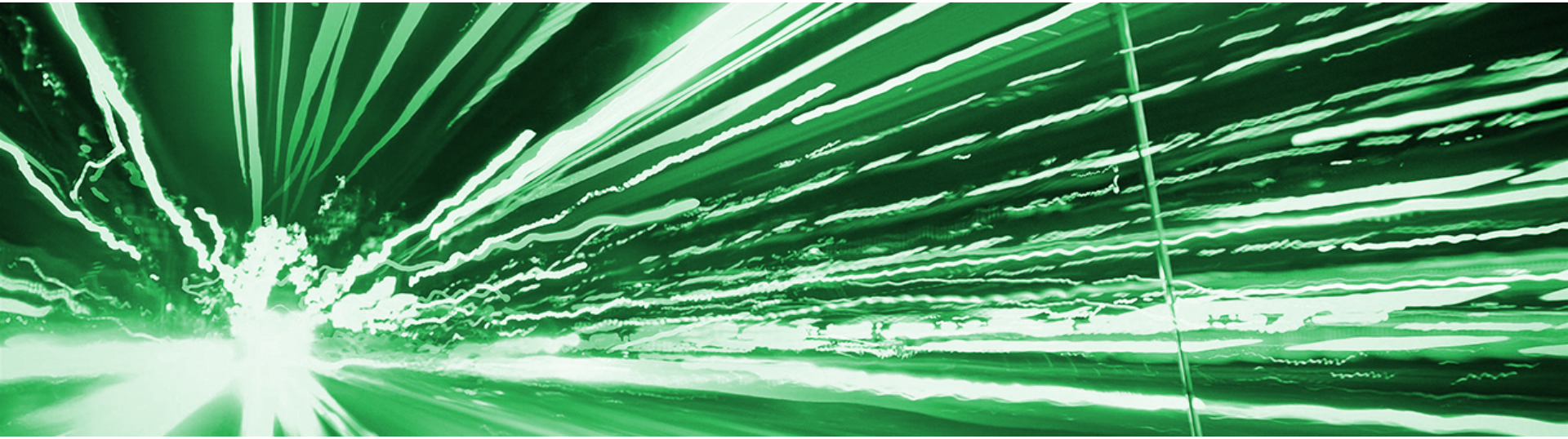
Chronicle



Latency Sensitive Microservices in Java

Reliability through highly reproducible systems

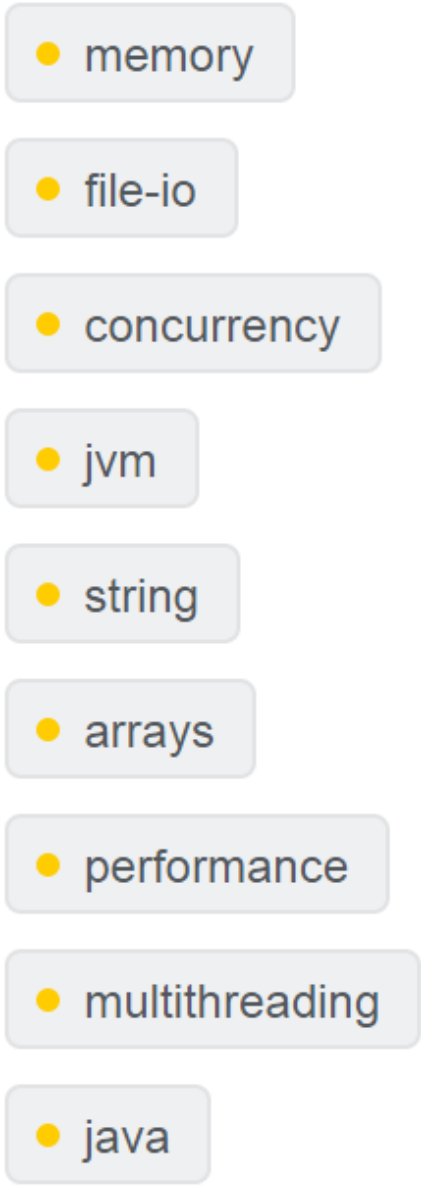
Peter Lawrey - CEO of Higher Frequency Trading
QCon London - 2017

Chronicle

Peter Lawrey

Java Developer / Consultant for investment banks and hedge funds for 10 years.

Most answers for Java and JVM on stackoverflow.com



Chronicle

Typical Solutions

Market data processing and distribution

Order generation and management

Position notification and distribution

Real time Compliance

30 micro-seconds typical,

100 micro-seconds, 99% of the time

Chronicle

Reliability means

Correct behaviour or die


MS-DOS V2.11 (UOP)
CAP ALT GR1 GR2
A
B
NEC
Advanced
Personal Computer
128 KB RAM

To go faster, do less

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupery

To go faster use private data

Micro-services do something simple
with privately held data.

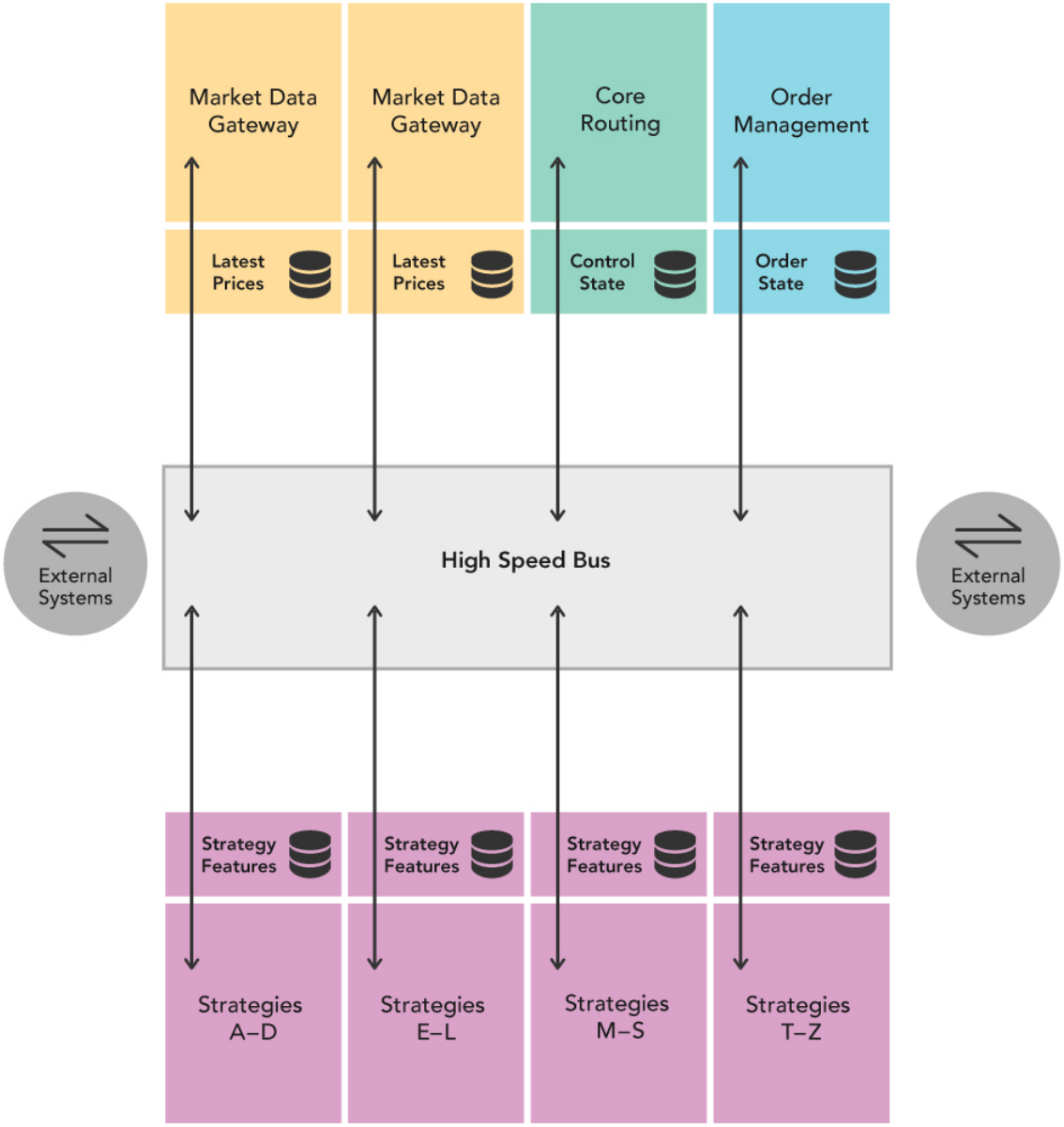
Cache	Size	Clock Cycles	Private
L1 Instruction	32 KB	3	Yes
L1 Data	32 KB	3	Yes
L2 Cache	256 KB	10	Yes
L3 Cache	1 MB – 48 MB	40 - 70	NO

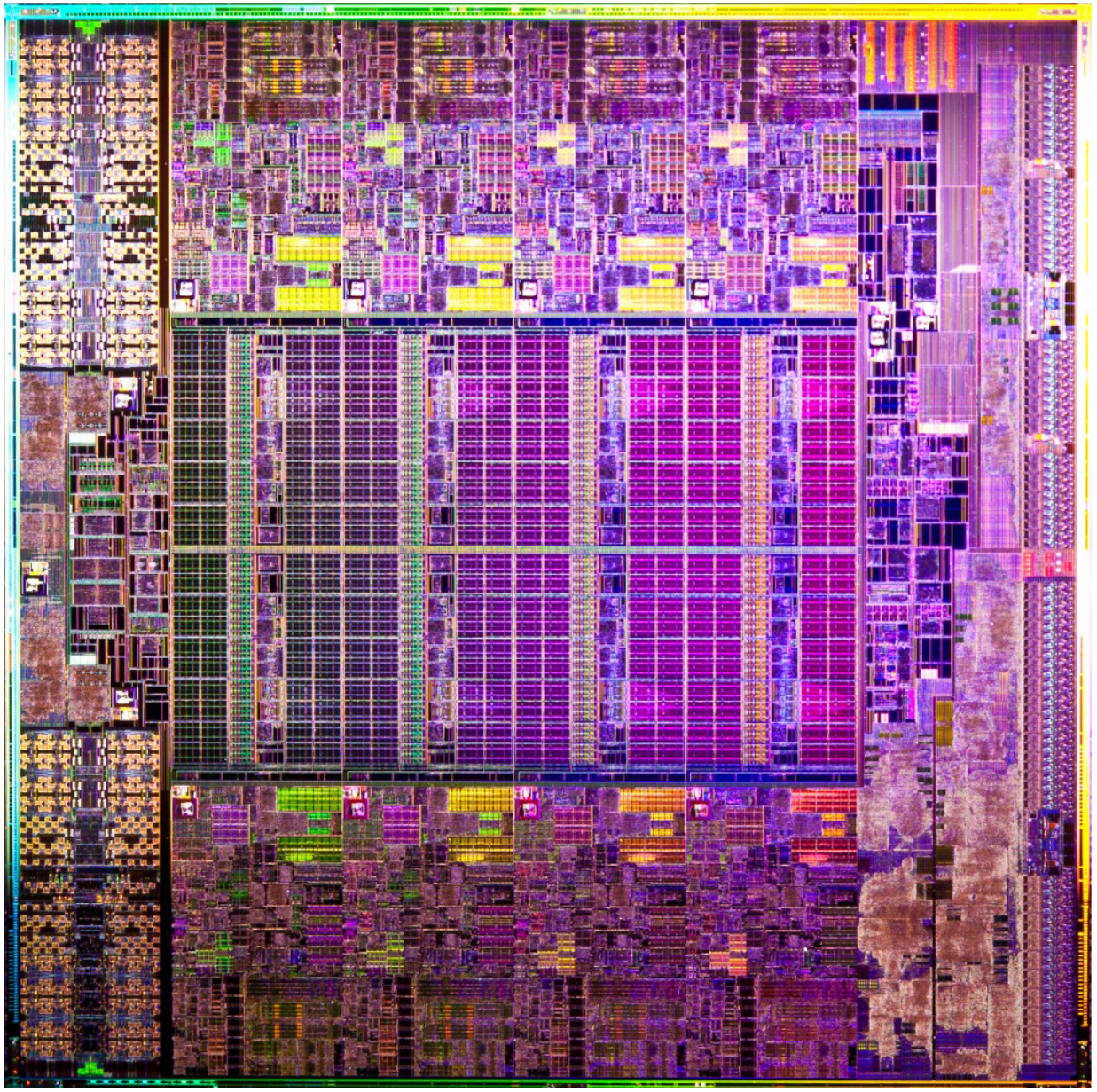
A Computer is a Distributed System.

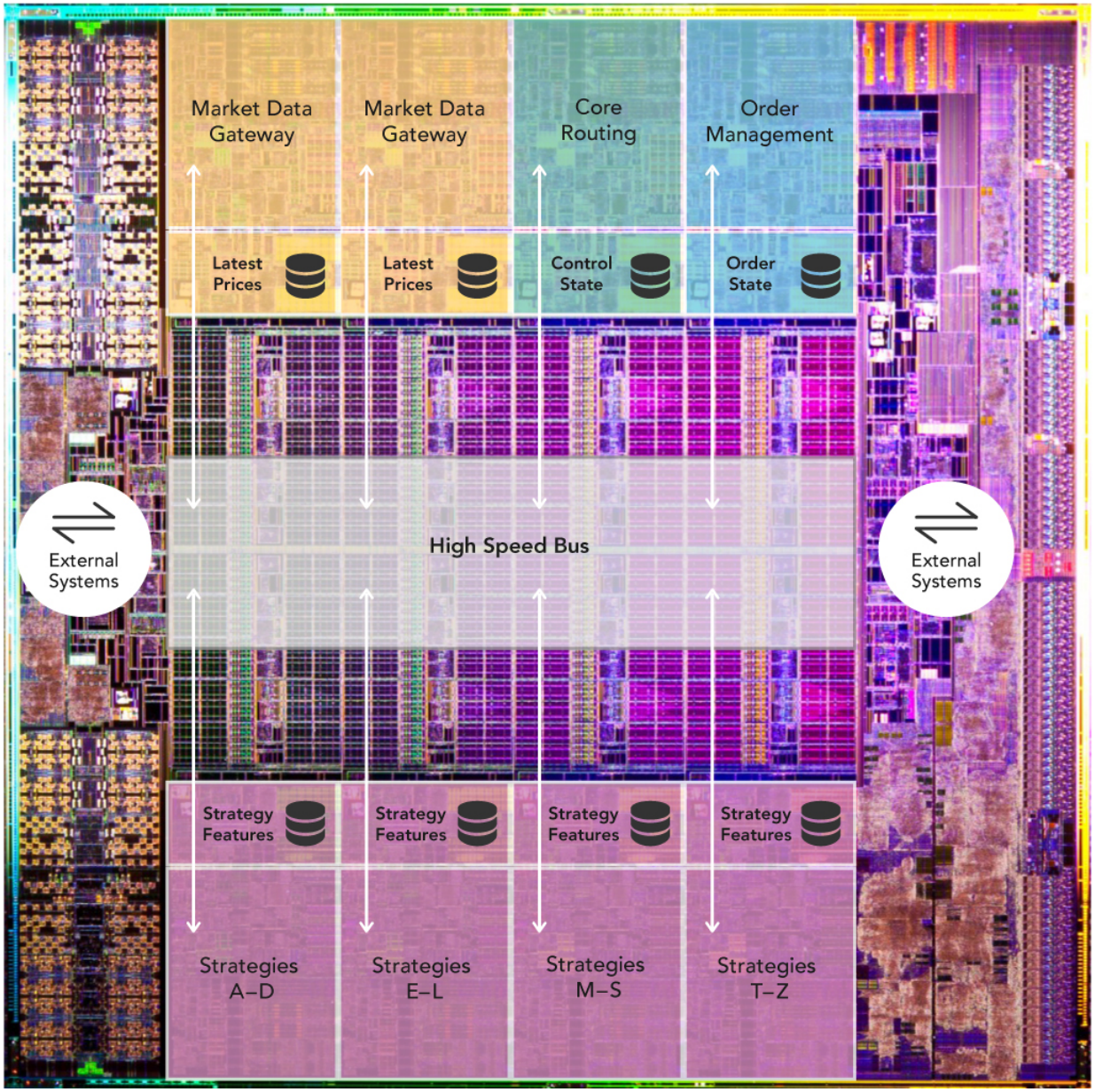
When you are considering short time scales of 10 micro-seconds or less, you have to consider that each core as a processor of it's own.

Each core

- has it's own memory (L1 & L2 caches)
- can run independently
- communicates with other cores via a L2 cache coherence bus.

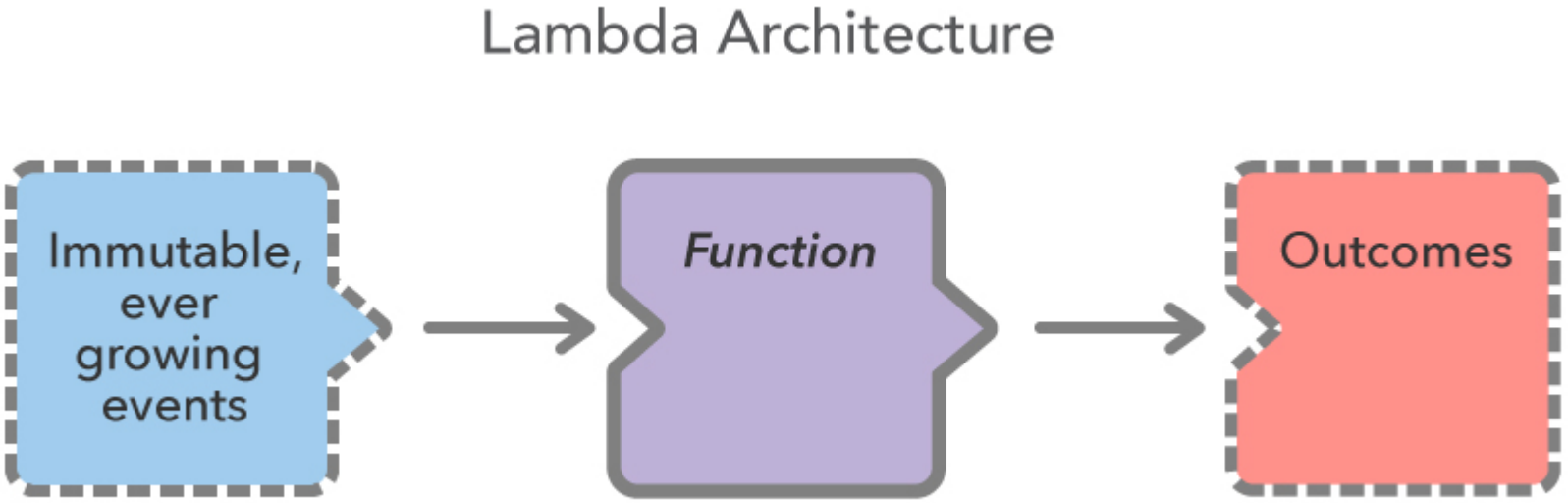

Market Data Gateway
Market Data Gateway
Core Routing
Order Management
Latest Prices
Latest Prices
Control State
Order State
External Systems
High Speed Bus
External Systems
Strategy Features
Strategy Features
Strategy Features
Strategy Features
Strategies A–D
Strategies E–L
Strategies M–S
Strategies T–Z




Market Data Gateway
Market Data Gateway
Core Routing
Order Management
Latest Prices
Latest Prices
Control State
Order State
External Systems
High Speed Bus
External Systems
Strategy Features
Strategy Features
Strategy Features
Strategy Features
Strategies A–D
Strategies E–L
Strategies M–S
Strategies T–Z

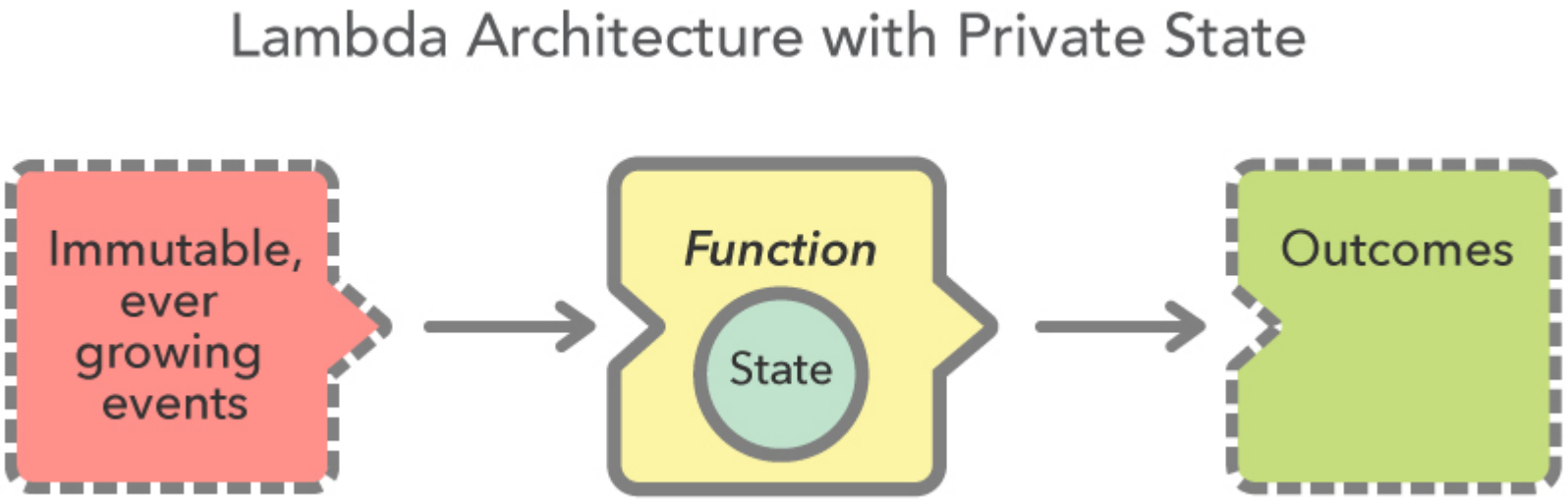
Building highly reproducible systems

Each output is the result of one input message. This is useful for gateways, both in and out of your system. Highly concurrent.



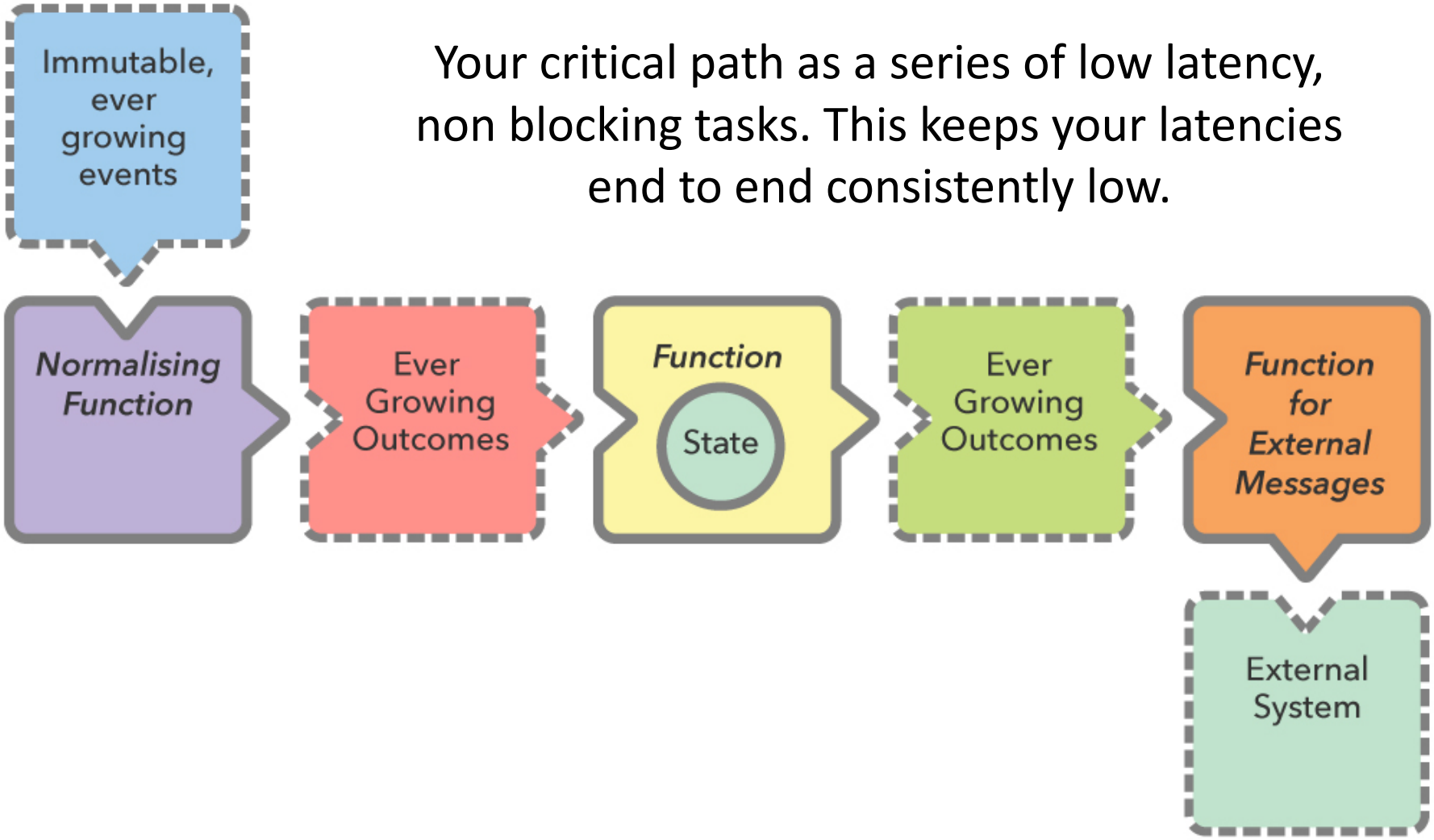
Building highly reproducible systems

Each output is the result of ALL the inputs. Instead of replying ALL input message each time, the Function could save an accumulated state.

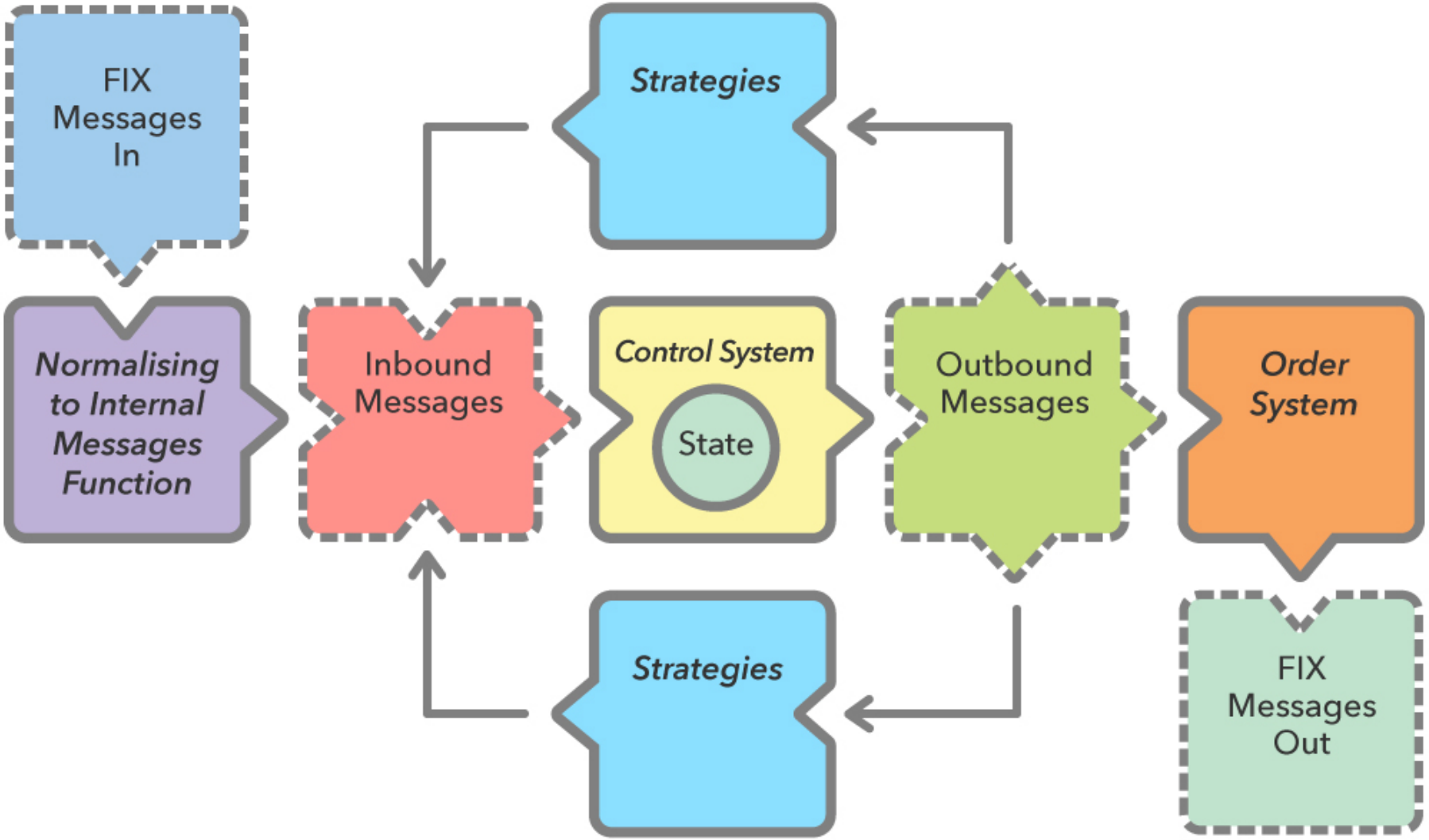


Lambda Architecture Services Chained

Your critical path as a series of low latency, non blocking tasks. This keeps your latencies end to end consistently low.



Lambda Architecture Services with Feedback



Record everything means

Greater Transparency

High Reproducibility

Faster time to fix

Faster delivery of a quality system

Is recording everything realist?

Cost?

Performance?

TBs in Java?

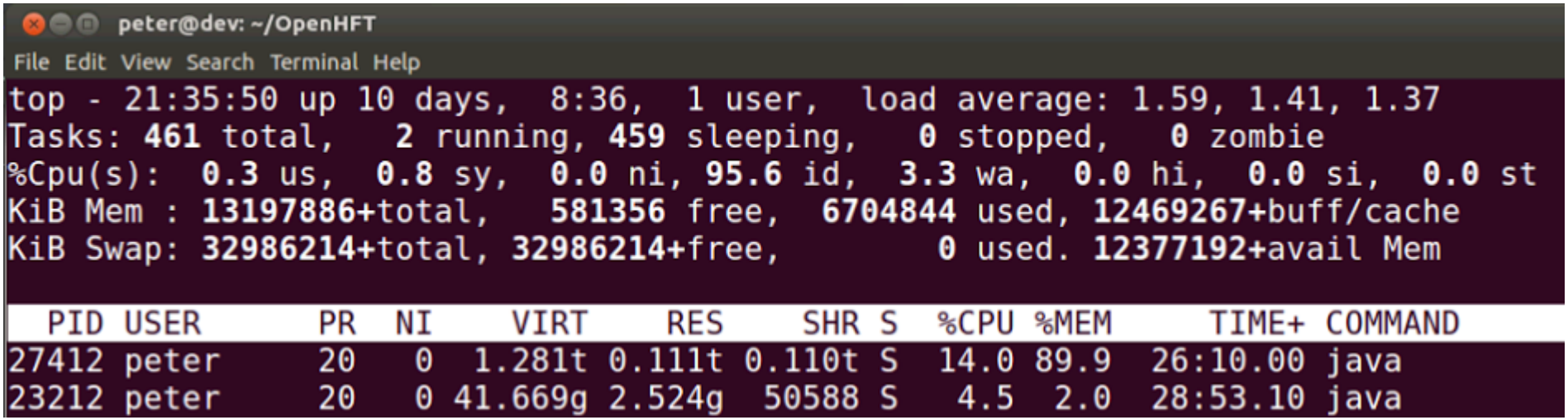
Flow Control?

How much does record everything cost

2 TB SSD ~ £1K

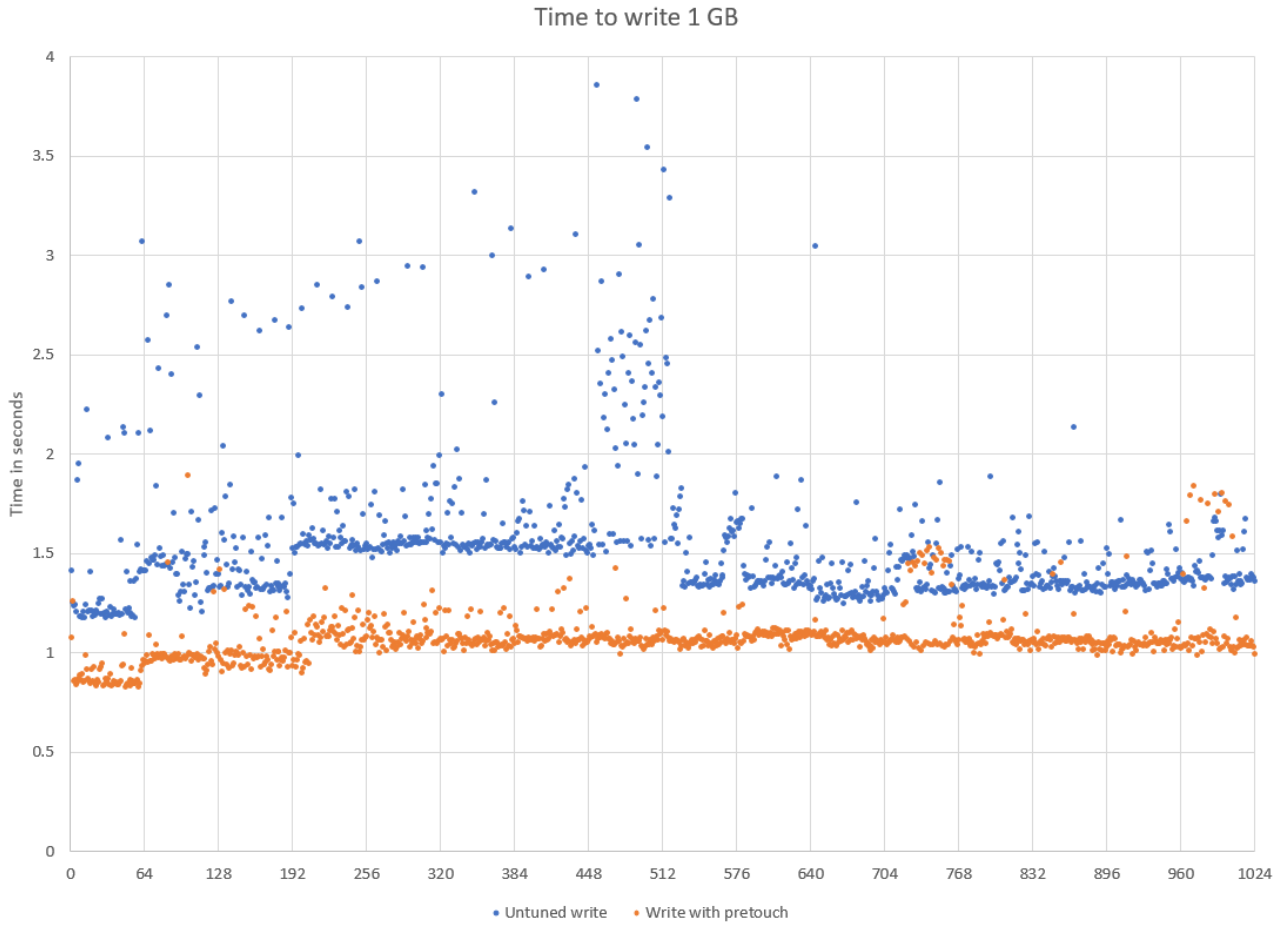
Scale to high volumes with less memory

Writing 1 TB on a 128 GB machine

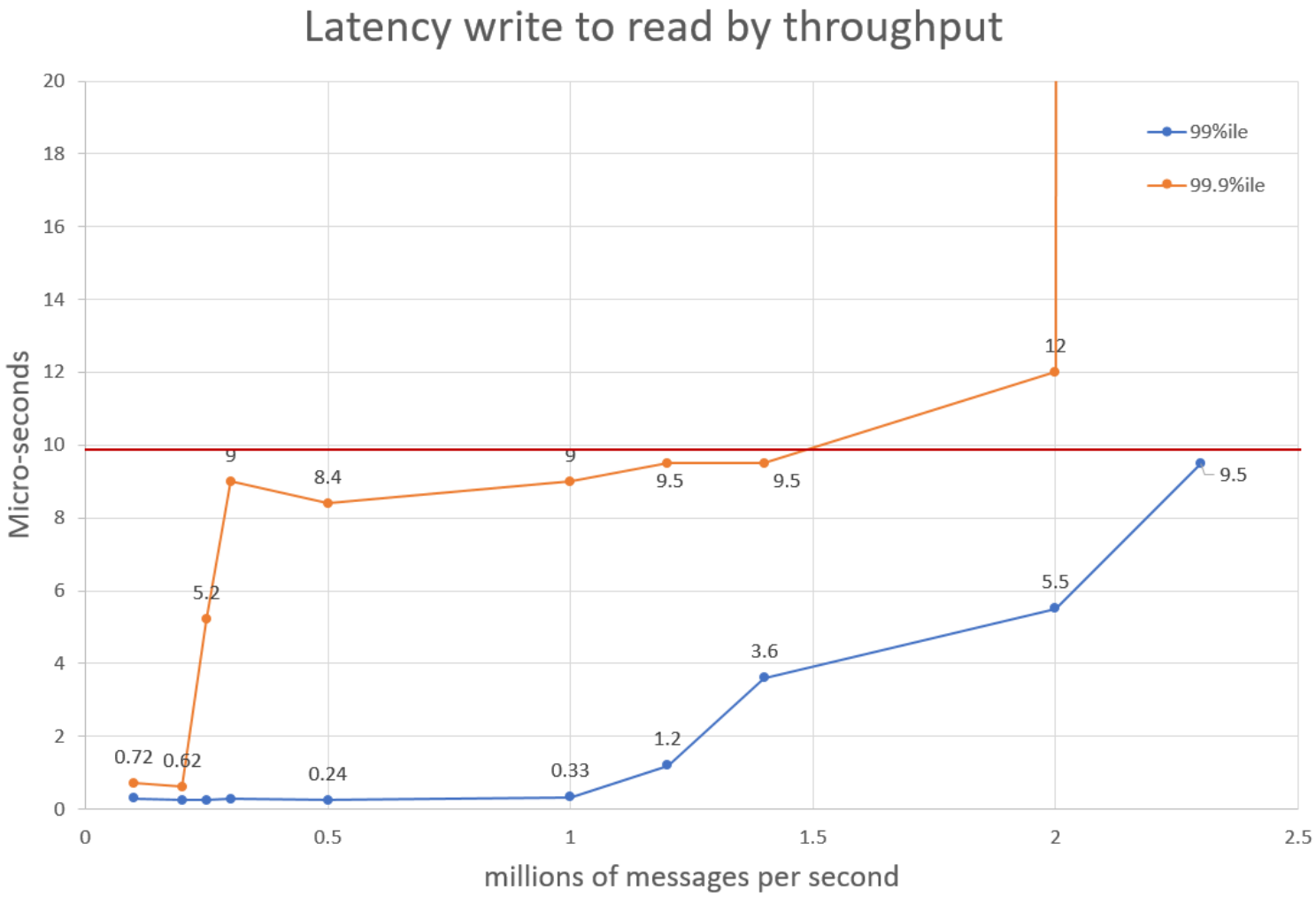


Scale to high volumes with less memory

Writing 1 TB on a 128 GB machine



Scale to high throughput with low latencies.



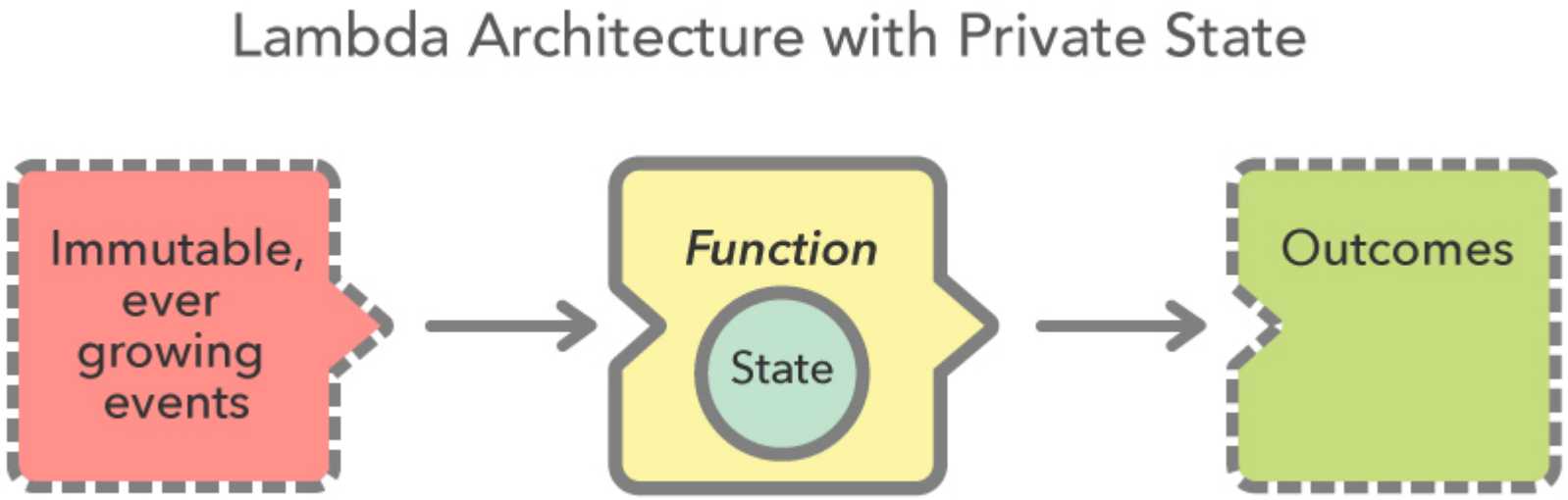
No Flow Control?

Market Data

Compliance

Reproduce each component independently

Whether you are enriching data from a database or production is complex, each service can be tested in isolation.



Testing and Debugging Microservices

Frameworks can make testing and debugging harder.

You need to be able to test and debug your components without the framework, or a transport.

Turning a Monolith into Microservices

Business Component + Transport = Service.

Starting with a simple contract

An asynchronous message has a type, a payload and doesn't return a result.

```
public interface SidedMarketDataListener {
    void onSidedPrice(SidedPrice sidedPrice);
}

public interface MarketDataListener {
    void onTopOfBookPrice(TopOfBookPrice price);
}
```

A Data Transfer Object

```
public class SidedPrice extends AbstractMarshallable {
    String symbol;
    long timestamp;
    Side side;
    double price, quantity;

    public SidedPrice(String symbol, long timestamp, Side side,
                      double price, double quantity) {
        this.symbol = symbol;
        this.timestamp = timestamp;
        this.side = side;
        this.price = price;
        this.quantity = quantity;
        return this;
    }
}
```

Deserializable toString()

For it to deserialize the same object, no information can be lost, which useful to creating test objects from production logs.

```
SidedPrice sp = new SidedPrice("Symbol", 123456789000L,
                                        Side.Buy, 1.2345, 1_000_000);
assertEquals("!SidedPrice {\n" +
        "  symbol: Symbol,\n" +
        "  timestamp: 123456789000,\n" +
        "  side: Buy,\n" +
        "  price: 1.2345,\n" +
        "  quantity: 1000000.0\n" +
        "}\n", sp.toString());

// from string
SidedPrice sp2 = Marshallable.fromString(sp.toString());
assertEquals(sp2, sp);
assertEquals(sp2.hashCode(), sp.hashCode());
```

Writing a simple component

We have a component which implements our contract and in turn calls another interface with a result

```
public class SidedMarketDataCombiner
        implements SidedMarketDataListener {

    final MarketDataListener mdListener;

    public SidedMarketDataCombiner(MarketDataListener mdListener) {
        this.mdListener = mdListener;
    }
```

Writing a simple component

The component calculates a result, using private state.

```
final Map<String, TopOfBookPrice> priceMap = new TreeMap<>();

public void onSidedPrice(SidedPrice sidedPrice) {
    TopOfBookPrice price = priceMap.computeIfAbsent(
                                    sidedPrice.symbol, TopOfBookPrice::new);
    if (price.combine(sidedPrice))
        mdListener.onTopOfBookPrice(price);
}
```

Testing our simple component

We can mock the output listener of our component.

```
MarketDataListener listener = createMock(MarketDataListener.class);
listener.onTopOfBookPrice(new TopOfBookPrice("EURUSD", 123456789000L,
                                             1.1167, 1_000_000, Double.NaN, 0));
listener.onTopOfBookPrice(new TopOfBookPrice("EURUSD", 123456789100L,
                                             1.1167, 1_000_000, 1.1172, 2_000_000));
replay(listener);

SidedMarketDataListener combiner = new SidedMarketDataCombiner(listener);
combiner.onSidedPrice(new SidedPrice("EURUSD", 123456789000L,
                                       Side.Buy, 1.1167, 1e6));
combiner.onSidedPrice(new SidedPrice("EURUSD", 123456789100L,
                                       Side.Sell, 1.1172, 2e6));

verify(listener);
```

Testing multiple components

We can mock the output listener of our component.

```
// what we expect to happen
OrderListener listener = createMock(OrderListener.class);

listener.onOrder(new Order("EURUSD", Side.Buy, 1.1167, 1_000_000));

replay(listener);

// build our scenario
OrderManager orderManager =
    new OrderManager(listener);

SidedMarketDataCombiner combiner =
    new SidedMarketDataCombiner(orderManager);
```

Testing multiple components

```
// events in: not expected to trigger
orderManager.onOrderIdea(
    new OrderIdea("EURUSD", Side.Buy, 1.1180, 2e6));

combiner.onSidedPrice(
    new SidedPrice("EURUSD", 123456789000L, Side.Sell, 1.1172, 2e6));
combiner.onSidedPrice(
    new SidedPrice("EURUSD", 123456789100L, Side.Buy, 1.1160, 2e6));
combiner.onSidedPrice(
    new SidedPrice("EURUSD", 123456789100L, Side.Buy, 1.1167, 2e6));

 // expected to trigger
orderManager.onOrderIdea(
    new OrderIdea("EURUSD", Side.Buy, 1.1165, 1e6));

verify(listener);
```

Adding a transport

Any messaging system can be used as a transport. You can use

- REST or HTTP
- JMS, Akka, MPI
- Aeron or a UDP based transport.
- Raw TCP or UDP.
- Chronicle Queue.

Making messages transparent

```
orderManager.onOrderIdea(
   new OrderIdea("EURUSD", Side.Buy, 1.1180, 2e6));
```

```
--- !!data #binary
onOrderIdea: {
    symbol: EURUSD,
    side: Buy,
    limitPrice: 1.118,
    quantity: 2000000.0
}
```

Why use Chronicle Queue

Chronicle Queue v4 has a number of advantages

- Broker less, only the OS needs to be up.
- Low latency, less than 10 microseconds 99% of the time.
- Persisted, giving your replay and transparency.
- Can replace your logging improving performance.
- Kernel Bypass, Shared across JVMs with a system call for each message.

```
--- !!meta-data #binary
header: !SCQStore { wireType: !WireType BINARY, writePosition: 777, roll: !SCQSRoll {
length: 86400000, format: yyyyMMdd, epoch: 0 }, indexing: !SCQSIndexing {
indexCount: !int 8192, indexSpacing: 64, index2Index: 0, lastIndex: 0 } }

# position: 227
--- !!data #binary
onOrderIdea: { symbol: EURUSD, side: Buy, limitPrice: 1.118, quantity: 2000000.0 }

# position: 306
--- !!data #binary
onTopOfBookPrice: { symbol: EURUSD, timestamp: 123456789000, buyPrice: NaN,
buyQuantity: 0, sellPrice: 1.1172, sellQuantity: 2000000.0 }

# position: 434
--- !!data #binary
onTopOfBookPrice: { symbol: EURUSD, timestamp: 123456789100, buyPrice: 1.116,
buyQuantity: 2000000.0, sellPrice: 1.1172, sellQuantity: 2000000.0 }

# position: 566
--- !!data #binary
onTopOfBookPrice: { symbol: EURUSD, timestamp: 123456789100, buyPrice: 1.1167,
buyQuantity: 2000000.0, sellPrice: 1.1172, sellQuantity: 2000000.0 }

# position: 698
--- !!data #binary
onOrderIdea: { symbol: EURUSD, side: Buy, limitPrice: 1.1165, quantity: 1000000.0 }
...
# 83885299 bytes remaining
```

Measuring the performance?

Measure the write latency with JMH (Java Microbenchmark Harness)

```
Percentiles, us/op:
      p(0.0000) =      2.552 us/op
     p(50.0000) =      2.796 us/op
     p(90.0000) =      5.600 us/op
     p(95.0000) =      5.720 us/op
     p(99.0000) =      8.496 us/op
     p(99.9000) =     15.232 us/op
     p(99.9900) =     19.977 us/op
     p(99.9990) =    422.475 us/op
     p(99.9999) =    438.784 us/op
    p(100.0000) =    438.784 us/op
```

FIX – Micro seconds customisable FIX Engine

Enterprise – Monitoring, Traffic Shaping, Security

Queue-Enterprise – Confirmed Replication Distributed Queue

Journal – Custom Data Store, Key-Queue

Engine – Customisable Data Fabric, Reactive Live Queries

Queue – Persist every event

Map – Persisted Key-Value

Wire – YAML, Binary YAML, JSON, CSV, Raw data

Network – Remote access

Bytes – 64-bit off heap native + memory mapped files

Threads – Low latency

Core – Low level access to OS and JVM

Chronicle

Where can I try this out?

Low Latency Microservices examples

https://github.com/Vanilla-Java/Microservices

The OSS Chronicle products are available

https://github.com/OpenHFT/

Chronicle

Q & A

Blog: http://vanilla-java.github.io/

http://chronicle.software

@ChronicleUG

sales@chronicle.software

https://groups.google.com/forum/#!forum/java-chronicle