

SIMPLICITY: THE WAY OF THE UNUSUAL ARCHITECT

Dan North
Dan North & Associates

In the beginning...

...the software was without form, and void

The Architects said “Let there be light,” and
they separated the light from the darkness

And they called the light Architecture and the
darkness Hacking

And that was the first project

On the second project...

The Architects used all the technologies of the heavens and the earth they hadn't got round to the first time

- The simple `new()` was replaced by a Factory
- which was replaced by Dependency Injection
 - which was replaced by an IoC Container
 - which was augmented by XML configuration
 - which was supplemented by `@nnotations`

But they were not done yet...

The simple save() was replaced by a DAO

- which was replaced by a Unit of Work pattern
- which was replaced by a custom ORM
- which was replaced by Hibernate
 - which is called NHibernate by the Redmondites
- which was (partly) replaced by iBatis
- which was replaced by EJB 3
- which was (not) replaced by Active Record

And still they toiled...

The simple compile was replaced by a Makefile

- which was replaced by an Ant build.xml
 - which is called NAnt by the Redmondites
- which was replaced by many build.xml files
- which were generated by an XSLT transform
- which was replaced by Maven

And Maven brought forth a Plague of Apache Commons,
and there was a flood of all the Libraries of the Internet
as a judgement upon the people

And that was the Second System

Architects were fruitful and multiplied

They decided to build an Architecture that would reach to the heavens, to show how clever and wise they were, and remote invocation would be its name

But it came to pass that they were scattered to the four winds and began to speak in different tongues

Some spoke in CORBA, which was called DCOM by the Redmondites. The Sunnites spoke the language of JNDI, of the EJBites, which was XMLish and verbose

And there was a plague of standards to test the people

These are the generations of RPC

RPC begat RMI

- which begat COM and Object Brokers

COM begat DCOM, which begat WCF

Object Brokers begat Web Services

Web Services married XML

- and they begat SOAP and WSDL

SOAP begat the twelve (hundred) tribes of WS-*

WSDL begat Code Generated Stubs

And the people wrung their hands and wept

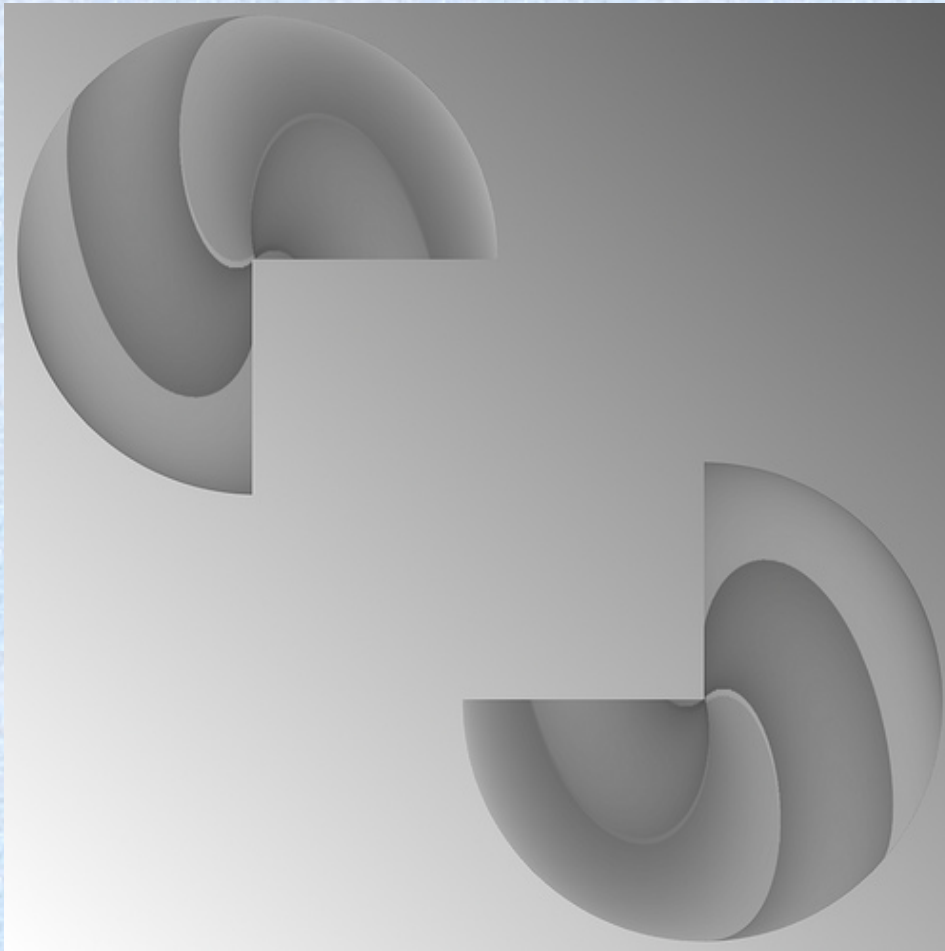
On the seventh day they
RESTed

The same story happens over and over

1. We observe a pattern
2. We create abstractions and generalisations
3. We turn the abstractions into a framework
4. The framework becomes a Golden Hammer
5. *People start to subvert the framework*
6. Finally, sometimes, simplicity grows out of adversity

Why do we keep doing this?

This is a pair of three-quarter circles



<http://www.flickr.com/photos/davidjoyner/2491859887/>

@tastapod

We are programmed to see structure

...even where none exist

We distort, delete and generalise

We complify where we should simplicate

“If I were going to Dublin...”

Try to see what is really there

Ask: What is actually slowing me down?

Get a pair, or a bath duck

“I would not give a fig for the simplicity this side of complexity, but I would give my life for the simplicity on the other side of complexity.”

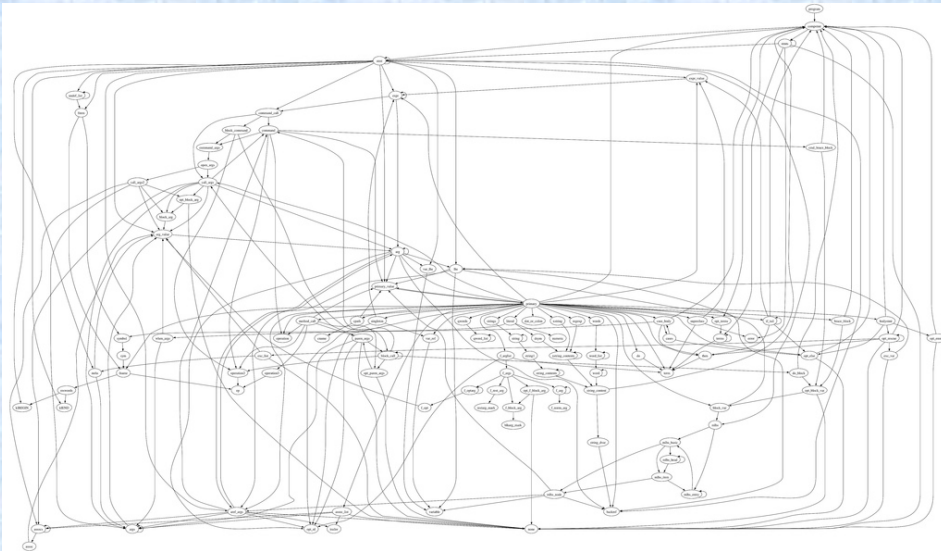
— Oliver Wendell Holmes

Thank you

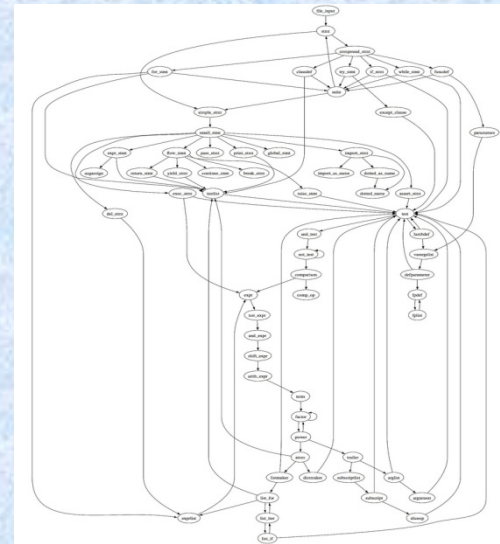
dan@dannorth.net

<http://dannorth.net>

@tastapod



<http://www.flickr.com/photos/nicksieger/280661836/>



<http://www.flickr.com/photos/nicksieger/281055485/>

@tastapod

Hard Things Made Easy

Bottleneck Analysis

Adrian Cockcroft
@adrianco

Netflix Inc.







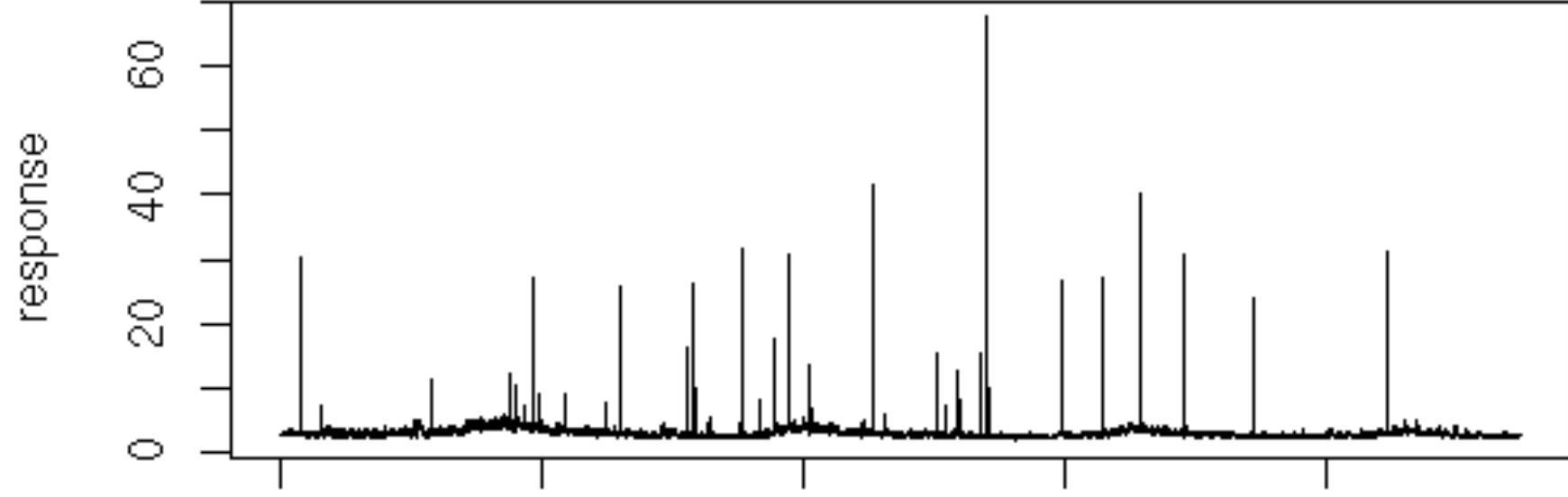


Code Like a Viking Pirate - Arrrrr

```
beer <- read.csv(url("http://beer.netflix.com/  
net?a=csv&gr=beer_operations&s=e-4d"))
```

```
response <- beer[,1]
```

```
plot(response, type="S",ylab="response")
```

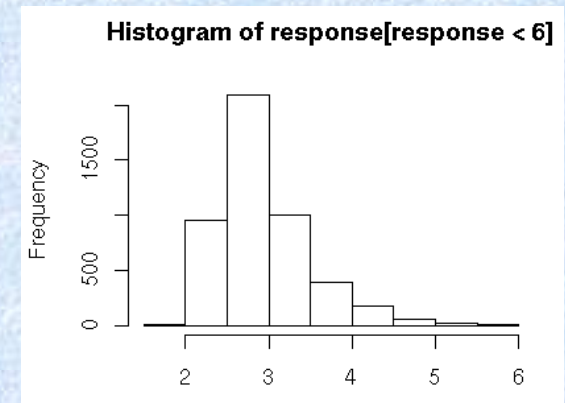
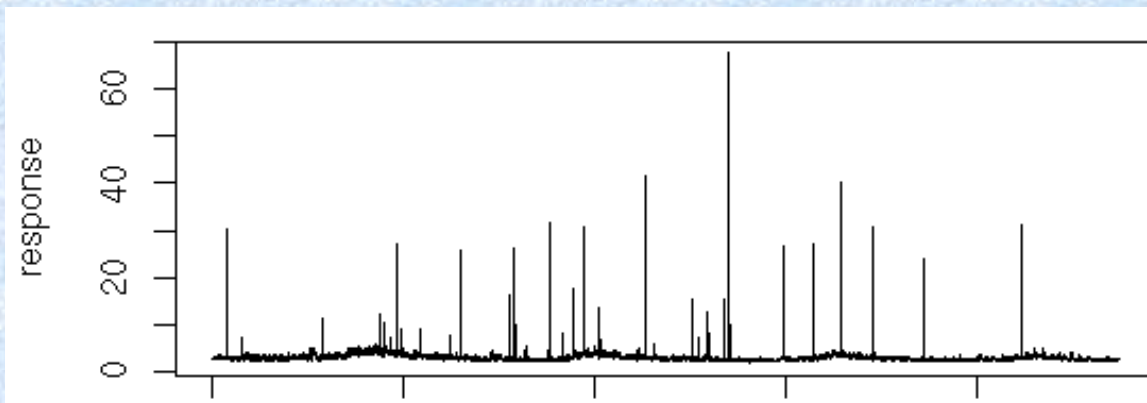
Hard Stuff

```
> summary(response)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.909  2.550  2.820  3.086  3.214  67.680

> quantile(response,c(0.95,0.99))
   95%    99%
4.149556 6.922115

> sd(response)
 1.941328

> mean(response) + 2 * sd(response)
 6.968416
```



Made



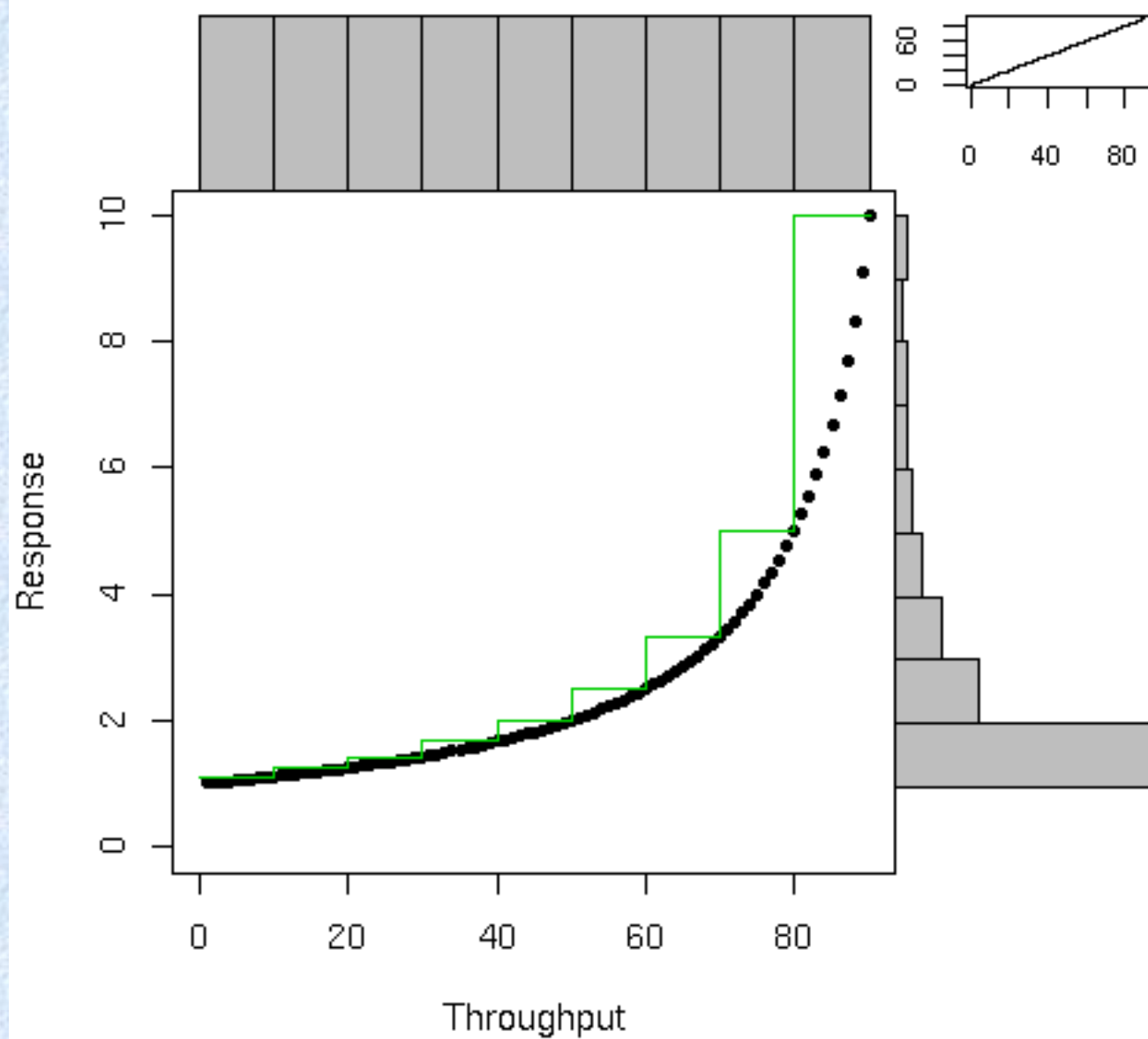
Easy

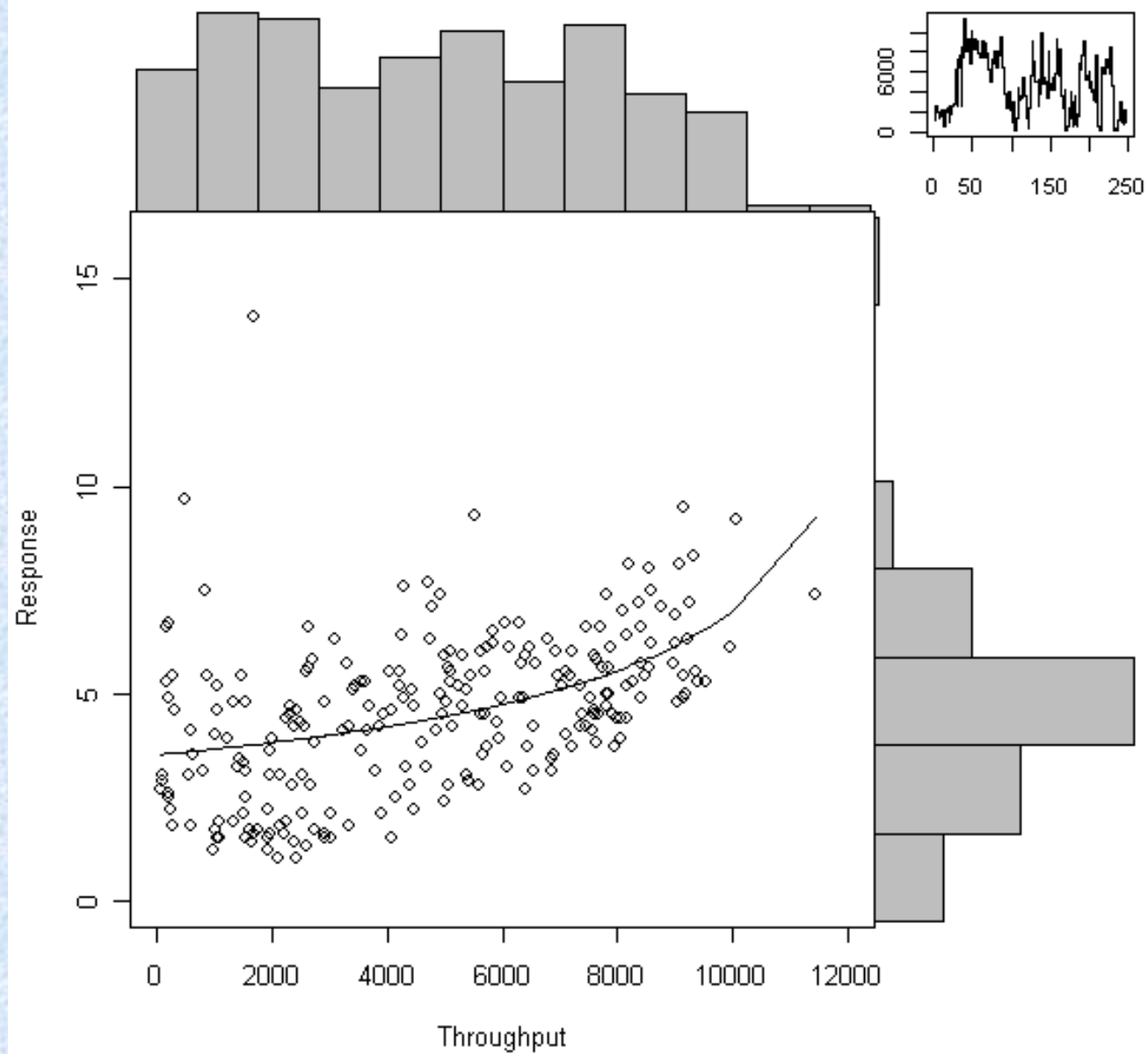

```
chp(beer[,1],beer[,2],q=1.0)
```

(See <http://perfcap.blogspot.com/search?q=chp>)

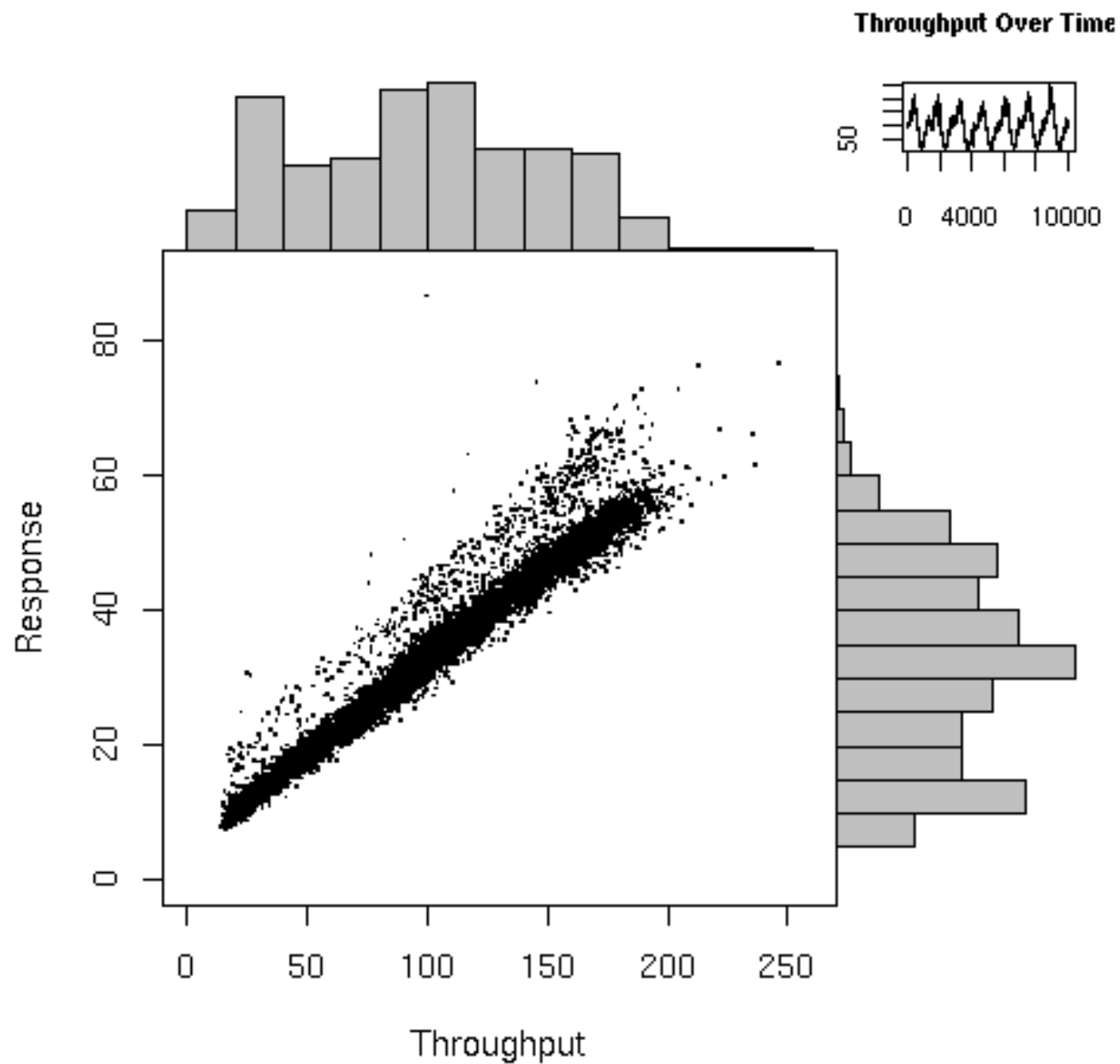
Headroom Plot

Throughput Over Time





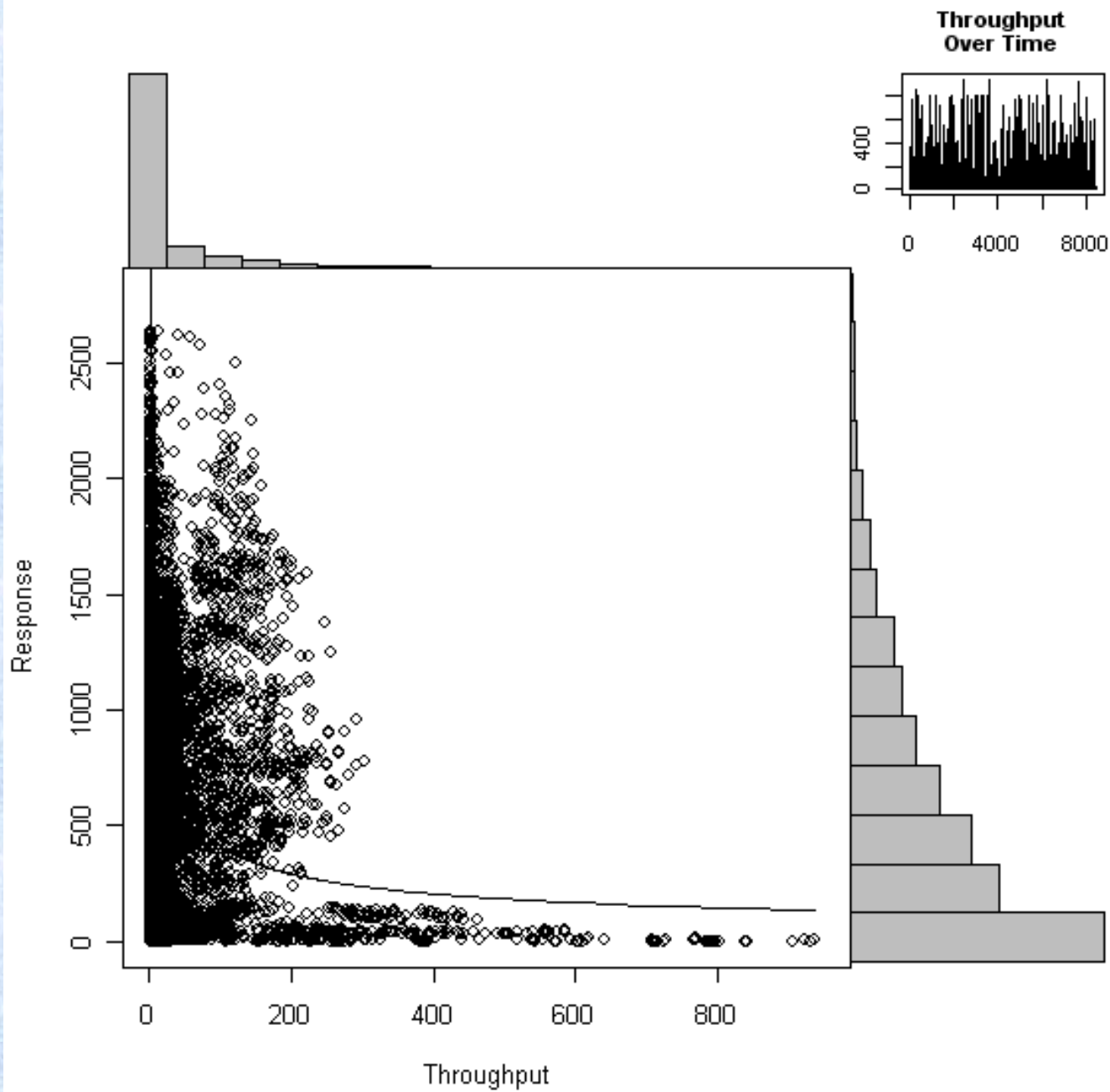


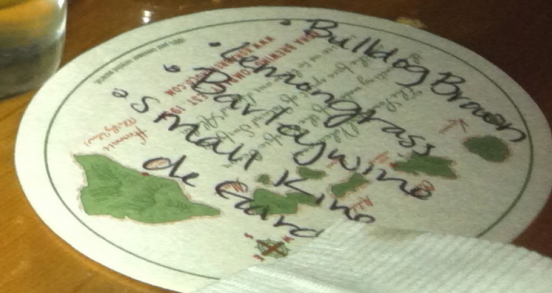




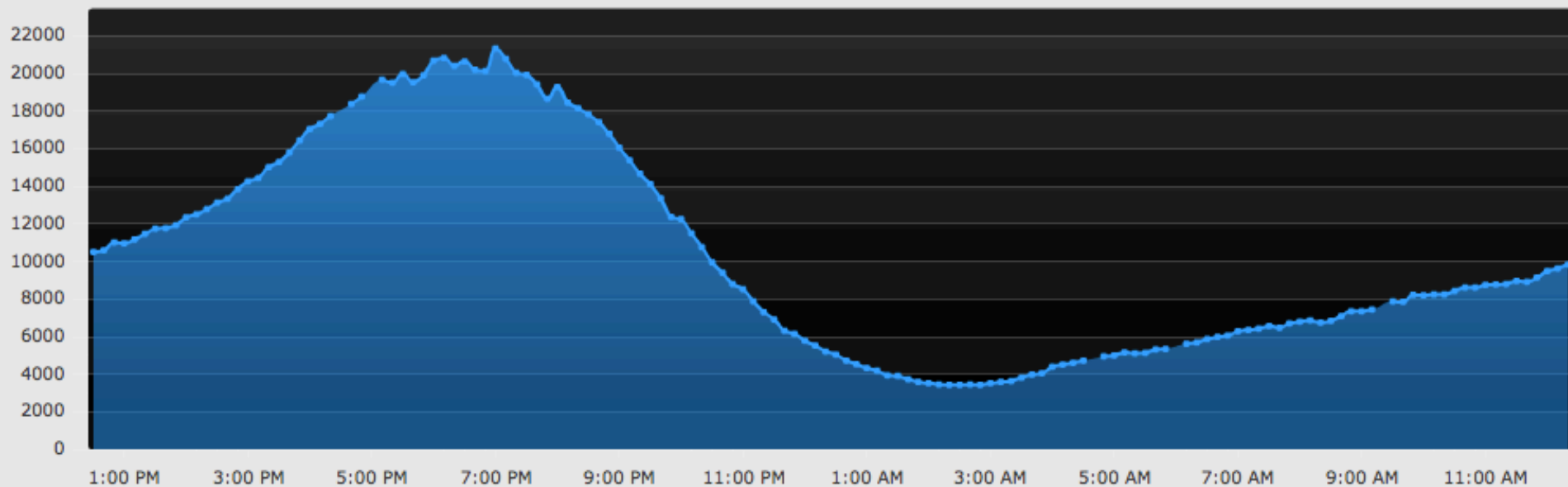





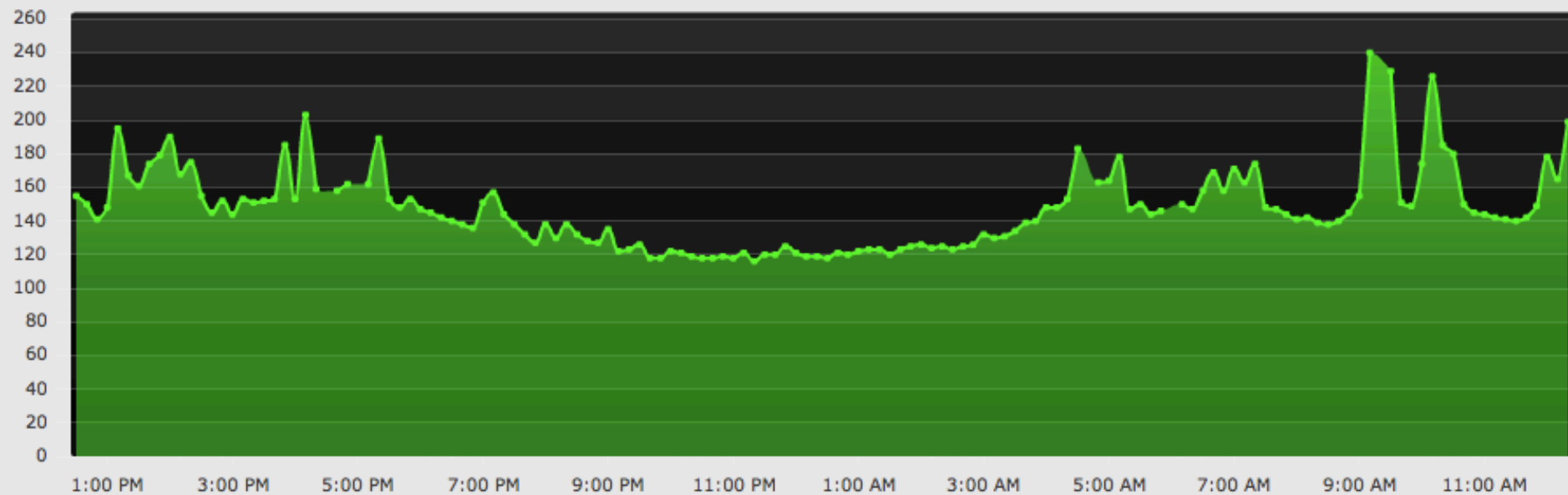


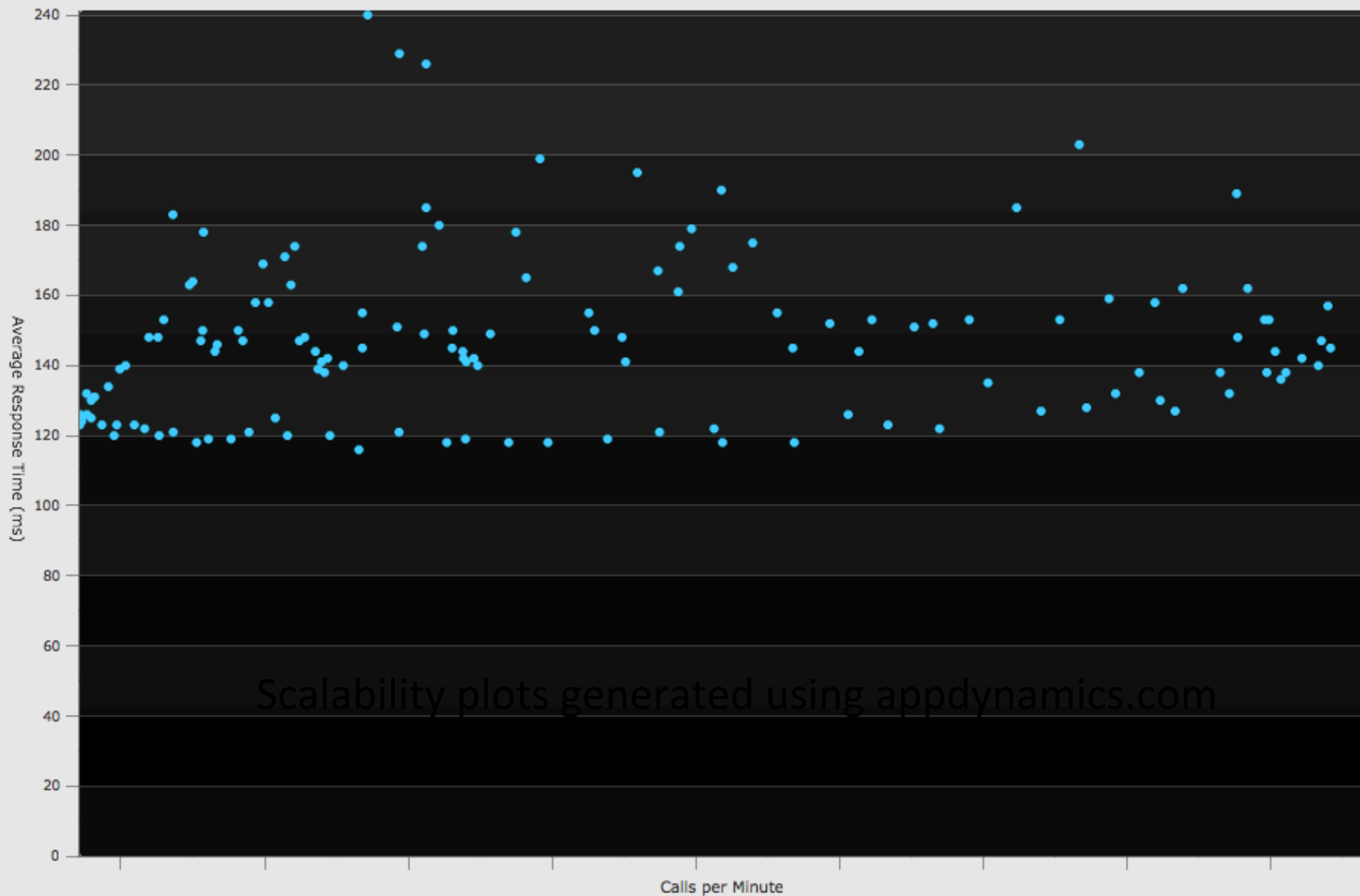


 [apiproxy-prod.AppleTV](#) Calls per Minute

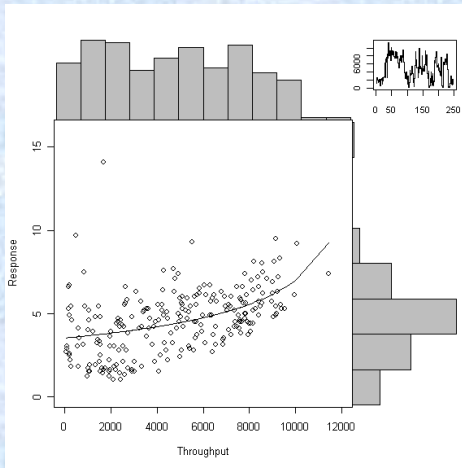


 [apiproxy-prod.AppleTV](#) Average Response Time (ms)



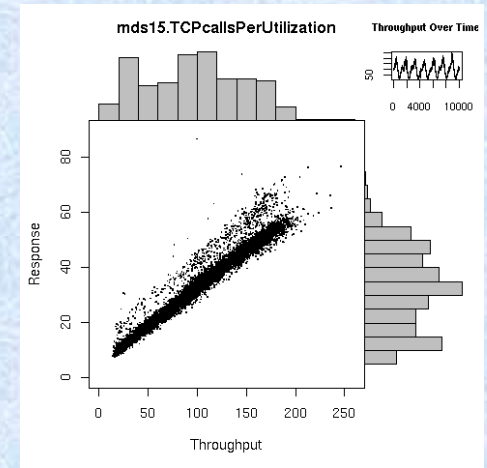


Scalability plots generated using appdynamics.com

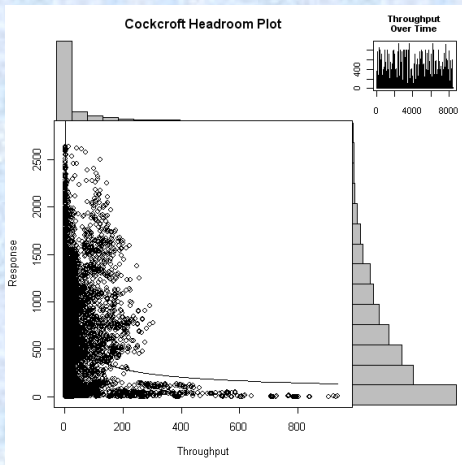


Well behaved

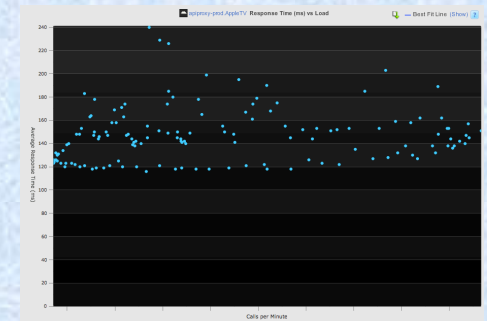
Hard Things Made Easy



Lock Contention



Oscillating, thread shortage



Looping autoscaled



Monads

Sadek Drobi
@sadache







An interface

Shared Semantics

Familiar, ready and operational

What is a Monad

An interface

Shared Semantics

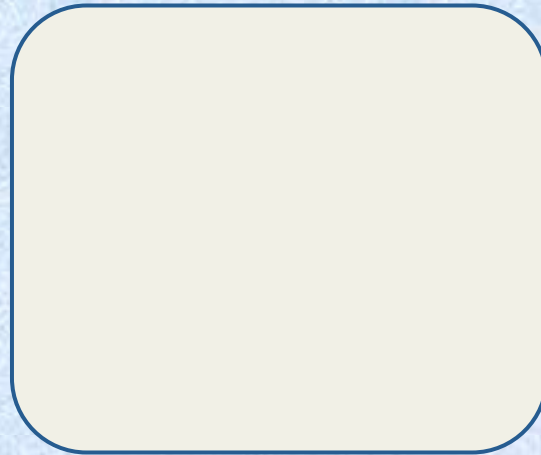
An Implementation

What is a Monad



A Container

What is a Monad



A Container of `a`

If we have



a

A Container of `a`

If we have



a

A List of `a`

If we have



a

An Option of `a`

If we have



a

A Tree of `a`

If we have



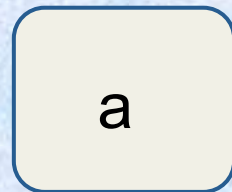
a

A Future of `a`

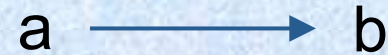
And I know how to transform

$a \longrightarrow b$

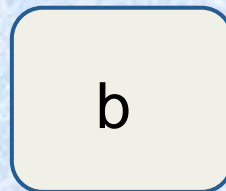
Provided that, it could be nice if



A Container of `a`



_if the container implements a way of getting a container of
`b`, handling all the necessary plumbing



Interesting!



A List of Int

i → "wow, got a " + i

What do we get?

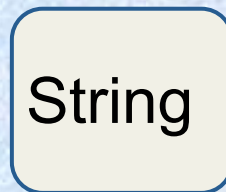
Interesting!



A List of Int

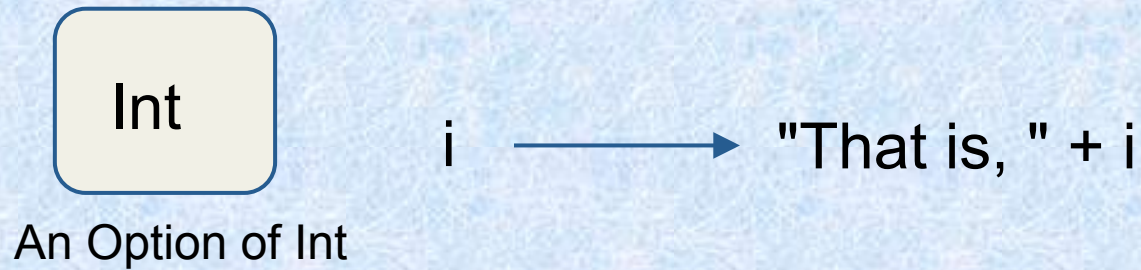
$i \longrightarrow \text{"That is, " + } i$

What do we get?

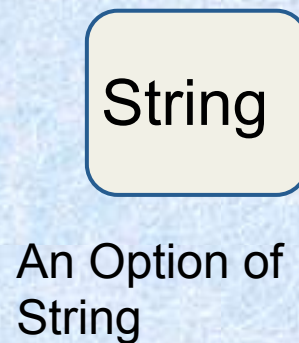


A List of String

Interesting!



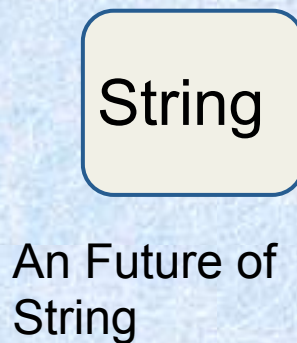
What do we get?



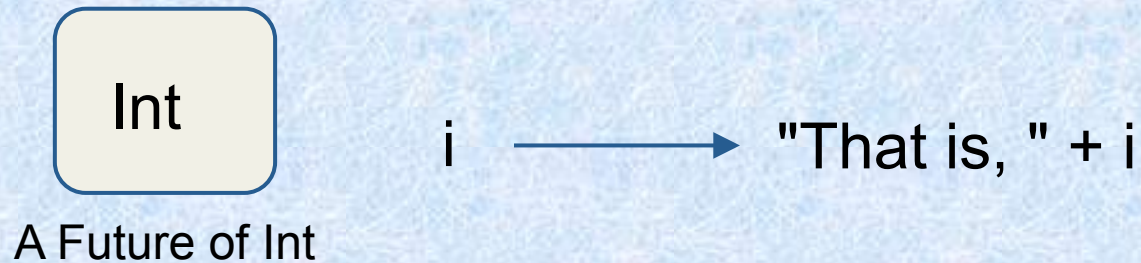
Interesting!



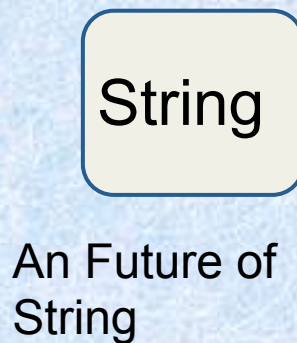
What do we get?



That is a Functor!



What do we get?



Functor interface

```
trait Functor[M[_]] {  
  def map[A,B]( ma:M[A], f: A => B): M[B]  
}
```

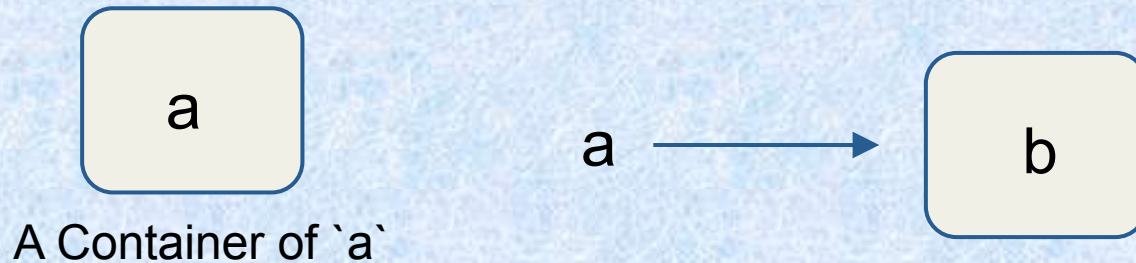
As a Functor interface implementer (API designer)

```
object ListFunctor extends Functor[List] {  
  
  def map[A,B]( ma:List[A], f: A => B): List[B] =  
    // apply the function to all elements and  
    // return a new list with results  
  
}
```


As a Developer

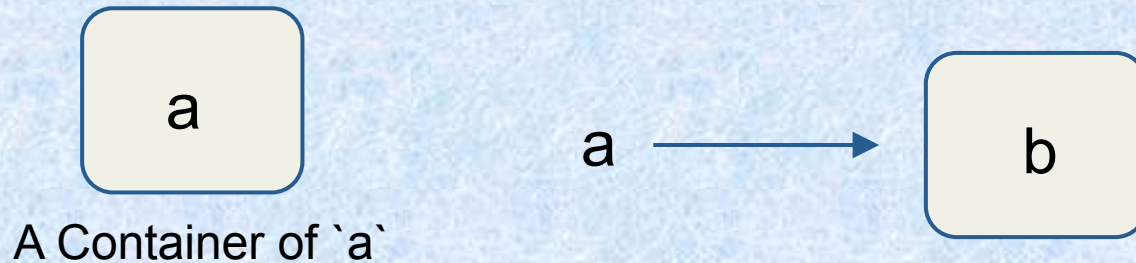
Use map as much as you want to transform what is
inside the container

Until, you run into a problem!

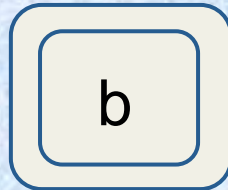


What do we get with a Functor?

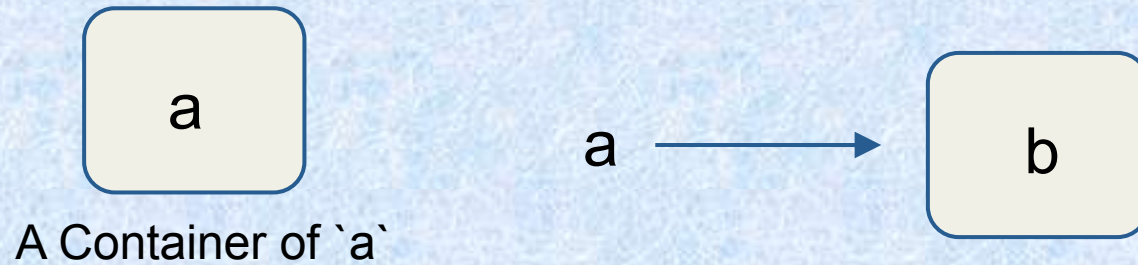
Until, you run into a problem!



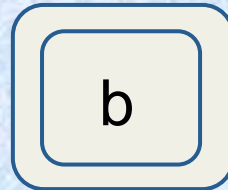
What do we get with a Functor?



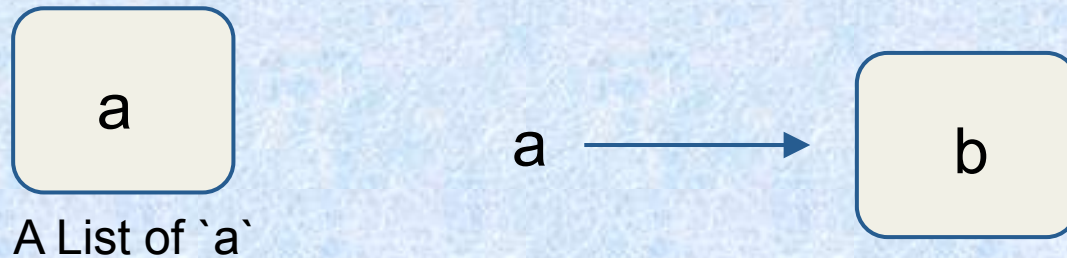
That is not nice



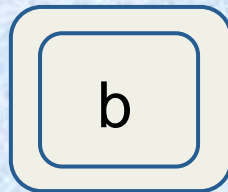
What do we get with a Functor?



That is not nice

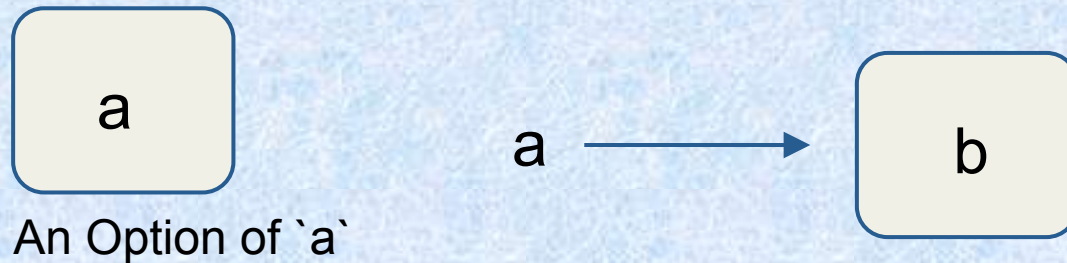


What do we get with a Functor?

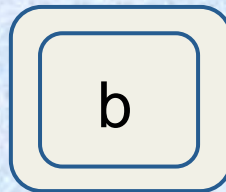


A List of Lists of
`a`

That is not nice

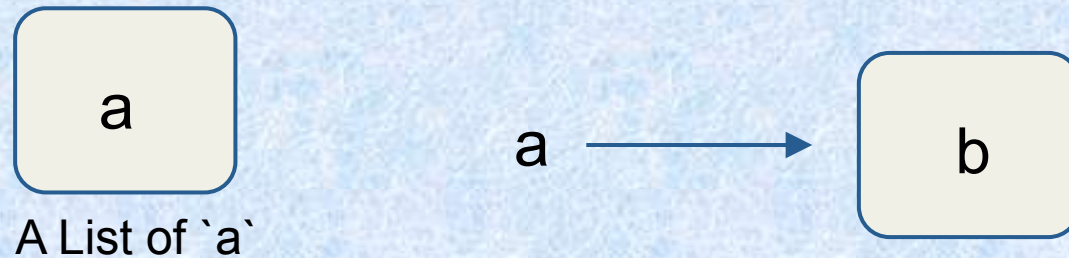


What do we get with a Functor?

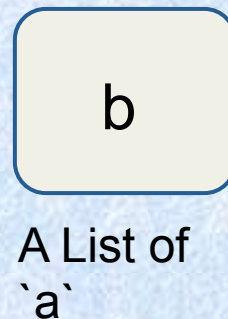


An Option of Option of
`a`

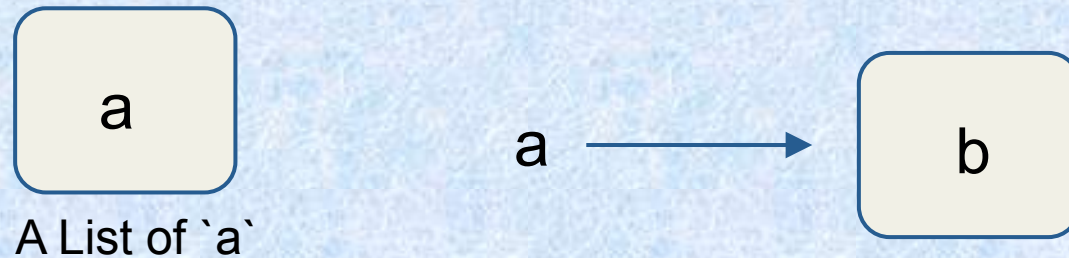
How nice would it be if,



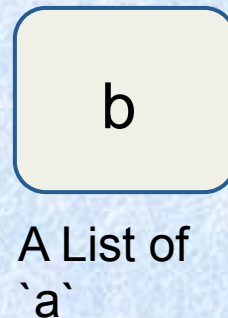
_if we could get this instead,
flattening the container!



And we get the Monad!



_if we could get this instead,
flattening the container!



Monad interface

```
trait Monad[M[_]] {  
  def map[A,B]( ma: M[A], f: A => B): M[B]  
  def flatMap[A,B](ma: M[A], f: A => M[B]): M[B]  
}
```


Almost!



Almost!

Some properties are not guaranteed with the structure,
you need to validate some laws

Left identity, Right Identity and Associativity

Category theory?

No more than a formal foundation that things won't go wrong with your monad implementation if you get the structure and the laws right

And that looks reassuring.