You are asking *the hottest backend architecture question of 2025!*

Let's break it down clearly, deeply, and practically, because this is what top backend architects are deciding right now.

Quick Answer:

Aspect	Spring WebFlux	Spring MVC + Virtual Threads (Project Loom)
Programming Model	Reactive (Mono/Flux, stream thinking)	Imperative (normal blocking style)
Code Style	Complex for beginners	Very simple, classic MVC
IO Model	Non-blocking, event-driven (Netty/NIO)	Blocking, but each blocking call uses cheap virtual thread
Performance under high load	Excellent (if done right)	Very good (much easier to write)
Complexity of App	Higher	Lower
Error handling	Reactive error flows needed	Normal try-catch, simpler error handling
Ecosystem/tools/libraries	Must be reactive everywhere	Works with existing libraries (blocking too!)

Virtual Threads solve the thread-scaling bottleneck, but they do NOT magically make blocking IO non-blocking.

WebFlux remains non-blocking end-to-end; Virtual Threads simply make blocking IO cheaper and scalable.

Deeper Explanation:

1. What are Virtual Threads?

- Java 21 introduces Project Loom: Virtual Threads.
- A Virtual Thread is a very lightweight thread (managed by JVM, not OS).
- You can spin millions of virtual threads without killing your machine.
- But: If you block on IO (e.g., JDBC), virtual thread suspends efficiently (no OS thread waste).

No need for complex async programming (like CompletableFuture hell).

2. Why did we need WebFlux before Loom?

- Blocking IO = heavy OS thread usage.

- Scaling 10,000+ concurrent connections required non-blocking servers (Netty, Vert.x, etc.).

- WebFlux + Netty solved that with event-driven, non-blocking IO.

3. Now with Virtual Threads (Spring MVC), what happens?

- You write normal blocking code (imperative, easy).

- Under the hood, virtual thread suspends and wakes up efficiently.

- No wastage of heavy OS threads.

- Scaling to millions of concurrent IO-bound requests becomes practical.

No need to redesign everything into Mono/Flux!

No need for "callback hell" or complex reactive chains!

OK, but now: which one should I use?

Question	Recommendation
New project, familiar team (Spring MVC experience)?	Spring MVC + Virtual Threads
Working with legacy blocking libraries? (e.g., JDBC)	Spring MVC + Virtual Threads
Need to integrate fully reactive systems? (Kafka Streams, R2DBC, WebClient)?	Spring WebFlux
Want ultimate resource efficiency at massive scale? (100k+ concurrent users)	Spring WebFlux (still more efficient)
Simpler error handling, easier debugging needed?	Spring MVC + Virtual Threads

Practical Example: Side-by-side

Spring MVC + Virtual Threads

```
@RestController
public class UserController {
    @GetMapping("/user/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.getUser(id); // normal blocking call
    }
}
```

Spring WebFlux (Reactive)

```
@RestController
public class UserReactiveController {
    @GetMapping("/user/{id}")
    public Mono<User> getUser(@PathVariable Long id) {
        return userService.getUser(id); // reactive non-blocking call
    }
}
```

High Level Summary

Decision Point	Spring MVC + Virtual Threads	Spring WebFlux
Simpler development?		
Full non-blocking stack?		

| Maximum performance at crazy high scale (100k+)? | | |

| Standard blocking libraries (JDBC, etc)? | | |

| Steep learning curve? | | |

| Future-proofing for fully reactive systems? | | |

Final Mental Model

- If you just want to scale blocking systems easily MVC + Virtual Threads is enough.

- If you are building next-gen reactive pipelines end-to-end WebFlux is needed.

Real Pro Tip:

Virtual Threads are still bound by the underlying IO performance.

WebFlux gives you backpressure, streaming control, and fine-grained timeout/reactive retries.

Bonus

Today (2025), top backend architects are generally following:

| App Type | Framework |

|:---|:---|

| CRUD, APIs, monolithic apps | MVC + Virtual Threads |

| Microservices, event-driven, real-time systems | WebFlux |

My Recommendation

If your app is simple CRUD APIs, database centric, and your team prefers imperative code, Spring MVC + Virtual Threads is perfect.

If your app is very complex, heavy real-time, heavy concurrency, or you want ultimate resource efficiency WebFlux still wins.

Bonus Offer

Would you like me to draw:

- A flow diagram showing "Request handling in MVC + Virtual Thread" vs "WebFlux + Event Loop"?

- Or show "Code evolution from blocking to reactive" step-by-step?

Just say "Show me!" I'll make a full visual for you!

Do you want me to show full real production app design choices between WebFlux and MVC with Loom? (super valuable!)