Async Await

AND MORE

CAMERON DEARDORFF

@camdeardoff

Cameron Deardorff

@camdeardoff

Cameron Deardorff

@camdeardoff

Cameron Deardorff

@camdeardoff

Cameron Deardorff

@camdeardoff

Cameron Deardorff

ALL CODE EXAMPLES

Cows

ALL CODE EXAMPLES

360|iDev

Why Cows?

Glad you asked

Why Cows?

Glad you asked

- Never in a hurry ➜ Asynchrony

Why Cows?

Glad you asked

- Never in a hurry ➜ Asynchrony
- Lots of cows ➜ Concurrency

Why Cows?

Glad you asked

- Never in a hurry ➜ Asynchrony
- Lots of cows ➜ Concurrency
- Emoji Rich Environment

Why Cows?

Glad you asked

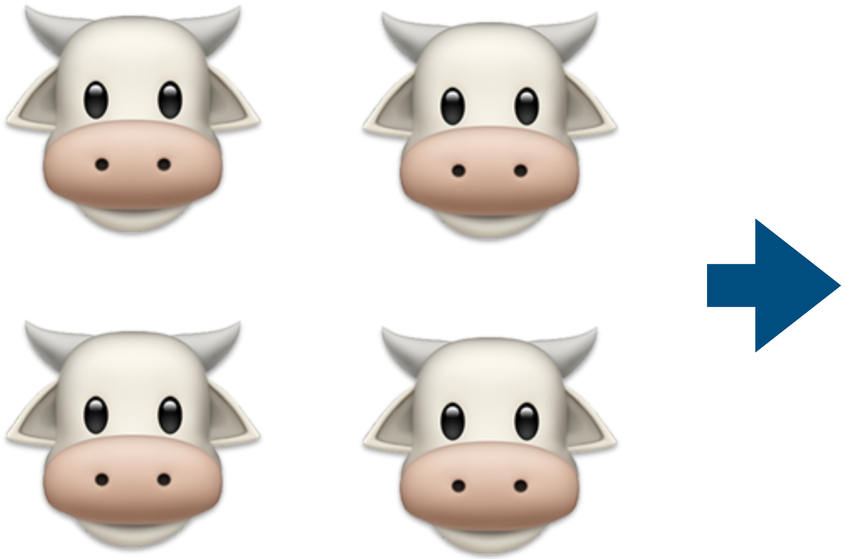
- Never in a hurry ➜ Asynchrony
- Lots of cows ➜ Concurrency
- Emoji Rich Environment

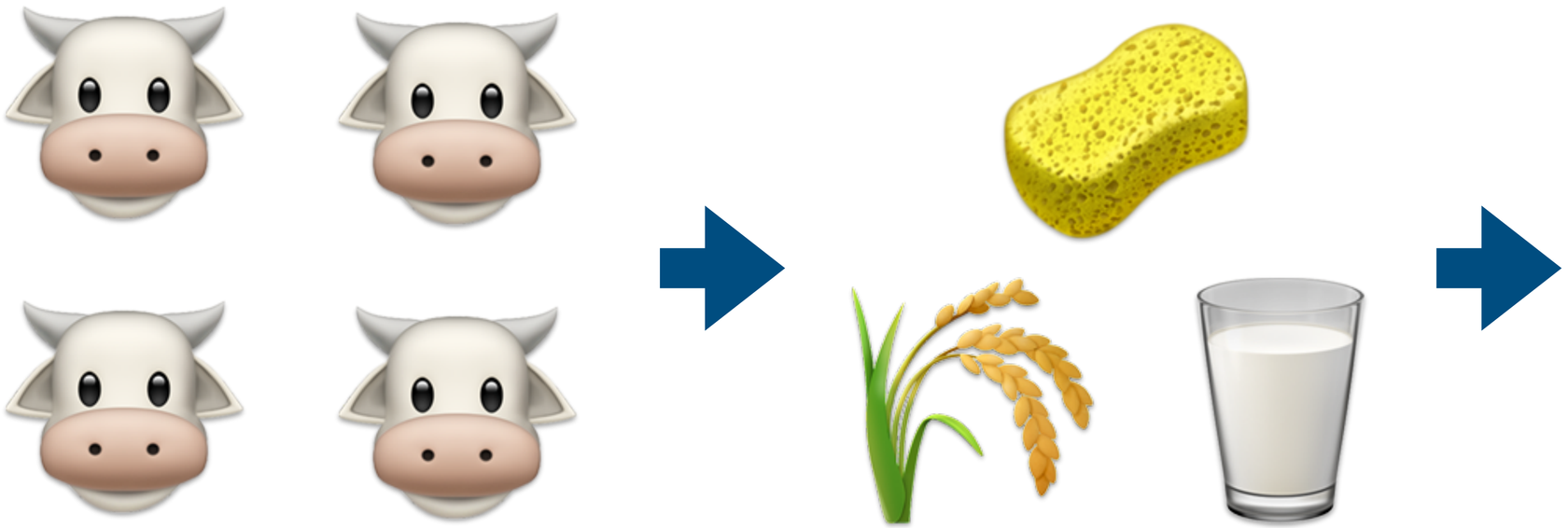


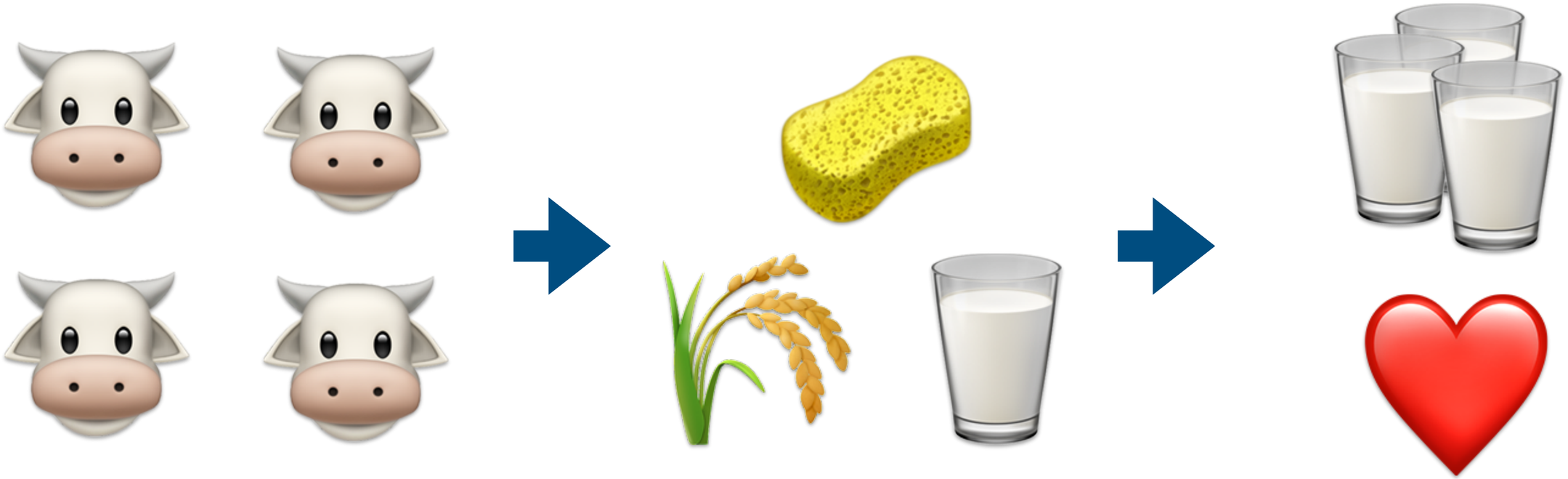












Coding for Asynchrony

with Async and Await

Asynchronous Programing Paradigms

A Farewell Tour

Completion Handlers

```
fetchCow { cow in
    feedCow(cow) { isFull in
        guard !isFull else {
            completion(.failure(CowError.unhappyCow))
            return
        }
        cow.collectMilk { milk in
            completion(.success(milk))
        }
    }
}
```

Completion Handlers

- Easy and simple to set up

```
fetchCow { cow in
    feedCow(cow) { isFull in
        guard !isFull else {
            completion(.failure(CowError.unhappyCow))
            return
        }
        cow.collectMilk { milk in
            completion(.success(milk))
        }
    }
}
```

Completion Handlers

- Easy and simple to set up
- Error prone

```
fetchCow { cow in
    feedCow(cow) { isFull in
        guard !isFull else {
            completion(.failure(CowError.unhappyCow))
            return
        }
        cow.collectMilk { milk in
            completion(.success(milk))
        }
    }
}
```

Completion Handlers

- Easy and simple to set up
- Error prone
- Hard to scale

```
fetchCow { cow in
    feedCow(cow) { isFull in
        guard !isFull else {
            completion(.failure(CowError.unhappyCow))
            return
        }
        cow.collectMilk { milk in
            completion(.success(milk))
        }
    }
}
```

Promises

```
fetchCow()
    .then { cow -> Promise<(Cow, Bool), CowError> in
        return feedCow(cow)
    }
    .map { (cow, isFull) -> Promise<Milk, CowError> in
        guard isFull else { return CowError.unhappyCow }
        return cow.collectMilk()
    }
    .then { milk in
        // do something with milk
    }
    .catch {
        // handle error
    }
```

Promises

- Chaining and transformations

```
fetchCow()
    .then { cow -> Promise<(Cow, Bool), CowError> in
        return feedCow(cow)
    }
    .map { (cow, isFull) -> Promise<Milk, CowError> in
        guard isFull else { return CowError.unhappyCow }
        return cow.collectMilk()
    }
    .then { milk in
        // do something with milk
    }
    .catch {
        // handle error
    }
```

Promises

- Chaining and transformations
- Poor local reasoning

```
fetchCow()
    .then { cow -> Promise<(Cow, Bool), CowError> in
        return feedCow(cow)
    }
    .map { (cow, isFull) -> Promise<Milk, CowError> in
        guard isFull else { return CowError.unhappyCow }
        return cow.collectMilk()
    }
    .then { milk in
        // do something with milk
    }
    .catch {
        // handle error
    }
```

Async & Await

```
func fetchCow() async -> Cow {
    // does asynchronous network fetch
}

...

let cow = await fetchCow()
let isClean = await washCow(cow)
let isFull = await feedCow(cow)
let milk = await cow.collectMilk()
```

Async & Await

- Mark asynchronous functions

```
func fetchCow() async -> Cow {
    // does asynchronous network fetch
}

...

let cow = await fetchCow()
let isClean = await washCow(cow)
let isFull = await feedCow(cow)
let milk = await cow.collectMilk()
```

Async & Await

- Mark asynchronous functions
- Function suspension

```
func fetchCow() async -> Cow {
    // does asynchronous network fetch
}

...

let cow = await fetchCow()
let isClean = await washCow(cow)
let isFull = await feedCow(cow)
let milk = await cow.collectMilk()
```

Async & Await

- Mark asynchronous functions
- Function suspension
- Sequentially execution

```
func fetchCow() async -> Cow {
    // does asynchronous network fetch
}

...

let cow = await fetchCow()
let isClean = await washCow(cow)
let isFull = await feedCow(cow)
let milk = await cow.collectMilk()
```

Async/Await ❤️ Throws/Try

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    let cow = try await fetchCow()
    let isClean = await washCow(cow)
    let isFull = try await feedCow(cow)
    let milk = try await cow.collectMilk()
} catch let error {
    // handle error
}
```

Async/Await ❤️ Throws/Try

- Similar Paradigm

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    let cow = try await fetchCow()
    let isClean = await washCow(cow)
    let isFull = try await feedCow(cow)
    let milk = try await cow.collectMilk()
} catch let error {
    // handle error
}
```

Async/Await ❤️ Throws/Try

- Similar Paradigm
- Compiler enforcement

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    let cow = try await fetchCow()
    let isClean = await washCow(cow)
    let isFull = try await feedCow(cow)
    let milk = try await cow.collectMilk()
} catch let error {
    // handle error
}
```

Async/Await ❤️ Throws/Try

- Similar Paradigm
- Compiler enforcement
- `async` & `throws` are composable

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    let cow = try await fetchCow()
    let isClean = await washCow(cow)
    let isFull = try await feedCow(cow)
    let milk = try await cow.collectMilk()
} catch let error {
    // handle error
}
```

Ye Olden Days

```
fetchCow { result in
    switch result {
    case .success(let cow):
        feedCow(cow) { result in
            switch result {
            case .success(let isFull):
                // ...
            case .failure(let error):
                // ...
            }
        }
    case .failure(error):
        // ...
    }
}
```

Ye Olden Days

- Completion handler with Result

```
fetchCow { result in
    switch result {
    case .success(let cow):
        feedCow(cow) { result in
            switch result {
            case .success(let isFull):
                // ...
            case .failure(let error):
                // ...
            }
        }
    case .failure(error):
        // ...
    }
}
```

Ye Olden Days

- Completion handler with Result
- Error handling without `throws`

```
fetchCow { result in
    switch result {
    case .success(let cow):
        feedCow(cow) { result in
            switch result {
            case .success(let isFull):
                // ...
            case .failure(let error):
                // ...
            }
        }
    case .failure(error):
        // ...
    }
}
```

Ye Olden Days

- Completion handler with Result
- Error handling without `throws`
- Pyramid of doom @2x

```
fetchCow { result in
    switch result {
    case .success(let cow):
        feedCow(cow) { result in
            switch result {
            case .success(let isFull):
                // ...
            case .failure(let error):
                // ...
            }
        }
    case .failure(error):
        // ...
    }
}
```


360|iDev

Completion Handlers with Result

```
fetchCow { result in
    switch result {
    case .success(let cow):
        feedCow(cow) { result in
            switch result {
            case .success(let isFull):
                // ...
            case .failure(let error):
                // ...
            }
        }
    case .failure(error):
        // ...
    }
}
```

Completion Handlers with Result

```
fetchCow { result in
    switch result {
    case .success(let cow):
        feedCow(cow) { result in
            switch result {
            case .success(let isFull):
                // ...
            case .failure(let error):
                // ...
            }
        }
    case .failure(error):
        // ...
    }
}
```

async/await with try/catch

```
do {
    let cow = try await fetchCow()
    let isClean = await washCow(cow)
    let isFull = try await feedCow(cow)
    let milk = try await cow.collectMilk()
} catch let error {
    // handle error
}
```

Function Suspension

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle
- Return actual value

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle
- Return actual value
- Nothing to work with synchronously

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle
- Return actual value
- Nothing to work with synchronously
- Non-blocking

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle
- Return actual value
- Nothing to work with synchronously
- Non-blocking

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle
- Return actual value
- Nothing to work with synchronously
- Non-blocking

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
    // checkpoint B
} catch let error {
    // ...
}
```

Function Suspension

- No future, promise, or handle
- Return actual value
- Nothing to work with synchronously
- Non-blocking

```
func fetchCow() async throws -> Cow {
    // does asynchronous network fetch
}

...

do {
    // checkpoint A
    let cow = try await fetchCow()
➜   // checkpoint B
} catch let error {
    // ...
}
```

Async Everywhere

Async Everywhere

- Functions

Async Everywhere

- Functions
- Property getters

Async Everywhere

- Functions
- Property getters
- Initializers

Async Everywhere

- Functions
- Property getters
- Initializers

Parallel Processing

with Structured Concurrency

Structured Concurrency

Structured Concurrency

- Structured Programming
  - if / else
  - for / while loops
  - functions

Structured Concurrency

- Structured Programming
  - if / else
  - for / while loops
  - functions
- Structured Programming with an understanding of concurrency

What does this mean for me writing Swift apps?

What does this mean for me writing Swift apps?

Concurrent and asynchronous code that looks and feels *normal*.

Async Let

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

Async Let

- Run in parallel

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

Async Let

- Run in parallel
- `await` when needed

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

Async Let

- Run in parallel
- `await` when needed
- Observe errors on `await`

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

Task

Foundational unit in running concurrent code

Task

Foundational unit in running concurrent code

- Run in the background

Task

Foundational unit in running concurrent code

- Run in the background
- Support Asynchrony
  - Running, Suspended, or Cancelled
  - All async work is done in Tasks

Task

Foundational unit in running concurrent code

- Run in the background
- Support Asynchrony
  - Running, Suspended, or Cancelled
  - All async work is done in Tasks
- Run in parallel

Task

Foundational unit in running concurrent code

- Run in the background
- Support Asynchrony
  - Running, Suspended, or Cancelled
  - All async work is done in Tasks
- Run in parallel
- Non-atomic

Async Let + Tasks

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

Async Let + Tasks

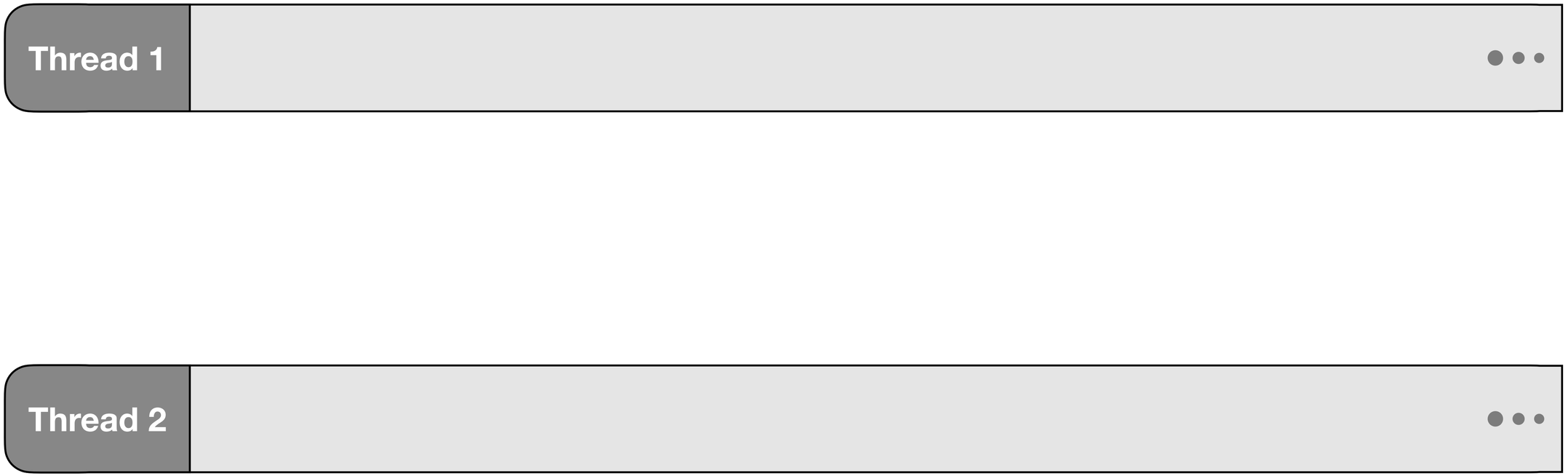
- Creates new child task

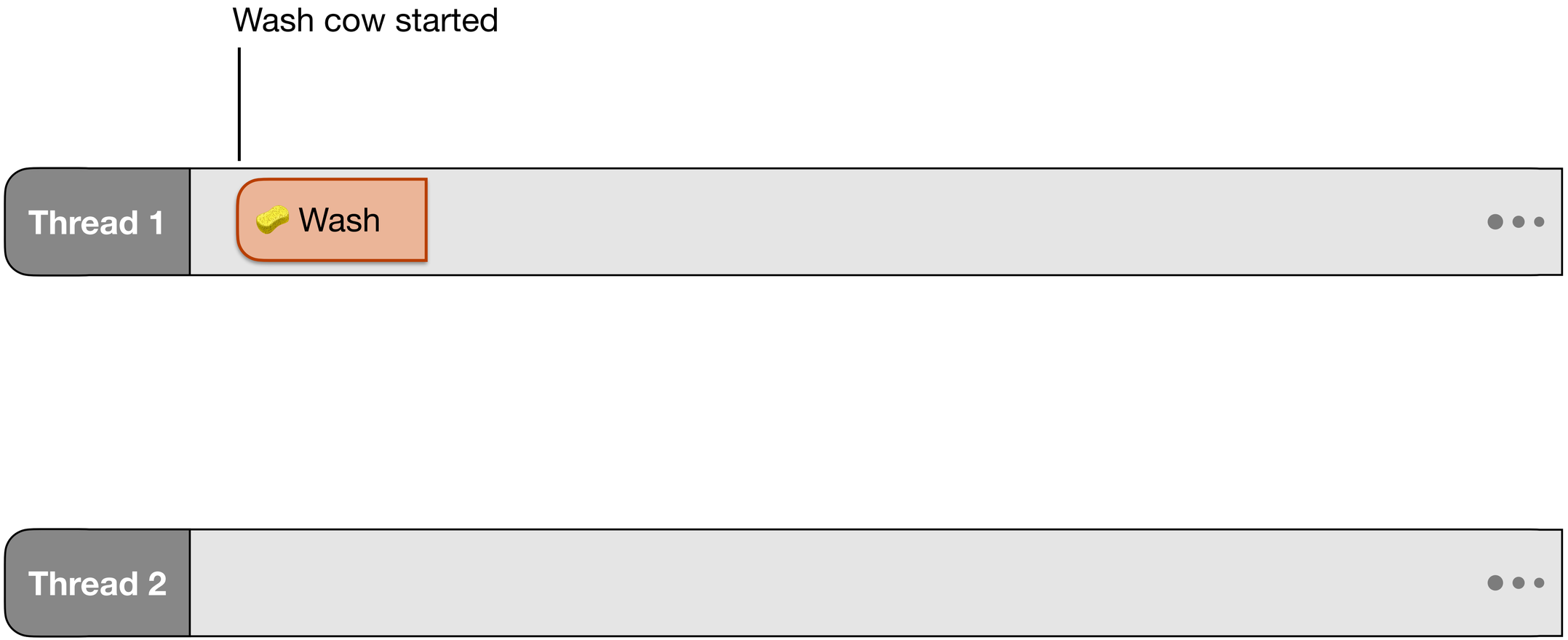
```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

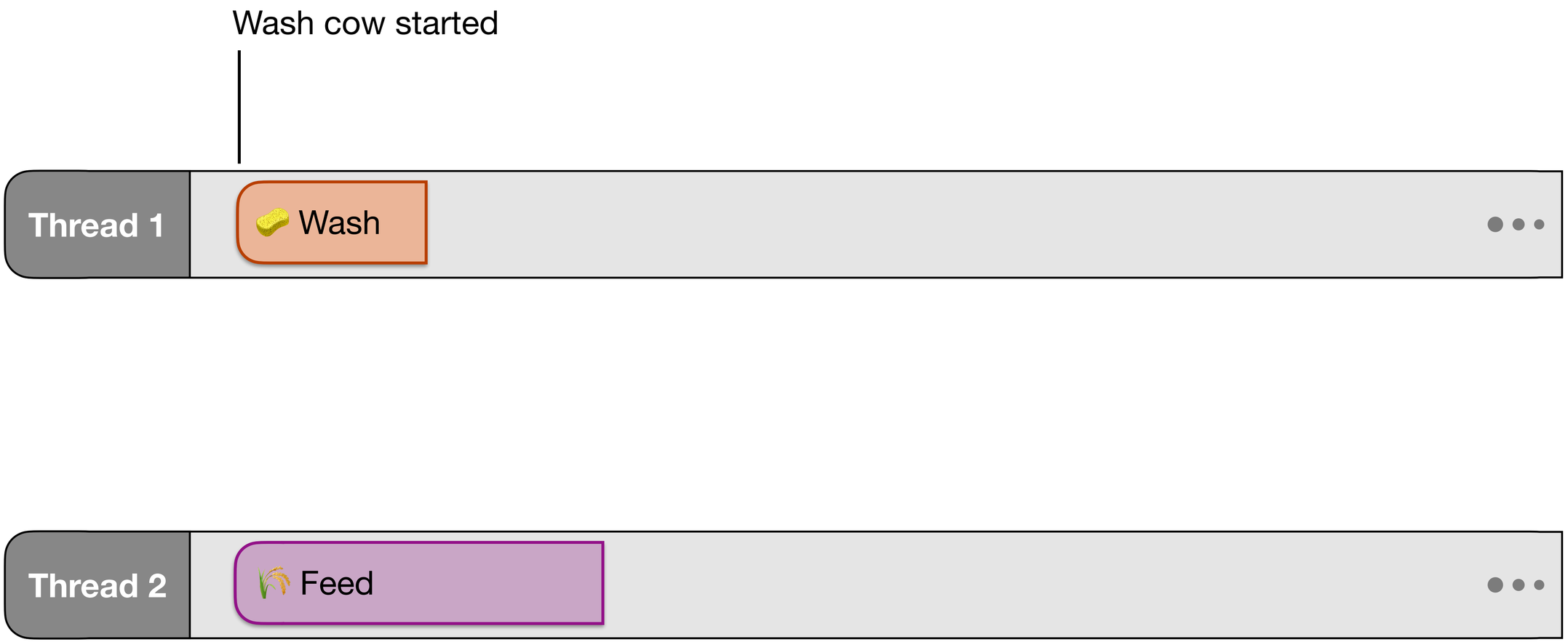
Async Let + Tasks

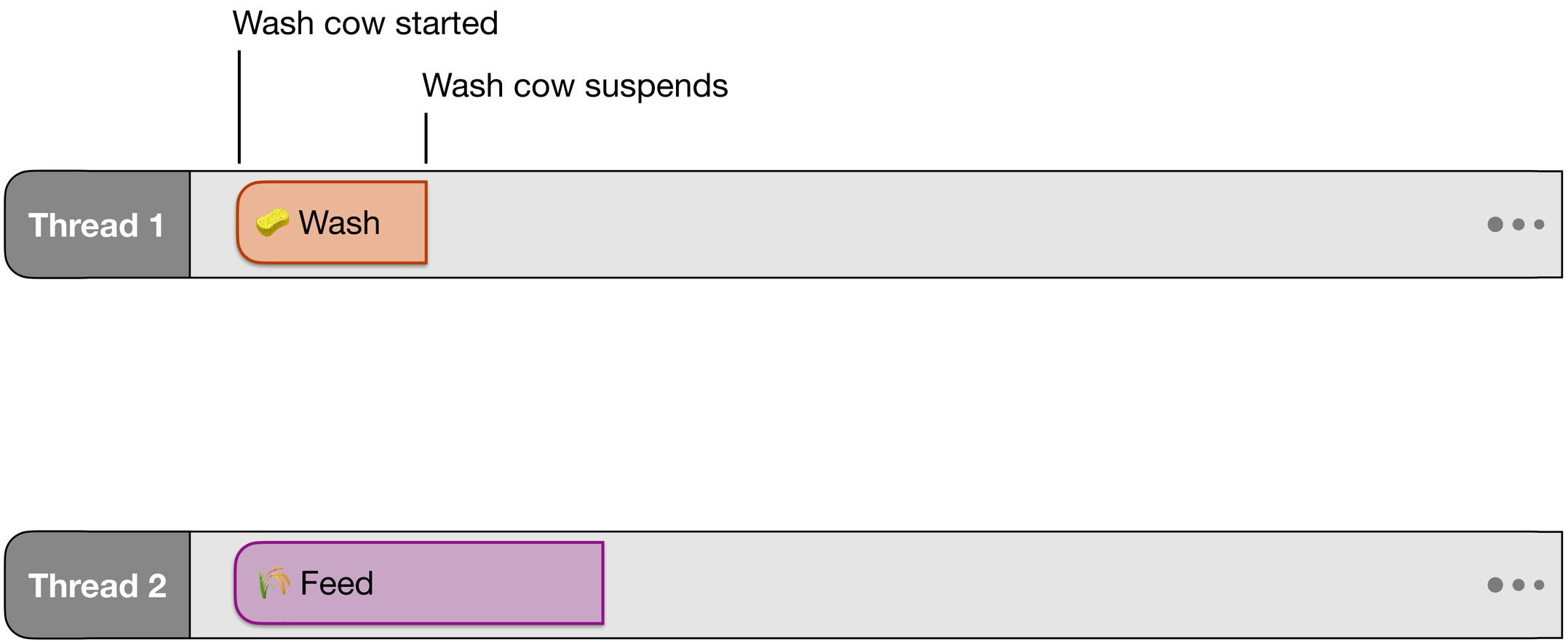
- Creates new child task
- Tasks run in parallel if/when possible

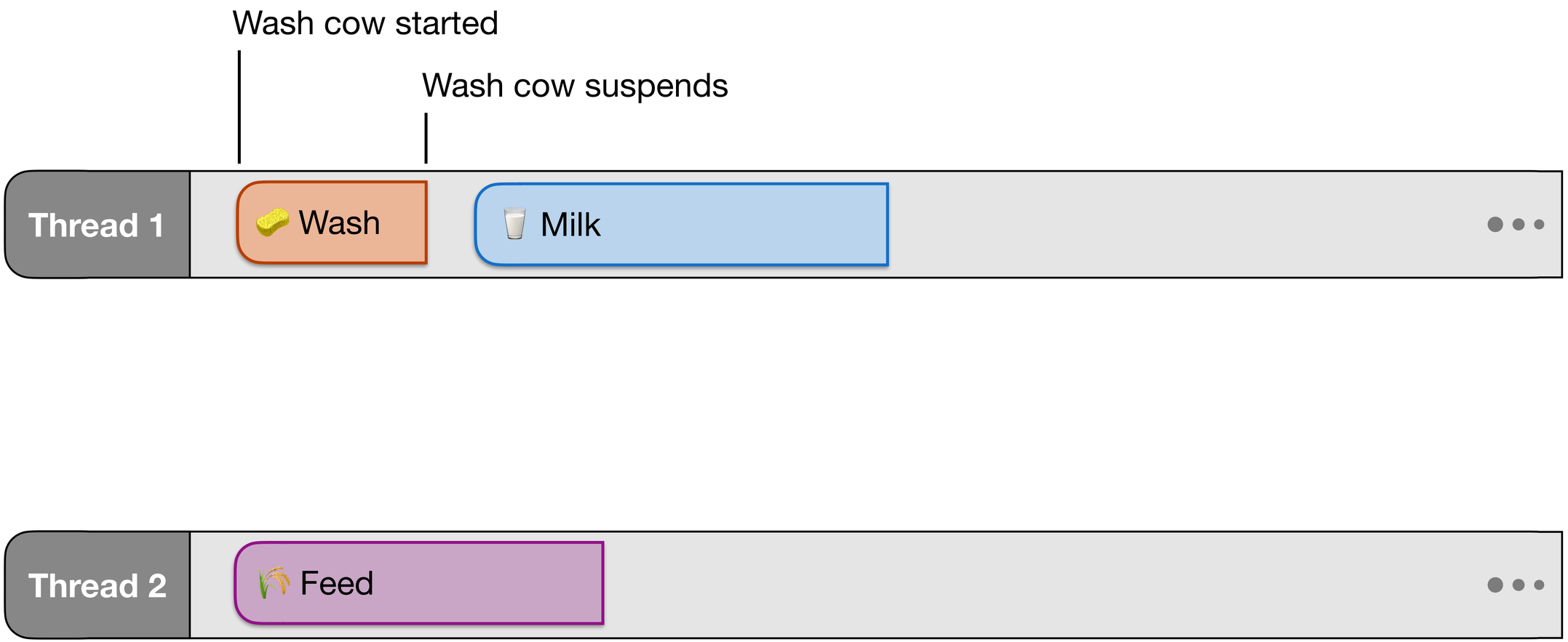
```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    // execution continues

    guard try await isFull,
        try await isClean else {
        throw FarmError.unhappyCow
    }
    return try await milk
}
```

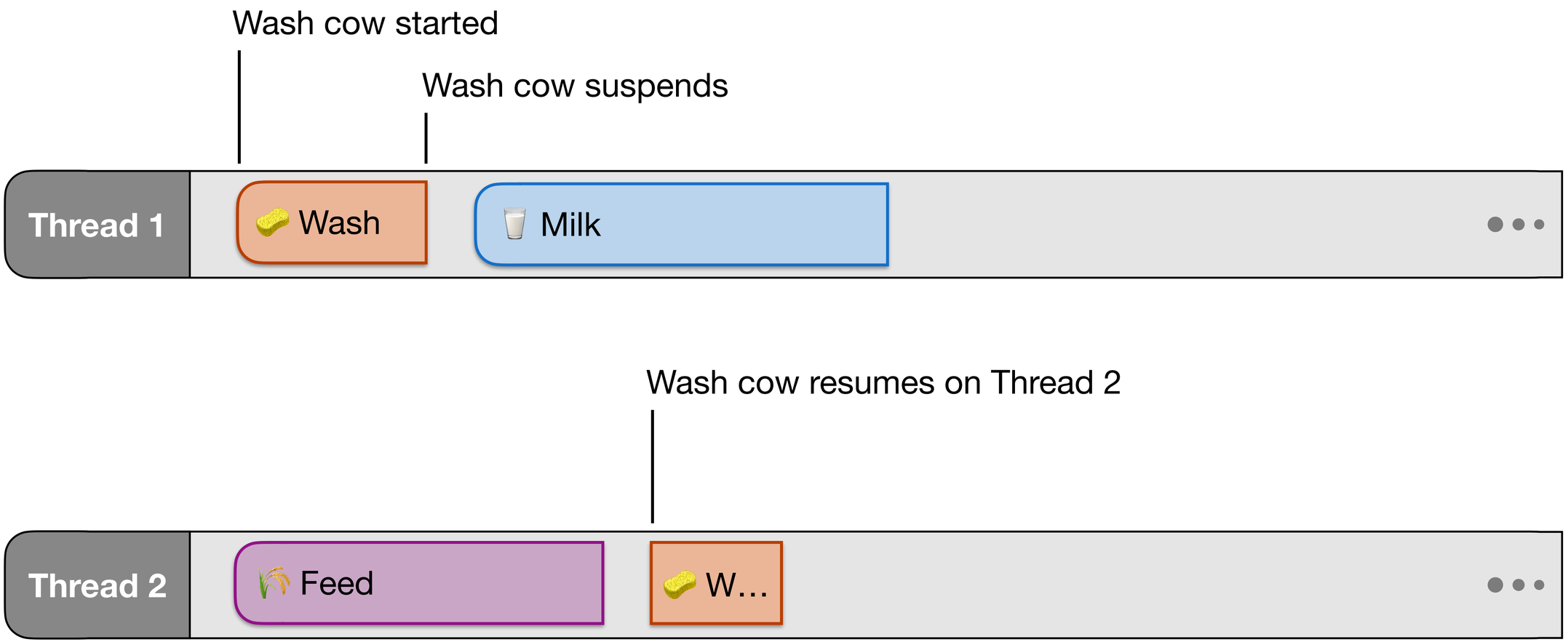

Thread 1
Thread 2

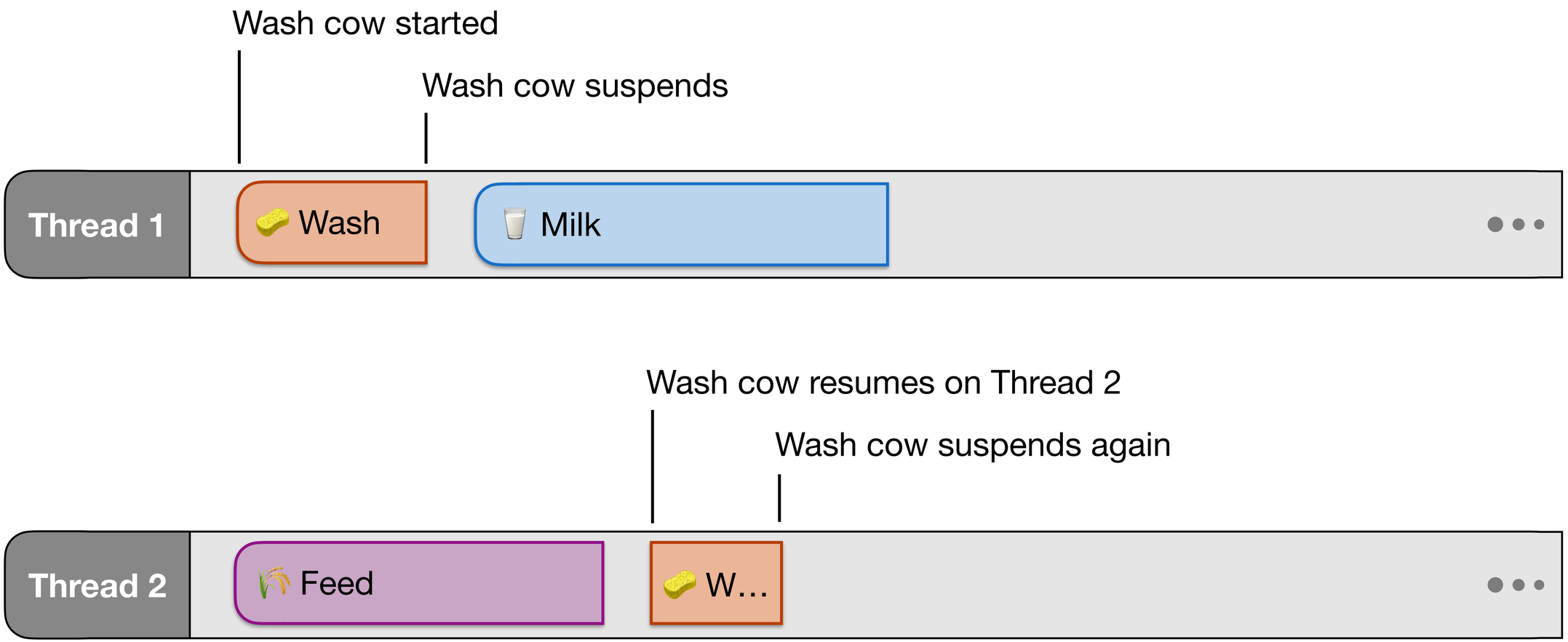

Wash cow started
Thread 1
Wash
Thread 2

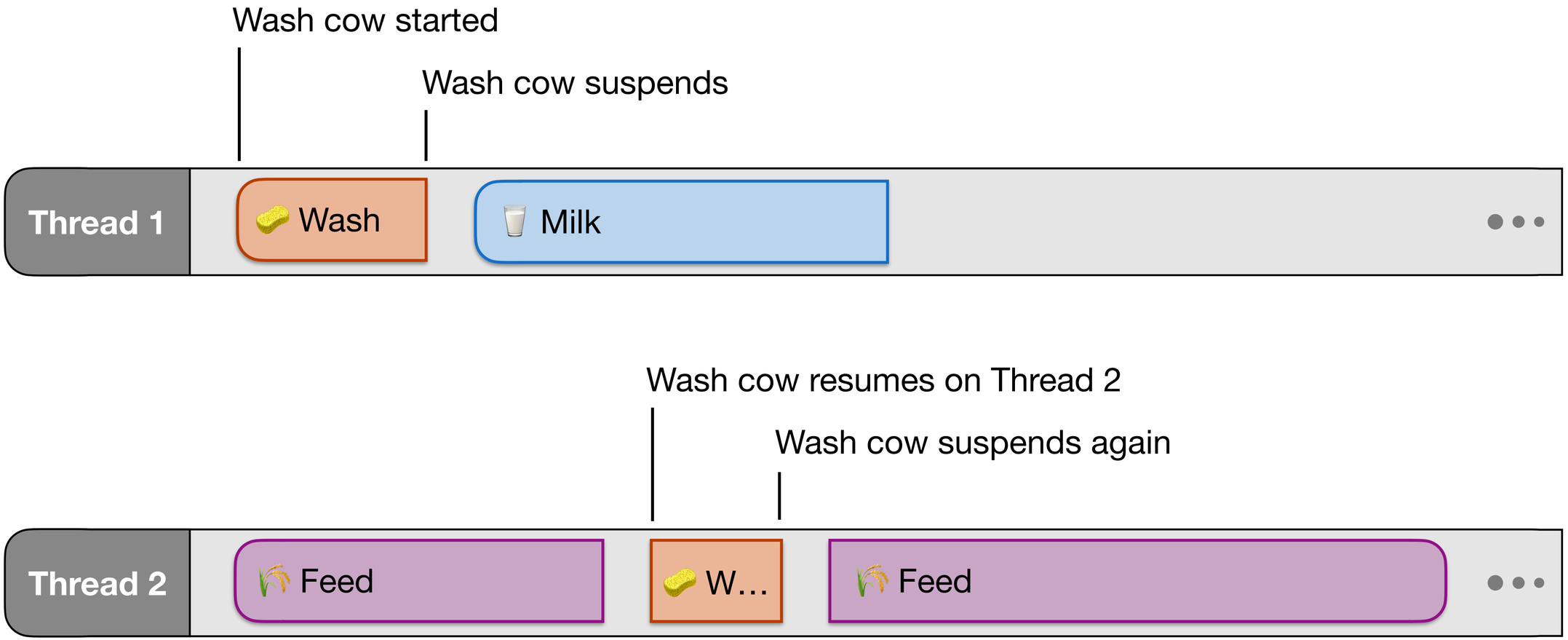

Wash cow started
Thread 1
Wash
Thread 2
Feed

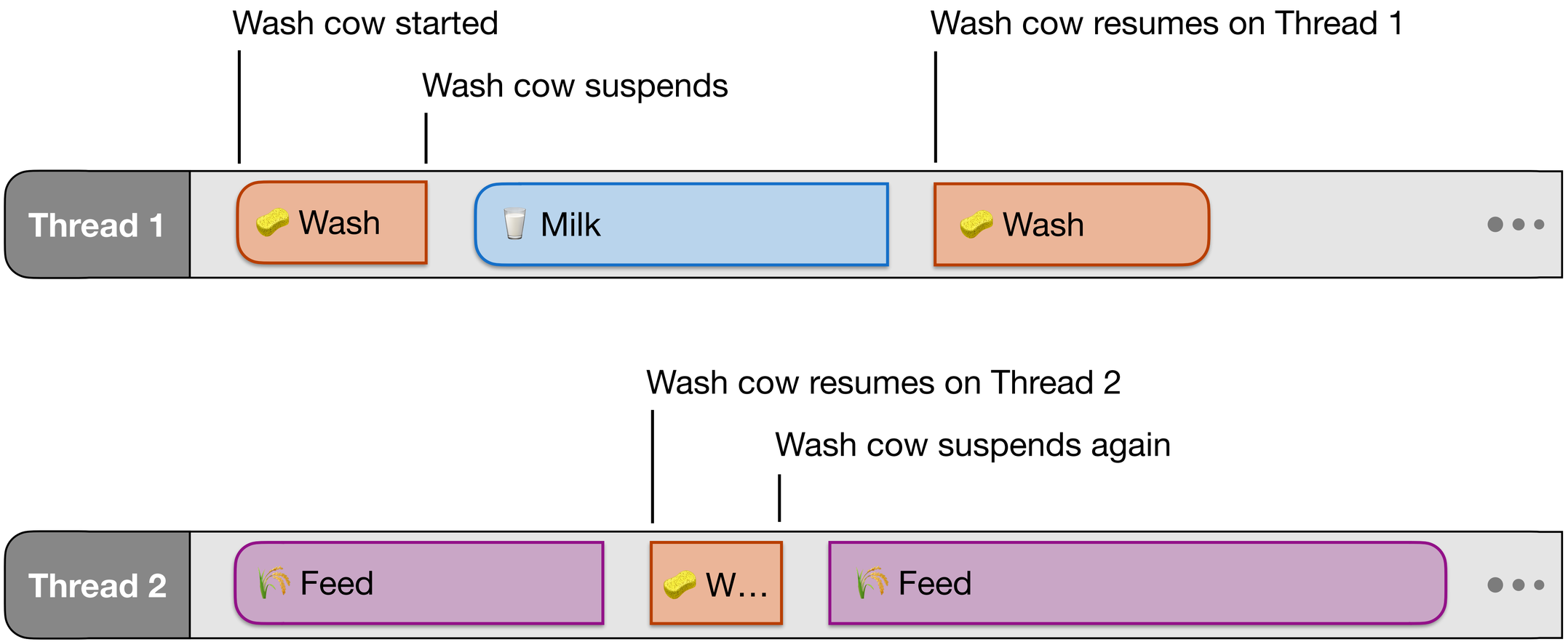

Wash cow started
Wash cow suspends
Thread 1
Wash
Thread 2
Feed

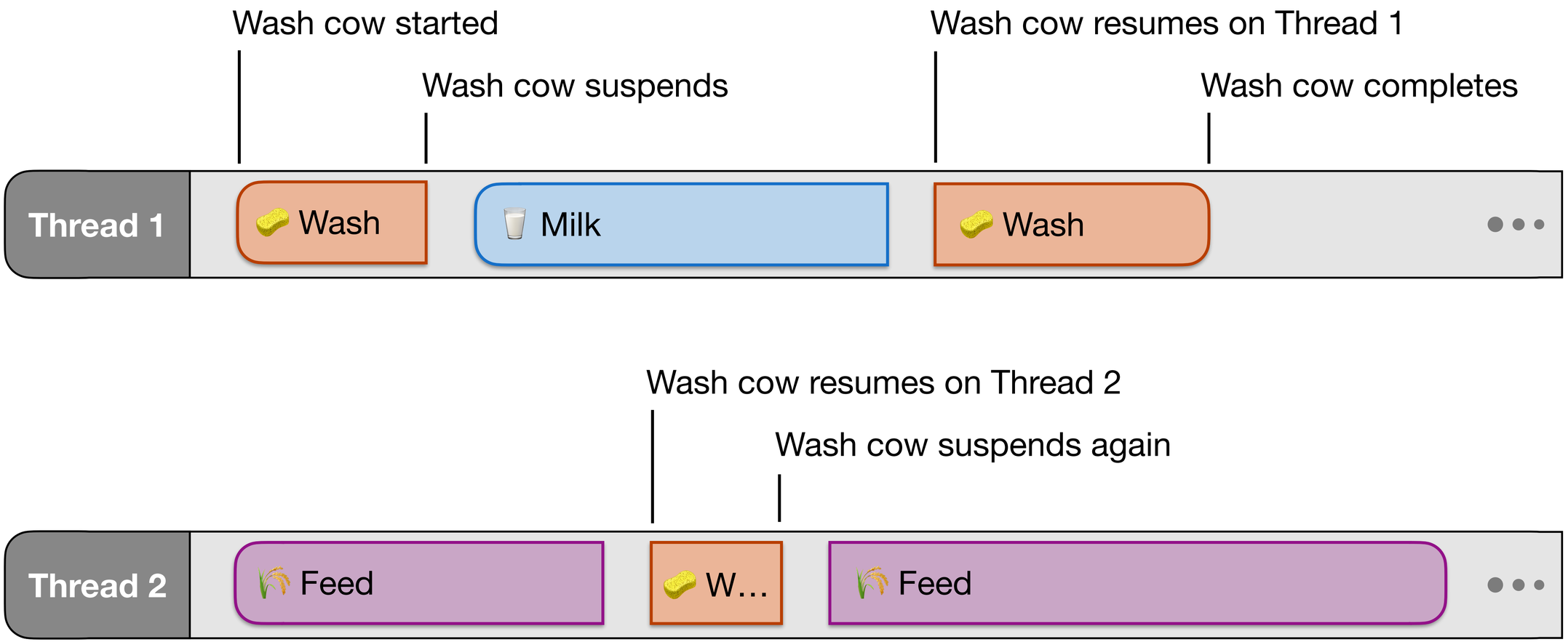

Wash cow started
Wash cow suspends
Thread 1
Wash
Milk
Thread 2
Feed

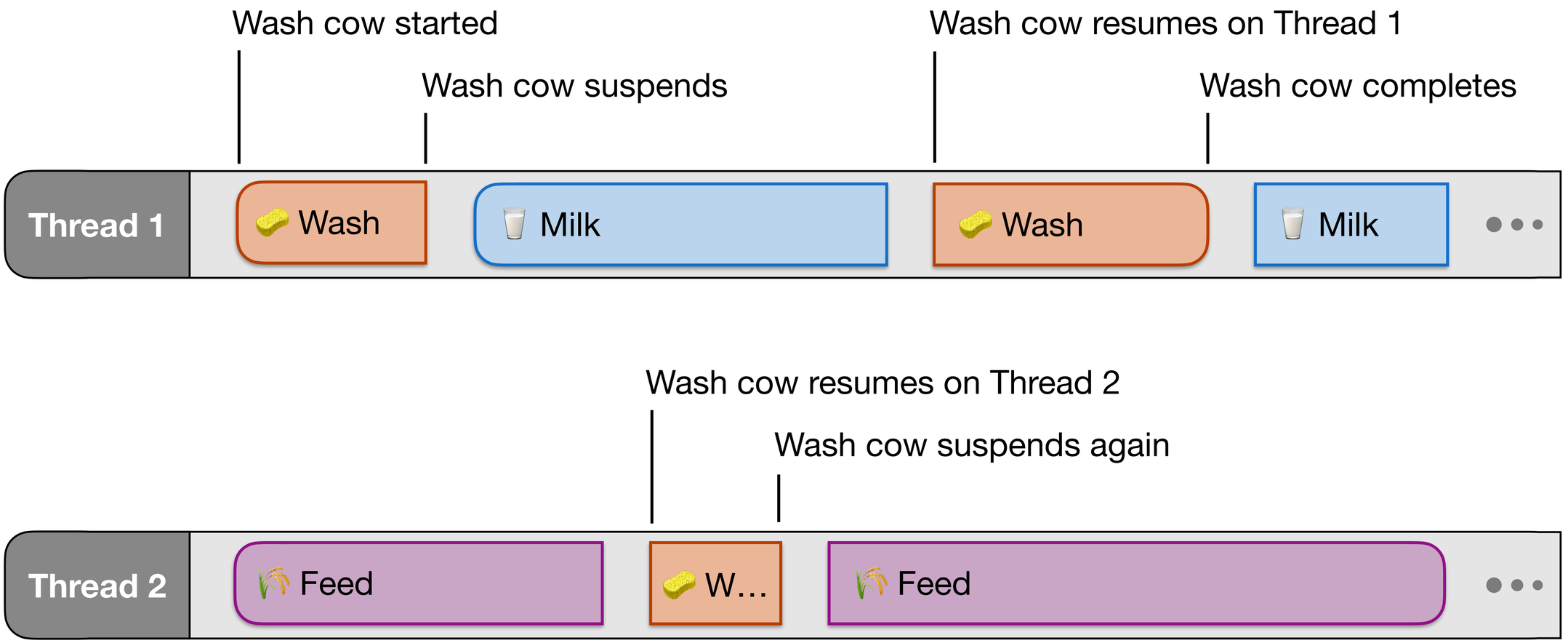

Wash cow started
Wash cow suspends
Thread 1
Wash
Milk
Wash cow resumes on Thread 2
Thread 2
Feed
W…


Wash cow started
Wash cow suspends
Thread 1
Wash
Milk
Wash cow resumes on Thread 2
Wash cow suspends again
Thread 2
Feed
W…


Wash cow started
Wash cow suspends
Thread 1
Wash
Milk
Wash cow resumes on Thread 2
Wash cow suspends again
Thread 2
Feed
W…
Feed


Wash cow started
Wash cow suspends
Wash cow resumes on Thread 1
Thread 1
Wash
Milk
Wash
Wash cow resumes on Thread 2
Wash cow suspends again
Thread 2
Feed
W…
Feed


Wash cow started
Wash cow suspends
Wash cow resumes on Thread 1
Wash cow completes
Thread 1
Wash
Milk
Wash
Wash cow resumes on Thread 2
Wash cow suspends again
Thread 2
Feed
W…
Feed


Wash cow started
Wash cow suspends
Wash cow resumes on Thread 1
Wash cow completes
Thread 1
Wash
Milk
Wash
Milk
Wash cow resumes on Thread 2
Wash cow suspends again
Thread 2
Feed
W…
Feed

Task Tree

Task Tree

- Tasks can have child tasks

Task Tree

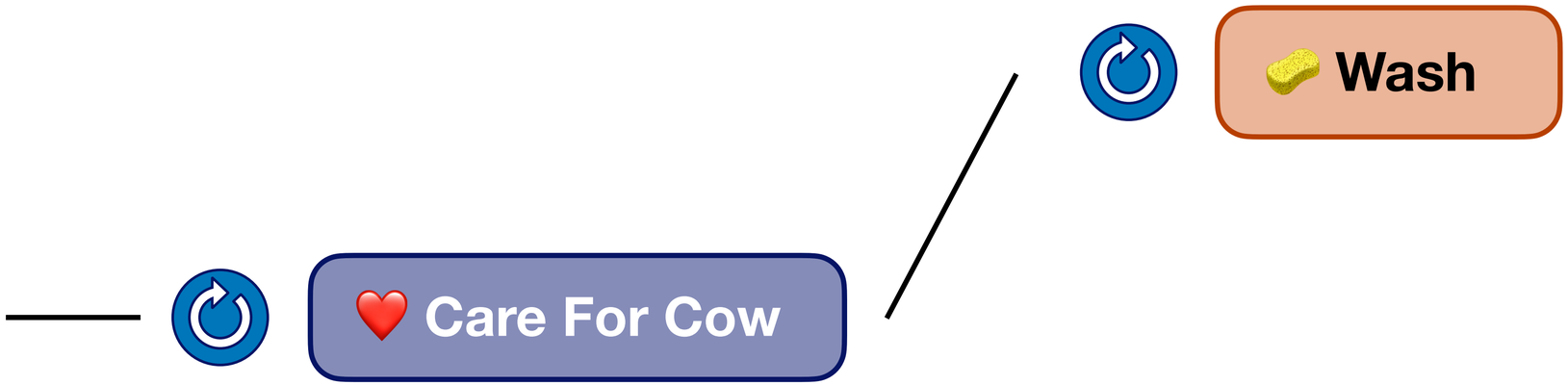
- Tasks can have child tasks
- Child tasks inherit properties of their parent task

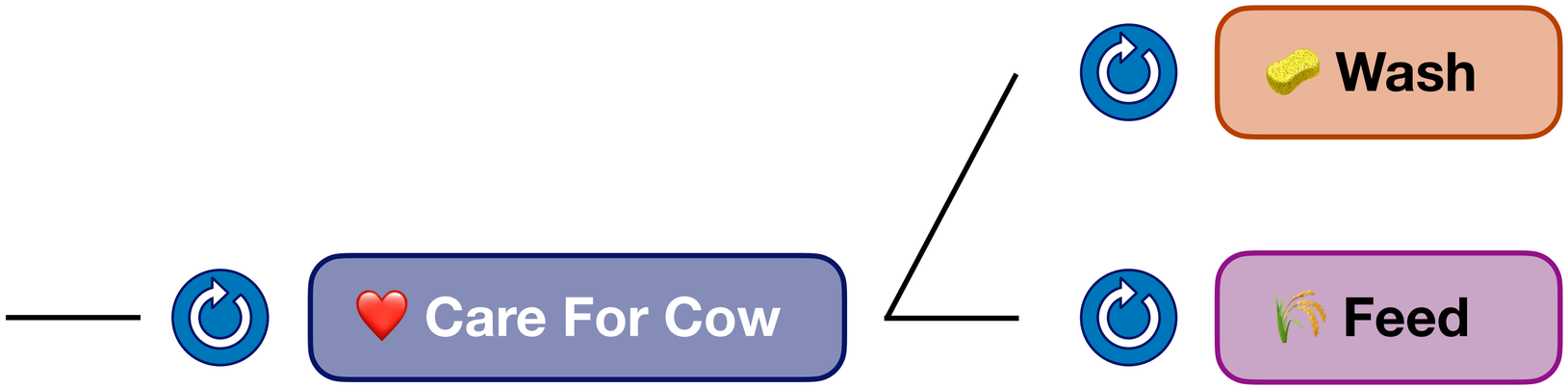
Task Tree

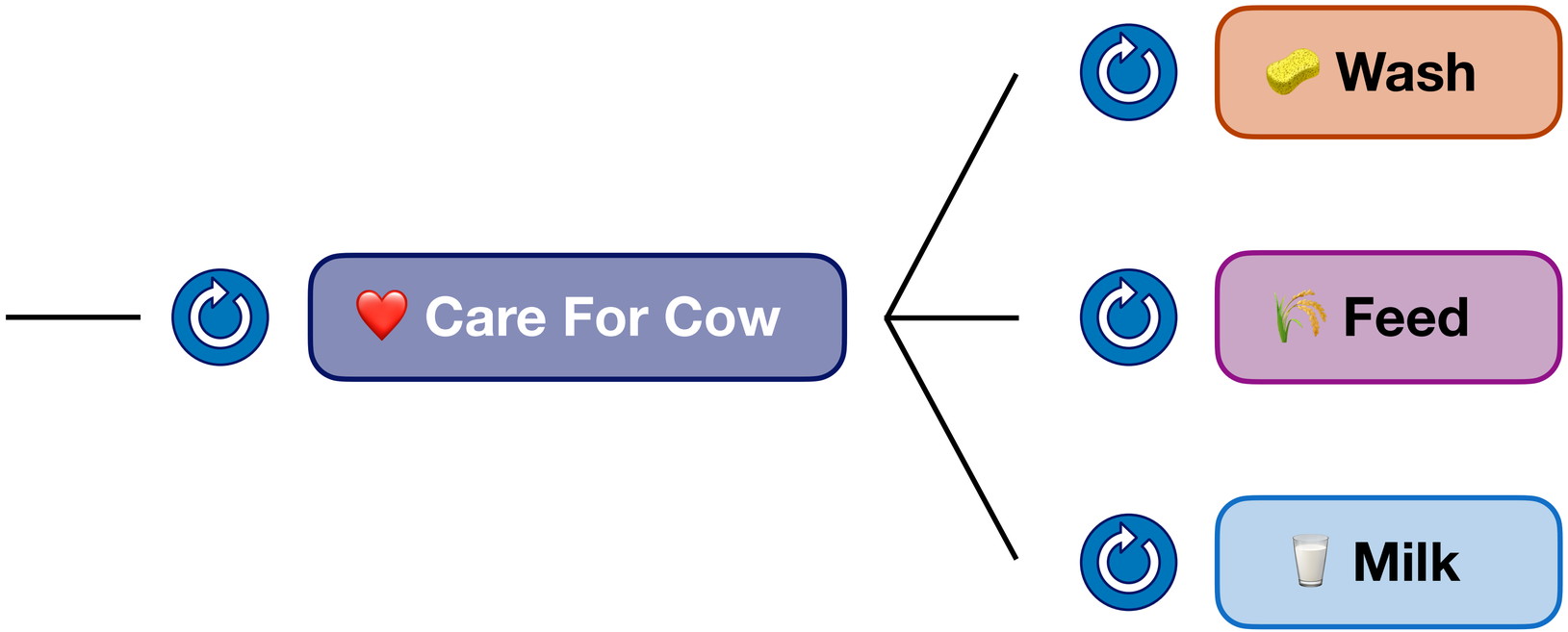
- Tasks can have child tasks
- Child tasks inherit properties of their parent task
- Child tasks cannot outlive their parent task

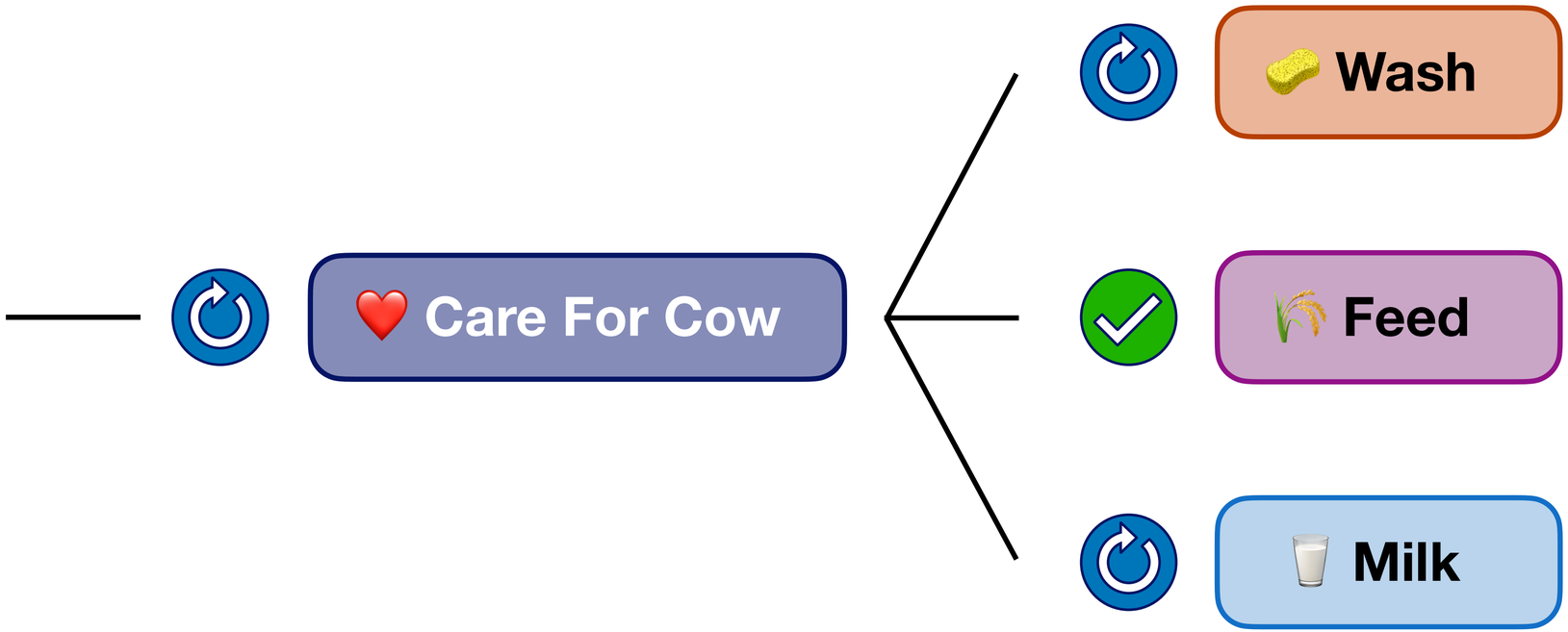
Structured Concurrency > Task > Completion

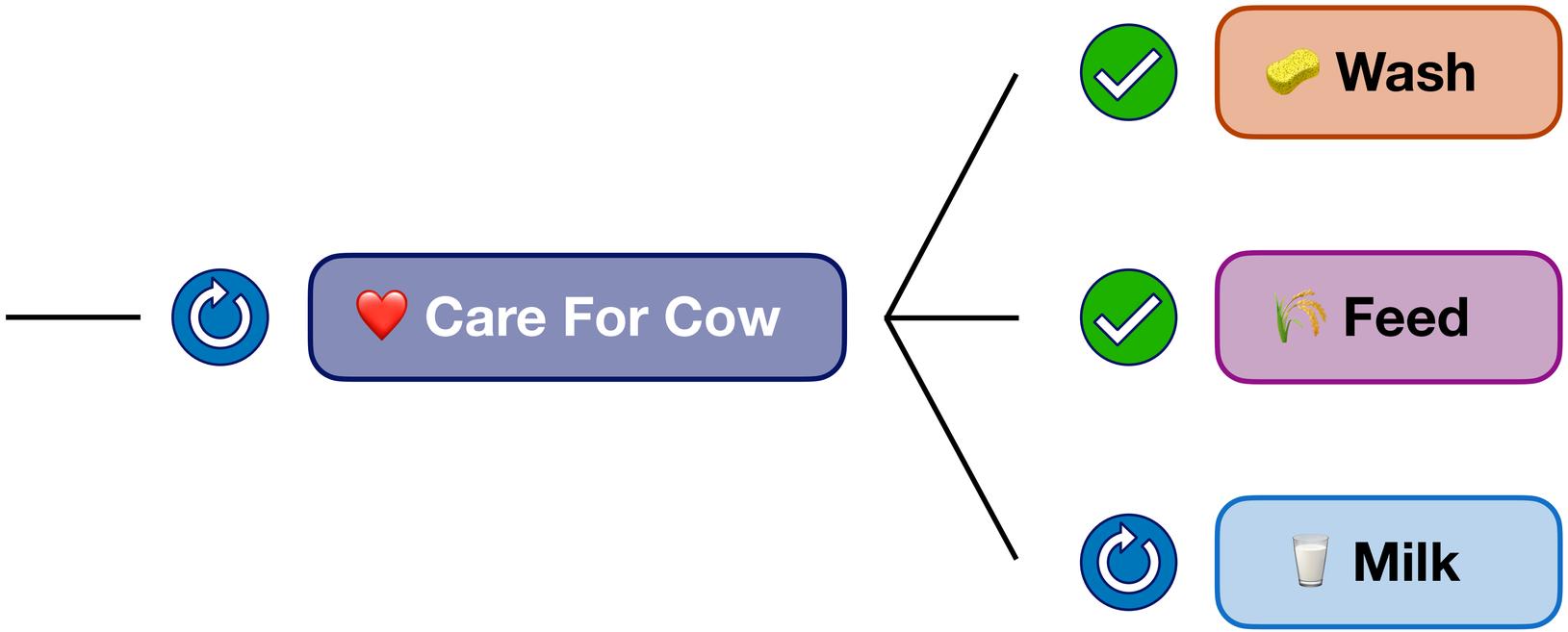

❤️ Care For Cow

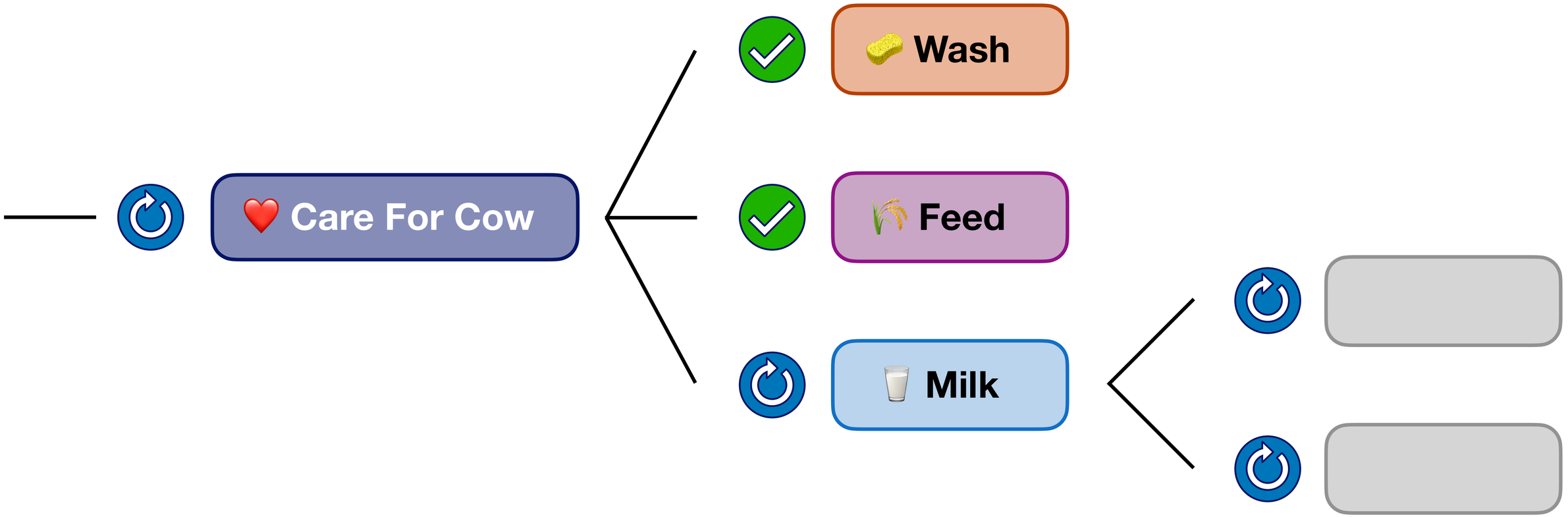

Wash
Care For Cow

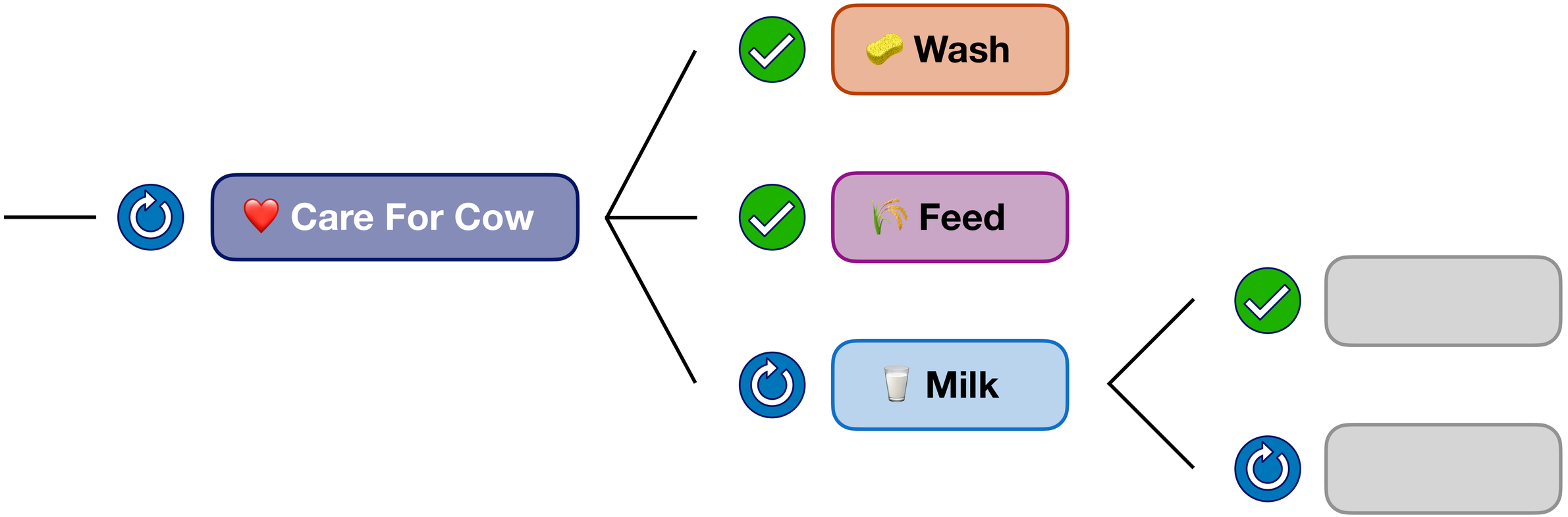

❤️ Care For Cow
🧽 Wash
🌾 Feed

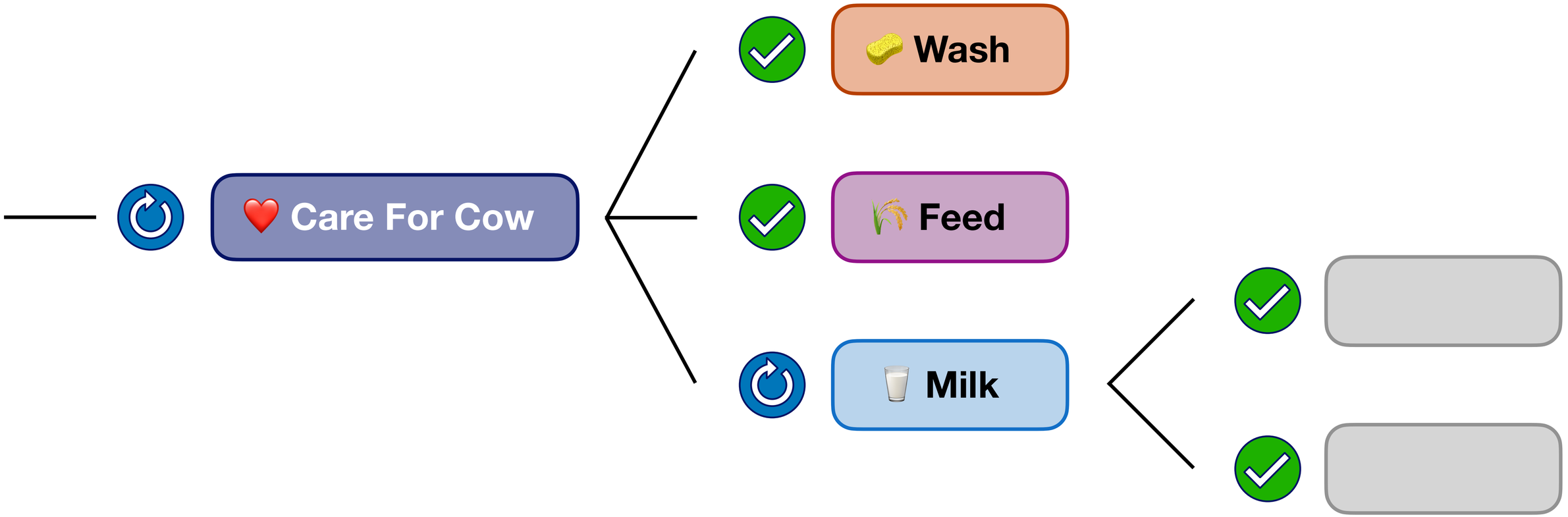

❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk

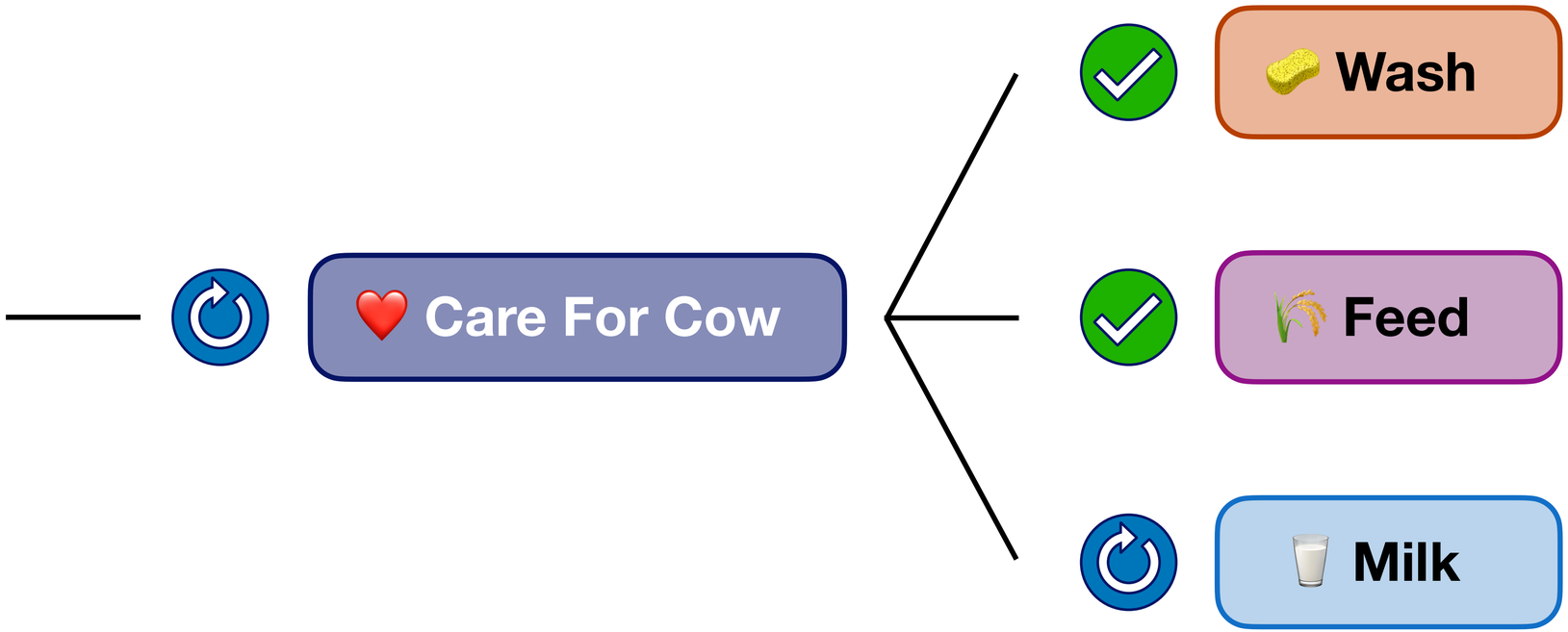

❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk

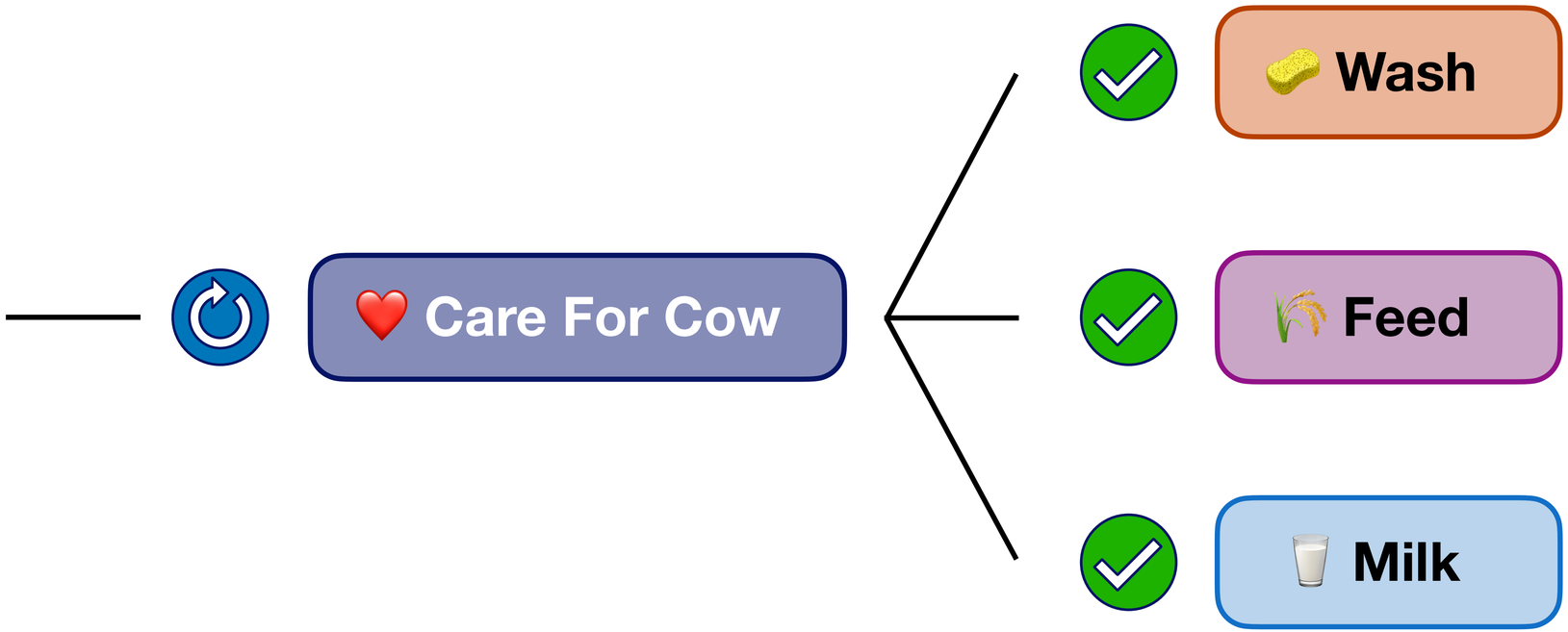

❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk


❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk


❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk


❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk


❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk


❤️ Care For Cow
🧽 Wash
🌾 Feed
🥛 Milk


❤️ Care For Cow


❤️ Care For Cow

Async Let + Task Scoping

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    return try await milk
    // implicit cancellation of `isClean`
    // implicit cancellation of `isFull`
    // implicit await of `isClean`
    // implicit await of `isFull`
}
```

Async Let + Task Scoping

- Async let variable binding

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    return try await milk
    // implicit cancellation of `isClean`
    // implicit cancellation of `isFull`
    // implicit await of `isClean`
    // implicit await of `isFull`
}
```

Async Let + Task Scoping

- Async let variable binding
- Child task lifetime

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    return try await milk
    // implicit cancellation of `isClean`
    // implicit cancellation of `isFull`
    // implicit await of `isClean`
    // implicit await of `isFull`
}
```

Async Let + Task Scoping

- Async let variable binding
- Child task lifetime
- Un-awaited tasks will be implicitly cancelled and awaited

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    return try await milk
    // implicit cancellation of `isClean`
    // implicit cancellation of `isFull`
    // implicit await of `isClean`
    // implicit await of `isFull`
}
```

Async Let + Task Scoping

- Async let variable binding
- Child task lifetime
- Un-awaited tasks will be implicitly cancelled and awaited

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    return try await milk
    // implicit cancellation of `isClean`
    // implicit cancellation of `isFull`
    // implicit await of `isClean`
    // implicit await of `isFull`
}
```

Async Let + Task Scoping

- Async let variable binding
- Child task lifetime
- Un-awaited tasks will be implicitly cancelled and awaited

- Structured Concurrency 👍

```
func careForCow(_ cow: Cow) async throws
-> Milk {
    async let isClean = washCow(cow)
    async let isFull = feedCow(cow)
    async let milk = cow.collectMilk()

    return try await milk
    // implicit cancellation of `isClean`
    // implicit cancellation of `isFull`
    // implicit await of `isClean`
    // implicit await of `isFull`
}
```

Task Cancellation

```
func washCow(_ cow: Cow) async throws -> Bool {
    // checkpoint 1
    try Task.checkCancellation()
    try await rinse(cow)

    // checkpoint 2
    guard !Task.isCancelled else {
        try await dry(cow)
        throw CancellationError()
    }

    // checkpoint 3
    try await scrub(cow)
    try await rinse(cow)
    try await dry(cow)
    return true
}
```

Task Cancellation

- Cancellation propagation

```
func washCow(_ cow: Cow) async throws -> Bool {
    // checkpoint 1
    try Task.checkCancellation()
    try await rinse(cow)

    // checkpoint 2
    guard !Task.isCancelled else {
        try await dry(cow)
        throw CancellationError()
    }

    // checkpoint 3
    try await scrub(cow)
    try await rinse(cow)
    try await dry(cow)
    return true
}
```

Task Cancellation

- Cancellation propagation
- Cooperative Cancellation
  - Check for cancellation
  - Clean up if needed

```
func washCow(_ cow: Cow) async throws -> Bool {
    // checkpoint 1
    try Task.checkCancellation()
    try await rinse(cow)

    // checkpoint 2
    guard !Task.isCancelled else {
        try await dry(cow)
        throw CancellationError()
    }

    // checkpoint 3
    try await scrub(cow)
    try await rinse(cow)
    try await dry(cow)
    return true
}
```

Task Cancellation

- Cancellation propagation
- Cooperative Cancellation
  - Check for cancellation
  - Clean up if needed
- Exiting from cancelled task

```
func washCow(_ cow: Cow) async throws -> Bool {
    // checkpoint 1
    try Task.checkCancellation()
    try await rinse(cow)

    // checkpoint 2
    guard !Task.isCancelled else {
        try await dry(cow)
        throw CancellationError()
    }

    // checkpoint 3
    try await scrub(cow)
    try await rinse(cow)
    try await dry(cow)
    return true
}
```

Task Group

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

Task Group

- Dynamic amount of concurrency

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

Task Group

- Dynamic amount of concurrency
  - Fetching a list of images

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

Task Group

- Dynamic amount of concurrency
  - Fetching a list of images
  - Processing a list of files

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

Task Group

- Dynamic amount of concurrency
  - Fetching a list of images
  - Processing a list of files
  - … list of async ➜ Task Group

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

Task Group

- Dynamic amount of concurrency
  - Fetching a list of images
  - Processing a list of files
  - … list of async ➜ Task Group
- Throwing & non-throwing variant

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

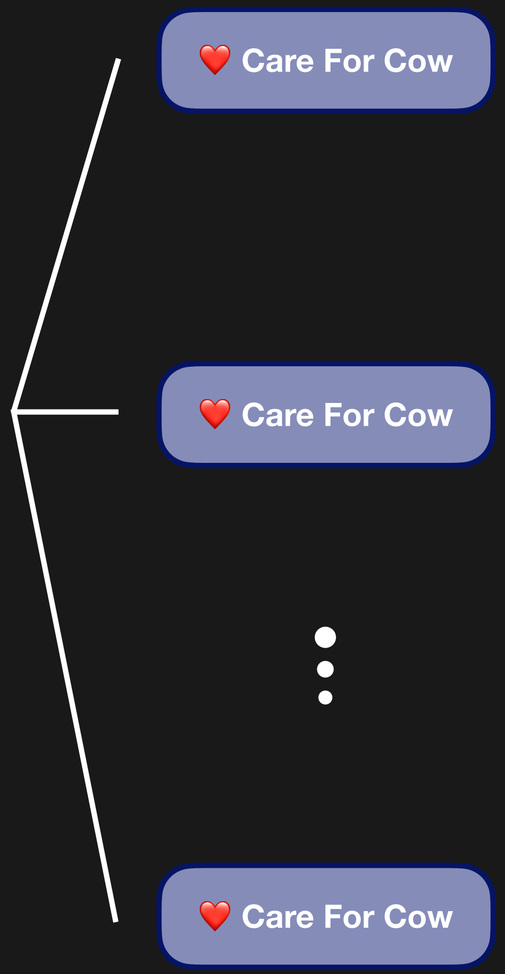
```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

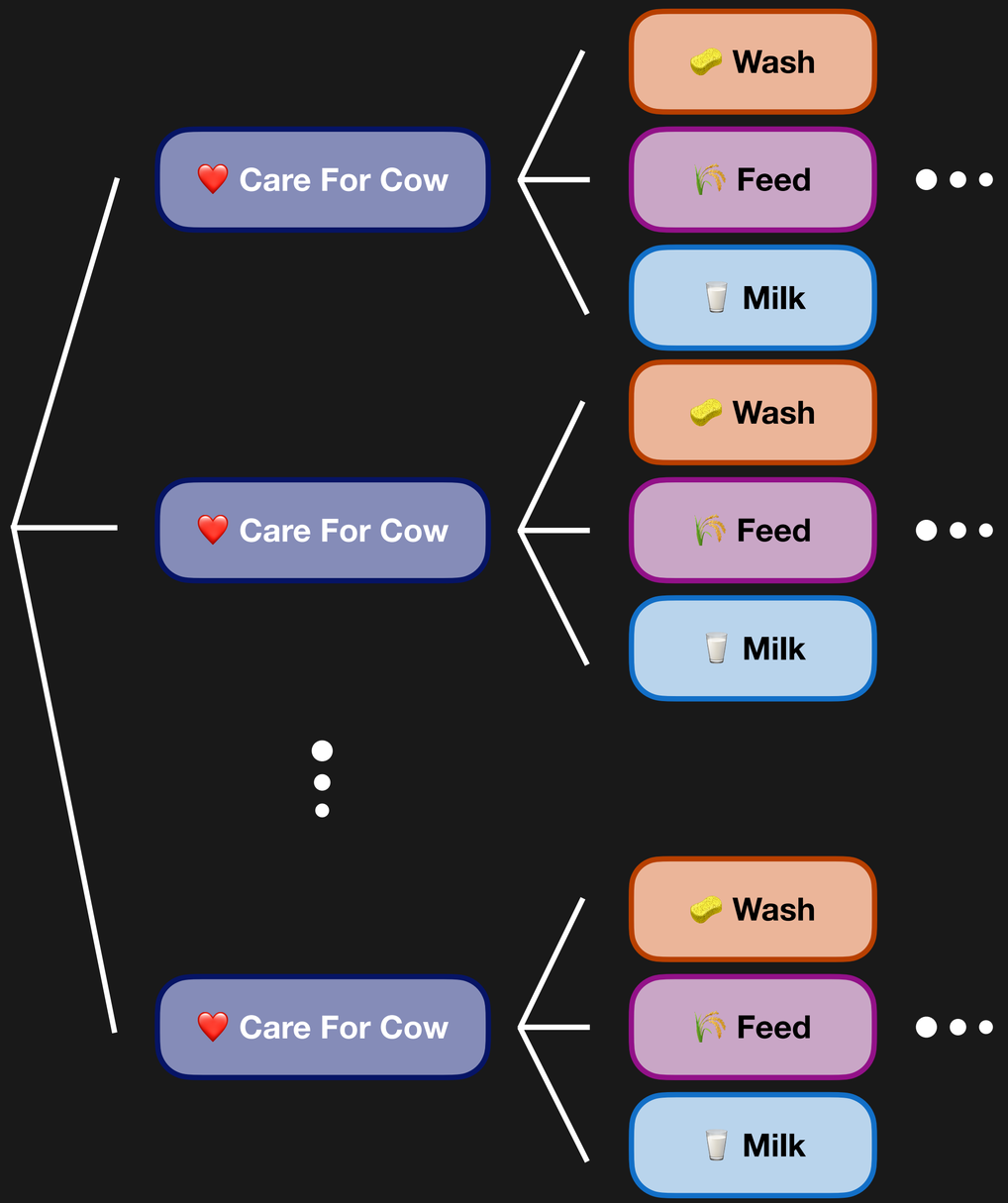
```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```

```
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```



```swift
func careForCows(_ cows: [Cow]) async throws
-> Milk {
    return try await withThrowingTaskGroup(
        of: Milk.self,
        returning: Milk.self) { group in

        for cow in cows {
            group.addTask {
                return try await careForCow(cow)
            }
        }

        var collectedMilk = Milk(gallons: 0)
        for try await milk in group {
            collectedMilk += milk
        }
        return collectedMilk
    }
}
```



Structured Concurrency

Structured Concurrency

- Single task with `async let`

Structured Concurrency

- Single task with `async let`
- List of tasks with `TaskGroup`

Structured Concurrency

- Single task with `async let`
- List of tasks with `TaskGroup`
- Task lifetime and scoping

Unstructured Tasks

Break The Rules!

Unstructured Tasks

Break The Rules!

- Task without a parent

Unstructured Tasks

Break The Rules!

- Task without a parent
- Flexibility & Management

Unstructured Tasks

Break The Rules!

- Task without a parent
- Flexibility & Management
- Inherit properties from the originating context

Unstructured Tasks

Break The Rules!

- Task without a parent
- Flexibility & Management
- Inherit properties from the originating context
- Starting async work from a non-async context

Unstructured Tasks

Break The Rules!

- Task without a parent
- Flexibility & Management
- Inherit properties from the originating context
- Starting async work from a non-async context
- `Task { ... }`

Unstructured Tasks

Break The Rules!

- Task without a parent
- Flexibility & Management
- Inherit properties from the originating context
- Starting async work from a non-async context
- `Task { ... }`
- `Task.Handle`

```
class ViewController: UIViewController {

    var cowManager: CowManager
    var fetchCowsReportHandle: Task.Handle<Void, Never>? = nil

    @objc func fetchButtonPressed() {
        fetchCowsReportHandle = Task {
            await cowManager.fetchReport()
        }
    }

    @objc func cancelButtonPressed() {
        fetchCowsReportHandle?.cancel()
    }
}
```

```
class ViewController: UIViewController {

    var cowManager: CowManager
    var fetchCowsReportHandle: Task.Handle<Void, Never>? = nil

    @objc func fetchButtonPressed() {
        fetchCowsReportHandle = Task {
            await cowManager.fetchReport()
        }
    }

    @objc func cancelButtonPressed() {
        fetchCowsReportHandle?.cancel()
    }
}
```

```
class ViewController: UIViewController {

    var cowManager: CowManager
    var fetchCowsReportHandle: Task.Handle<Void, Never>? = nil

    @objc func fetchButtonPressed() {
        fetchCowsReportHandle = Task {
            await cowManager.fetchReport()
        }
    }

    @objc func cancelButtonPressed() {
        fetchCowsReportHandle?.cancel()
    }
}
```

Detached Tasks

Detached Tasks

- Most control, most overhead

Detached Tasks

- Most control, most overhead
- Nearly identical to unstructured tasks

Detached Tasks

- Most control, most overhead
- Nearly identical to unstructured tasks
  - Do not inherit any properties from originating context

Detached Tasks

- Most control, most overhead
- Nearly identical to unstructured tasks
  - Do not inherit any properties from originating context
- `Task.detached`

Detached Tasks

- Most control, most overhead
- Nearly identical to unstructured tasks
  - Do not inherit any properties from originating context
- `Task.detached`
- analytics, logging, and ephemeral data storage

Shared Mutable State

with Actors

Actors > Data Races

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Data Races

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Data Races

- Concurrent read and write

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Data Races

- Concurrent read and write
- Easy to introduce

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Data Races

- Concurrent read and write
- Easy to introduce
- Hard to track down

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Data Races

- Concurrent read and write
- Easy to introduce
- Hard to track down

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Data Races

- Concurrent read and write
- Easy to introduce
- Hard to track down

Reference to captured var 'herd' in concurrently-executing code

```
class CowHerd {
    var name: String = "untitled herd"

    func fetchCow(_ id: Int) -> Cow { ... }
}

...

var herd = CowHerd()

Task {
    herd.name = "legen-dairy"
    print("task 1, herd name: \(herd.name)")
}

Task {
    herd.name = "moo-riffic"
    print("task 2, herd name: \(herd.name)")
}
```

Actors

New entity that safely provides access to shared mutable state

Actors

New entity that safely provides access to shared mutable state

- Protect access to mutable state

Actors

New entity that safely provides access to shared mutable state

- Protect access to mutable state
- Actor isolation

Actors

New entity that safely provides access to shared mutable state

- Protect access to mutable state
- Actor isolation
- Serialization mechanism

```
actor CowHerd {
    var name: String = "untitled herd"
}

...

var herd = CowHerd()

Task {
    herd.name = "deja moo"
    print("task 1, herd name: \(herd.name)")
}
```

```swift
actor CowHerd {
    var name: String = "untitled herd"
}

...

var herd = CowHerd()

Task {
    herd.name = "deja moo"
    print("task 1, herd name: \(herd.name)")
}
```

```
actor CowHerd {
    var name: String = "untitled herd"
}

...

var herd = CowHerd()

Task {
    herd.name = "deja moo"
    print("task 1, herd name: \(herd.name)")
}
```

Actor-isolated property ‘name' can not be mutated from a non-isolated context

```
actor CowHerd {
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

...

var herd = CowHerd()

Task {
    herd.setName("deja moo")
    print("task 1, herd name: \(herd.name)")
}
```

```
actor CowHerd {
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

...

var herd = CowHerd()

Task {
    herd.setName("deja moo")
    print("task 1, herd name: \(herd.name)")
}
```

```
actor CowHerd {
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

...

var herd = CowHerd()

Task {
    herd.setName("deja moo")
    print("task 1, herd name: \(herd.name)")
}
```

Expression is 'async' but is not marked with 'await'

```
actor CowHerd {
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

...

var herd = CowHerd()

Task {
    await herd.setName("deja moo")
    print("task 1, herd name: \(await herd.name)")
}
```

```
actor CowHerd {
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

...

var herd = CowHerd()

Task {
    await herd.setName("deja moo")
    print("task 1, herd name: \(await herd.name)")
}
```

✅

Actor

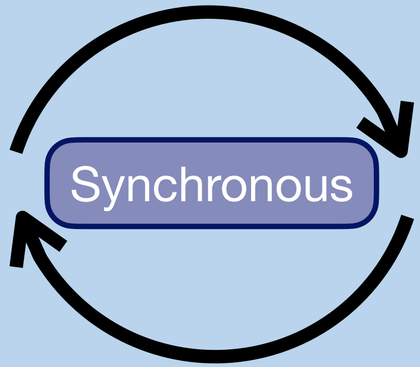
Isolated

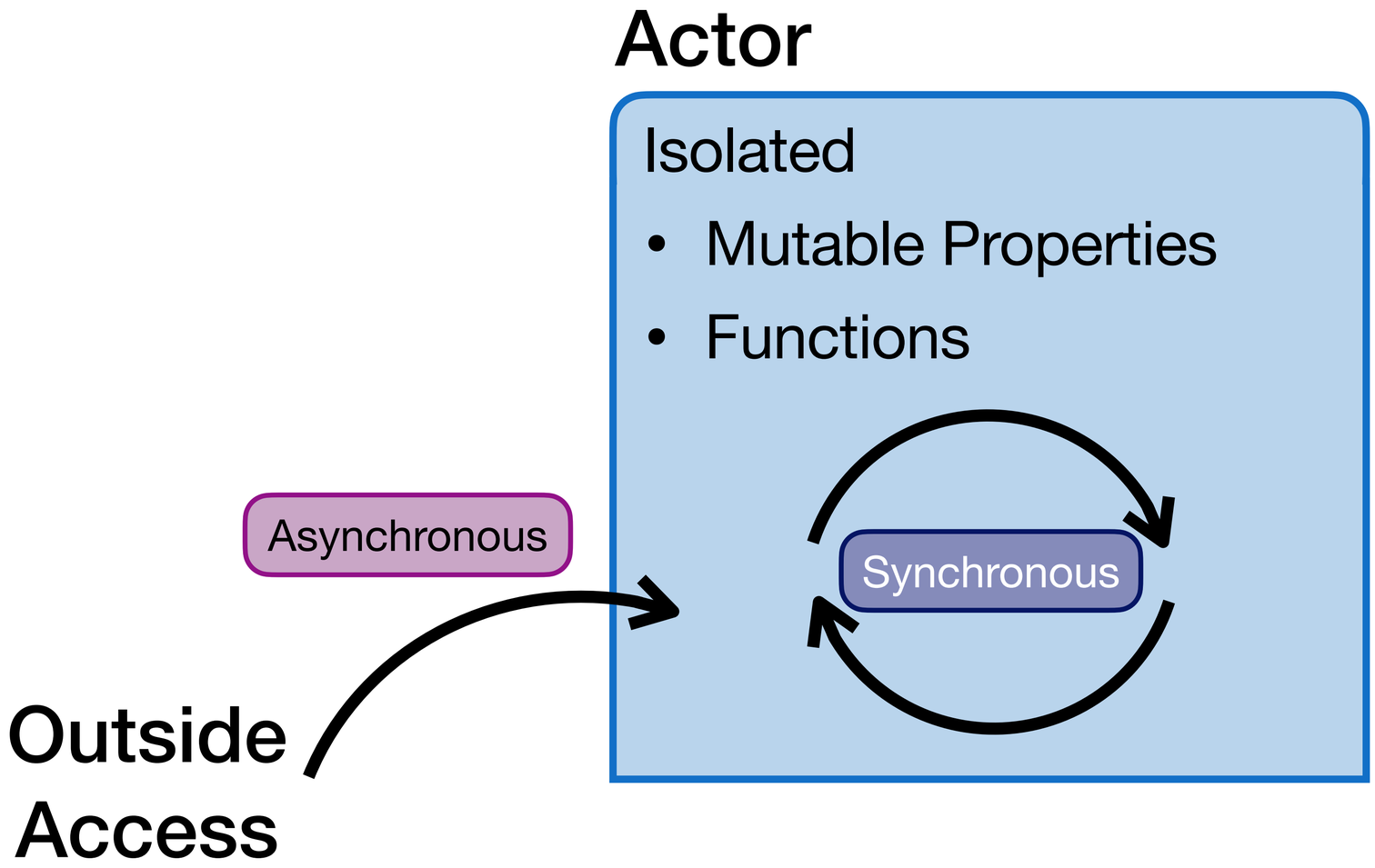
- Mutable Properties
- Functions

Actor

Isolated

- Mutable Properties
- Functions



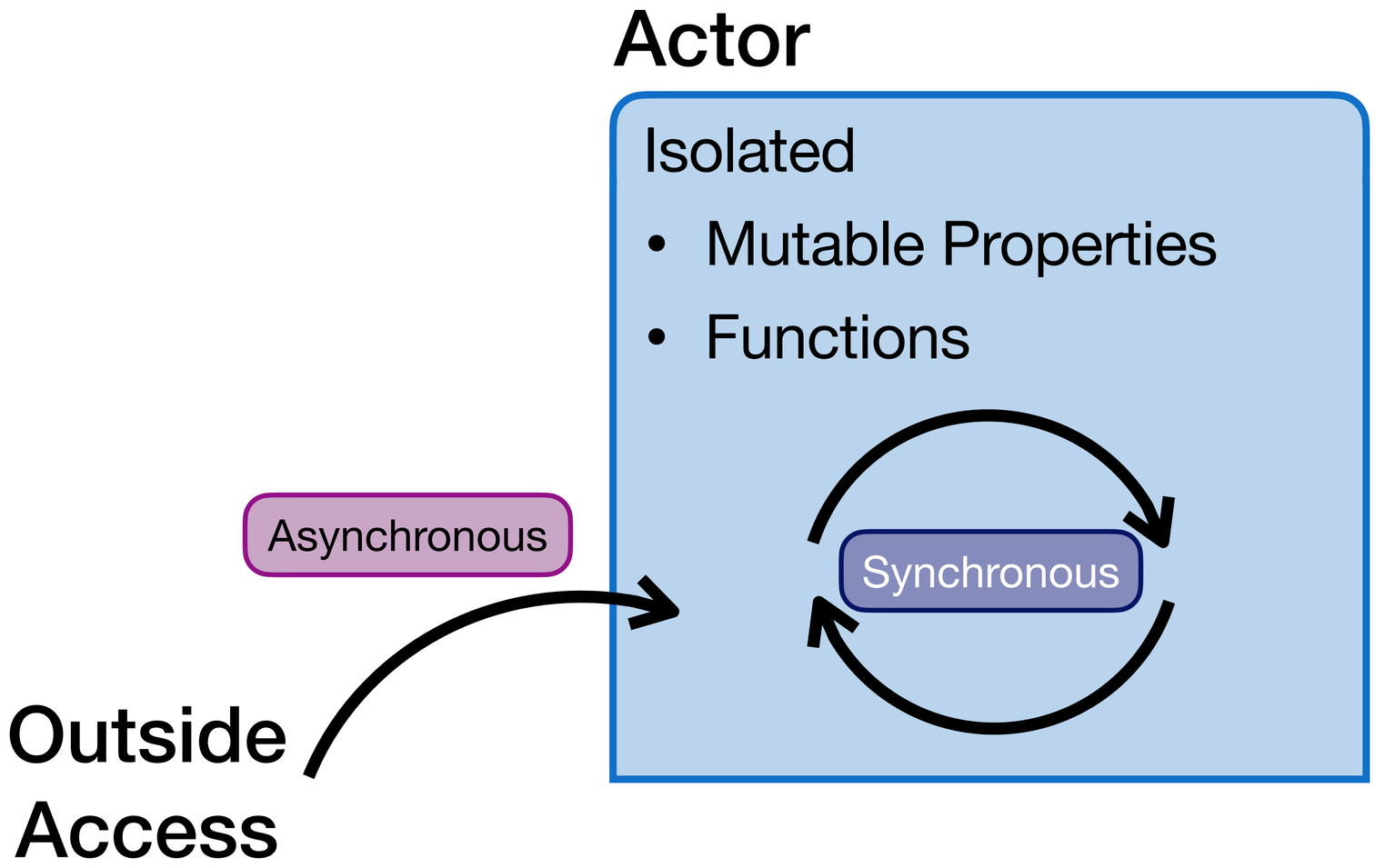

Actor
Isolated
• Mutable Properties
• Functions
Synchronous
Asynchronous
Outside
Access

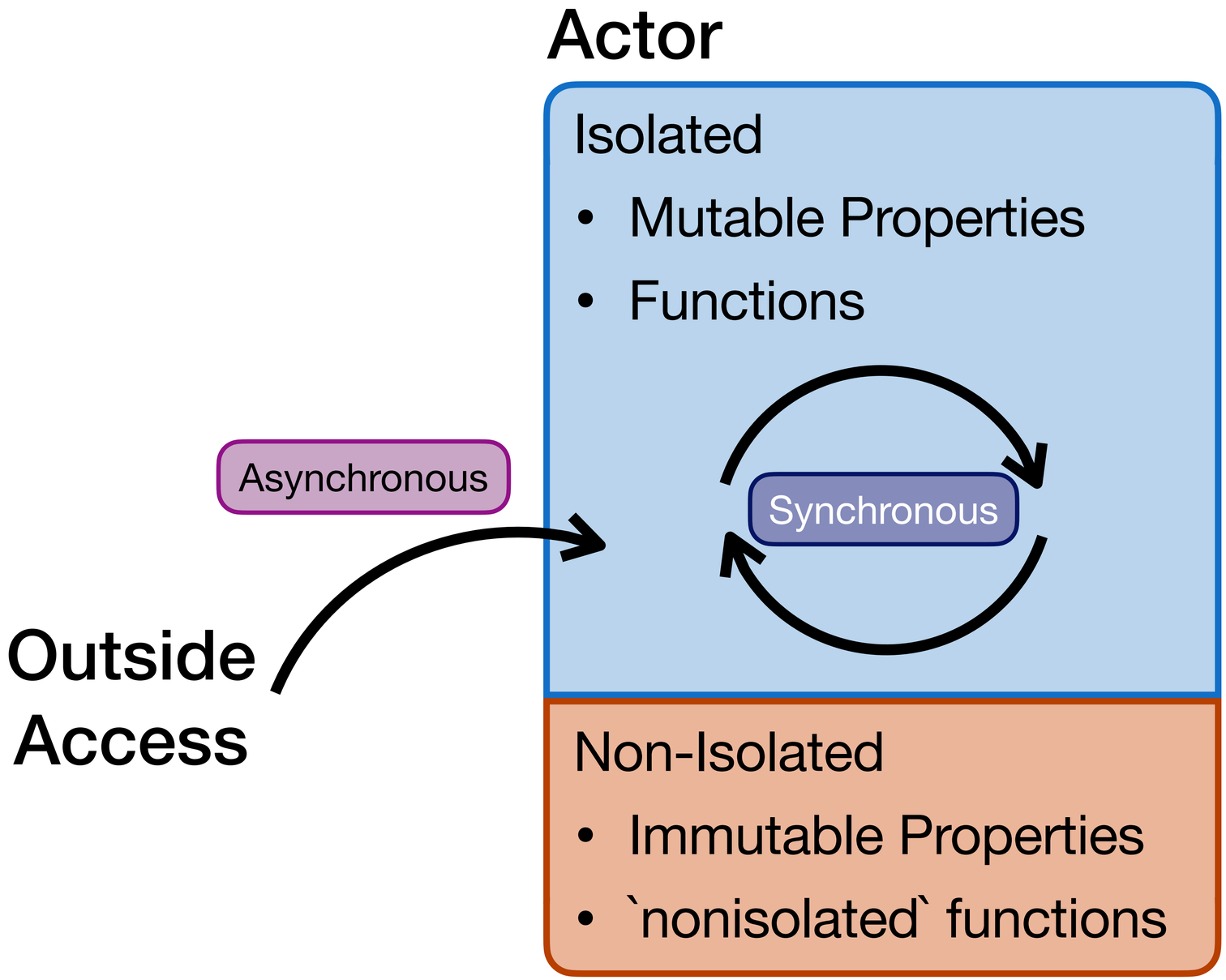
```
actor CowHerd {
    let animalType = Cow.self
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

extension CowHerd {
    nonisolated func whatDoesTheCowSay() {
        print("MOOOOO")
    }
}

...

var herd = CowHerd()

Task {
    print(store.animalType) // prints "Cow"
    herd.whatDoesTheCowSay() // prints "MOOOOO"
}
```

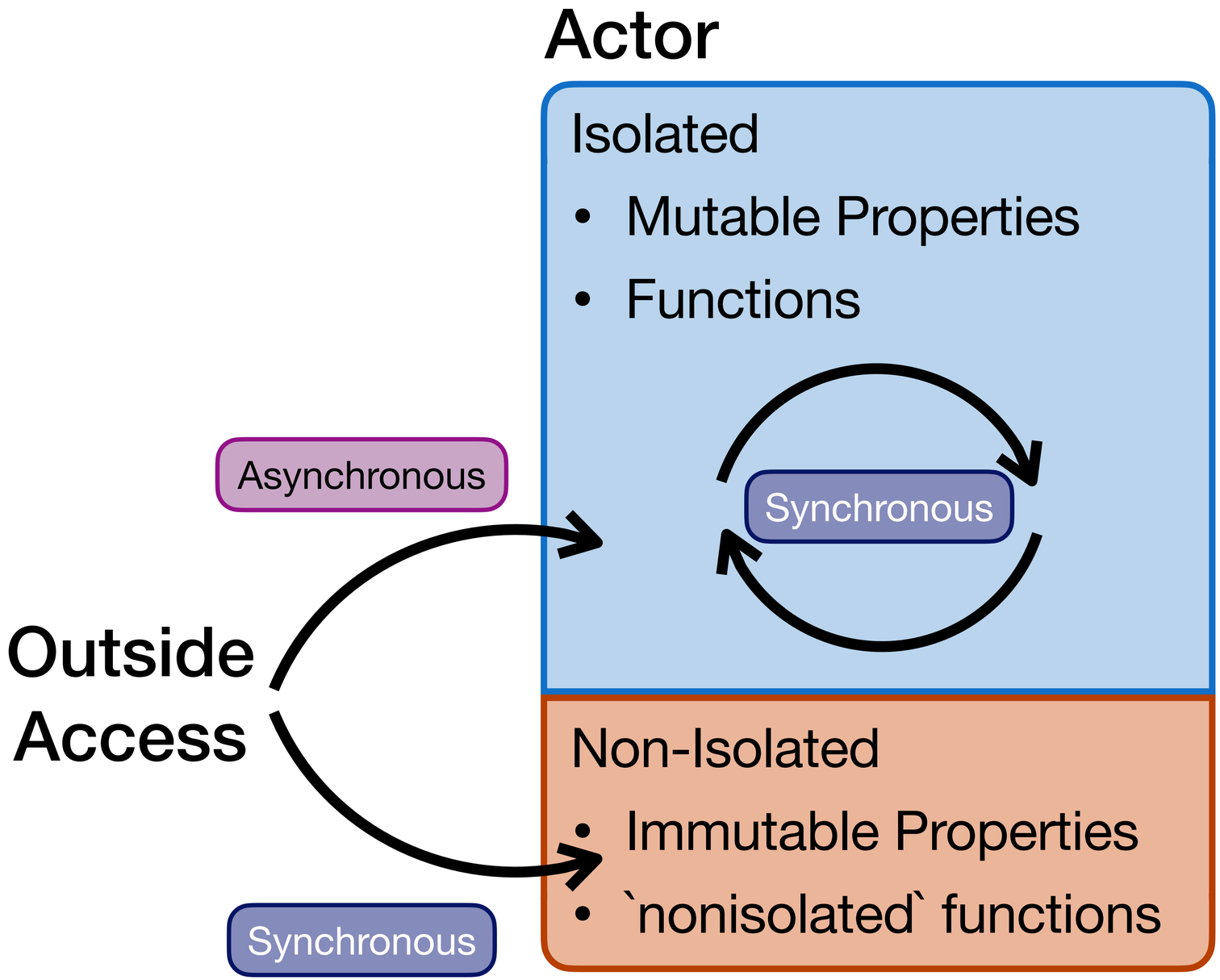
```swift
actor CowHerd {
    let animalType = Cow.self
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

extension CowHerd {
    nonisolated func whatDoesTheCowSay() {
        print("MOOOOO")
    }
}

...

var herd = CowHerd()

Task {
    print(store.animalType) // prints "Cow"
    herd.whatDoesTheCowSay() // prints "MOOOOO"
}
```

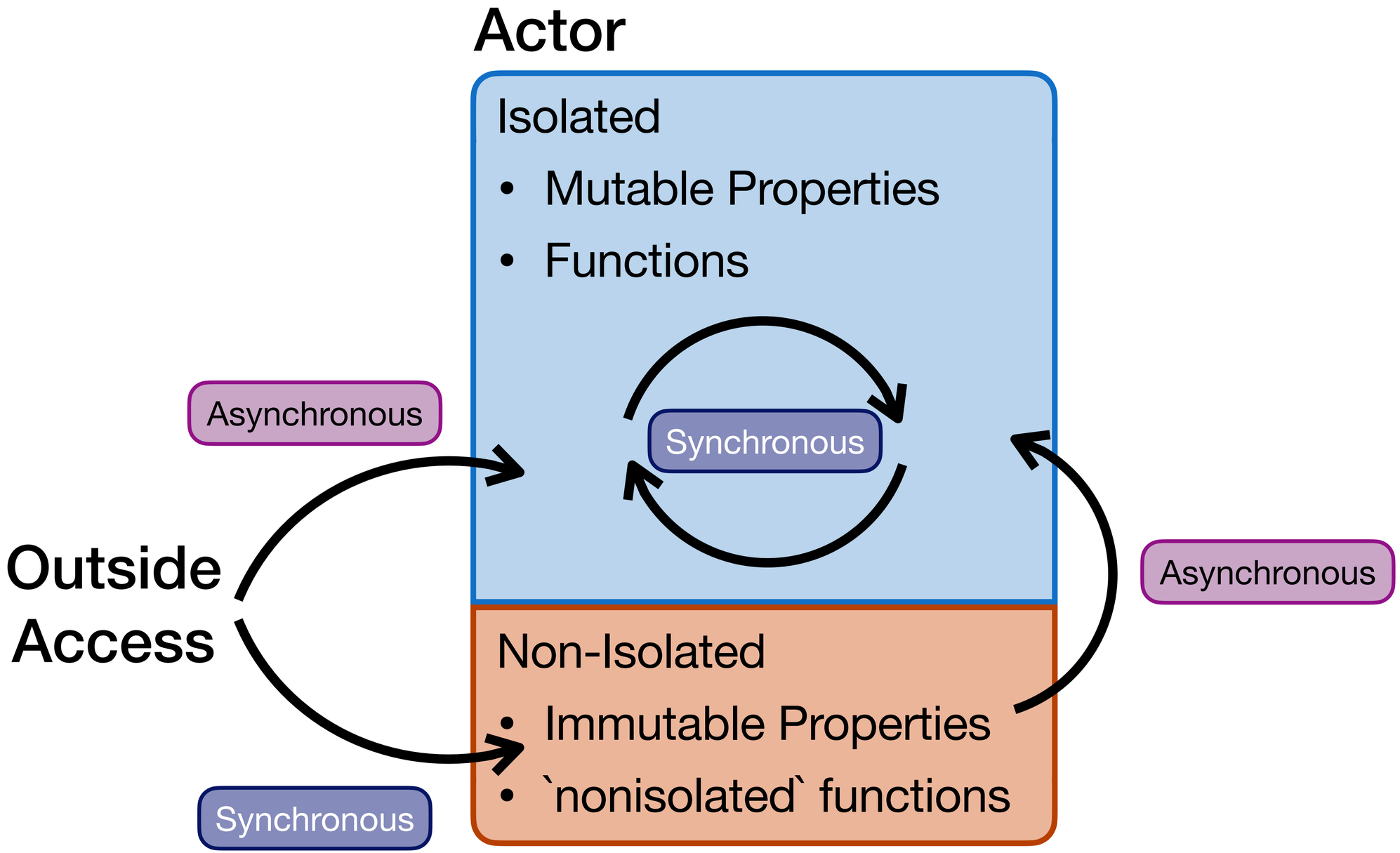
```
actor CowHerd {
    let animalType = Cow.self
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

extension CowHerd {
    nonisolated func whatDoesTheCowSay() {
        print("MOOOOO")
    }
}

...

var herd = CowHerd()

Task {
    print(store.animalType) // prints "Cow"
    herd.whatDoesTheCowSay() // prints "MOOOOO"
}
```

```
actor CowHerd {
    let animalType = Cow.self
    var name: String = "untitled herd"

    func setName(_ name: String) {
        self.name = name
    }
}

extension CowHerd {
    nonisolated func whatDoesTheCowSay() {
        print("MOOOOO")
    }
}

...

var herd = CowHerd()

Task {
    print(store.animalType) // prints "Cow"
    herd.whatDoesTheCowSay() // prints "MOOOOO"
}
```


Actor
Isolated
• Mutable Properties
• Functions
Synchronous
Asynchronous
Outside
Access


Actor
Isolated
• Mutable Properties
• Functions
Synchronous
Asynchronous
Outside Access
Non-Isolated
• Immutable Properties
• `nonisolated` functions


Actor
Isolated
• Mutable Properties
• Functions
Synchronous
Asynchronous
Outside Access
Synchronous
Non-Isolated
• Immutable Properties
• `nonisolated` functions


Actor
Isolated
• Mutable Properties
• Functions
Synchronous
Asynchronous
Outside Access
Non-Isolated
• Immutable Properties
• `nonisolated` functions
Asynchronous
Synchronous

Actor Continued…

Actor Continued…

- Threading

Actor Continued…

- Threading
  - Implicit background threading with `await`

Actor Continued…

- Threading
  - Implicit background threading with `await`
    - MainActor

Actor Continued…

- Threading
  - Implicit background threading with `await`
    - MainActor
  - System controlled thread scheduling

Actor Continued…

- Threading
  - Implicit background threading with `await`
    - MainActor
  - System controlled thread scheduling
- Create multiple instances of an actor

Actor Continued…

- Threading
  - Implicit background threading with `await`
    - MainActor
  - System controlled thread scheduling
- Create multiple instances of an actor
- Logical controllers, managers, and data stores.

MainActor

```
@MainActor
class View: UIView {
    ...

    func updateName(_ name: String) {
        ...
    }
}

...

Task {
    let name = await herd.name
    view.updateName(name)
    await view.updateName(name)
}
```

MainActor

- Concrete actor implementation

```
@MainActor
class View: UIView {
    ...

    func updateName(_ name: String) {
        ...
    }
}

...

Task {
    let name = await herd.name
    view.updateName(name)
    await view.updateName(name)
}
```

MainActor

- Concrete actor implementation
- Represents the main thread

```
@MainActor
class View: UIView {
    ...

    func updateName(_ name: String) {
        ...
    }
}

...

Task {
    let name = await herd.name
    view.updateName(name)
    await view.updateName(name)
}
```

MainActor

- Concrete actor implementation
- Represents the main thread
- `@MainActor`

```
@MainActor
class View: UIView {
    ...

    func updateName(_ name: String) {
        ...
    }
}

...

Task {
    let name = await herd.name
    view.updateName(name)
    await view.updateName(name)
}
```

MainActor

- Concrete actor implementation
- Represents the main thread
- `@MainActor`

```
@MainActor
class View: UIView {
    ...

    func updateName(_ name: String) {
        ...
    }
}

...

Task {
    let name = await herd.name
    view.updateName(name) 🛑
    await view.updateName(name) ✅
}
```

MainActor

- Concrete actor implementation
- Represents the main thread
- `@MainActor`

`DispatchQueue.main.async` 👋

```
@MainActor
class View: UIView {
    ...

    func updateName(_ name: String) {
        ...
    }
}

...

Task {
    let name = await herd.name
    view.updateName(name) 🛑
    await view.updateName(name) ✅
}
```

Key Points

Key Points

Key Points

- **Ergonomics++**
  - Function suspension
  - Error handling

Key Points

- **Ergonomics++**
  - Function suspension
  - Error handling
- **Compile time checks**
  - Async requires await
  - Actors and the MainActor

Key Points

- **Ergonomics++**
  - Function suspension
  - Error handling
- **Compile time checks**
  - Async requires await
  - Actors and the MainActor
- **Local reasoning & scoping**
  - Easily visualize concurrent processing
  - Structured concurrency scoping

Recommendations

Recommendations

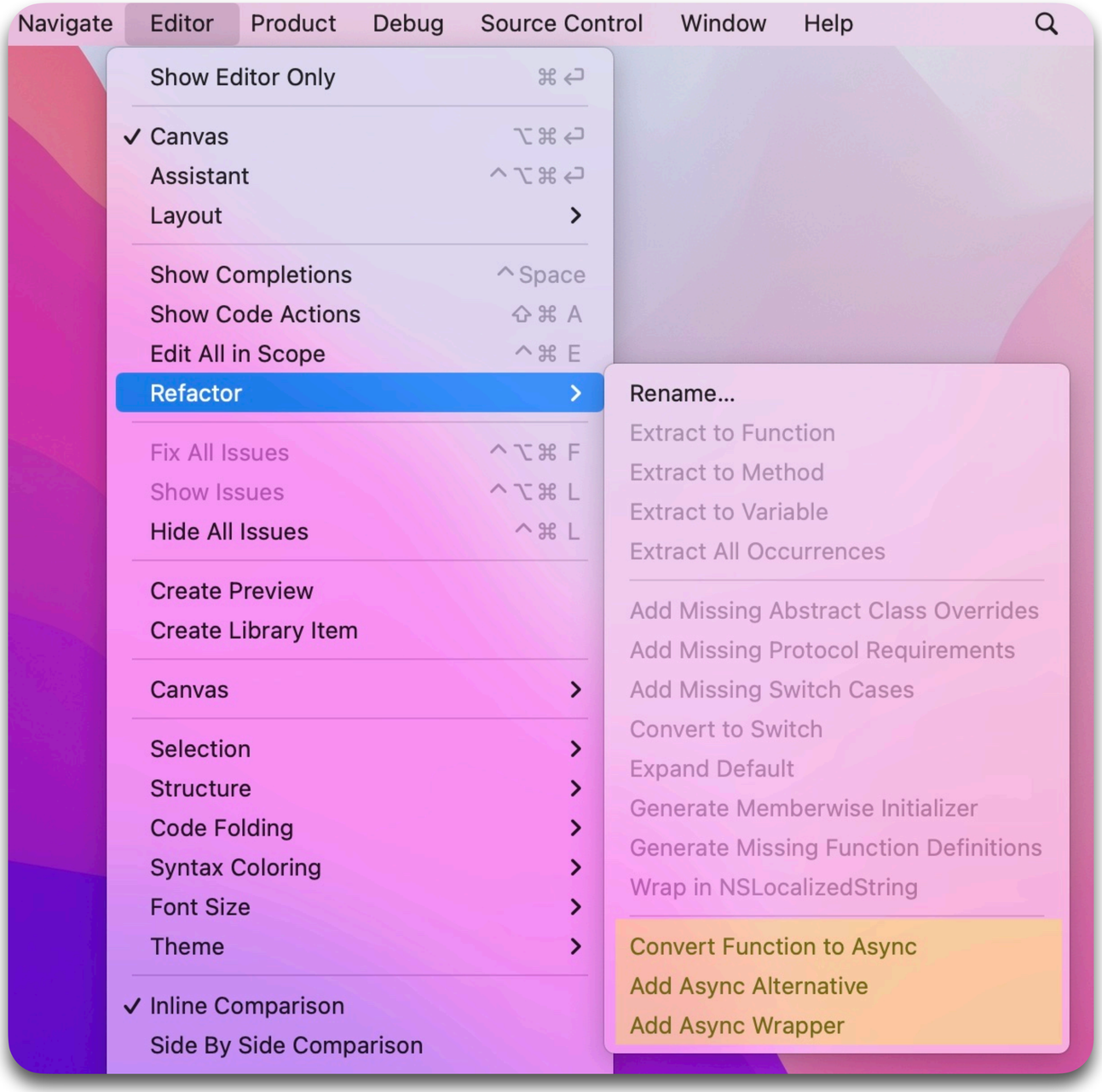
- Use `try`/`catch` error handling

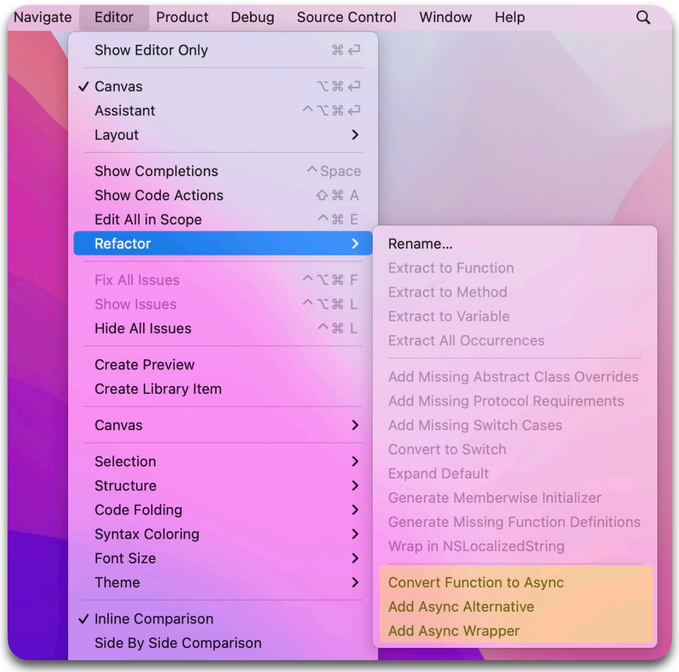
Recommendations

- Use `try`/`catch` error handling
- Convert modules at a time

Recommendations

- Use `try`/`catch` error handling
- Convert modules at a time
- Create `async` shims for your APIs using Continuations
  - Utilize Xcode’s refactor tools


Navigate
Editor
Product
Debug
Source Control
Window
Help
Show Editor Only ⌘↩
Canvas ⌥⌘↩
Assistant ^⌥⌘↩
Layout
Show Completions ^Space
Show Code Actions ⇧⌘A
Edit All in Scope ^⌘E
Refactor
Fix All Issues ^⌥⌘F
Show Issues ^⌥⌘L
Hide All Issues ^⌘L
Create Preview
Create Library Item
Canvas
Selection
Structure
Code Folding
Syntax Coloring
Font Size
Theme
Inline Comparison
Side By Side Comparison
Rename...
Extract to Function
Extract to Method
Extract to Variable
Extract All Occurrences
Add Missing Abstract Class Overrides
Add Missing Protocol Requirements
Add Missing Switch Cases
Convert to Switch
Expand Default
Generate Memberwise Initializer
Generate Missing Function Definitions
Wrap in NSLocalizedString
Convert Function to Async
Add Async Alternative
Add Async Wrapper




```
func fetchCow(completion: (Result<Cow, Error>) -> Void) {
    // does asynchronous network fetch
}

func fetchCow() async throws -> Cow {
    return try await withCheckedThrowingContinuation {
        continuation in
        fetchCow() { result in
            continuation.resume(with: result)
        }
    }
}
```

Recommendations Cont…

Recommendations Cont…

- Using thread locks

Recommendations Cont…

- Using thread locks
- Marking everything with `async`

Recommendations Cont…

- Using thread locks
- Marking everything with `async`
- Progress Updates