

Server-side Kotlin with Coroutines

Roman Elizarov

 relizarov

Speaker: Roman Elizarov



- Professional developer since 2000
- Previously developed high-perf trading software
@ Devexperts
- Teach concurrent & distributed programming
@ St. Petersburg ITMO University
- Chief judge
@ Northern Eurasia Contest / ICPC
- Now team lead in Kotlin Libraries
@ JetBrains



Kotlin – Programming Language for



JVM



Android



Browser



Native

Kotlin – Programming Language for



Android



Browser



Native

Server-side

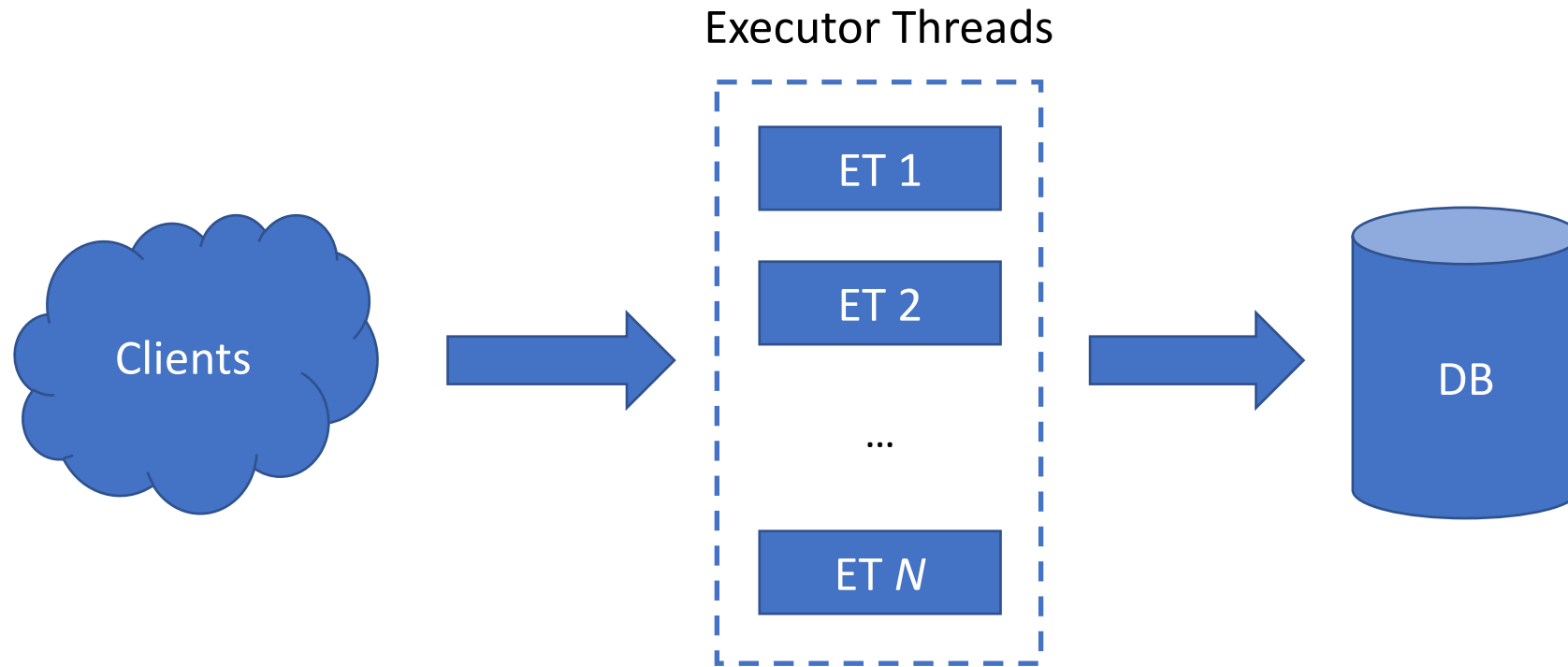


This talk

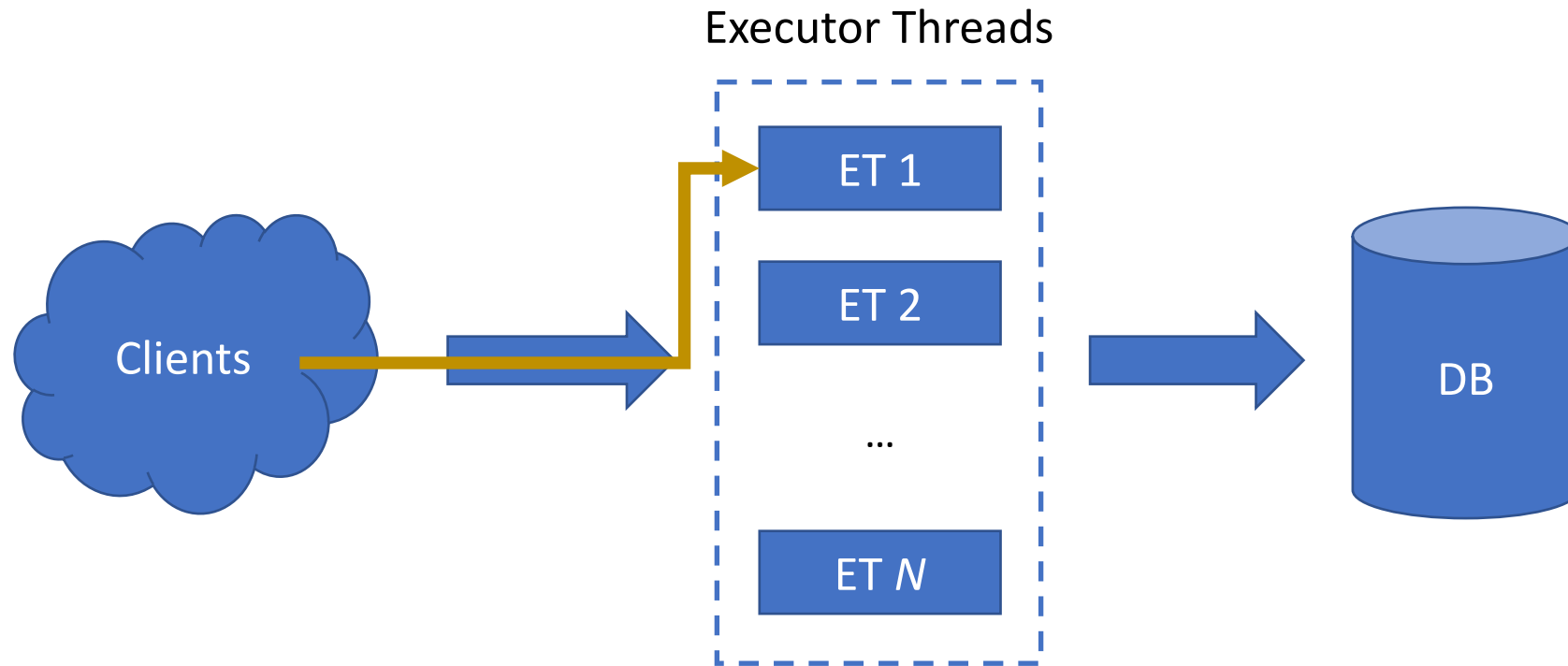
Backend evolution

Starting with “good old days”

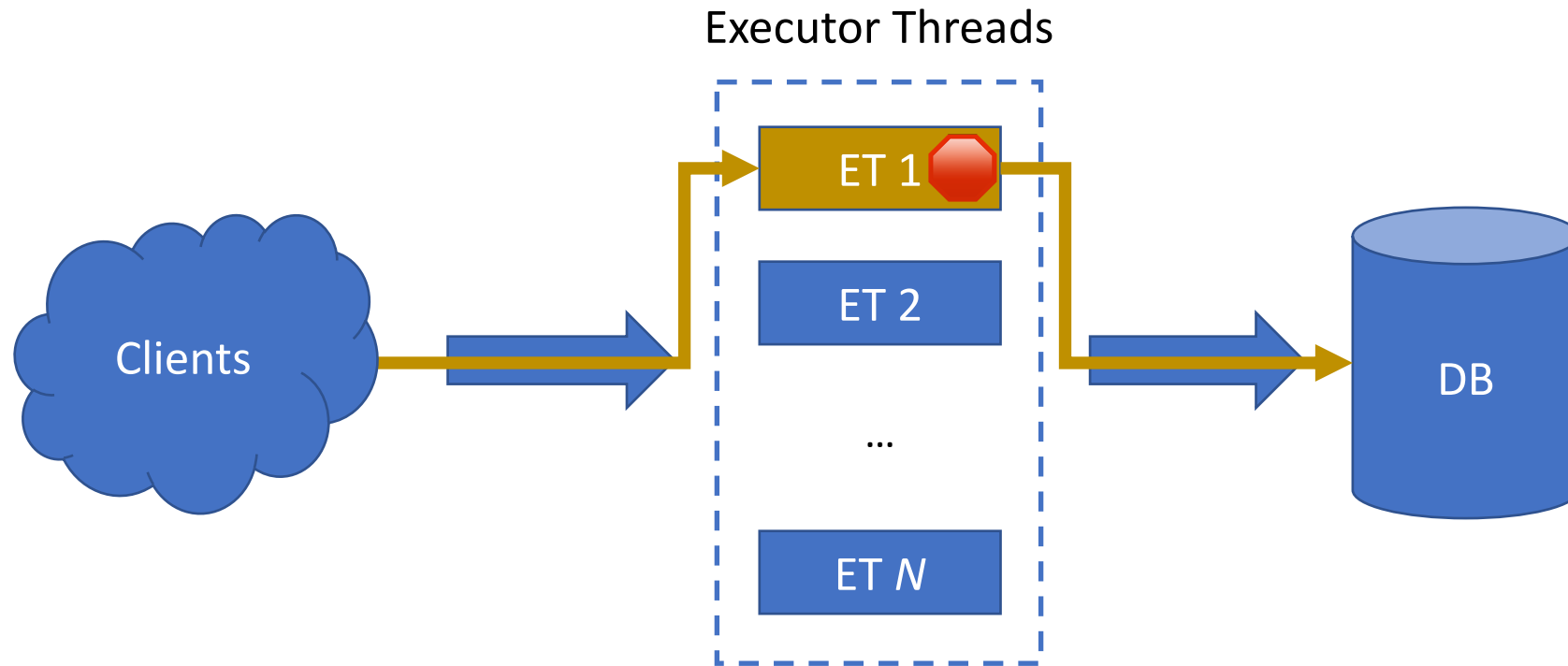
Old-school client-server monolith



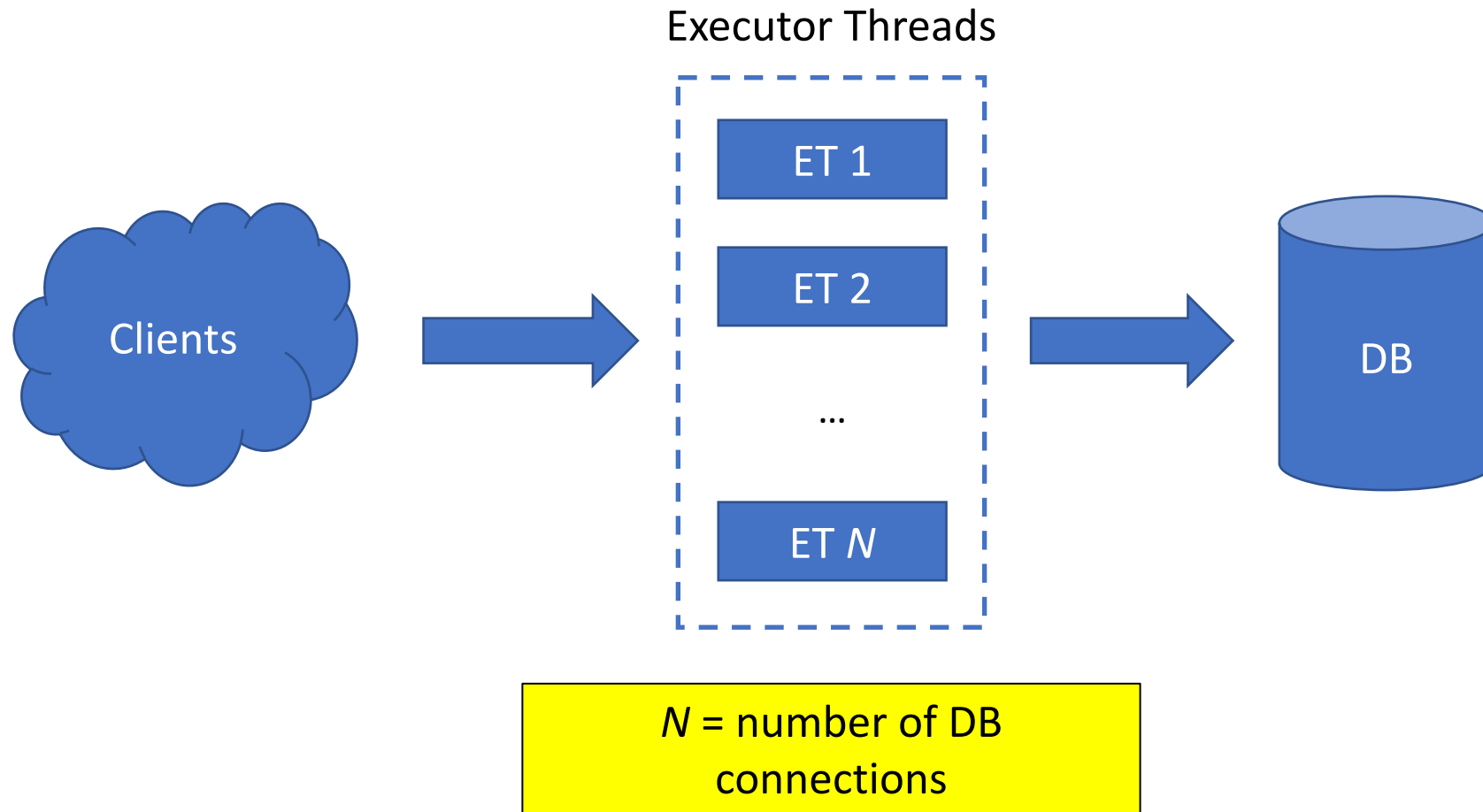
Incoming request



Blocks tread

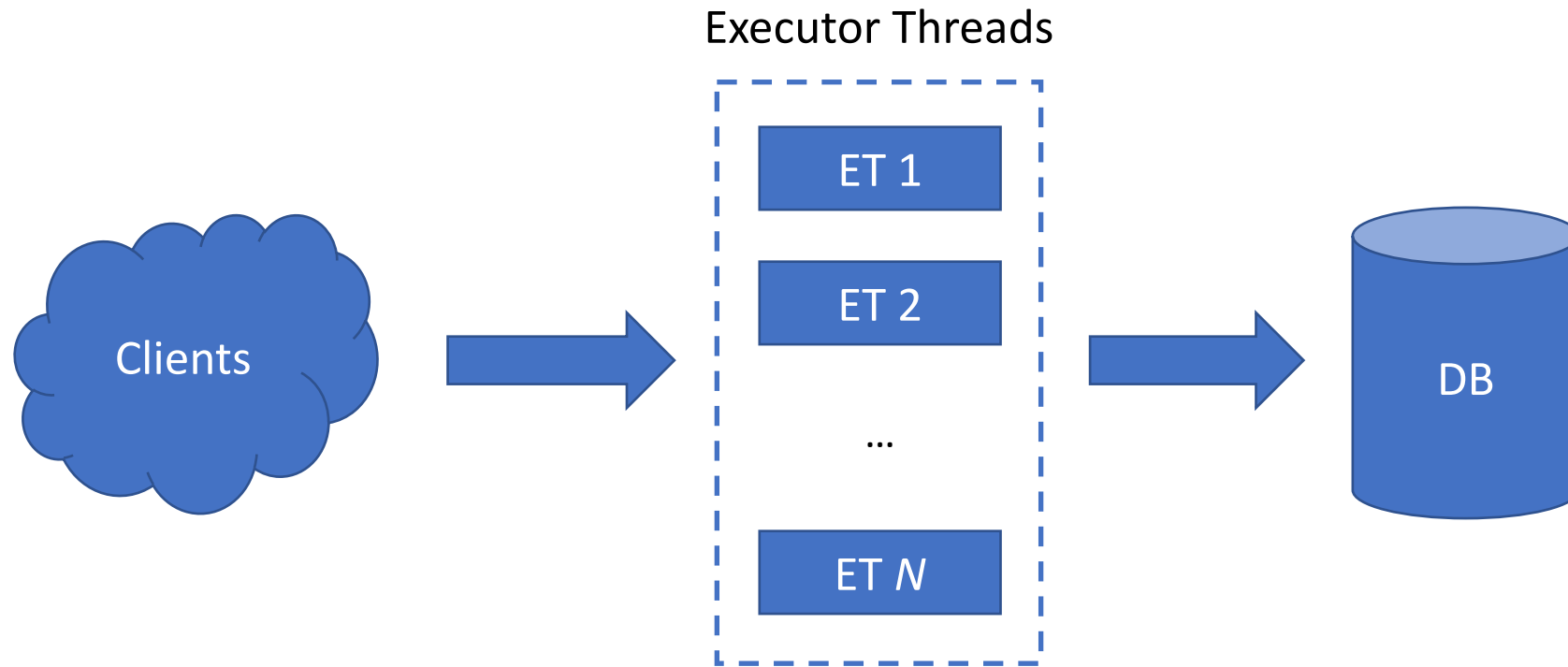


Sizing threads – easy

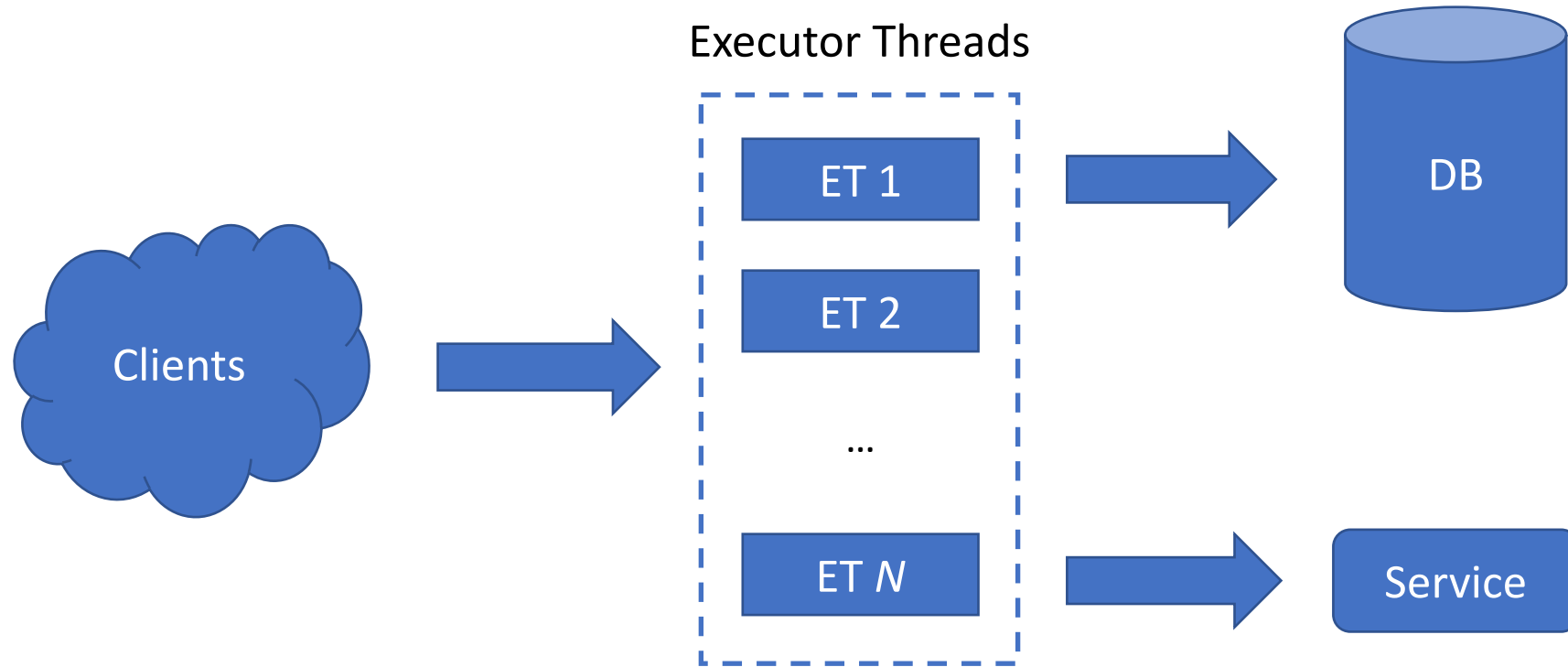


Services

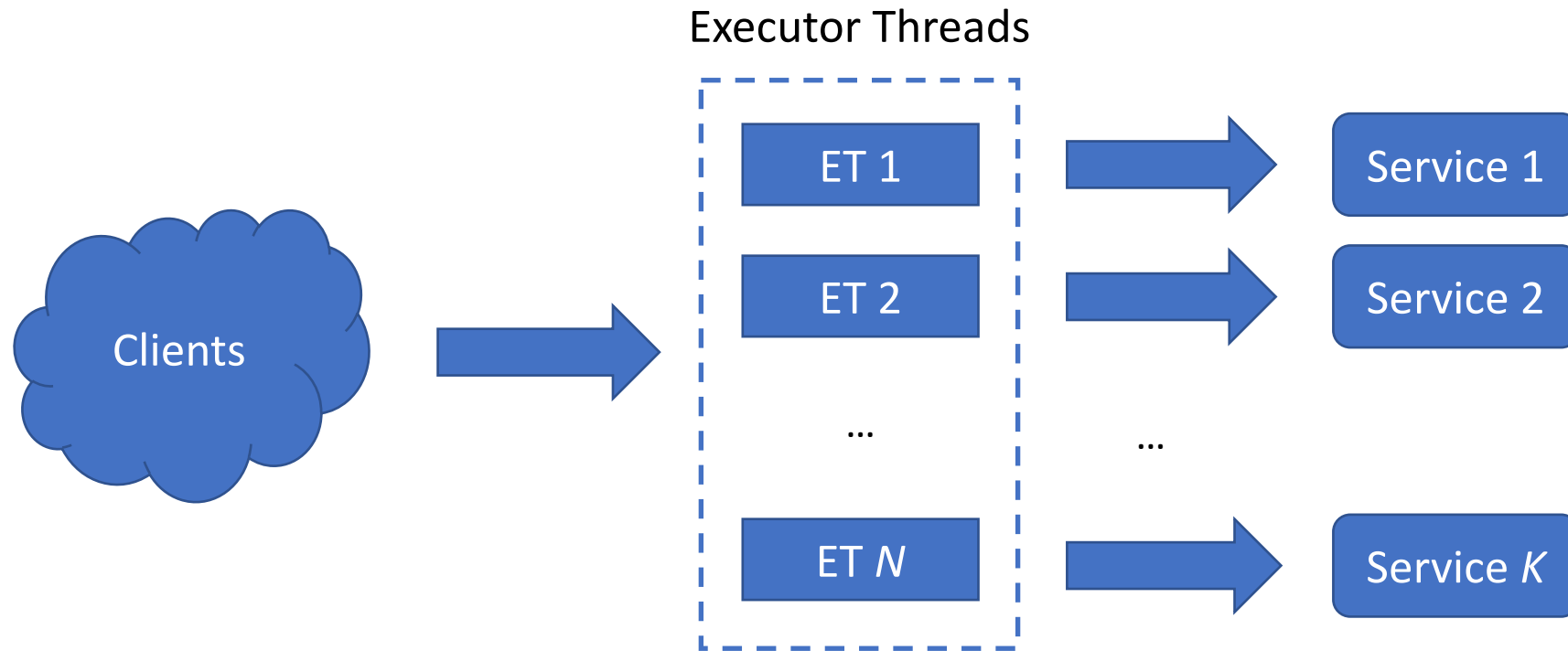
Old-school client-server monolith



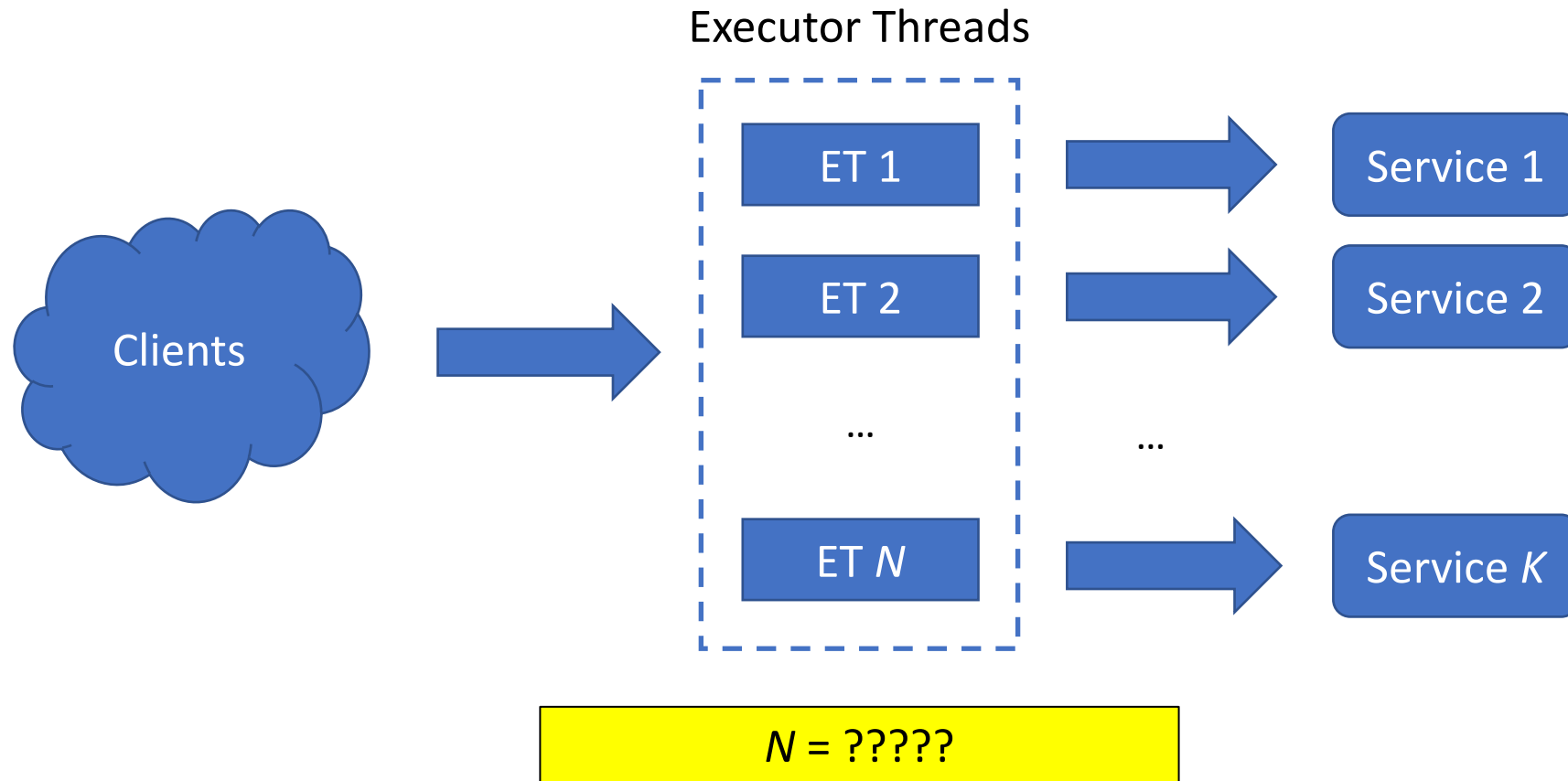
Now with Services



Services everywhere



Sizing threads – not easy



Complex business logic

```
fun placeOrder(order: Order): Response {  
    ...  
}
```

Complex business logic

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    ...  
}
```

Complex business logic

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount)  
        ...  
}
```

Complex business logic

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account)  
    } else {  
        defaultMargin  
    }  
    ...  
}
```

Complex business logic

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account)  
    } else {  
        defaultMargin  
    }  
    return validateOrder(order, margin)  
}
```

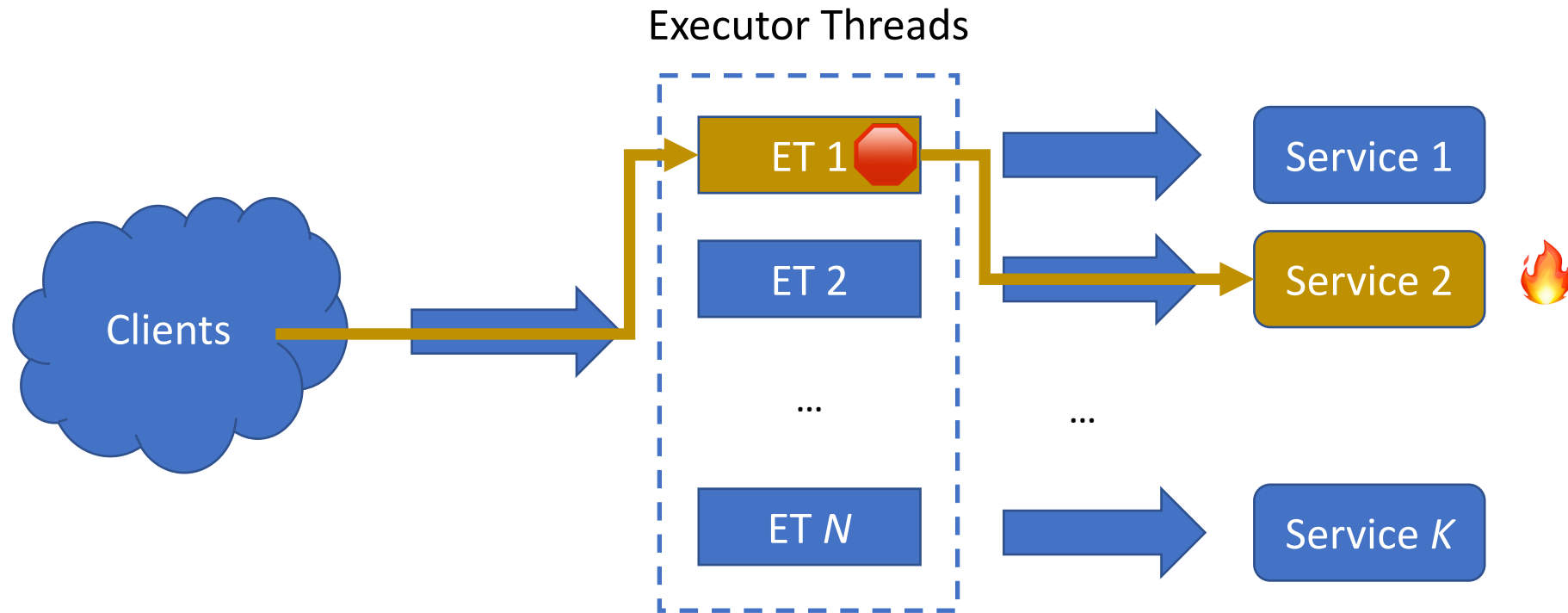
Complex business logic

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account)  
    } else {  
        defaultMargin  
    }  
    return validateOrder(order, margin)  
}
```

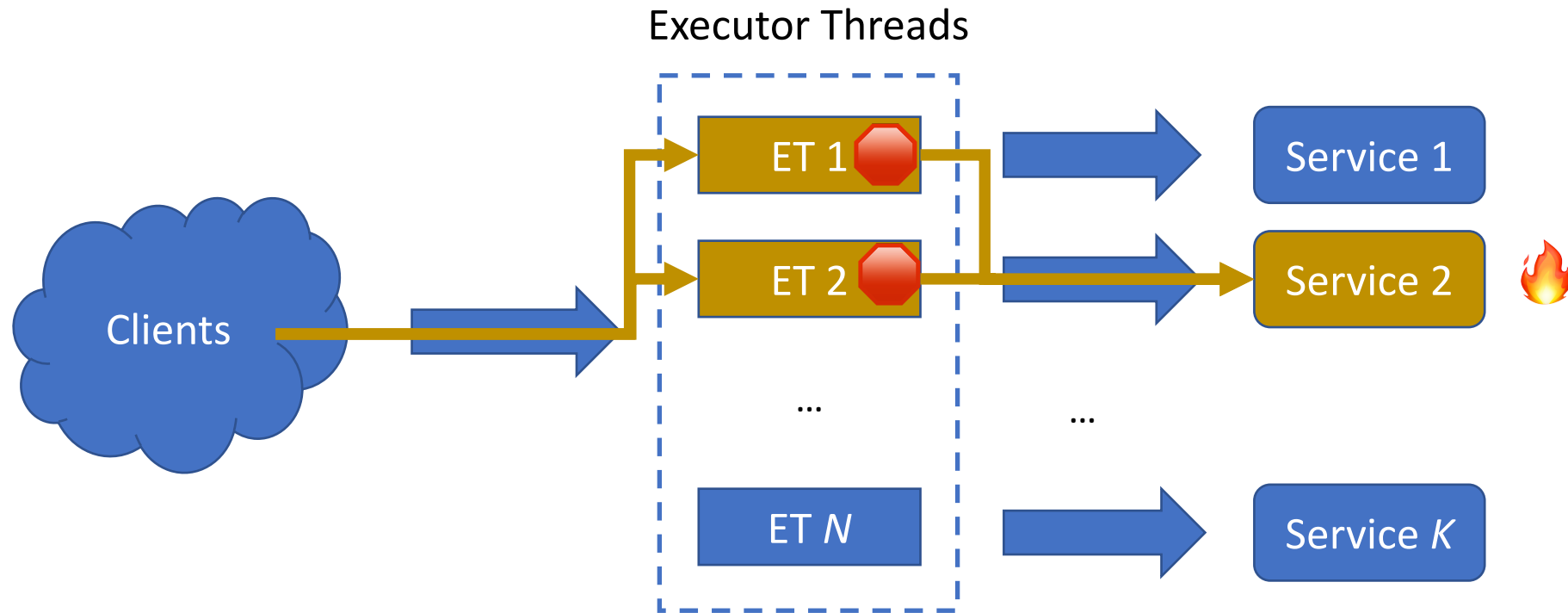
What if a service is slow?

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account) _____ 🔥  
    } else {  
        defaultMargin  
    }  
    return validateOrder(order, margin)  
}
```

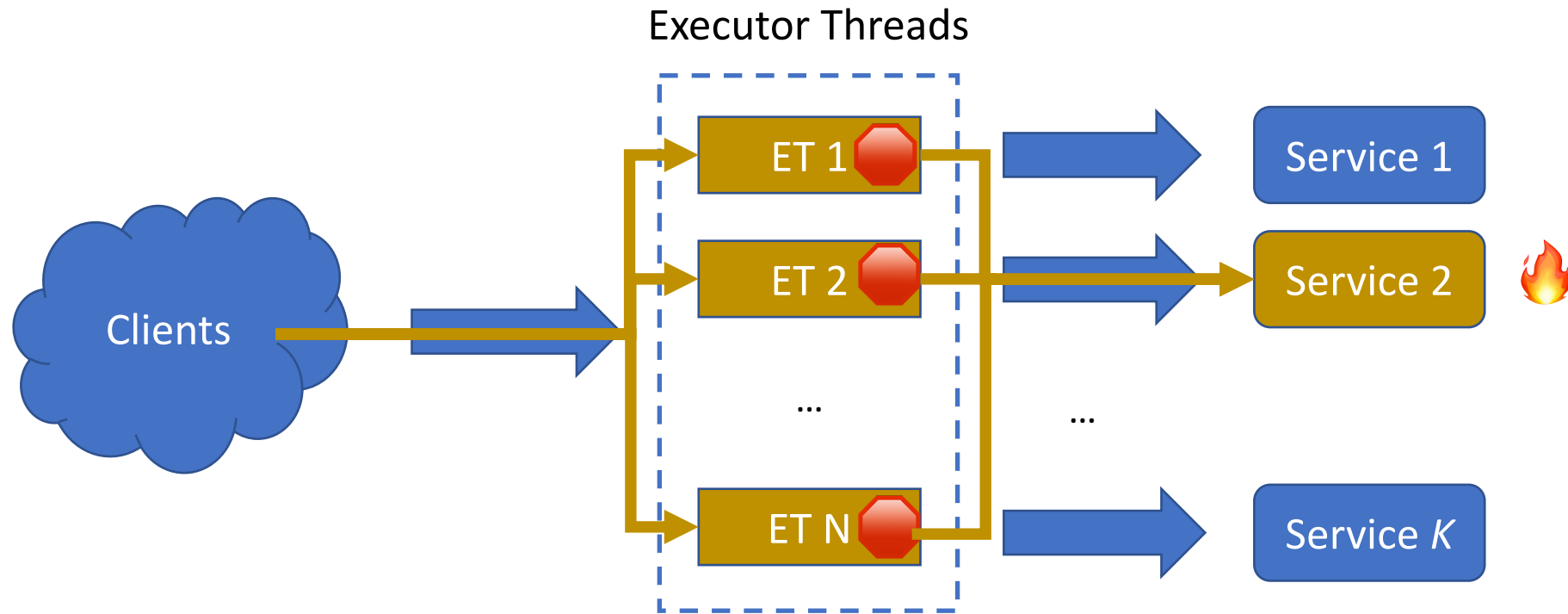
Blocks threads



Blocks threads



Blocks threads



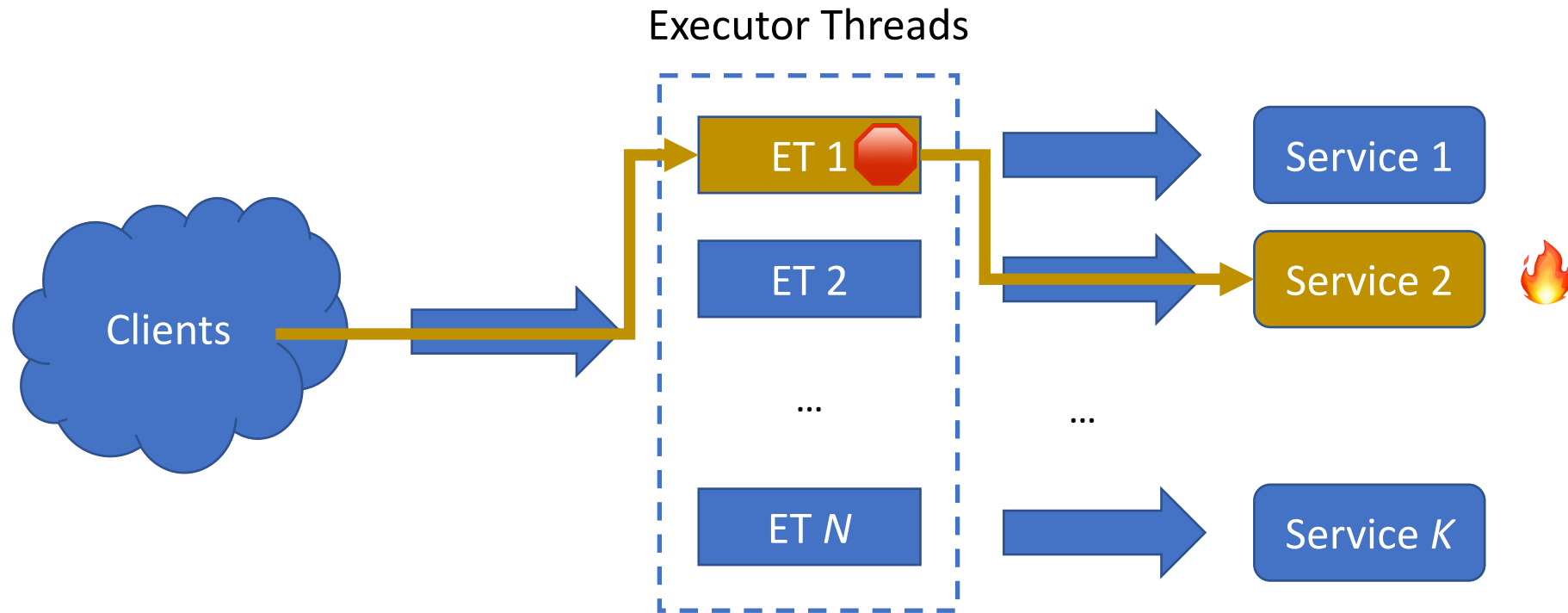


Code that **waits**

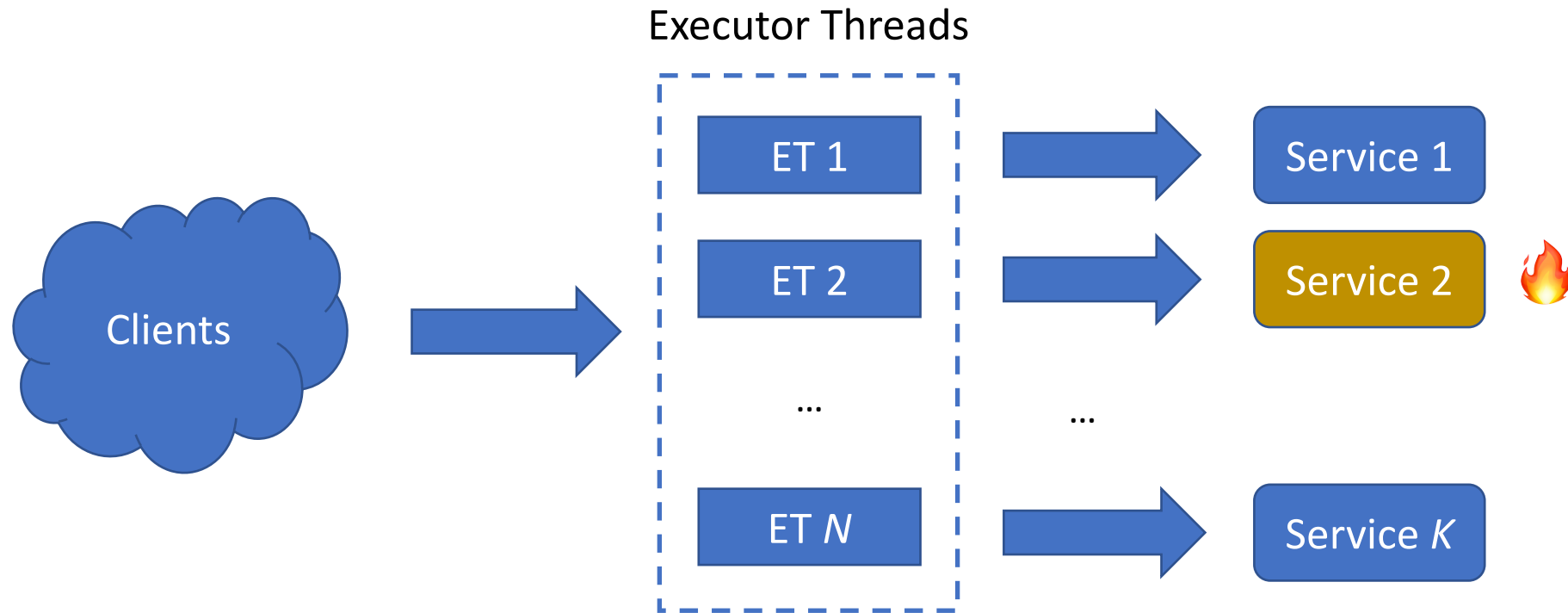
Asynchronous programming

Writing code that waits

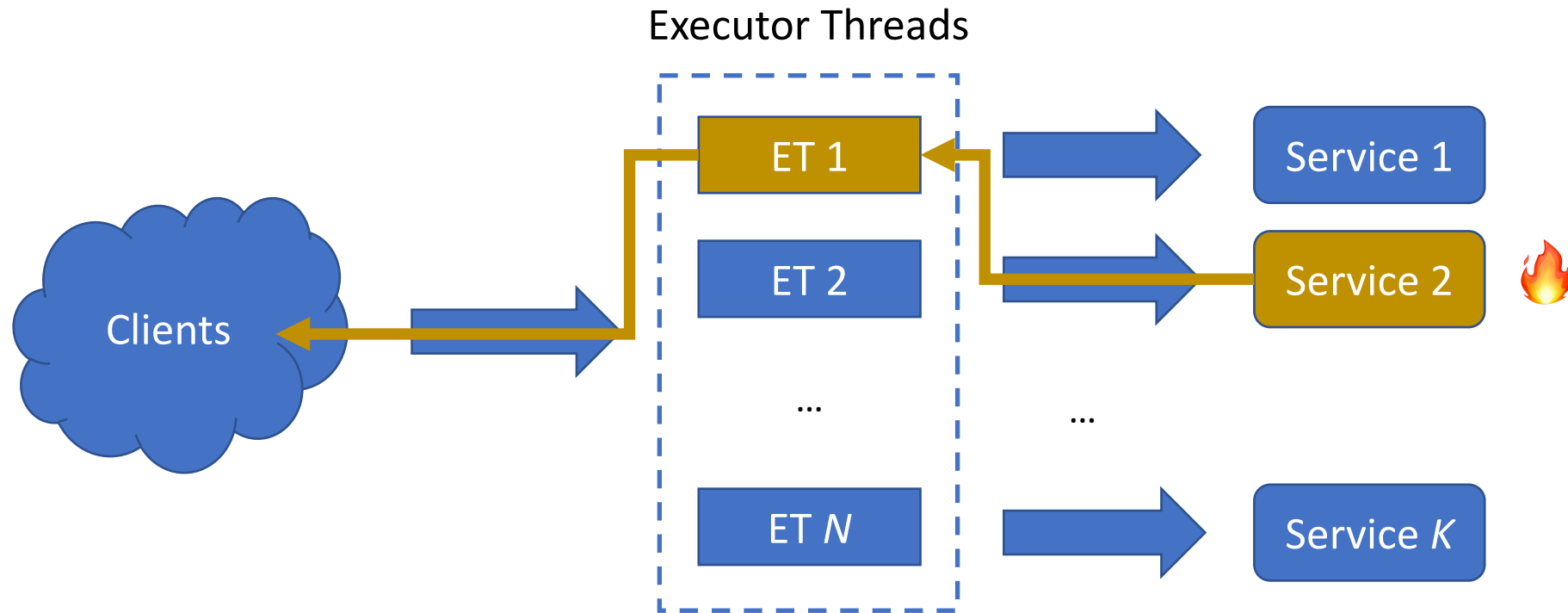
Instead of blocking...



Release the thread



Resume operation later



But how?

```
fun loadMargin(account: Account): Margin
```

But how?

```
fun loadMargin(account: Account, callback: (Margin) -> Unit)
```

- Callbacks

But how?

```
fun loadMargin(account: Account): Future<Margin>
```

- Callbacks
- Futures/Promises

But how?

```
fun loadMargin(account: Account): Mono<Margin>
```

- Callbacks

- ~~Futures/Promises/Reactive~~

But how?

```
async fun loadMargin(account: Account): Task<Margin>
```

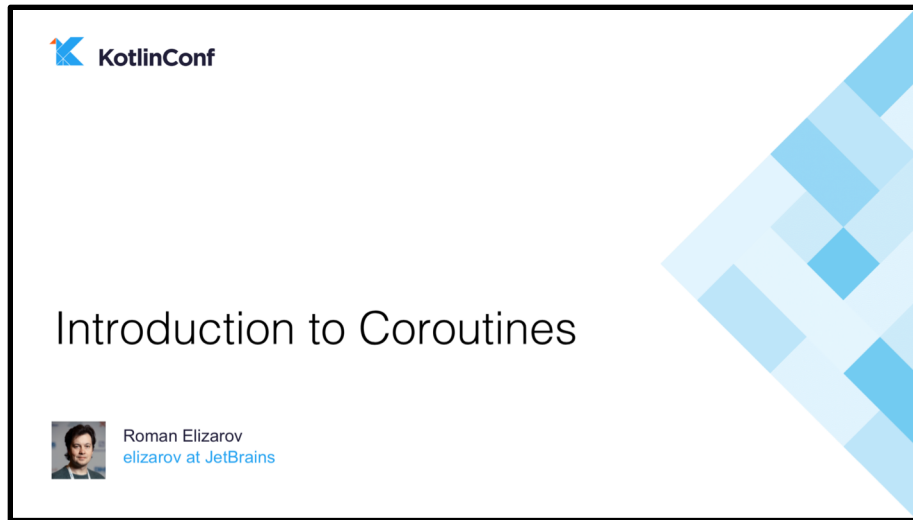
- Callbacks
- Futures/Promises/Reactive
- **async/await**

But how?

```
suspend fun loadMargin(account: Account): Margin
```

- Callbacks
- Futures/Promises/Reactive
- async/await
- **Kotlin Coroutines**

Learn more



KotlinConf (San Francisco) 2017



GOTO Copenhagen 2018

Suspend behind the scenes

```
suspend fun loadMargin(account: Account): Margin
```

Suspend behind the scenes

```
suspend fun loadMargin(account: Account): Margin
```

```
fun loadMargin(account: Account, cont: Continuation<Margin>)
```

But why callback and not future?

Performance!

- **Future** is a *synchronization* primitive
- **Callback** is a *lower-level* primitive
- **Integration** with async IO libraries is easy

Integration

```
suspend fun loadMargin(account: Account): Margin
```

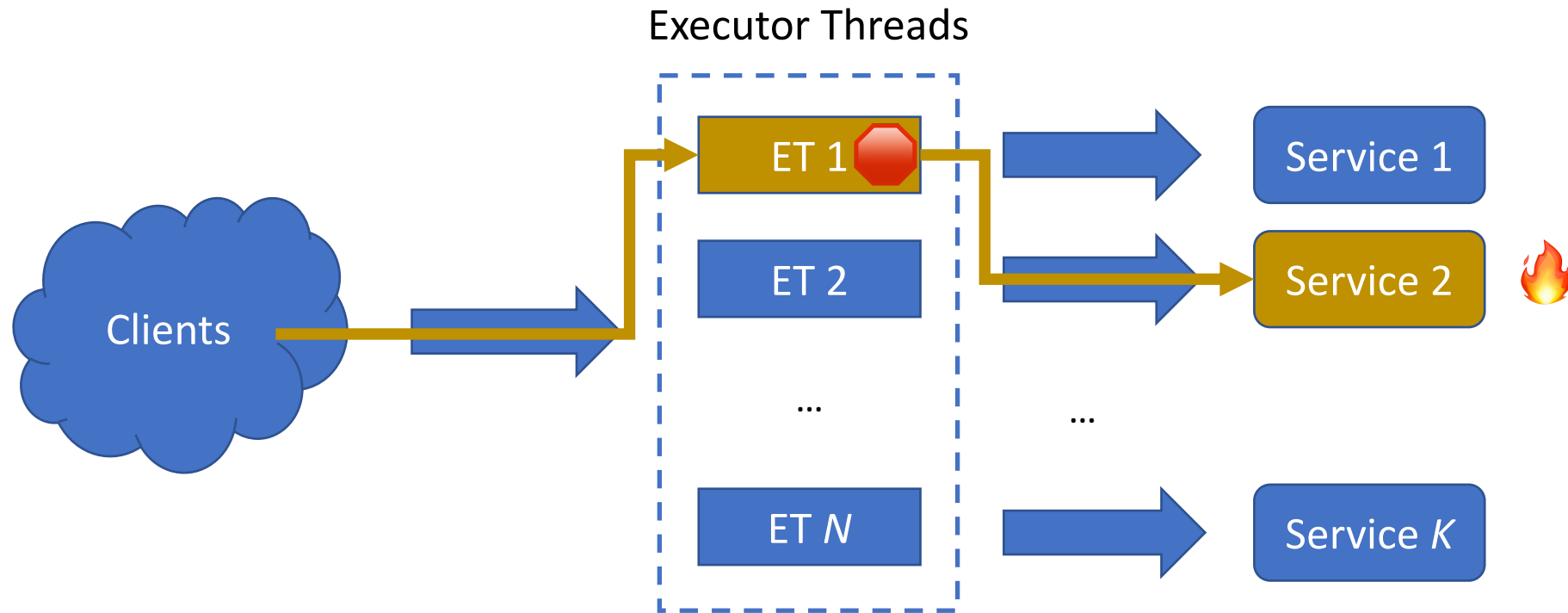
Integration

```
suspend fun loadMargin(account: Account): Margin =  
    suspendCoroutine { cont ->  
        // install callback & use cont to resume  
    }
```

Integration at scale

Going beyond slide-ware

Release thread?



Blocking server

```
fun placeOrder(order: Order): Response {  
    // must return response  
}
```

Asynchronous server

```
fun placeOrder(order: Order): Mono<Response> {  
    // may return without response  
}
```

Convenient?

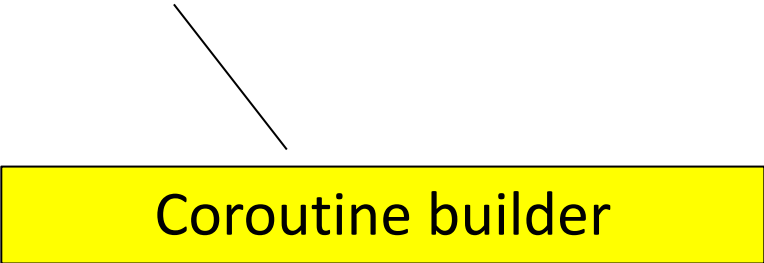
```
fun placeOrder(order: Order): Mono<Response> {  
    // response from placed order cache  
    return Mono.just(response)  
}
```

Server integrated with coroutines

```
suspend fun placeOrder(order: Order): Response {  
    // response from placed order cache  
    return response  
}
```

Server not integrated with coroutines

```
fun placeOrder(order: Order) = GlobalScope.mono {  
    // response from placed order cache  
    return@mono response  
}
```



Coroutine builder

The server shall support
asynchrony *is some way*

Suspend

```
suspend fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account)  
    } else {  
        defaultMargin  
    }  
    return validateOrder(order, margin)  
}
```

Suspend

```
suspend fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account)  
    } else {  
        defaultMargin  
    }  
    return validateOrder(order, margin)  
}
```

Invoke suspending funs

Suspend is convenient

```
suspend fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order.accountId)  
    val margin = if (account.isOptionsAccount) {  
        marginService.loadMargin(account)  
    } else {  
        defaultMargin  
    }  
    return validateOrder(order, margin)  
}
```

Invoke suspending funs

Write regular code!

Suspend is efficient

One object allocated

```
suspend fun placeOrder(order: Order): Response {  
->    val account = accountService.loadAccount(order.accountId)  
->    val margin = marginService.loadMargin(account)  
    return validateOrder(order, margin)  
}
```

Futures/Promises/Reactive – less efficient

```
fun placeOrder(order: Order): Mono<Response> =  
    accountService.loadAccountAsync(order.accountId)  
        .flatMap { account -> marginService.loadMargin(account) }  
        .map { margin -> validateOrder(order, margin) }
```

Future allocated

Future allocated

Lambda allocated

Lambda allocated*

Let's go deeper

```
fun placeOrder(params: Params): Mono<Response> {  
    // check pre-conditions  
    return actuallyPlaceOrder(order)  
}
```

```
fun actuallyPlaceOrder(order: Order): Mono<Response>
```

Let's go deeper (with coroutines)

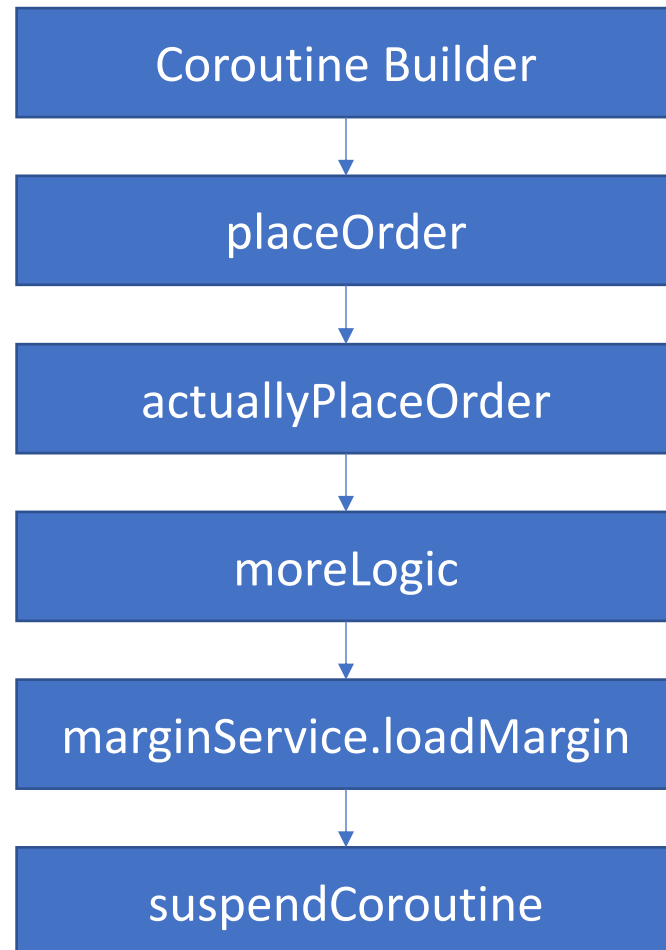
```
suspend fun placeOrder(params: Params): Response {  
    // check pre-conditions  
    return actuallyPlaceOrder(order)  
}
```

Tail call

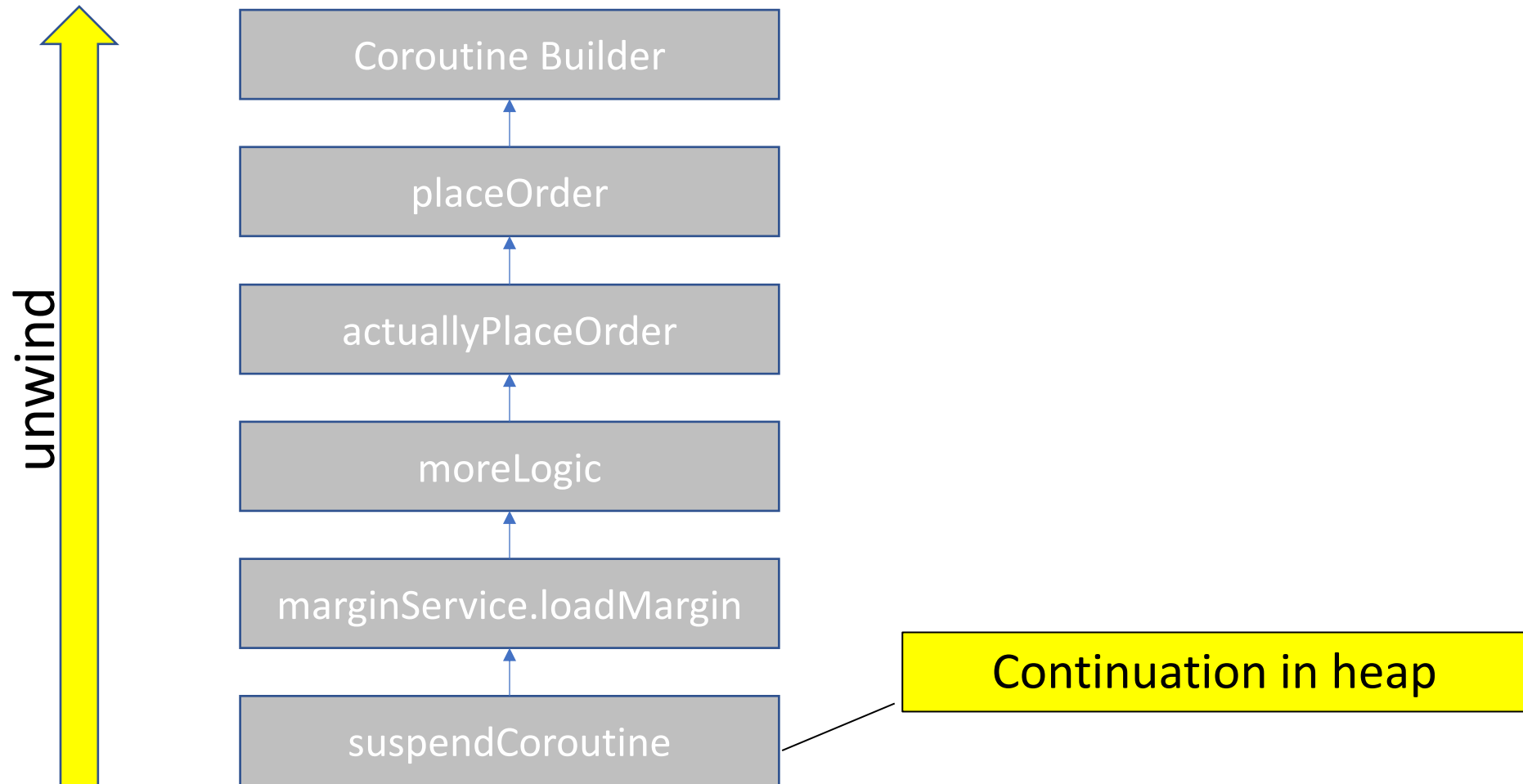
```
suspend fun actuallyPlaceOrder(params: Params): Response
```

Tail call optimization

Call stack with coroutines



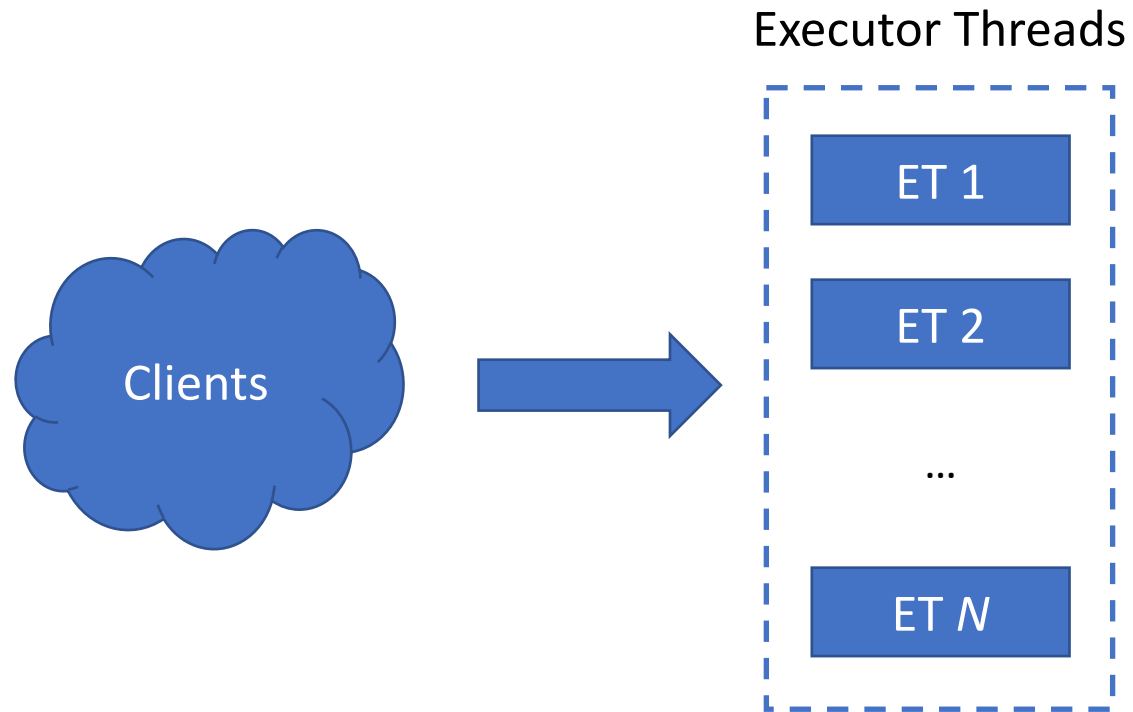
Call stack with coroutines



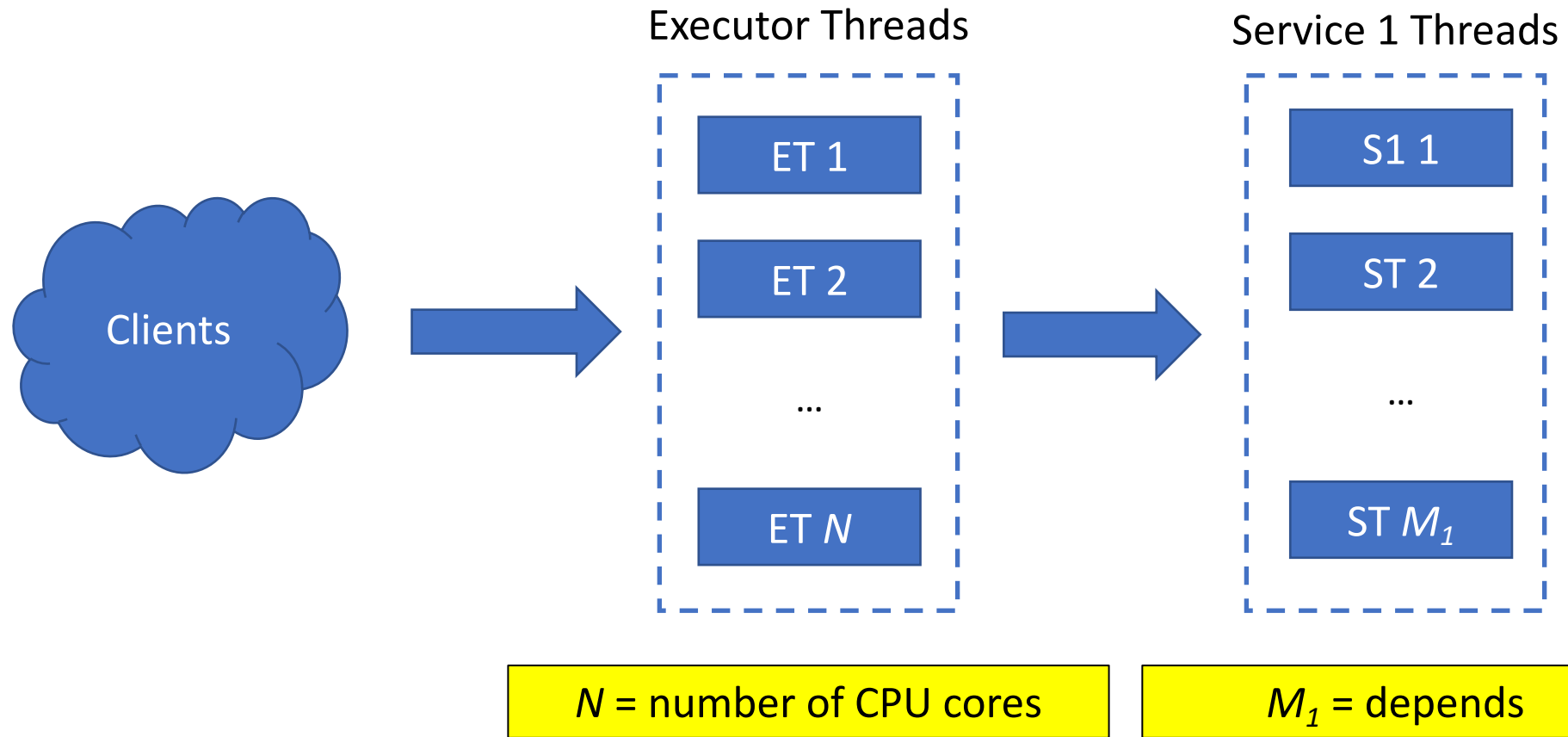
Scaling with coroutines

With thread pools

Thread pools



Thread pools



IO-bound (blocking)

```
fun loadAccount(order: Order): Account {  
    // some blocking code here....  
}
```

IO-bound

```
suspend fun loadAccount(order: Order): Account {  
    // some blocking code here....  
}
```

IO-bound withContext

```
suspend fun loadAccount(order: Order): Account =  
    withContext(dispatcher) {  
        // some blocking code here....  
    }
```

IO-bound withContext

```
suspend fun loadAccount(order: Order): Account =  
    withContext(dispatcher) {  
        // some blocking code here....  
    }
```

```
val dispatcher =  
    Executors.newFixedThreadPool(M2).asCoroutineDispatcher()
```

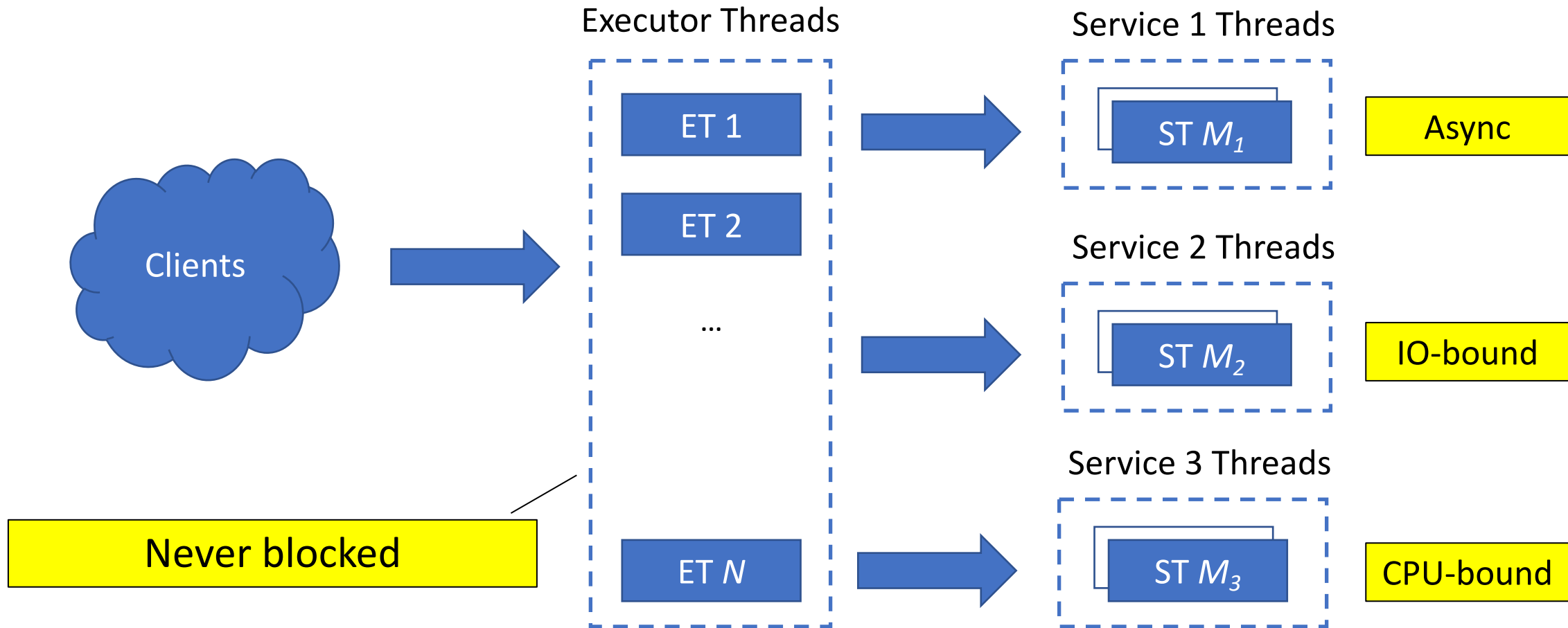
CPU-bound code

```
fun validateOrder(order: Order, margin: Margin): Response {  
    // perform CPU-consuming computation  
}
```

CPU-bound code

```
suspend fun validateOrder(order: Order, margin: Margin): Response =  
    withContext(compute) {  
        // perform CPU-consuming computation  
    }  
  
val compute =  
    Executors.newFixedThreadPool(M3).asCoroutineDispatcher()
```

Fine-grained control and encapsulation



But there's more!

Cancellation

withTimeout

```
suspend fun placeOrder(order: Order): Response =  
    withTimeout(1000) {  
        // code before  
        loadMargin(account)  
        // code after  
    }
```

withTimeout propagation

```
suspend fun placeOrder(order: Order): Response =  
    withTimeout(1000) {  
        // code before  
        loadMargin(account)  
        // code after  
    }
```

```
suspend fun loadMargin(account: Account): Margin =  
    suspendCoroutine { cont ->  
        // install callback & use cont to resume  
    }
```

withTimeout propagation

```
suspend fun placeOrder(order: Order): Response =  
    withTimeout(1000) {  
        // code before  
        loadMargin(account)  
        // code after  
    }
```

```
suspend fun loadMargin(account: Account): Margin =  
    suspendCancellableCoroutine { cont ->  
        // install callback & use cont to resume  
    }
```

withTimeout propagation

```
suspend fun placeOrder(order: Order): Response =  
    withTimeout(1000) {  
        // code before  
        loadMargin(account)  
        // code after  
    }
```

```
suspend fun loadMargin(account: Account): Margin =  
    suspendCancellableCoroutine { cont ->  
        // install callback & use cont to resume  
        cont.invokeOnCancellation { ... }  
    }
```

Concurrency

Multiple things at the same time

Example

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order)  
    val margin = marginService.loadMargin(order)  
    return validateOrder(order, account, margin)  
}
```

Example

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccount(order)  
    val margin = marginService.loadMargin(order)  
    return validateOrder(order, account, margin)  
}
```

No data dependencies

Concurrency with async (futures)

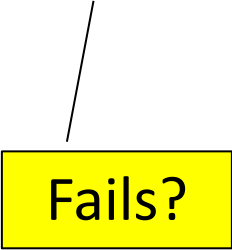
```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccountAsync(order)  
    val margin = marginService.loadMarginAsync(order)  
    return validateOrder(order, account.await(), margin.await())  
}
```

Concurrency with async (futures)

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccountAsync(order)  
    val margin = marginService.loadMarginAsync(order)  
    return validateOrder(order, account.await(), margin.await())  
}
```

Concurrency with async (futures)

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccountAsync(order)  
    val margin = marginService.loadMarginAsync(order)  
    return validateOrder(order, account.await(), margin.await())  
}
```



Fails?

Concurrency with async (futures)

```
fun placeOrder(order: Order): Response {  
    val account = accountService.loadAccountAsync(order)  
    val margin = marginService.loadMarginAsync(order)  
    return validateOrder(order, account.await(), margin.await())  
}
```

Fails?

Leaks!

Structured concurrency

Concurrency with coroutines

```
suspend fun placeOrder(order: Order): Response =  
    coroutineScope {  
        val account = async { accountService.loadAccount(order) }  
        val margin = async { marginService.loadMargin(order) }  
        validateOrder(order, account.await(), margin.await())  
    }
```

Concurrency with coroutines

```
suspend fun placeOrder(order: Order): Response =  
    coroutineScope {  
        val account = async { accountService.loadAccount(order) }  
        val margin = async { marginService.loadMargin(order) }  
        validateOrder(order, account.await(), margin.await())  
    }
```

Concurrency with coroutines

```
suspend fun placeOrder(order: Order): Response =  
    coroutineScope {  
        val account = async { accountService.loadAccount(order) }  
        val margin = async { marginService.loadMargin(order) }  
        validateOrder(order, account.await(), margin.await())  
    }
```

Concurrency with coroutines

```
suspend fun placeOrder(order: Order): Response =  
    coroutineScope {  
        val account = async { accountService.loadAccount(order) }  
        val margin = async { marginService.loadMargin(order) }  
        validateOrder(order, account.await(), margin.await())  
    }
```

Fails?

Concurrency with coroutines

```
suspend fun placeOrder(order: Order): Response =
```

```
    coroutineScope {
```

```
        val account = async { accountService.loadAccount(order) }
```

```
        val margin = async { marginService.loadMargin(order) }
```

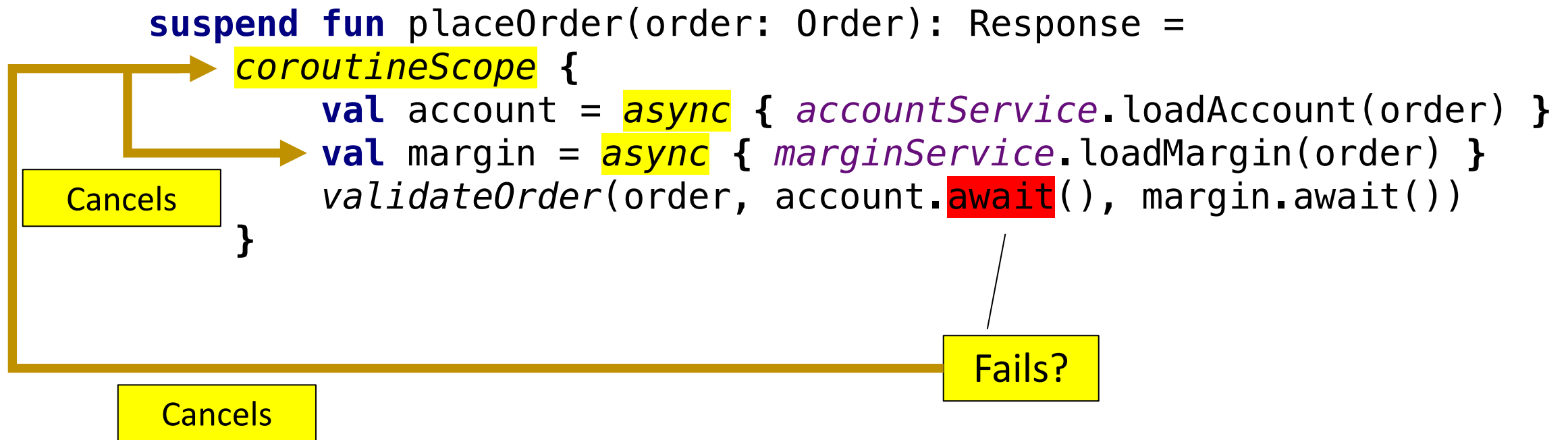
```
        validateOrder(order, account.await(), margin.await())
```

```
    }
```

Cancels

Fails?

Concurrency with coroutines



Concurrency with coroutines

```
suspend fun placeOrder(order: Order): Response =  
    coroutineScope {  
        val account = async { accountService.loadAccount(order) }  
        val margin = async { marginService.loadMargin(order) }  
        validateOrder(order, account.await(), margin.await())  
    }
```

Waits for completion of all children

Enforcing structure

Without coroutine scope?

```
suspend fun placeOrder(order: Order): Response {  
    val account = async { accountService.loadAccount(order) }  
    val margin = async { marginService.loadMargin(order) }  
    return validateOrder(order, account.await(), margin.await())  
}
```

Without coroutine scope?

```
suspend fun placeOrder(order: Order): Response {  
    val account = async { accountService.loadAccount(order) }  
    val margin = async { marginService.loadMargin(order) }  
    return validateOrder(order, account.await(), margin.await())  
}
```

ERROR: Unresolved reference.

Extensions of CoroutineScope

```
fun <T> CoroutineScope.async(  
    context: CoroutineContext = EmptyCoroutineContext,  
    start: CoroutineStart = CoroutineStart.DEFAULT,  
    block: suspend CoroutineScope.() -> T  
) : Deferred<T>
```

Convention

Launches new coroutine

```
fun CoroutineScope.bg(params: Params) = launch {  
    // ...  
}
```

Types as documentation

fun foo(params: Params): Response

Fast, local

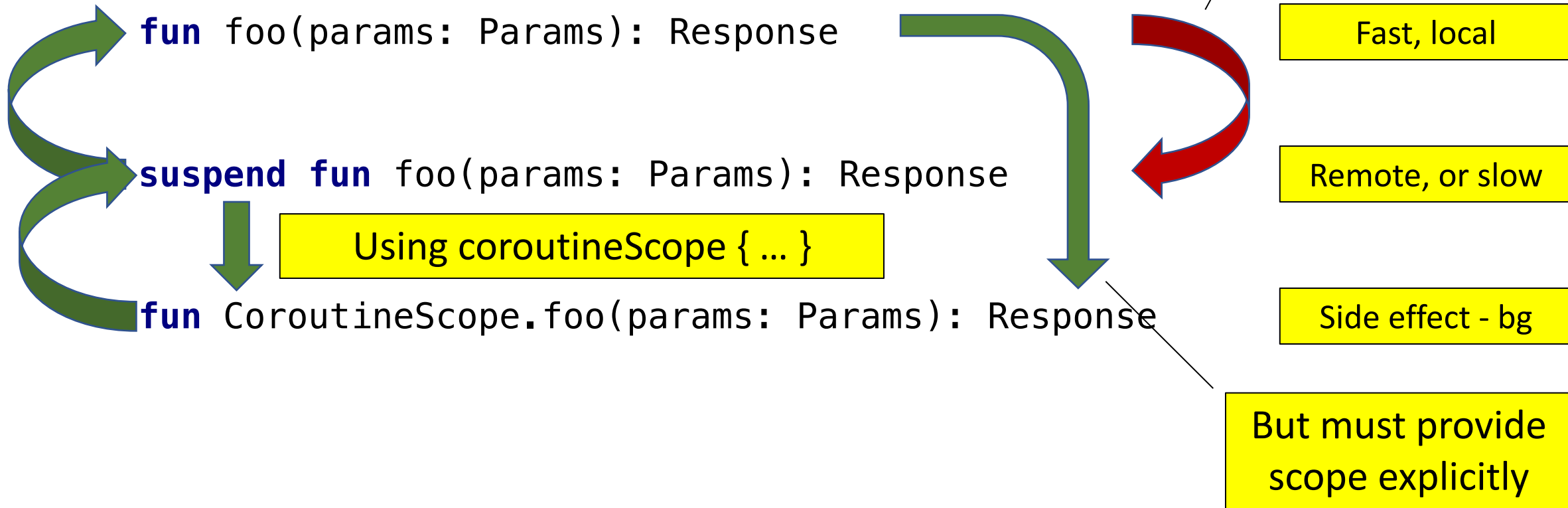
suspend fun foo(params: Params): Response

Remote, or slow

fun CoroutineScope.foo(params: Params): Response

Side effect - bg

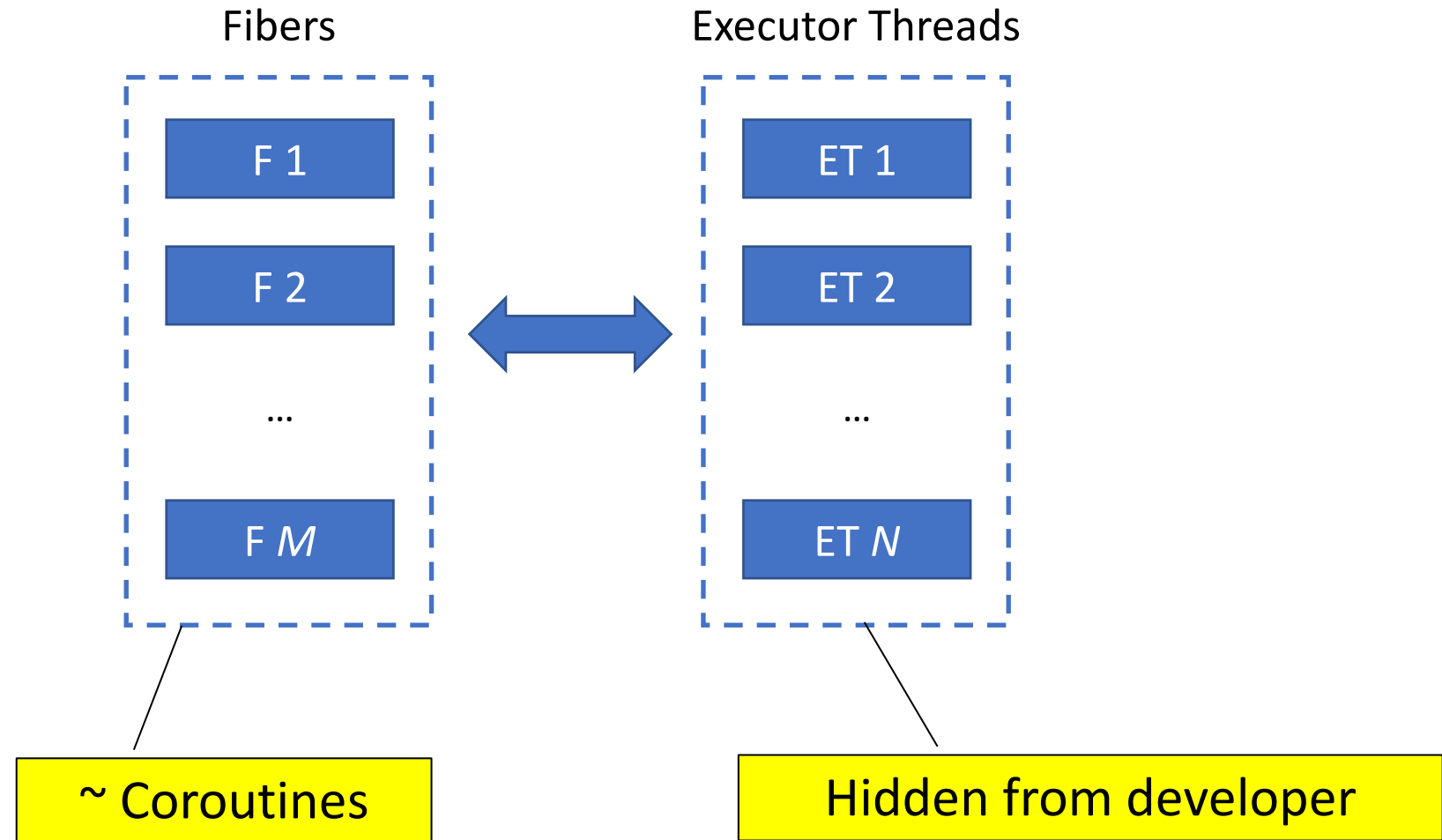
Types are enforced



Green threads / fibers

Alternative way to async

Green threads / Fibers



Fibers promise

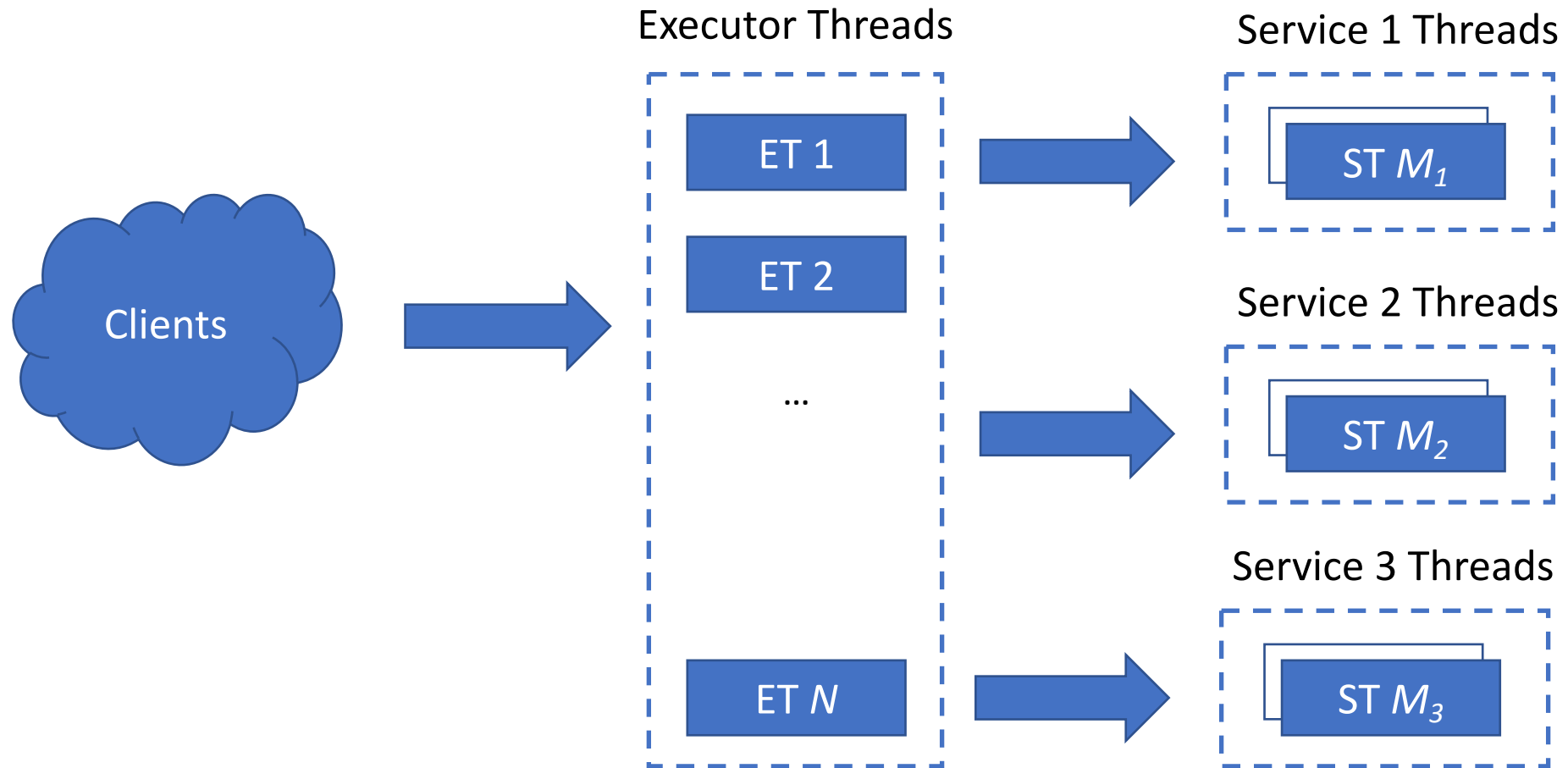
- Develop just like with threads
- Everything is effectively suspendable

Marking with **suspend**
pays off at scale

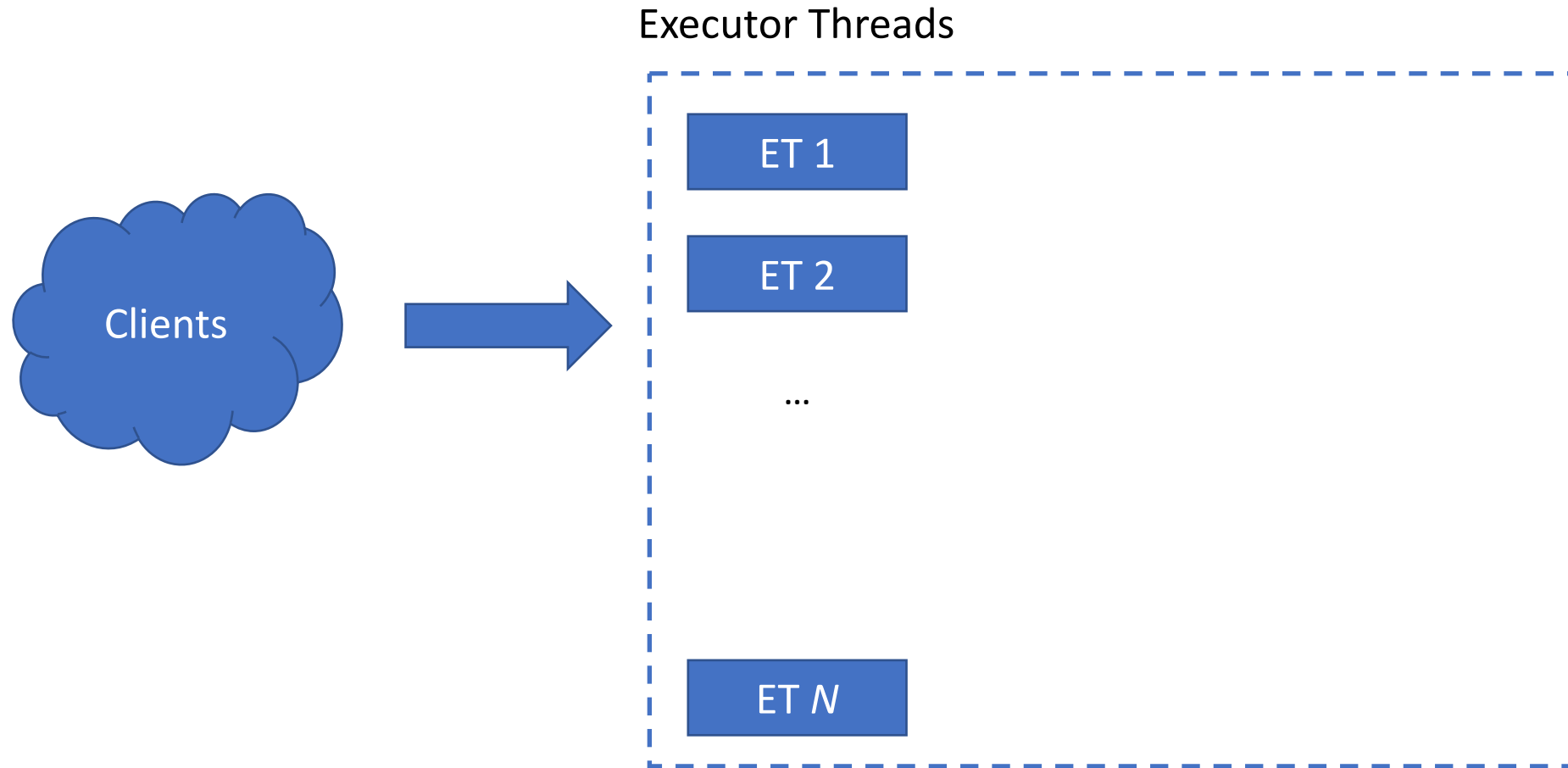
Thread switching

And how to avoid it

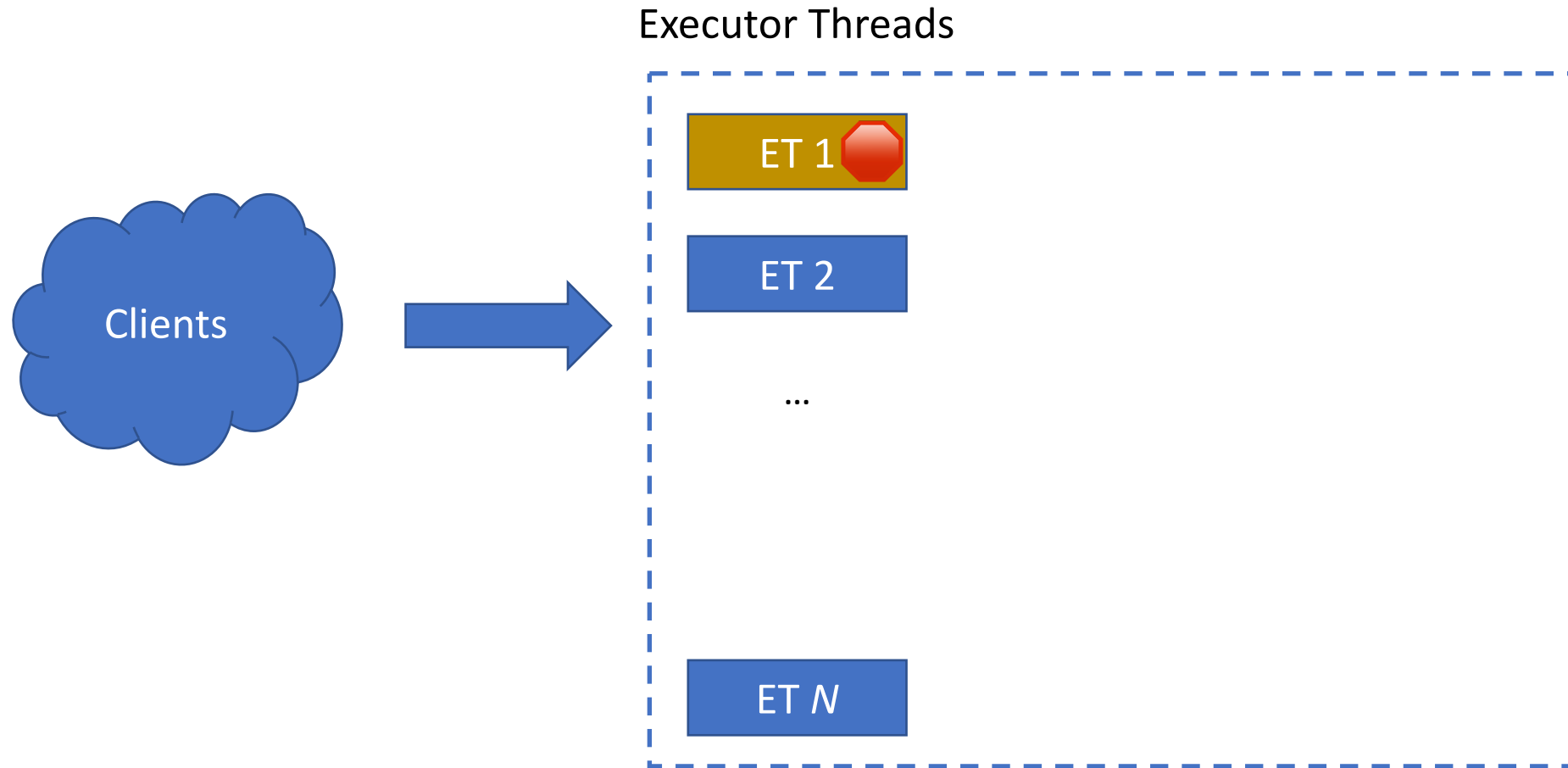
Threads



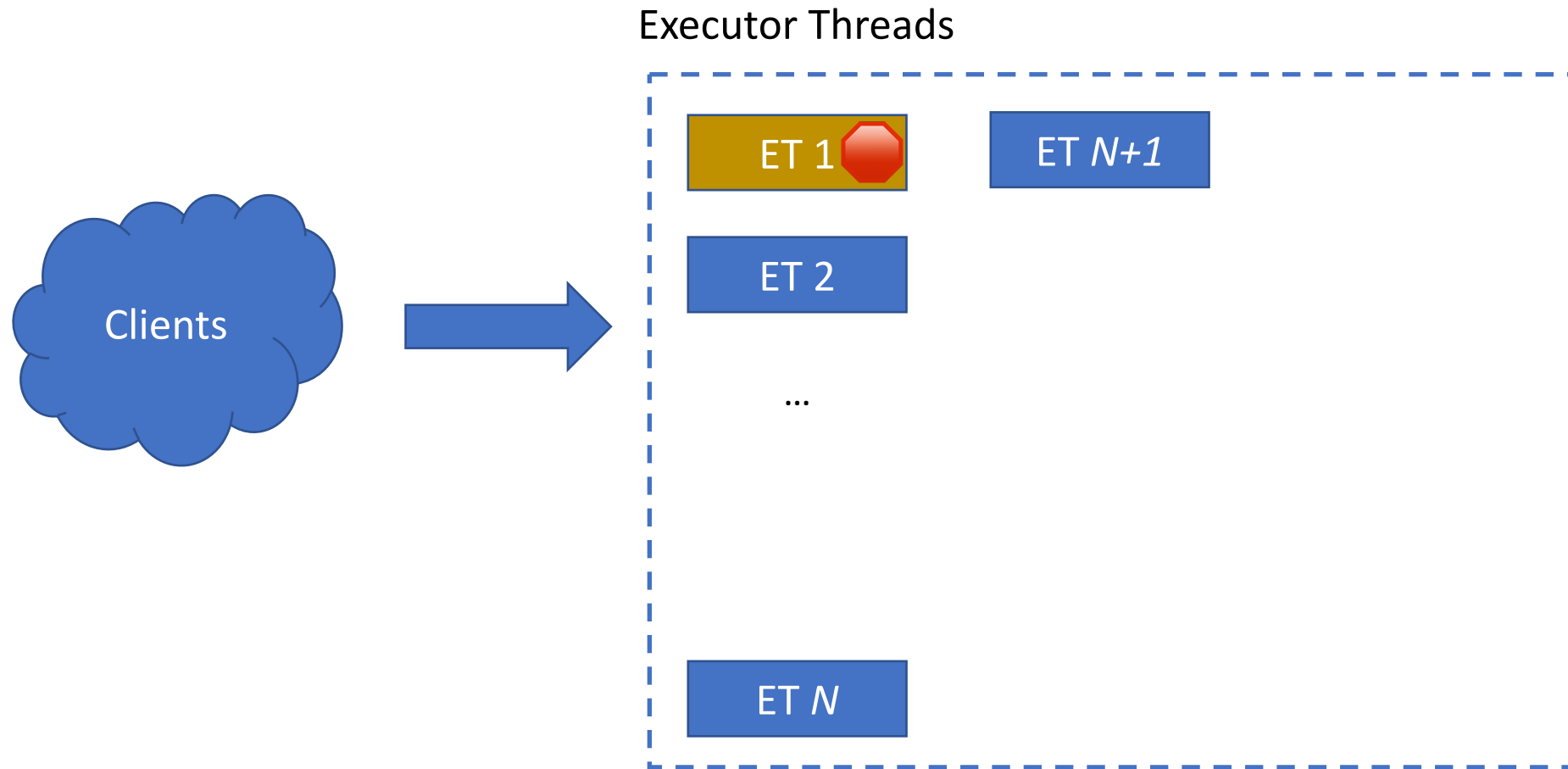
Solution – shared thread pool



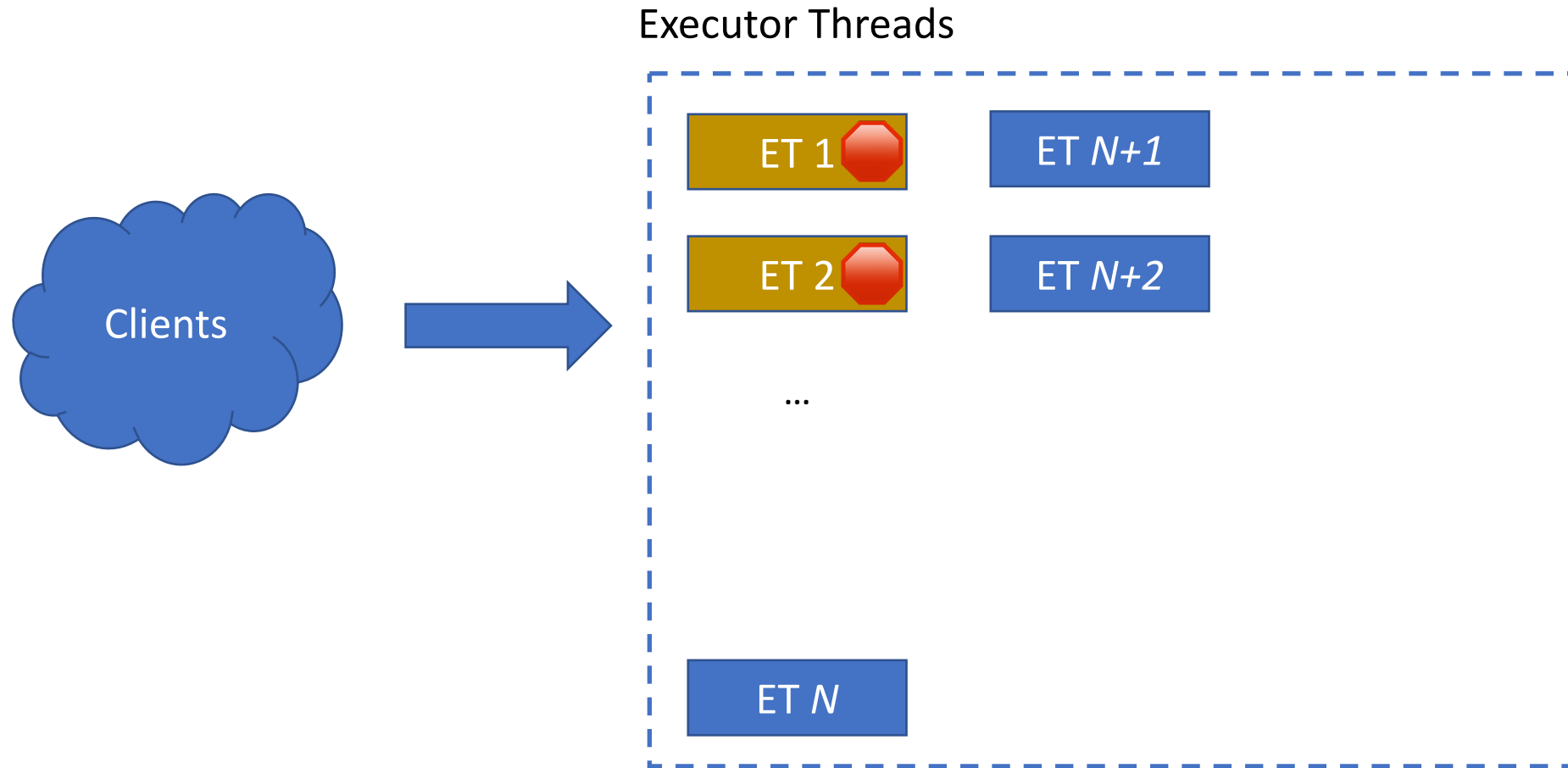
Solution – shared thread pool



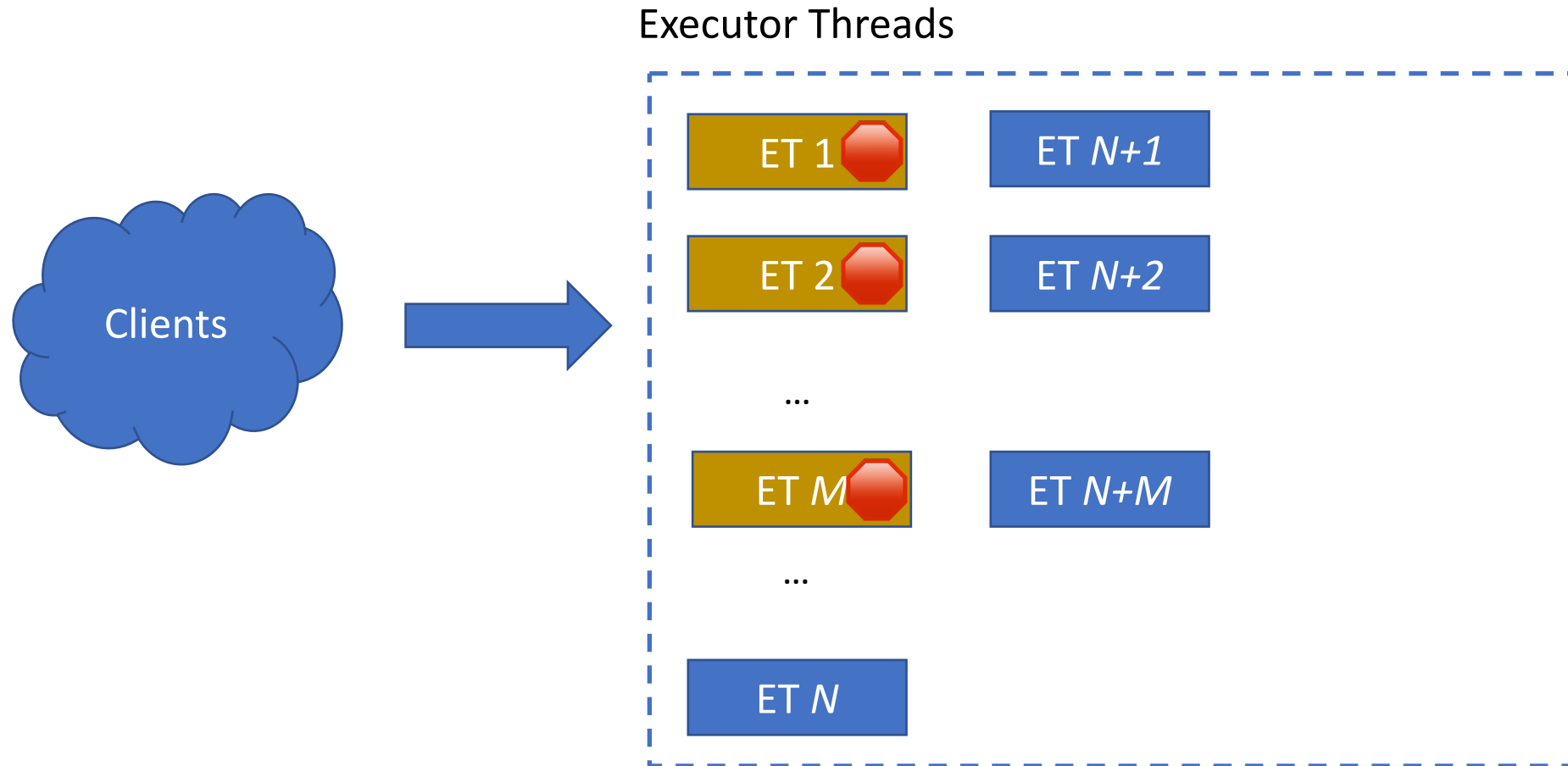
Solution – shared thread pool



Solution – shared thread pool



Solution – shared thread pool



withContext for IO

```
suspend fun loadAccount(order: Order): Account =  
    withContext(dispatcher) {  
        // some blocking code here....  
    }
```

```
val dispatcher =  
    Executors.newFixedThreadPool(M2).asCoroutineDispatcher()
```

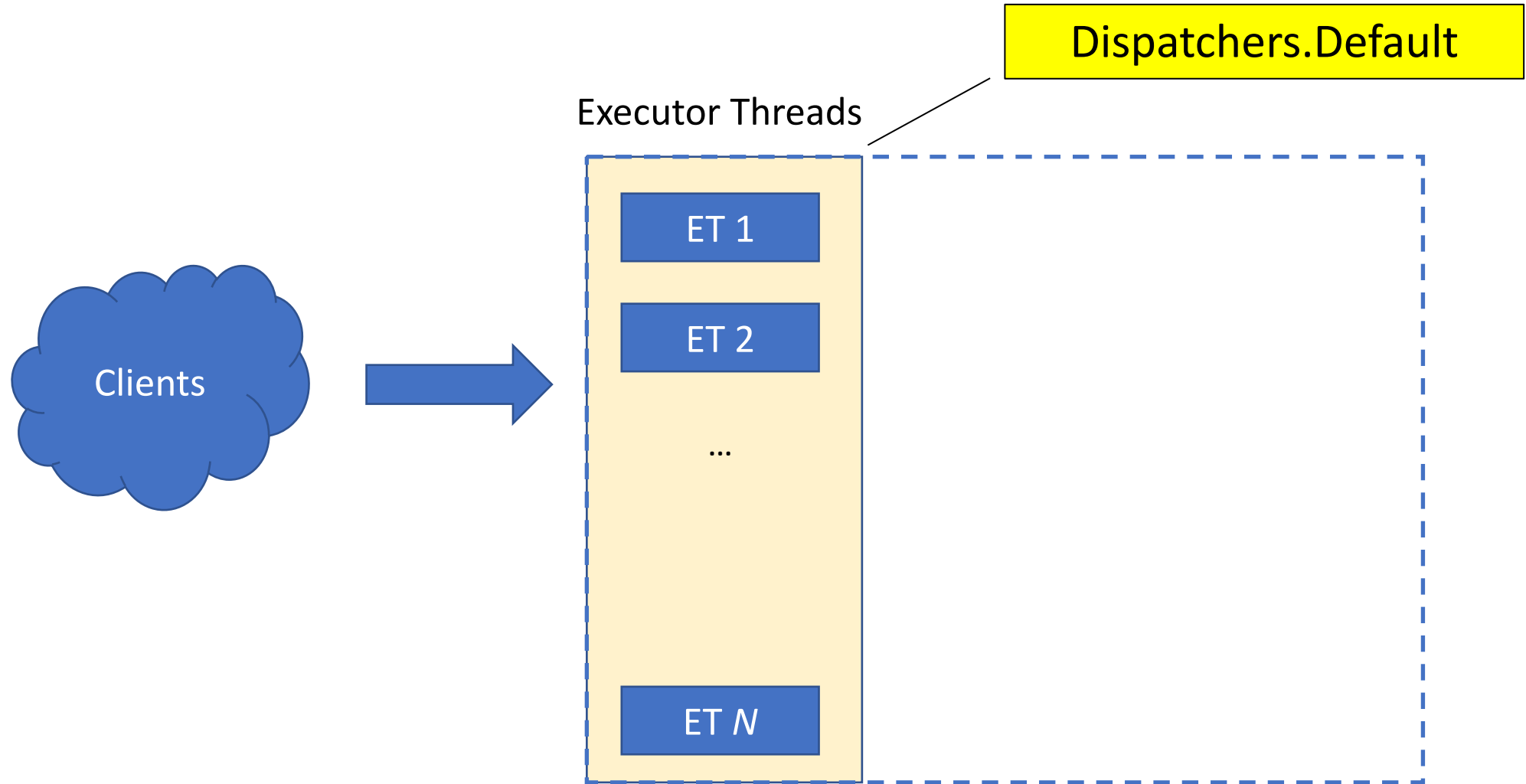
withContext for Dispatchers.IO

```
suspend fun loadAccount(order: Order): Account =  
    withContext(Dispatchers.IO) {  
        // some blocking code here....  
    }
```

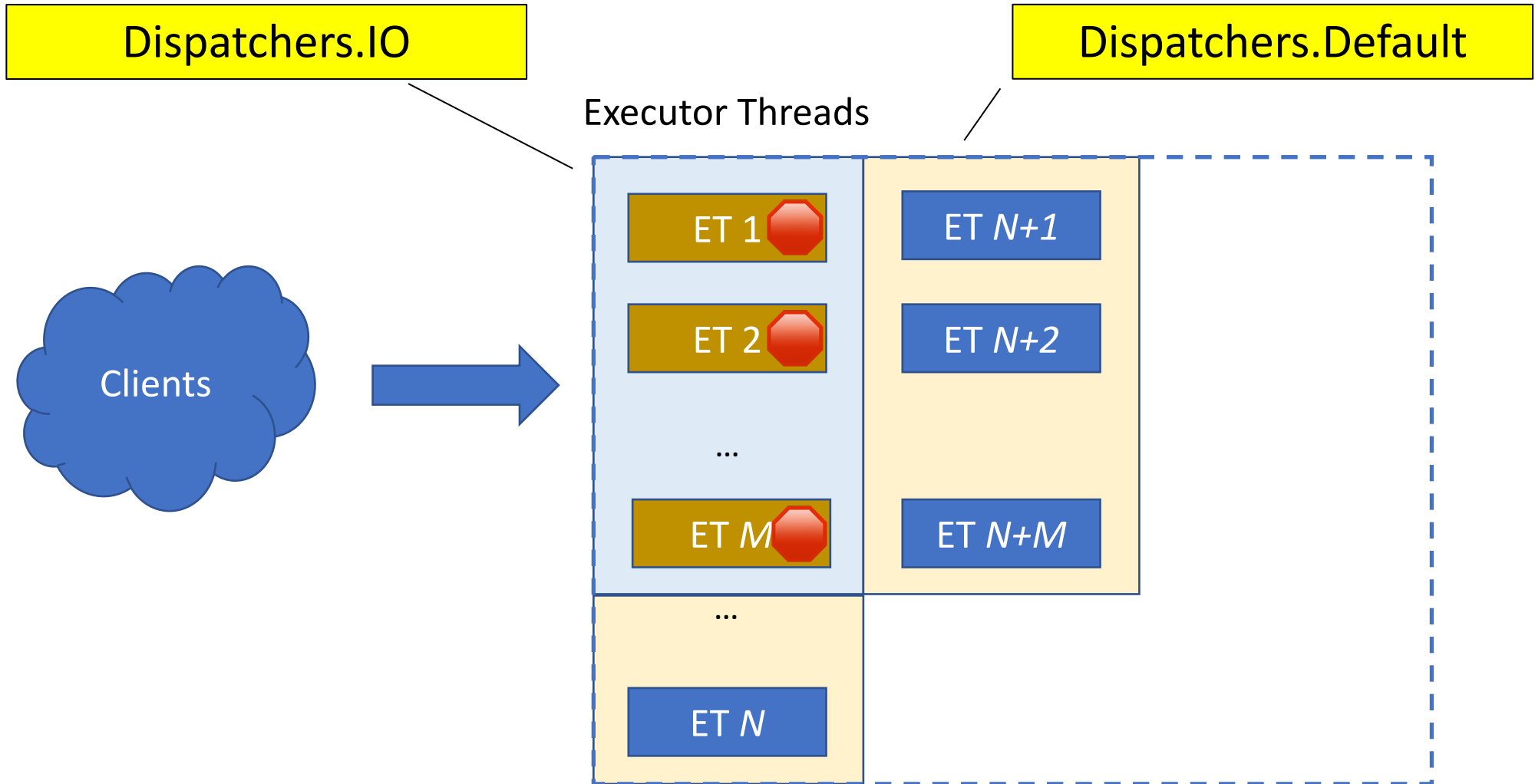


No thread switch from
Dispatchers.Default pool

Solution – shared thread pool



Solution – shared thread pool



Coroutines and data streams

Returning many responses

suspend fun foo(params: Params): Response

One response

suspend fun foo(params: Params): List<Response>

Many responses

suspend fun foo(params: Params): ???<Response>

Many responses async?

Channel



Producer

Channel type

Builder

```
fun CoroutineScope.foo(): ReceiveChannel<Int> = produce {  
    for (i in 1..10) {  
        send(i)  
        delay(100)  
    }  
}
```

Can be async

Consumer

```
fun CoroutineScope.foo(): ReceiveChannel<Int> = produce {  
    for (i in 1..10) {  
        send(i)  
        delay(100)  
    }  
}  
  
fun main() = runBlocking<Unit> {  
    for (x in foo()) {  
        println(x)  
    }  
}
```

Where's the catch?

```
fun CoroutineScope.foo(): ReceiveChannel<Int> = produce {  
    for (i in 1..10) {  
        send(i)  
        delay(100)  
    }  
}  
  
fun main() = runBlocking<Unit> {  
    for (x in foo()) {  
        println(x)  
    }  
}
```

Where's the catch?

Creates coroutine

```
fun CoroutineScope.foo(): ReceiveChannel<Int> = produce {  
    for (i in 1..10) {  
        send(i)  
        delay(100)  
    }  
}  
  
fun main() = runBlocking<Unit> {  
    for (x in foo()) {  
        println(x)  
    }  
}
```

Try this!

```
fun CoroutineScope.foo(): ReceiveChannel<Int> = produce {  
    for (i in 1..10) {  
        send(i)  
        delay(100)  
    }  
}
```



```
fun main() = runBlocking<Unit> {  
    foo()  
}
```

Waits for completion of children

Kotlin Flows

Disclaimer: available in preview only, not stable yet

Flow example

~ Asynchronous *sequence*

```
fun bar(): Flow<Int> = flow {  
    for (i in 1..10) {  
        emit(i)  
        delay(100)  
    }  
}
```

Flow example

```
fun bar(): Flow<Int> = flow {  
    for (i in 1..10) {  
        emit(i)  
        delay(100)  
    }  
}  
  
fun main() = runBlocking<Unit> {  
    bar().collect { x ->  
        println(x)  
    }  
}
```

Try this!

```
fun bar(): Flow<Int> = flow {  
    for (i in 1..10) {  
        emit(i)  
        delay(100)  
    }  
}
```

Flow is *cold*: describes the data,
does not run it until collected

```
👉 fun main() = runBlocking<Unit> {  
    bar()  
}
```

Flow example

```
fun bar(): Flow<Int> = flow {  
    for (i in 1..10) {  
        emit(i)  
        delay(100)  
    }  
}  
  
fun main() = runBlocking<Unit> {  
    bar()  
        .map { it * it }  
        .toList()  
}
```

Write regular code!
Similar to collections / sequences

Thank you

Want to learn more? Questions?



Roman Elizarov



elizarov @
relizarov





Please

**Remember to
rate this session**

Thank you!

