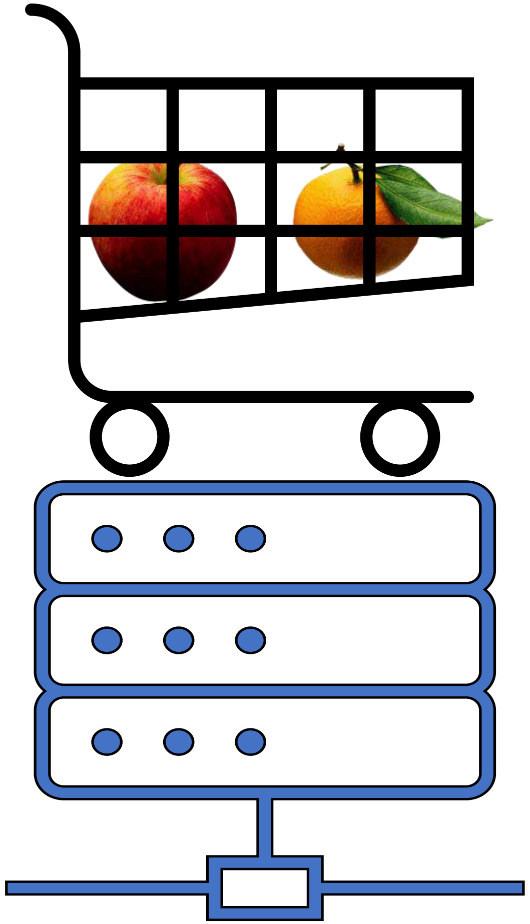
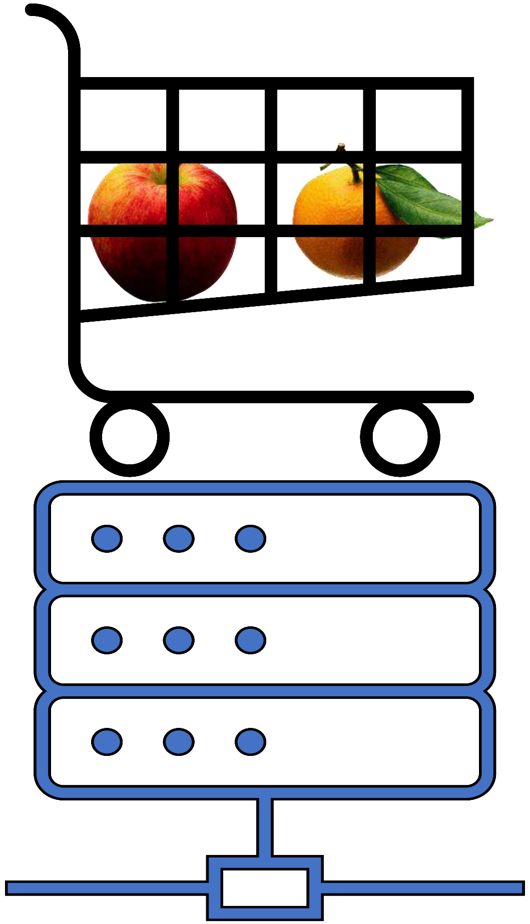
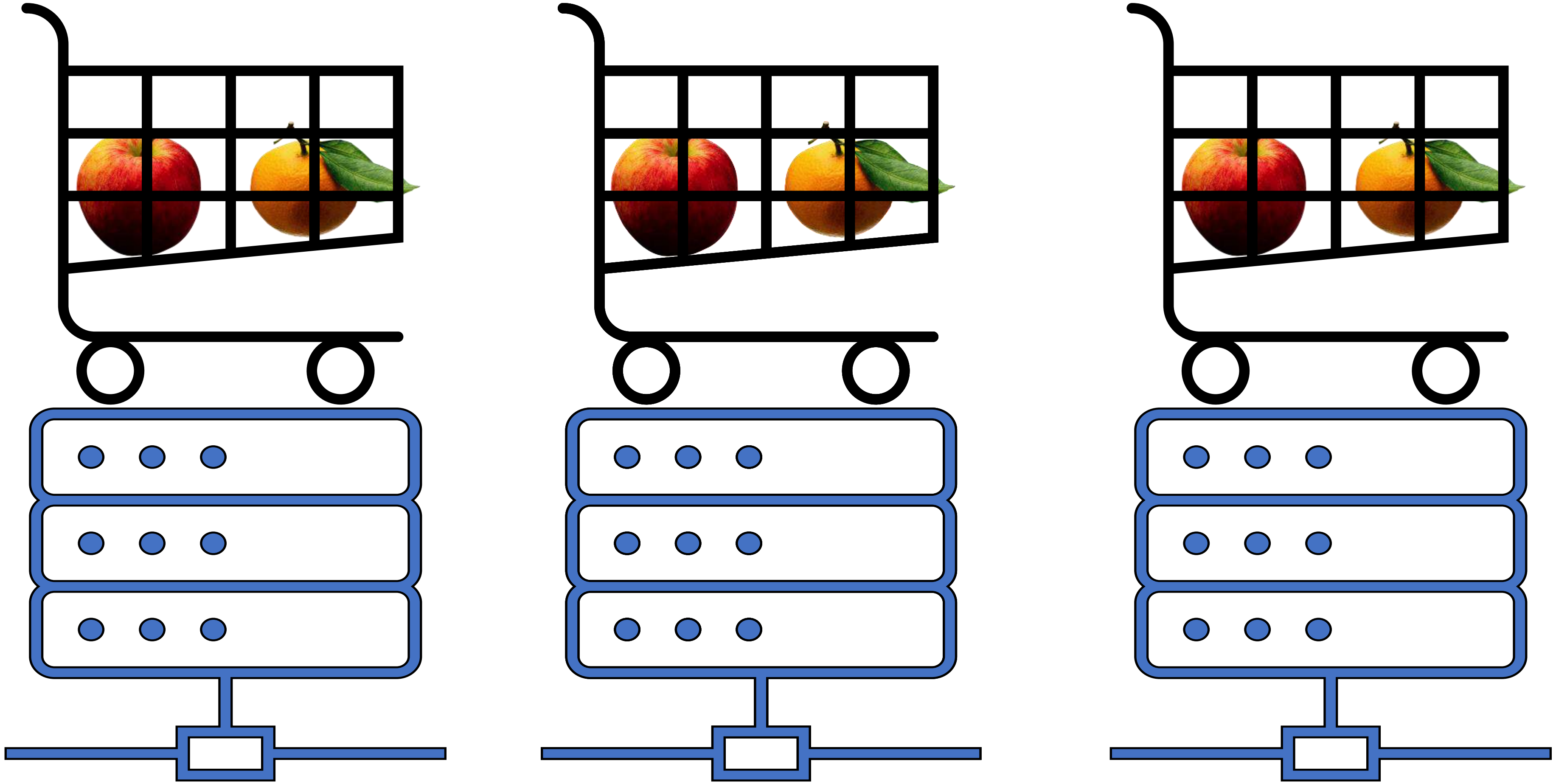
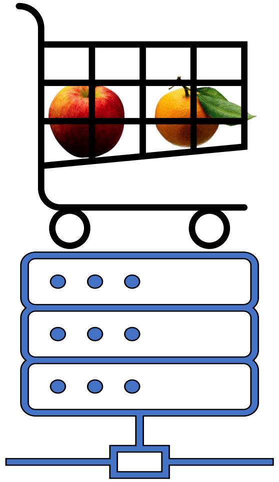
Towards a Compiler for Distributed Programs

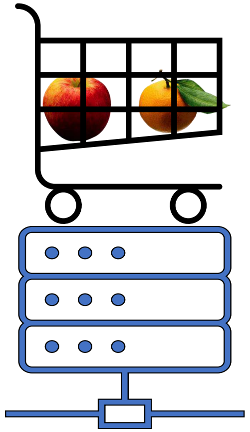
MAE MILANO
UC BERKELEY
PRINCETON UNIVERSITY

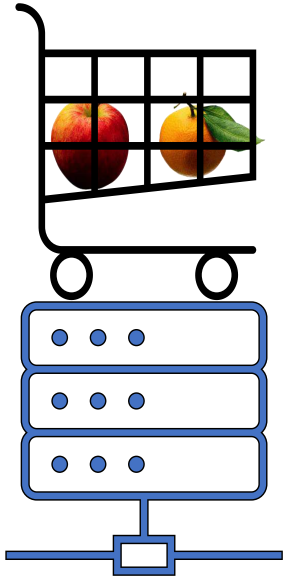


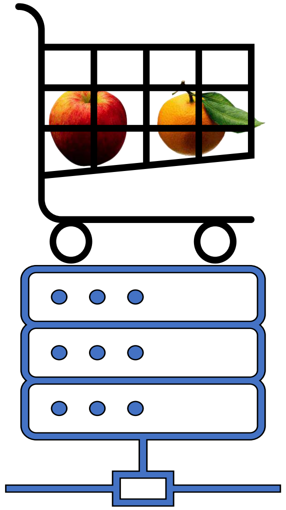


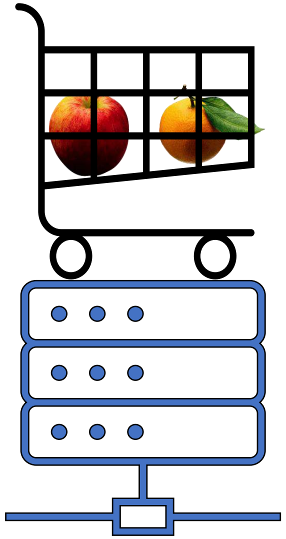


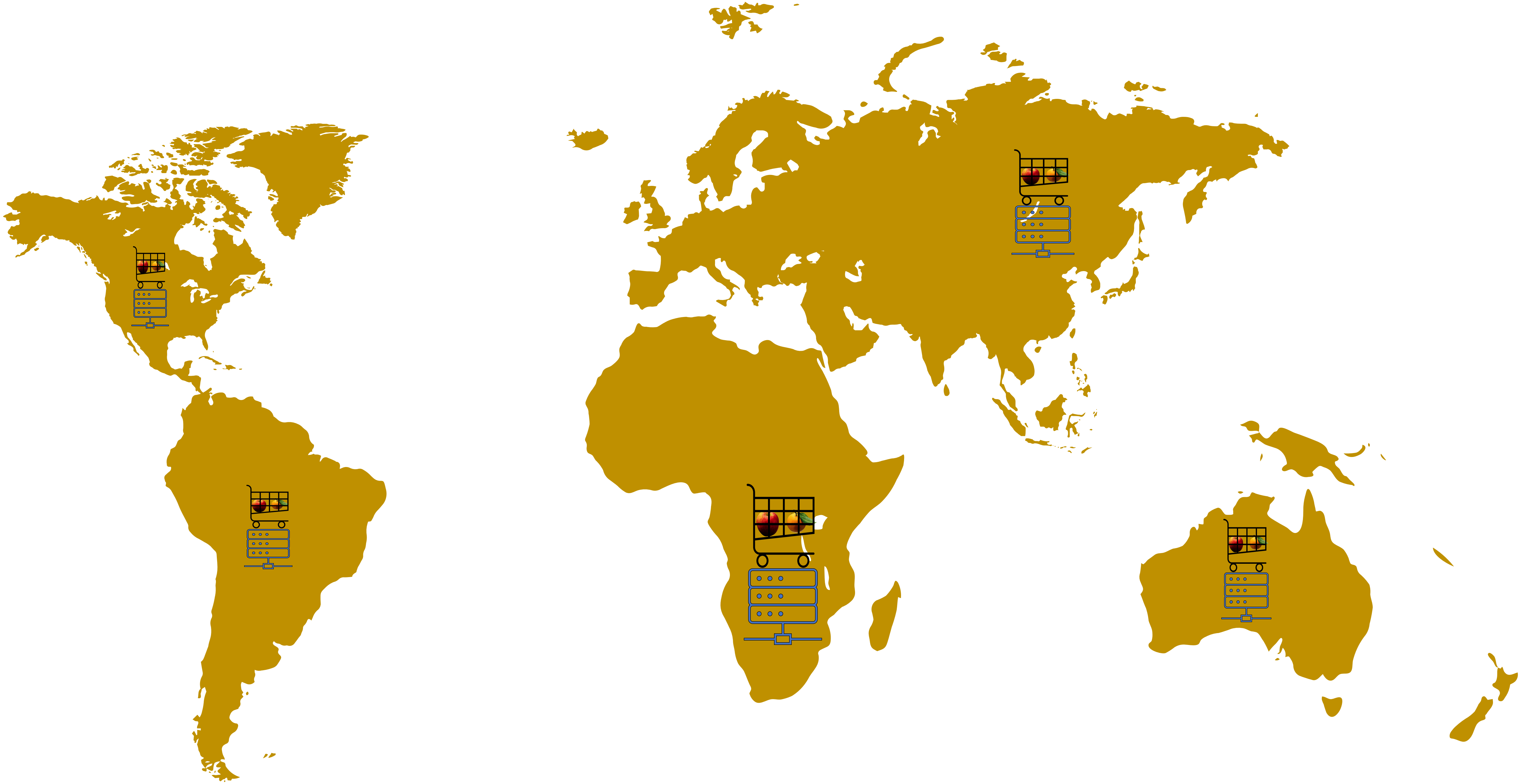


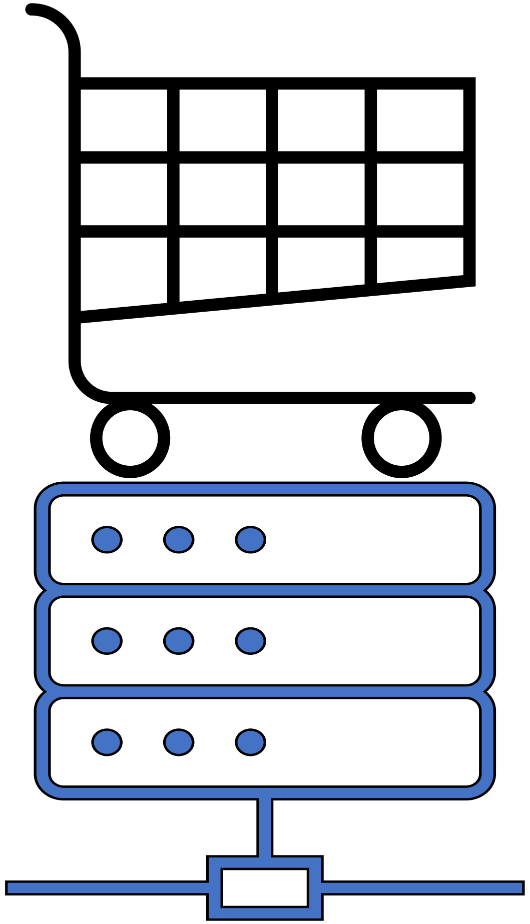
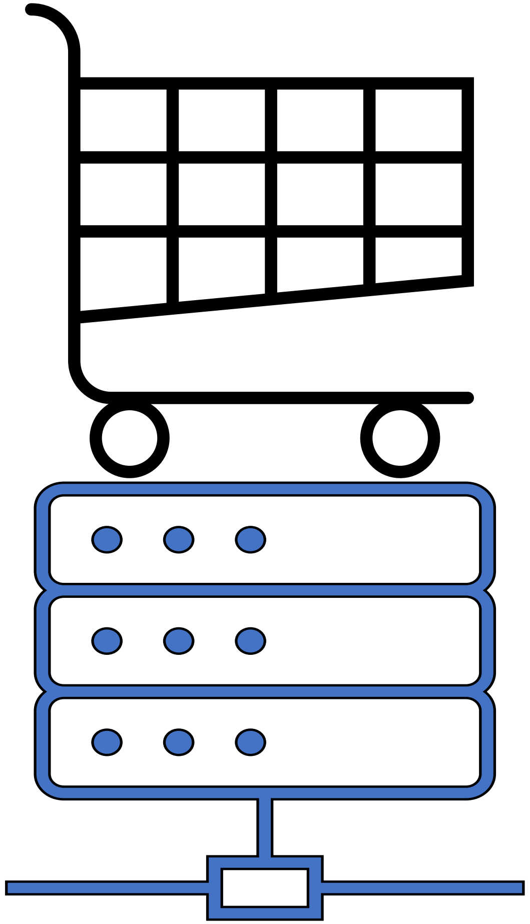
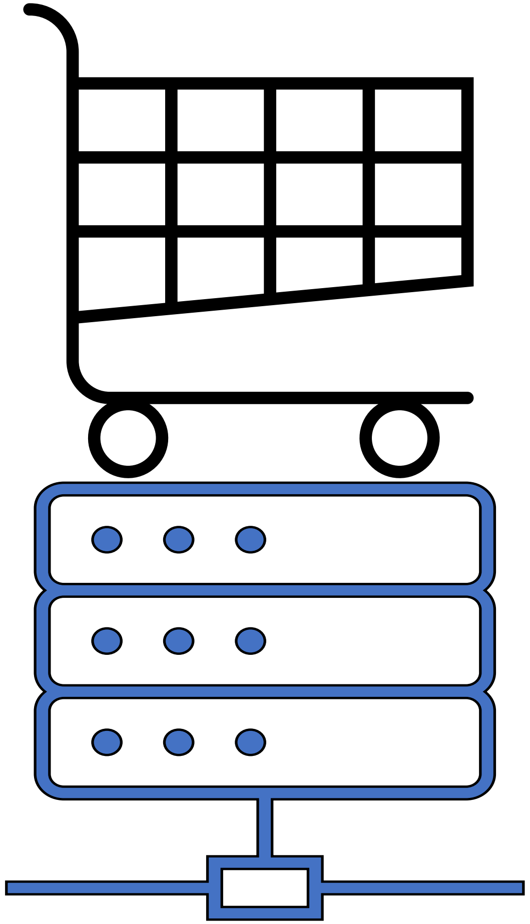


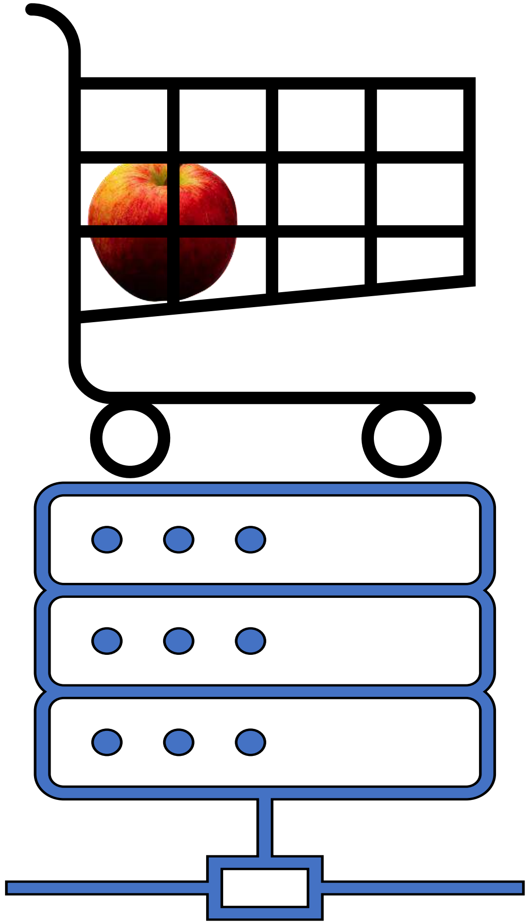
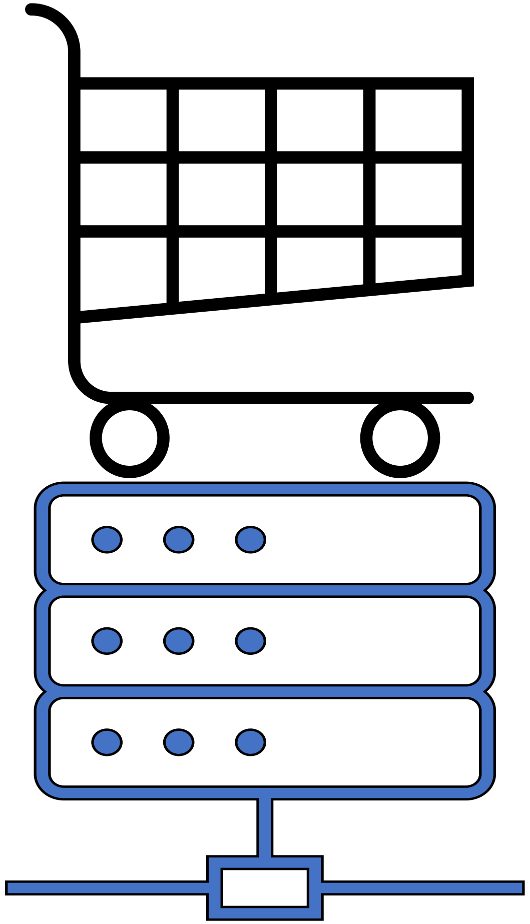
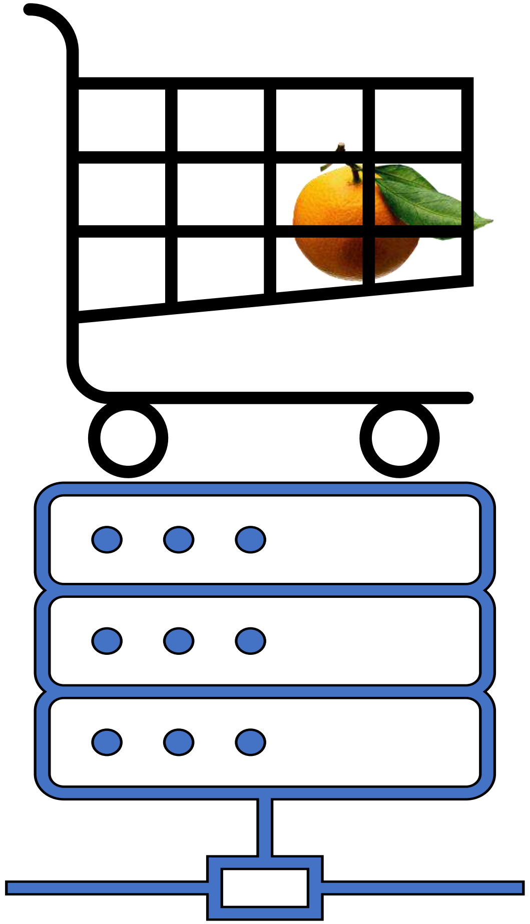


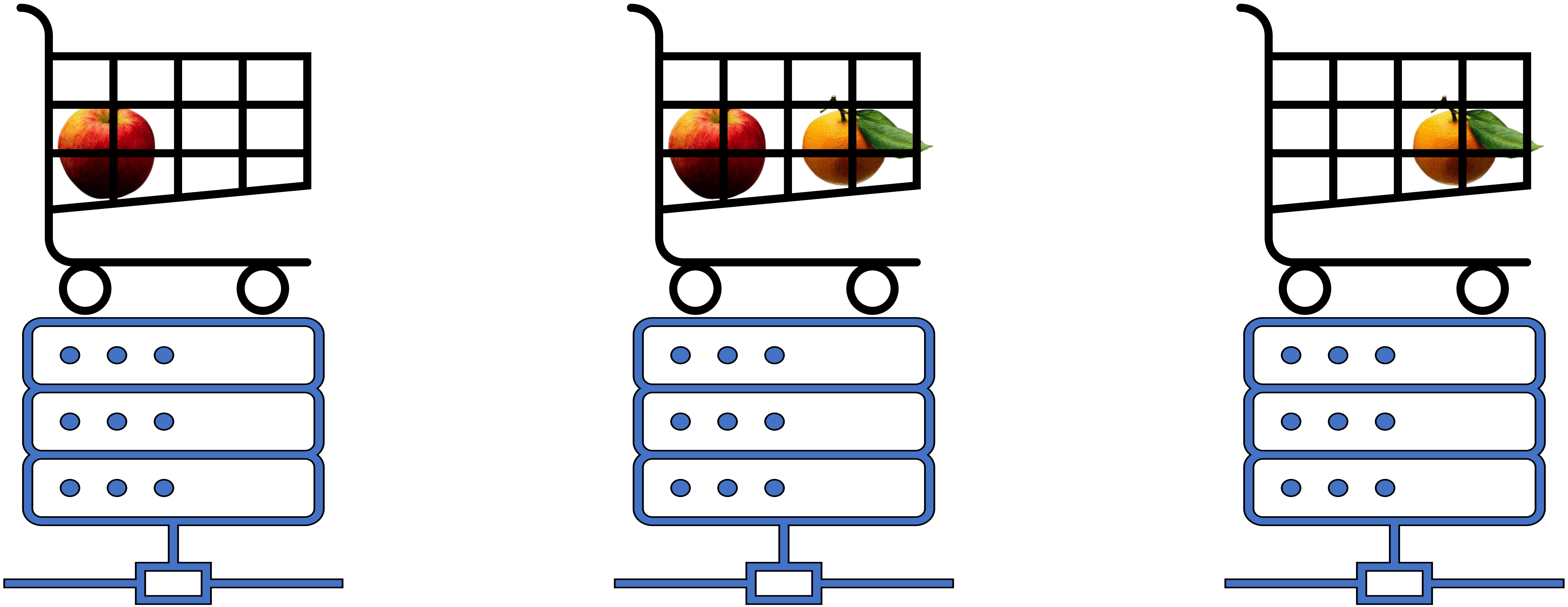






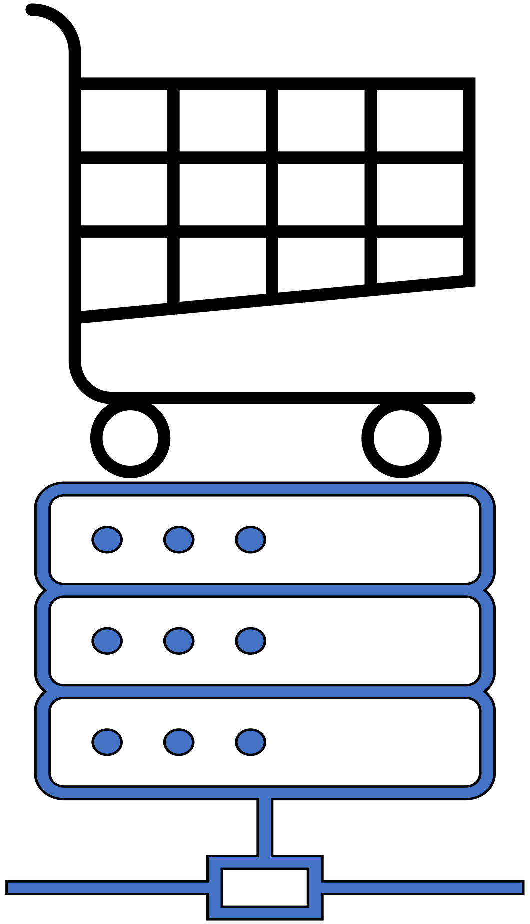




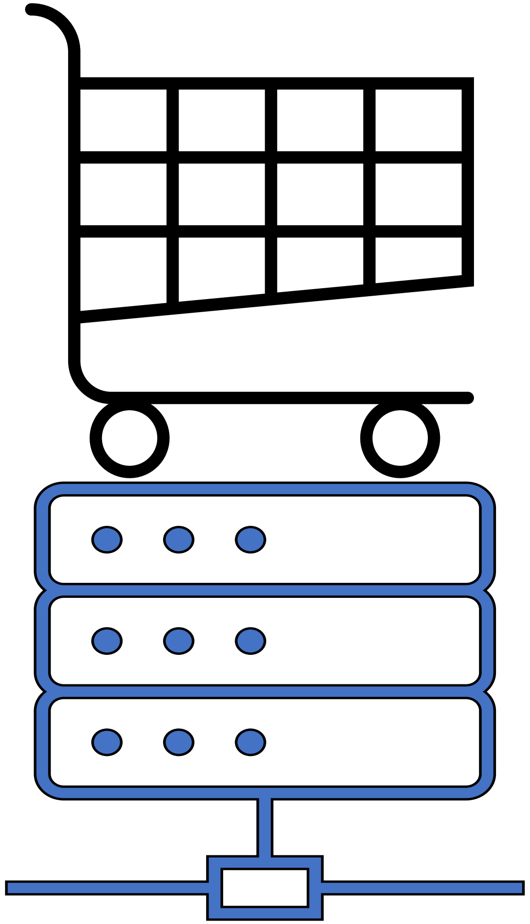
  

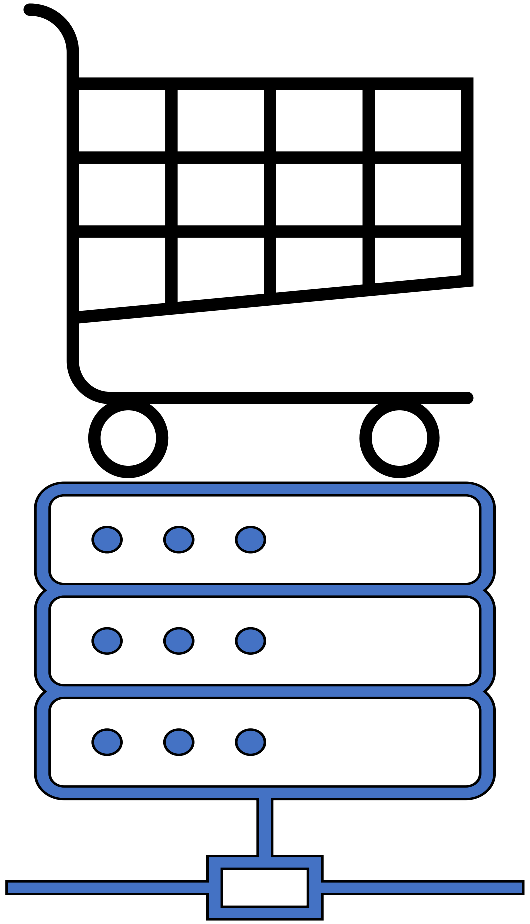



Available
and
Performant!

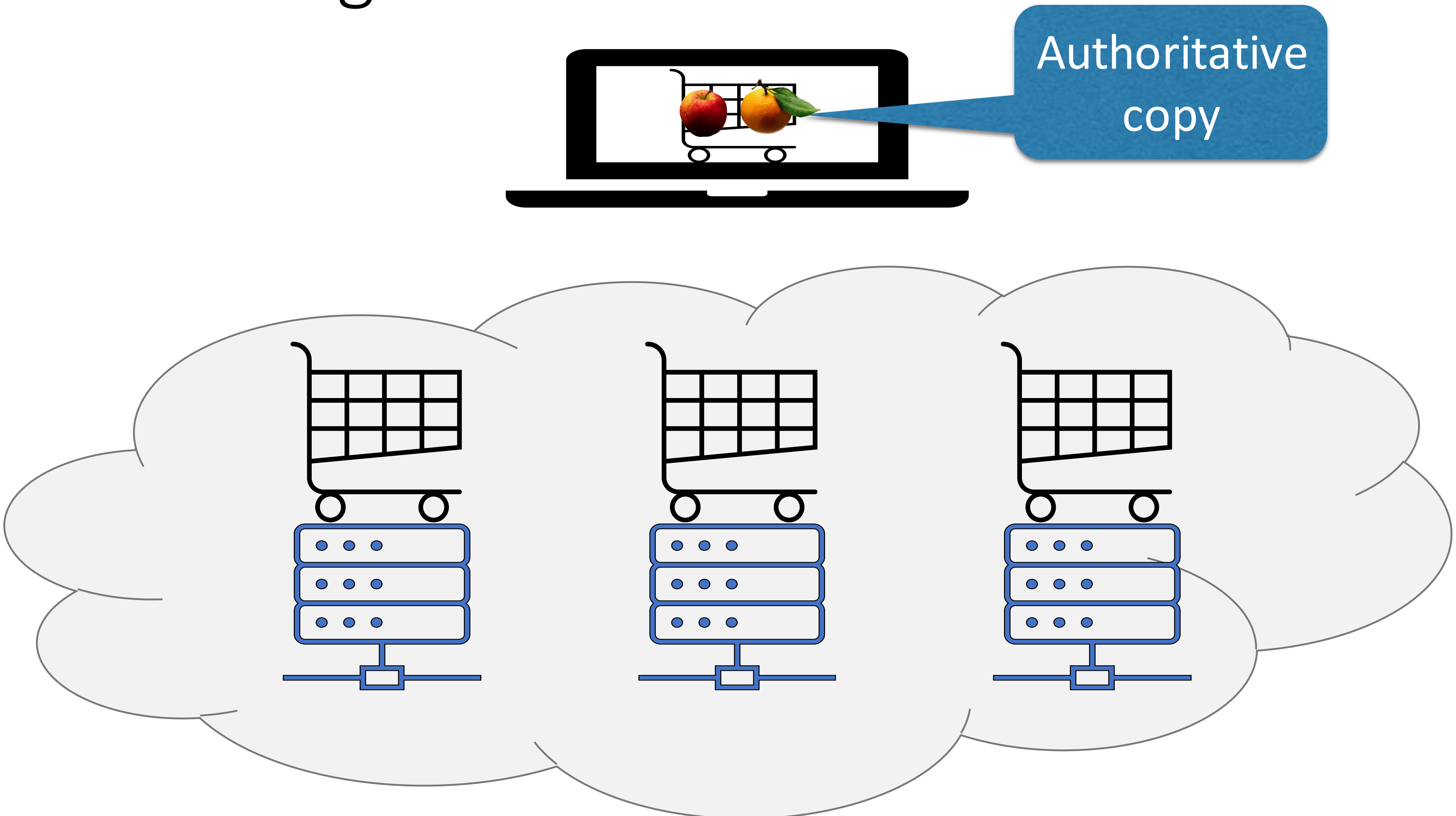
Manifesting Checkout



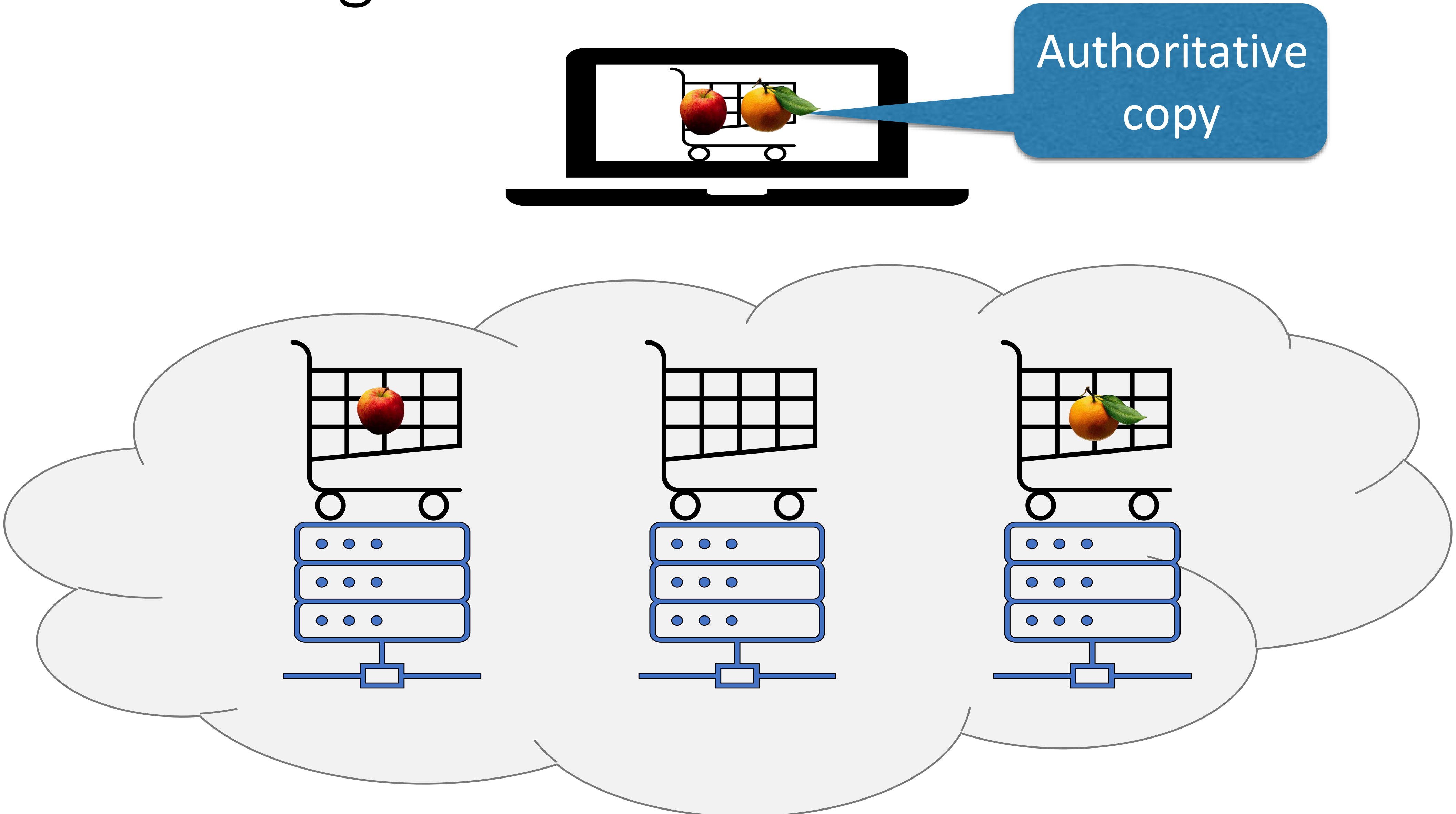




Manifesting Checkout

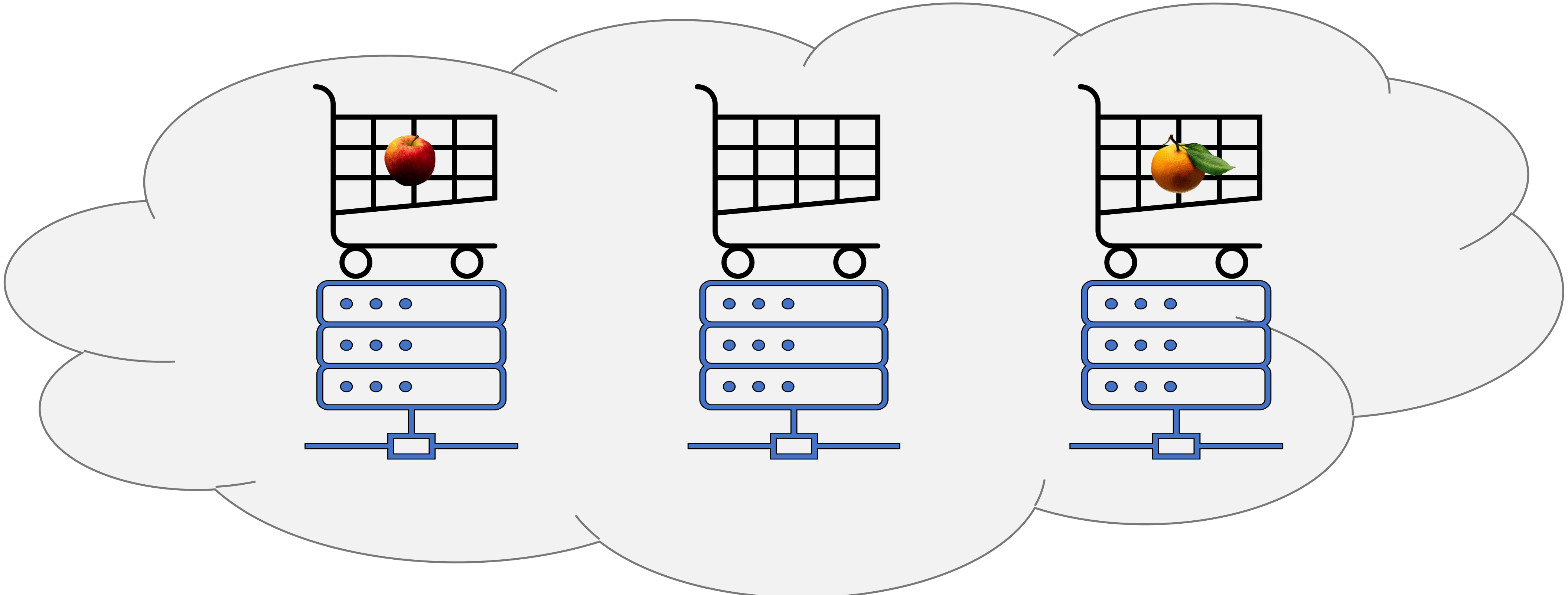


Manifesting Checkout



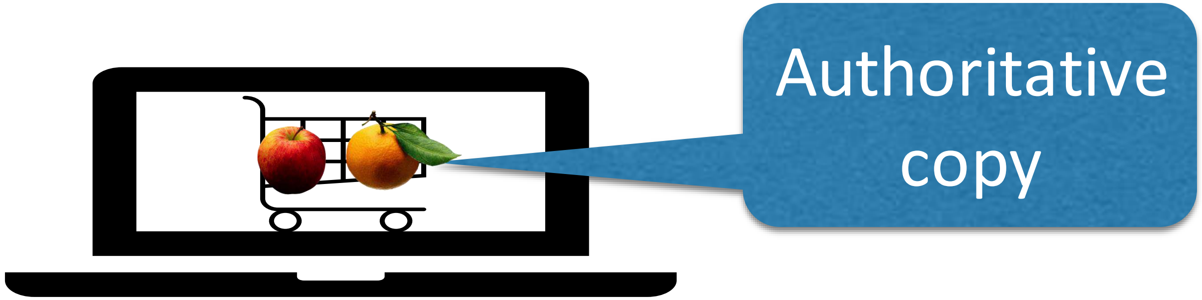
Manifesting Checkout

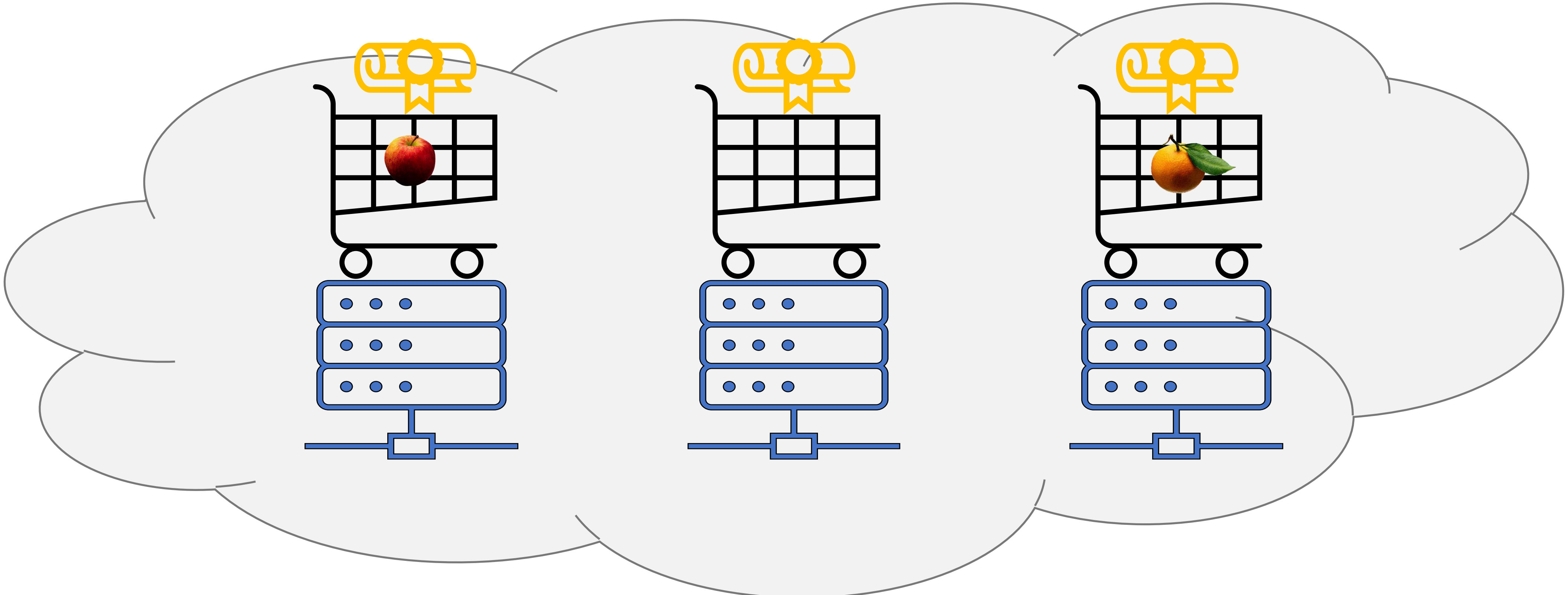
 = manifest



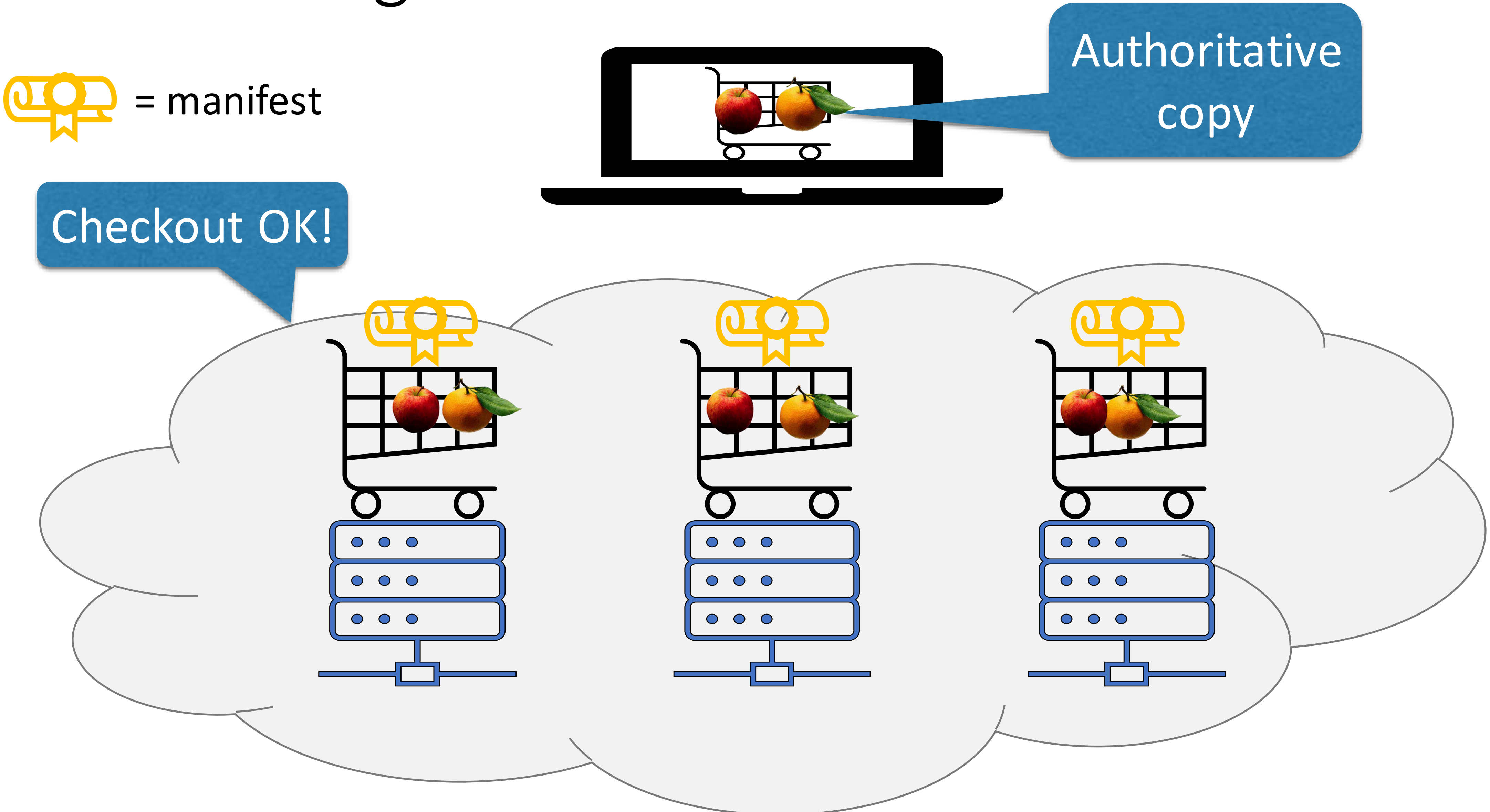
Manifesting Checkout

 = manifest





Manifesting Checkout



Programmers need more than shopping carts

Hard to find safe replication and optimizations!

Hydro


Joe Hellerstein


Prof. Alvin Cheung


Prof. Natacha Crooks


Mingwei Samuel


Lucky Katahanas


Prof.* Mae Milano


Chris Douglas

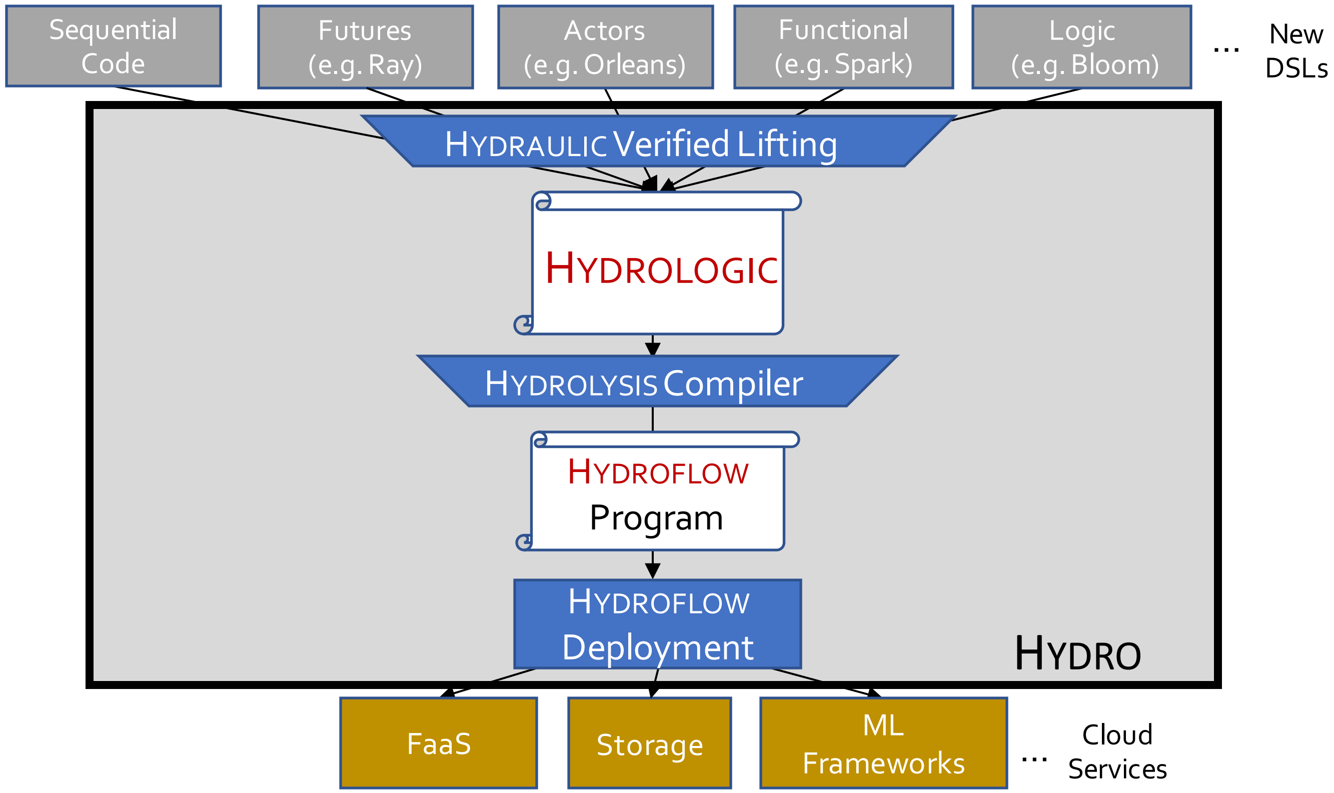

Conor Power


David Chu


Shadaj Laddad


Dr. Tiemo Bang

The Hydro Stack



The Hydro Stack

Common **intermediate language**

Mix consistency guarantees in a single distributed system

HYDROLOGIC

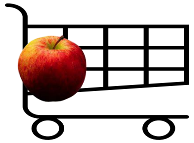
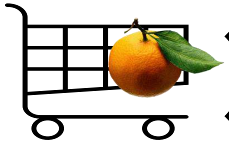
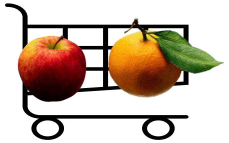
Allows **compositional, multi-paradigm** reasoning

enforce **strong** user-visible **consistency,** automatically

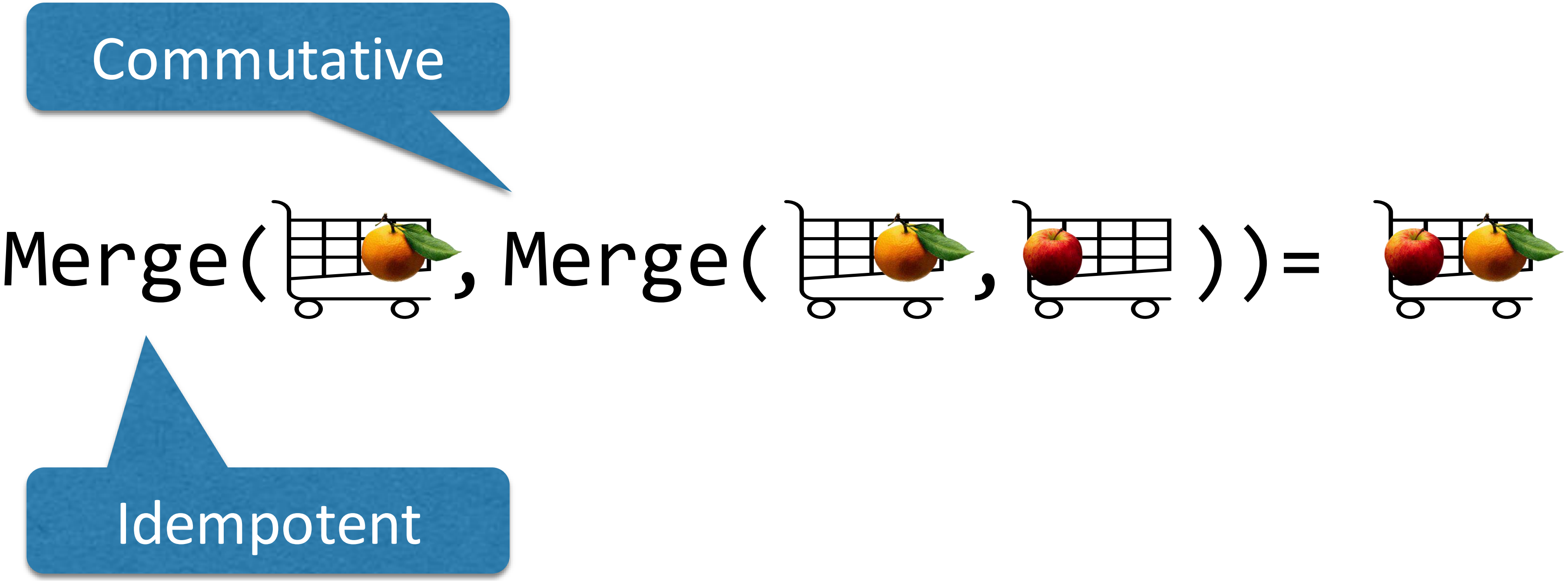
What makes a shopping cart so special?

What makes a shopping cart so special?

Commutative

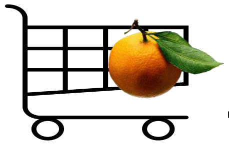
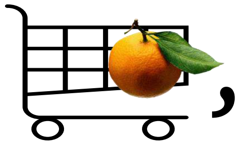
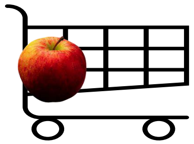
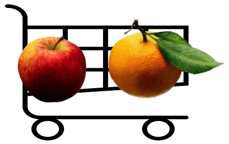
Merge( , ) = 

What makes a shopping cart so special?



What makes a shopping cart so special?

Commutative

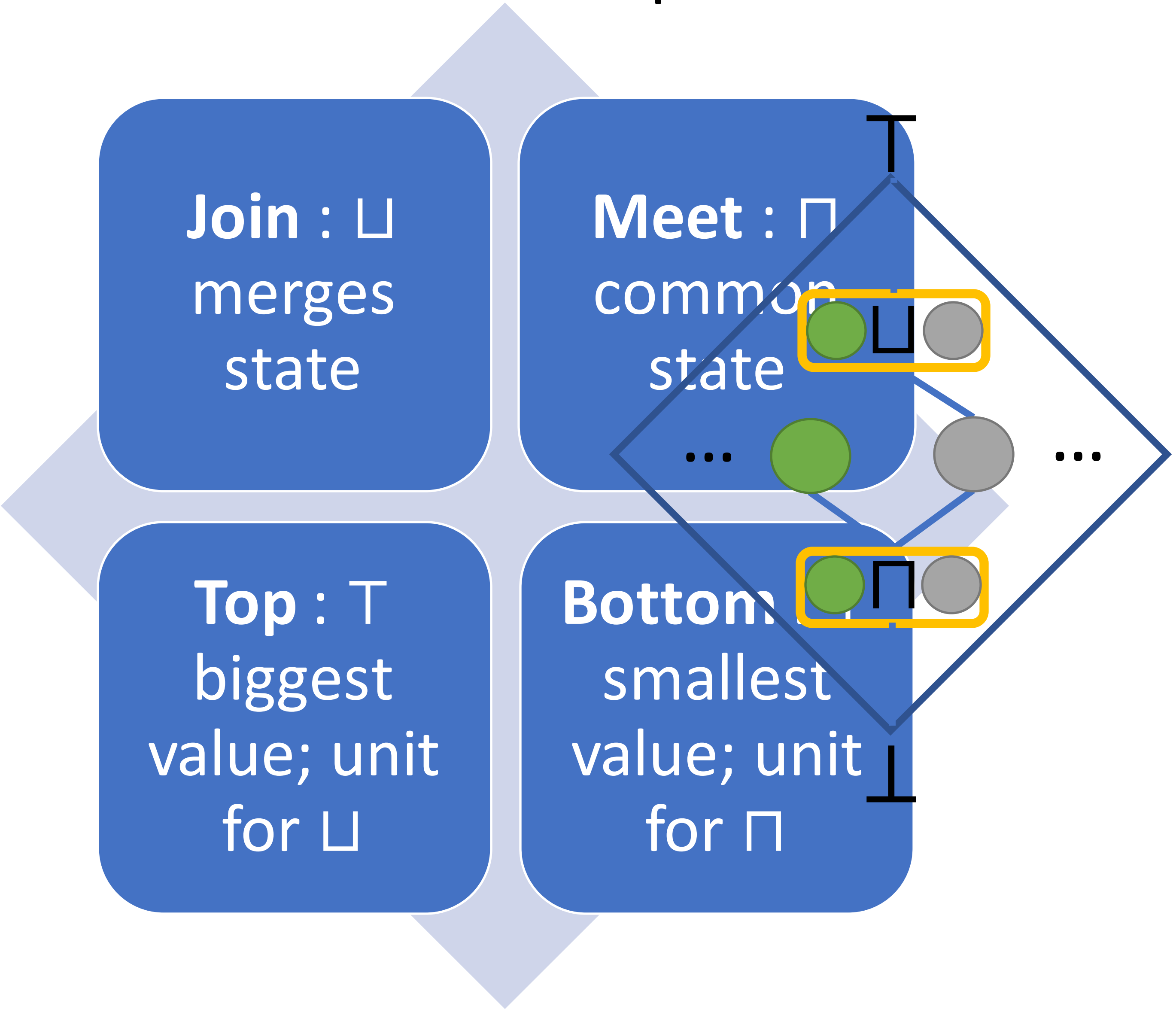
Merge(Merge(, , ))= 

Idempotent

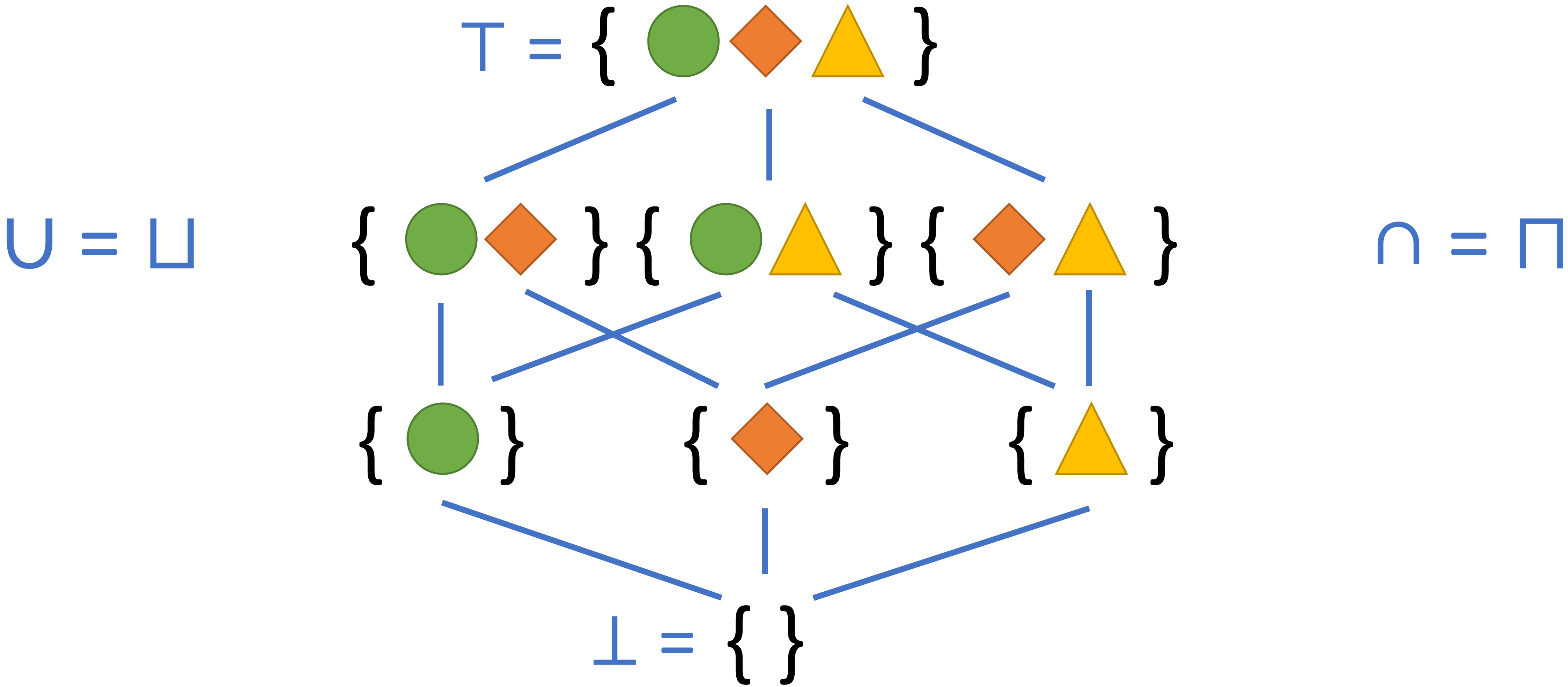
associative

Replication for free*!

Lattices: when replication is easy

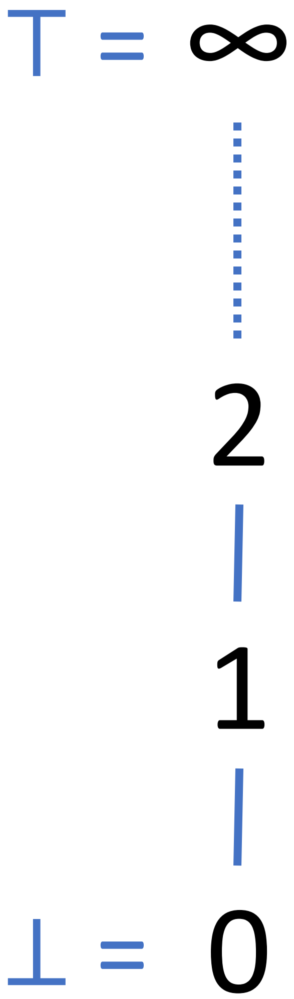


Lattice Examples: Power Set



Max Naturals

$\max = \sqcup$



$\min = \sqcap$

How do we safely observe **lattice-valued** replicated state?

Observing Replicated State

- Need **guarantees** about distributed state
- More **restricted mutations** allow more **general observations**

Is anyone over 18?

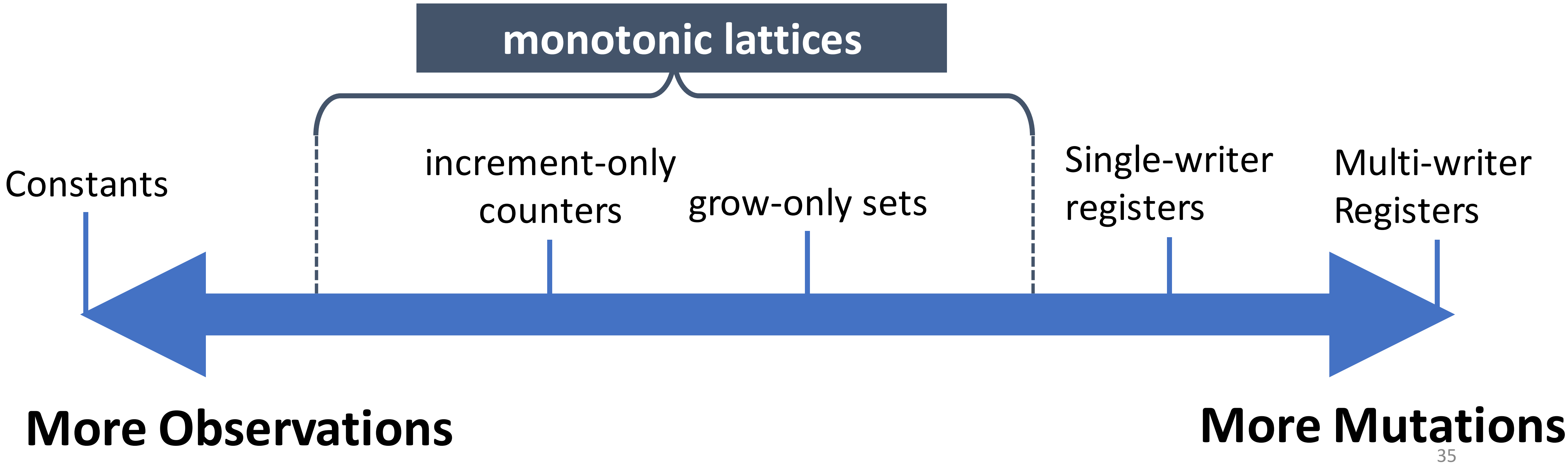
$\exists x.\ x > 18?$

No concurrent mutations can violate this

Reliable Observations

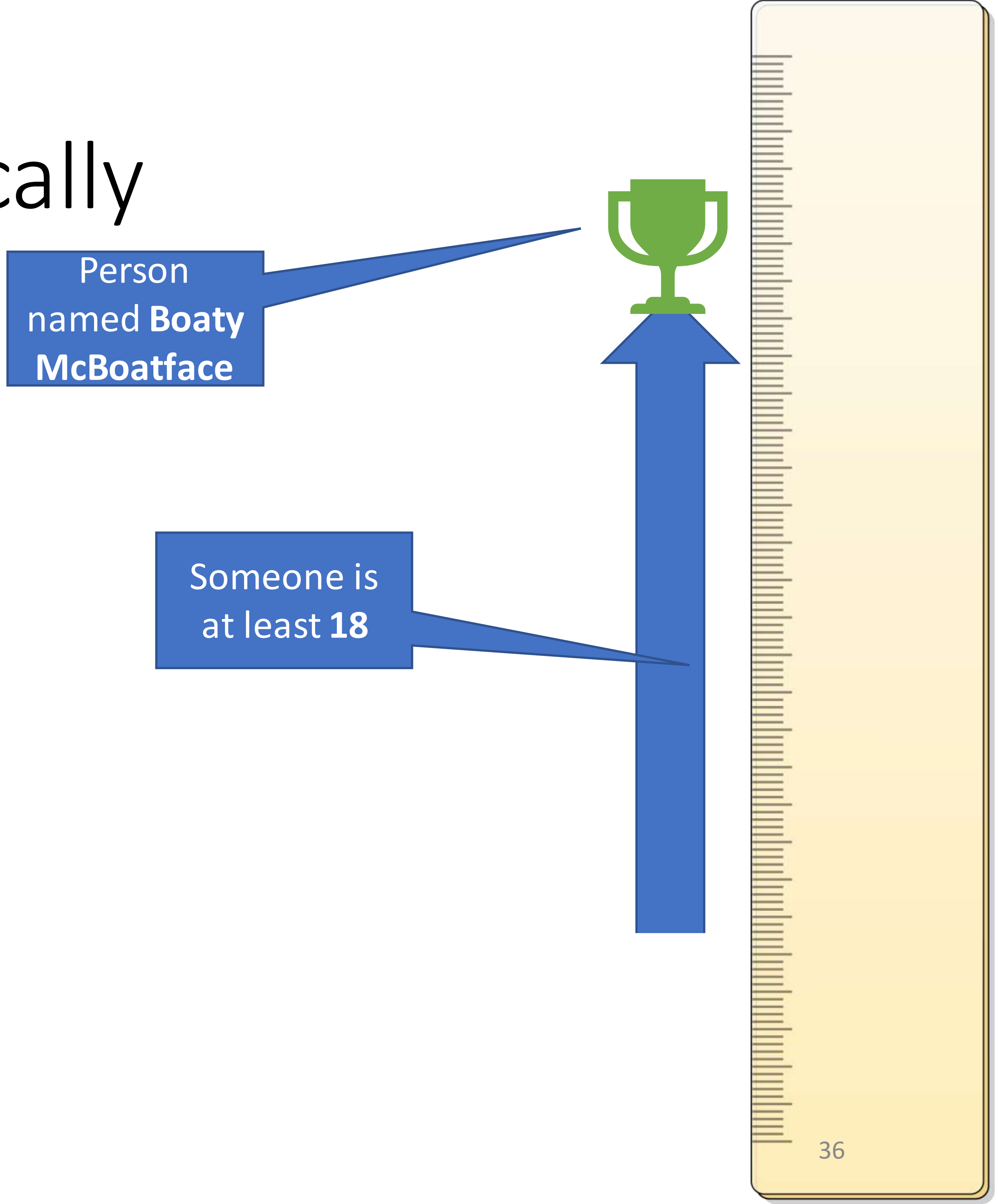
Monotonic object:
mutations are inflationary
with respect to lattice order.

Threshold observation:
comparisons with constants
are **stable predicates**



Programming monotonically

- If state at replicas **only grows**…
- And we only observe **thresholds**…
- Or **stable characteristics…**
- Our program can be correct **without coordination**

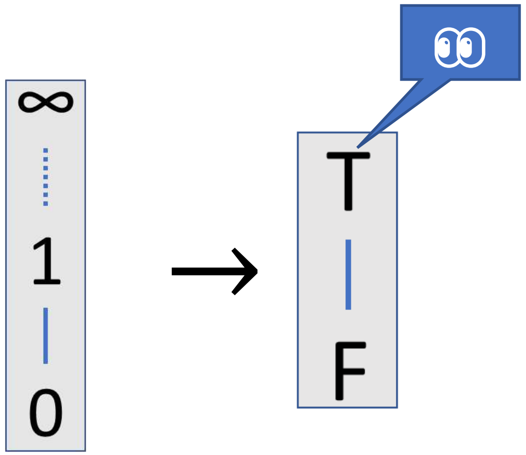


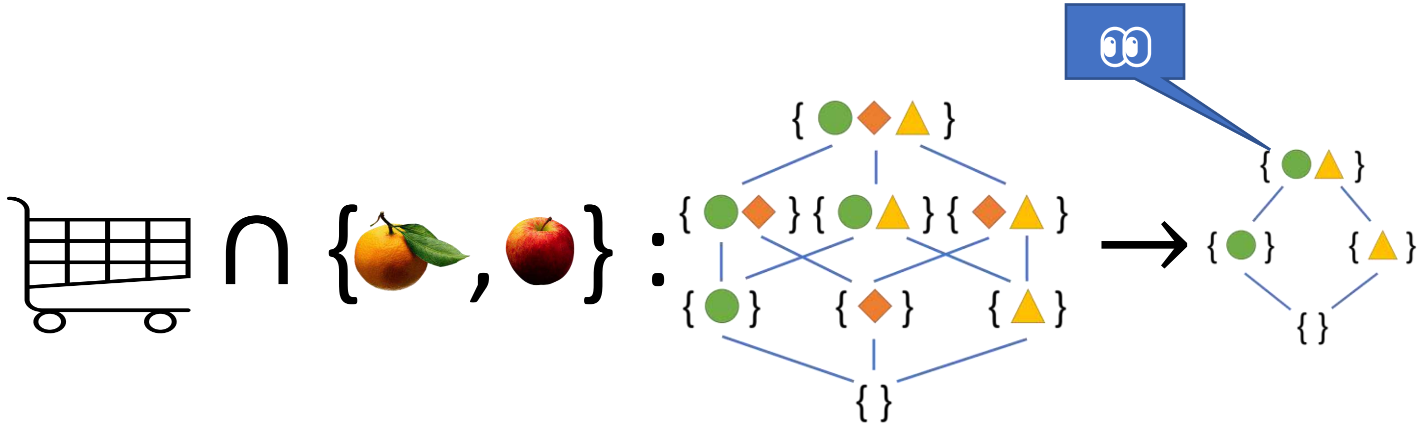
Beyond Threshold observations

Threshold observation:
comparisons with constants are **stable predicates**

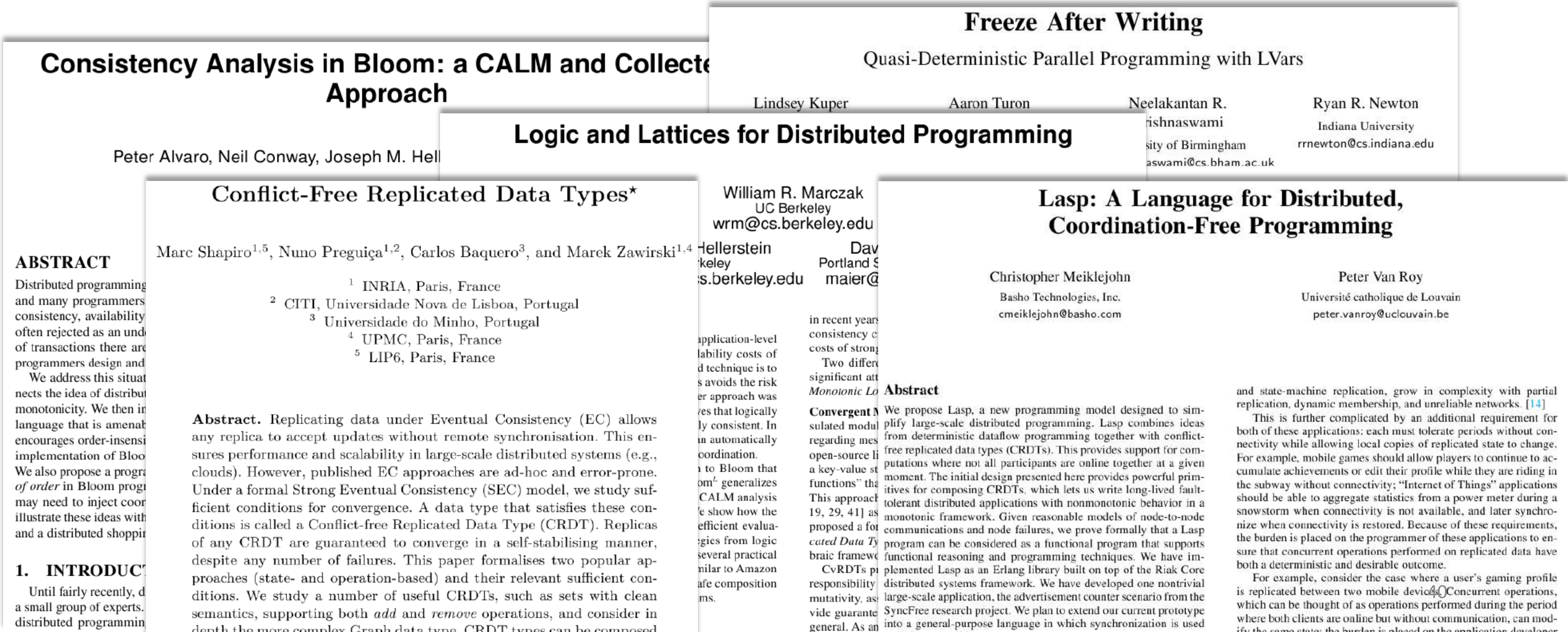
General observation:
Monotonic functions whose codomains have a **finite T** allow the **observation of T**

$x > 18$:





Standing on the shoulders of giants



Freeze After Writing

Quasi-Deterministic Parallel Programming with LVars

Lindsey Kuper   Aaron Turon   Neelakantan R. [...]rishnaswami   Ryan R. Newton

[...]sity of Birmingham   Indiana University

[...]aswami@cs.bham.ac.uk   rrnewton@cs.indiana.edu

Consistency Analysis in Bloom: a CALM and Collecte[...] Approach

Peter Alvaro, Neil Conway, Joseph M. Hell[...]

Logic and Lattices for Distributed Programming

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in [...]

Lasp: A Language for Distributed, Coordination-Free Programming

Christopher Meiklejohn
Basho Technologies, Inc.
cmeiklejohn@basho.com

Peter Van Roy
Université catholique de Louvain
peter.vanroy@uclouvain.be

Abstract

We propose Lasp, a new programming model designed to simplify large-scale distributed programming. Lasp combines ideas from deterministic dataflow programming together with conflict-free replicated data types (CRDTs). This provides support for computations where not all participants are online together at a given moment. The initial design presented here provides powerful primitives for composing CRDTs, which lets us write long-lived fault-tolerant distributed applications with nonmonotonic behavior in a monotonic framework. Given reasonable models of node-to-node communications and node failures, we prove formally that a Lasp program can be considered as a functional program that supports functional reasoning and programming techniques. We have implemented Lasp as an Erlang library built on top of the Riak Core distributed systems framework. We have developed one nontrivial large-scale application, the advertisement counter scenario from the SyncFree research project. We plan to extend our current prototype into a general-purpose language in which synchronization is used [...]

and state-machine replication, grow in complexity with partial replication, dynamic membership, and unreliable networks. [14]

This is further complicated by an additional requirement for both of these applications: each must tolerate periods without connectivity while allowing local copies of replicated state to change. For example, mobile games should allow players to continue to accumulate achievements or edit their profile while they are riding in the subway without connectivity; "Internet of Things" applications should be able to aggregate statistics from a power meter during a snowstorm when connectivity is not available, and later synchronize when connectivity is restored. Because of these requirements, the burden is placed on the programmer of these applications to ensure that concurrent operations performed on replicated data have both a deterministic and desirable outcome.

For example, consider the case where a user's gaming profile is replicated between two mobile devices. Concurrent operations, which can be thought of as operations performed during the period where both clients are online but without communication, can mod[...]

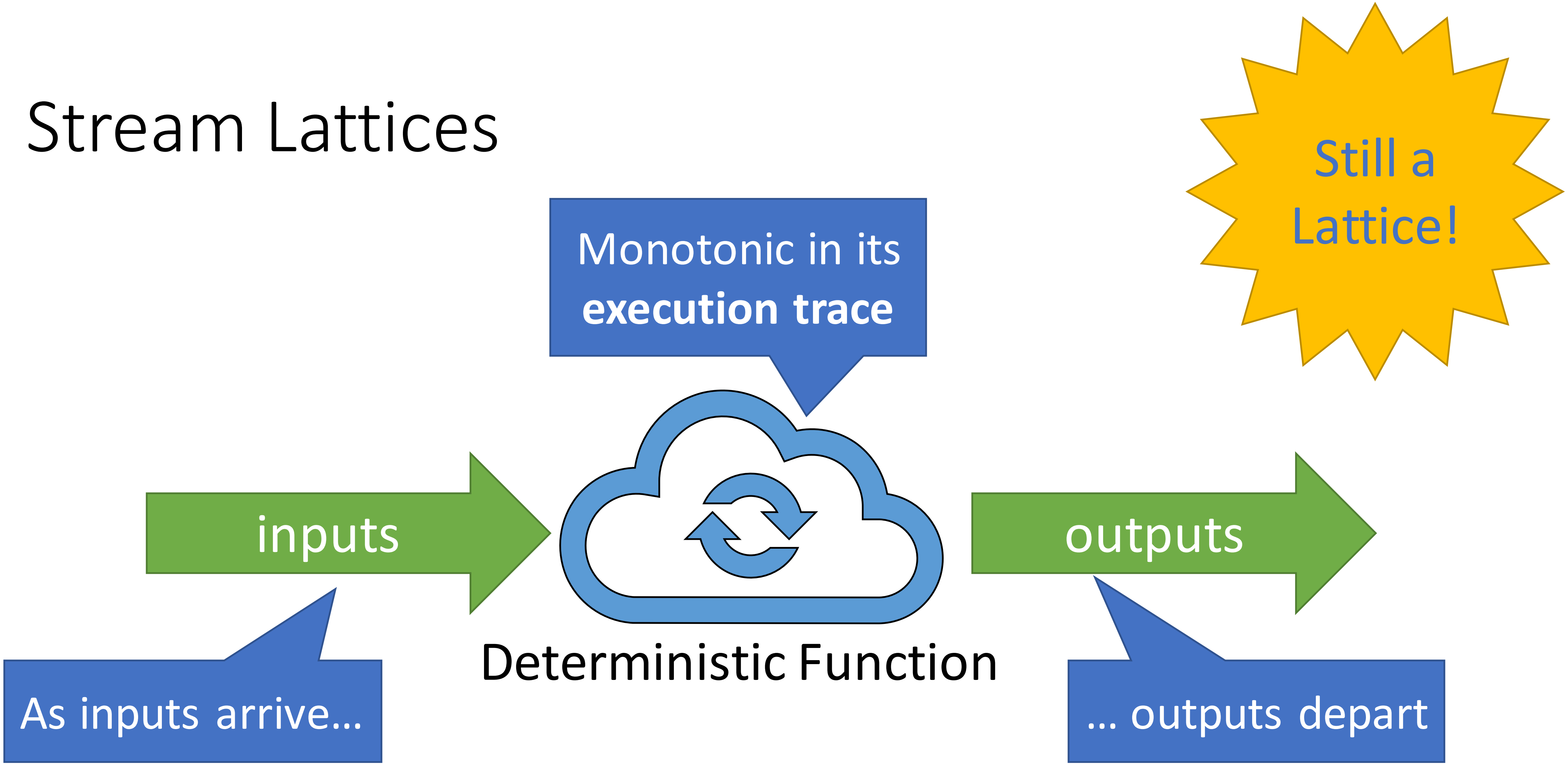
This paper: what about these?

SEQUENTIAL PROCESSES?

COMMUNICATING STATE MACHINES?

TRADITIONAL CODE?

Stream Lattices



Prefix Lattice

$\top = [\infty]$

$[i_1, i_2, i_3]$

$[i_1, i_2]$

$[i_1]$

$\bot = [\ ]$

keep-longer = $\sqcup$

Keep-shorter = $\sqcap$

Isomorphic to MaxNat!

Bounded Prefix Lattice

keep-longer = ⊔

$\top = [i_1, i_2, i_3]$

|

$[i_1, i_2]$

|

$[i_1]$

|

$\bot = [\ ]$

∞

Keep-shorter = ⊓

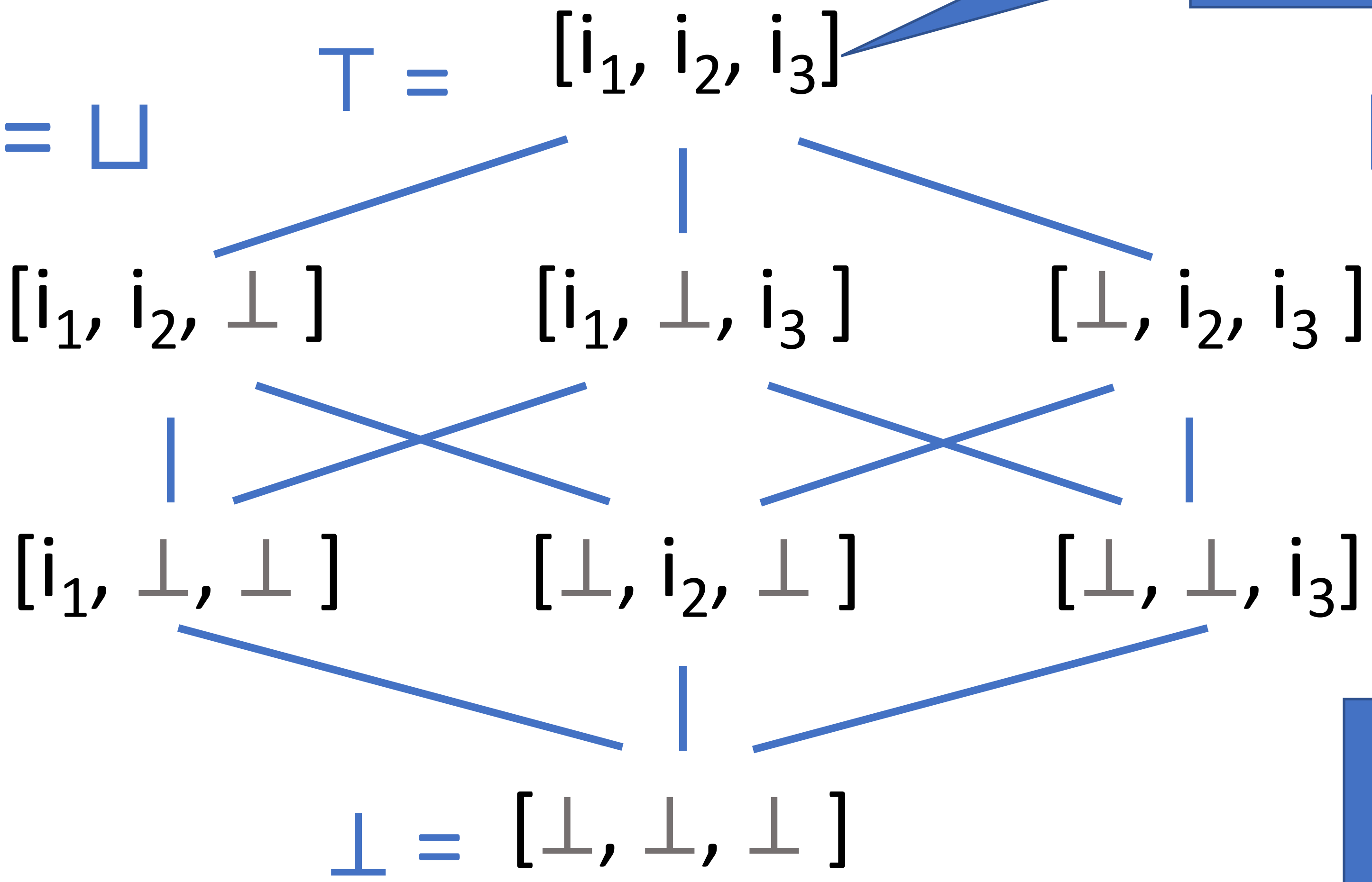
Most Programs Terminate!

Sealed Set of Indexed Values

This is sealed! Safe to read 👀

keep-value = ⊔

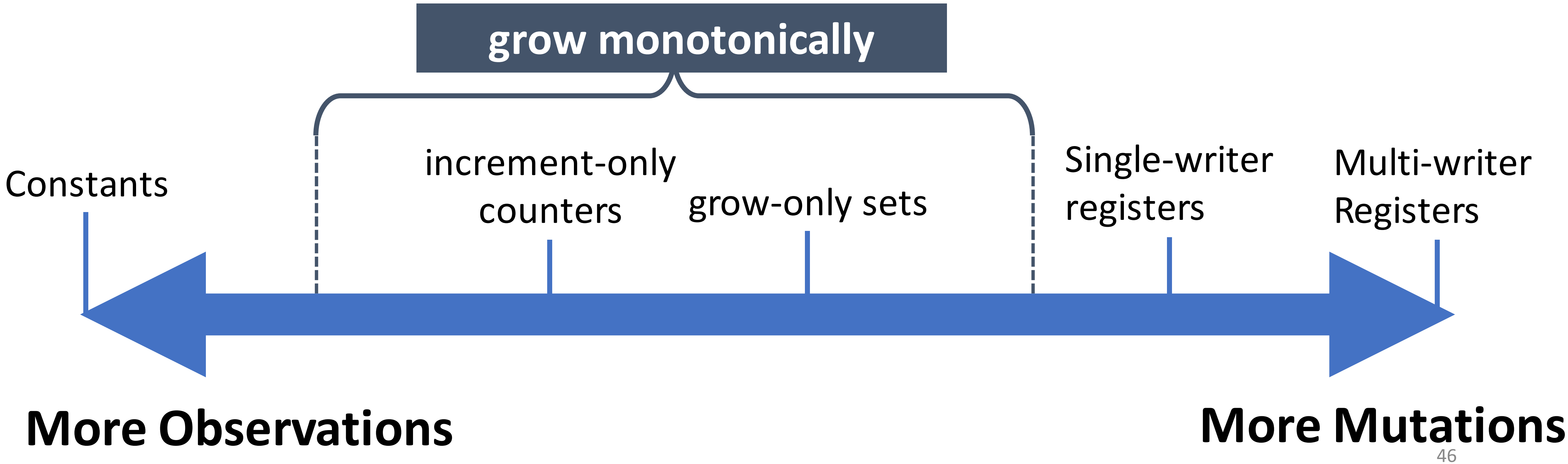
keep-⊥ = ⊓



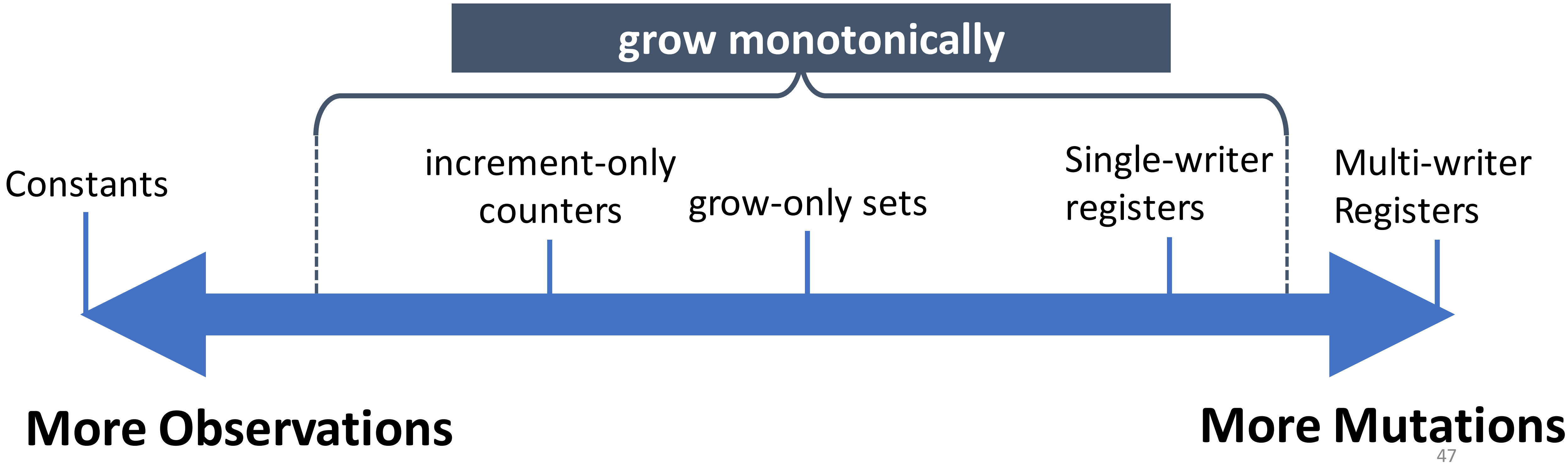
Allows out-of-order arrival!

Isomorphic to powerset!

Reliable Observations

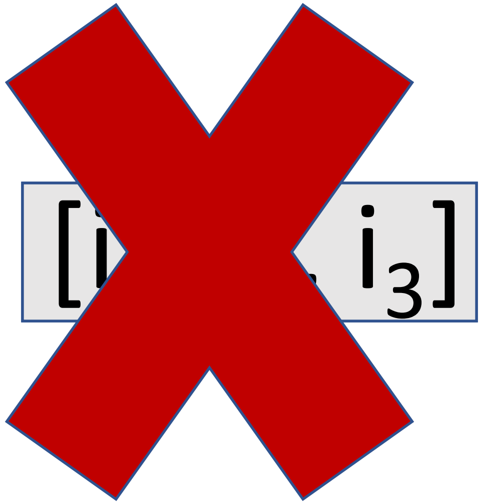


Reliable Observations



Let’s merge some streams!

$$[i_1, i_2, i_3] \sqcup [i_1, i_2, o_3] =$$



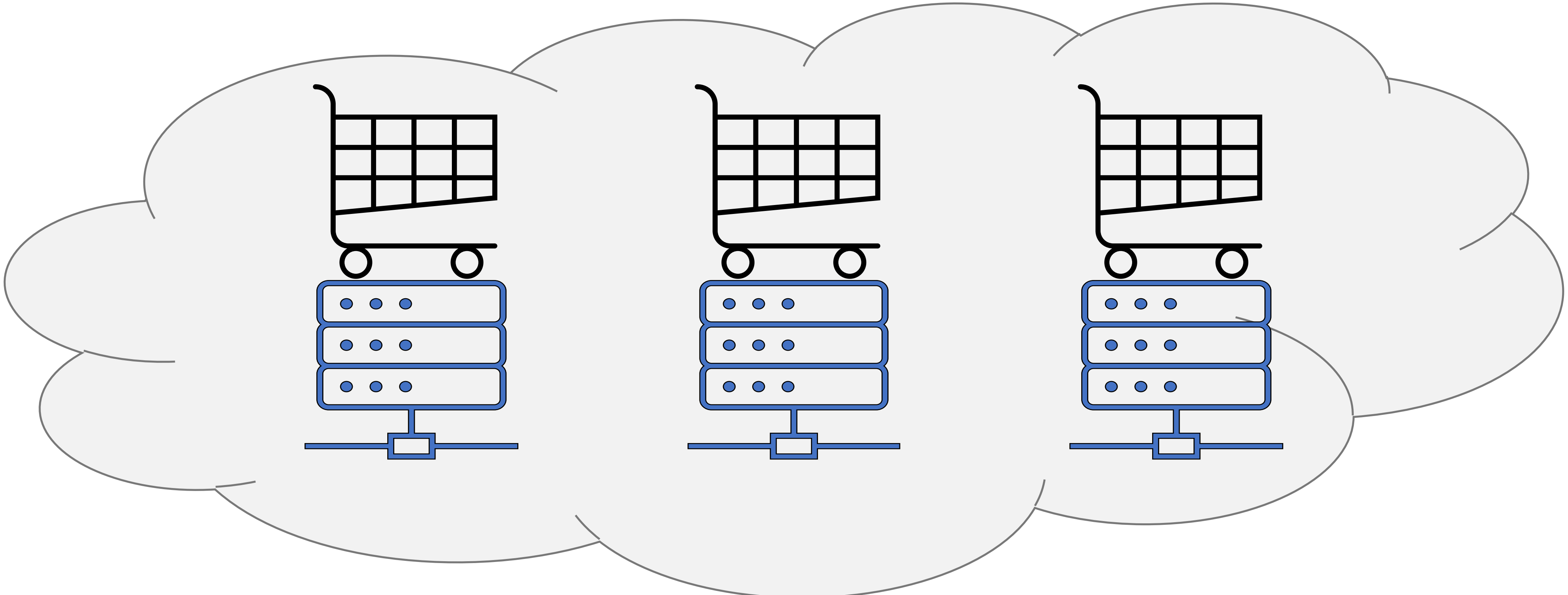
Streams need **same origin** to merge!

Enforce with a type system!

Leveraging lattices: a shopping cart reborn

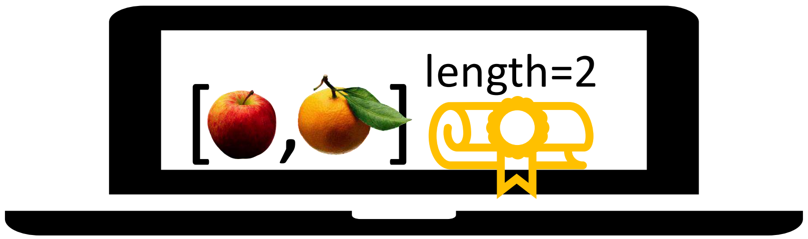
Manifesting Checkout

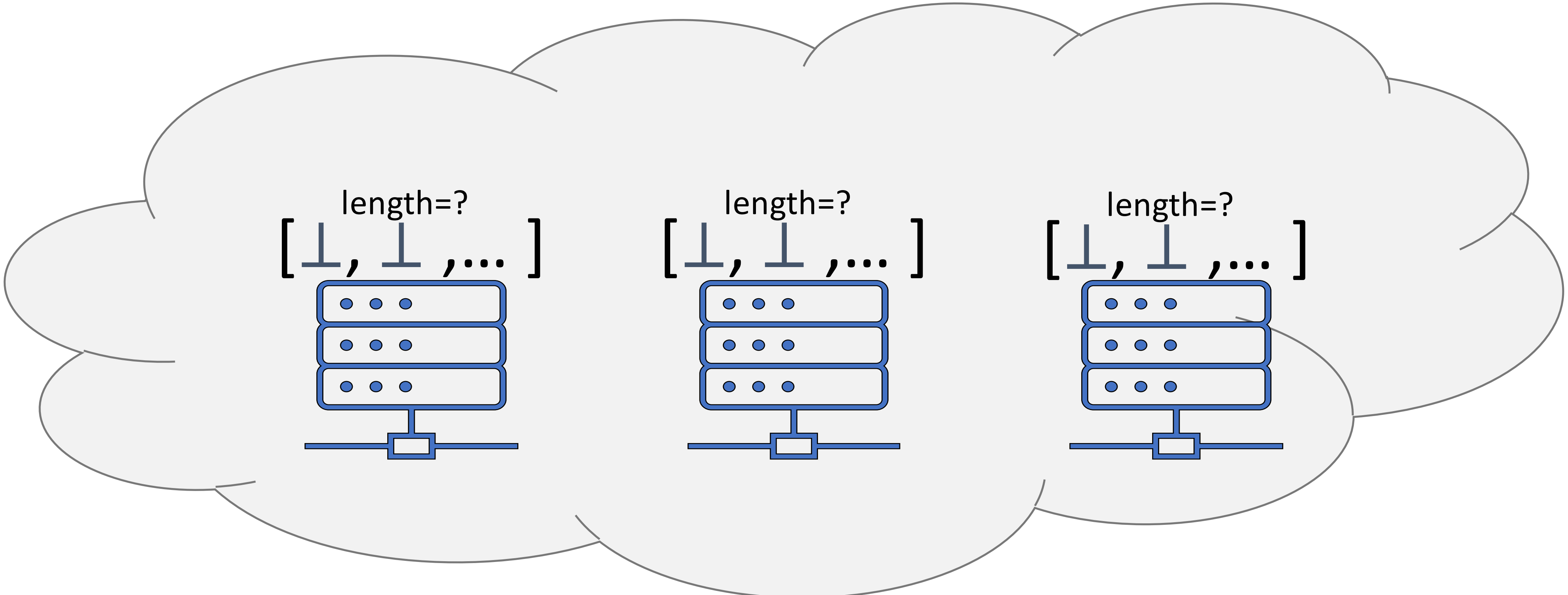
= manifest



Manifesting Checkout

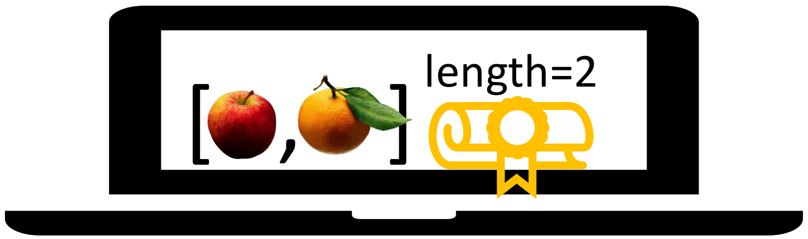
 = manifest

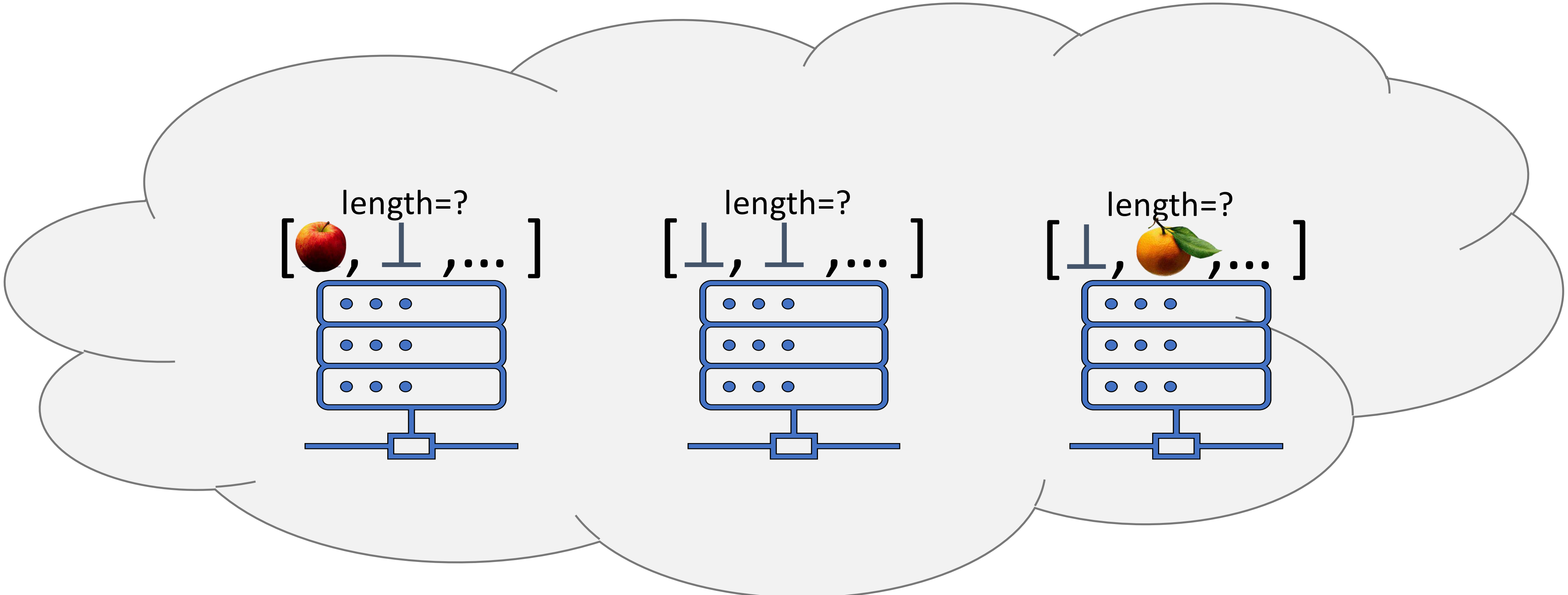




Manifesting Checkout

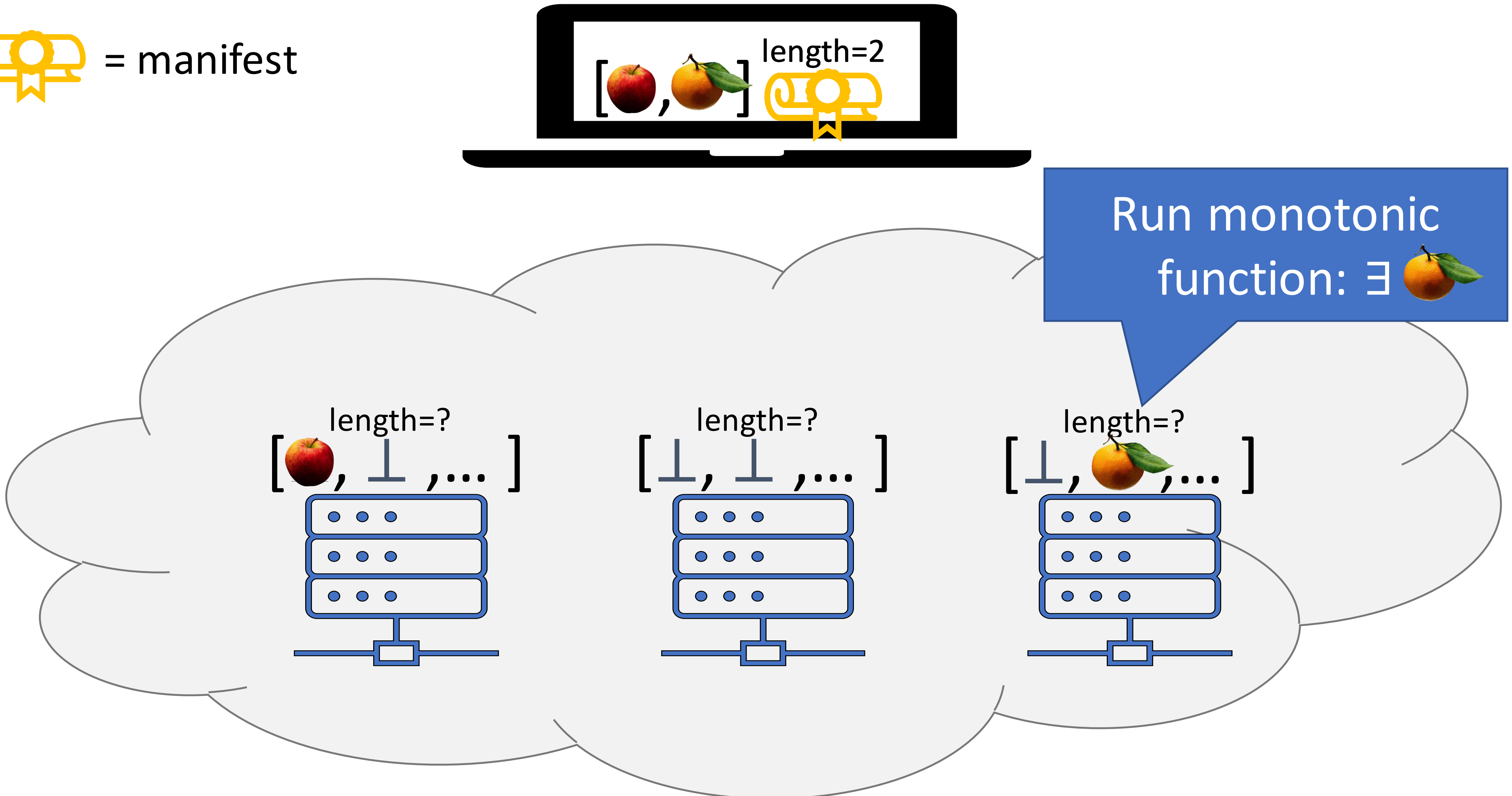
 = manifest



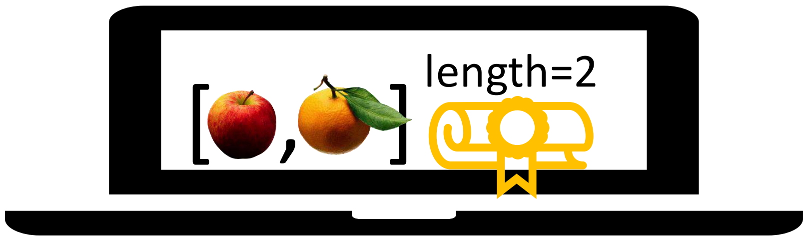


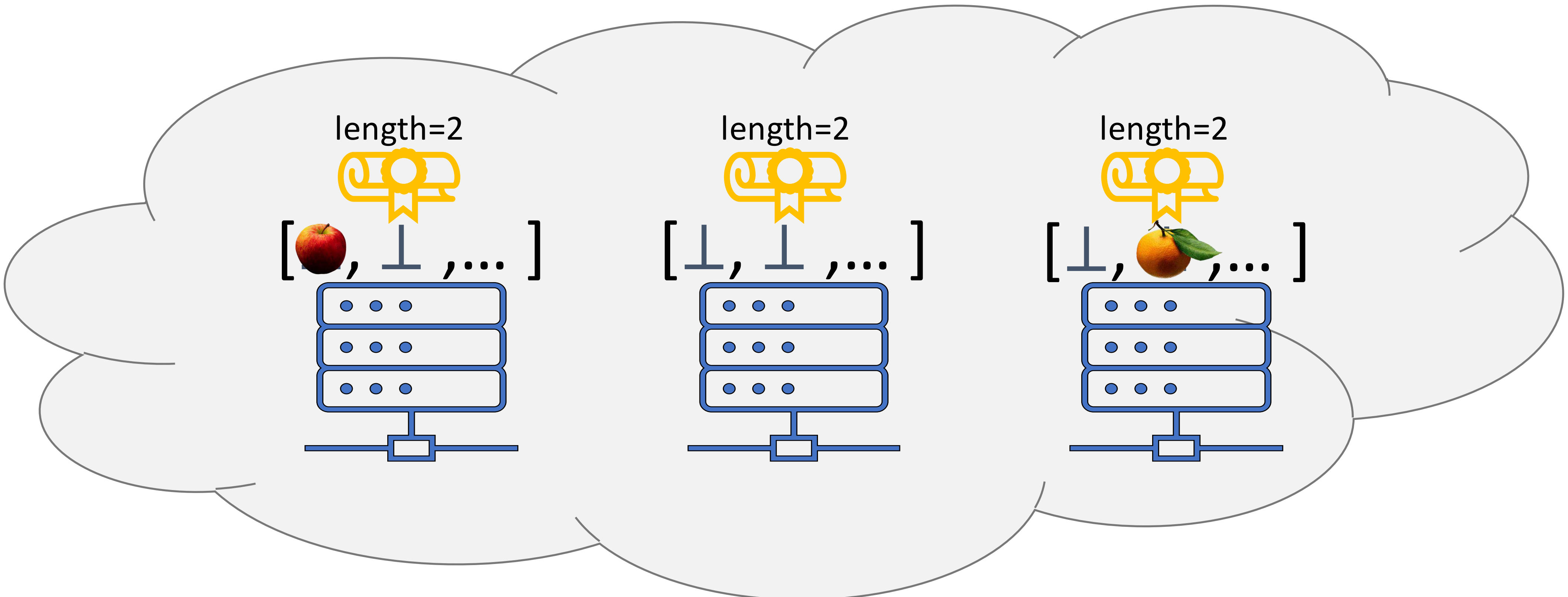
Manifesting Checkout

= manifest



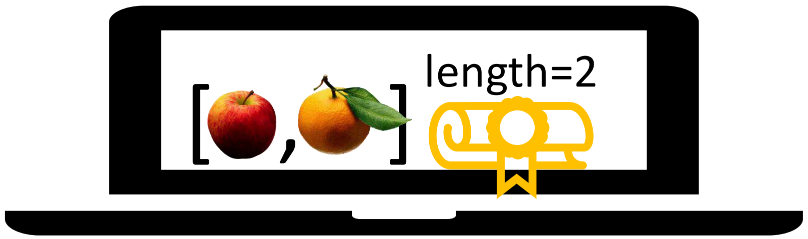
Manifesting Checkout

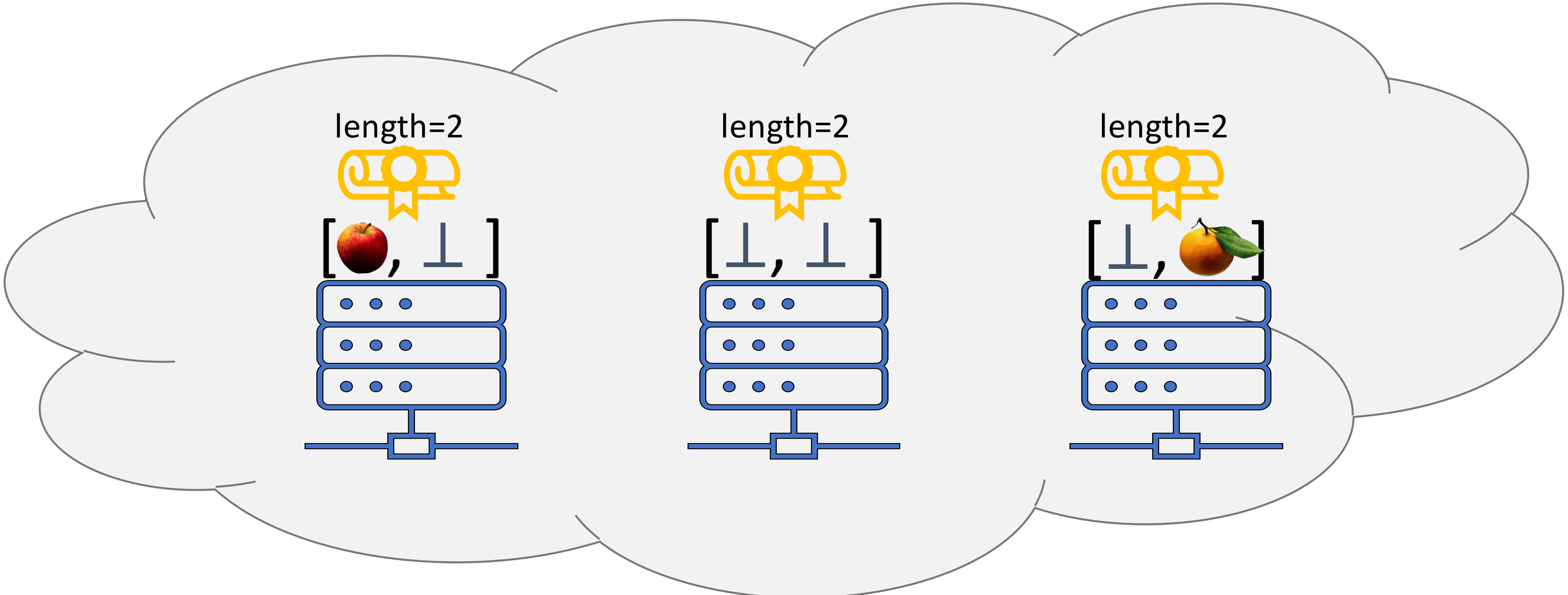




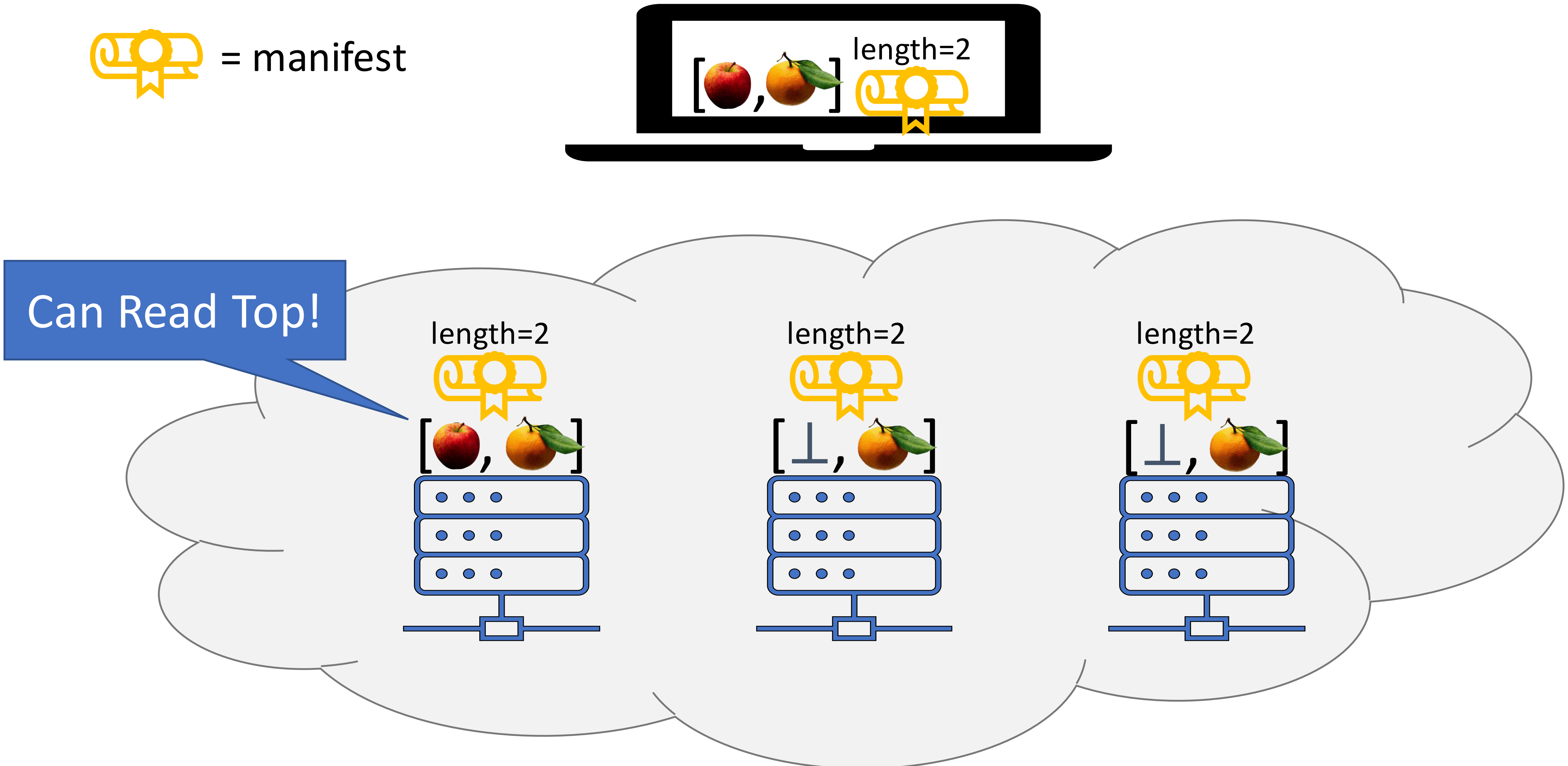
Manifesting Checkout

 = manifest



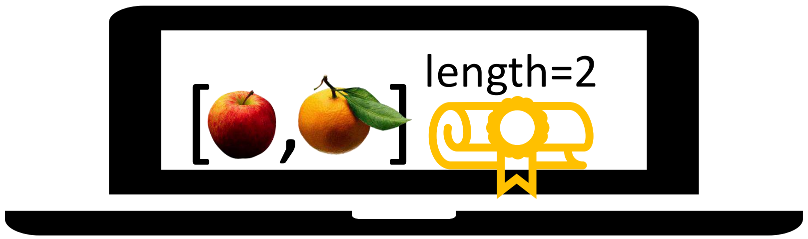


Manifesting Checkout

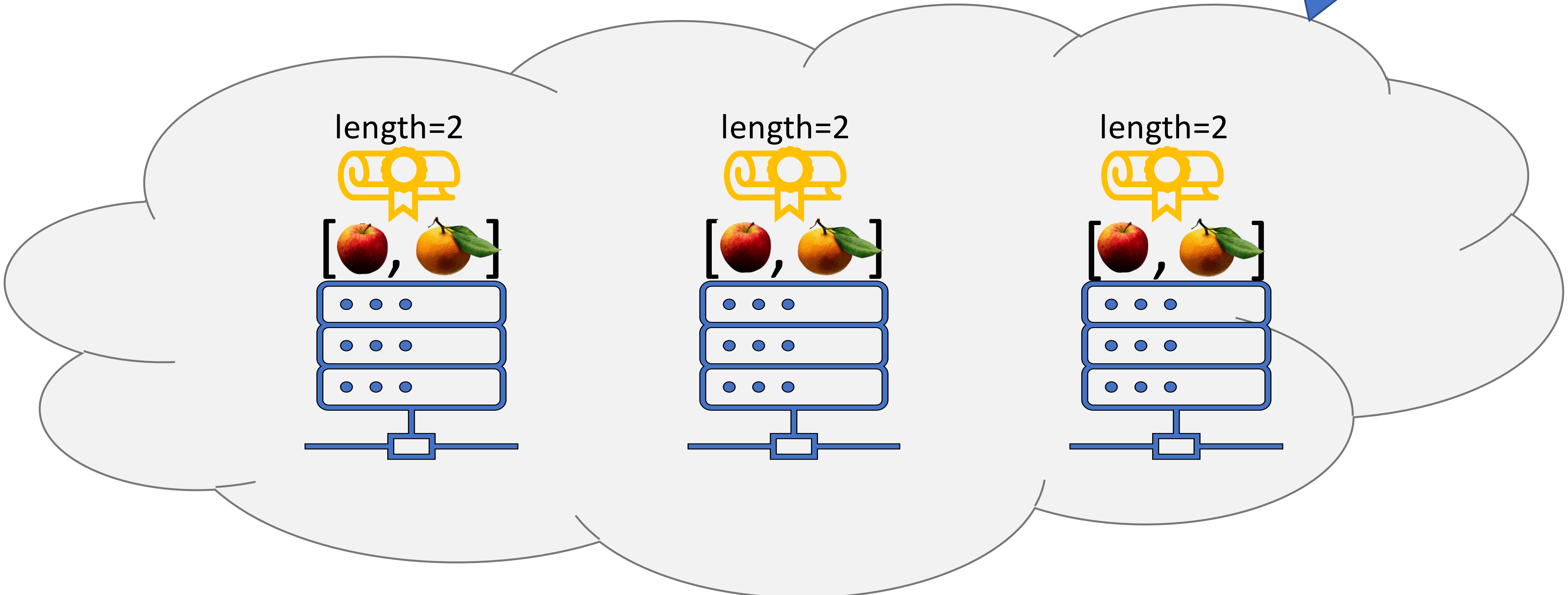


Manifesting Checkout

 = manifest

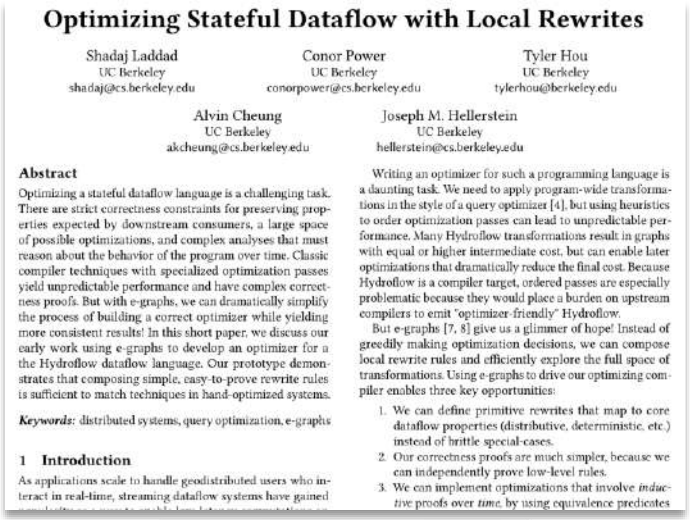


Top Everywhere!
Checkout at will



Use it Today!

- Rust library implementing **all lattices described here**
- Embedded DSL correctly implements ⊔, ⊓ on sequential processes!
- Beats previous record-setting performance!
- Checker only, **optimizer not ready**
- Check out our **e-graphs paper** for optimization story!



Optimizing Stateful Dataflow with Local Rewrites

Shadaj Laddad, UC Berkeley, shadaj@cs.berkeley.edu
Conor Power, UC Berkeley, conorpower@cs.berkeley.edu
Tyler Hou, UC Berkeley, tylerhou@berkeley.edu
Alvin Cheung, UC Berkeley, akcheung@cs.berkeley.edu
Joseph M. Hellerstein, UC Berkeley, hellerstein@cs.berkeley.edu

Abstract

Optimizing a stateful dataflow language is a challenging task. There are strict correctness constraints for preserving properties expected by downstream consumers, a large space of possible optimizations, and complex analyses that must reason about the behavior of the program over time. Classic compiler techniques with specialized optimization passes yield unpredictable performance and have complex correctness proofs. But with e-graphs, we can dramatically simplify the process of building a correct optimizer while yielding more consistent results! In this short paper, we discuss our early work using e-graphs to develop an optimizer for a the Hydroflow dataflow language. Our prototype demonstrates that composing simple, easy-to-prove rewrite rules is sufficient to match techniques in hand-optimized systems.

Keywords: distributed systems, query optimization, e-graphs

1 Introduction

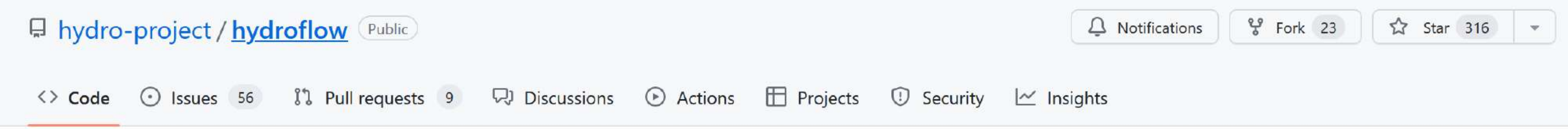
As applications scale to handle geodistributed users who interact in real-time, streaming dataflow systems have gained

Writing an optimizer for such a programming language is a daunting task. We need to apply program-wide transformations in the style of a query optimizer [4], but using heuristics to order optimization passes can lead to unpredictable performance. Many Hydroflow transformations result in graphs with equal or higher intermediate cost, but can enable later optimizations that dramatically reduce the final cost. Because Hydroflow is a compiler target, ordered passes are especially problematic because they would place a burden on upstream compilers to emit "optimizer-friendly" Hydroflow.

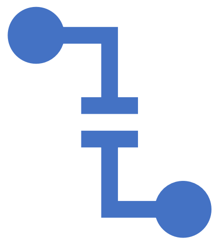
But e-graphs [7, 8] give us a glimmer of hope! Instead of greedily making optimization decisions, we can compose local rewrite rules and efficiently explore the full space of transformations. Using e-graphs to drive our optimizing compiler enables three key opportunities:

1. We can define primitive rewrites that map to core dataflow properties (distributive, deterministic, etc.) instead of brittle special-cases.
2. Our correctness proofs are much simpler, because we can independently prove low-level rules.
3. We can implement optimizations that involve *inductive* proofs over *time*, by using equivalence predicates

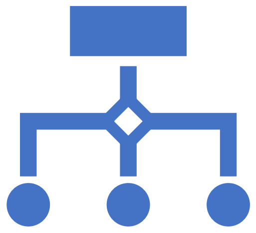
Shadaj Laddad



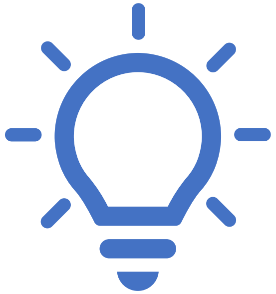
Monotonicity for all



A *well-typed* **lattice** interpretation of **streams**



compositional reasoning over both **weakly-** and **strongly-**synchronized replicas



Sequential code is a *special case* of **monotonicity**


Thank you!