Gabriel Gregori

The Pragmatic Programmer's Playbook: From Blazor to Scalability, Diet to Architecture

EN

Unlock the secrets to building robust, high-performance software and optimizing your productivity. This book distills hard-won lessons from real-world projects, from mastering Blazor at scale to leveraging event sourcing and clean architecture, while also revealing how a ketogenic diet can supercharge your coding focus. No fluff, just actionable insights for developers who want to ship better code and perform at their peak.

June 18, 2026

Contents

Chapter 1: Why Do Companies Choose Blazor for Their Projects but Fail When it Scale?

> *Chapter 1: Why Do Companies Choose Blazor for Their Projects but Fail When It Scales? – Learn from common pitfalls and proven strategies to make Blazor work at scale, based on hands-on experience from multiple projects and collaborations.*

Why Do Companies Choose Blazor for Their Projects?

With Blazor, especially when using frameworks such as Syncfusion, **development becomes extremely fast and easy to maintain while maintaining a very high level of quality.**

With practically a Ctrl+C and Ctrl+V, you can implement an amazing Kanban board and focus only on the project's input and output variables in @Code section instead of spending time managing events, UI logic and components life cycles.

```
<SfKanban CssClass="kanban-overview" KeyField="Status" DataSource="@CardData"
EnableTooltip="true">
    <KanbanColumns>
    </KanbanColumns>
    <KanbanCardSettings ContentField="Summary" HeaderField="Title"
SelectionType="@SelectionType.Multiple">
    </KanbanCardSettings>
    <KanbanSwimlaneSettings KeyField="Assignee"></KanbanSwimlaneSettings>
</SfKanban>@code{
    private List<KanbanDataModel> CardData = new KanbanDataModel().GetCardTasks();
    private List<ColumnModel> ColumnData = new List<ColumnModel>()
    {
        new ColumnModel(){ HeaderText= "To Do", KeyField= new List<string>()
{ "Open" }, AllowToggle= true },
        new ColumnModel(){ HeaderText= "In Progress", KeyField= new List<string>()
{ "In Progress" }, AllowToggle= true },
        new ColumnModel(){ HeaderText= "In Review", KeyField= new List<string>()
{ "Review" }, AllowToggle= true },
        new ColumnModel(){ HeaderText= "Done", KeyField= new List<string>()
{ "Close" }, AllowToggle= true }
    };
}
```

This code results

Documentation and real demo: https://blazor.syncfusion.com/demos/kanban/overview?theme=bootstrap5

One thing worth mentioning is that we're not even talking about Copilot, Claude Code, or Codex yet.

Long before AI agents existed, **Blazor was already a productivity hack.**

With Blazor, we could build an entire portal in two weeks because we already had **entity scaffolding**. We would simply create an entity with its properties, and then generate Update, Insert, Delete, and List pages for that entity, already including basic validations, ASP.NET Identity permissions, and a responsive Bootstrap layout.

After that, we only had to implement some business validations and custom logic.

Today, when we combine that reality with AI agents, we're talking about an insane level of productivity, the amount of software that can be produced is becoming absurd it creates a new challenge: having enough mental energy and focus to keep up with the speed at which things can now be built, that's why I made article about **Bio hack for brain energy using ketones** https://www.gabrielgregori.com/Articles/Details/ketogenic-diet-the-hack-for-programmers-productivity

A Unified Stack

Another major advantage is not having to constantly keep up with multiple technologies.

Instead of mastering a back-end stack and a completely different front-end stack, you can use C# throughout the entire application.

You can also reuse entities, validations, and business logic directly in the front-end, sharing code across different layers of the system.

All of this can be done from a single IDE with seamless integration.

Every 6 months Microsoft launch a new .NET version with a new C# version and many features, imagine having to keep updated with all that at same time keep being updated with a whole different ecosystem like Angular, Javascript, TypeScript etc.

If you only care for Blazor you can focus more on learning design patterns, concepts rather than keep updated into multiple languages and frameworks.

Less Complexity

Depending on the architecture you choose, you may not even need an API.

With Blazor, it is possible to inject services and repositories directly into the application. If an API is needed, it can always be created as a separate project.

This flexibility can significantly reduce the initial complexity of many applications.

Integration with Visual Studio and Azure

The integration with Visual Studio and Azure is excellent.

With only a few clicks, you can create a Blazor project using Microsoft's templates and publish it to Azure shortly afterward.

For teams already working with .NET, this dramatically reduces setup and deployment time.

The Same .NET Ecosystem

The same packages you already use on the back-end can often be used on the front-end as well.

Everything is managed through NuGet.

This simplifies maintenance and reduces the number of technologies the team must learn and support.

Back-End Developers Building Front-End Applications

One of the things I like most about Blazor is that back-end developers can build modern front-end applications with a very low learning curve.

Using component libraries such as Syncfusion or Telerik, developers can create dashboards, grids, charts, forms, and complex interfaces without becoming experts in JavaScript frameworks.

MAUI Blazor Hybrid

There is a Visual Studio template called MAUI Blazor Hybrid.

With it, you can reuse both your back-end and front-end code and generate applications for:

- Web
- Windows
- macOS
- Android
- iOS

All while using virtually the same codebase, the same IDE, and the same technologies.

From a productivity standpoint, it is difficult to find something comparable within the Microsoft ecosystem.

More Flexible Teams

There is also an organizational benefit.

When a team member takes leave due to vacation, health issues, parental leave, or any other reason, another developer can take over with much less effort.

Because the entire application revolves around the same ecosystem, knowledge transfer becomes significantly easier.

Continuous Platform Evolution

By choosing Blazor, you are investing in a platform that is actively supported by Microsoft.

New features are constantly being introduced.

A good example was the introduction of Render Modes in .NET 8.

With Render Modes, you can decide whether a component should execute on the server or on the client.

This makes it possible to build much smarter architectures that balance productivity, user experience, and resource consumption.

A Concise List of Blazor Advantages

- Full-stack development using only C#.
- Reuse of entities, validations, and business rules between back-end and front-end including case of multiple platforms.
- Reduced code duplication and increased productivity.
- Rapid UI development using component libraries such as Syncfusion and Telerik.
- Native integration with Visual Studio, Identity, Dependency Injection, and the .NET ecosystem.
- Simplified deployment to Azure.
- Automatic CRUD generation through scaffolding.
- Blazor Webassembly support offline use
- Render modes allow use SSR, Server and Client rendering.

So Why Do Some Companies Fail with Blazor?

I have personally seen companies abandon Blazor.

I also have friends working in large corporations, including the automotive industry, that moved away from Blazor primarily because of performance concerns.

Last year, when my assignment with a client ended, I participated in both internal and external interviews.

One thing I found interesting was that several companies were interested in me for one specific reason: understanding how to make Blazor projects successful in production.

The Real Problem

Without marketing and without buzzwords, the problem is usually very simple:

The websocket from SignalR scalling for every single user.

I have seen software architects, senior engineers, and professionals with decades of experience

completely overlook this detail.

A traditional JavaScript application mostly communicates through HTTP requests.

Blazor Server, on the other hand, maintains a persistent SignalR connection between the client and the server.

Each connected user represents an active connection.

The Cost of Scaling

While a traditional application mostly handles GET, POST, PUT, and DELETE requests for example when you enter a page in a traditional app it sends an GET Request then Server respond with a cached query while in a Blazor Server it maintains a continuous communication channel with users.

If you have 1,000 connected users, you roughly have 1,000 active WebSocket connections sending and receiving events and updating DOM.

There is a lot of engineering behind this process.

I do not need to understand every internal detail to understand the final result: CPU, memory, and server resource consumption.

This is where many projects begin to experience performance issues.

The Most Common Mistake

The developer sees performance problems and concludes:

"Blazor doesn't scale."

Then they abandon the technology and move to a JavaScript framework.

Another error it's putting javascript developers to work with Blazor without training, it's completely different the good practices.

How to Solve It

Today there are several ways to take advantage of Blazor without relying exclusively on Blazor Server.

You can use:

- Blazor WebAssembly consuming APIs
- Render Modes
- Hybrid architectures
- Client-side components
- Server-side components

Internal dashboards with a limited number of users may work perfectly with Blazor Server.

Public-facing features with large traffic volumes may be better suited for WebAssembly with API or Razor Pages (Blazor uses .razor components).

The Most Underrated Framework in the .NET Ecosystem

Now comes what I believe is the most interesting insight.

ASP.NET Razor Pages

Razor Pages is probably one of the most underrated frameworks in the .NET ecosystem.

It offers many of the same characteristics that attract developers to Blazor.

You still get:

- C#
- .NET
- Razor components
- Shared code
- Libraries such as Syncfusion

But with significantly lower computational overhead for public, high-traffic applications.

This is because it does not rely on persistent SignalR connections in the same way that Blazor Server does.

How It Works

With Razor Pages, you have a native HTTP-based architecture.

Each page has its own view and code-behind responsible for handling requests.

It is an extremely efficient solution for public-facing pages that receive a large volume of traffic.

How This Website Was Built

The website you are currently reading uses a hybrid architecture.

Public pages are built with Razor Pages.

The dashboard uses Blazor Render Modes.

Charts and heavier client-side features are executed in the browser when appropriate.

Administrative features, such as article management, use server-side rendering to provide a smoother SPA-like experience.

And if necessary, I can still integrate JavaScript libraries and synchronize them with C# with relatively

little effort.

Chapter 2: Ketogenic Diet the BioHack for Programmers Productivity

> *Chapter 2: Ketogenic Diet the BioHack for Programmers' Productivity – Discover how your brain runs on ketones, giving you sustained energy and mental clarity that rivals performance-enhancing focus, turning your diet into a cognitive superpower.*

Ketones for brain, a source of energy much better than glucose

For a programmer, it is gold: less mental fatigue and more consistent energy throughout the day, without crashes. Unlike glucose, which often comes with spikes and drops in energy and focus, ketones provide a cleaner and more stable fuel source for the brain. Many people report improved concentration, mental clarity, sustained productivity, and fewer distractions during long coding sessions.

Why are ketones a superior source of energy?

Humans learned to farm 10 thousand years ago, while our bodies evolved over millions of years by eating animal meat.

Because of that, when we eat carbs our blood glucose spikes; our bodies are still not adapted to carbs, and we will need much more time to adapt.

When eating carbs, our body produces free radicals (which are bad), meanwhile when eating fat, it produces antioxidants.

Also, about 80% of our brain's structural breakdown is made of lipid pathways, meaning it thrives on ketones as a premium fuel source. A famous case of this involves Dr. Mary Newport, a medical doctor whose husband, Steve, suffered from severe early-onset Alzheimer's disease to the point where he faced debilitating motor control issues. After she began feeding him therapeutic doses of coconut oil and MCT oil to elevate his ketone levels, he experienced a remarkable recovery in his movements, visual processing, and was even able to drive again. She documented this incredible journey in her book, *"Alzheimer's Disease: What If There Was a Cure? The Story of Ketones."*

The Story of Ketones - Mary T. Newport

Ketones are so powerful they can cure or control diabetes

I was diagnosed with Type 2 diabetes, and everyone on my mother's side of the family is diabetic. When I did the ketogenic diet for some months, my next diagnosis showed nothing at all. My blood sugar used to be around 140 when I woke up, and it dropped to around 90.

It is actually so powerful that when combined with intermittent fasting, many researchers are studying it as a complementary metabolic therapy for cancer. This approach capitalizes on the **Warburg**

Effectthe scientific observation that most cancer cells rely heavily on glucose fermentation for survival and cannot effectively utilize ketones for fuel. By restricting carbohydrates and inducing fasting-driven **autophagy** (the body's cellular cleanup process), you effectively starve the metabolic pathways of tumor cells while keeping healthy brain and body tissue fully energized.

Overall benefits of the Ketogenic diet

- **More stable energy levels** throughout the day
- **Reduced brain fog** and improved mental clarity
- **Better focus and concentration** for deep work
- **Lower hunger** and fewer cravings between meals
- **Improved insulin sensitivity** and blood sugar control
- **Supports fat loss** by increasing fat utilization as fuel
- **Fewer energy crashes** compared to high-carb diets
- **May reduce inflammation** and improve metabolic health
- **Can improve endurance** and steady-state physical performance
- **More consistent mood** and motivation
- **May support longevity** and healthy aging pathways
- **Can increase metabolic flexibility** between glucose and fat usage

How does the ancient diet of our ancestors connect with AI Agents such as Codex and Claude?

By working at a high level with ChatGPT, then writing MD files and implementing them with Claude after reviewing every single line, we are all exhausted. With so much more information to process, the challenge today is energy not just for AI roadmap goals, but for the human brain itself.

Chapter 3: Event Sourcing Made Simple (with .NET Examples)

> *Chapter 3: Event Sourcing Made Simple (with .NET Examples) – Demystify event sourcing with straightforward .NET examples, cutting through academic jargon so you can apply it immediately to your own projects.*

What is Event Sourcing?

Instead of **saving the final state**, you store events that, when processed, result in the **final state**.

That simple? Yes!

This concept can be expanded, but let's start simple. Check this aggregate:

```
public class BankAccount
{
    private decimal balance { get; set; } = 0;
    List<Event> changes = new();

    void When(Event @event)
    {
        switch (@event)
        {
            case Withdraw:
                balance -= @event.amount;
                break;

            case Deposit:
                balance += @event.amount;
                break;
        }
    }
}
```

You can see that the state for attribute balance is defined by processing events instead of being retrieved directly from a database query. That's Event Sourcing. We store objects (events) that represent state changes over entities.

In a banking system, we can create events such as:

- Withdrawn
- Deposit
- TaxApplied
- LimitChanged

- Transfer
- And many others

These events are stored in an event log, which can be a simple `List<Event>` or a more robust solution like an Event Store (people implement different approaches).

The most important concept in Event Sourcing is that the event log is **immutable (IT IS OUR SOURCE OF TRUTH WHEN TALKING ABOUT STATE!)**. There are no updates or deletes. In practice, your database should enforce a policy that prevents `UPDATE` and `DELETE` operations on this table.

Performance

Imagine a bank account after 20 years with hundreds of thousands of operations, multiplied by millions of users. Replaying all events every time would create significant overhead.

That's where design patterns come in to improve performance.

Projections

A projection keeps the current (final) state of an entity.

For example, every time a new event happens in `BankAccount`, you update a `BankAccountProjection` with the latest balance. So instead of recalculating everything, you always have a ready-to-use state. Every time a event happen you update the BankAccountProjection

If you suspect inconsistency, you can replay all events and rebuild the state for a projection, or even having different projections with different events computation. This process is called **event replay**.

Snapshot

How do you replay 500,000 events for millions of users efficiently?

You don't. You use snapshots.

Every N events (e.g., 50 or 500), you store a snapshot of the current state, it save which aggregate version the snapshot was made and all the state for that aggregate.

When rebuilding, you start from the latest snapshot instead of from the beginning. For example, if a snapshot exists at event 550,000, you only replay the remaining events after that, because at that point the state was reliable.

This significantly reduces computation cost.

Cool things about Event Sourcing

- Supports offline-first applications (events can be synced later)
- Full audit history of all changes
- Ability to rebuild past states (time travel)

- Useful for analytics, data science, and big data
- Can simulate or project future states

Concepts people will talk about

Eventual Consistency

This is a conscious decision that data might not be immediately consistent, but will become consistent over time.

Example: You finish watching something on Netflix, but your phone still shows it as incomplete. After a few seconds, systems sync and the state becomes correct.

Optimistic Concurrency

- Add a `version` field to the aggregate
- Each event expects a specific version

Example:

A Withdraw event of $500 expects the account to be at version 203. When saving, the system checks the current version in the database.

If the version is not 203, an exception is thrown, and the user must retry the operation.

Tools in the .NET ecosystem

- **Marten** – A NoSQL database built for Event Sourcing with many built-in features
- **Azure Cosmos DB** – Reliable storage for events
- **Azure Service Bus** – Helps distribute events across systems
- **MediatoR & CQRS** - Helps easily notify events with separation of write and read operations

My all-time favorite Event Sourcing article

https://martinfowler.com/eaaDev/EventSourcing.html

When I was working as a contractor Software Engineer, I was basically desperate. I had to build a microservice using Event Sourcing with projections, and nothing made sense at the time.

After reading Martin Fowler's article and seeing the "Ship" examples, everything started to click. It made so much sense that I didn't even finish reading the whole article.

In just a week, I had:

- An event log in place
- State being built from event computation

- A projection storing the final state

That article was a turning point for me in understanding Event Sourcing in practice.

Event Sourcing Training

To reinforce and revisit these concepts in practice, I created training project focused on Event Sourcing using optimistic concurrency, projections, snapshot, cosmos DB Simulator, Azure Event Bus Queue.

The project was built for I training myself.

View the Event Sourcing Training Repository on GitHub

Chapter 4: How I Would Improve the Performance of a High-Traffic API

> *Chapter 4: How I Would Improve the Performance of a High-Traffic API – Borrow battle-tested performance tricks from top companies and freelancers to optimize high-traffic APIs, shared from years of real-world experience.*

How I Would Improve the Performance of a High-Traffic API

First thing, I don't start changing code right away. I need to understand how the API behaves from **START** to the very **END**.

So I MEASURE(Key).

I'd use tools like **.NET benchmarking** (Stopwatch is BAD!) or logs, graphics, insights from Azure Monitoring, Google Analytics and Cloudflare Dashboard to see what's actually slow. No guessing here I want to know where time is being spent.

Understanding the Bottleneck

After that, I separate things into two main types:

- I/O operations
- CPU work

For I/O, I'm looking at things like:

- Database queries
- External API calls
- Disk/file access

For CPU:

- Loops
- Data processing
- Any heavy computation

This helps me understand where to focus.

Code-Level Improvements

Now I start improving the code itself.

If it's I/O heavy:

- Make sure everything that can be async is actually async
- Avoid blocking threads

If it's CPU heavy:

- Use parallel programming where it makes sense

Also important to see how many threads the CPU Server has to make use of Parallel.ForEach library

Using GPU for parallel programming it is a card for VERY EXTREME cases and I never needed to used it, but it's a rare field called *GPU Computing for General Cases*

I also review:

- Data structures (try to go from O(n) to O(1) when possible), and AI today can scan and help identify improvements.
- Memory usage and garbage collector pressure
- Instead using a class sometimes we can use record or struct
- Immutable data structures always give gains even if marginals but when 100.000 users hit API per second it make a difference (100k * 0.01)

Sometimes small changes here already give a big gain.

Reducing Load with Caching

If the API is being hit a lot, caching helps a lot.

- Use Redis for distributed cache
- Cache responses that don't change often
- Use static files if possible

This alone can remove a huge amount of load from the system.

Payload and Communication

If the payload is too big, I look into reducing it.

In some cases, switching from REST to gRPC can help because it reduces payload size a lot (sometimes even around 40%).

Not always necessary, but it's an option.

Database Optimization

A lot of performance issues come from the database.

- Analyze queries
- Rewrite them if needed
- Reduce unnecessary joins
- Check how data is being accessed

If needed, I can change the approach:

- Use CQRS to separate reads and writes
- Use something like MongoDB for faster reads depending on the use case

Technology Stack

In the beginning of my path in IT, I used to work with PHP. It was the only language I had experience with at the time.

When I started working on heavy data processing in a real estate application, I began to hit PHP's limits pretty hard.

At that time, PHP didn't offer strong support for async operations, multithreading, or efficient data structures. Garbage collection and performance optimizations were also limited, and on top of that, it's an interpreted language.

To solve a specific problem, I wrote part of the processing in C, which is a **compiled** and very fast language. I used it to run scripts that populated the real estate database from CSV files.

What was taking days in PHP ended up taking just minutes in C.

That experience taught me an important lesson: each language has its own domain.

If you're dealing with a large-scale application or heavy traffic, you need to choose the right language and framework for the job. In some cases, it makes sense to combine approaches for example, using a more performant language for specific endpoints or workloads, even inside a larger system architecture.

Once the code is in a good place, I look at infrastructure.

Questions I ask:

- Does the server have enough CPU/RAM?
- Is disk speed an issue?

Then scaling:

- Vertical scaling (better machine)
- Horizontal scaling (more instances + load balancer)

Latency and Location

Location matters more than people think.

If my users are in one region but the server/database is far away, latency will hurt performance.

- Keep API and database close
- Use CDNs for global distribution
- Use geographically distributed databases if needed

Handling Traffic

- Use load balancers to distribute requests
- Avoid overloading a single instance

AI Age

Brain storm with chatGPT ideas about how to improve and research from blogs with him techniques

Claude Code it's being used Linus Torvald and Donald Knuth, do with responsibility changes in a instant, where it would take a month

Even for this article, I asked ChatGPT and Gemini to review it and help me improve the grammar, logic, and coherence. I'm not ashamed of that, and people are reaching me.

Final Thought

For me, it always comes down to this:

1. Measure first
2. Fix the real bottleneck
3. Then scale

Not everything needs Redis, gRPC, or CQRS. Sometimes the problem is just a bad query or a blocking call.

Chapter 5: Easily understand async/await, multithreading, threads and parallel processing

> *Chapter 5: Always Use Clean Architecture Even if You Don't Need It – Embrace clean architecture as a mastery tool that simplifies complex systems, making every future project easier to maintain and extend.*

The Trick That Makes Applications Scale

- **I/O-bound operations (API Call, DB Call or Read file)** use async/await
- **CPU-bound operations (For loops or process file)** use parallel processing ex: Parallel.ForEachAsync (.NET)

Now understanding the trick

I/O Operations: Use Async/Await

For I/O operations such as API calls, database queries, or file access, use **async/await**.

These operations spend most of their time waiting for external systems. Async programming allows the thread to return to the **thread pool** while the I/O operation is in progress.

This prevents threads from being blocked and allows the system to process more requests concurrently.

CPU Intensive Work: Use Parallel Processing

For CPU-heavy tasks, such as large data processing or complex calculations, parallel processing can improve performance.

In .NET, a common example is:

```
Parallel.ForEach(items, item ⇒
{
    Process(item);
});
```

This distributes work across multiple CPU cores, allowing tasks to execute simultaneously.

Before using parallelism, it's important to verify that your server actually has multiple CPU cores available.

Prefer Task Over Creating Threads

In modern .NET applications, developers rarely create threads manually. Instead, they use **Task**, which

is a higher-level abstraction that integrates with the **Thread Pool**.

Tasks provide better resource management and work naturally with async and parallel programming patterns.

Understanding Threads

What Is a Thread?

A thread is the **smallest unit of execution inside a process**.

It represents a sequence of instructions that the CPU executes. Threads allow programs to perform multiple tasks concurrently.

Hardware Threads (CPU Threads)

A CPU thread is a **hardware execution unit inside the processor** capable of executing an instruction stream.

Modern CPUs include:

- Multiple **cores**
- Multiple **hardware threads per core** (for example using Hyper-Threading)

Example:

- 4 cores
- 8 hardware threads

This means the CPU can execute up to **8 instruction streams concurrently**, depending on workload and scheduling.

Threads in Programming

In programming, a thread is a **software abstraction** that represents a sequence of instructions we want to execute independently.

When a thread is created in code, the program requests the **operating system scheduler** to allocate CPU time for that thread.

Example in C#:

```
new Thread(() ⇒
{
    Console.WriteLine("Hello world");
}).Start();

new Thread(() ⇒
```

```
{
    Console.WriteLine("Hello world");
}).Start();
```

These threads may run concurrently depending on how the operating system schedules them.

Threads at the Operating System Level

At the operating system level, a thread is still the **smallest unit of execution within a process**.

The operating system is responsible for:

- Scheduling threads
- Assigning them to CPU hardware threads
- Managing execution
- Performing context switching

Thread Memory Model

Each thread contains its own:

- **Stack** – used for local variables and method calls
- **CPU registers** – temporary storage during execution
- **Instruction pointer** – indicates the next instruction to execute
- **Execution state** – running, waiting, or blocked

However, threads inside the same process share the same memory space.

This shared memory allows threads to collaborate, but it can also introduce concurrency problems such as **race conditions**.

Key Takeaway

High-performance applications scale by using the correct strategy for each type of workload:

- **Async/await** for I/O-bound operations
- **Parallel processing** for CPU-bound workloads
- **Task and the Thread Pool** instead of manually managing threads

Understanding these concepts is essential for don't rely entirely in expensive cloud horizontal and vertical scale.

Chapter 6: Always use Clean Architecture even if you don't need it.

> *Chapter 6: Easily Understand Async/Await, Multithreading, Threads, and Parallel Processing – Finally grasp the big picture of concurrency with clear explanations, so you can boost application performance correctly from day one.*

Clean Architecture (CA) has become a widely adopted and beloved standard in the development community. Even if it doesn't make much sense to use it in some cases, **every developer should get familiar with it**. At the end of the day, what dominates the industry almost always becomes the rule, regardless of whether you agree or not.

Even for simple CRUDs

Throughout my career, I've always worked on projects with over-engineering—projects that were just 4 pages of tables with CRUD operations but had multiple layers of abstraction. That's why I advise: don't rely on simple solutions for personal or small projects; instead, use what is dominant in the industry, like Clean Architecture. When you really need it, you'll be ready to perform with over-engineering or even pass a tough interview.

The value of what is socially dominant

The main point is: **it's not just about what makes technical sense**, but also about what is socially dominant in the industry. Learning and mastering Clean Architecture means you'll be prepared for when your manager throws you into a sprint board with dozens of tasks in a system full of over-engineering and tight deadlines.

Mastering is better than resisting

Even if you are building a simple CRUD in .NET, it's worth structuring the project following Clean Architecture. This way, you gain practical experience with something that challenges developers of all levels.

Ready for any challenge

Once we tackle a complex feature within a system using Clean Architecture, we soon deal with responsibilities of each layer, such as making a user entity in the domain work with Identity and EF Core without making the domain layer depend on NuGet packages. The earlier we solve these challenges, the less stuck we get in the future.

People who say they are against live in another reality

In reality, most developers don't choose the architecture of the systems they work on. Software engineers at the lower levels of the hierarchy usually don't make those decisions they follow the direction defined by technical leadership.

At some point in your career, you will probably be assigned to a project that uses Clean Architecture without any necessity. That is simply how the industry works.

I remember once being assigned to a very complex microservices project built around Clean Architecture and everything else Kubernetes, DDD, Vertical Slice, Event-Sourcing. This is not unusual. Many large tech companies adopted certain architectural patterns, and eventually the rest of the market followed.

You might go through an interview process with hundreds of candidates just to join a project that is heavily over-engineered. At that point, you are there to deliver software not to challenge the architectural decisions. That is the reality of the software industry.