# WebFlux vs Virtual Threads - Bonus Architect's Guide

Bonus: WebFlux vs Virtual Threads - Architect's Guide

1. Decision Tree: Should I use WebFlux or MVC + Virtual Threads?

- Is your system simple CRUD + database app?

  -> Use Spring MVC + Virtual Threads

- Do you need real-time streams, websockets, server-sent events?

  -> Use Spring WebFlux

- Are you targeting 50k - 500k concurrent clients?

  -> Use Spring WebFlux

- Is most of your code synchronous, blocking?

  -> MVC + Virtual Threads is better (no major rewrite)

- Are you building for reactive systems (R2DBC, Reactive Kafka, Reactive Redis)?

  -> Spring WebFlux

2. Benchmark Results (approximate under 2024 tests):

- Spring MVC (classic threads)

  - ~2000 concurrent users easily handled

    - High memory and thread usage

- Spring MVC + Virtual Threads
    - ~10000+ concurrent users
    - Low memory, better throughput
    - Latency stays stable

- Spring WebFlux (Reactive, Netty)
    - 100000+ concurrent users possible
    - Minimal memory usage
    - Needs non-blocking DB and external systems to stay fully reactive

3. Migration Strategy: MVC to WebFlux (if needed)

- Step 1: Identify blocking calls (DB, HTTP, File IO)
- Step 2: Migrate Repositories to R2DBC if DB used
- Step 3: Replace RestTemplate with WebClient
- Step 4: Use RouterFunctions or Annotated Controllers
- Step 5: Gradually introduce Mono/Flux return types
- Step 6: Configure Netty server properly
- Step 7: Load test both versions before production

Reminder: MVC + Virtual Threads gives 80% of benefits with almost 0 refactor cost!

Conclusion:

- For new reactive systems -> WebFlux

- For existing apps / fast delivery -> MVC + Virtual Threads

Choose wisely based on TEAM skillset + SYSTEM type!