KU LEUVEN

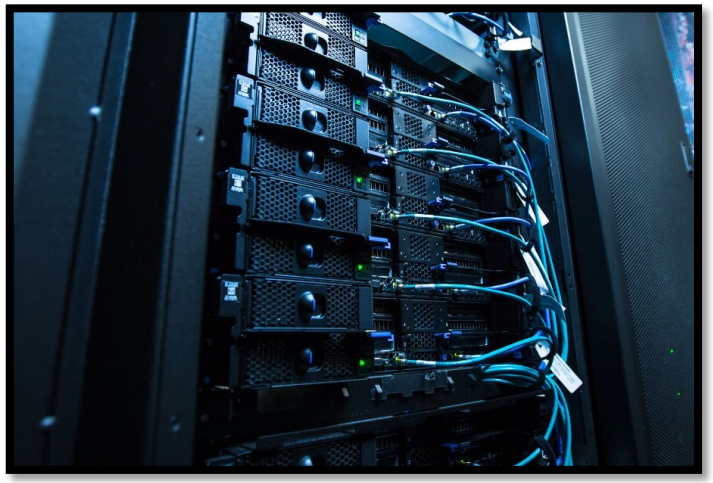
How Rust boosts confidence and productivity

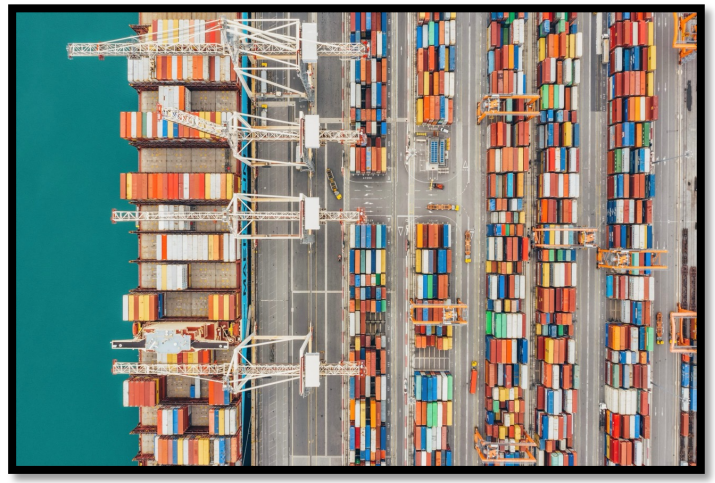
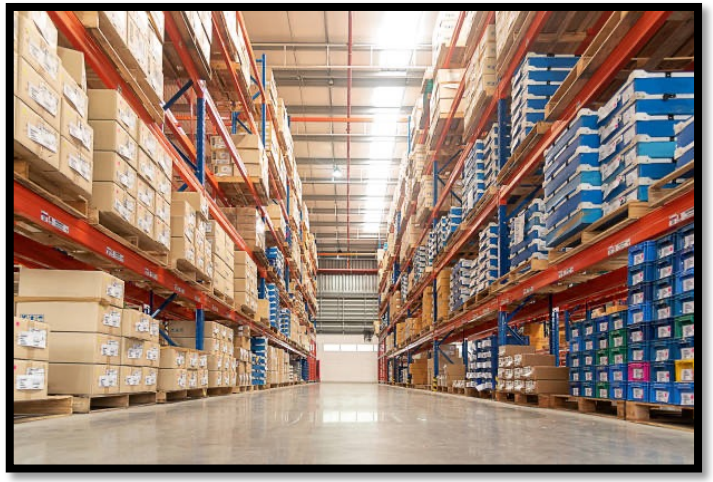
my experiences as an algorithm researcher

Jeroen Gardeyn
PhD student - CODeS

Background

Combinatorial Optimization




Combinatorial Optimization

- Combinatorial explosion
- Algorithms to *search* good solutions

- Lots of moving parts, complex datastructures
- **High-level, high-performance** programming

n	The number of Latin squares of order n
1	1
2	2
3	12
4	576
5	161,280
6	812,851,200
7	61,479,419,904,000
8	108,776,032,459,082,956,800
9	5,524,751,496,156,892,842,531,225,600
10	9,982,437,658,213,039,871,725,064,756,920,320,000
11	776,966,836,171,770,144,107,444,346,734,230,682,311,065,600,000

`https://en.wikipedia.org/wiki/combinatorial_explosion`

Rust for *algorithmic* programming

Types of *bugs*

Types of *bugs*

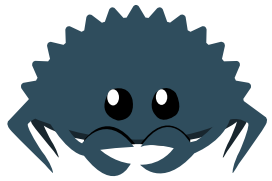
Compile-time Errors

Runtime Crashes

Incorrect Execution

Silent Incorrect Execution

difficulty of detection



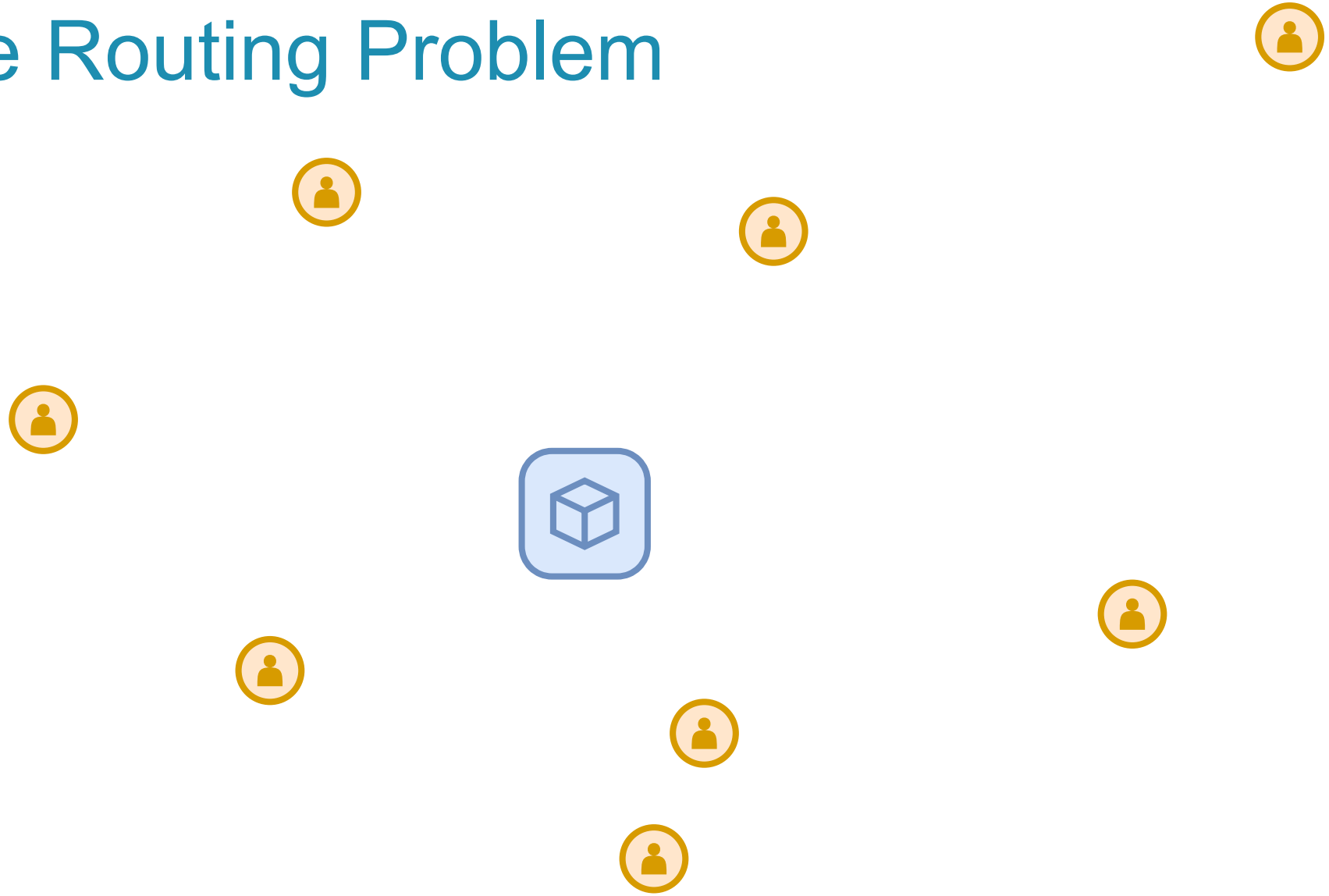
Compile-time Errors

Rust for high-level programming

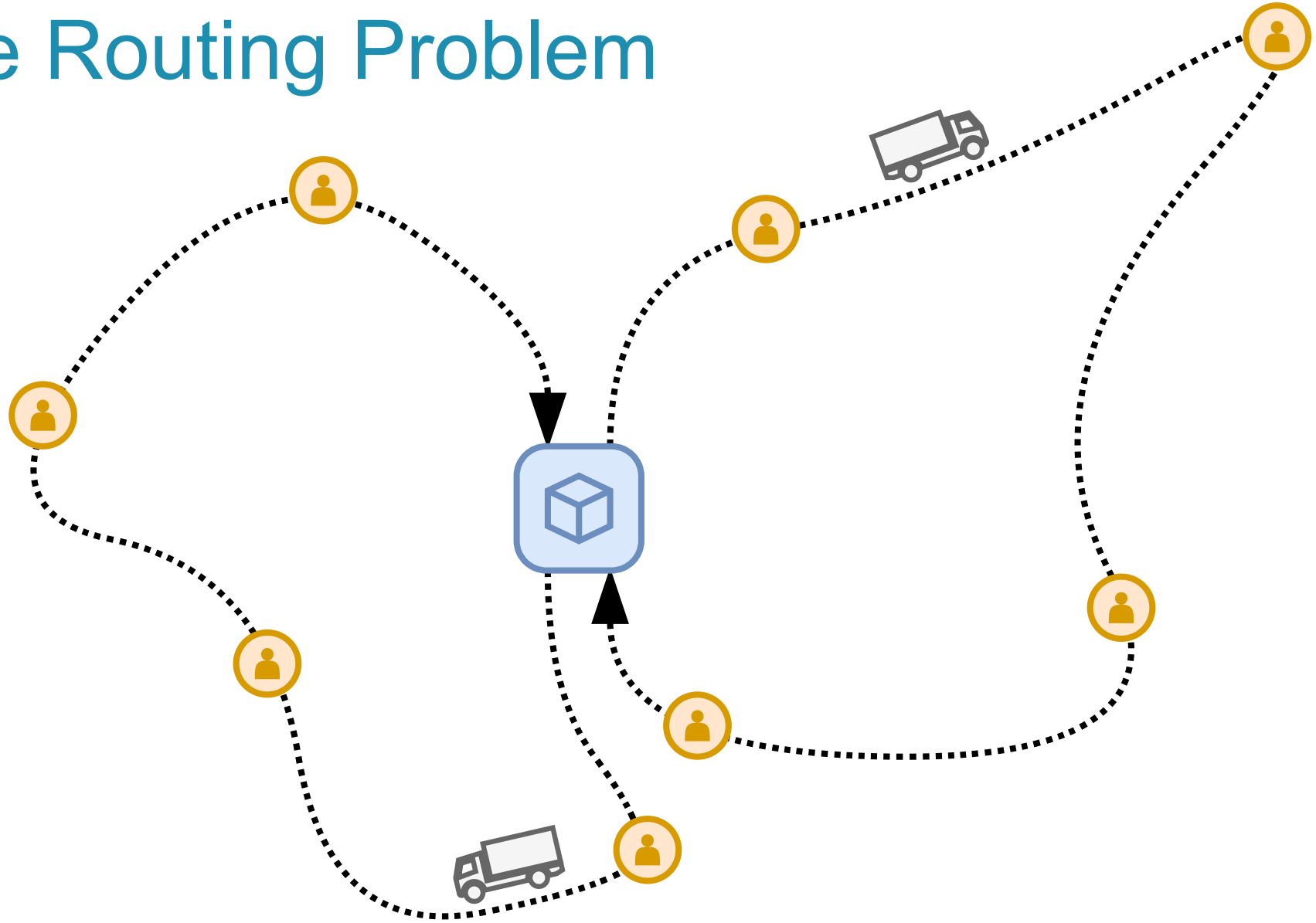
- Correctness/robustness → ⇧ confidence & ⇧ productivity
- Speed & memory safety are just the 🍒 on top

- Prime choice for writing complex algorithms, even if:
  - GC had no overhead
  - C/C++ did not suffer from memory safety issues

Vehicle Routing Problem



Vehicle Routing Problem



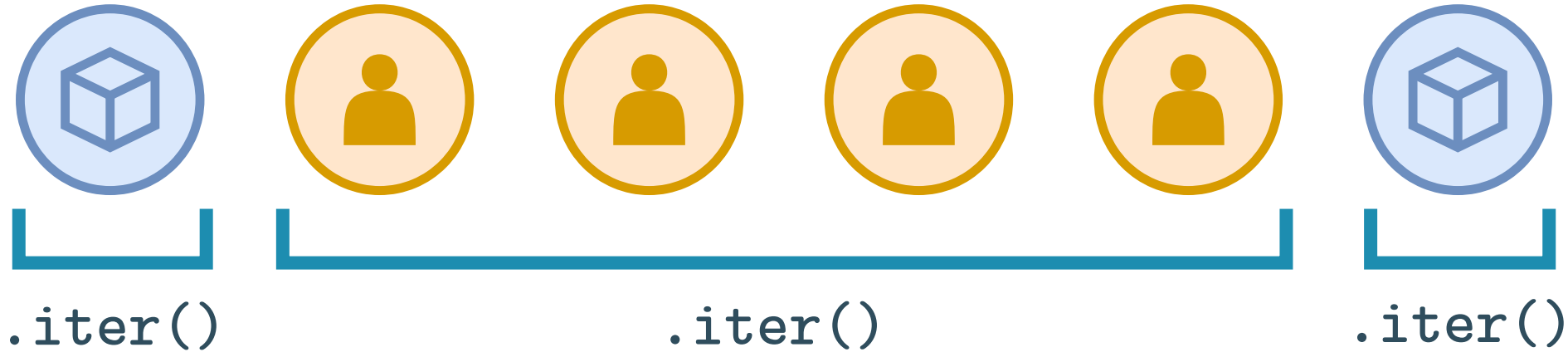
Case 1: The compiler is your friend

Case 1: The compiler is your friend

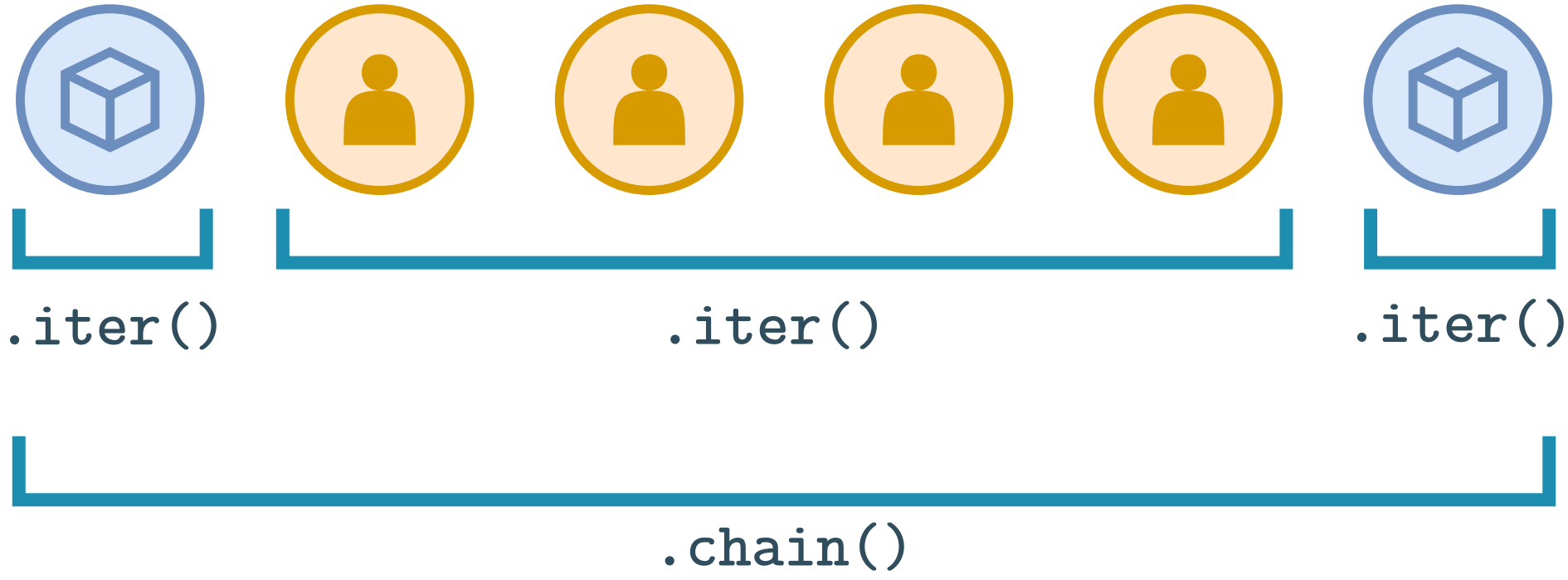
- Move potential errors as much as possible to compile time:
  - Make invalid states unrepresentable
  - Sum types (enums)
  - Pattern matching (exhaustive)
  - Destructuring
  - Rust has no null and no exceptions (`Option<T>` & `Result<T>`)

Case 2: Functional programming

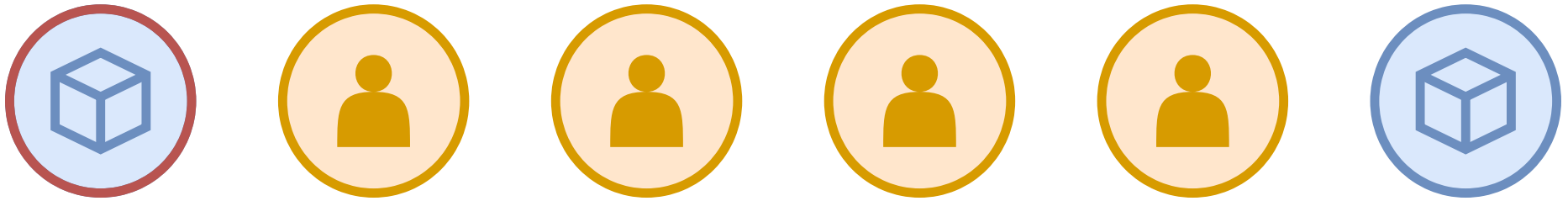
Case 2: Functional programming



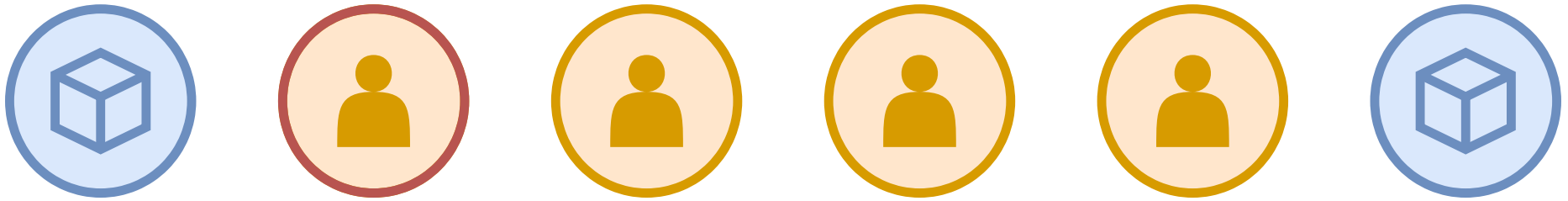
Case 2: Functional programming



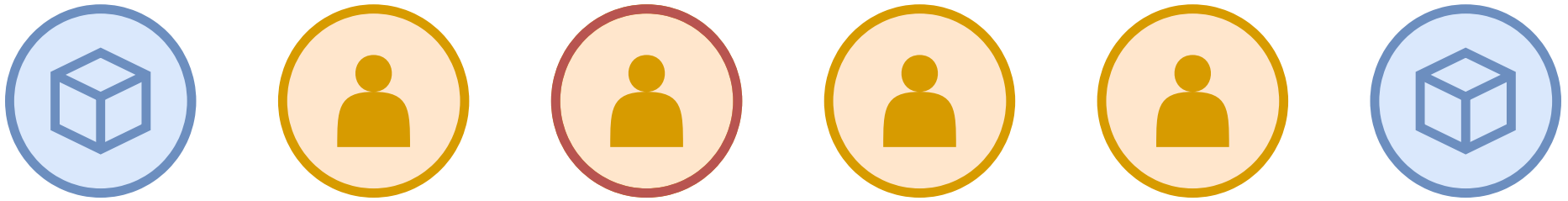
Case 2: Functional programming



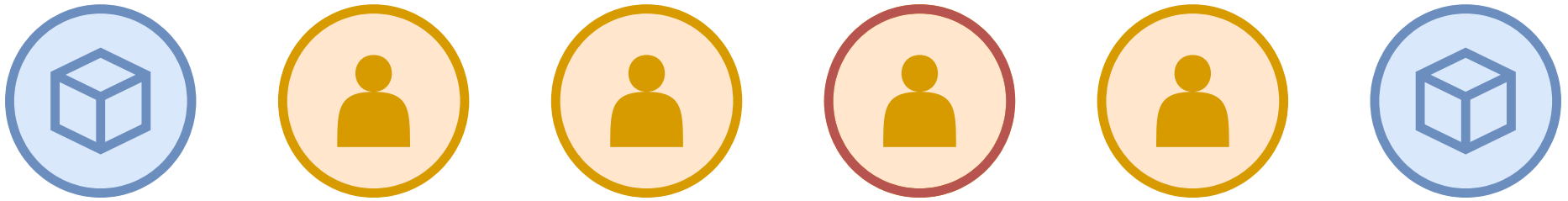
Case 2: Functional programming



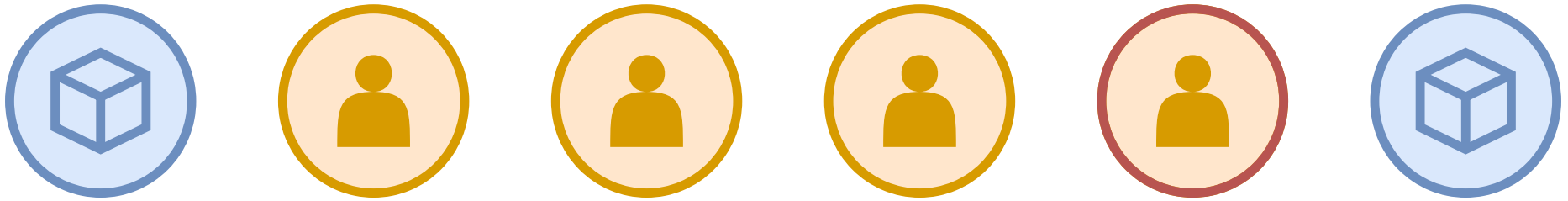
Case 2: Functional programming



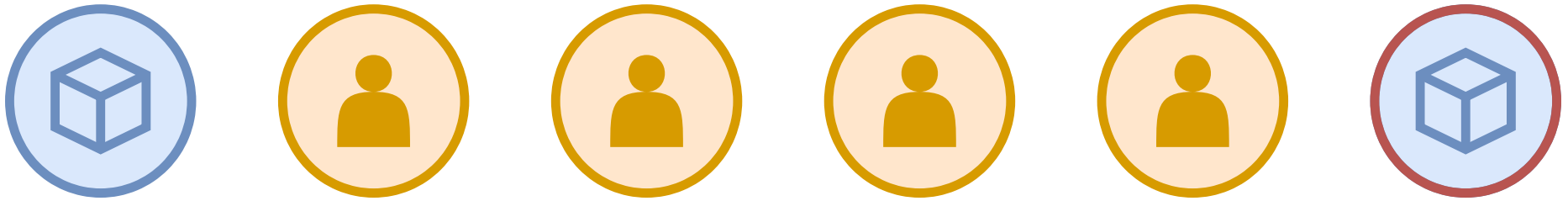
Case 2: Functional programming



Case 2: Functional programming

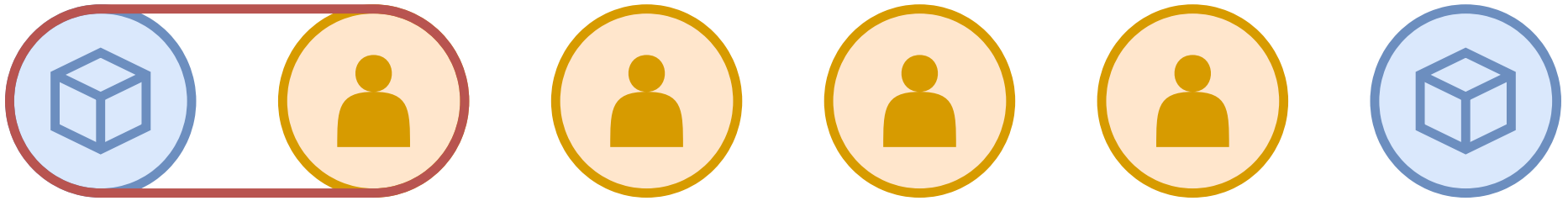


Case 2: Functional programming



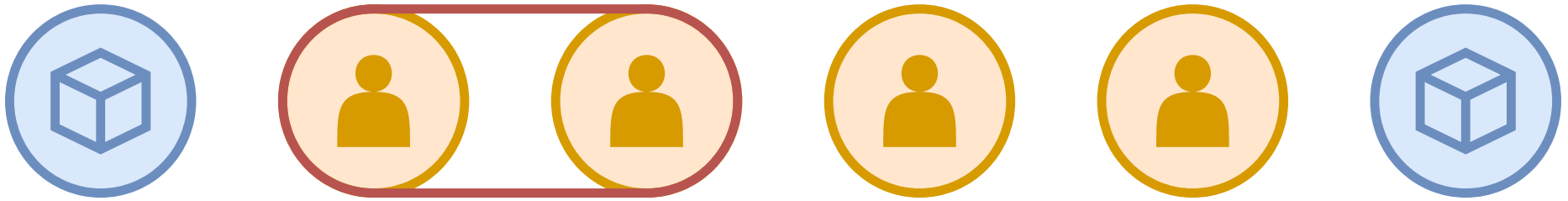
Case 2: Functional programming

```
.tuple_windows()
```

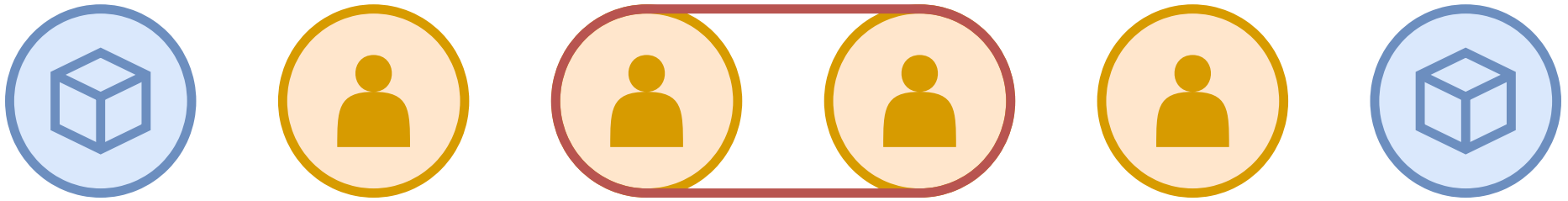


```
(from, to)
```

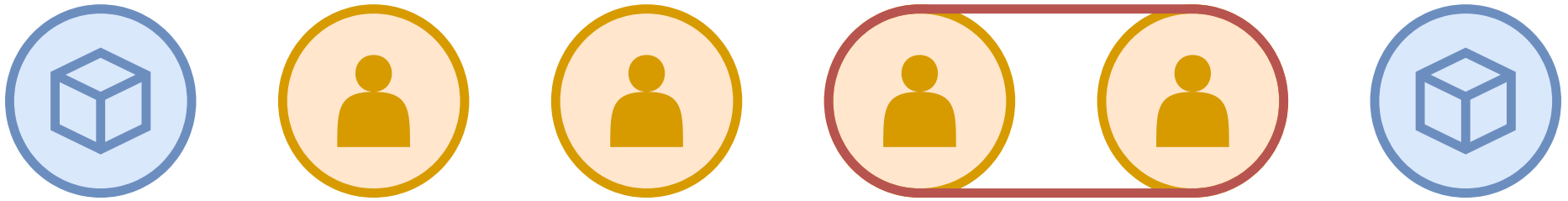
Case 2: Functional programming



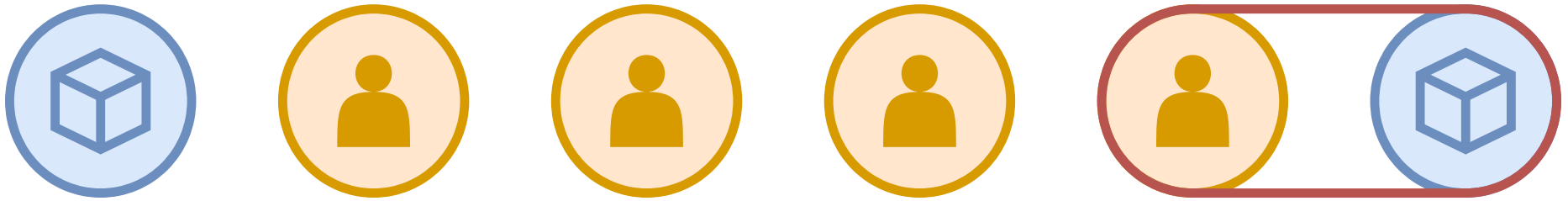
Case 2: Functional programming



Case 2: Functional programming



Case 2: Functional programming



Case 2: Functional programming

- Functional programming for many cases is:
  - More expressive
  - Documents itself
  - Easier to write without bugs
- No worries about performance
  - Zero cost abstractions
  - Lazy evaluation

Case 3: Locality of mutability

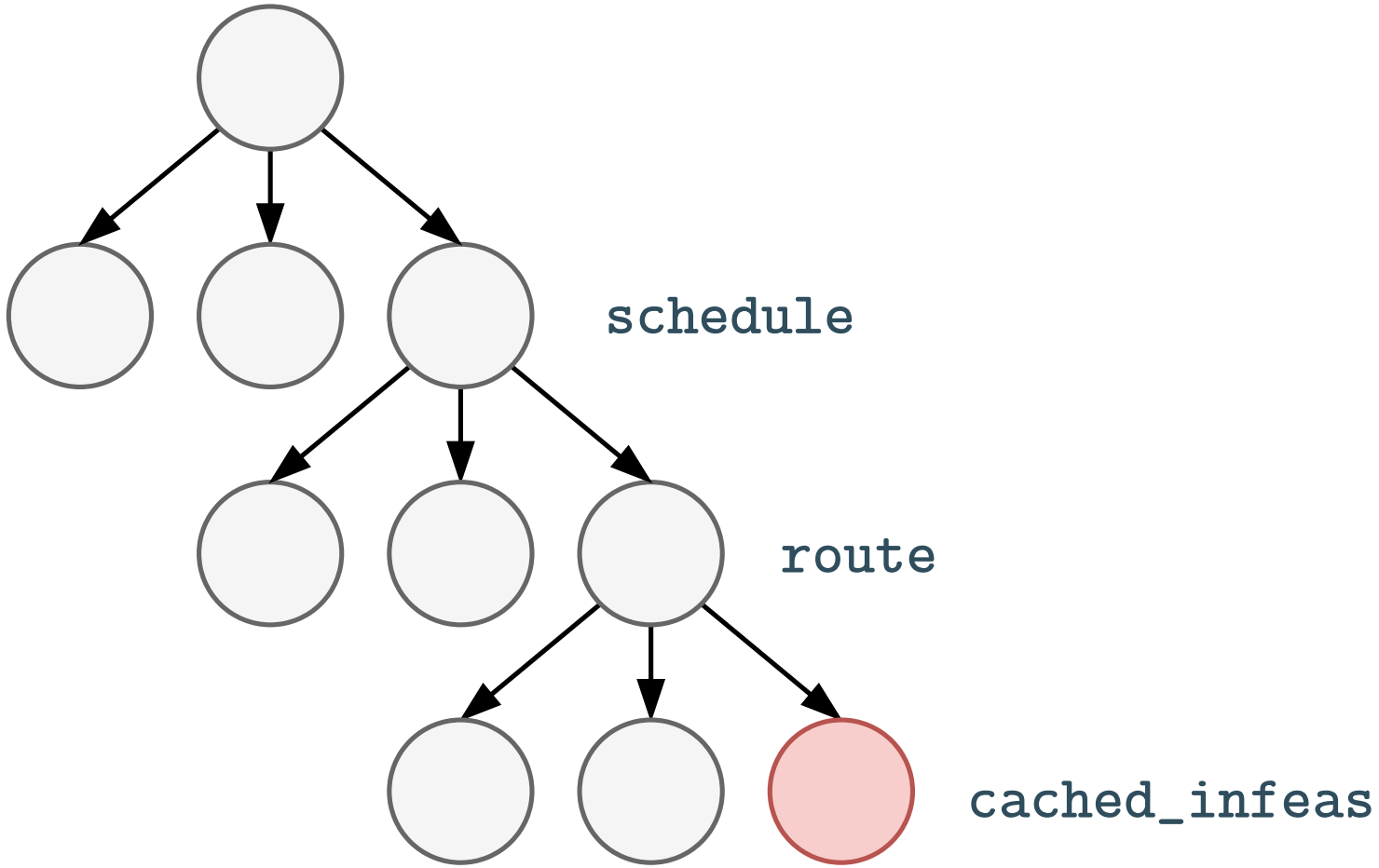
Case 3: Locality of mutability

- Borrow checker & ownership model
- → Explicit mutability
- → Hierarchical data structures

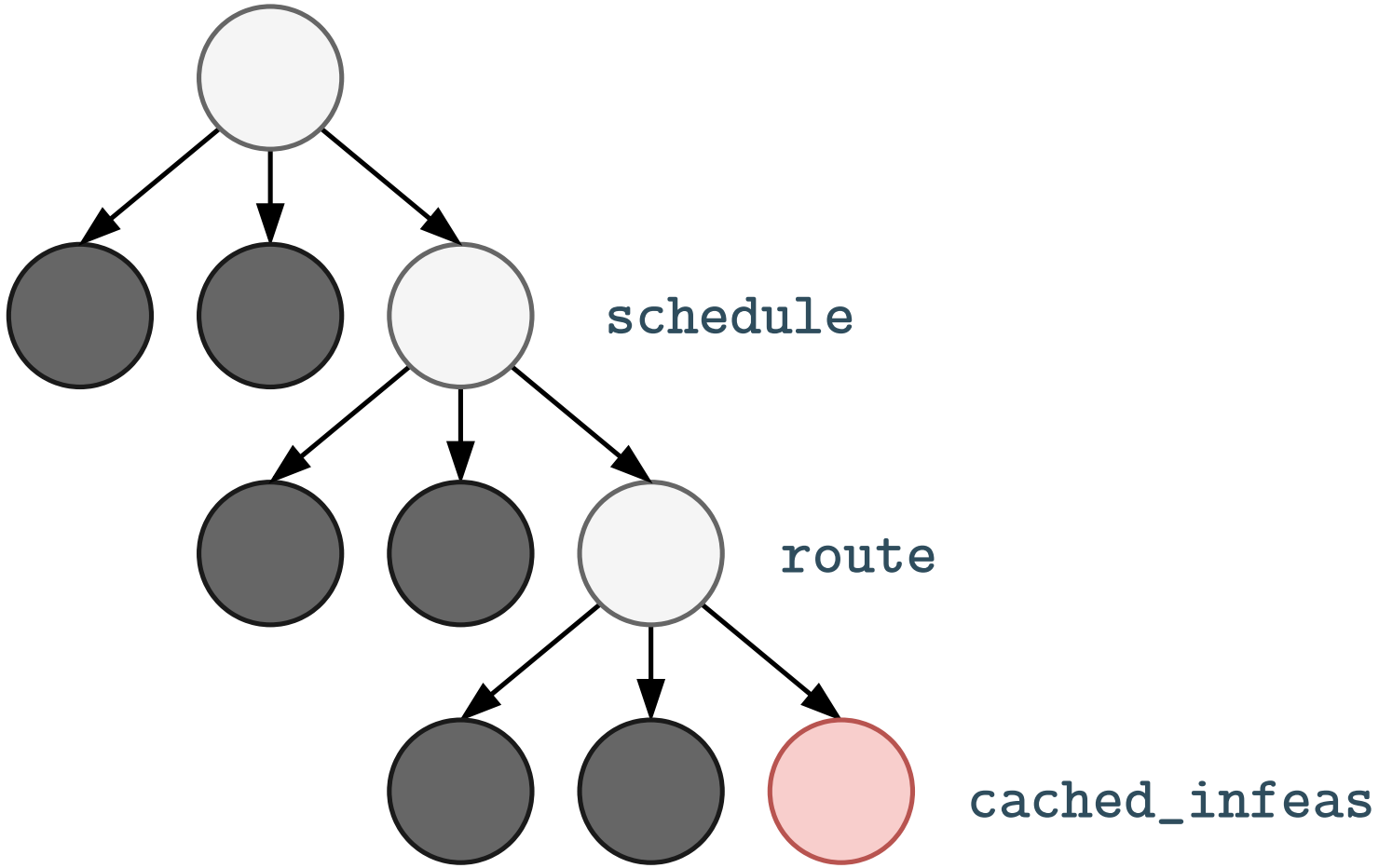
- Rust provides wrappers to be more flexible with ownership:
  - `Rc<T>, Arc<T>, RefCell<T>, Mutex<T>, RwLock<T>`
  - Also *explicit*

- Path of least resistance → *ownership tree*

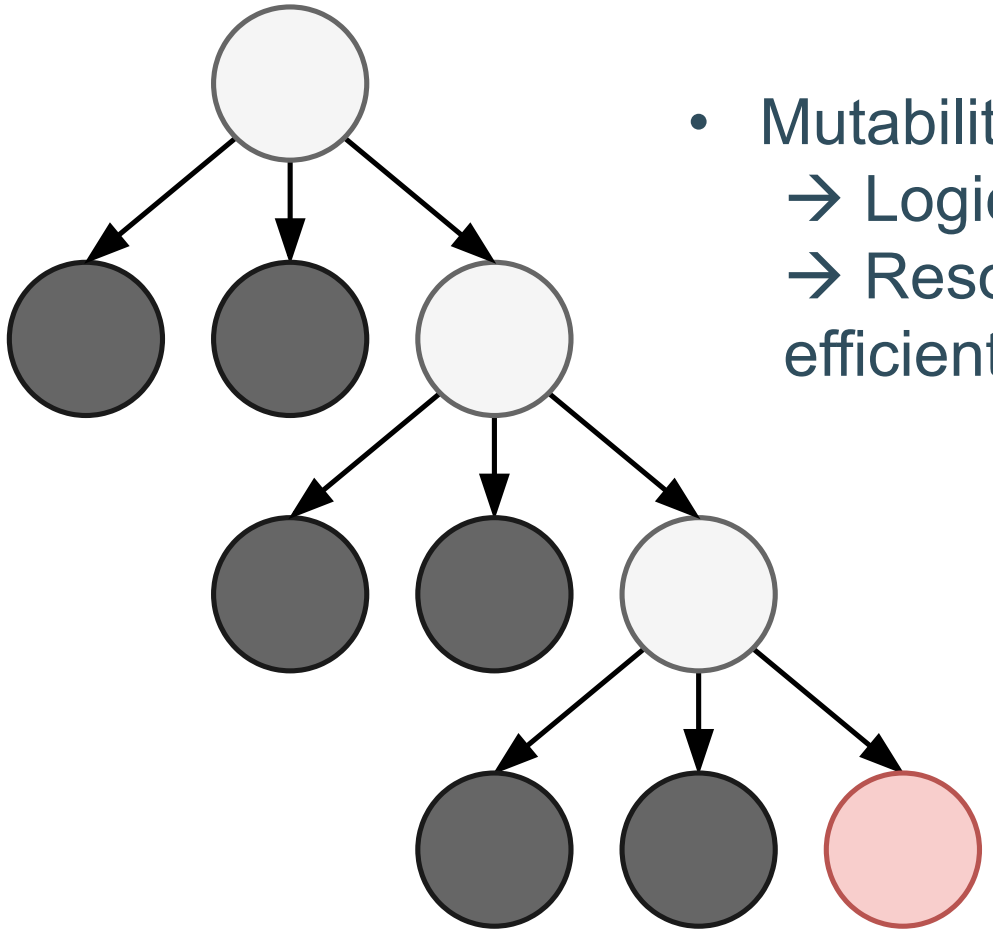
Case 3: Locality of mutability



Case 3: Locality of mutability



Case 3: Locality of mutability



- Mutability is *explicit & local*
  → Logic bugs are *encapsulated*
  → Resolving bugs is way more efficient

Thanks for your attention!