Hydro: A Data-Centric Compiler Stack for the Cloud

NorCalDB 2023

JOE HELLERSTEIN
UC BERKELEY
SUTTER HILL VENTURES
View from the top (of the I-80 Bay Bridge HOV lane)
Sea Changes in Computing

- PDP-11, 1970
- Cray-1, 1976
- Supercomputers
- Macintosh, 1984
- Personal Computers
- iPhone, 2007
- Smart Phones
New Platform + New Language = Innovation

PDP-11, 1970

Cray-1, 1976

Minicomputers

Supercomputers

Cray

MPI

Personal Computers

Macintosh, 1984

HyperCard

LabVIEW

iPhone, 2007

Smart Phones

Swift

Android

THE
C
PROGRAMMING
LANGUAGE

iPhone, 2007

Macintosh, 1984

HyperCard

LabVIEW

Swift

Android
The Big Question

How will folks program the cloud?

- In a way that fosters unexpected innovation
- Distributed programming is hard!
  - Parallelism, consistency, partial failure, …
- Autoscaling makes it harder!

Programming the Cloud: A Grand Challenge for Computing
Serverless Computing

State & Coordination

The CALM Theorem

Hydro: Programming the Cloud
FaaS Expectation: a boundless programmable cloud.

FaaS Reality: an elastic army of incommunicado amnesiacs
Serverless Function Limitations

Reality:

✅ Boundless *single-node* compute
❌ No network messages (no *distributed* computation!)
❌ No low-latency data access
❌ Processes reboot every few minutes (amnesia)

Serverless Computing: One Step Forward, Two Steps Back

Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov and Chenggang Wu
UC Berkeley
(hellerstein,jmfaleiro,jegonzal,jsmith,vikrams,atumanov,gwu)@berkeley.edu

ABSTRACT
Serverless computing offers the potential to program the cloud in an autoscaling, pay-as-you-go manner. In this paper we address critical gaps in first-generation serverless computing, which place its autoscaling potential at odds with dominant trends in modern computing: notably data-centric and distributed computing, but also open source and custom hardware. Put together, these gaps make current serverless offerings a bad fit for cloud innovation and particularly bad for data systems innovation. In addition to pinpointing some of the main shortfalls of current serverless architectures, we raise a set of challenges we believe must be met to unlock the radical potential that the cloud—with its exabytes of storage and millions of cores—should offer to innovative developers.

1 INTRODUCTION
Amazon Web Services recently celebrated its 12th anniversary, marking over a decade of public cloud availability. While the cloud has been a place to timeshare machines, it was clear from the beginning...
Still...Serverless as a Leading Economic Indicator

- First 15-20 years: The Boring Revolution
  - Stealing business from legacy enterprise software vendors

- 2022: The Establishment has incentives to grow
  - Expose the platform, foster innovation!

- Infrastructure folks need to roll up our sleeves
  - Say Yes to the hard challenges!
Good News: long lead time for research!

Cloud Programming: From Doom and Gloom to BOOM and Bloom

Neil Conway
UC Berkeley

Joint work with Peter Alvaro, Ras Bodik, Tyson Condie, Joseph M. Hellerstein, David Maier (PSU), William R. Marczak, and Russell Sears (Yahoo! Research)

Datalog 2.0 Workshop

The Declarative Imperative
Experiences and Conjectures in Distributed Logic

Joseph M. Hellerstein Berkeley

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley

Joint work with: Peter Alvaro, Peter Bailis, David Maier, Bill Marczak, Joe Hellerstein, Siriram Srinivasan

Basho Chats #004
June 27, 2012

<~ bloom

Disorderly programming for a distributed world

Peter Alvaro
UC Berkeley
Rolling Up our Sleeves

General-Purpose Cloud Programming

Three main goals:

1. **Simplicity**: easy to learn, debug, operate
2. **Correctness**: make programs work as intended
3. **Dynamic Cost/Performance**: Run fast, consume only the resources needed
Toward Generality: Embracing State

- Program State: Local data that is managed across invocations
  - What’s so hard about State?

Challenge 1: Distributed Consistency
  - This correctness and simplicity problem is difficult and unavoidable!

Challenge 2: Data Placement and Movement
  - This performance problem is a much more traditional challenge.
The Challenge: Consistency

- Ensure that distant agents agree (or will agree) on common knowledge.
- Classic example: data replication
  - How do we know if they agree on the value of a mutable variable x?
The Challenge: Consistency

- Ensure that distant agents agree (or will agree) on common knowledge.
- Classic example: data replication
  - How do we know if they agree on the value of a mutable variable x?
  - If they disagree now, what could happen later?

- Split Brain divergence!
Classical Consistency Mechanisms: Coordination

- Consensus (Paxos, etc), Commit (Two-Phase Commit, etc)

TRICKY STUFF!

ALSO BAD...
Coordination Avoidance (a poem)

the first principle of successful scalability is

to batter the consistency mechanisms down to a minimum
move them off the critical path
hide them in a rarely visited corner of the system, and then

make it as hard as possible
for application developers

to get permission to use them

—James Hamilton (IBM, MS, Amazon)
### Generational Shift to Reasoning at the App Level

<table>
<thead>
<tr>
<th>20th Century</th>
<th>21st Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write</td>
<td>Immutable State</td>
</tr>
<tr>
<td>Access/Store</td>
<td>Dataflow Dependencies</td>
</tr>
<tr>
<td>Linearizability</td>
<td>Pure Functions</td>
</tr>
<tr>
<td>Serializability</td>
<td>Monotonic Logic</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

*worst-case assumptions*

*app-specific assumptions*

**Tired:** Reasoning about memory access

**Wired:** Reasoning about App Semantics
Big Queries: When? Why?

- When do I *need* Coordination?
- Why?
  - Conflicts, race conditions in space & time, etc.
Big Queries: When? Why?

- When do I *need* Coordination?
- Why?
- No really: Why?
  - When is Coordination *required*?
Suppose you understand your program's semantics...

- Which programs have a coordination-free implementation?
- Which programs require coordination?

A question of computability!
Serverless Computing

- Expectations vs Reality
- Indicators of a brighter future for cloud programming

State & Coordination

- Consistency
- Avoiding Coordination
- Semantics to the rescue?

The CALM Theorem

- Monotonicity: “bright line” test for whether coordination is needed.
- Lessons and performance payoffs

Hydro: Programming the Cloud
CALM: CONSISTENCY AS LOGICAL MONOTONICITY

Theorem (CALM): A distributed program has a consistent, coordination-free distributed implementation if and only if it is monotonic.

Hellerstein JM. The Declarative Imperative: Experiences and conjectures in distributed logic. 
*ACM PODS* Keynote, June 2010

*ACM SIGMOD Record*, Sep 2010.

Ameloot TJ, Neven F, Van den Bussche J. Relational transducers for declarative networking.
*JACM*, Apr 2013.

Ameloot TJ, Ketsman B, Neven F, Zinn D. Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the CALM-conjecture.


Hellerstein, JM, Alvaro, P. Keeping CALM: When Distributed Consistency is Easy.
In distributed systems theory, CALM presents a result that delineates the frontier of the possible.

BY JOSEPH M. HELLERSTEIN AND PETER ALVARO

Keeping CALM: When Distributed Consistency Is Easy

Distributed systems are tricky. Multiple unreliable
across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

The challenge of building correct distributed systems is increasingly urgent, but it is not new. One traditional answer has been to reduce this complexity with memory consistency guarantees—assurances that accesses to memory (heap variables, database keys, and so on) occur in a controlled fashion. However, the mechanisms used to enforce these guarantees—coordination protocols—are often criticized as barriers to high performance, scale, and availability of distributed systems.

The high cost of coordination. Coordination protocols enable autonomous, loosely coupled machines to jointly decide how to control basic behaviors, including the order of access to shared memory. These protocols are among the most clever and widely cited ideas in distributed computing. Some well-known techniques include the Paxos and Two-Phase Commit (2PC) protocols, and global barriers underly-

key insights

- Coordination is often a limiting factor in system performance. While sometimes necessary for consistent outcomes,
We’ll need some definitions

consistency  mononicity

coordination
Intuitively…

Consistency: *same app outcome everywhere* regardless of NW shenanigans.

Monotonicity: The *outcomes only grow* as inputs grow. Early outcomes are part of the final outcome! Stream without regret!

Coordination: Messages we *must wait for even though we have all the data*.

\[ f \text{ monotone} \iff X \subseteq Y \rightarrow f(X) \subseteq f(Y) \]

**CALM**

Monotonicity $\iff$ Coordination-free Consistency
Easy and Hard Questions

Is anyone over 18?  Who is the youngest?
Easy and Hard Questions

Is anyone over 18?

\( \exists x \ x > 18 \)

Who is the youngest?

\( \exists x \forall y \ (x < y) \)
Easy and Hard Questions

Is anyone over 18?

∃x \ x > 18

Who is the person nobody is younger than?

∃x \ ∀y (y < x)
Serverless Computing

- Expectations vs Reality
- Indicators of a brighter future for cloud programming

State & Coordination

- Consistency
- Avoiding Coordination
- Semantics to the rescue?

The CALM Theorem

- Monotonicity: “bright line” test for whether coordination is needed.

Hydro: Programming the Cloud
A CALM Look Back at CAP

- You may be familiar with Brewer’s CAP Theorem
  - “Only get 2 of 3”: Consistency, Availability, and Partitioning
  - Unspoken assumption: Consistency that requires Coordination*
A CALM Look Back at CAP

You may be familiar with Brewer’s CAP Theorem

- “Only get 2 of 3”: Consistency, Availability, and Partitioning
- Unspoken assumption: Consistency that requires Coordination*

The CALM Theorem explains *when* and *why* we can get all 3!

- Coordination-free Consistency* is possible even if we’re Available under Partitioning!
- CALM: Monotonicity $\Leftrightarrow$ Coordination-free Consistency
- Only the non-monotone programs are subject to CAP’s negative result

*This is at the heart of technical differences in the formal proofs. Brewer and I are on the same page here.*
Design Patterns from CALM

- The **Anna** “Any Scale” KVS
  - Many choices of consistency guarantees: causal, read-committed, etc.
  - No coordination! Full-tilt parallelism and autoscaling

- The **Cloudburst** Stateful Serverless system
  - Corollary from Ameloot: Monotonic programs can be “oblivious” to membership
  - “Statelessness” is just degenerate monotonicity


Chenggang Wu
Vikram Sreekanti

https://aqueducthq.com
Lessons from Staying Monotonic

Update anywhere, copy at will!

- Maximizes system “Goodput”
  - Even under contention!

- Profligate Data Replication:
  - Horizontally (to peers): for fault tolerance, load balancing, latency
  - Vertically (to faster caches, slower storage): pay more $$ only for hot data
Serverless Computing

- Expectations vs Reality
- Indicators of a brighter future for cloud programming

State & Coordination

- Consistency
- Avoiding Coordination
- Semantics to the rescue?

The CALM Theorem

- Monotonicity: “bright line” test for whether coordination is needed.
- Lessons and performance payoffs

Hydro: Programming the Cloud
What More Could We Want?

Mix of monotone and non-monotone stuff. E.g.

- Strong consistency (transactions)
- Termination detection
- Computationally complex (PTIME) distributed algorithms
What More Could We Want?

- Mix of monotone *and* non-monotone stuff.
- A language/compiler/debugger that addresses distributed concerns!
What More Could We Want?

- Mix of monotone and non-monotone stuff.
- A language/compiler/debugger that addresses distributed concerns! E.g.:
  - Is my program consistent or will different machines disagree?
  - How can I partition state safely?
  - How can I replicate state safely?
  - What failures can this tolerate and how many?
  - What data is going where and who can see it?
  - Tunable objective functions. Please optimize for:
    - $\$, not latency.
    - 99'th percentile, not 95'th
    - Etc.
What More Could We Want?

- Mix of monotone and non-monotone stuff.
- A language/compiler/debugger that addresses distributed concerns! E.g.:
  - Is my program consistent or will different machines disagree?
  - How can I partition state safely?
  - How can I replicate state safely?
  - What failures can this tolerate and how many?
  - What data is going where and who can see it?
  - Tunable objective functions. Please optimize for:
    - $$\$, not latency.
    - 99'th percentile, not 95'th
    - Etc.
What More Could We Want?

- Mix of monotone and non-monotone stuff.

A language/compiler/debugger that addresses distributed concerns! E.g.:

- Is my program consistent or will different machines disagree?

- How can I partition state safely?

- How can I replicate state safely?

- What failures can this tolerate and how many?

- What data is going where and who can see it?

- Tunable objective functions. Please optimize for:
  - $$\$, not latency.
  - 99'th percentile, not 95th
  - Etc.
Inspiration is a Query Away: SQL!

The single biggest success story in autoparallelized programming? **SQL.**

- We’ve been doing it since the 1980’s! Gamma, Teradata, …

Relational queries scale like crazy

- Transactions do *not*. Focus on query languages, processing, optimization.

Relational databases were invented to hide how data is laid out and how queries are executed.
Inspiration is a Query Away: SQL!

- The single biggest success story in autoparallelized programming? **SQL**.
  - We’ve been doing it since the 1980’s! Gamma, Teradata, …

- Relational queries scale like crazy
  - Transactions do *not*. Focus on query languages, processing, optimization.

Relational databases were invented to hide how data is laid out and how queries are executed.

The cloud was invented to hide how computing resources are laid out and how general-purpose computations are executed.
Our First Approach: Relational, Data-Centric Programming

- Build on the success and lessons of SQL
- From Datalog to Dedalus and Bloom
- “Datalog in time and space”
- We demonstrated many of the benefits in prototypes

---

**DEDALUS: Datalog in Time and Space**

Peter Alvaredo1, William R. Marczan1, Neil Conway1, Joseph M. Hellerstein1, David Maier2, and Russell Sears3

1 University of California, Berkeley
2 Portland State University
3 Yahoo! Research

**ABSTRACT**

Datalog in time and space is a temporal, data-centric programming language that can be used to build applications that operate over databases with changing, non-stationary data. We demonstrate the benefits of Datalog in time and space in a number of case studies, including a news-aggregation application and an application for analyzing traffic patterns. We show how Datalog in time and space can be used to build applications that are more efficient and more flexible than those built using traditional relational databases.


---

**Consistency Analysis in Bloom: a CALM and Collected Approach**

Peter Alvaredo, Neil Conway, Joseph M. Hellerstein, William R. Marczan

**ABSTRACT**

Datalog in time and space is a temporal, data-centric programming language that can be used to build applications that operate over databases with changing, non-stationary data. We demonstrate the benefits of Datalog in time and space in a number of case studies, including a news-aggregation application and an application for analyzing traffic patterns. We show how Datalog in time and space can be used to build applications that are more efficient and more flexible than those built using traditional relational databases.


---

**Logic and Lattices for Distributed Programming**

Neil Conway
U.C. Berkeley
moc@cs.berkeley.edu

William R. Marczan
U.C. Berkeley
wrm@cs.berkeley.edu

Joseph M. Hellerstein
U.C. Berkeley
hellerstein@cs.berkeley.edu

Peter Alvaredo
U.C. Berkeley
palaric@cs.berkeley.edu

David Maier
Portland State University
maier@pdx.edu

**ABSTRACT**

In recent years there has been interest in applying distributed systems techniques to build large-scale, distributed systems. However, building systems that are both scalable and reliable is challenging. We propose a new approach to building distributed systems, based on the use of lattices and modal logics. This approach allows us to reason about system behavior in a way that is both more powerful and more flexible than previous approaches.


---

Our First Approach: Relational, Data-Centric Programming

- Build on the success and lessons of SQL
- From Datalog to Dedalus and Bloom
- “Datalog in time and space”
- We demonstrated many of the benefits in prototypes

---


Our First Approach: Relational, Data-Centric Programming

- Build on the success and lessons of SQL
- From Datalog to Dedalus and Bloom
  - “Datalog in time and space”
  - We demonstrated many of the benefits in prototypes
- A walled garden
  - Did not focus on incremental adoption, integration or performance
- Also, many common constructs are clumsy in relational languages
  - E.g. counters are commonly used in distributed systems
  - E.g. ordered data

https://bloom-lang.net
Competing Approach: Semi-Lattices

CRDTs: Easy for programmers to understand and embrace

- An object class
- With an A.C.I. “merge” function
- I.e. a semi-lattice

CRDT

Associative + Commutative + Idempotent

any batch size!
any order!
retry at will!
Competing Approach: Semi-Lattices

- Easy for programmers to understand and embrace
  - An object class
  - With an A.C.I. “merge” function
  - I.e. a semi-lattice

- Unfortunately, kind of broken: the Hotel California of distributed state
  - You can merge state any time you like but … you can never Read (safely)!

- Still … we’re starting to see adoption and impact
Hydro largely inspired by:

- Logic languages (SQL, Datalog, Dedalus, Bloom)
- Analytic power for correctness and optimization
- Lattice Compositions (CRDTs)
- Generalize relations to other data types
- Functional Dataflow (Spark, Flink, Timely)
- Focus on low latency!
- Compiler Infrastructure stacks (llvm, Halide)
- Multiple languages, targeting both generalists and experts

New Directions in Cloud Programming

Alvin Cheung  Natacha Crooks  Joseph M. Hellerstein  Matthew Milano
(scheung,ncrooks,hellerstein,mpmilano)@berkeley.edu
UC Berkeley

ABSTRACT
Nearly twenty years after the launch of AWS, it remains difficult for most developers to harness the enormous potential of the cloud. In this paper we lay out an agenda for a new generation of cloud programming research aimed at bringing research ideas to programmers in an evolutionary fashion. Key to our approach is a separation of distributed programs into a PACT of four facets: Program semantics, Availability, Consistency and Targets of optimization. We propose to migrate developers gradually to PACT programming by lifting familiar code into our more declarative level of abstraction. We then propose a multi-stage compiler that emits human-readable code at each stage that can be hand-tuned by developers.

is a low-level assembly language for the cloud, a simple infrastructure for launching sequential code, a UDF framework without a programming model to host it. As cloud programming matures, it seems inevitable that it will depart from traditional sequential programming. The cloud is a massive, globe-spanning distributed computer made up of heterogeneous multicore machines. Parallelism abounds at all scales, and the distributed systems challenges of non-deterministic network interleavings and partial failures exist at most of those scales. Creative programmers are held back by the need to account for these complexities using legacy sequential programming models originally designed for single-processor machines.

HYDRO: A Programming Stack for the Cloud

- Many languages for programmers
- Declarative IR
- Low-Level “Lattice-flow”
- Adaptive self-deploying exe’s

Diagram:
- HYDROLOGIC
- HYDRAULIC
- HYDROLYSIS
- HYDROFLOW
- HYDRO全国人民

- New DSLs
- Availability
- Consistency
- Trust
- Program Semantics

- Futures (e.g. Ray)
- Actors (e.g. Orleans)
- Functional (e.g. Spark)
- Logic (e.g. Bloom)
- Sequential Code

- FaaS
- Storage
- ML Frameworks

- Cloud Services
Initial wins from Hydro

- **Hydroflow**: High performance Rust kernel for distributed execution
  - Flows of lattice compositions, with typechecking for distributed properties induced via Rust
Original Anna KVS. **C++**
2018 Amazon m4.16xlarge instances
(64 vCPU, 256GB RAM.)

Anna KVS. **Hydro**
2023 GCP n2-standard-64 instances
(64 vCPU, 256GB RAM)
Initial wins from Hydro

- Hydroflow: High performance Rust kernel for distributed execution
- Compiler optimizations for auto-distribution, auto-scaling
  - Coordination-free replication: CALM analysis. (“Most programs are mostly monotone”)
  - Auto-sharding: Functional Dependency analysis
  - With compiler guarantees of correctness!

---

SOSP 2021

THE ACM SIGOPS SOSP 2021
STUDENT RESEARCH COMPETITION
GRADUATE STUDENT WINNER AWARD

is presented to

David Chu (UC Berkeley)

for their work on

Automatic Compartamentalization of Distributed Protocols

David Chu
Initial wins from Hydro

- Hydroflow: High performance Rust kernel for distributed execution
- Compiler optimizations for auto-distribution, auto-scaling
- Algebraic Upgrade: Idempotence enforcement in $O(1)$ memory
  - Via agg-tree, flooding and lattices

---

TopoloTree: From $O(n)$ to $O(1)$ CRDT Memory Consumption Via Aggregation Tree Gossip Topologies

Conor Power  
UC Berkeley  
conorpower@cs.berkeley.edu

Shadaj Laddad  
UC Berkeley  
shadaj@cs.berkeley.edu

Chris Douglas  
UC Berkeley  
chris_douglas@cs.berkeley.edu

Joseph M. Hellerstein  
UC Berkeley  
hellerstein@cs.berkeley.edu

Dan Suciu  
University of Washington  
suciu@cs.washington.edu

Abstract

Many CRDTs such as PN-Counters appear to require vector-clock-like state representations, incurring $O(n)$ memory consumption where $n$ is the number of replicas in the system. We show that by customizing the communication topology into an aggregation tree structure, the memory consumption can be reduced to a small constant. This solution is effective for CRDTs, but is generic to replicated data types with commutative updates that must tolerate gossip duplication.

PaPoC 2023
Initial wins from Hydro

- Hydroflow: High performance Rust kernel for distributed execution
- Compiler optimizations for auto-distribution, auto-scaling
- Algebraic Upgrade: Idempotence enforcement in O(1) memory
- Semi-automatic conversion of sequential to distributed code
  - Via Program Synthesis

Katara: Synthesizing CRDTs with Verified Lifting

SHADAJ LADDAD, University of California, Berkeley, USA
CONOR POWER, University of California, Berkeley, USA
MAE MILANO, University of California, Berkeley, USA
ALVIN CHEUNG, University of California, Berkeley, USA
JOSEPH M. HELLERSTEIN, University of California, Berkeley, USA

Conflict-free replicated data types (CRDTs) are a promising tool for designing scalable, coordination-free distributed systems. However, constructing correct CRDTs is difficult, posing a challenge for even seasoned developers. As a result, CRDT development is still largely the domain of academics, with new designs often awaiting peer review and a manual proof of correctness. In this paper, we present Katara, a program synthesis-based system that takes sequential data type implementations and automatically synthesizes verified CRDT designs from them. Key to this process is a new formal definition of CRDT correctness that combines a reference sequential type with a lightweight ordering constraint that resolves conflicts between non-commutative operations. Our process follows the tradition of work in verified lifting, including an encoding

Upcoming Research Directions

Multi-layer Dataflow Optimizers
Shadaj Laddad

Automatic Algebraic Upgrades for Correct Distributed Code
Conor Power

Fault Tolerance, Object Stores
Chris Douglas

Distributed Deployment & Load Balancing
Dr. Tiemo Bang

Protocol Compilers
David Chu

How do cloud-hosted distributed systems compose?
Also, pick a job!
Dr. Mae Milano

And…the ubiquitous LLM questions.
Embrace State, Avoid Coordination

Consistency for Apps
Not for I/Os

CALM Theorem
Monotonicity
⇔
Coordination-Free Consistency

Monotone Designs
The Anna KVS example

A New Language Stack
for the Cloud!

Programming the Cloud:
A Grand Challenge for Computing
Thank You!

Hydro Project:  https://hydro.run

Github:  https://github.com/hydro-project

hellerstein@berkeley.edu
@joe_hellerstein
Backup Slides
Gimme a “useful” example!

Twitter threads: growing trees

- Add a tweet (node) with edge to parent, tell the world
- Worst case: a tweet arrives before its parent. Fix locally.
  - **MONOTONE!** No coordination required. Handle updates in any order!

Database Log: totally ordered list

- Add a node with reference to predecessor, and tell the world
- Worst case: somebody else already added a node pointing to that predecessor!
  - **NON-MONOTONE!** Solution: coordinate so just one node “gets” to point to the predecessor
CALM Performance

- “Shared-nothing” at all scales (even across threads)
- Crazy fast under contention
  - Up to 700x faster than Masstree within a multicore machine
  - Up to 10x faster than Cassandra in a geo-distributed deployment
  - 350x the performance of DynamoDB for the same price
- CALM, coordination-free
  - No atomics, no locks, no waiting ever!
- Rich consistency (causal, read committed, …)
Stateful Serverless Computing

- Corollary to CALM (Ameloot et al 2011): Obliviousness!
  - Monotonic ⇔ Coordination-Free ⇔ “Oblivious”
  - Monotonic programs don’t need to know membership (or even identity!)

- Freedom to Scale:
  - **Up**: Add another node, copy state to it at any time
  - **Down**: Stop taking updates, copy state elsewhere and decommission at any time

- Shortcuts taken for FaaS aren’t necessary – Monotonicity for the win!