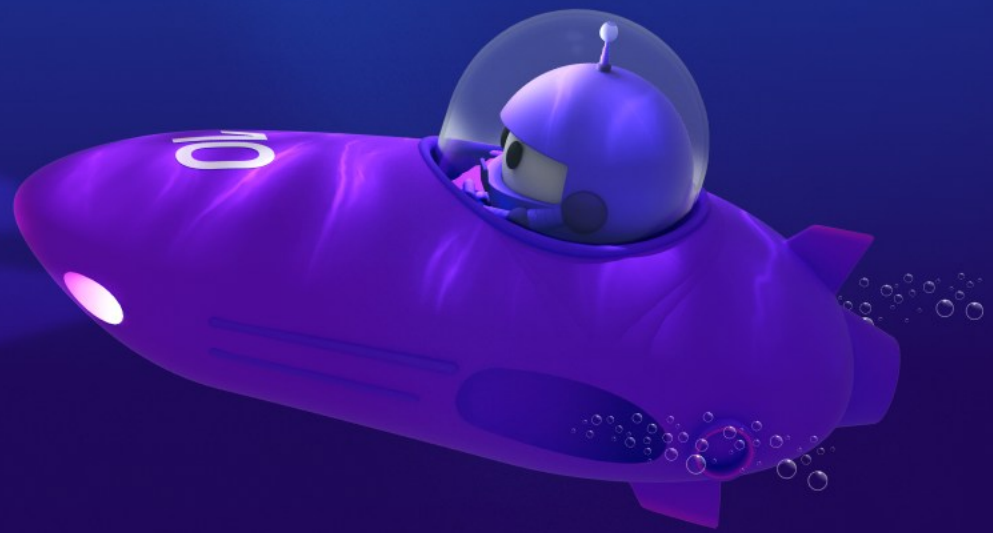
Timeline:

- **Nov 2024:** rilascio astrazione HybridCache (con .NET 9)
- **Gen 2025:** rilascio FusionCache v2 (con HybridCache adapter)
- **Mar 2025:** rilascio implementazione HybridCache di default (Microsoft)

Prima implementazione production-ready al mondo



# Quindi, hybrid caching?



# 🚀 Quindi, hybrid caching?

Per passare al caching ibrido possiamo:

- usare **FusionCache**, direttamente
- usare l'astrazione **HybridCache**, con l'**implementazione default** di Microsoft (*)
- usare l'astrazione **HybridCache**, con l'implementazione **FusionCache**

Abbiamo diverse scelte, sfruttiamole al meglio.

# Grazie!

github.com/jodydonetti

twitter.com/jodydonetti

linkedin.com/in/jody-donetti

# Su Dometrain:

# Feedback, please 🥲