Microsoft® Research

Faculty Summit 2010

Microsoft® Research
Faculty Summit 2010

Cloud Programming

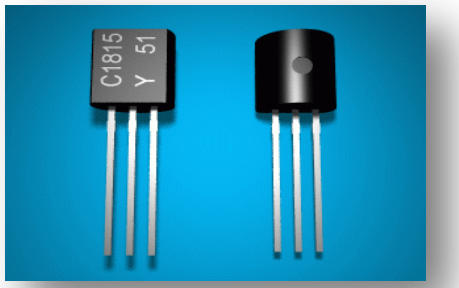
James Larus
Microsoft Research

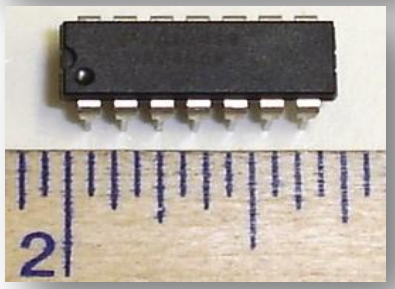
July 13, 2010

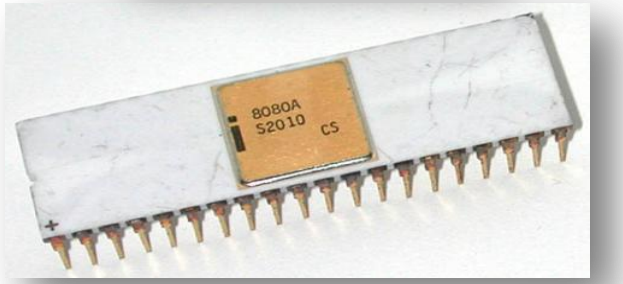
What is Cloud Computing?

- Next computing platform
  - Client + Cloud
  - All computing will be cloud computing
- Anything could be a client
  - Conventional: PCs + phone + TV
  - Single function: Kindle + car + appliance, ..
  - New HW enables killer apps
- Platform as a service is just a component
  - Amazon Web Services / Microsoft Azure / Google AppEngine
  - On-demand, hosted, internet computing resources
  - Commodification of distributed computing

Change Driven By New Computing Platforms

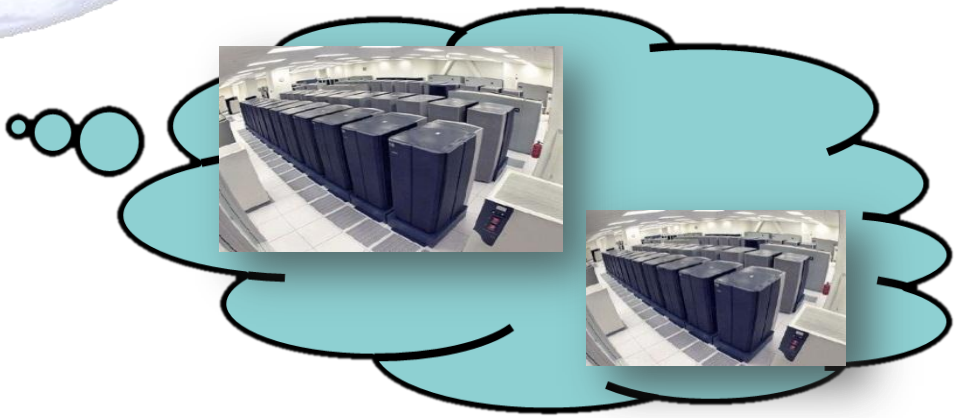






Cloud Computing

- Inherently distributed
- Wide range of clients (single purpose → rich)
- “Unlimited” computation and data
- Ubiquitous access to information and computation



Layers

Client SW

Client HW

App DC SW

DC OS

DC HW

dows Azure

Microsoft

Defining the future.

EXTREME COMPUTING GROUP

Scale is Challenging

- Service needs to handle the world
  - Hundreds of millions of users
- Continuously available
- Built on unreliable, commodity platform
- Make money

Microsoft's Datacenter Evolution

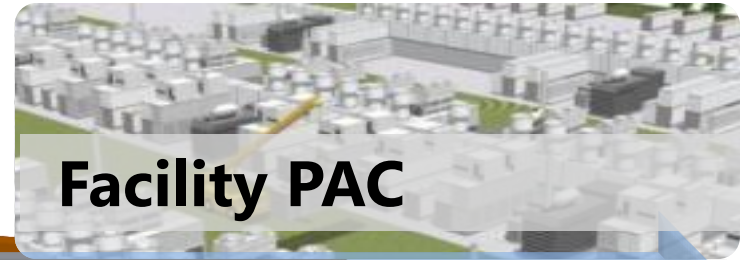
2005 2006 2007 2008 2009 2010

Generation 1
Datacenter Co-Location

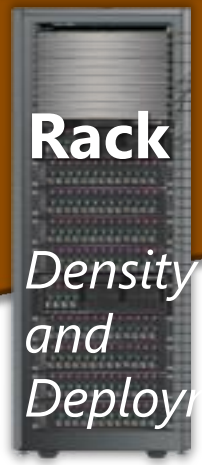
Generation 2
Quincy and San Antonio

Generation 3
Chicago and Dublin

Generation 4
Modular Datacenter



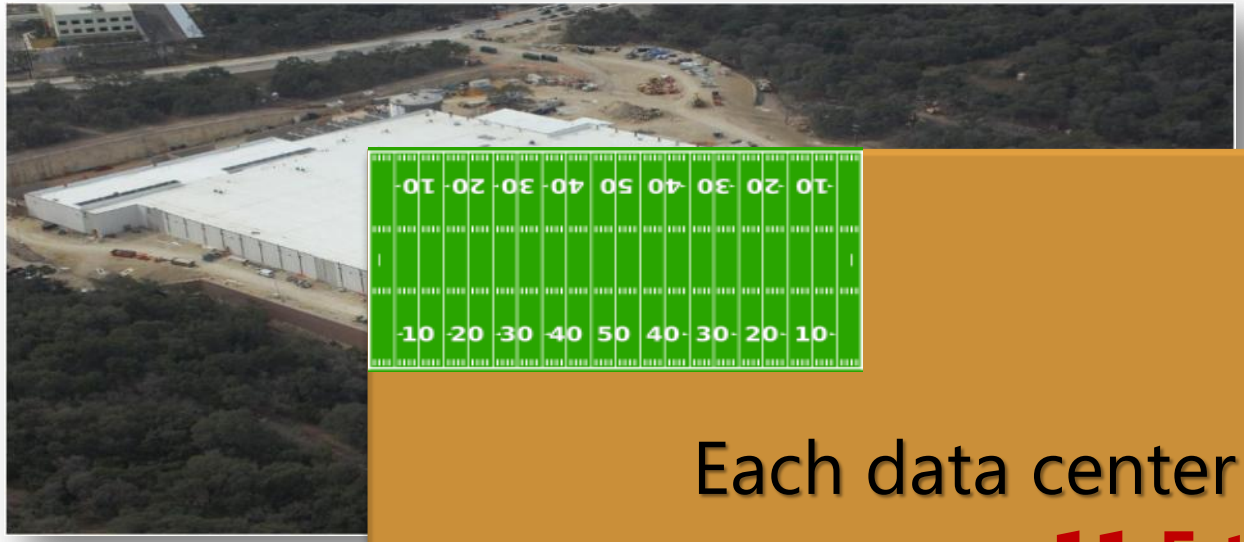
Deployment Scale Unit



Generation 2/3 – Data Centers

9

Generation 2/3 – Data Centers



Each data center is approximately
11.5 times
the size of a football field

10

Generation 3 - Chicago Data Center

Generation 3 - Chicago Data Center

$500M+ investment

1.5 million man-hours-of-labor

300(

707,

Data center is approximately
17 times
the size of a football field
and …
….Uses Containers

7.5 miles of chilled water piping

26,000 cubic yards of concrete

New Programming Model, New Problems (and some old, unsolved ones)

Concurrency

Parallelism

Message passing

Distribution

High availability

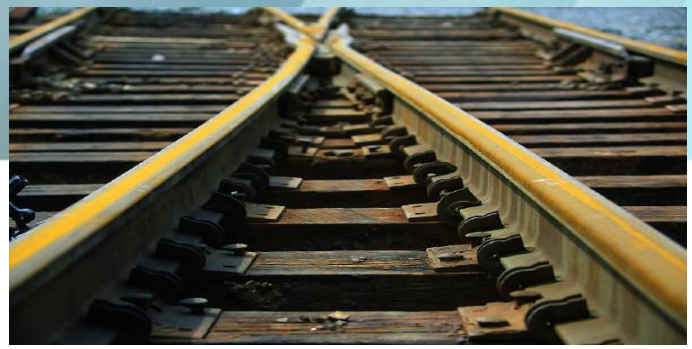
Performance

Application partitioning

Defect detection

High-level abstractions

Concurrency



- Inherently concurrent programming
  - Asynchronous, message-driven model
  - Multiple requests streams
- Threads or events??
  - Threads offer familiar sequential programming model
    - But, state can change when thread is preempted (synchronization)
    - Cost of thread and context switch limits concurrency
  - Handlers fracture program control flow
    - Logic split across event handlers
    - Explicit manipulation of local state (no stack frames)
- Higher-level (state machine, Actor, …) models?
- Lack of consensus inhibits research, development, reuse, interoperability
  - Parallel programming, redux

Parallelism

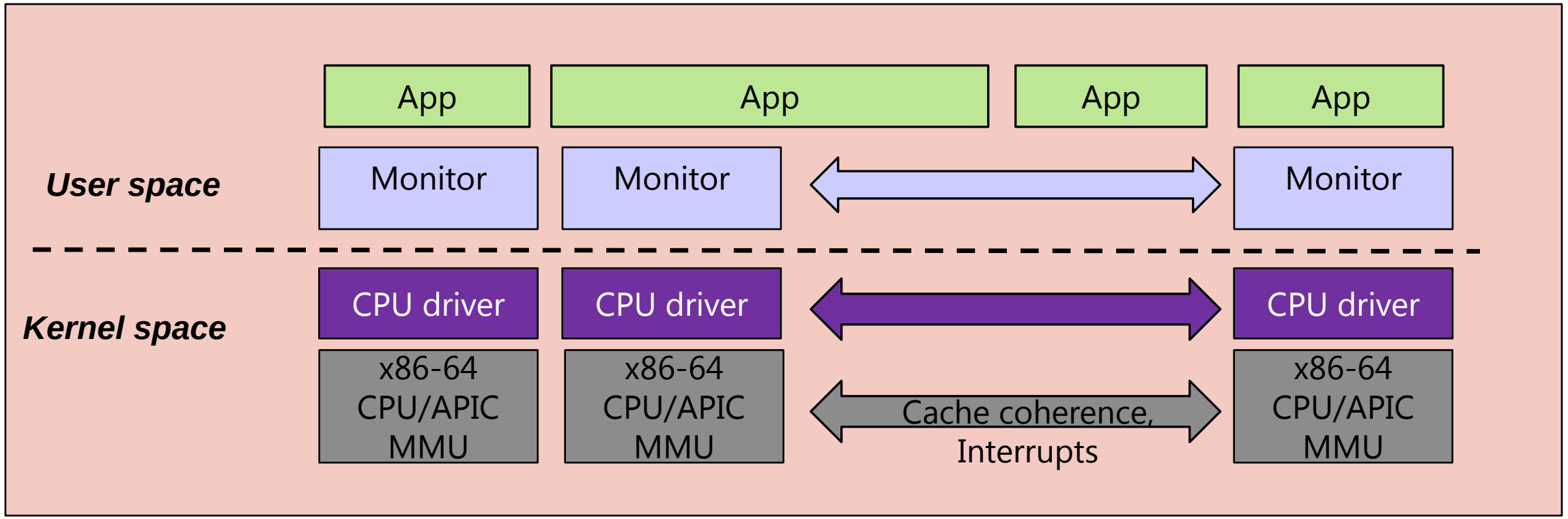
- Computers are parallel
  - Increased performance + power efficiency
- Computers will be heterogeneous
  - Multiple, non-isomorphic functional units
- Data centers are vast message-passing clusters
  - Availability and throughput
- Parallel programming is long-standing sore point for computer science
  - State of the art: threads and synchronization (assembly language)
  - No consensus on shared memory semantics
- New research on higher-level models is not panaceas
  - Transactional memory
  - Deterministic parallelism

- Radical proposal: abolish shared memory
  - Message passing is inherent in distributed systems, so why 2 models?
  - Shared memory is difficult and error prone

Message Passing

- Fundamental in distributed systems and better programming model
  - Performance / correctness isolation
  - Well-defined points of interaction
  - Scalable
- More difficult to use
  - Little language support
    - Erlang integrates message with pattern matching
    - Sing# channel contracts
    - Sing# postbox semantics
  - Message passing libraries
    - Fundamental mismatch: asynchronous strange in a synchronous world
- Open problems
  - Control structures for asynchronous messages
  - Communications contracts
  - Integration of messages in type system and memory model

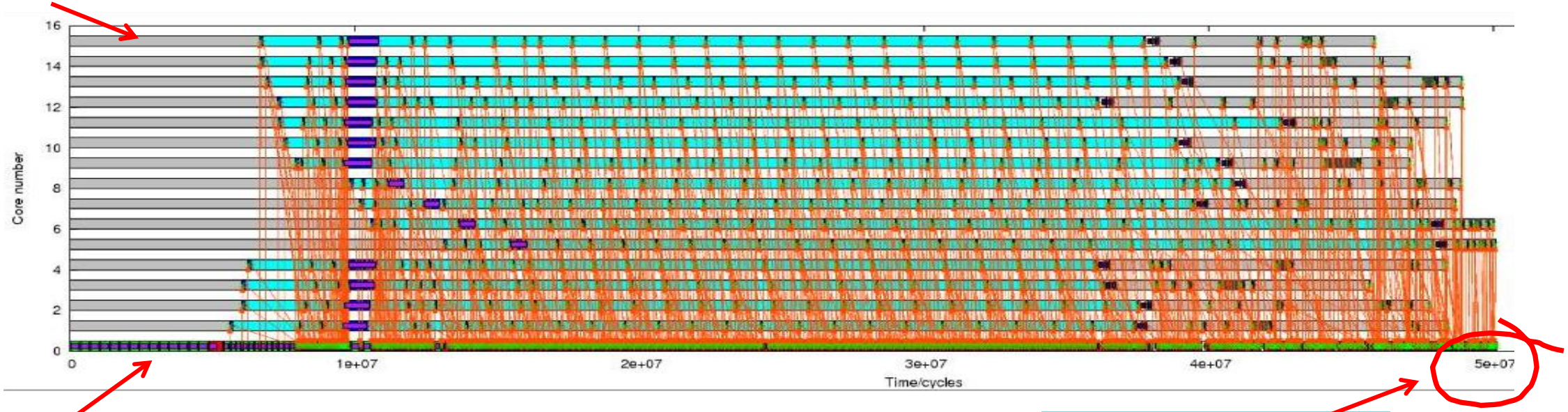
Barrelfish OS



Domain Spanning Visualization

- First version:
  - Centralized, poor scalability, but correct
  - 1021 messages, 487 memory allocation RPCs

Core 16

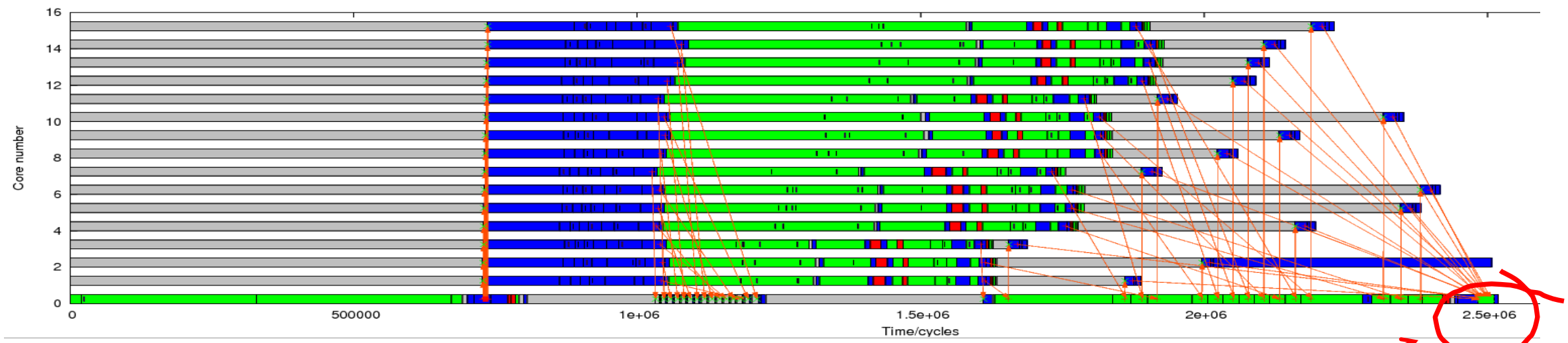


Core 0

50 million cycles (approx 40ms)

Improvements (3)

- Change the API
  - Create domains on all cores at once
  - 76 messages



2.5 million cycles (approx 1ms)

Distribution

- Distributed systems are rich source of difficult problems
  - Replication
  - Consistency
  - Quorum
- Well-studied field with good solutions
  - Outsider's perspective: research has focused on fundamental problems and techniques used in real systems
- Common abstractions
  - Replication
  - Relaxed consistency
  - Persistence
- How can these techniques be incorporated into programming model?
  - Libraries
  - Language integration
  - New models

Availability

- Services must be highly available
  - Blackberry/Google/… outage gets national media attention
  - Affect millions of people simultaneously
  - Service becomes part of national infrastructure
- High availability is challenge
  - Starts with design and engineering
  - Hard to eliminate all “single points of failure”
  - Murphy’s law rules
  - Antithetical to rapid software evolution
- Programming models provide little support for systematic error handling
  - Disproportionate software defects in error-handling code
    - Afterthought
    - Run in inconsistent state
    - Difficult to test
  - Erlang has systematic philosophy of fail and notify (but stateless)
  - Could lightweight transactions simplify rollback for stateful languages?

Performance

- Performance is system-level concern
  - Goes far beyond the code running on a machine
  - Most performance tools focus on low-level details
- Current approach is wasteful and uncertain
  - Build, observe, tweak, overprovision, pray
- Performance should be specified as part of behavior
  - SLAs as well as pre-/post-conditions
- Need scalability
  - Grow by adding machines, not rewriting software
- Architecture should be the starting point
  - Model and simulate before building a system
  - What is equivalent of Big-O notation for scalability?
- Adaptivity
  - Systems need to be introspective and capable of adapting behavior to load
  - e.g., simplify home page when load spikes, defer low-priority tasks, provision more machines, …

Application Partitioning

- Static partition of functionality between client and server
  - Clients have different architectures and capabilities
  - Adapt to changing constraints (e.g., battery)
  - Move computation to data, particularly when communications constrained
  - Code mobility
    - Exists in data center (VMs), why not across data center boundary?
- Currently, client and server are two fundamentally different applications
  - Evolution around interfaces
  - Volta (Microsoft)
    - Single program model, compiled for server and client

Defect Detection

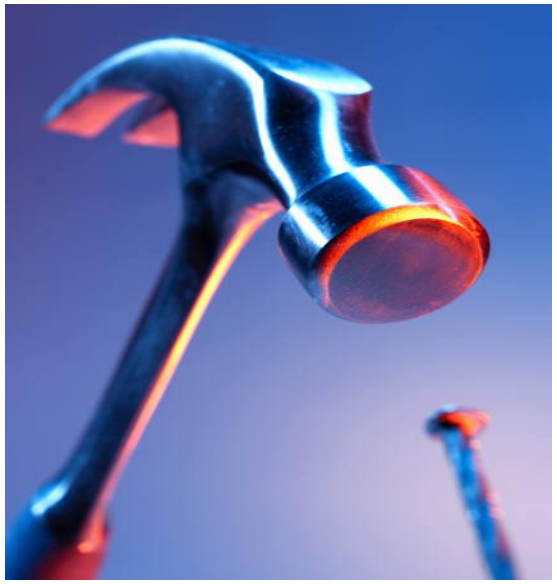
- Considerable progress in past decade on defect detection tools
  - Tools focused on local properties (e.g., buffer overruns, test coverage, races, etc.)
  - Little work on system-wide properties
- Modular checking
  - Whole program analysis expensive and difficult
  - Not practical for services
  - Assertions and annotations at module boundaries
  - Can check global properties locally
  - e.g., Rajamani & Rehof's Conformance Checking
- New domain of defects
  - Message passing
  - Code robustness
  - Potential performance bottlenecks

High-Level Abstractions

- Map-reduce and dataflow abstractions simplify large-scale data analysis in data centers
  - Convenient way to express problems
  - Hide complex details (distribution, failure, restart)
  - Allow optimization (speculation)
  - Not appropriate for services
- Need abstractions for wider range of problems
  - Interactive applications

Clearly a Programming Problem

- At least for a language and tools researcher

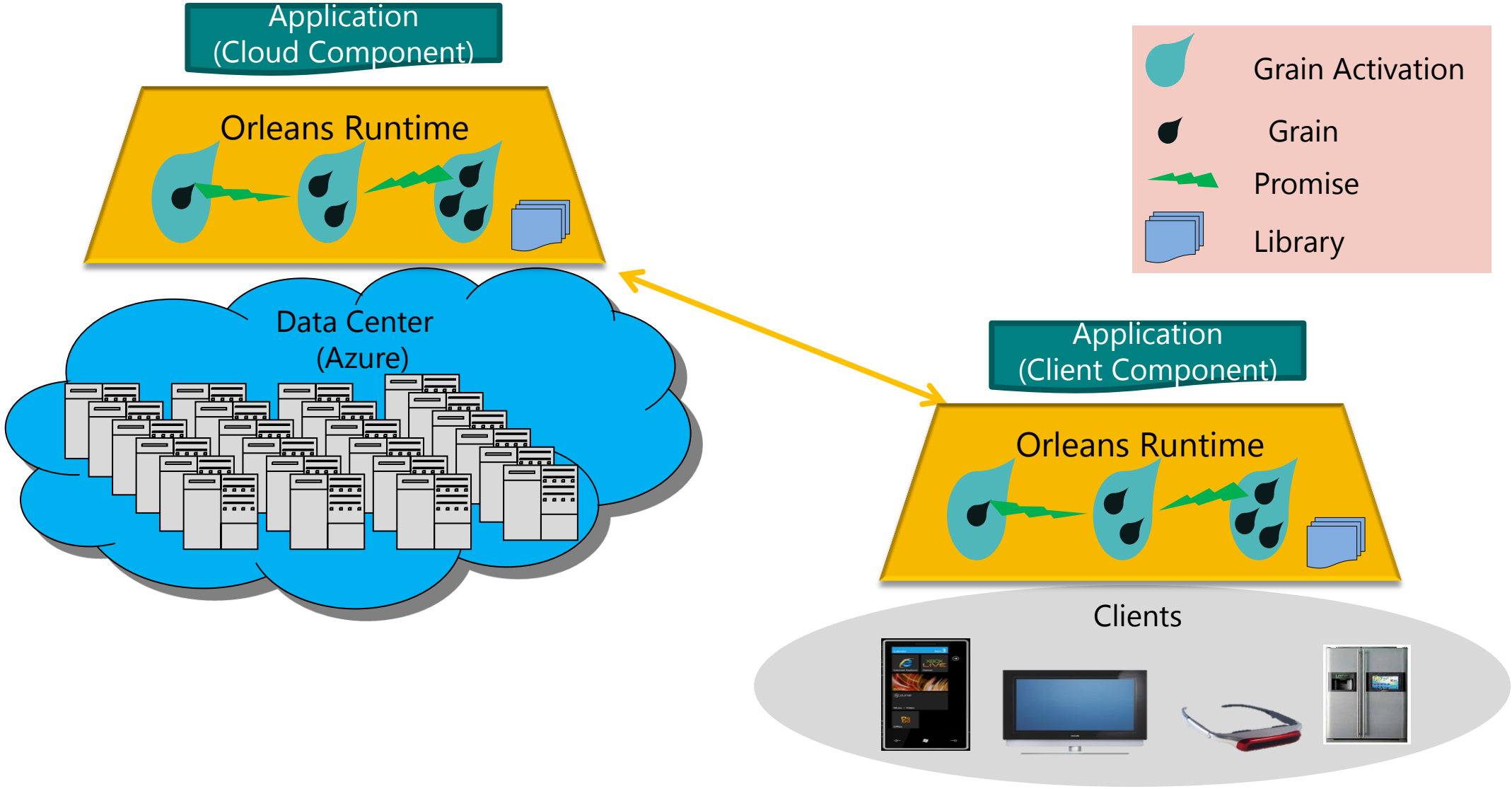


When your only tool is a hammer, everything looks like a nail.
-- Paul Hilfinger (PhD advisor)

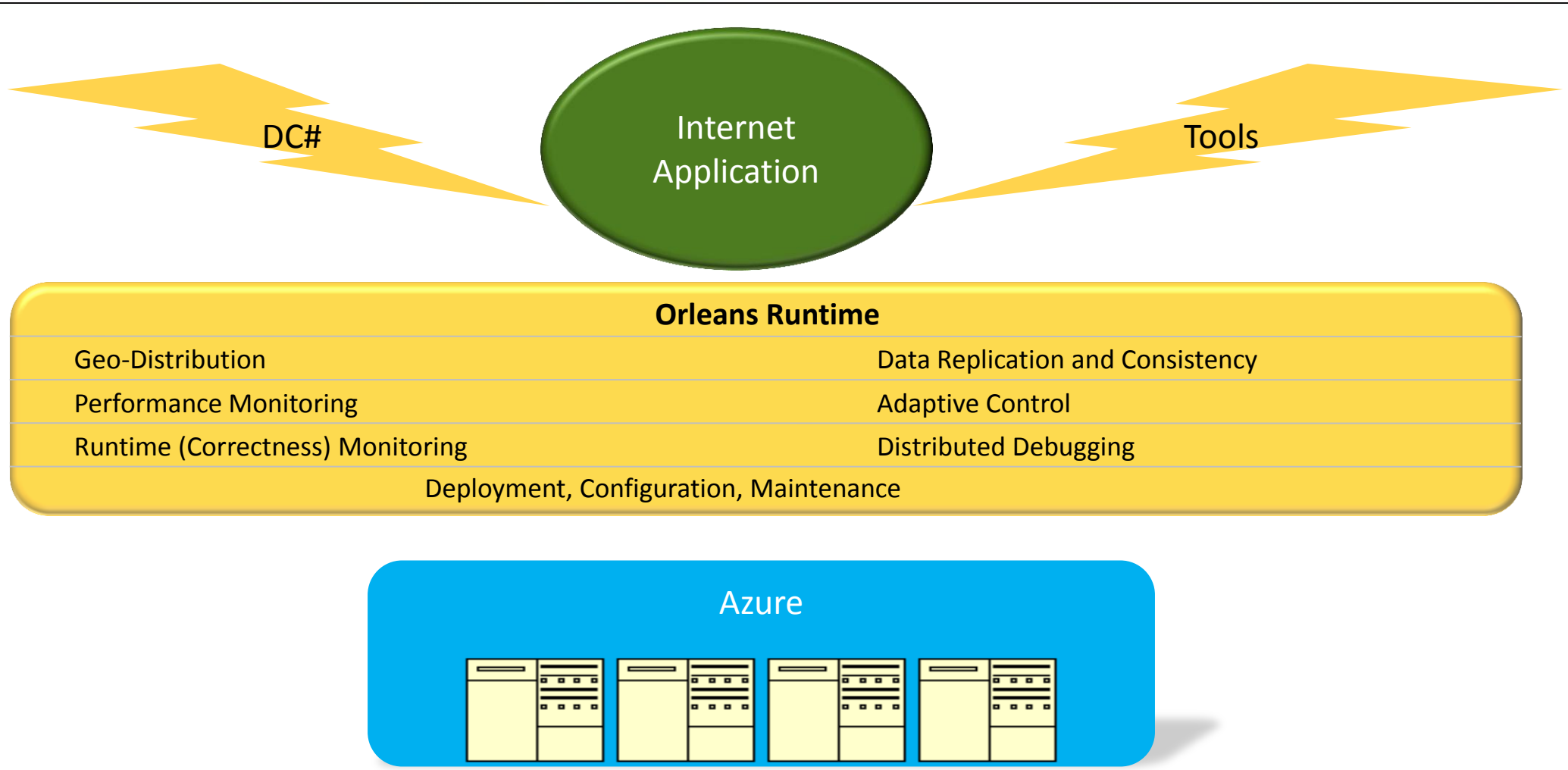
Orleans

- Goals
  - Simple, widely accessible programming model
  - Encourage use of scalable, resilient software architectures
  - Raise level of abstraction (CLR – Windows ≈ Orleans – Azure)
- Grains are unit of computation and data storage (Actors)
  - Can migrate between data centers
  - Replication, consistency, persistence handled by runtime system
- One programming model for client and server
  - Simplify development, debugging, performance tuning, etc.
  - Single-source distributed programs (eg Volta)
  - Enable code mobility

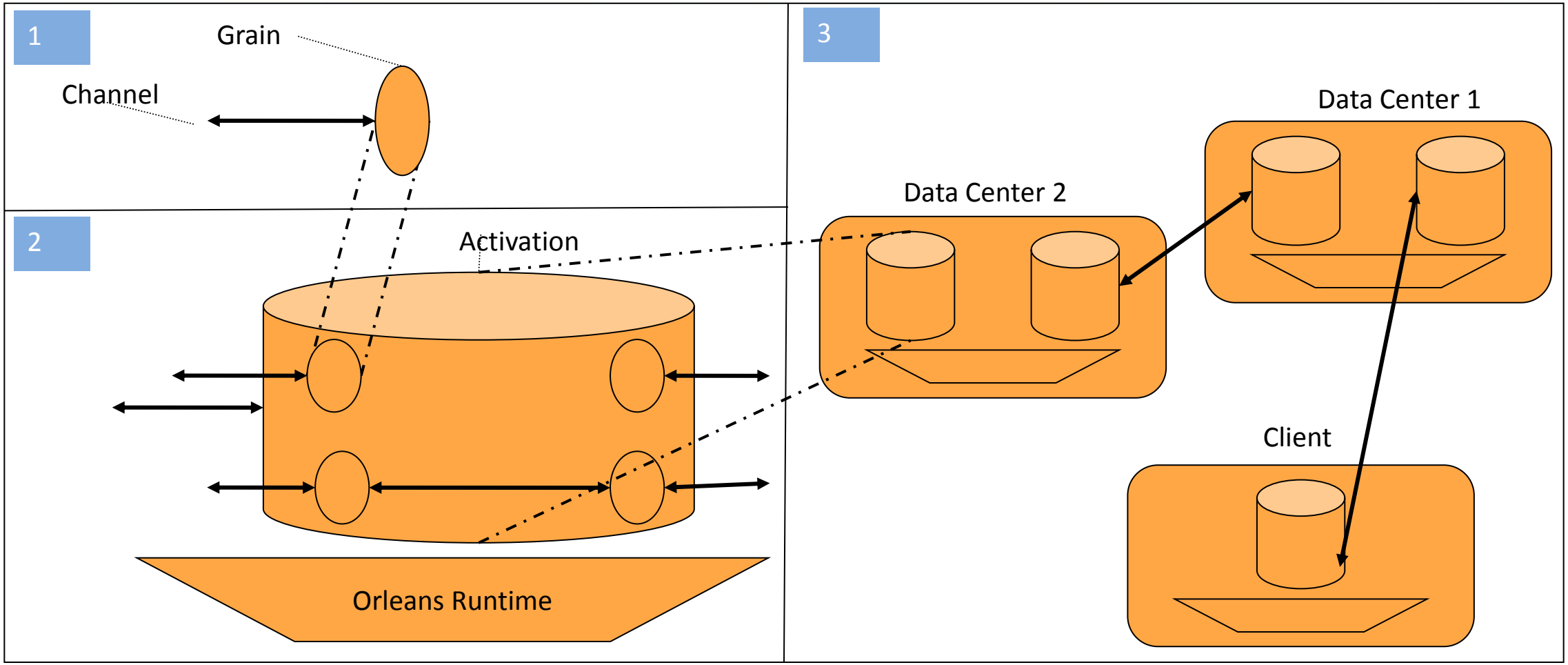
Orleans


Application
(Cloud Component)
Orleans Runtime
Data Center
(Azure)
Grain Activation
Grain
Promise
Library
Application
(Client Component)
Orleans Runtime
Clients

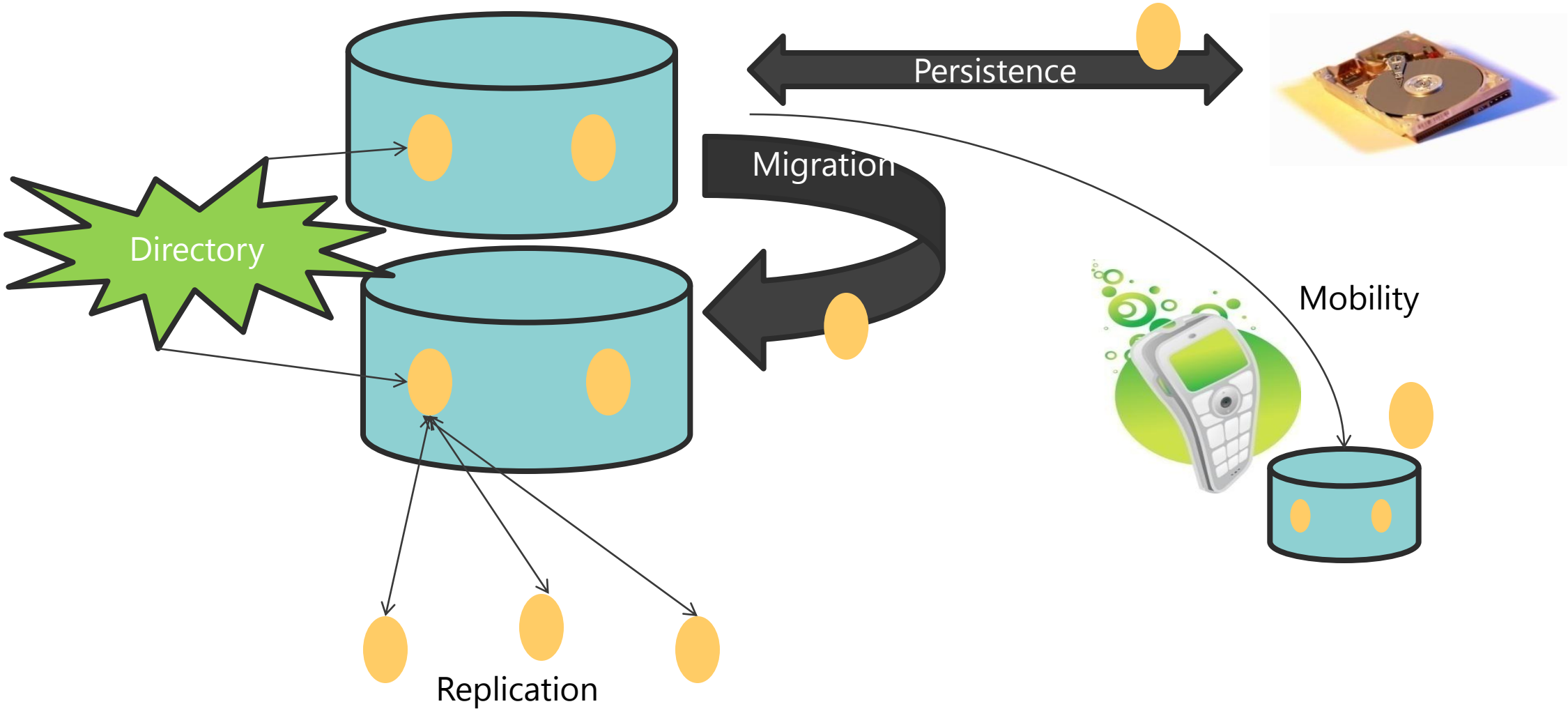
Orleans Architecture



Orleans Programming Model

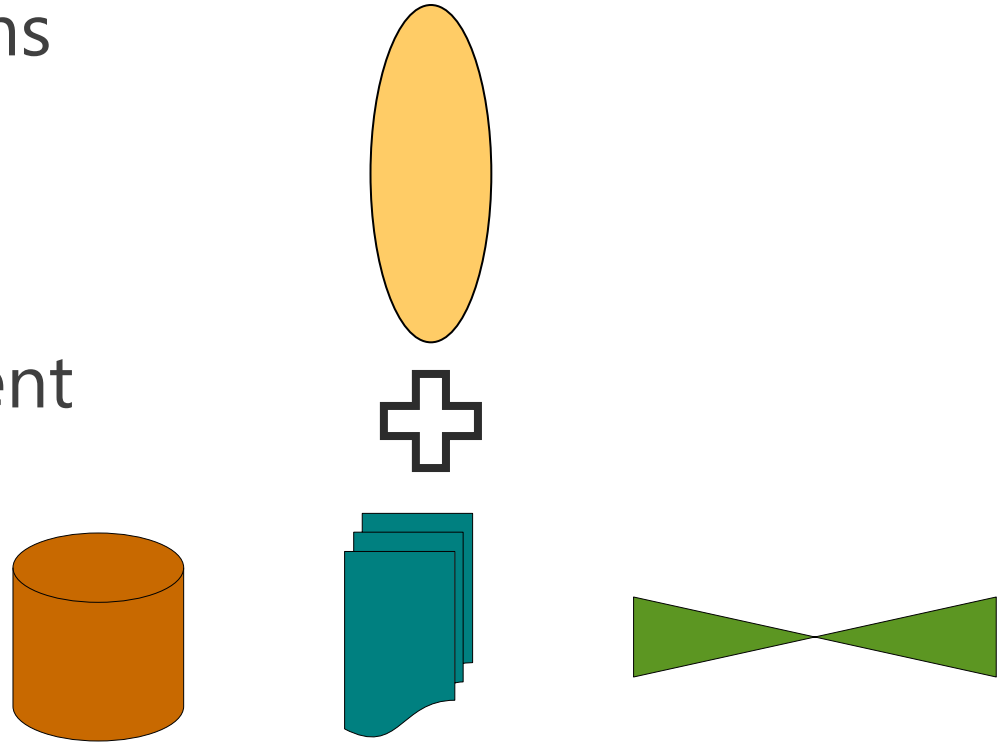


Data Model

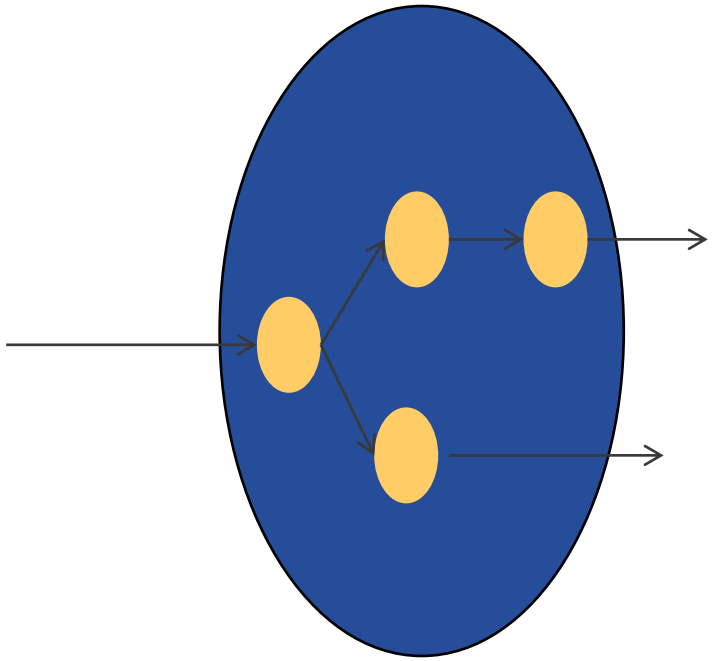


Separation of Concerns

- Orleans runtime provides functionality common to cloud apps
- Building blocks of distributed systems
  - Persistence
  - Replication
  - Consistency
- Configuration, versioning, deployment
- Monitoring, debugging, auditing

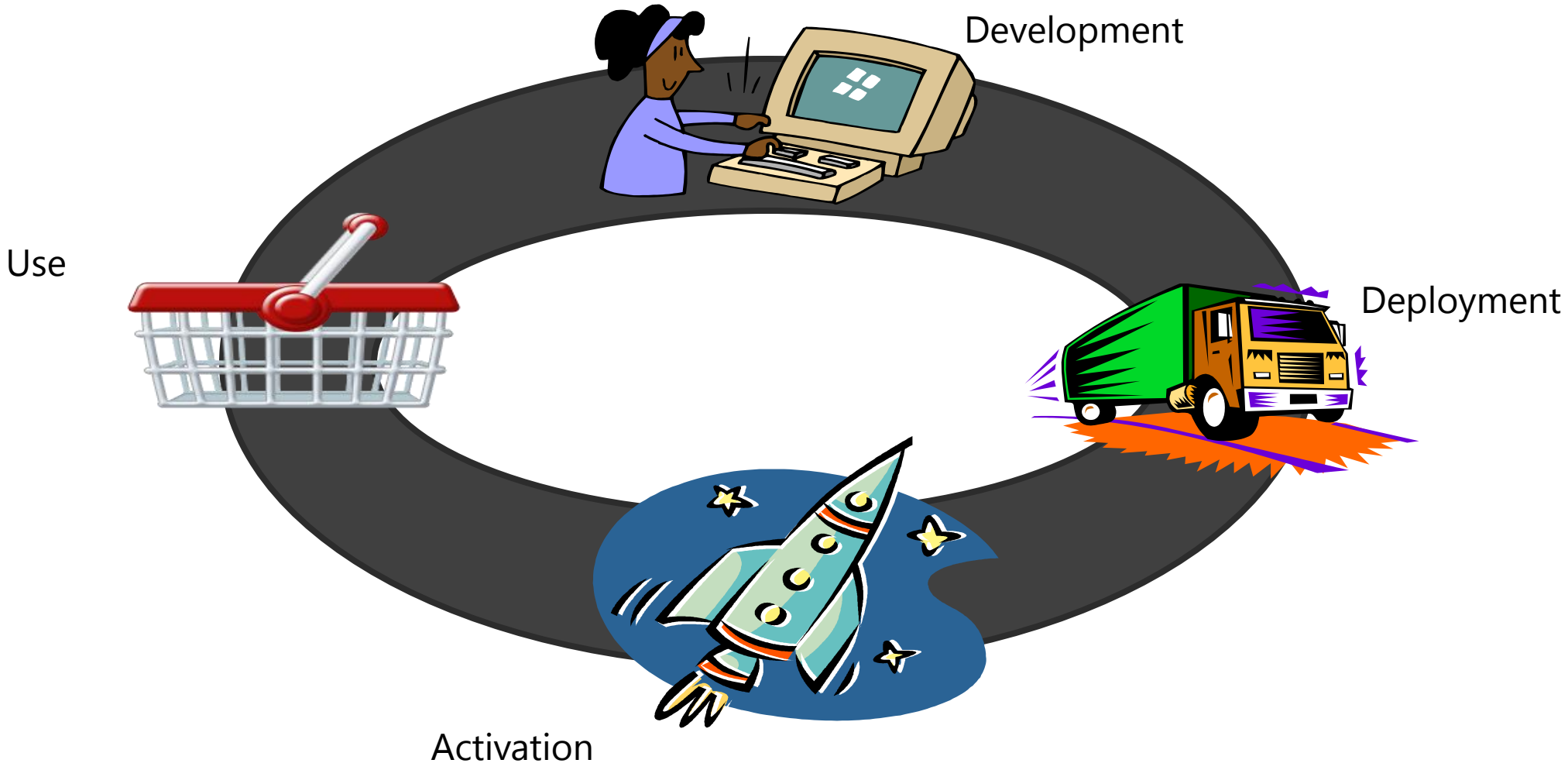


Composition



- How do we connect grains?
  - Internal: grain creates channel to another grain
  - External: grain talks to port, which is wired to port on another grain
- Coordination language describes wiring?
- Dataflow vs request-response model
  - What support do each need?

CC Application Lifecycle



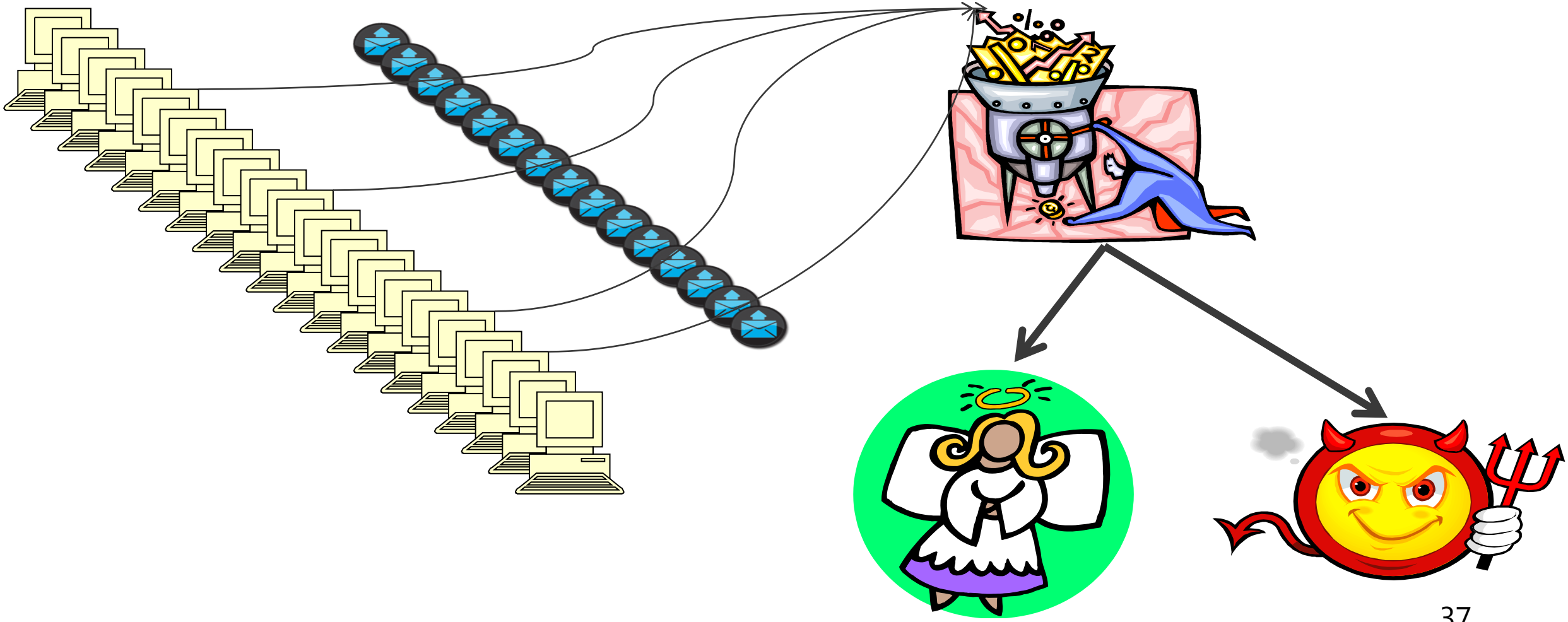
Continuous-running Systems

- Update code and data in place
  - Cannot stop system
  - Rapid, frequent code releases
  - Seamless evolution
- Multiple versions execute simultaneously
  - Test during deployment
  - Deployment not instantaneous

Cloud Tools

- Existing tools do not address problems of scale
  - Debug 10K machines running 1M separate tasks
  - Understanding performance of 10K machines and identifying performance bottlenecks
  - Monitor behavior of 10K machines to identify unexpected behavior, attacks, HW failure
- Few concurrent or parallel defect detection tools

Collaborative Debugging



Programming Languages Support

- Existing languages provide little support for message passing
  - Asynchronous stranger in a synchronous world
- Failure handling is afterthought
  - Disproportionate fraction of bugs in error handling code
  - Run when state is inconsistent
  - Difficult to test
- Programming model/architecture as well as language

Geo-Distributing

- Complex tradeoffs
  - Replicate data?
    - Consistency issues
    - Bandwidth cost
  - Partition data?
    - Partitioning criteria
    - Migration
  - Replicate computation?
    - Consistency issues
    - Inherent inefficiency
- What is goal of distribution?

Conclusion

- Cloud computing is more than VMs, data centers, web services, …
  - New form of computation
- Opportunity to correct problems with existing computing
  - Cost, complexity, reliability, …
- Exciting new challenges for the programming languages, compiler, programming tools communities