Microsoft

# Project Orleans
# Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein
Microsoft Research
Joint work with Sergey Bykov, Alan Geller, Gabriel Kliot, Michael Roberts, Jorgen Thelin

Microsoft Research
Faculty Summit
2015
July 8-9, 2015

# Project Orleans
## Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein
Microsoft Research
Joint work with Sergey Bykov, Alan Geller, Gabriel Kliot, Michael Roberts, Jorgen Thelin

July 8, 2015

Project “Orleans” is a programming model and runtime for building *cloud native* services

It’s available as open source on github

# What is Project “Orleans”?

- Distributed C#
- You define .NET interfaces and classes, as if they run in a single process.
- Orleans runs your app on a cluster of servers
- Orleans ensures your app is scalable, reliable, and elastic
- Performance is near-real-time (milliseconds)
- 3-5x less code to write than on a bare virtual machine

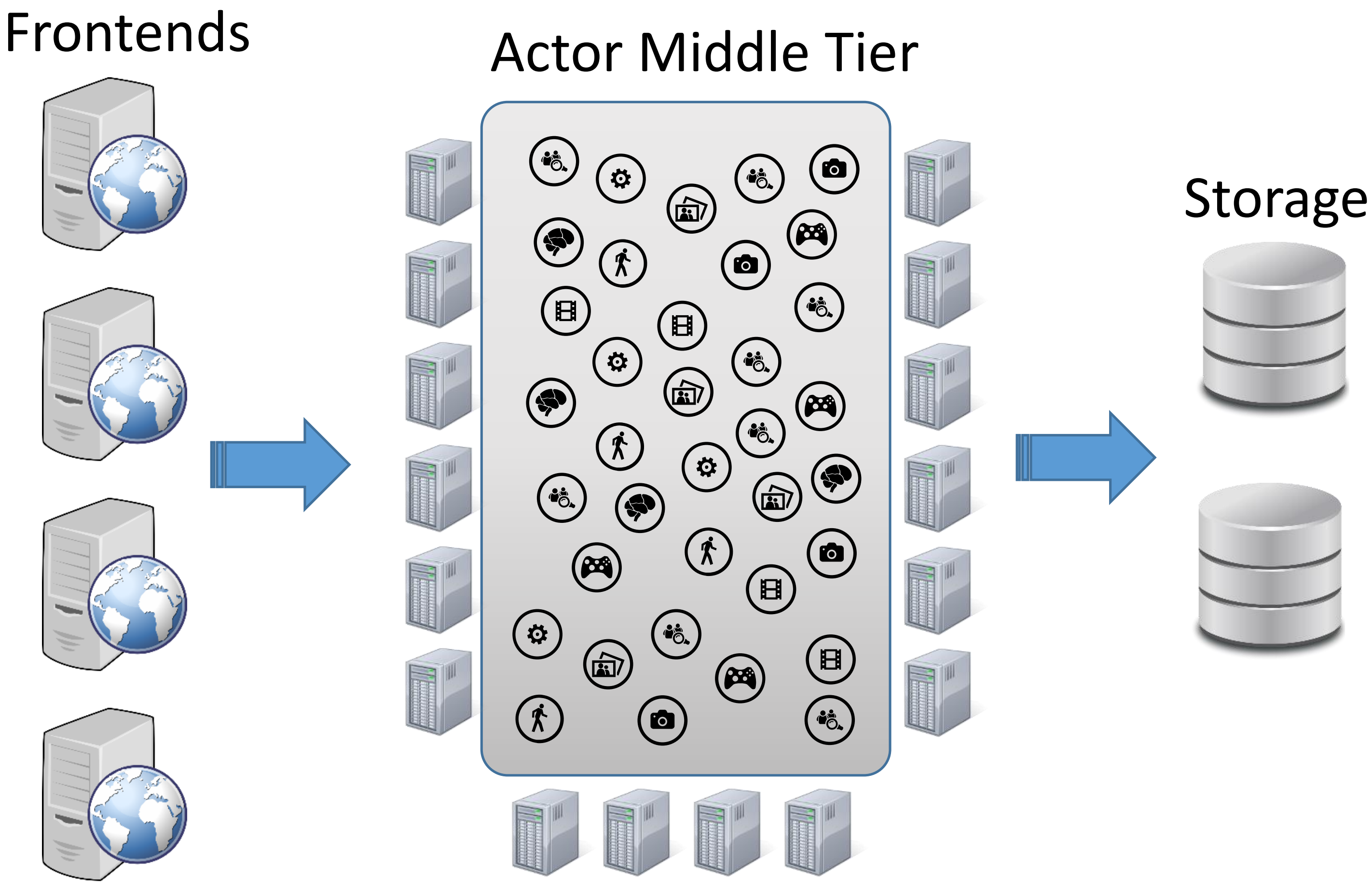
# Motivation

- Developer Productivity
  - Challenges: concurrency, distribution, fault tolerance, resource management…
  - Domain of distributed systems experts
  - Orleans helps desktop developers [and experts] succeed
  - You write much less code.
- Scalability by default
  - Designs and architectures break at scale
  - Failure to scale may be fatal for business
  - Code must be scale-proof – must scale out without rewriting

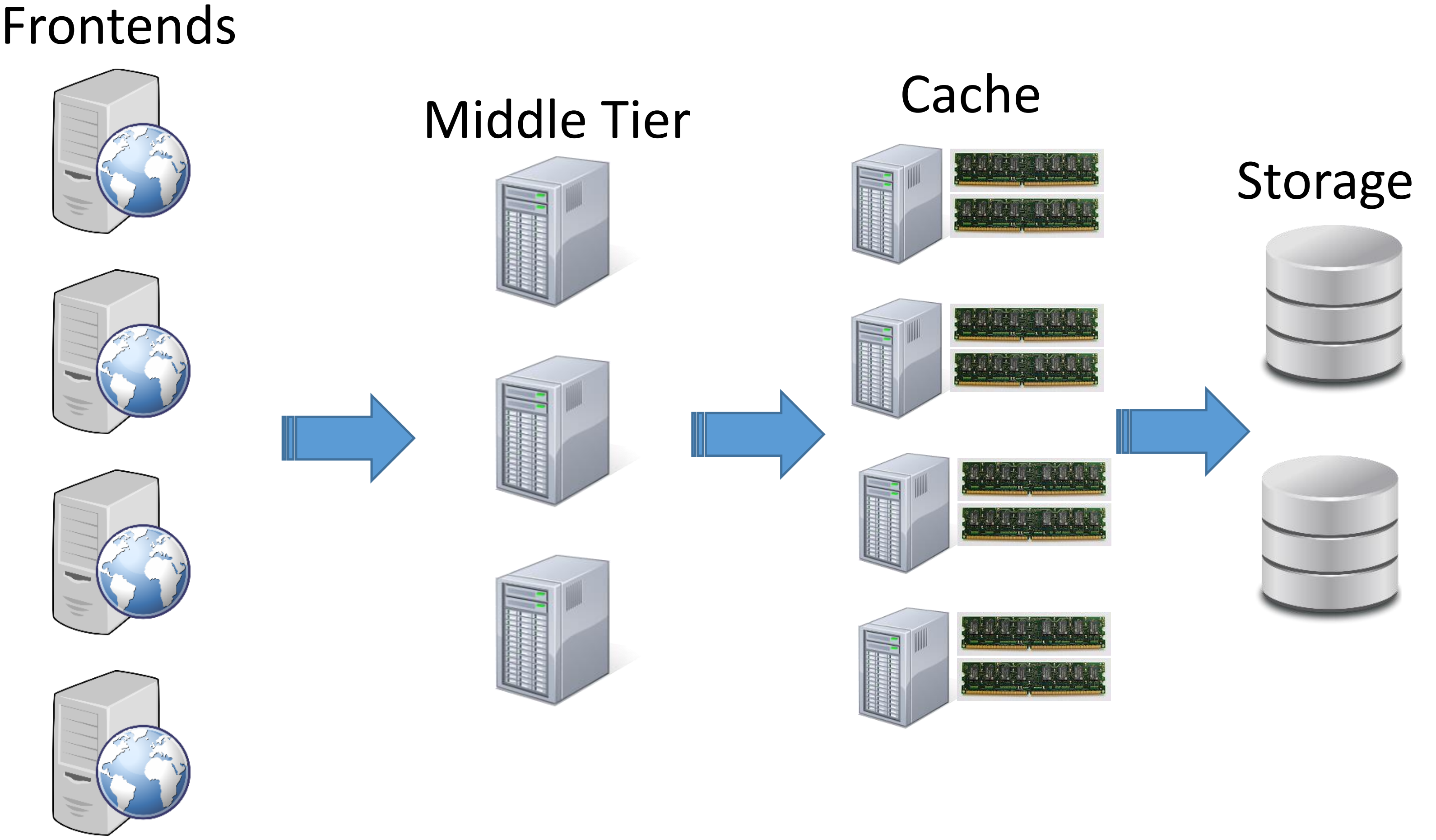
# Actor Model

- Orleans programs use the actor model
- Actors are objects that don’t share variables
- Orleans adapts the actor model for challenges of cloud computing

# Actor Model as Stateful Middle Tier



# What's the Alternative?
# A Cache Tier




- **Lost semantics of storage**
- **Lost concurrency control**
- **Data shipping**
- **Actor model can solve all of these problems**

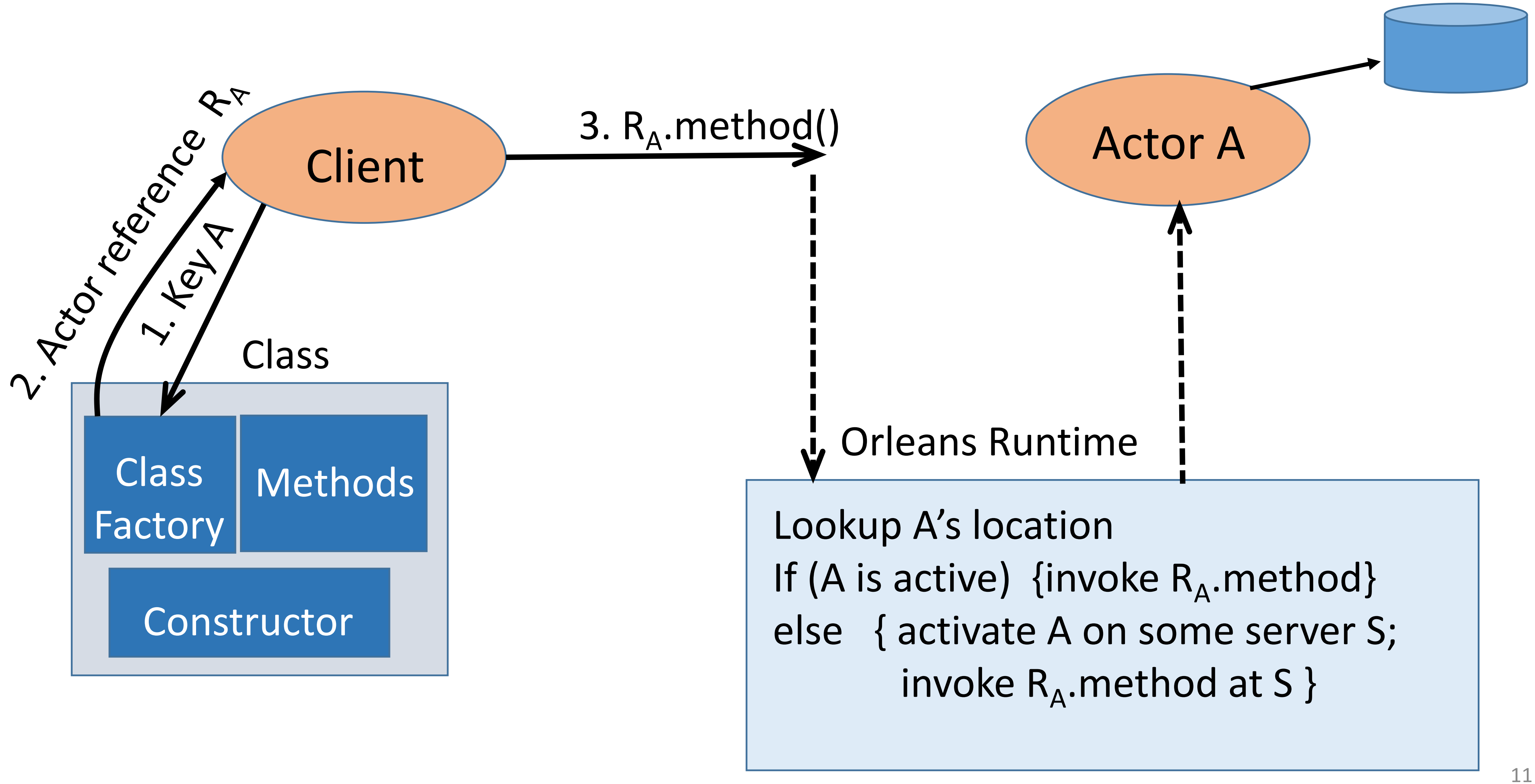
# Problems with Actor Model Frameworks

- Too low level
  - App manages lifecycle of actors, exposed to distributed races
  - App has to deal with actor failures, supervision trees
  - App manages placement of actors – resource management
- *Developer has to be a distributed systems expert*
- Orleans avoids these problems with a higher level actor model

# Orleans Programming Model

- Each class has a key, whose values uniquely identify actors (i.e. instances)
  - Game, player, phone, device, scoreboard, location, etc.
- To invoke a method M on an actor A of class C:
  - Call C's local class factory with A's key as parameter
  - Class factory returns an actor reference $R_A$
  - The caller invokes M on $R_A$
- The Orleans runtime manages activations of actors

# Invoking a method on actor A



# Key Innovation: *Virtual* actors

1. Actor instances always exist, virtually
    - Application neither creates nor deletes them. They never fail.
    - Code can always call methods on an actor

2. Activations of actors are created on-demand
    - If there is no existing activation, a message sent to it triggers instantiation
    - Transparent recovery from server failures
    - If an actor isn’t used for a while, it is deactivated
    - The Orleans runtime manages the actor’s lifecycle

3. Location transparency
    - Actors can pass actor references as parameters to a method and can persist them
    - These are logical (virtual) references, always valid, not tied to a specific activation

# Asynchronous RPC

- Method invocations are asynchronous
- Method returns a “task” (i.e., a promise), and caller continues executing
- When caller references a task’s result, it blocks until the task completes
- .NET has language support for this (Async/Await)

```
async Task<int> MyMethodAsync() { ... };
. . .
Task<int> myTask = MyMethodAsync();
// Other work
int x = await myTask; //blocks until MyMethod returns
```

# Single Threading

- Orleans runtime schedules invocations of actor methods on hardware threads
- Activations are single-threaded
  - Since actors don't share state, there's no need for locks
  - Optionally re-entrant
  - Multiplexed across hardware threads
- Cooperative multitasking
  - Since multithreading is at the user level, all I/O and method calls must be asynchronous
  - Synchronous call would block the hardware thread

# Actor State Management

- The runtime instantiates an actor by invoking the actor's constructor
  - The constructor typically reads the actor's state based on its key
  - Usually from storage, but possibly from a device (e.g. phone, game console, sensor)
- The actor saves its state to storage whenever it wants
  - Typically before returning from a method call that mutates its state
  - Or could be after *n* seconds, or after *n* calls, etc.
- Declarative persistence
  - Attach all state variables to an interface that inherits from IState
  - Declare a persistence provider for the class (Azure Table, Azure SQL DB, Redis…)
  - Invoke "WriteStateAsync" to save the state to the persistent store

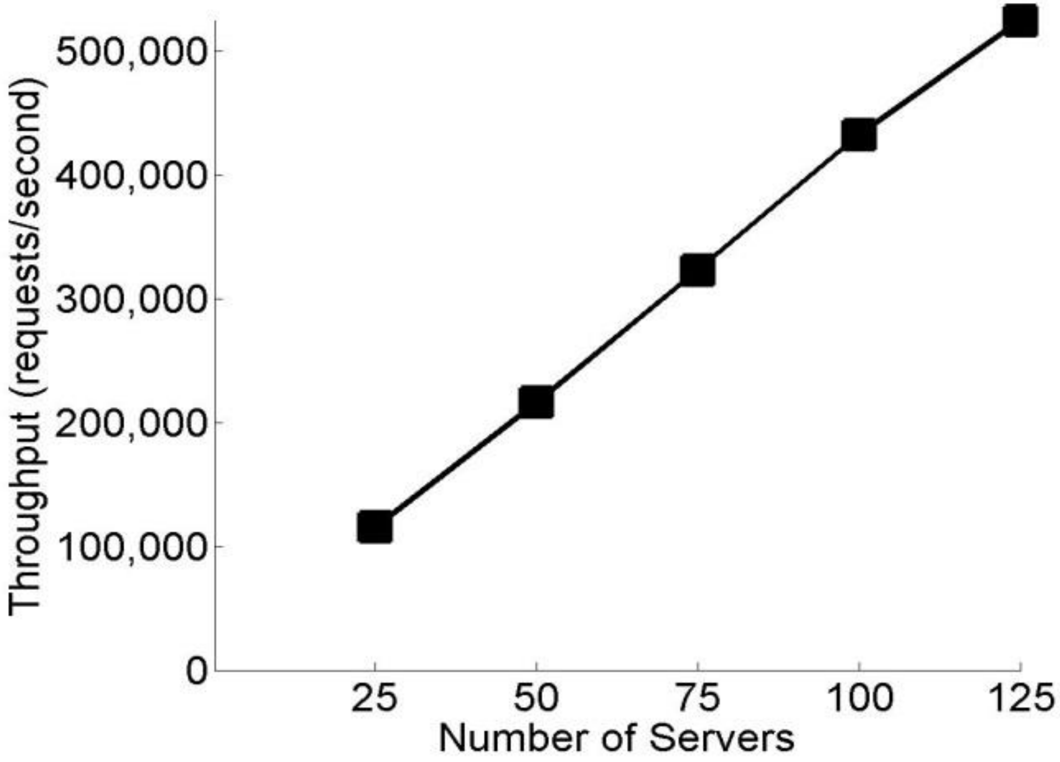
# Stateless Actors

- By default, there’s at most one activation of an actor
- But if an actor is declared to be stateless, then the runtime creates an activation local to the caller
- So there will be multiple activations of the actor
- Enables high throughput on actors with immutable state (e.g., a cache)

# Scalability

- Near linear scaling to hundreds of thousands of requests per second
- Scalable in number of actors
- Multiplexes resources for efficiency
- Location transparency simplifies scaling up or down
- Elastic – transparently adjusts to adding or removing servers

Test Lab Numbers




Request: Client → Actor 1 → Actor 2

# Orleans was built for…

## Scenarios

- Social graphs
- Mobile backend
- Internet of things
- Real-time analytics
- 'Intelligent' cache
- Interactive entertainment

## Common characteristics

- Large numbers of independent actors
- Free-form relations between actors
- High throughput/low latency
- Fine-grained partitioning is natural
- Cloud-based scale-out & elasticity
- Broad range of developer experience

- Not good for a service where different requests span different combinations of records over a large database

# Production usage

- First production deployment in 2011
- Halo 4 (December 2012) - all back end services
  - Players, games, statistics, regions, scoreboards, ….
  - Dozens of services, 10s to 100s of machines each
  - 100Ks of requests per second
  - Bursty load (evenings, weekends) and peak load at product launch
- Public preview since April 2014. Open source since January 2015.
- Back end services of many other Microsoft game studios
- About ten other Microsoft services run on Orleans
  - Examples: intelligent cache, telemetry.

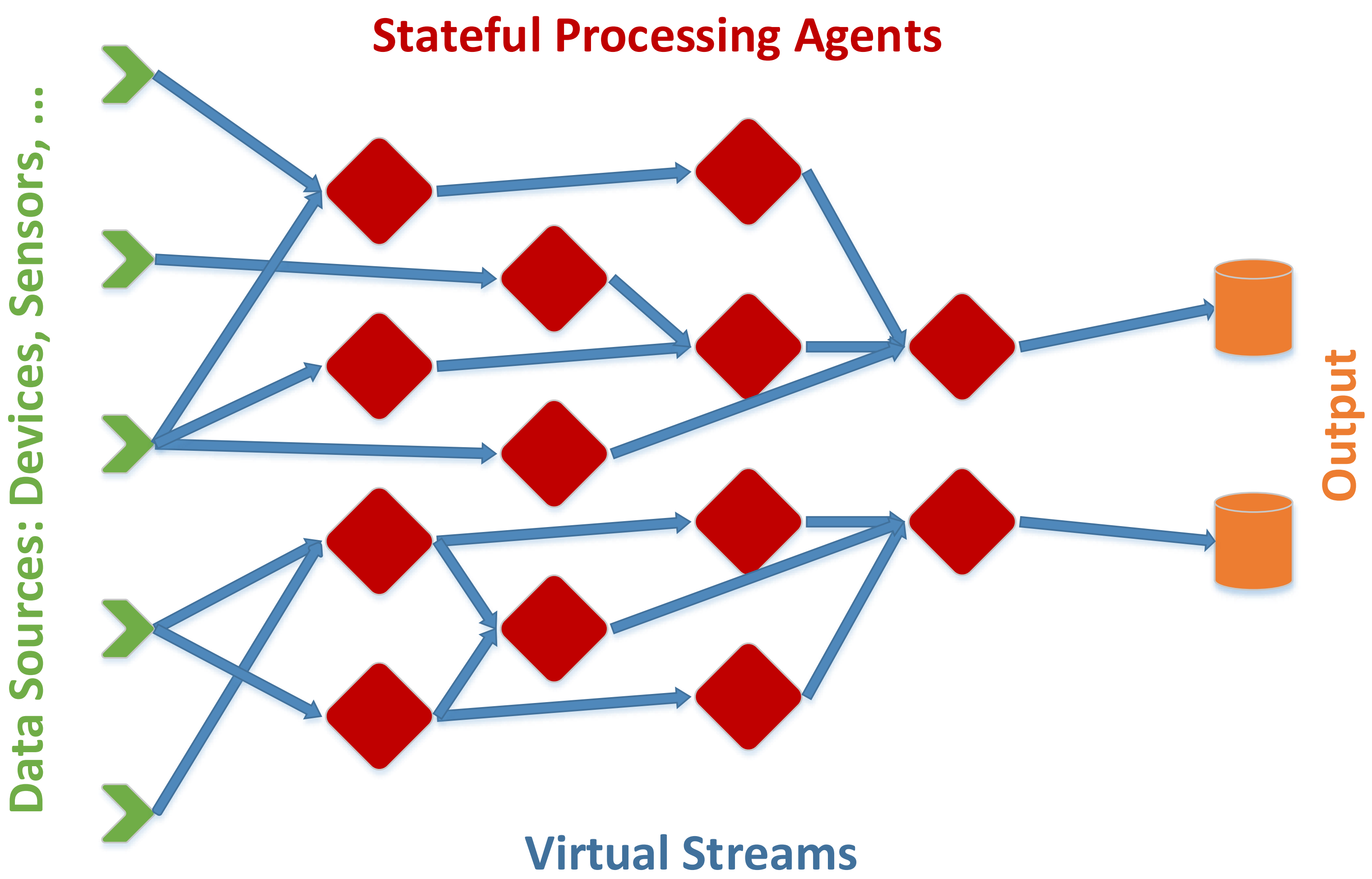
# Conclusion

- Orleans Benefits
  - Significantly improved developer productivity
  - Makes cloud-scale programming attainable to desktop developers
  - Scalability by default. Excellent performance
  - Proven in multiple production services
- A main innovation: Virtual actor programming model
- What I skipped: Virtual streams, timers, reminders, exceptions
- Future work: transactions, dynamic optimization, geo-distribution

Open Source Release: https://github.com/dotnet/orleans

# Backup slides

# Orleans Streams
## Combines dataflow & imperative styles

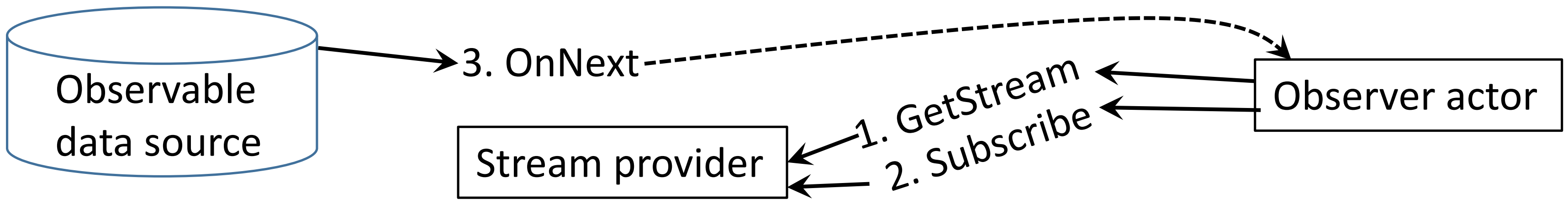



- Event source
- Processing agent (actor)
  - Imperative code
  - LINQ query
  - Stream processing engine
- Output
- Virtual stream
  - 'Stream provider' plug-in maps physical stream to virtual stream

# Orleans Streams – Programming model

- Programming model innovation – Virtual Streams
  - Stream is always available (i.e. fault tolerant).
  - No need to explicitly create or delete it.

- API –a session from observable data source to observer actor
  - Observer calls a *stream provider* with a stream identity and callback method
  - Stream provider registers the observer for the observable stream
  - For each of its events, the observable stream calls the observer's callback method
  - Similar to .NET's Rx interface, extended for remote, asynchronous access




- In production with a major internal customer