Main sponsor

LUXOFT
Engineering Business Performance

# Micro Services – Java the Unix way

James Lewis

James Lewis

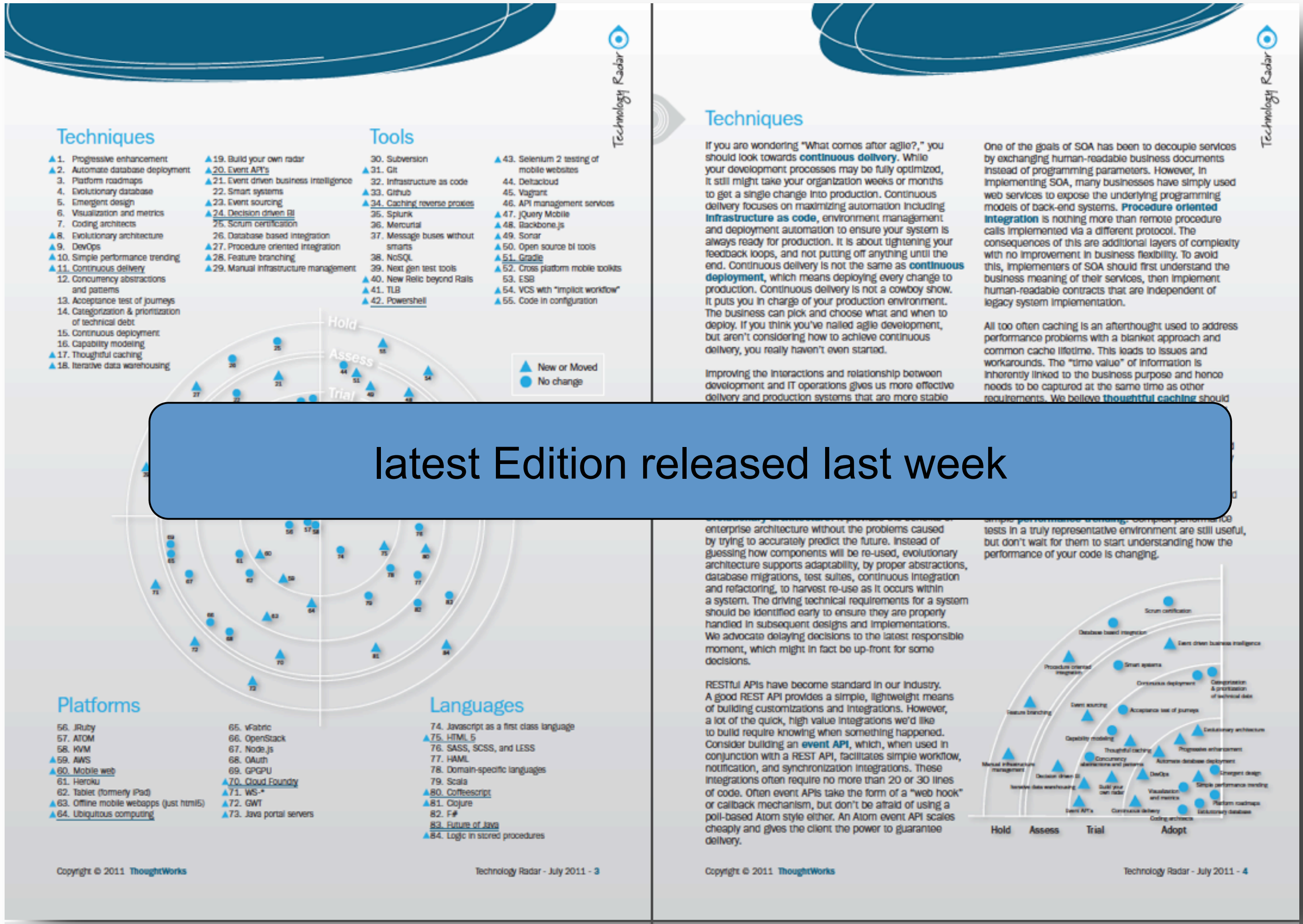
Senior Engineer / Principal Consultant / TAB member

Your Storyteller today

# ThoughtWorks® radar

Technology Radar
http://www.thoughtworks.com/radar
Prepared by the ThoughtWorks Technical Advisory Board
August 2010
ThoughtWorks®

## latest Edition released last week



## Techniques

▲1. Progressive enhancement
▲2. Automate database deployment
3. Platform roadmaps
4. Evolutionary database
5. Emergent design
6. Visualization and metrics
7. Coding architects
▲8. Evolutionary architecture
▲9. DevOps
▲10. Simple performance trending
▲11. Continuous delivery
12. Concurrency abstractions and patterns
13. Acceptance test of journeys
14. Categorization & prioritization of technical debt
15. Continuous deployment
16. Capability modeling
▲17. Thoughtful caching
▲18. Iterative data warehousing
▲19. Build your own radar
▲20. Event API's
▲21. Event driven business intelligence
22. Smart systems
▲23. Event sourcing
▲24. Decision driven BI
25. Scrum certification
26. Database based integration
▲27. Procedure oriented integration
▲28. Feature branching
▲29. Manual infrastructure management

## Tools

30. Subversion
▲31. Git
32. Infrastructure as code
▲33. Github
▲34. Caching reverse proxies
35. Splunk
36. Mercurial
37. Message buses without smarts
38. NoSQL
39. Next gen test tools
▲40. New Relic beyond Rails
▲41. TLB
▲42. Powershell
▲43. Selenium 2 testing of mobile websites
44. Deltacloud
45. Vagrant
46. API management services
▲47. jQuery Mobile
▲48. Backbone.js
▲49. Sonar
▲50. Open source bi tools
▲51. Gradle
▲52. Cross platform mobile toolkits
53. ESB
▲54. VCS with "implicit workflow"
▲55. Code in configuration

▲ New or Moved
● No change

## Platforms

56. JRuby
57. ATOM
58. KVM
▲59. AWS
▲60. Mobile web
61. Heroku
62. Tablet (formerly iPad)
▲63. Offline mobile webapps (just html5)
▲64. Ubiquitous computing
65. vFabric
66. OpenStack
67. Node.js
68. OAuth
69. GPGPU
▲70. Cloud Foundry
▲71. WS-*
▲72. GWT
▲73. Java portal servers

## Languages

74. Javascript as a first class language
▲75. HTML 5
76. SASS, SCSS, and LESS
77. HAML
78. Domain-specific languages
79. Scala
▲80. Coffeescript
▲81. Clojure
82. F#
83. Future of Java
▲84. Logic in stored procedures

Copyright © 2011 ThoughtWorks

Technology Radar - July 2011 - 3

## Techniques

If you are wondering "What comes after agile?," you should look towards **continuous delivery**. While your development processes may be fully optimized, it still might take your organization weeks or months to get a single change into production. Continuous delivery focuses on maximizing automation including **infrastructure as code**, environment management and deployment automation to ensure your system is always ready for production. It is about tightening your feedback loops, and not putting off anything until the end. Continuous delivery is not the same as **continuous deployment**, which means deploying every change to production. Continuous delivery is not a cowboy show. It puts you in charge of your production environment. The business can pick and choose what and when to deploy. If you think you've nailed agile development, but aren't considering how to achieve continuous delivery, you really haven't even started.

Improving the interactions and relationship between development and IT operations gives us more effective delivery and production systems that are more stable

enterprise architecture without the problems caused by trying to accurately predict the future. Instead of guessing how components will be re-used, evolutionary architecture supports adaptability, by proper abstractions, database migrations, test suites, continuous integration and refactoring, to harvest re-use as it occurs within a system. The driving technical requirements for a system should be identified early to ensure they are properly handled in subsequent designs and implementations. We advocate delaying decisions to the latest responsible moment, which might in fact be up-front for some decisions.

RESTful APIs have become standard in our industry. A good REST API provides a simple, lightweight means of building customizations and integrations. However, a lot of the quick, high value integrations we'd like to build require knowing when something happened. Consider building an **event API**, which, when used in conjunction with a REST API, facilitates simple workflow, notification, and synchronization integrations. These integrations often require no more than 20 or 30 lines of code. Often event APIs take the form of a "web hook" or callback mechanism, but don't be afraid of using a poll-based Atom style either. An Atom event API scales cheaply and gives the client the power to guarantee delivery.

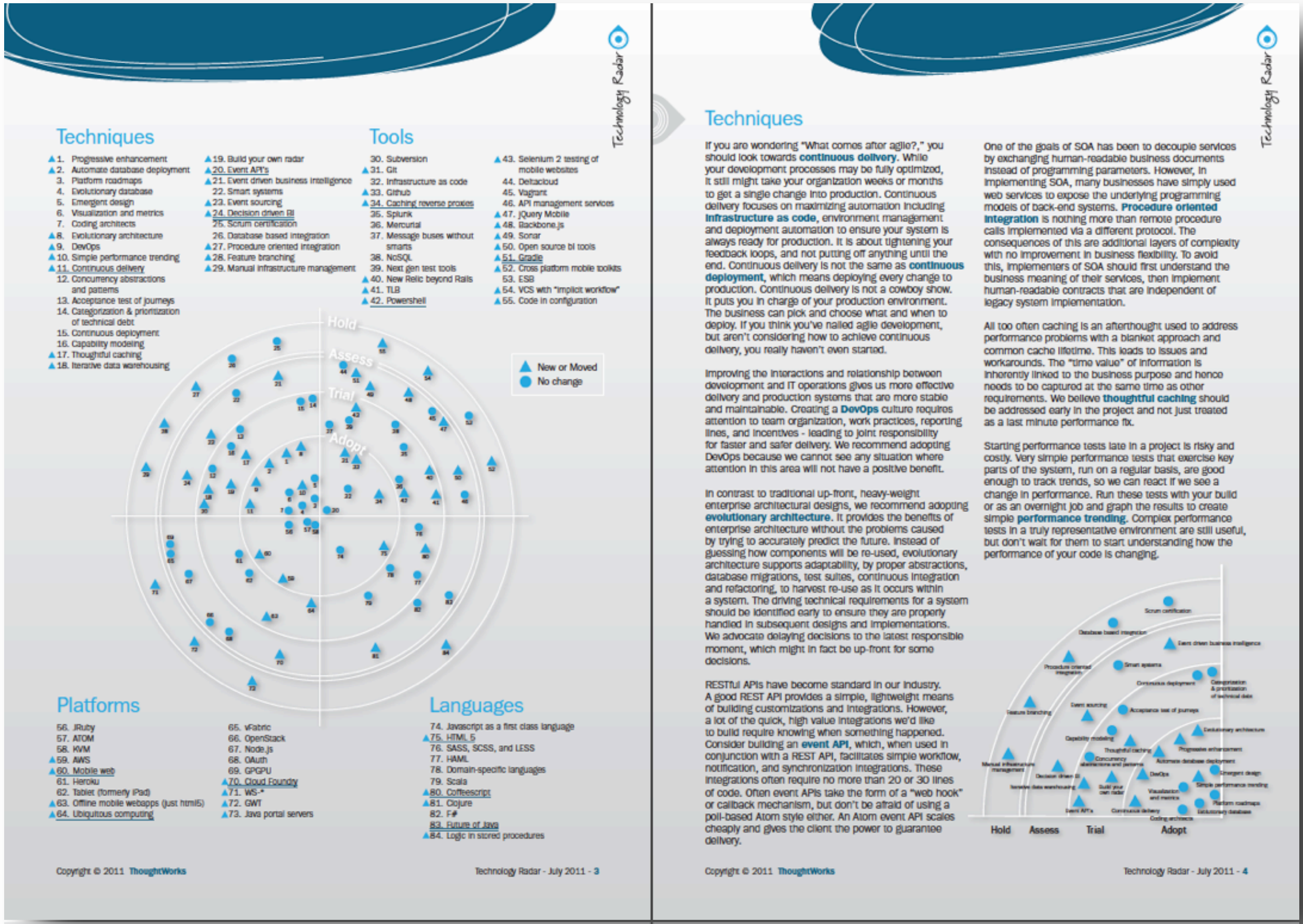
One of the goals of SOA has been to decouple services by exchanging human-readable business documents instead of programming parameters. However, in implementing SOA, many businesses have simply used web services to expose the underlying programming models of back-end systems. **Procedure oriented integration** is nothing more than remote procedure calls implemented via a different protocol. The consequences of this are additional layers of complexity with no improvement in business flexibility. To avoid this, implementers of SOA should first understand the business meaning of their services, then implement human-readable contracts that are independent of legacy system implementation.

All too often caching is an afterthought used to address performance problems with a blanket approach and common cache lifetime. This leads to issues and workarounds. The "time value" of information is inherently linked to the business purpose and hence needs to be captured at the same time as other requirements. We believe **thoughtful caching** should

tests in a truly representative environment are still useful, but don't wait for them to start understanding how the performance of your code is changing.

Copyright © 2011 ThoughtWorks

Technology Radar - July 2011 - 4



## Techniques

▲1. Progressive enhancement
▲2. Automate database deployment
3. Platform roadmaps
4. Evolutionary database
5. Emergent design
6. Visualization and metrics
7. Coding architects
▲8. Evolutionary architecture
▲9. DevOps
▲10. Simple performance trending
▲11. Continuous delivery
12. Concurrency abstractions and patterns
13. Acceptance test of journeys
14. Categorization & prioritization of technical debt
15. Continuous deployment
16. Capability modeling
▲17. Thoughtful caching
▲18. Iterative data warehousing
▲19. Build your own radar
▲20. Event API's
▲21. Event driven business intelligence
22. Smart systems
▲23. Event sourcing
▲24. Decision driven BI
25. Scrum certification
26. Database based integration
▲27. Procedure oriented integration
▲28. Feature branching
▲29. Manual infrastructure management

## Tools

30. Subversion
▲31. Git
32. Infrastructure as code
▲33. Github
▲34. Caching reverse proxies
35. Splunk
36. Mercurial
37. Message buses without smarts
38. NoSQL
39. Next gen test tools
▲40. New Relic beyond Rails
▲41. TLB
▲42. Powershell
▲43. Selenium 2 testing of mobile websites
44. Deltacloud
45. Vagrant
46. API management services
▲47. jQuery Mobile
▲48. Backbone.js
▲49. Sonar
▲50. Open source bi tools
▲51. Gradle
▲52. Cross platform mobile toolkits
53. ESB
▲54. VCS with "implicit workflow"
▲55. Code in configuration

## Platforms

56. JRuby
57. ATOM
58. KVM
▲59. AWS
▲60. Mobile web
61. Heroku
62. Tablet (formerly iPad)
▲63. Offline mobile webapps (just html5)
▲64. Ubiquitous computing
65. vFabric
66. OpenStack
67. Node.js
68. OAuth
69. GPGPU
▲70. Cloud Foundry
▲71. WS-*
▲72. GWT
▲73. Java portal servers

## Languages

74. Javascript as a first class language
▲75. HTML 5
76. SASS, SCSS, and LESS
77. HAML
78. Domain-specific languages
79. Scala
▲80. Coffeescript
▲81. Clojure
82. F#
83. Future of Java
▲84. Logic in stored procedures

Copyright © 2011 ThoughtWorks

Technology Radar - July 2011 - 3

## Techniques

If you are wondering "What comes after agile?," you should look towards **continuous delivery**. While your development processes may be fully optimized, it still might take your organization weeks or months to get a single change into production. Continuous delivery focuses on maximizing automation including **infrastructure as code**, environment management and deployment automation to ensure your system is always ready for production. It is about tightening your feedback loops, and not putting off anything until the end. Continuous delivery is not the same as **continuous deployment**, which means deploying every change to production. Continuous delivery is not a cowboy show. It puts you in charge of your production environment. The business can pick and choose what and when to deploy. If you think you've nailed agile development, but aren't considering how to achieve continuous delivery, you really haven't even started.

Improving the interactions and relationship between development and IT operations gives us more effective delivery and production systems that are more stable and maintainable. Creating a **DevOps** culture requires attention to team organization, work practices, reporting lines, and incentives - leading to joint responsibility for faster and safer delivery. We recommend adopting DevOps because we cannot see any situation where attention in this area will not have a positive benefit.

In contrast to traditional up-front, heavy-weight enterprise architectural designs, we recommend adopting **evolutionary architecture**. It provides the benefits of enterprise architecture without the problems caused by trying to accurately predict the future. Instead of guessing how components will be re-used, evolutionary architecture supports adaptability, by proper abstractions, database migrations, test suites, continuous integration and refactoring, to harvest re-use as it occurs within a system. The driving technical requirements for a system should be identified early to ensure they are properly handled in subsequent designs and implementations. We advocate delaying decisions to the latest responsible moment, which might in fact be up-front for some decisions.

RESTful APIs have become standard in our industry. A good REST API provides a simple, lightweight means of building customizations and integrations. However, a lot of the quick, high value integrations we'd like to build require knowing when something happened. Consider building an **event API**, which, when used in conjunction with a REST API, facilitates simple workflow, notification, and synchronization integrations. These integrations often require no more than 20 or 30 lines of code. Often event APIs take the form of a "web hook" or callback mechanism, but don't be afraid of using a poll-based Atom style either. An Atom event API scales cheaply and gives the client the power to guarantee delivery.

One of the goals of SOA has been to decouple services by exchanging human-readable business documents instead of programming parameters. However, in implementing SOA, many businesses have simply used web services to expose the underlying programming models of back-end systems. **Procedure oriented integration** is nothing more than remote procedure calls implemented via a different protocol. The consequences of this are additional layers of complexity with no improvement in business flexibility. To avoid this, implementers of SOA should first understand the business meaning of their services, then implement human-readable contracts that are independent of legacy system implementation.

All too often caching is an afterthought used to address performance problems with a blanket approach and common cache lifetime. This leads to issues and workarounds. The "time value" of information is inherently linked to the business purpose and hence needs to be captured at the same time as other requirements. We believe **thoughtful caching** should be addressed early in the project and not just treated as a last minute performance fix.

Starting performance tests late in a project is risky and costly. Very simple performance tests that exercise key parts of the system, run on a regular basis, are good enough to track trends, so we can react if we see a change in performance. Run these tests with your build or as an overnight job and graph the results to create simple **performance trending**. Complex performance tests in a truly representative environment are still useful, but don't wait for them to start understanding how the performance of your code is changing.

Copyright © 2011 ThoughtWorks

Technology Radar - July 2011 - 4

Micro-services

Embedded Servlet Containers

Or how we designed and nearly built a Resource Oriented, Event Driven System out of applications about 1000 lines long…

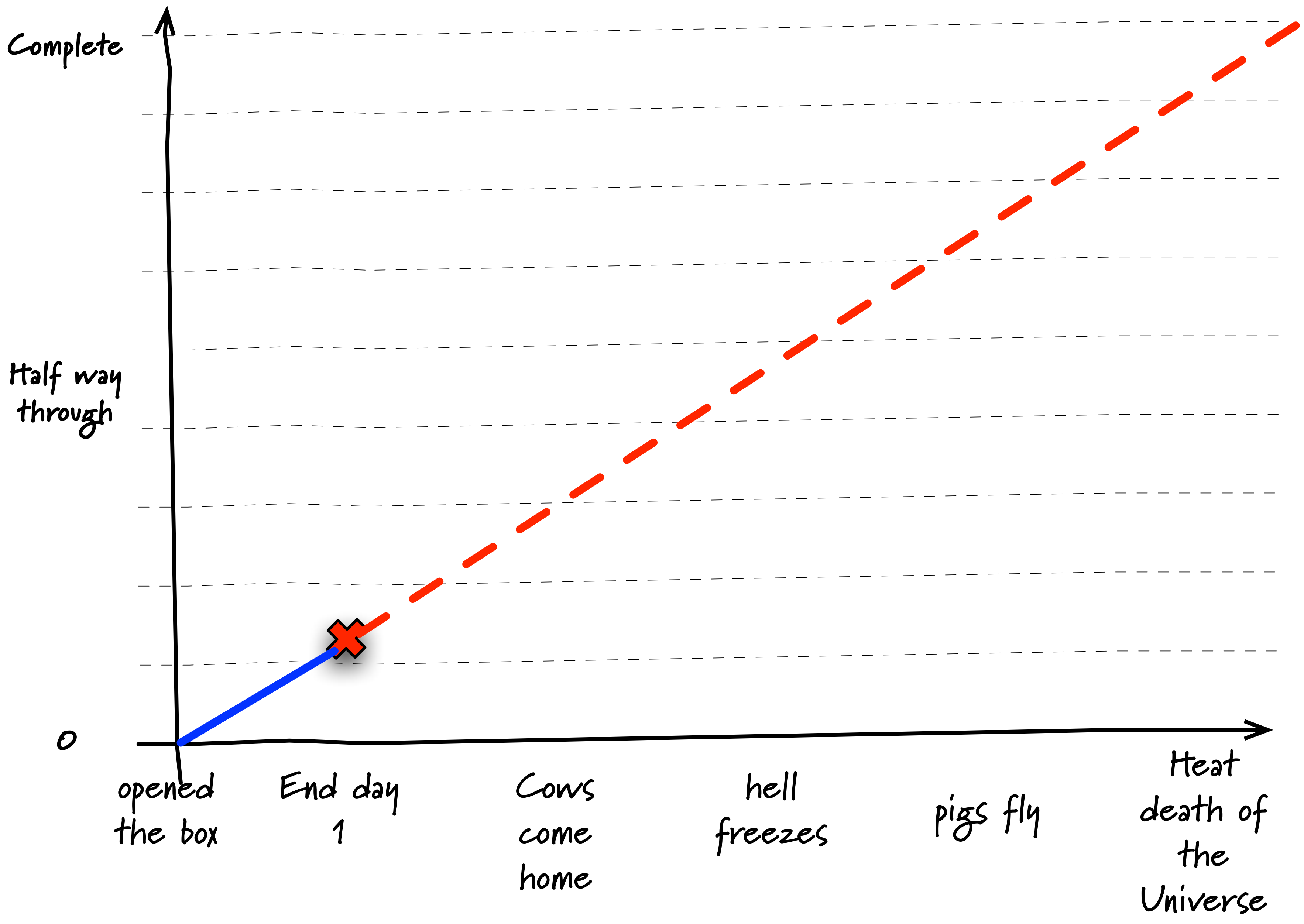
## WHAT I DID LAST SUMMER

## In the beginning…

- There was a new product being developed by an organisation in London
- The organisation had gathered their list of high level requirements
- And they asked ThoughtWorks if we could help them design and build it…

## So we took a look at their requirements

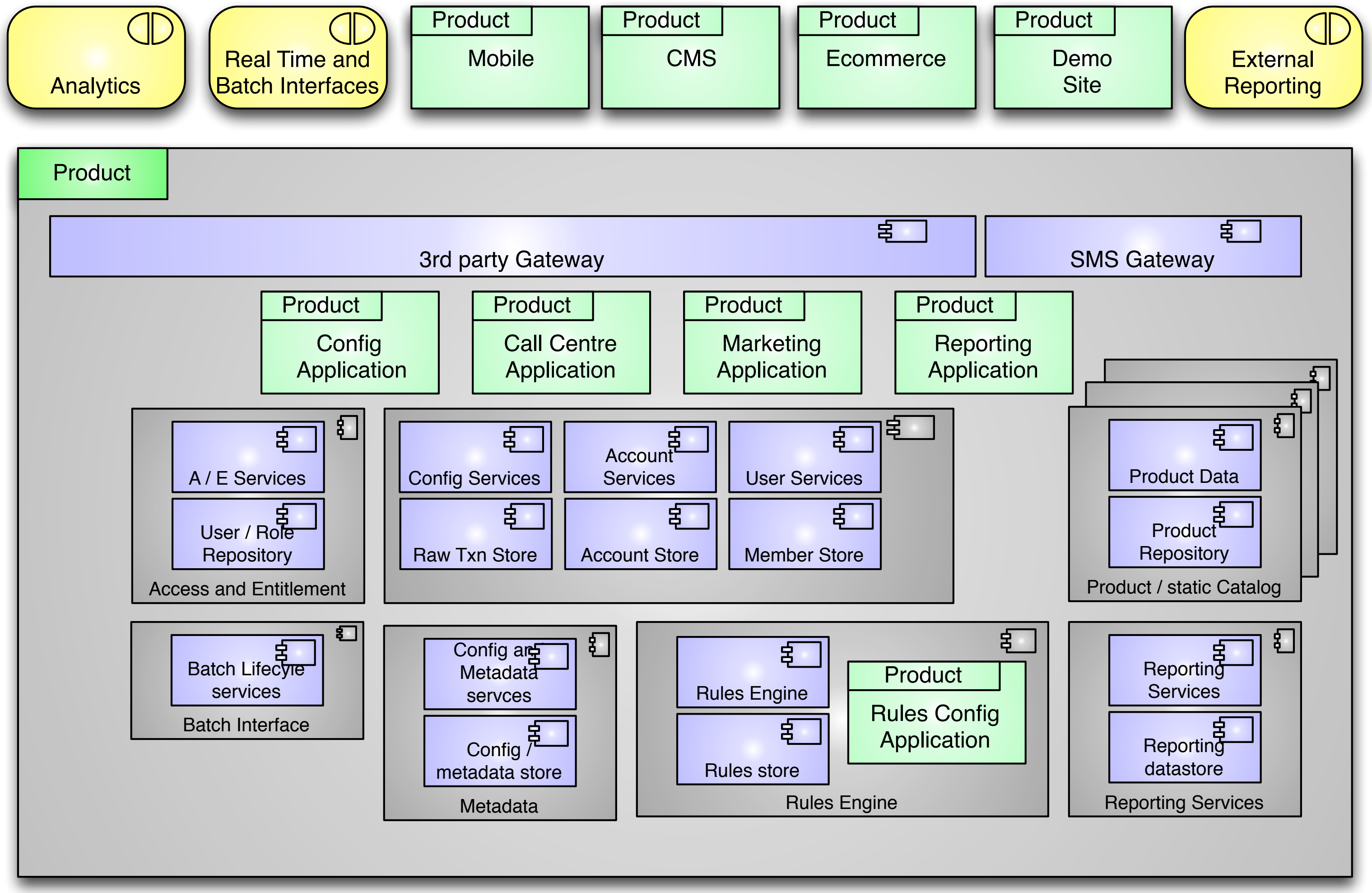
- Me and my mates at ThoughtWorks
- Worked out to be about 5000 points worth of User Stories
  - At 7 points per pair of developers per week


Complete
Half way through
0
opened the box
End day 1
Cows come home
hell freezes
pigs fly
Heat death of the Universe

## Tip 1

Divide and conquer

Start on the outside and model business capabilities

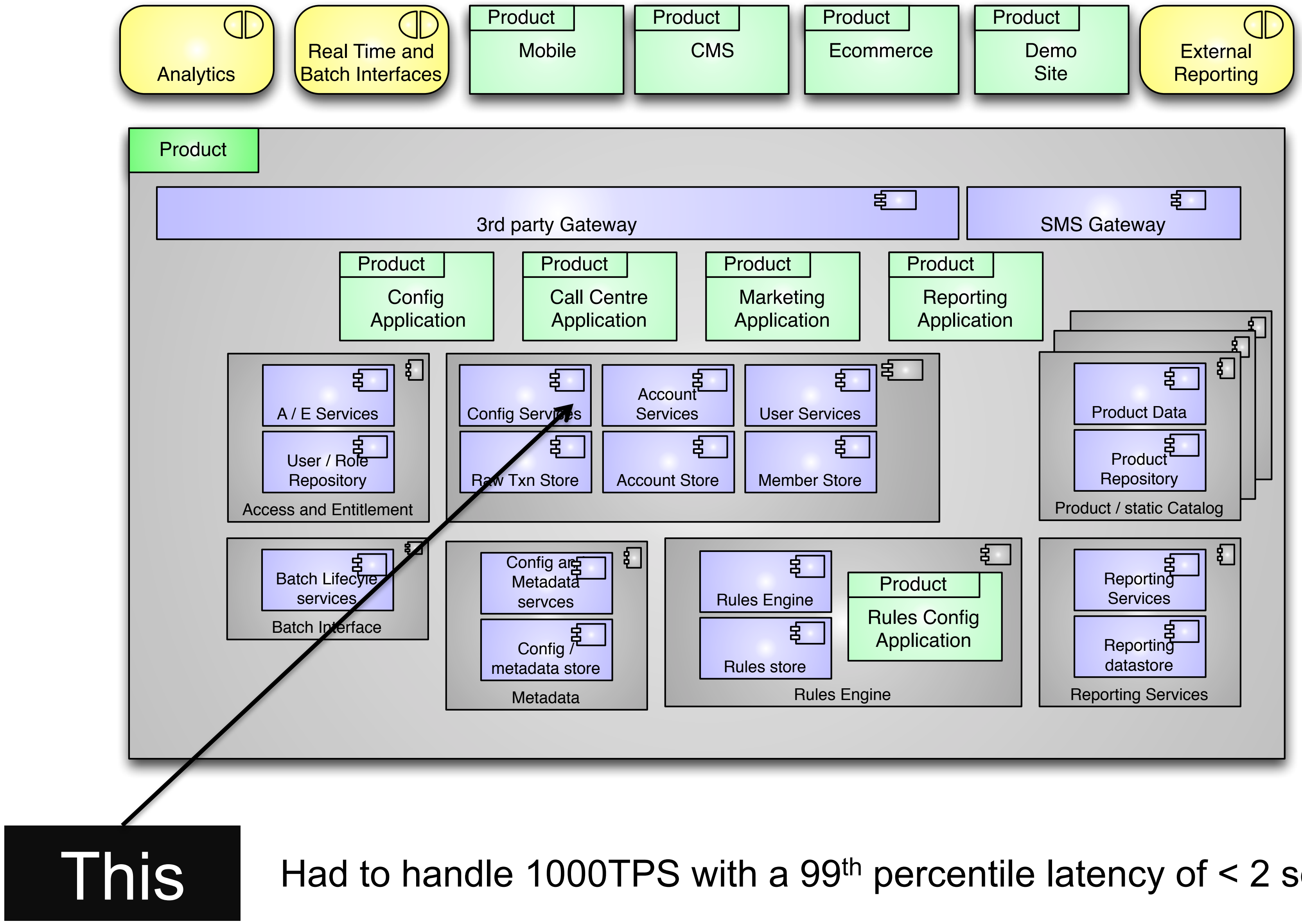


Each small box represents a capability,

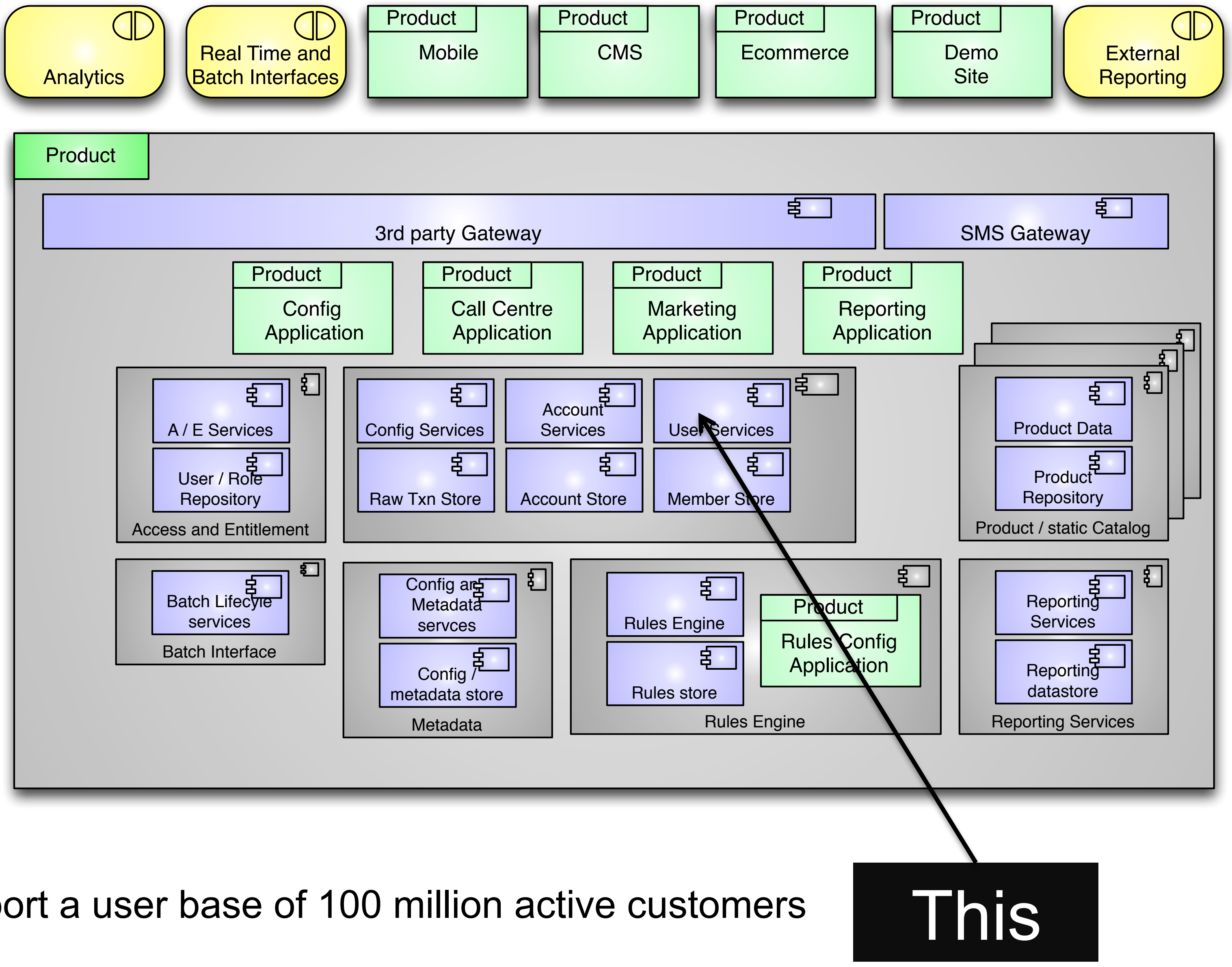
composed of one or more services

- The only way we could hit anything like the timescales required was to scale the programme quickly
- And that meant multiple teams in multiple workstreams

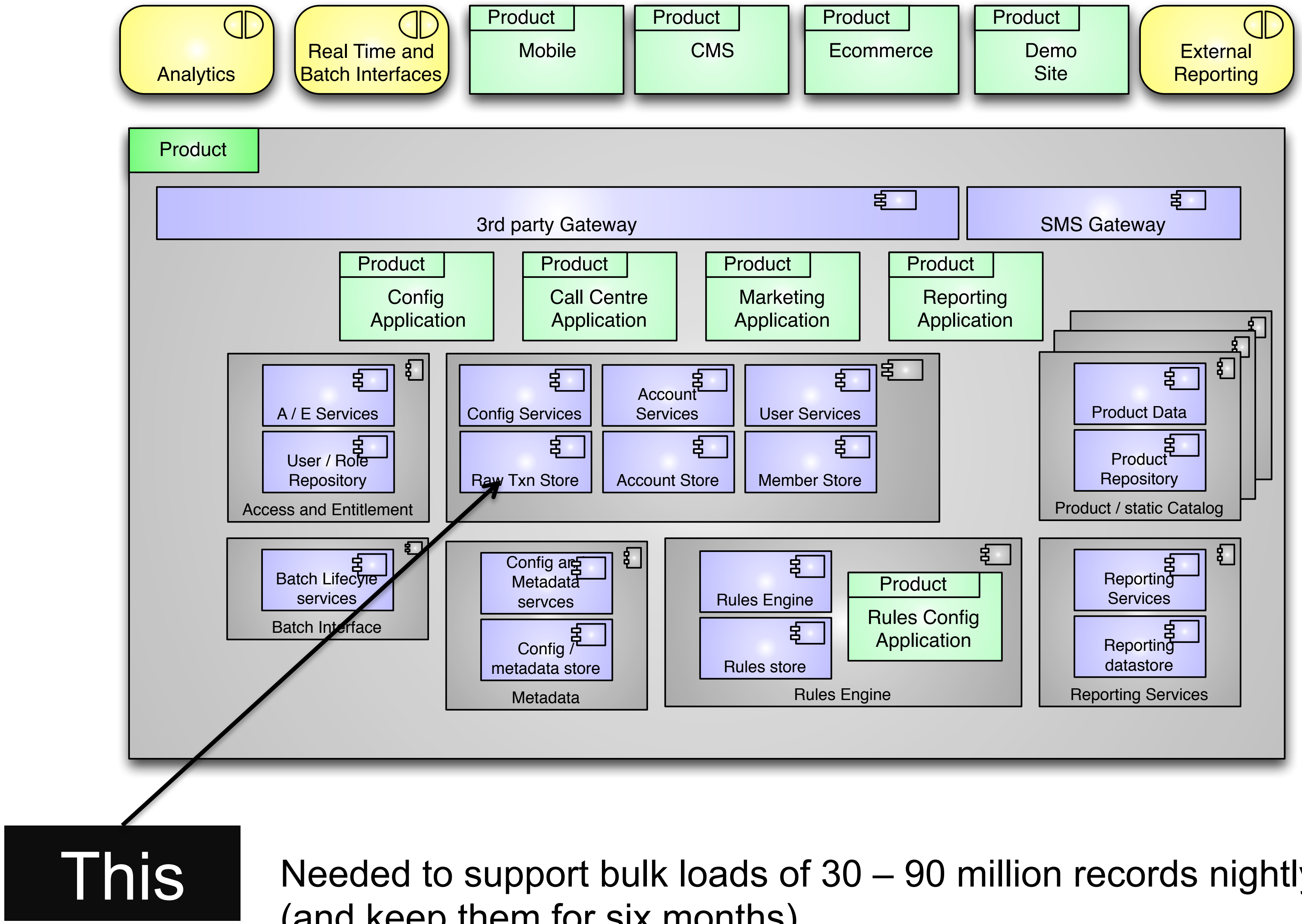
And there were some, umm, interesting non-functional requirements too



Had to handle 1000TPS with a 99th percentile latency of < 2 seconds



Support a user base of 100 million active customers



Needed to support bulk loads of 30 – 90 million records nightly (and keep them for six months)

## Did I mention PCI Level 1?

Finally, this is a product build.

So it needed to be modular / <cough> “infinitely configurable”

And deployable on Cloud and Tin

# The product need to to be…

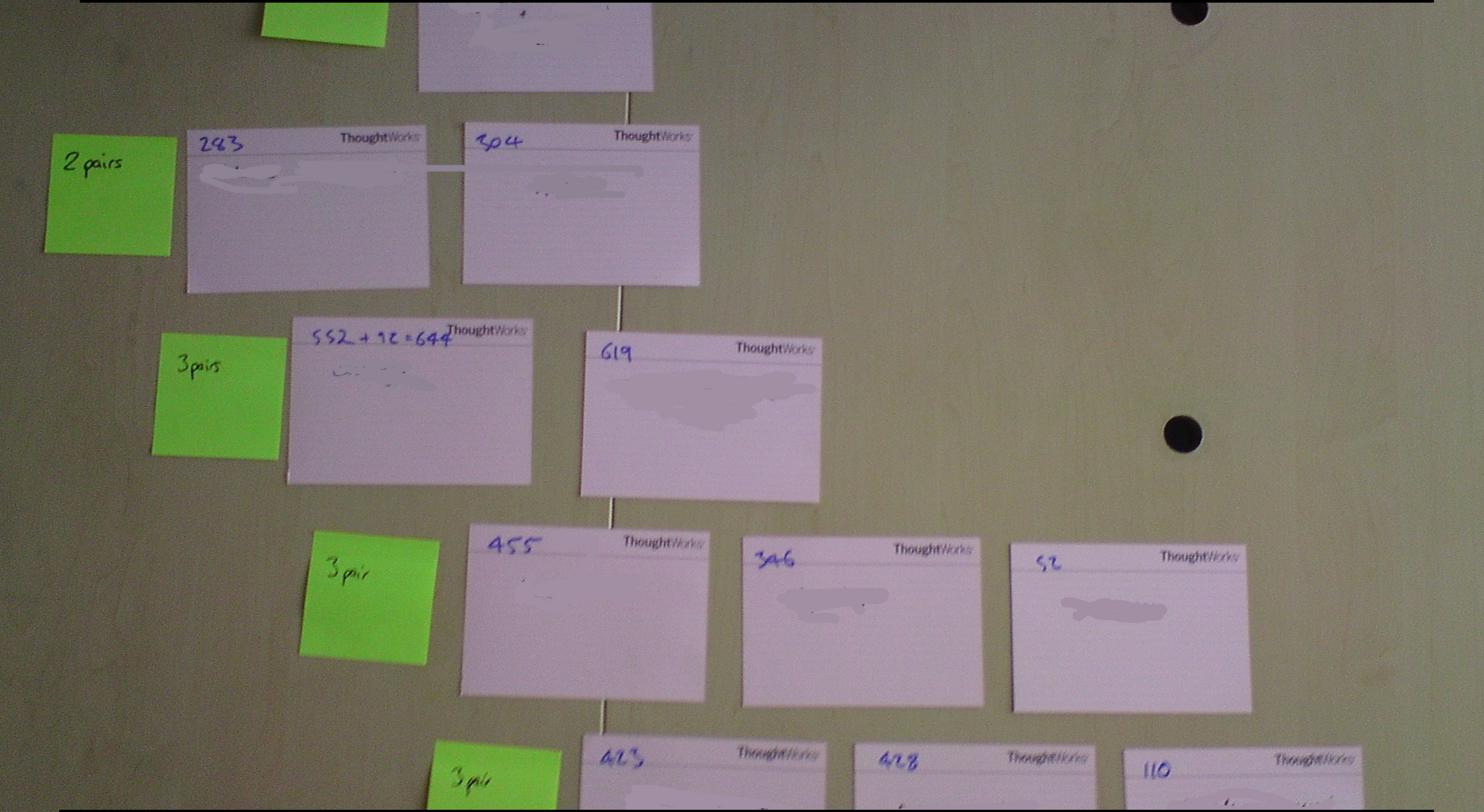
- Performance
  - fairly high throughput both transactional and batch
- Fault tolerant
  - One thing about the cloud, you are designing for failure right?
- Configurable
  - On a per install or PaaS basis
- Portable
  - Fortunately not to Windows…
- Maintainable
  - over multiple versions and years
- Supporting big data sets
  - Billions of transactions available
  - Millions of customers available

and capable of being built quickly without sacrificing the other principles

## Plus ça change, plus c'est la même chose.

(The more things change the more they stay the same)

So, after five weeks we had broken the problem down into capabilties



Now we had to start scaling the teams to deliver these capabilities

## Tip 2

## Use Conway's Law to structure teams

"…organizations which design systems … are constrained to produce designs which are copies of the communication structure of those organizations"

Melvin Conway, 1968

## The first business capability - Users

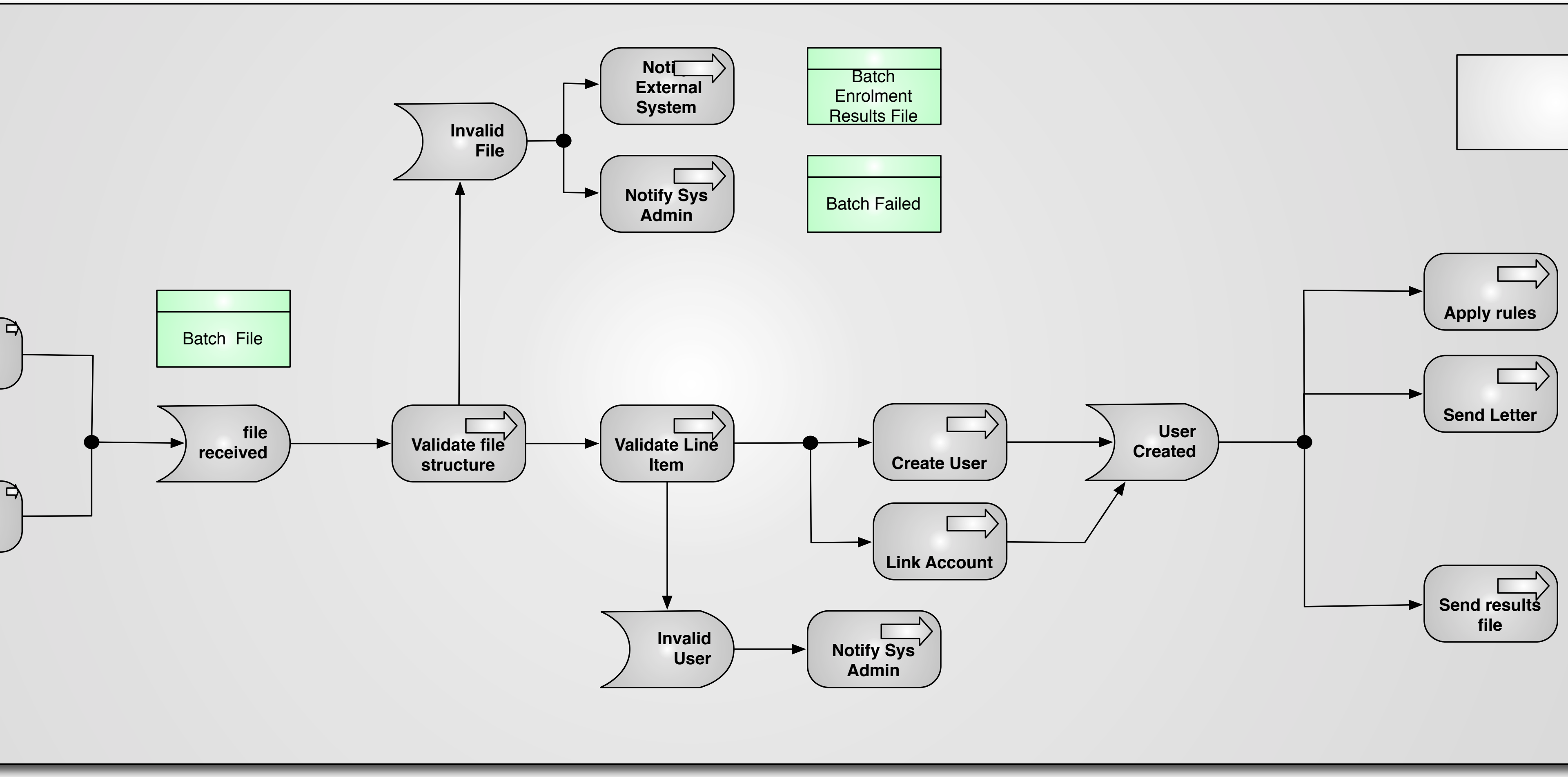
- Responsible for creation and maintenance of users in the system
  - Up to 100 million of them per instance of the product
- Used by many clients with many usage patterns
  - Call centre and website – CRUD
  - Inbound batch files – CRUD x hundreds of thousands per night
- Many downstream consumers of the data
  - Fulfilment systems for example

## Tip 3

The Last Responsible Moment

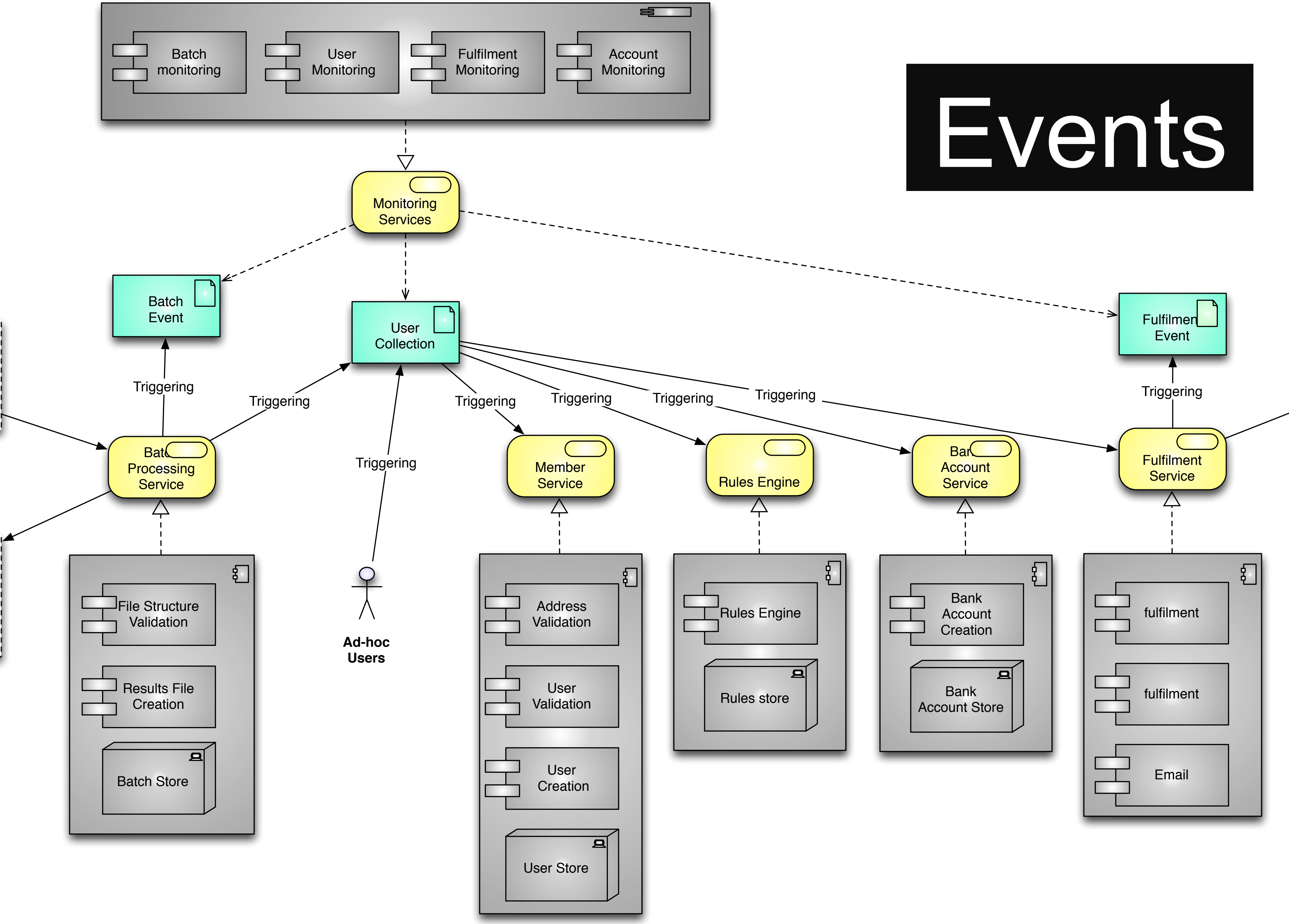
Don’t decide everything at the point you know least

# We started with a business process…



and noticed something funny…

# Events


Batch monitoring
User Monitoring
Fulfilment Monitoring
Account Monitoring
Monitoring Services
Batch Event
User Collection
Fulfilmen Event
Triggering
Triggering
Triggering
Triggering
Triggering
Triggering
Triggering
Triggering
Batch Processing Service
Member Service
Rules Engine
Bank Account Service
Fulfilment Service
File Structure Validation
Results File Creation
Batch Store
Ad-hoc Users
Address Validation
User Validation
User Creation
User Store
Rules Engine
Rules store
Bank Account Creation
Bank Account Store
fulfilment
fulfilment
Email

## I know what you are thinking…

# ESB*

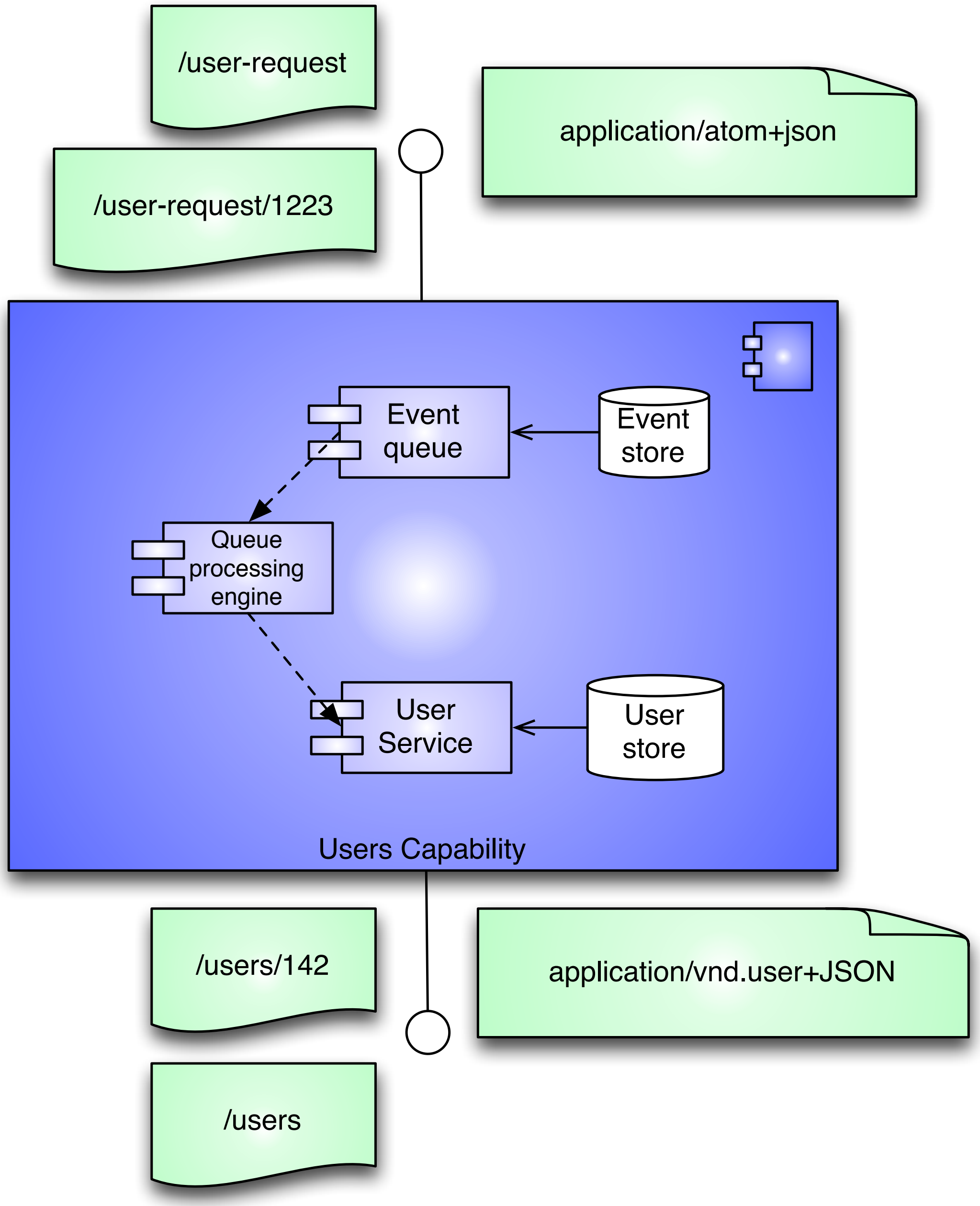
* Dan North coined the term Enterprise Night Bus…

## Or you could use the web

## Tip 4

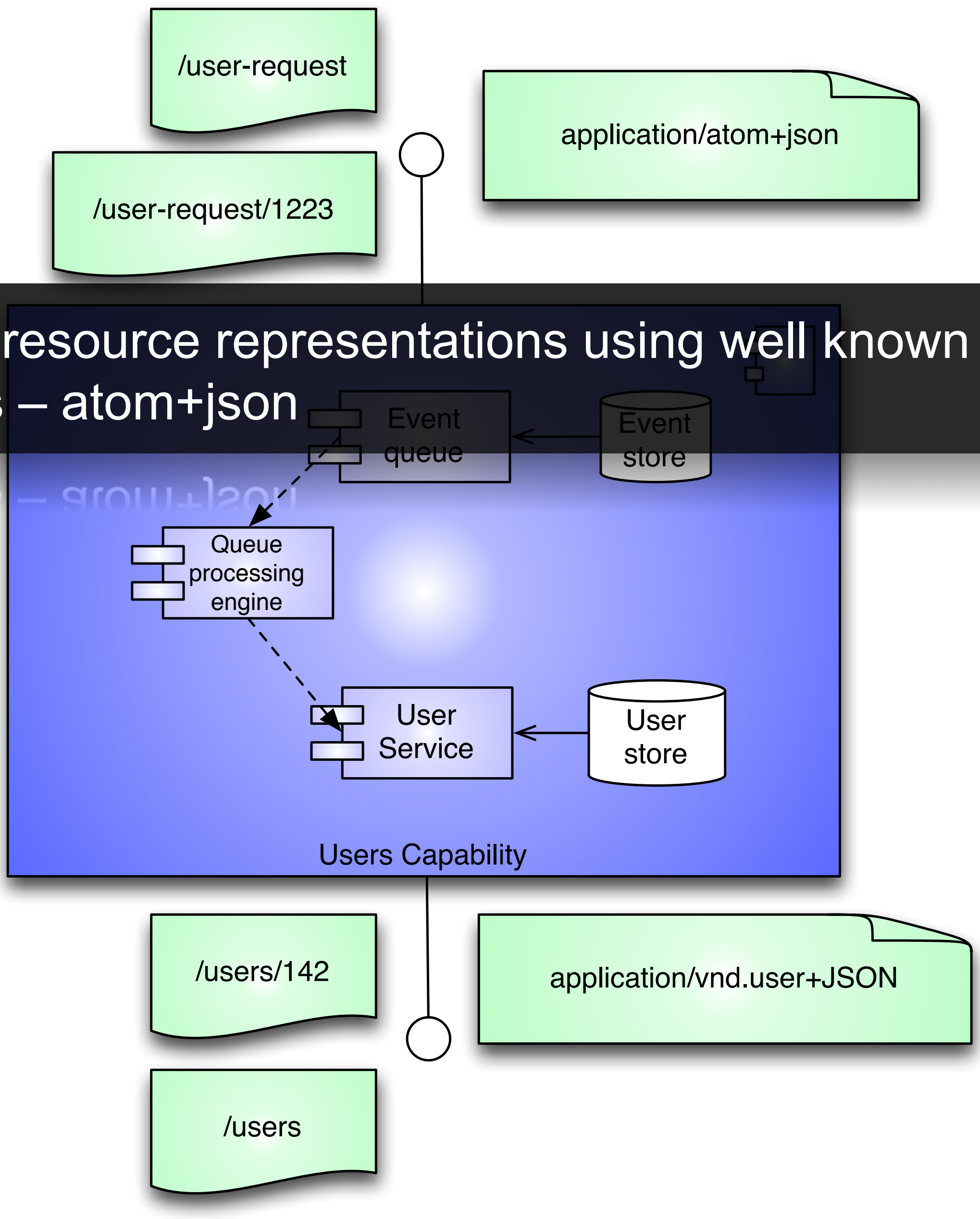
Be of the web, not behind the web

RFC 5023 to be precise

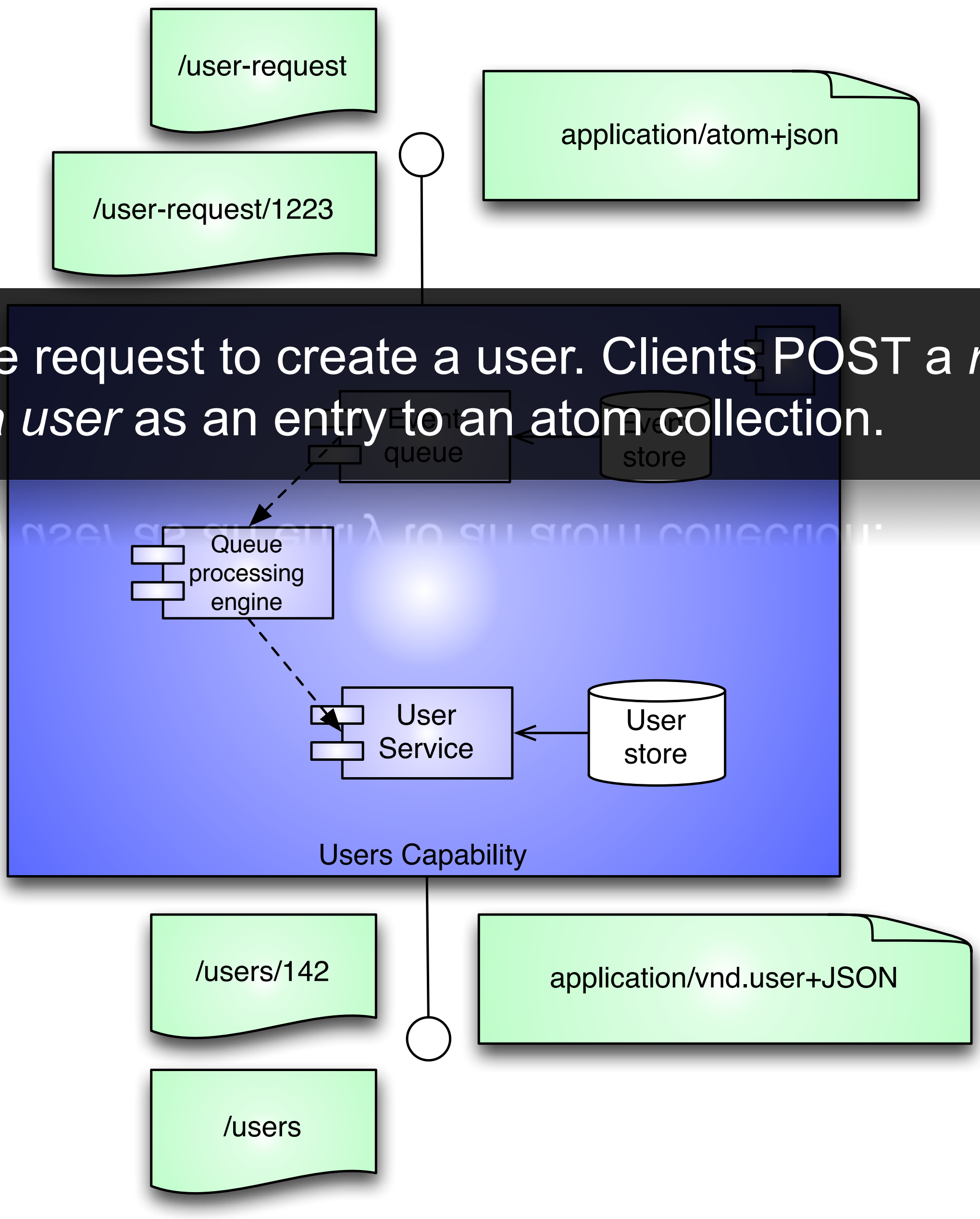


and this is what we built

Standard resource representations using well known web standards – atom+json



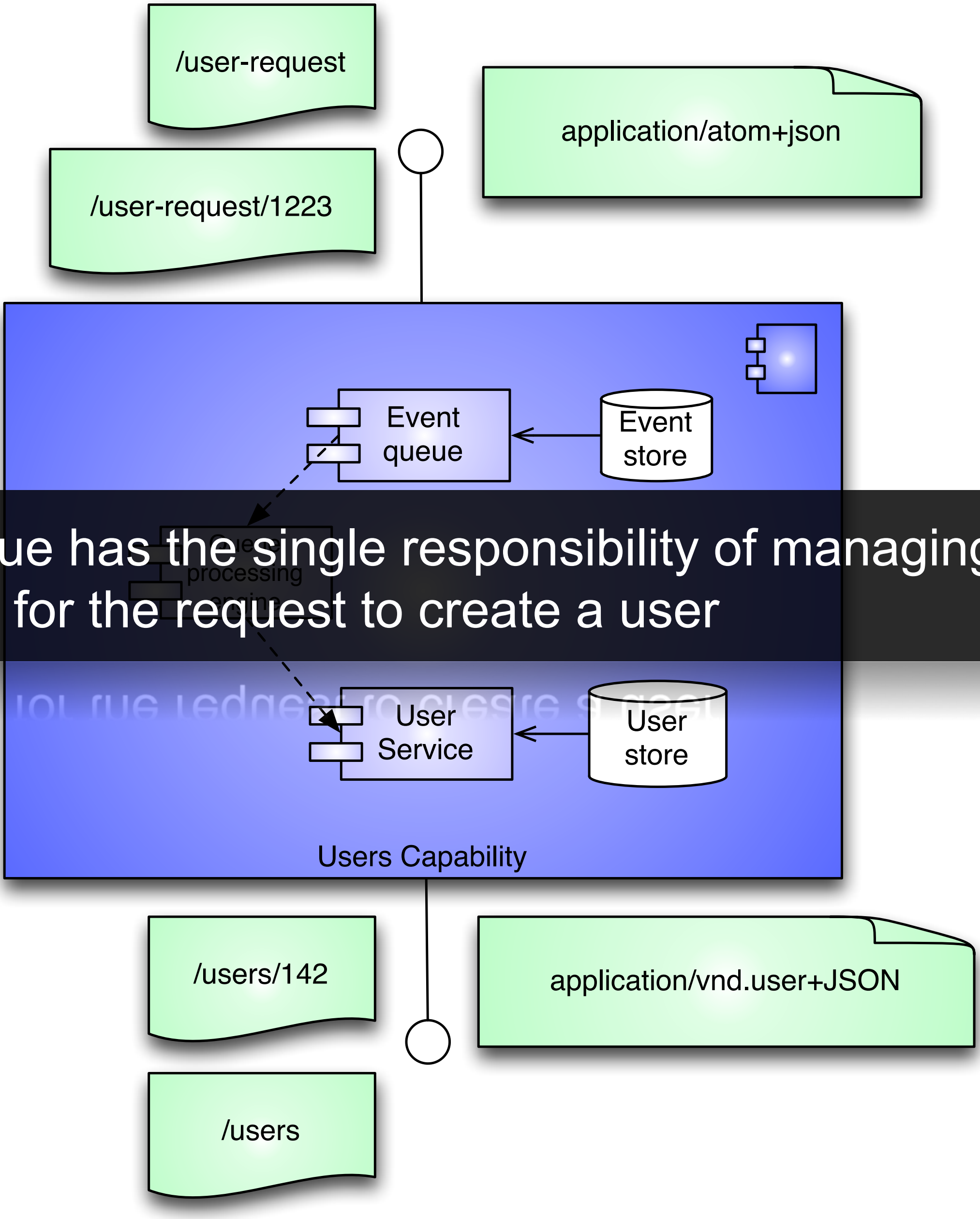
***Reified*** the request to create a user. Clients POST a *request to create a user* as an entry to an atom collection.



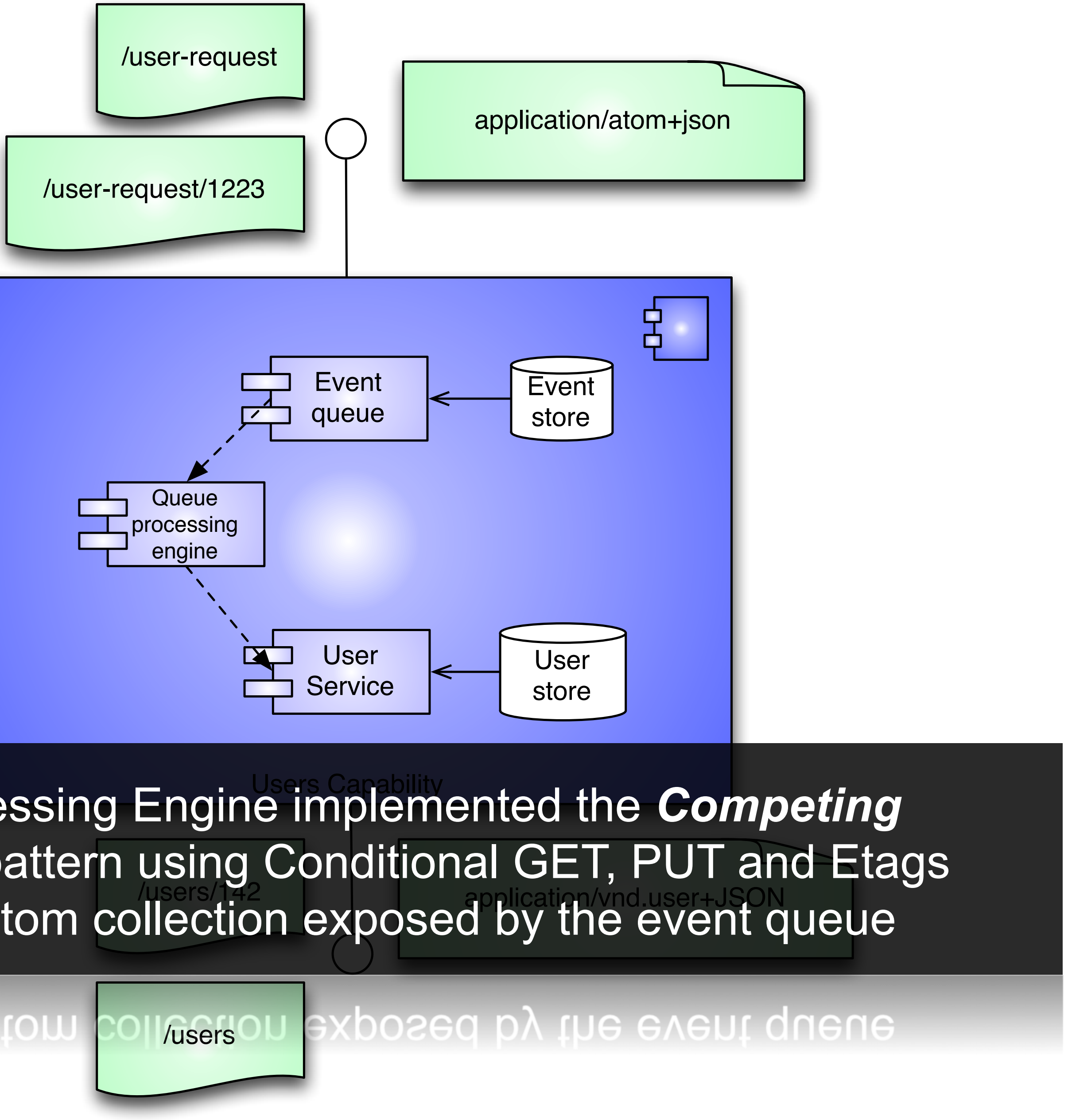
## Tip 5

If something is important, make it an explicit part of your design

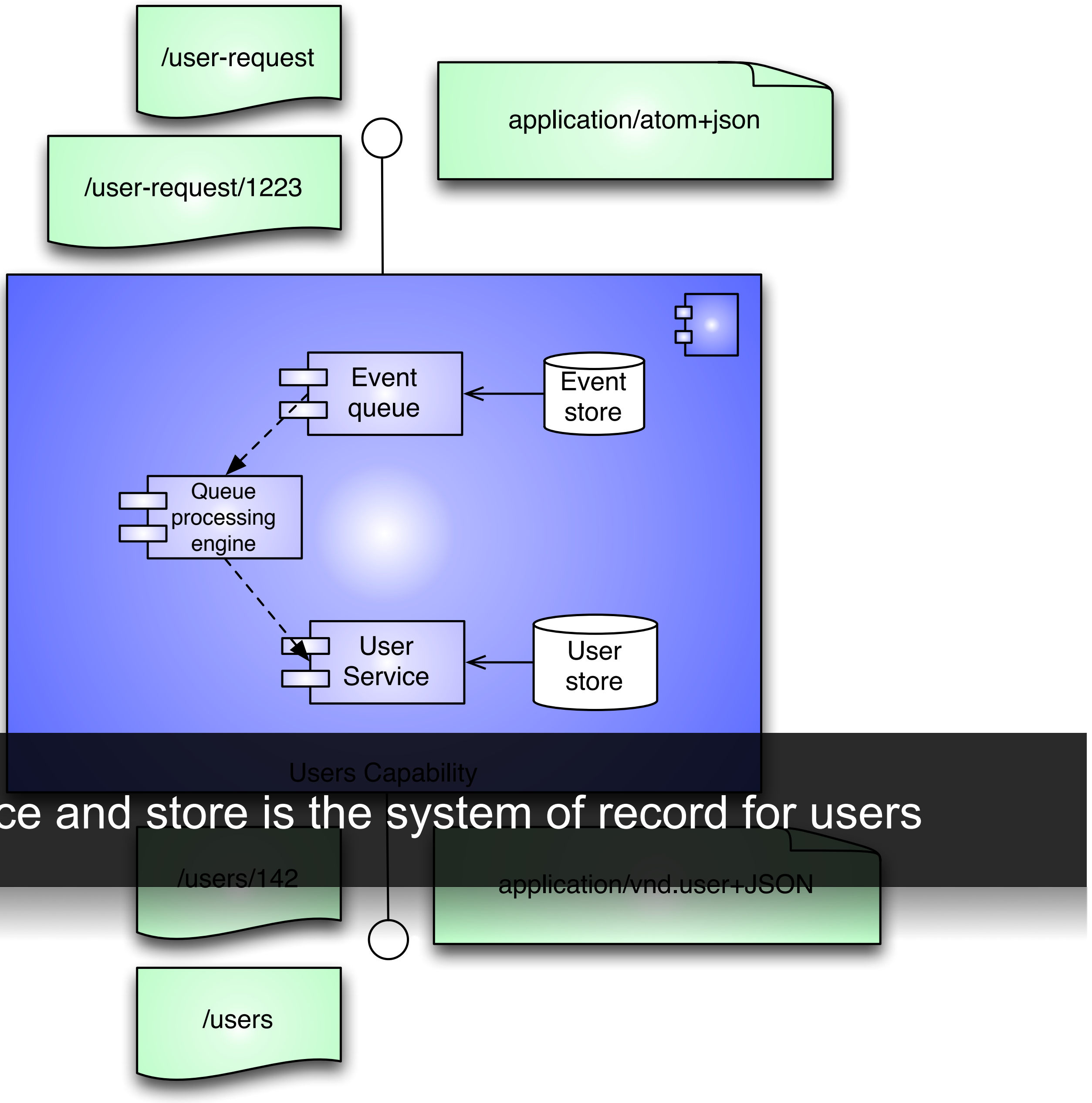
Reify
to convert into or regard as a concrete thing: to reify a concept.



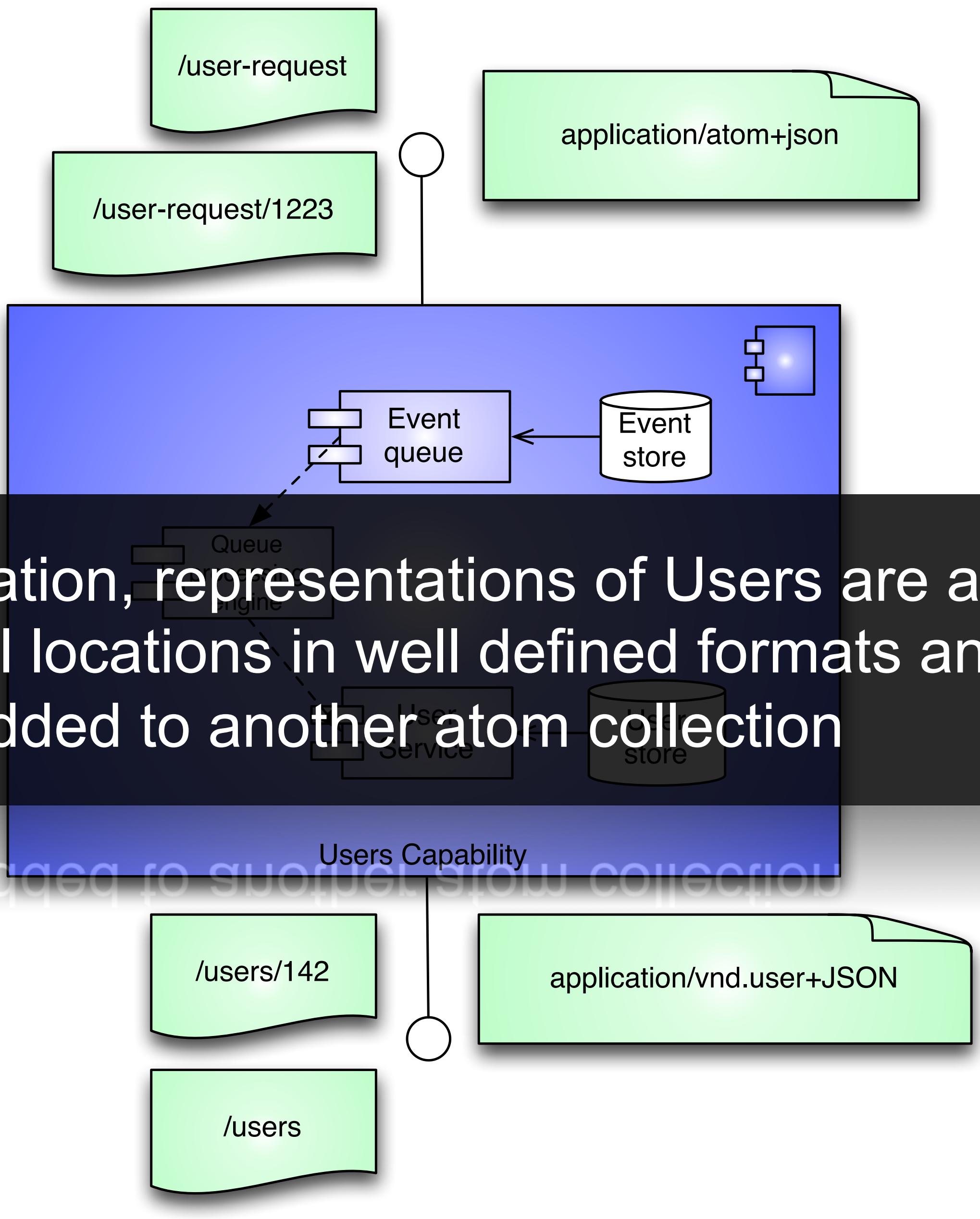
Event queue has the single responsibility of managing *state transitions* for the request to create a user



Queue Processing Engine implemented the ***Competing Consumer*** pattern using Conditional GET, PUT and Etags against the atom collection exposed by the event queue



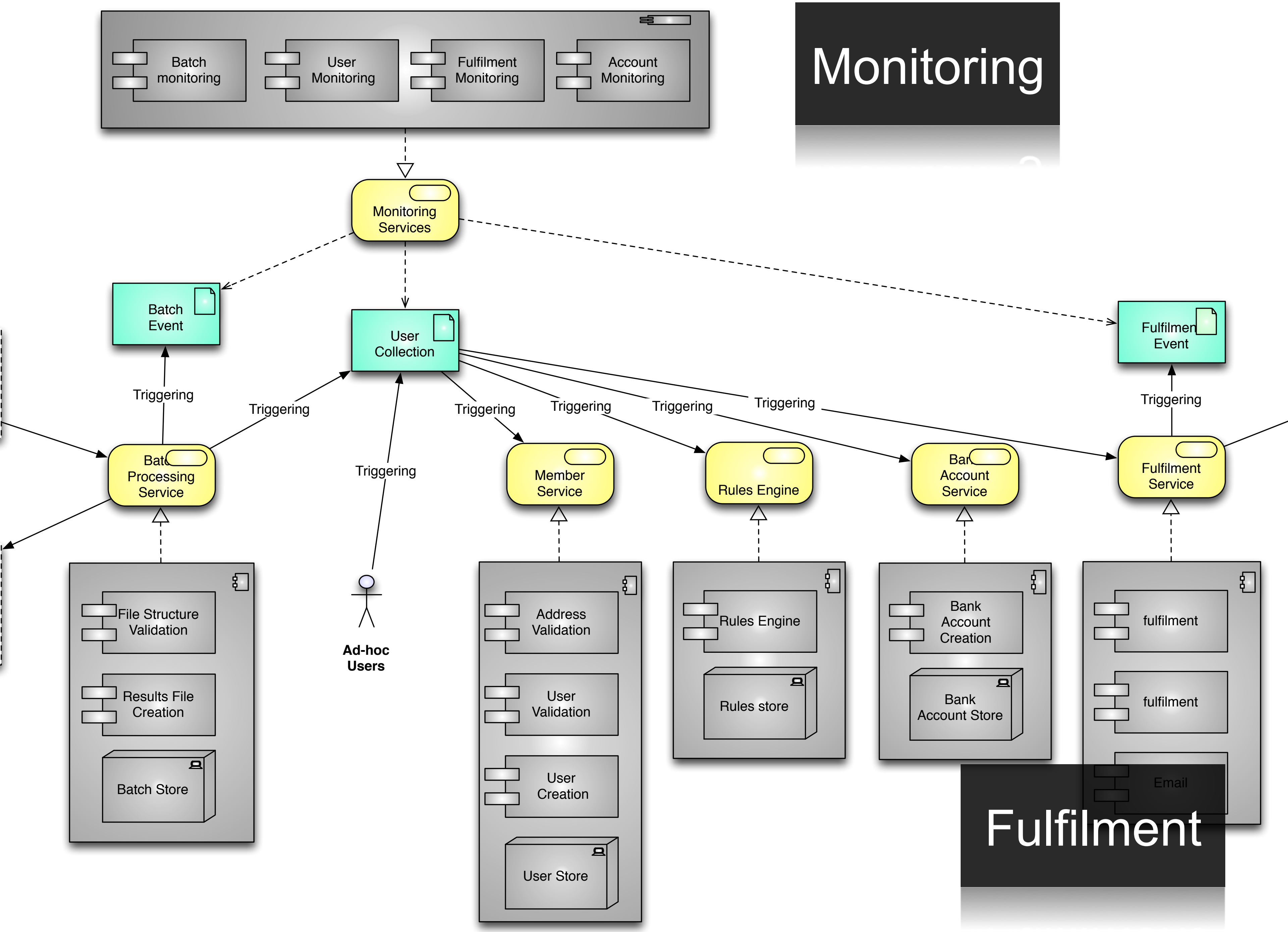
User Service and store is the system of record for users



After creation, representations of Users are available at canonical locations in well defined formats and creation events added to another atom collection

Where they are available for consumption by other downstream systems

# Monitoring



Fulfilment

# Our micro-services

- User Request Queue
  - Forms the transactional boundary of the system
- Request Queue Processor
  - Competing Consumer processes events on the queue and POSTs them to
- User Service
  - System of record for Users in the system
  - Responsible for all state changes of those users
  - Exposes events on those users to other systems

## CHARACTERISTICS OF MICRO-SERVICES

## Small with a single responsibility

- Each application only does one thing
- Small enough to fit in your head
  - James' heuristic
  - "If a class is bigger than my head then it is too big"
- Small enough that you can throw them away
  - Rewrite over Maintain

## Containerless and installed as well behaved Unix services

- Embedded web container
  - Jetty / SimpleMind
  - This has a lot of benefits for testing (inproctester for example) and eases deployment
- Packaged as a single executable jar
  - Along with their configuration
  - And unix standard rc.d scripts
- Installed in the same way you would install httpd or any other application
  - Why recreate the wheel? Daemons seem to work ok for everything else. Unless you are *special*?

# Located in different VCS roots

- Each application is completely separate
- Software developers see similarities and abstractions
  - And before you know it you have One Domain To Rule Them All
- Domain Driven Design / Conways Law
  - Domains in different bounded contexts should be distinct – and its ok to have duplication
  - Use physical separation to enforce this
- There will be common code, but it should be ***library and infrastructure*** code
  - Treat it as you would any other open source library
  - Stick it in a nexus repo somewhere and treat it as a binary dependency

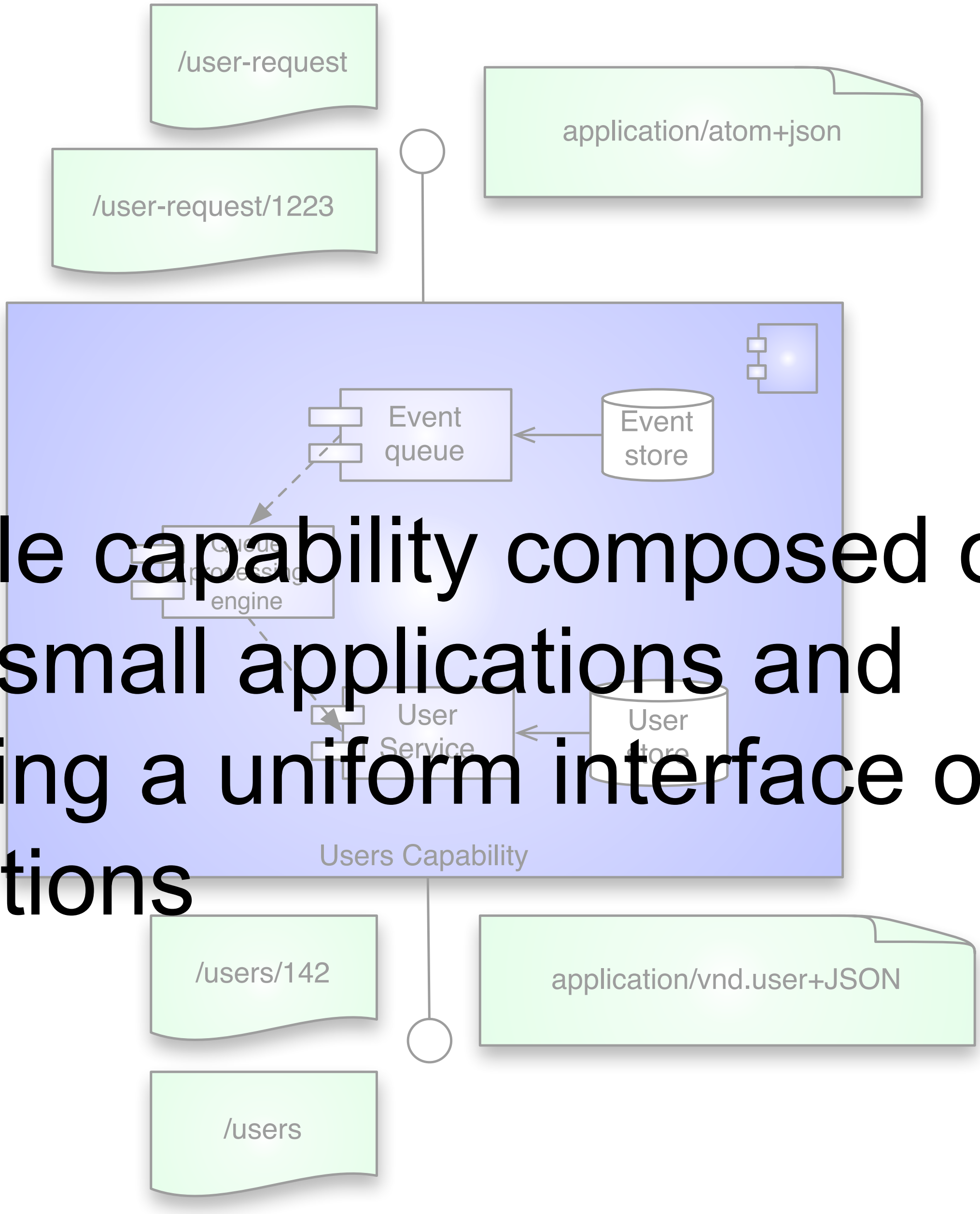
## Provisioned automatically

- The way to manage the complexity of many small applications is declarative provisioning
  - UAT:
    - 2 * service A, Load Balanced, Auto-Scaled
    - 2 * service B, Load Balanced, Auto-Scaled
    - 1 * database cluster

## Status aware and auto-scaling

- What good is competing consumer if you only have one consumer?
  - We don’t want to wake Peter up at three in the morning any more to start a new process

- Use watchdog processes to monitor in-app status pages
  - Each app exposes metrics about itself
  - In our case, queue-depth for example
  - This allows others services to auto-scale to meet throughput requirements



A single capability composed of many small applications and exposing a uniform interface of Atom Collections

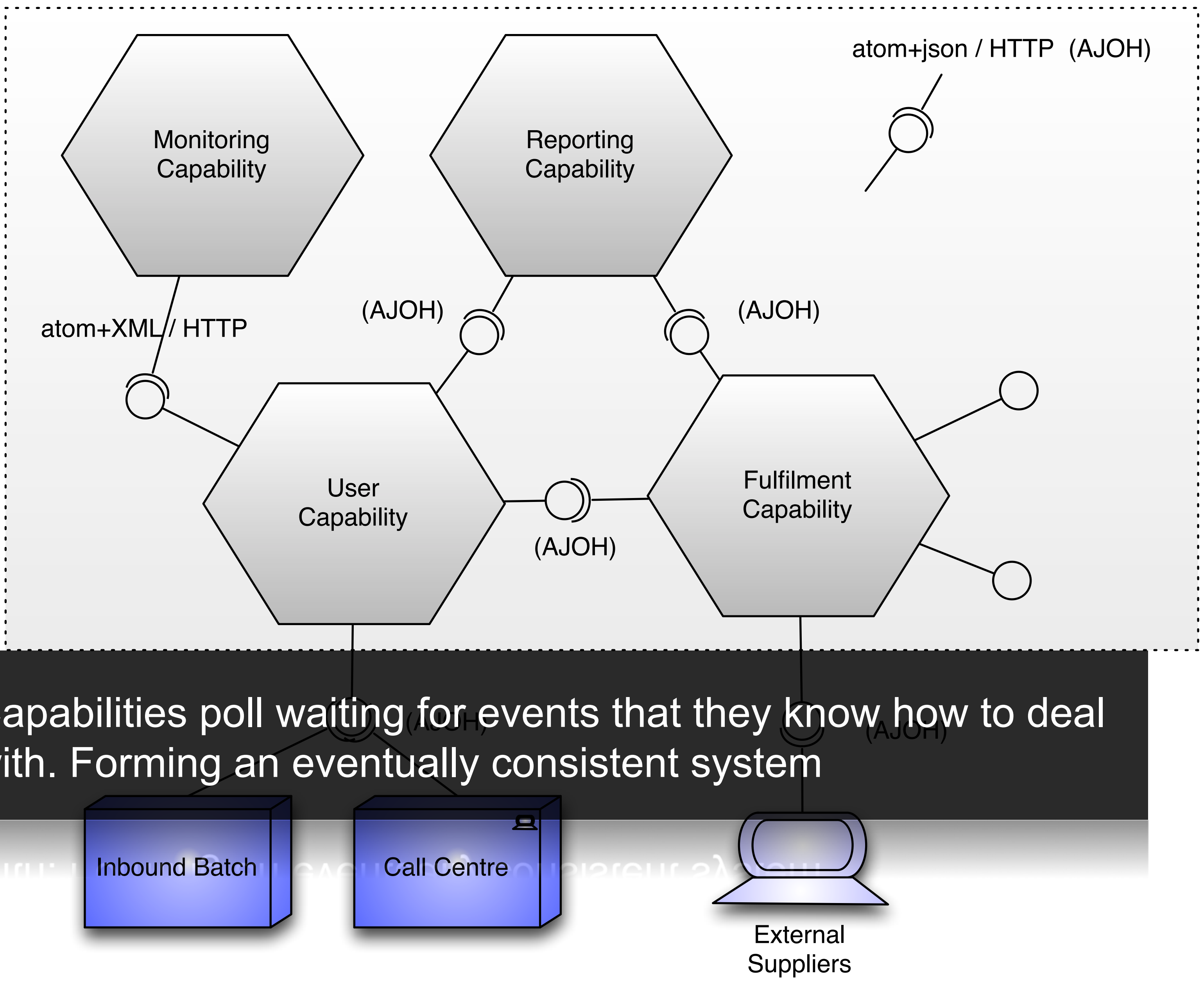
## How the capabilities form a product

# They interact via the uniform interface

- HTTP
  - Don’t fight the battles already won
  - Use no-brainer force multipliers like reverse proxies

- HATEOS
  - Link relations drive state changes
  - Its an anti-corruption layer that allows the capability to evolve independently of its clients

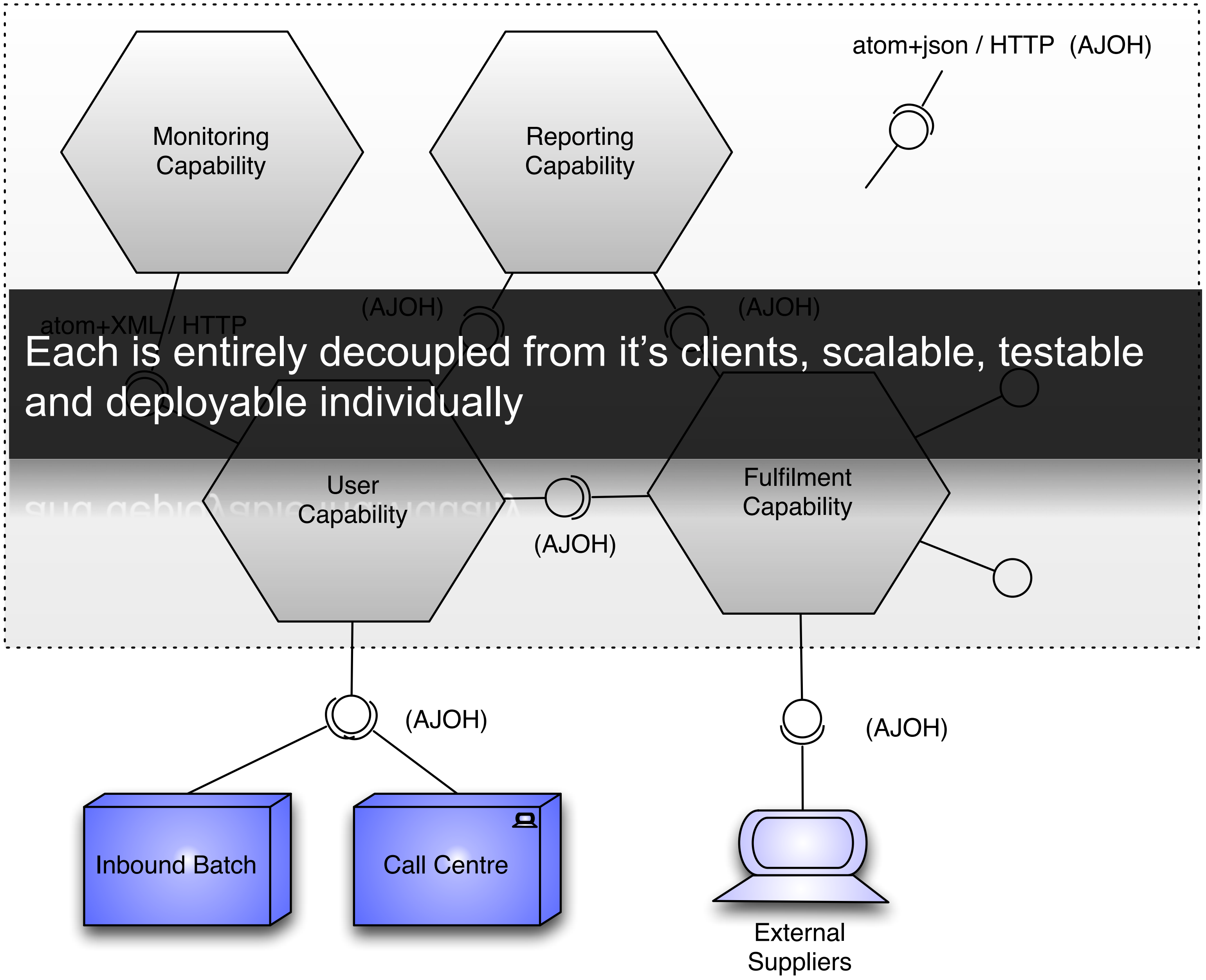
- Standard media types
  - Can be used by many different clients
  - You can monitor it using a feed reader if you want…



Capabilities poll waiting for events that they know how to deal with. Forming an eventually consistent system

## Tip 6

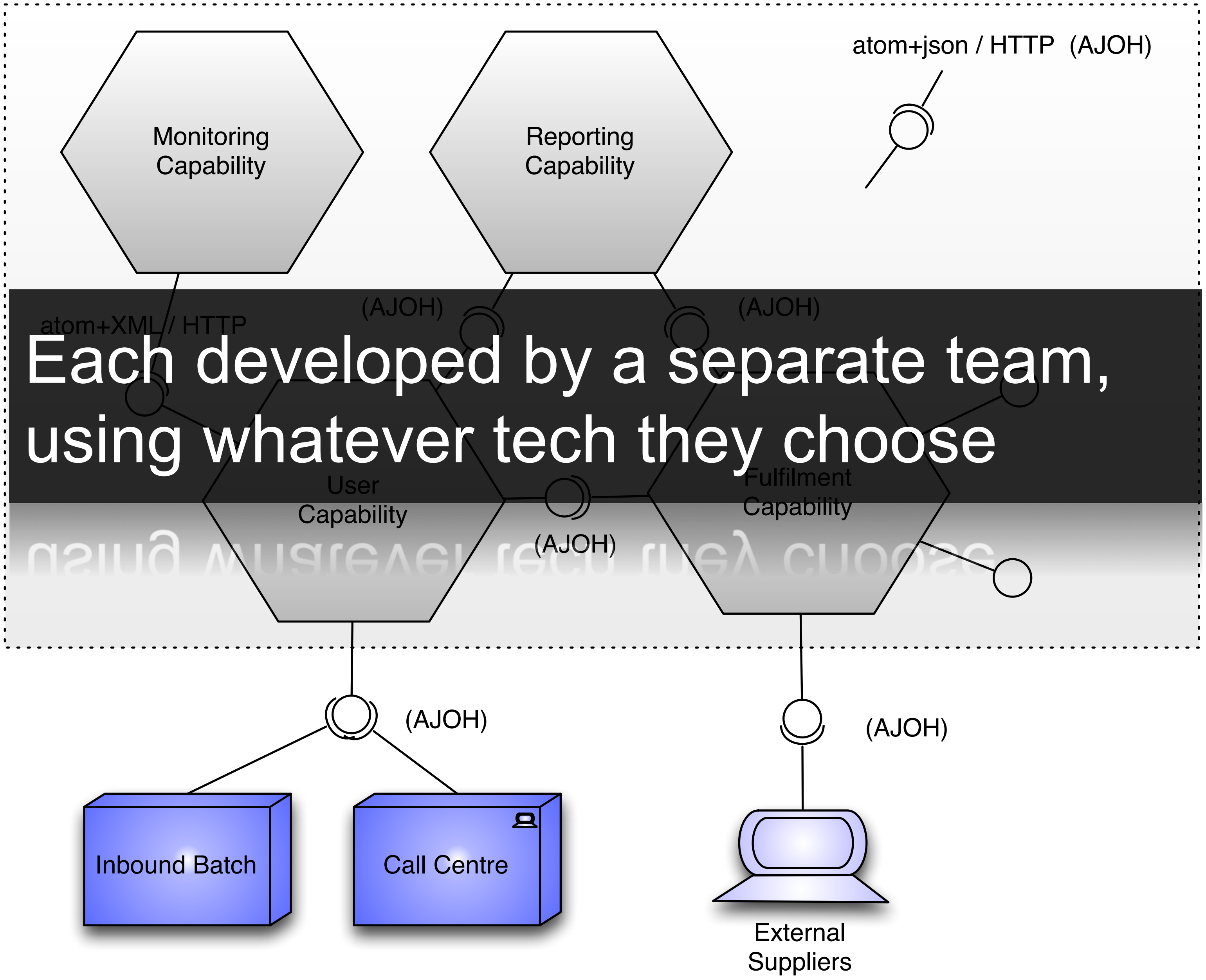
Favour service choreography over orchestration



Each is entirely decoupled from it’s clients, scalable, testable and deployable individually

## Tip 7

Use hypermedia controls to decouple services



Each developed by a separate team, using whatever tech they choose

# Our stack

- Embedded Jetty (current project uses SimpleWeb)
- PicoContainer for DI
- Hibernate (but wrote our own SQL)
- Abdera for Atom
- Smoothie charts
- Metrics @codehass
- Graphite

## Infrastructure automation stack

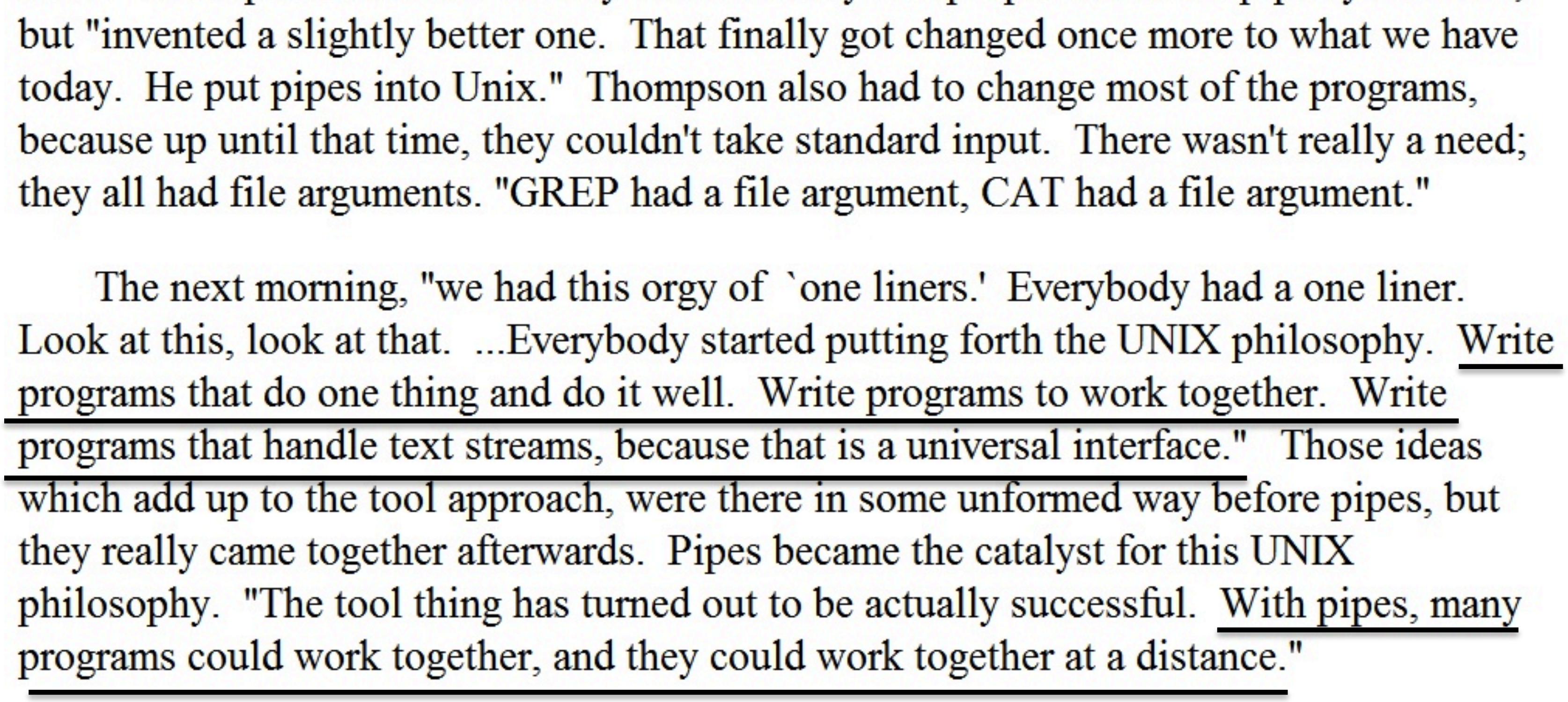
- Fabric with boto
- AWS, but deployable to anything with SSH
- Maven (boo)
- Git
- Puppet for provisioning

## NO SILVER BULLETS

## This stuff is hard

- We haven’t even talked about
  - Versioning
  - Integration
  - Testing
  - Deployment
- Eventual Consistency can be tricky for people to get there head around
- Developers like using ***enterprisy software***
  - No one got fired for choosing an ESB
  - Convincing people to use the web is hard

## SUMMARY



but "invented a slightly better one. That finally got changed once more to what we have today. He put pipes into Unix." Thompson also had to change most of the programs, because up until that time, they couldn't take standard input. There wasn't really a need; they all had file arguments. "GREP had a file argument, CAT had a file argument."

The next morning, "we had this orgy of `one liners.' Everybody had a one liner. Look at this, look at that. ...Everybody started putting forth the UNIX philosophy. Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface." Those ideas which add up to the tool approach, were there in some unformed way before pipes, but they really came together afterwards. Pipes became the catalyst for this UNIX philosophy. "The tool thing has turned out to be actually successful. With pipes, many programs could work together, and they could work together at a distance."

## The Unix Philosophy

:s/pipes/http/

Lions commentary on Unix 2nd edition

## Consistent and reinforcing practices

Hexagonal Business capabilities composed of:

Micro Services that you can

Rewrite rather than maintain and which form

A Distributed Bounded Context.

Deployed as containerless OS services

With standardised application protocols and message semantics

Which are auto-scaling and designed for failure

ThoughtWorks®

Is hiring!

# Thanks!

jalewis@thoughtworks.com
@boicy
http://bovon.org