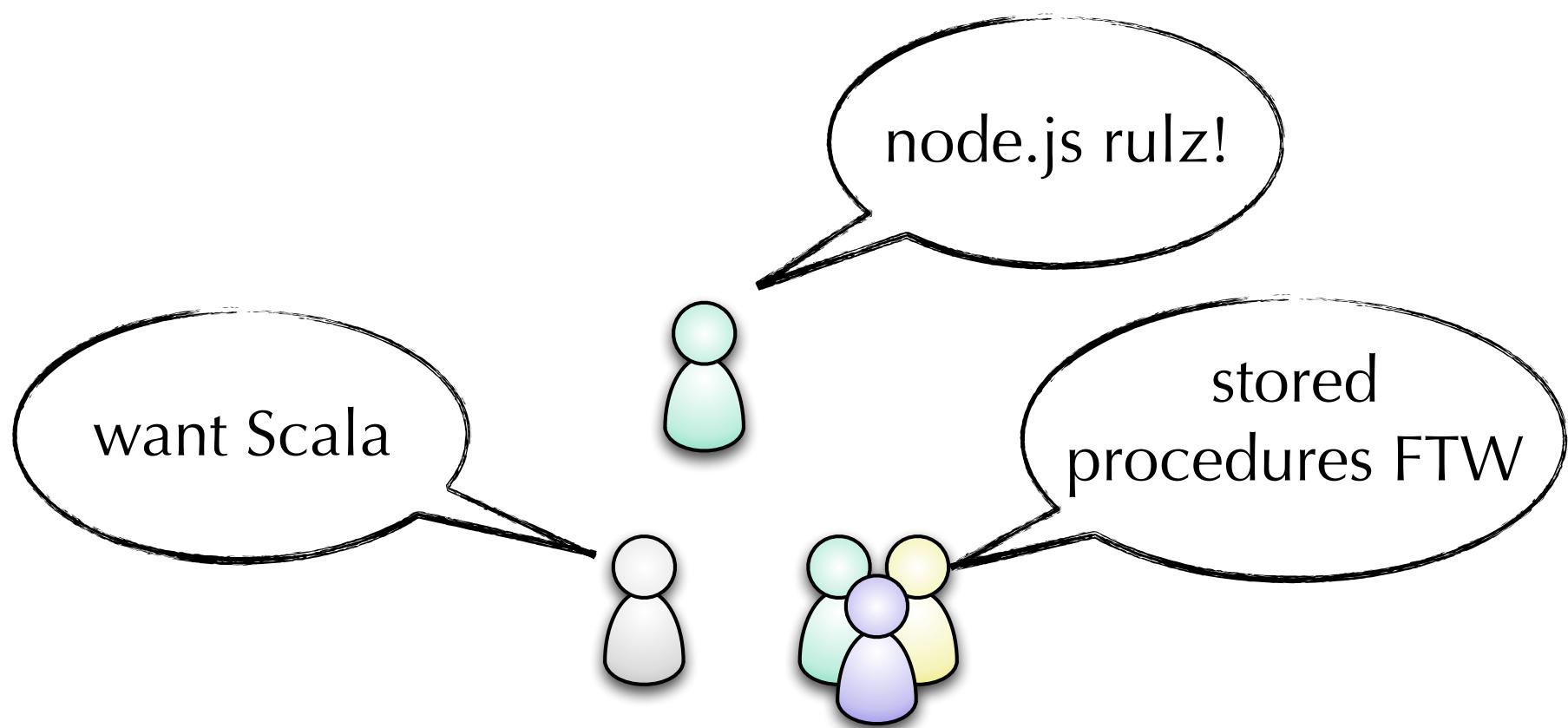
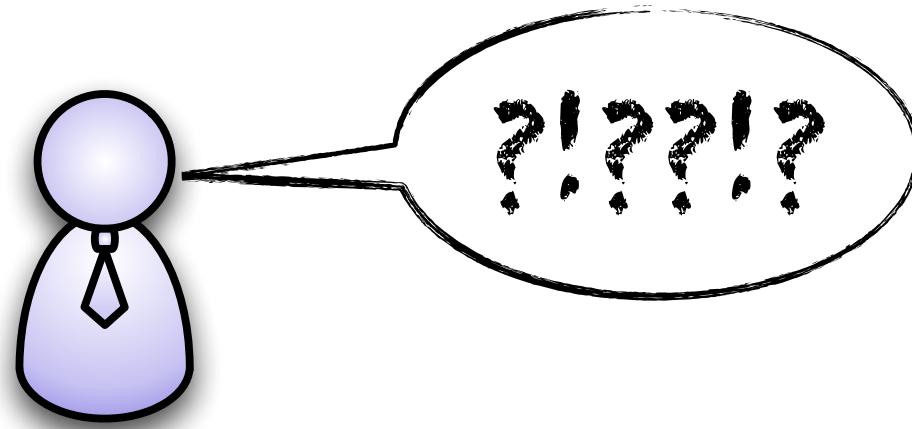
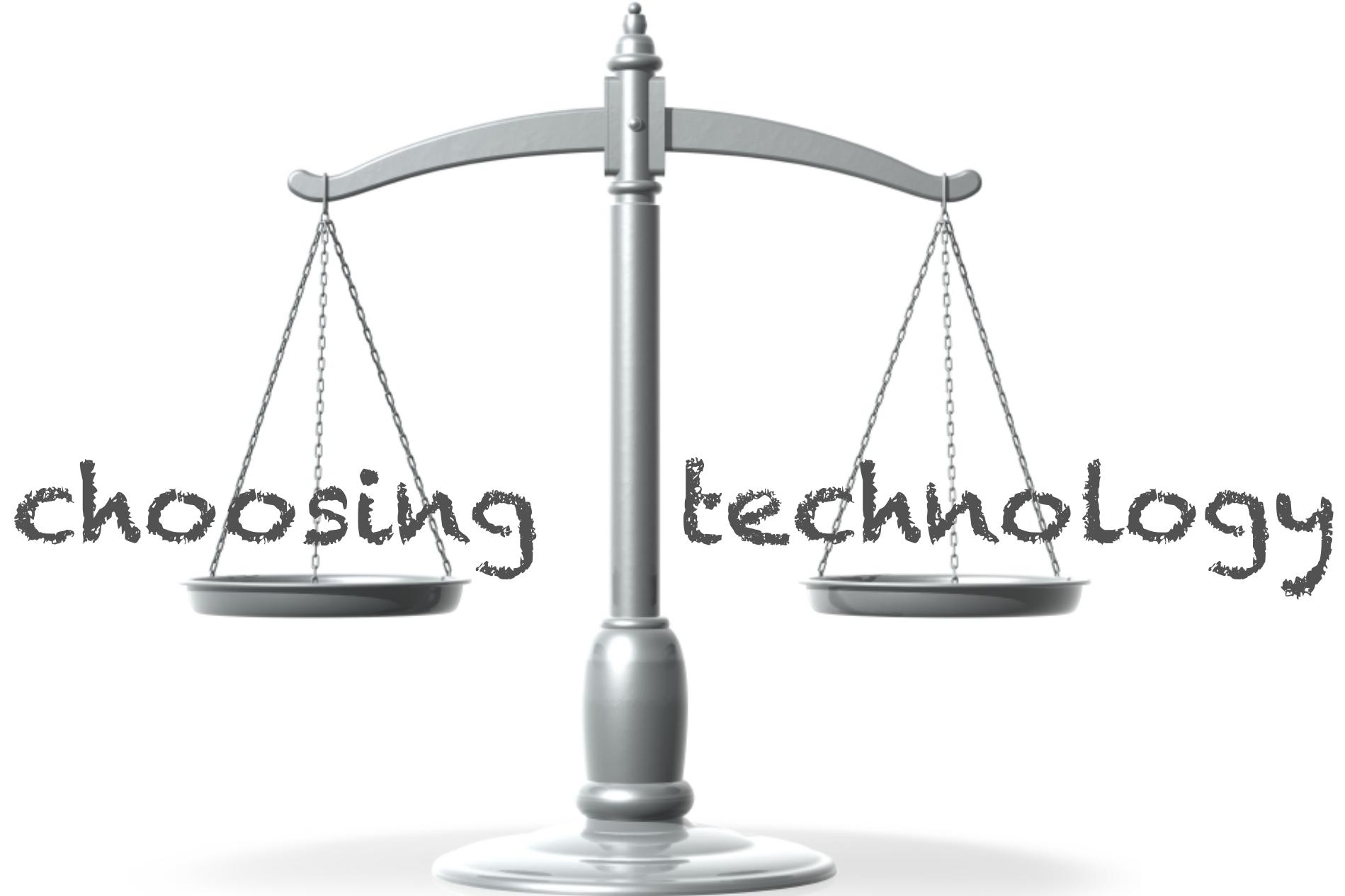


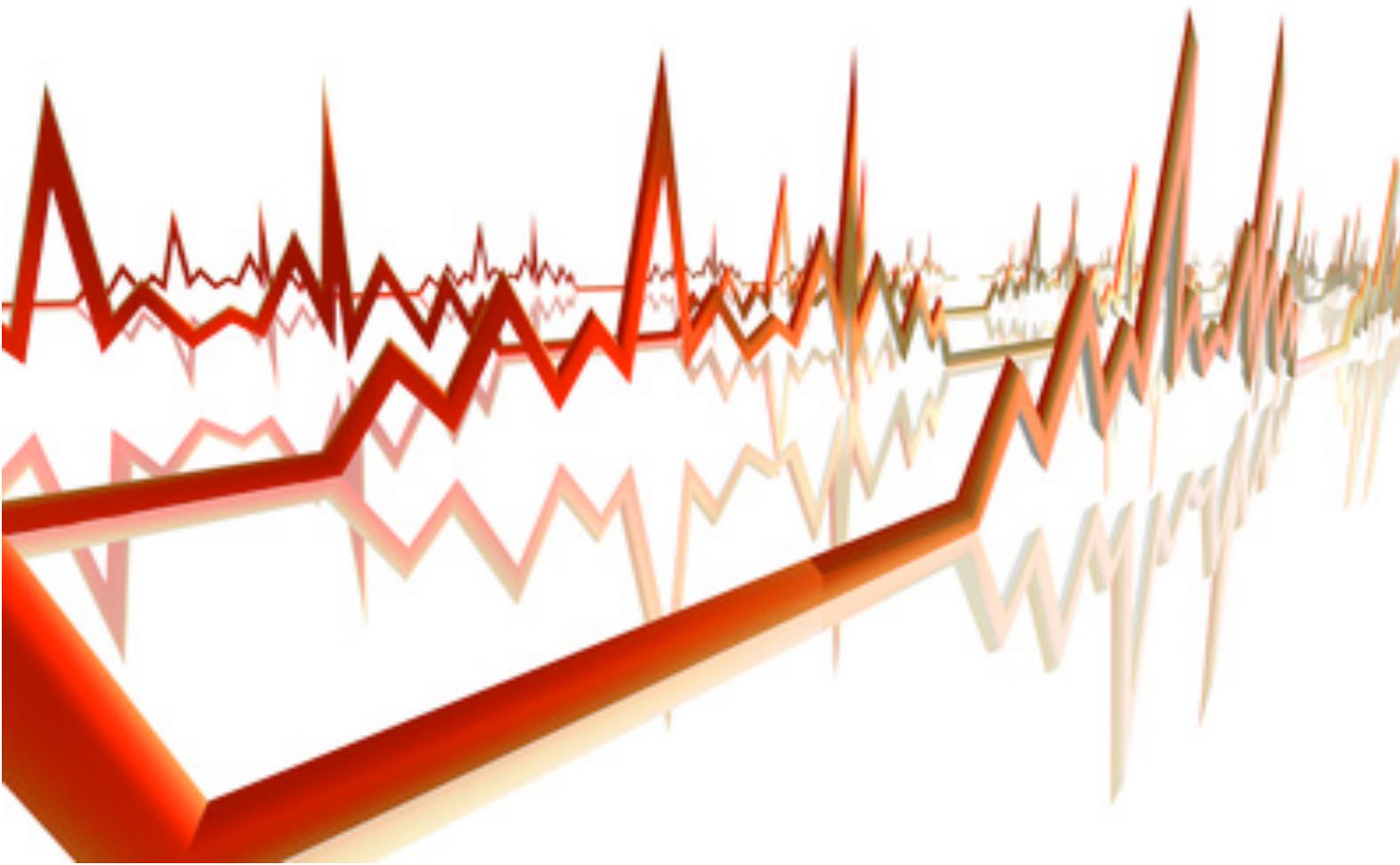
comparing architectures







understand current trends



normalize feature sets



build your own feature matrix

good, bad, ugly

avoid the “suck/rock” dichotomy



nealford.com • The Suck/Rock Dichotomy

Home | Presentation Patterns now available. Read more... [Reader](#)

JF
nealford.com

Downloads, Past Conferences Biography Books Video Blog, Article Series Abstracts

The Suck/Rock Dichotomy

Lots of people are passionate about software development (much to the confusion and chagrin of our significant others), and that unfortunately leads to what I call the “Suck/Rock Dichotomy”: everything in the software world either sucks or rocks, with nothing in between. While this may lead to interesting, endless debates (*Emacs vs. vi*, anyone?), ultimately it ill serves us as a community.

Having been in software communities for a while, I've seen several tribes form, thrive, then slowly die. It's a sad thing to watch a community die because many of the people in the community live in a state of denial: how could their wonderful thing (which rocks) disappear under this other hideous, inelegant, terrible thing (which sucks). I was part of the *Clipper* community (which I joined at its height) and watched it die rather rapidly when Windows ate DOS. I was intimately part of the Delphi community which, while not dead yet, is rapidly approaching death. When a community fades, the fanaticism of the remaining members increases proportionally for every member they lose, until you are left with one person whose veins stick out on their forehead when they try to proselytize people to join their tribe, which rocks, and leave that other tribe, which sucks.

Why is this dichotomy so stark in the software development world? I suspect a couple of root causes. First, because it takes a non-trivial time investment for proficiency in software tribes, people fear that they have chosen poorly and thus wasted their time. Perhaps the degree in which something rocks is proportional to the time investment in learning that technology. Second, technologists and particularly developers stereotypically tend to socialize via tribal ritual. How many software development teams have you seen that are not too far removed from fraternities? Because software is fundamentally a communication game, I think that the fraternal nature of most projects makes it easier to write good software. But tribal ritual implies that one of the defining characteristics of your tribe is the denigration of other tribes (we rock, they suck). In fact, some tribes within software seem to define themselves in how loudly they can say that everything sucks, except of course their beautiful thing, which rocks.

Some communities try to purposefully pick fights with others just so they can thump their collective chests over how much they rock compared to how much the other guys suck. Of course, you get camps that are truly different in many, many ways (*Emacs vs. vi*, anyone?) But you also see this in communities that are quite similar; one of the most annoying characteristics of some communities is how much some a few of their members try to bait other communities that aren't interested in fighting.

The Suck/Rock Dichotomy hurts us because it obscures legitimate conversations about the real differences between things. Truly balanced comparisons are rare (for an outstanding example of a balanced, well considered, sober comparison of Prototype and JQuery, check out [Glenn Vanderburg's post](#)). I try to avoid this dichotomy (some would say with varying degrees of success). For example, for the past 2 years, I've done a Comparing Groovy & JRuby talk at JavaOne, and it's been mostly well received by members of both communities. Putting together such a talk or blog entry takes a lot of effort, though: you have to learn not just the surface area details of said technologies, but how to use it idiomatically as well, which takes time. I suspect that's why you don't see more nuanced comparisons: it's a lot easier to resort to either suck or rock.

Ultimately, we need informed debates about the relative merits of various choices. The Suck/Rock Dichotomy adds heat but not much light. Technologists marginalize our influence within organizations because the non-techie hear us endless debating stuff that sounds like arguments over how many *angels can dance on the head of a pin*. If we argue about seemingly trivial things like that, then why listen to us when we passionately argue about stuff that is immediately important, like technical debt or why we can't disprove [The Mythical Man Month](#) on this project. To summarize: the Suck/Rock Dichotomy sucks!

Go to "<http://nealford.com/bio>"

nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html



spikes



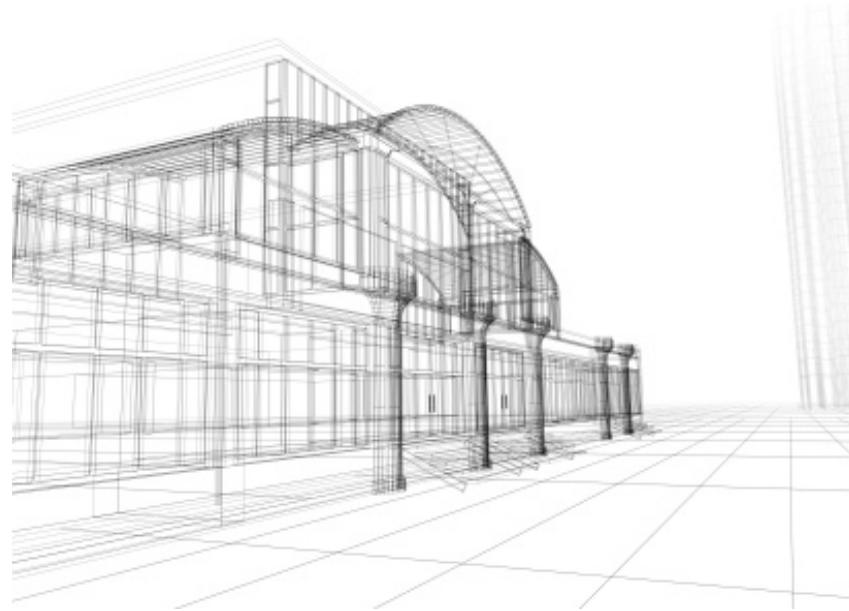
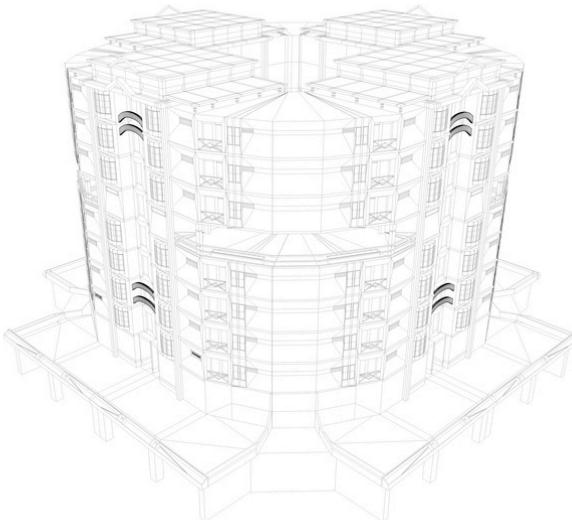
time-boxed experimental coding exercise

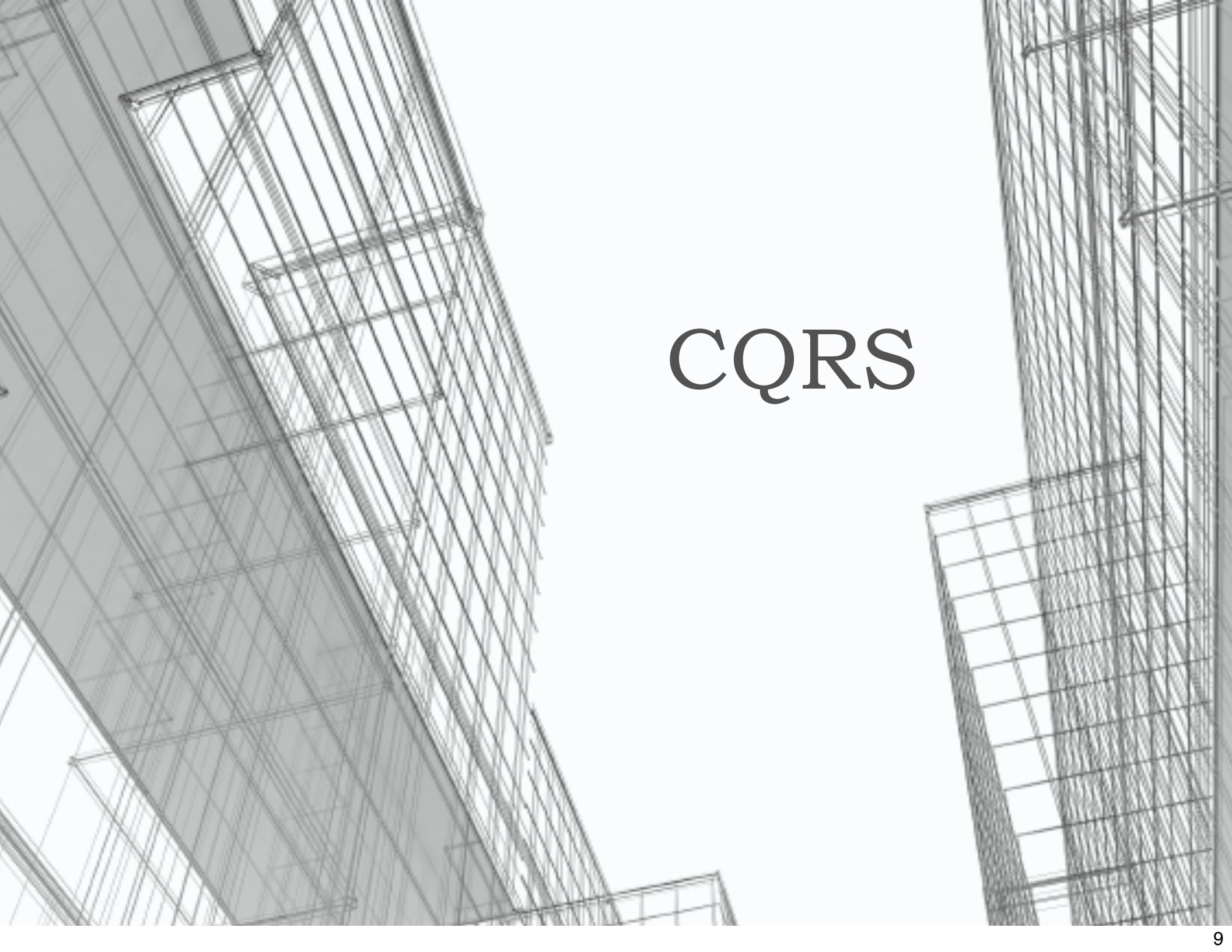
pure knowledge acquisition

NOT a prototype!

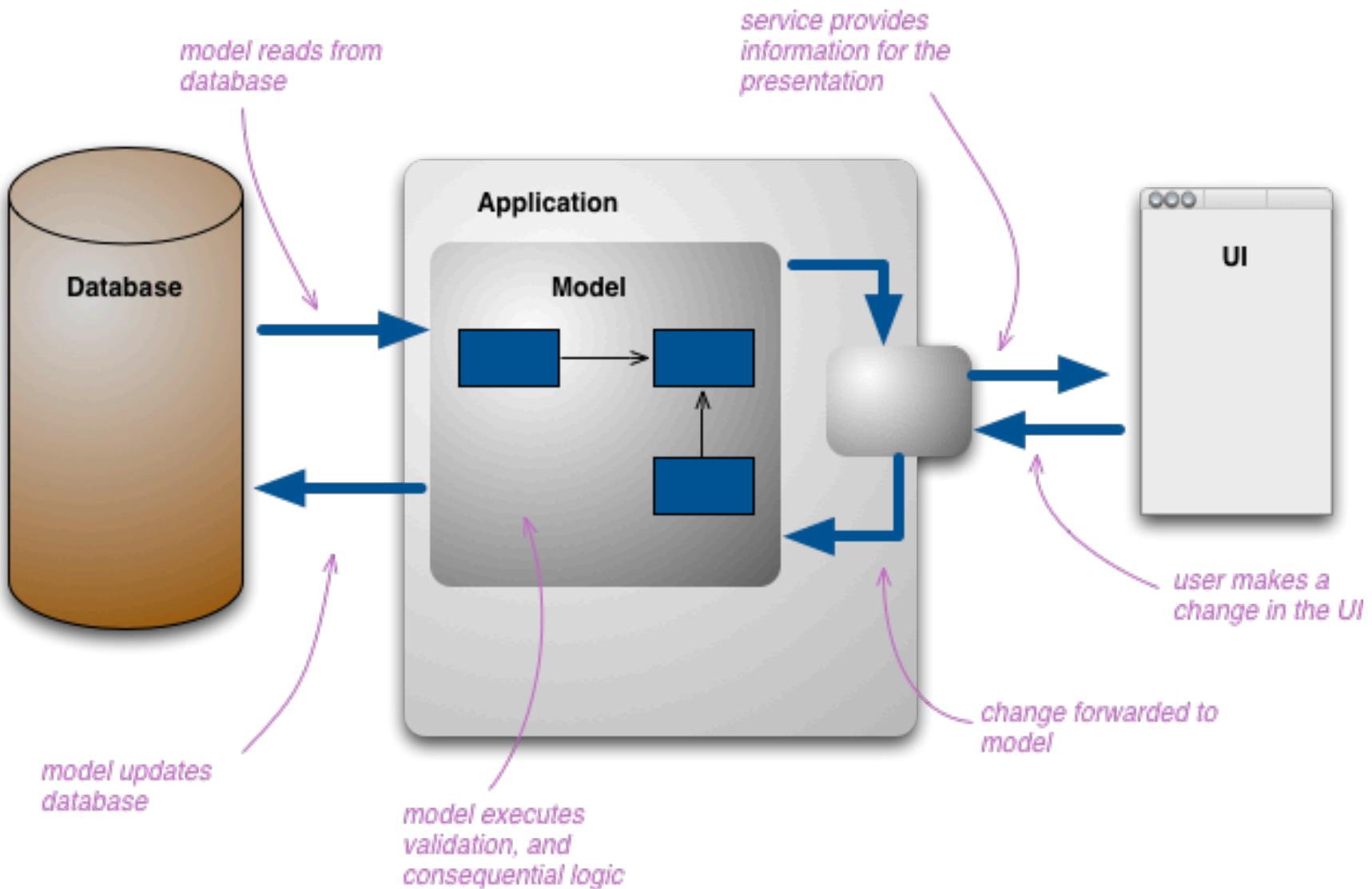
developed on a “dead end” branch

comparing architectures

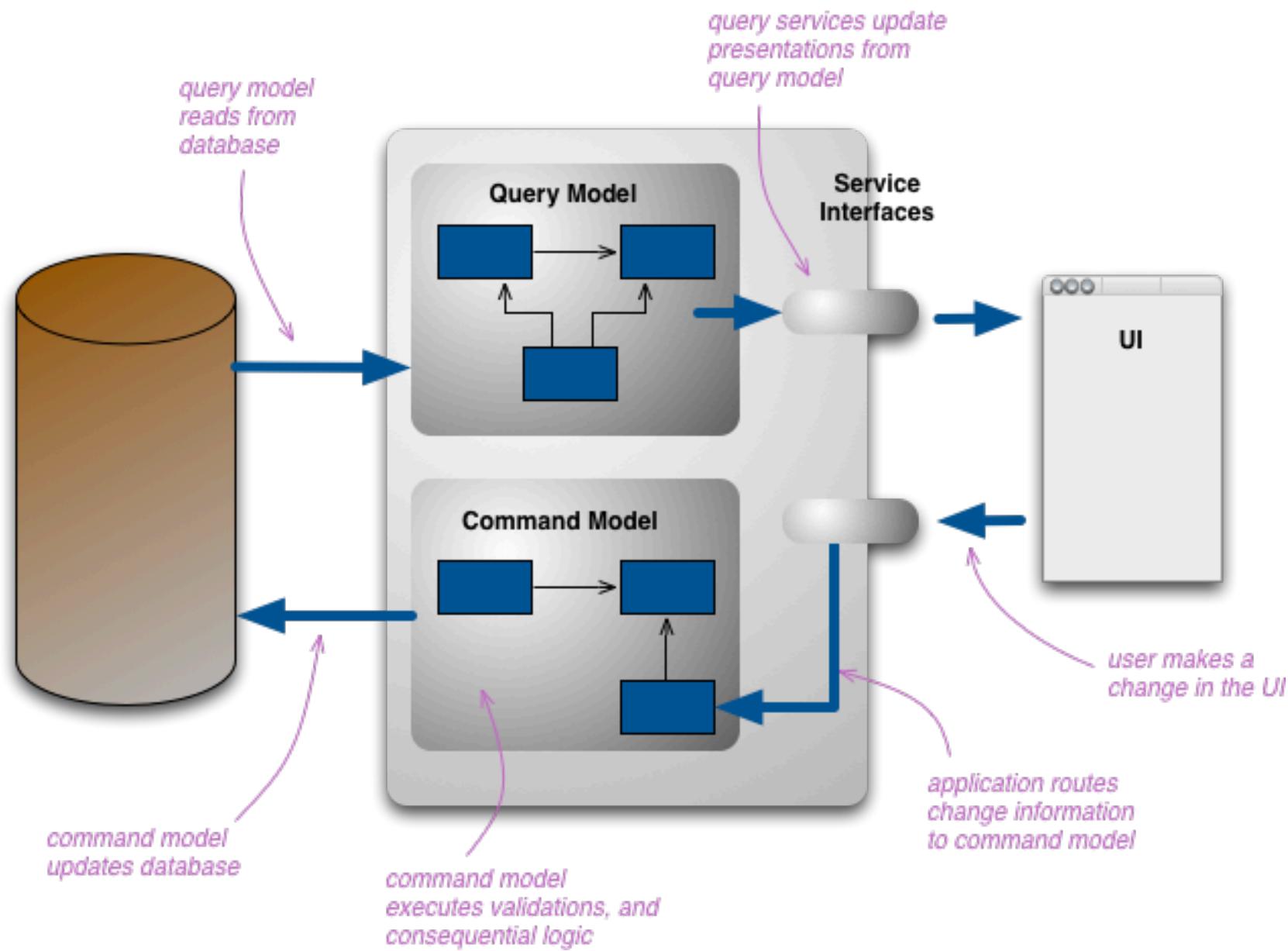


A complex, abstract wireframe structure composed of many thin, dark lines forming a grid-like pattern. It has a perspective effect, appearing as a large, slanted rectangular volume on the left and a smaller, vertical rectangular volume on the right.

CQRS



traditional



CQRS Command Query Responsibility Separation

CQRS natural fits

task-based user interface

meshes well with event sourcing

eventual consistency

eventual consistency

Eventually Consistent – Revisited – All Things Distributed
Werner Vogels' weblog on building scalable and robust distributed systems.

Eventually Consistent - Revisited

By Werner Vogels on 22 December 2008 04:15 PM | [Permalink](#) | [Comments \(14\)](#)

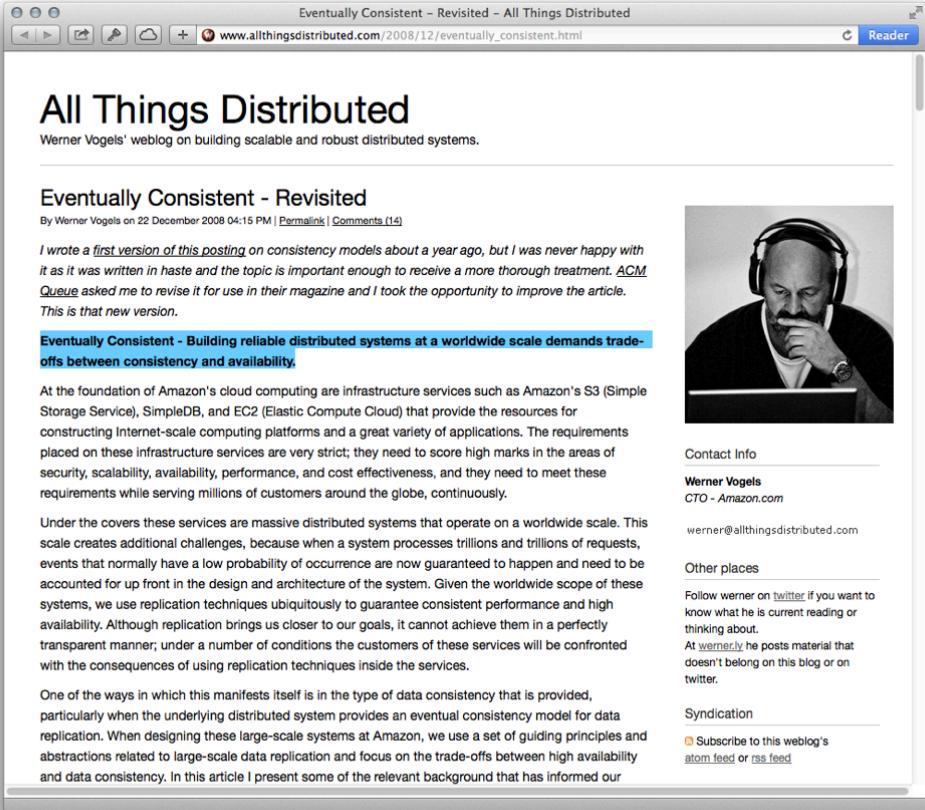
I wrote a [first version of this posting](#) on consistency models about a year ago, but I was never happy with it as it was written in haste and the topic is important enough to receive a more thorough treatment. [ACM Queue](#) asked me to revise it for use in their magazine and I took the opportunity to improve the article. This is that new version.

Eventually Consistent - Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

At the foundation of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and need to be accounted for up front in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when the underlying distributed system provides an eventual consistency model for data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. In this article I present some of the relevant background that has informed our



“Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.”

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

CQRS natural fits

task-based user interface

meshes well with event sourcing

eventual consistency

consistency or availability
(but never both)

complex or granular domains

domain logic classification

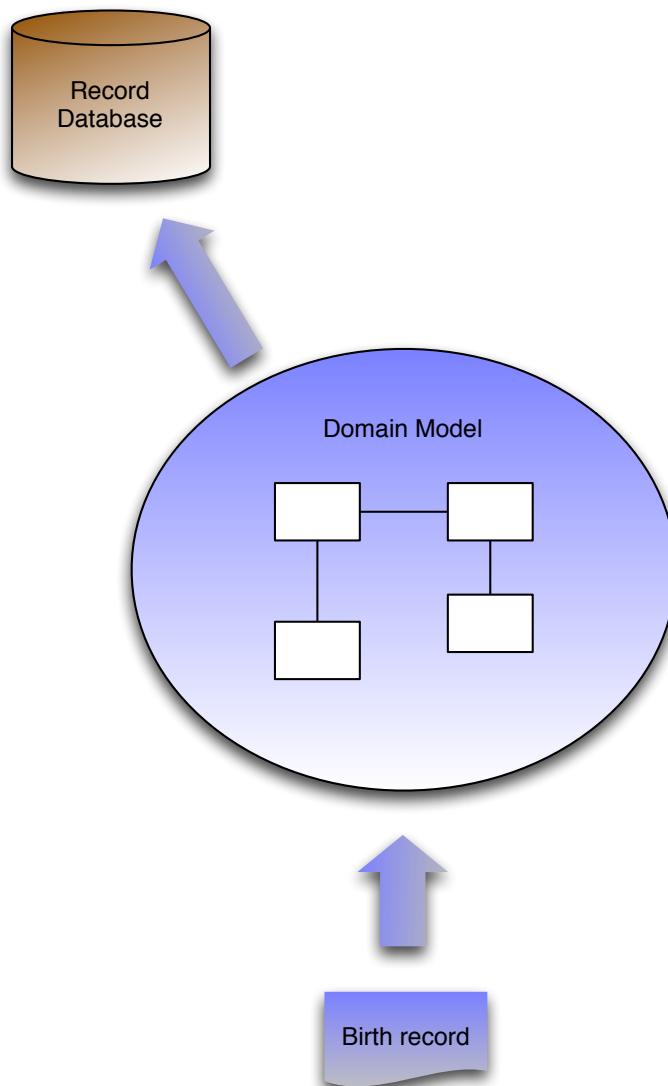
<http://martinfowler.com/bliki/EagerReadDerivation.html>

validations: sensible inputs, valid objects

consequences: initiating a world-changing action

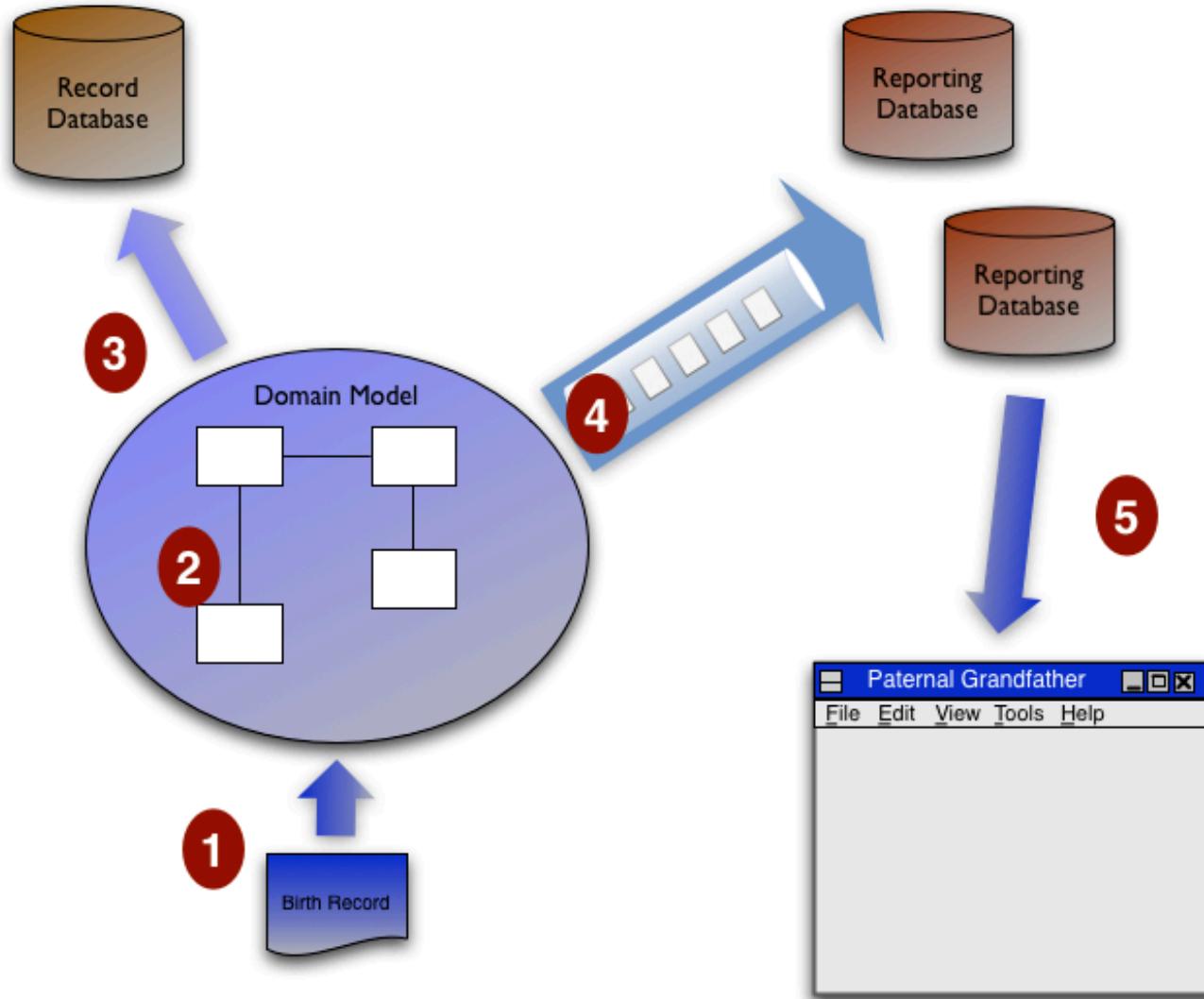
derivations: figuring out some information based on information we already have

typical case

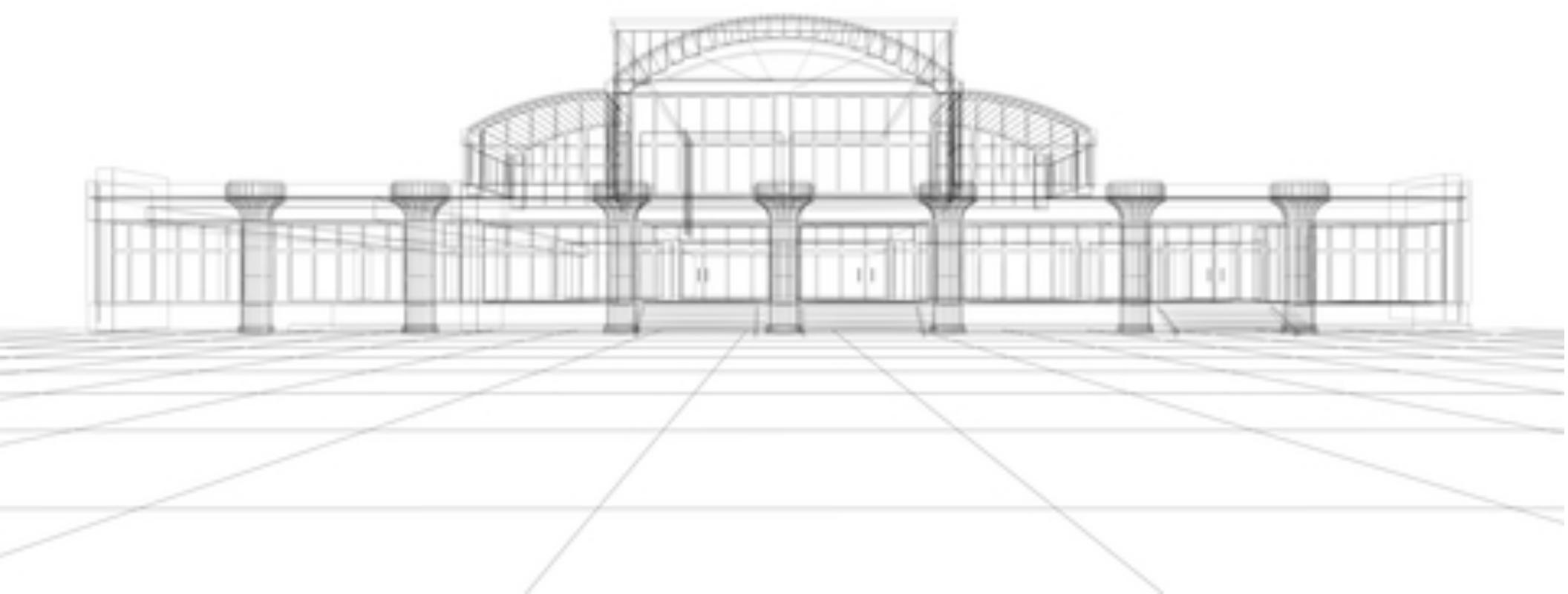


EagerReadDerivation

<http://martinfowler.com/bliki/EagerReadDerivation.html>



LMAX



LMAX

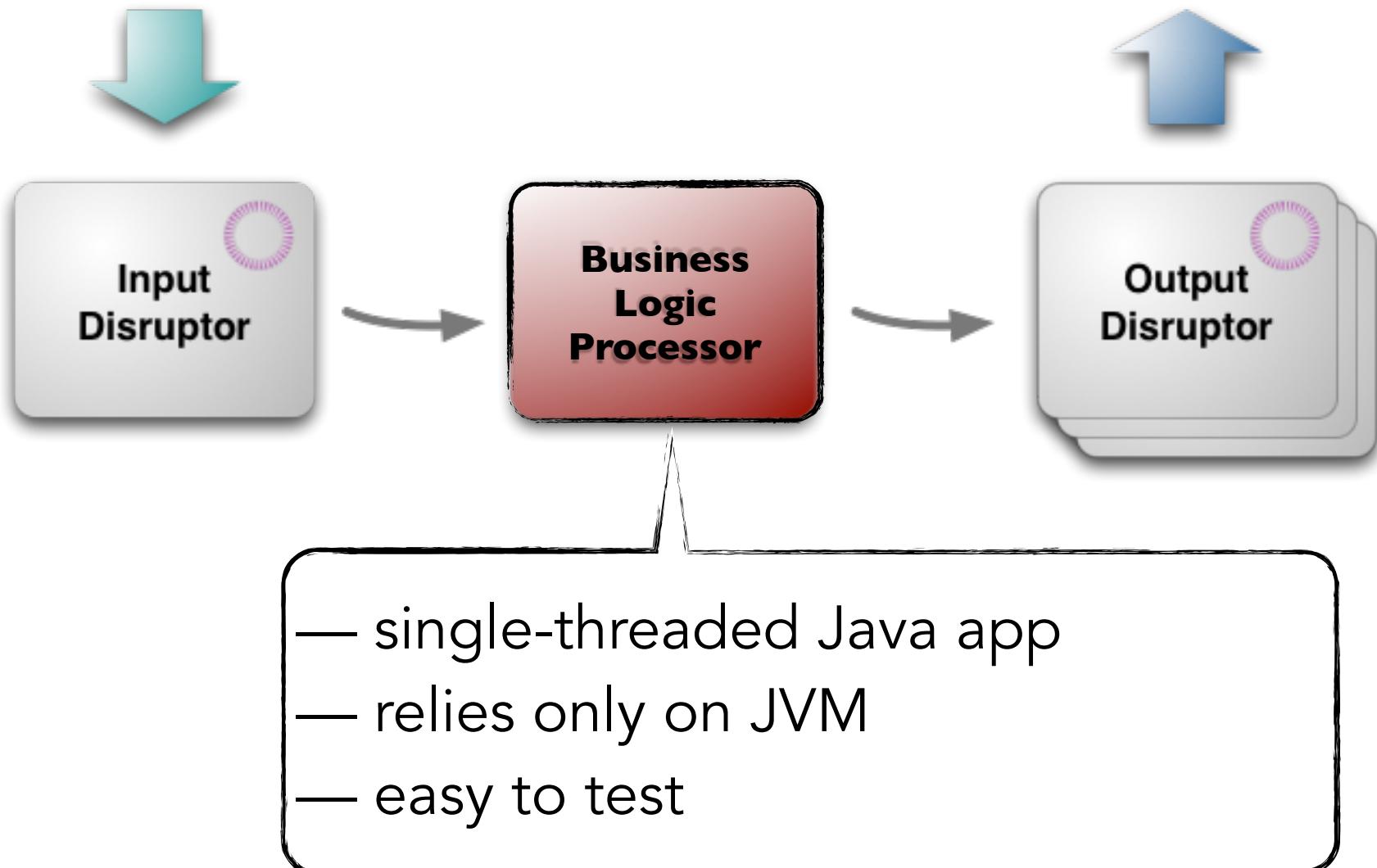
<http://martinfowler.com/articles/lmax.html>

JVM-based retail financial trading platform

centers on Business Logic Processor
handling 6,000,000 orders/sec on 1 thread

surrounded by Disruptors, network of
lock-less queues

overall structure



business logic processor

in-memory

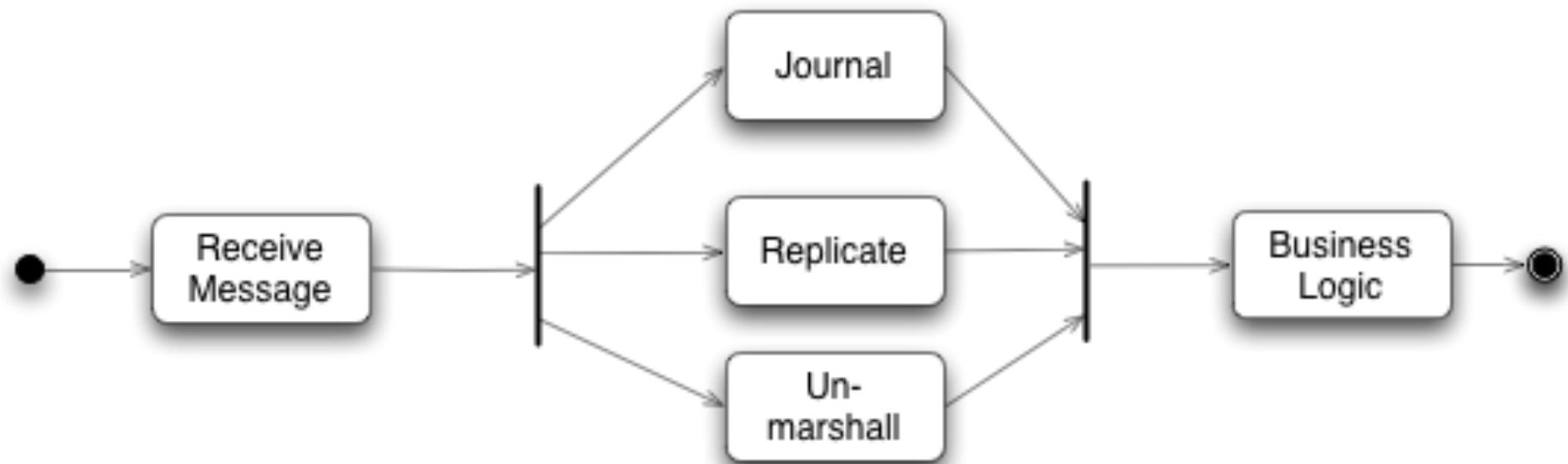
event sourcing via input disruptor

snapshots (full restart—JVM + snapshots—less than 1 min)

multiple instances running

each event processed by multiple processors but only one result used

input/output disruptors



disruptors

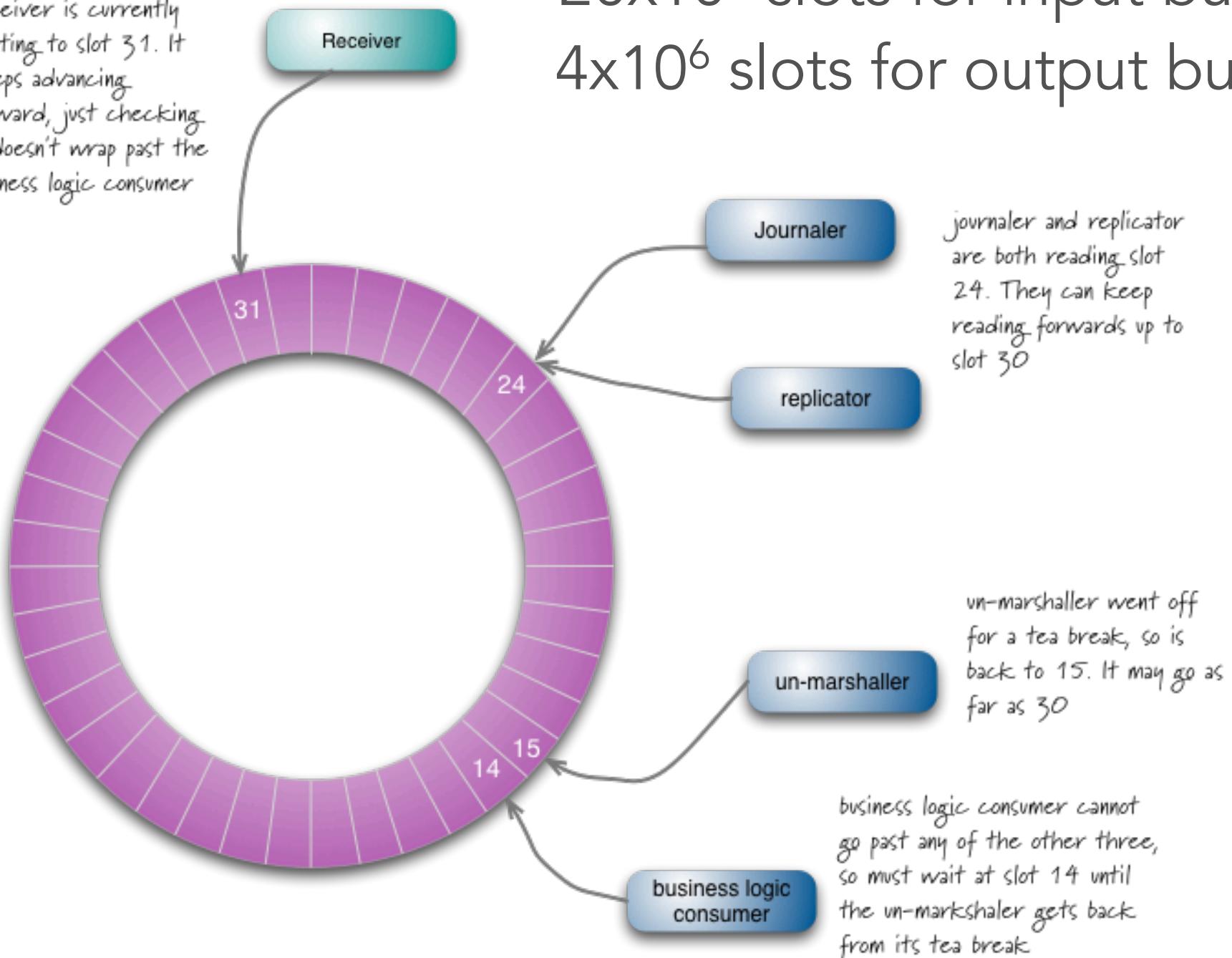
custom concurrency component

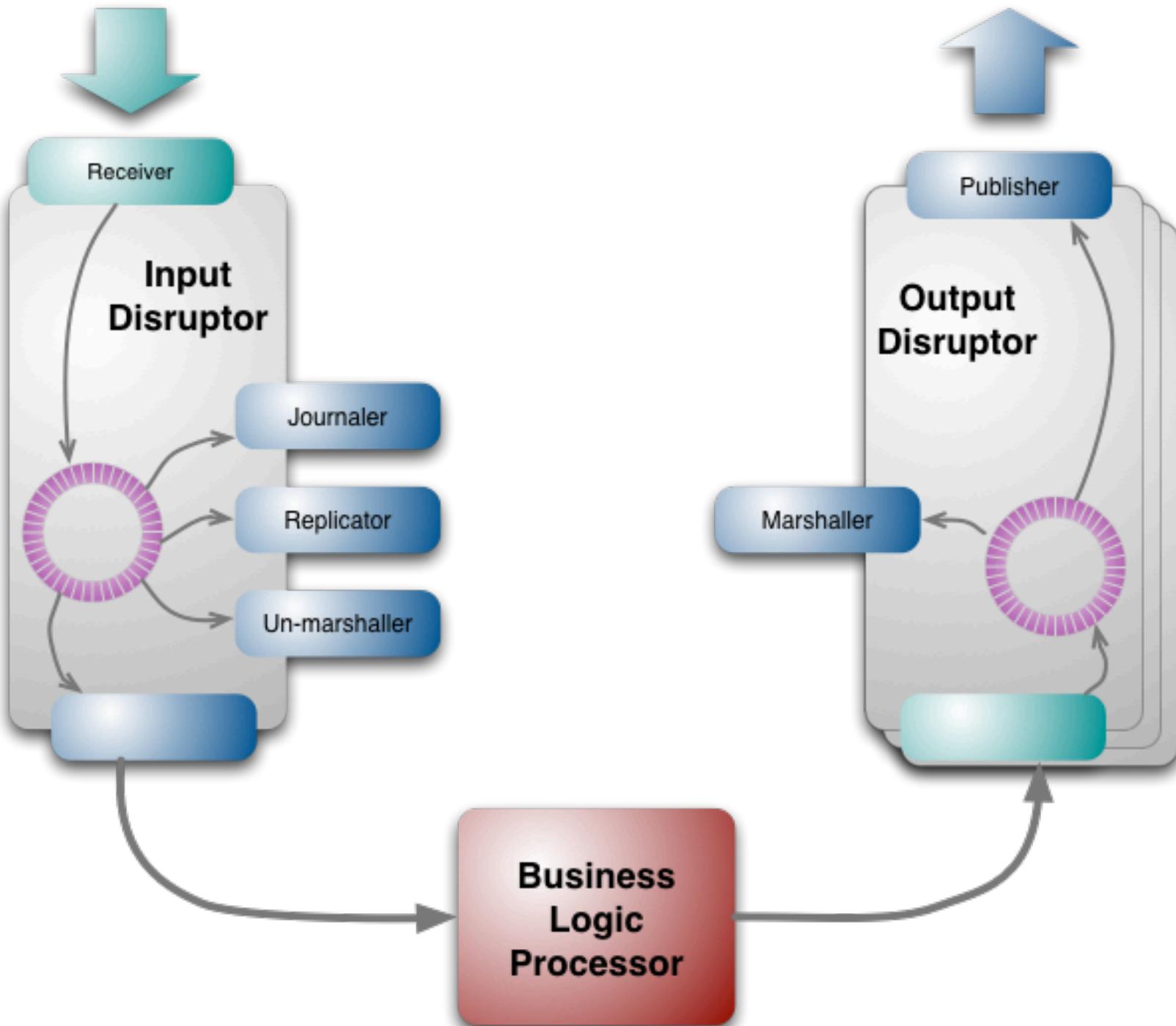
multi-cast graph of queues where
producers enqueue objects and consumers
dequeue in parallel

ring buffer with sequence counters

receiver is currently writing to slot 31. It keeps advancing forward, just checking it doesn't wrap past the business logic consumer

20×10^6 slots for input buffer
 4×10^6 slots for output buffer





“mechanical sympathy”

started with transactions

switched to Actor-based concurrency

hypothesized & measured results

CPU caching is key ➔ single writer principle

when would you use this
architecture?

when would you avoid this architecture?

IT
DEPENDS !

“One ring to rule them all...”

no architecture fits every circumstance

evaluate good, bad, and ugly

watch for primrose paths that turn ugly

embrace pragmatism

what's your day job?

building wicked cool architectures is an
stimulating, rewarding intellectual
challenge...

...building CRUD applications isn't
coolness sometimes == accidental
complexity



architect for
change vs YAGNI

? ' S



Mark Richards

Independent Consultant

Hands-on Enterprise / Integration Architect

Published Author / Conference Speaker

<http://www.wmrichards.com>

<http://www.linkedin.com/pub/mark-richards/0/121/5b9>

Published Books:

Java Message Service, 2nd Edition

97 Things Every Software Architect Should Know

Java Transaction Design Strategies



Neal Ford

Director / Software Architect /

Meme Wrangler

ThoughtWorks®

2002 Summit Blvd, Level 3, Atlanta, GA 30319, USA

T: +1 404 242 9929 Twitter: @neal4d

E: nford@thoughtworks.com W: thoughtworks.com