cisa | Centre for Intelligent Systems and their Applications

THE UNIVERSITY of EDINBURGH informatics

Declarative System Configurations with Constraints

Paul Anderson

dcspaul@ed.ac.uk

http://homepages.inf.ed.ac.uk/dcspaul

http://homepages.inf.ed.ac.uk/dcspaul/publications/paris-2014.pdf

Overview

System configuration

- imperative approaches
- a more declarative approach

Specifications with constraints

- aspect composition
- autonomics & error-recovery

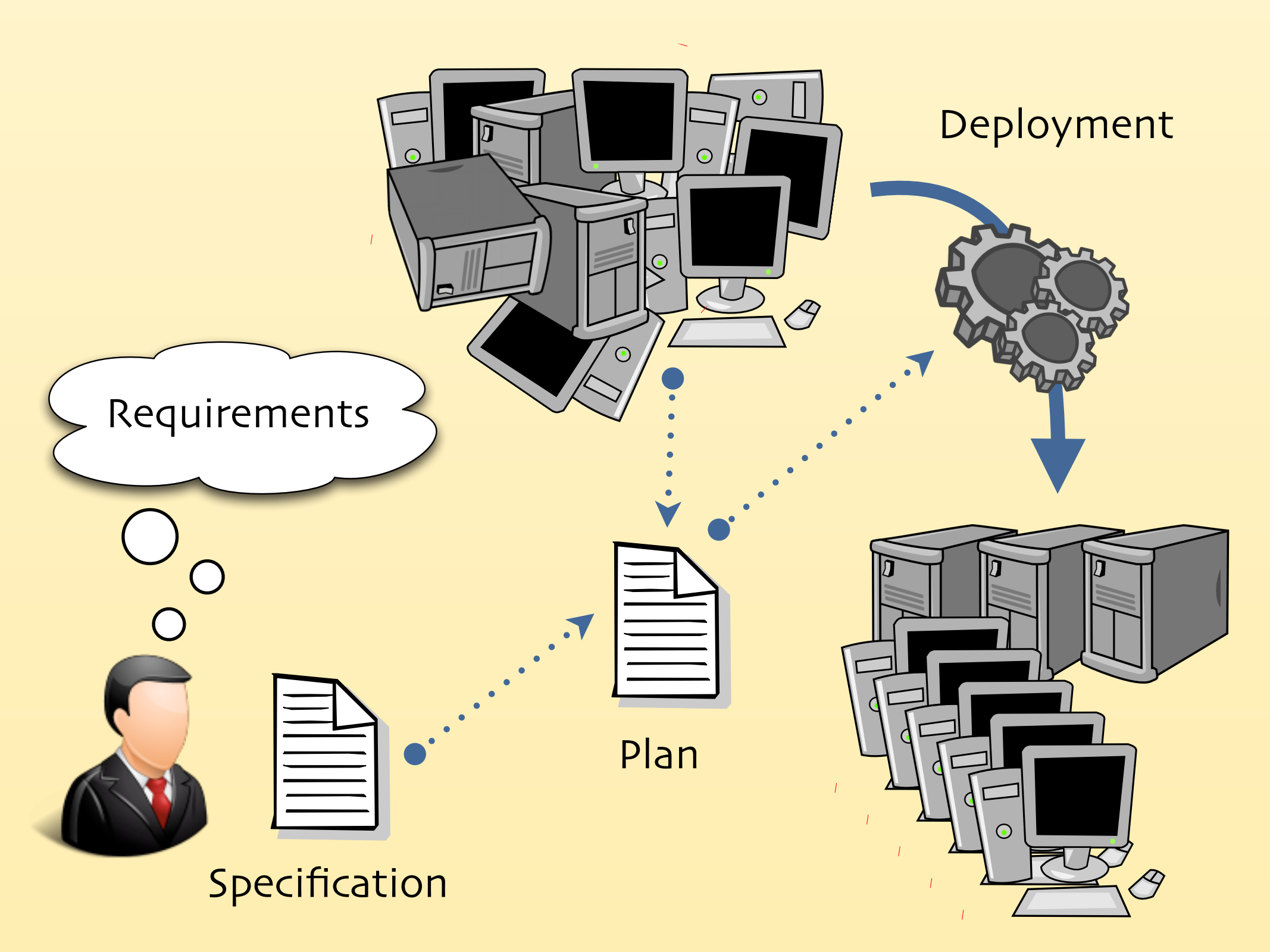
Confsolve

- a constraint-based specification language

System Configuration

"Programming the infrastructure"

- corporate IT infrastructure, "grid", "datacentre", "cloud service", distributed application, …
- virtual machines & networks mean that everything is now "soft"


Deployment
Requirements
Plan
Specification

Imperative Approaches

The traditional approach is to use "imperative" scripts

- these are created by a human to implement a workflow which they have designed to achieve the desired state
- workflows may run in response to "events" (eg. a failure)

But ..

- there is no often explicit specification of the desired state
  - even if there is, it is not easy to prove that the workflow achieves it
- a new workflow is needed for every new initial state
  - and/or the workflow includes complex hand-coded conditionals
  - for use in autonomic recovery, the number of possible states is large

A Declarative Approach

We advocate a more "declarative" approach

- the human specifies the desired state
- a monitoring system determines the current state
- a planner automatically creates a workflow
- a deployment engine executes this and validates the result

So ..

- the user provides (only) a specification of the final, desired state
  - and possibly some declarative constraints on the intermediate states
  - this is clearly separated from the actions required to achieve it
- the system can achieve this state from any starting point
  - if this is possible
- we can prove properties of the final (and intermediate) state

Configuration Languages

Imperative configuration uses conventional scripting languages

- or a DSL with a roughly equivalent power
  - they describe the process (computation) of changing the configuration

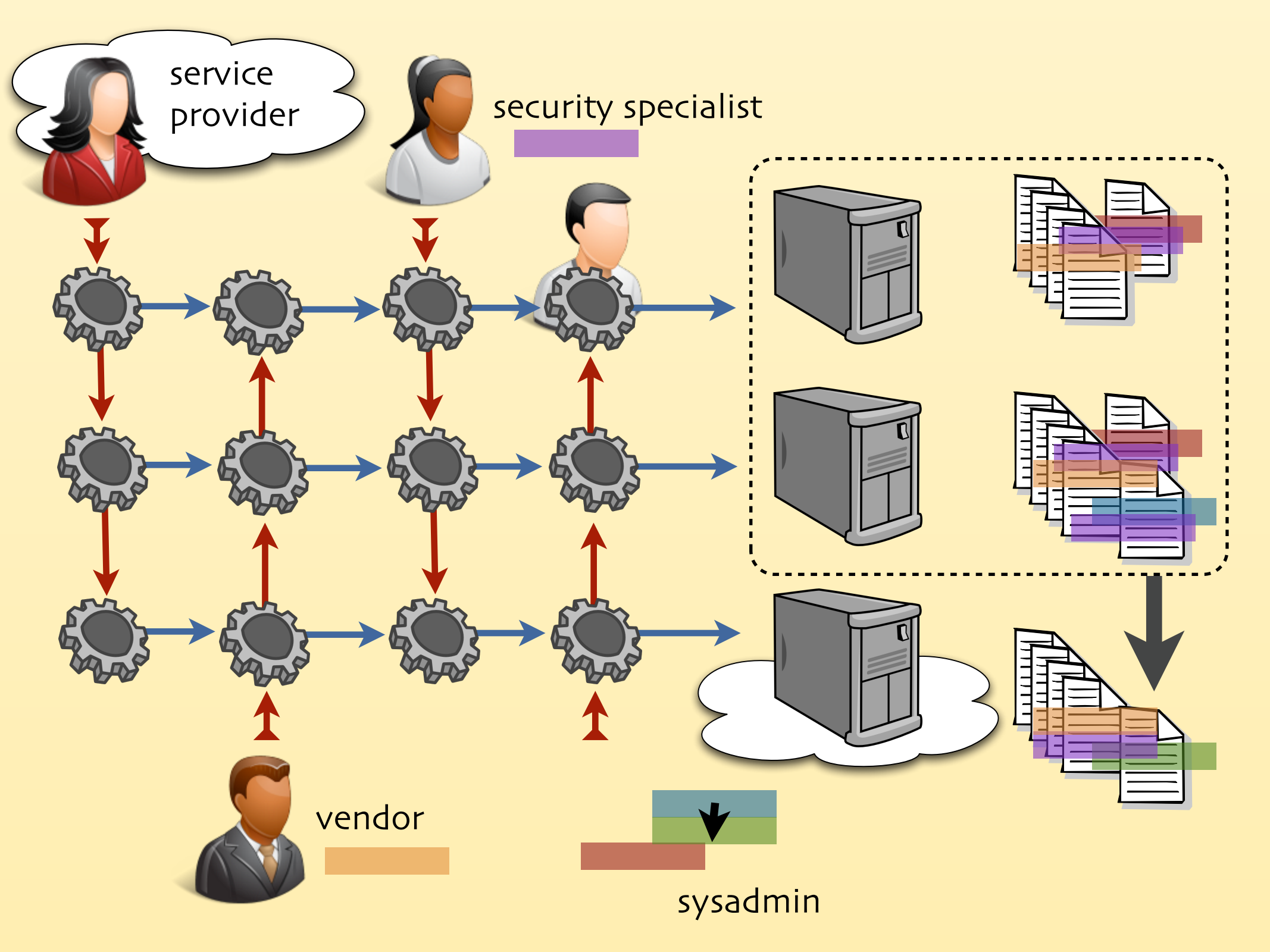
Declarative configuration languages are quite different

- they describe the desired state - not a computation
  - in theory, they should have a simpler semantics
  - and be easier to reason about
- they describe the requirements at a higher level
  - these are translated into explicit, detailed configuration parameters
- they compose the requirements from many independent people
  - the declarative nature allows us to do this composition …
- the deployment of the configuration is a separate problem

Aspects & Composition

A good configuration language can compose requirements

- this has no real equivalent in most programming languages


service provider
security specialist
vendor
sysadmin

Aspect Composition

Many different people are responsible for different "aspects"

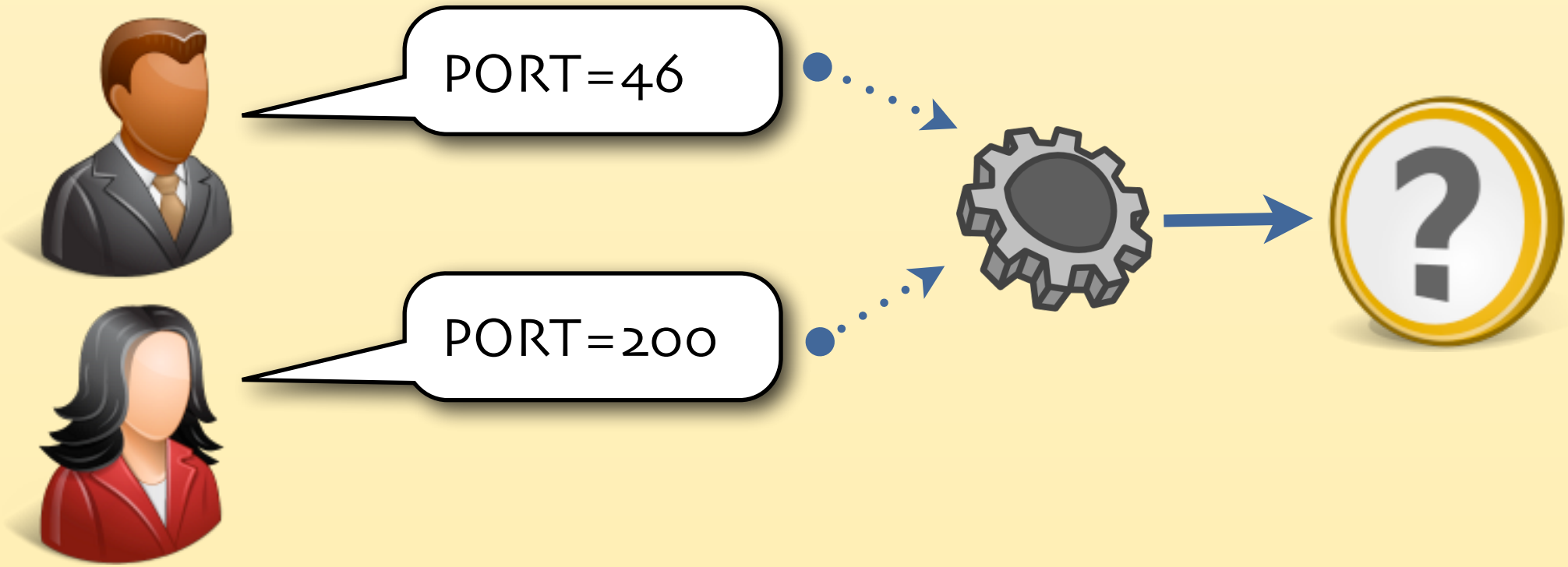
- one of our goals for a configuration language is to help people collaborate & compose their requirements without unnecessary conflict
- A configuration tool composes the independent "aspects" to form a consistent specification

Different tools support different languages and approaches

- "prototypes" and "instance inheritance" are common
- simple order precedence
- explicit composition functions

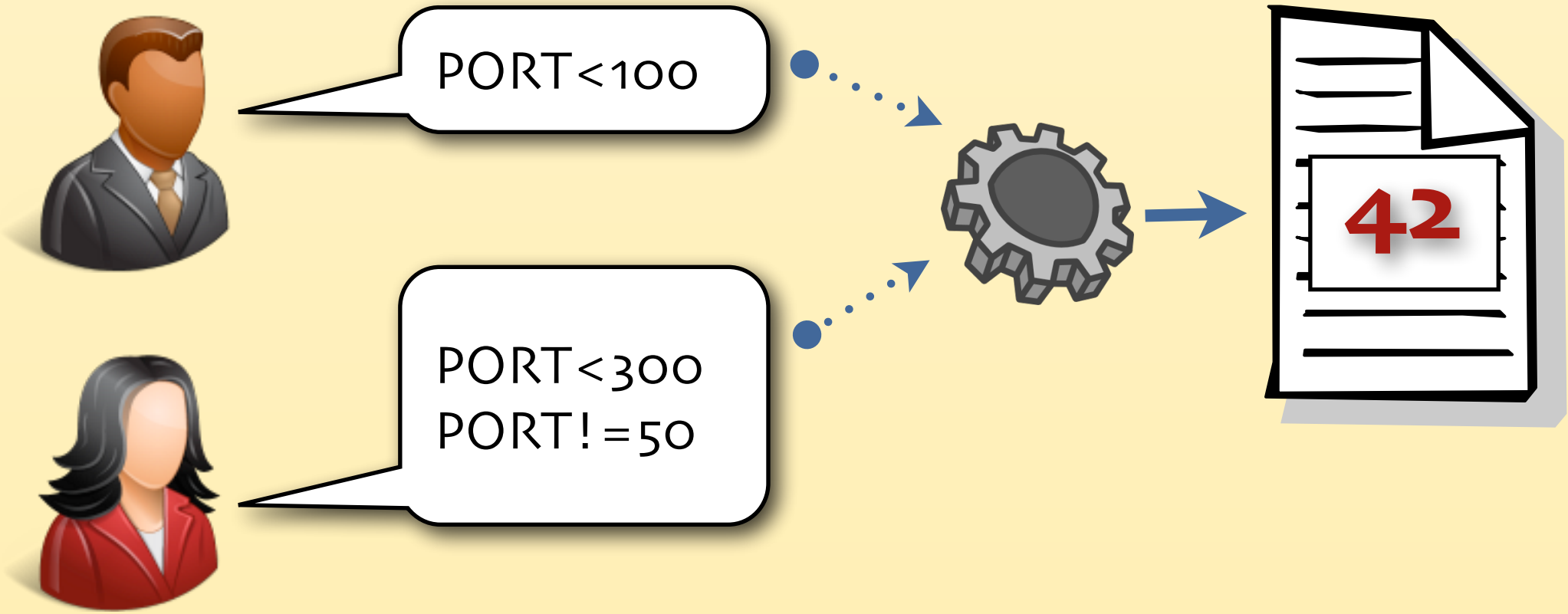
People's real requirements are often quite loose:

- "configure one machine as a web server" (but I don't care which)
- but most systems force the user to specify an arbitrary value



With a declarative approach, we can specify loose constraints ..

- this allows us to compose aspects without conflict or unnecessary negotiation



I want at least two DHCP servers on each network segment

I don't want any core services running on any machines that students are authorised to log in to

I want my two database servers to be on separate networks if possible for robustness

I need at least one database machine that students can log in to

Autonomics

Systems need the ability to reconfigure in response to failures and other external events

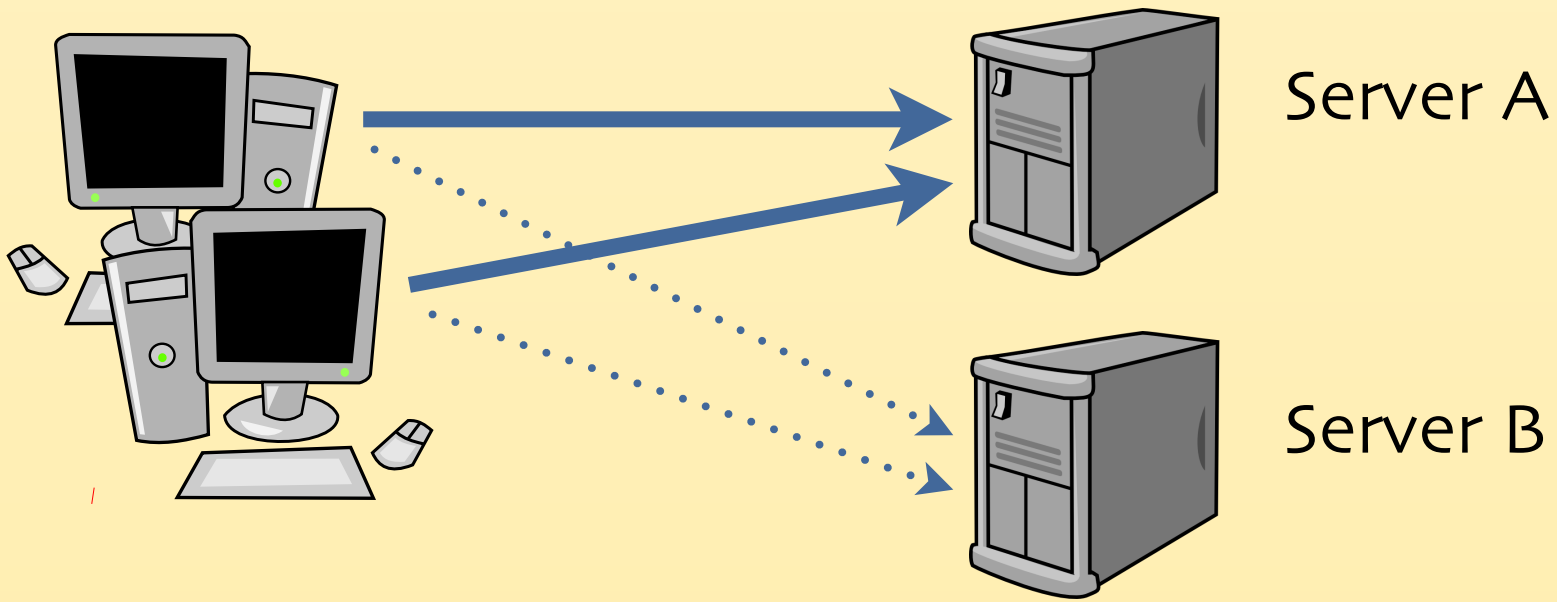
- traditionally, these involves "event-condition-action" (ECA) rules
- but we can use constraints to avoid these imperative specifications

Autonomic recovery …

use Server A

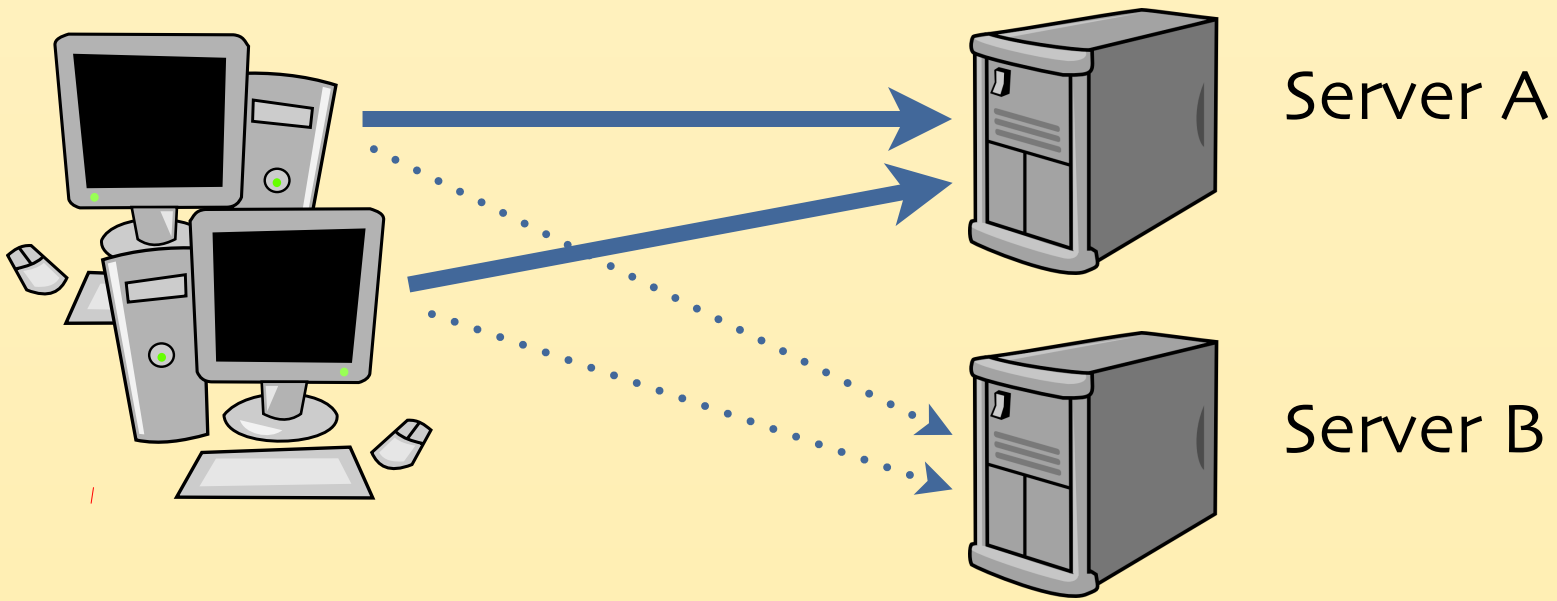
- we may have a declarative specification
- which requires an imperative ECA rule to handle autonomic reconfiguration

if Server A fails, then ….
do some imperative stuff
to change the configuration so that we use Server B instead



Using declarative constraints …

- use Server A or B
- don't use a failed Server
- prefer Server A



Confsolve

A constraint-based language for system configuration

- by John Hewson

Constraints

Using constraint solvers for configuration problems is not new

- Alloy for network configuration
- Cauldron (HP)
- VM allocation (Google challenge)

But we have a different motivation which changes the emphasis

- we want to integrate the constraints with a (usable) configuration language to support a separation of concerns
- the constraint problems are often comparatively simple to solve, but they are embedded in large volumes of "constant" configuration data
- some specific properties are important (see later) ...
  - preferences (soft constraints)
  - stability

Modelling

The most popular practical configuration languages ..

- are very good at reliably deploying large numbers of configuration parameters to large numbers of machines
- but they are not good at modelling higher-level abstractions
- they have "evolved" gradually without a clear semantics
- and they have implementations which are not amenable to experimental extensions

Confsolve is an experimental constraint-based configuration language

- supports the necessary modelling
- generates an intermediate language which can be transformed fairly easily into an existing configuration language

Confsolve

An experimental constraint-based configuration language

- by John Hewson<john.hewson@ed.ac.uk>
  http://homepages.inf.ed.ac.uk/s0968244/
  (Sponsored by Microsoft Research)
- a general-purpose configuration language
  - no domain-specific knowledge
  - output can easily be transformed into some other language (eg. Puppet)
- the data model is an object-oriented hierarchy
  - constraints are possible at all levels
- compiles down to a standard constraint solver (MiniZinc)
- supports soft constraints and optimisation
- has a formal semantics for the translation
- supports "change minimisation"

Some Confsolve Classes

```
class Service {
   var host as ref Machine
   ...
}
class Datacenter {
   var machines as Machine[8]
   ....
}
class Machine { }
class Web_Srv extends Service { }
class Worker_Srv extends Service { }
class DHCP_Srv extends Service { }
```

Two Datacenters & Three Services

```
var cloud as Datacenter
var enterprise as Datacenter

var dhcp as DHCP_Service[2]
var worker as Worker_Service[3]
var web as Web_Service[3]
```

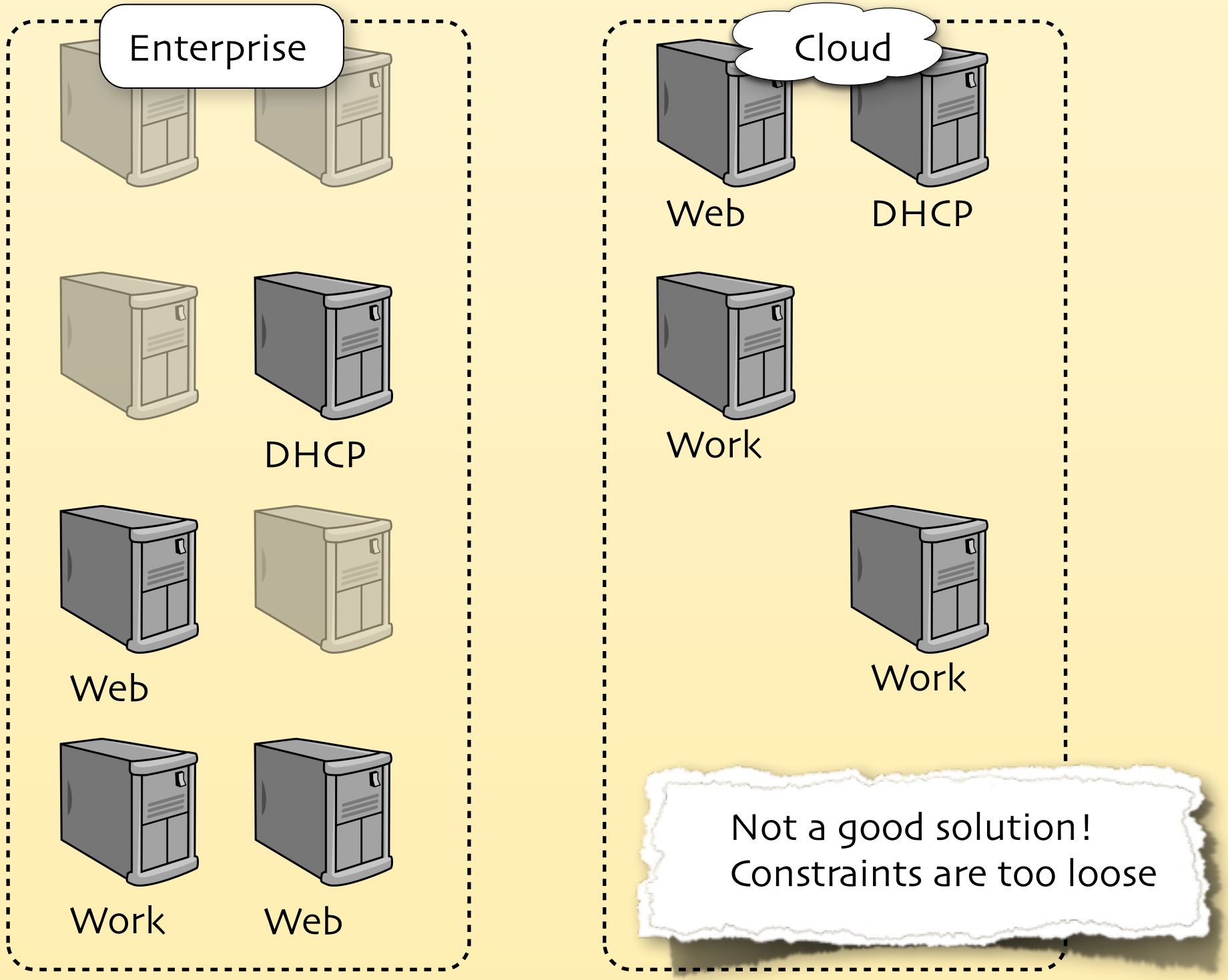
A Constraint

```
var services as ref Service[7]

where foreach (s1 in services) {
   foreach (s2 in services) {
      if (s1 != s2) {
         s1.host != s2.host
      }
   }
}
```

No two services on the same machine:

- this generates a correct configuration
  - no explicit assignment at all
  - not just validation
- this can be independently authored
  - no collaboration with the service authors, or system managers is required


Enterprise
DHCP
Web
Work
Web
Cloud
Web
DHCP
Work
Work
Not a good solution!
Constraints are too loose

An Optimisation Constraint

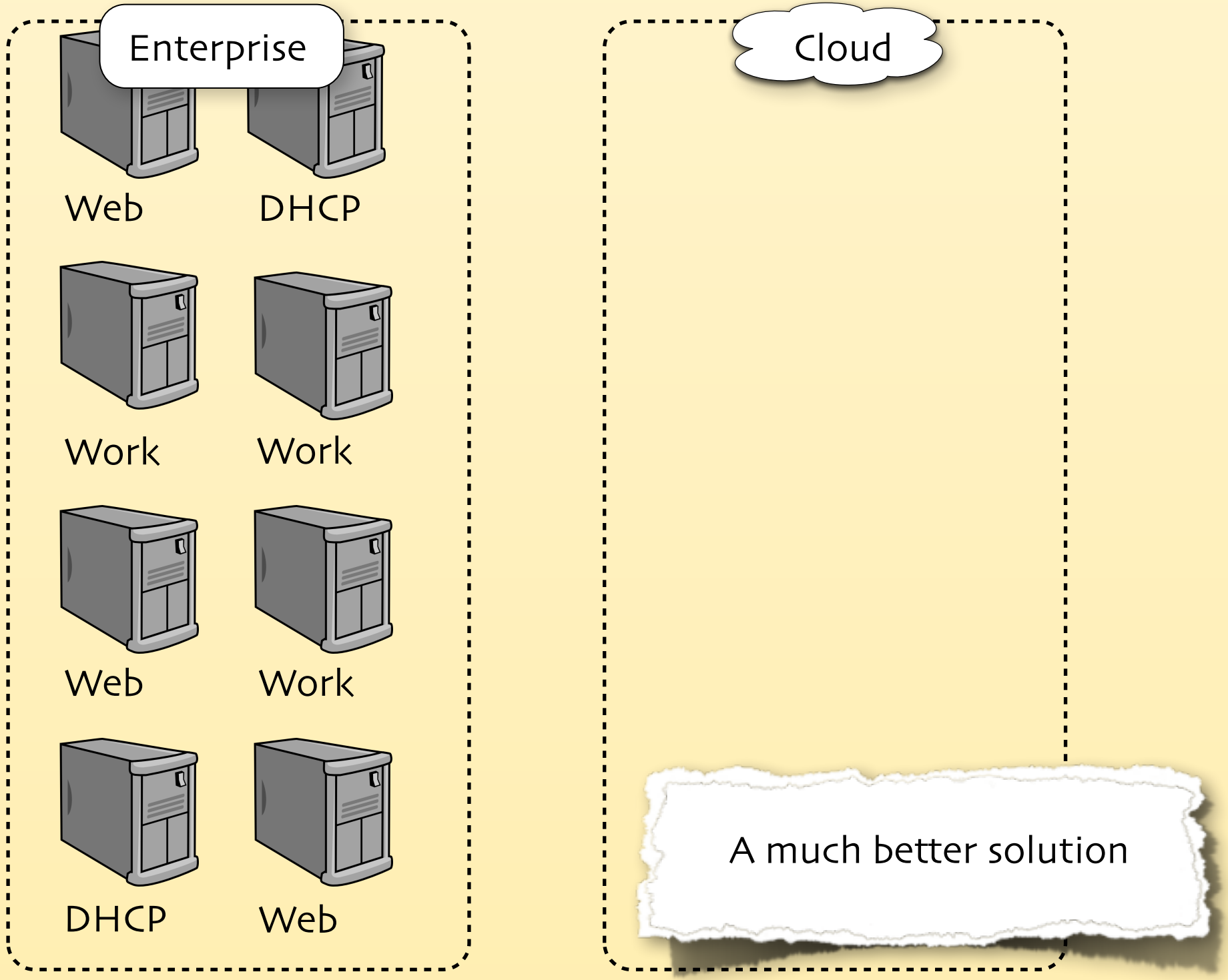
```
var utilisation as int

where utilisation == count (
  s in services
  where s.host in enterprise.machines)

maximize utilisation
```

"Favour placement of machines in the enterprise"

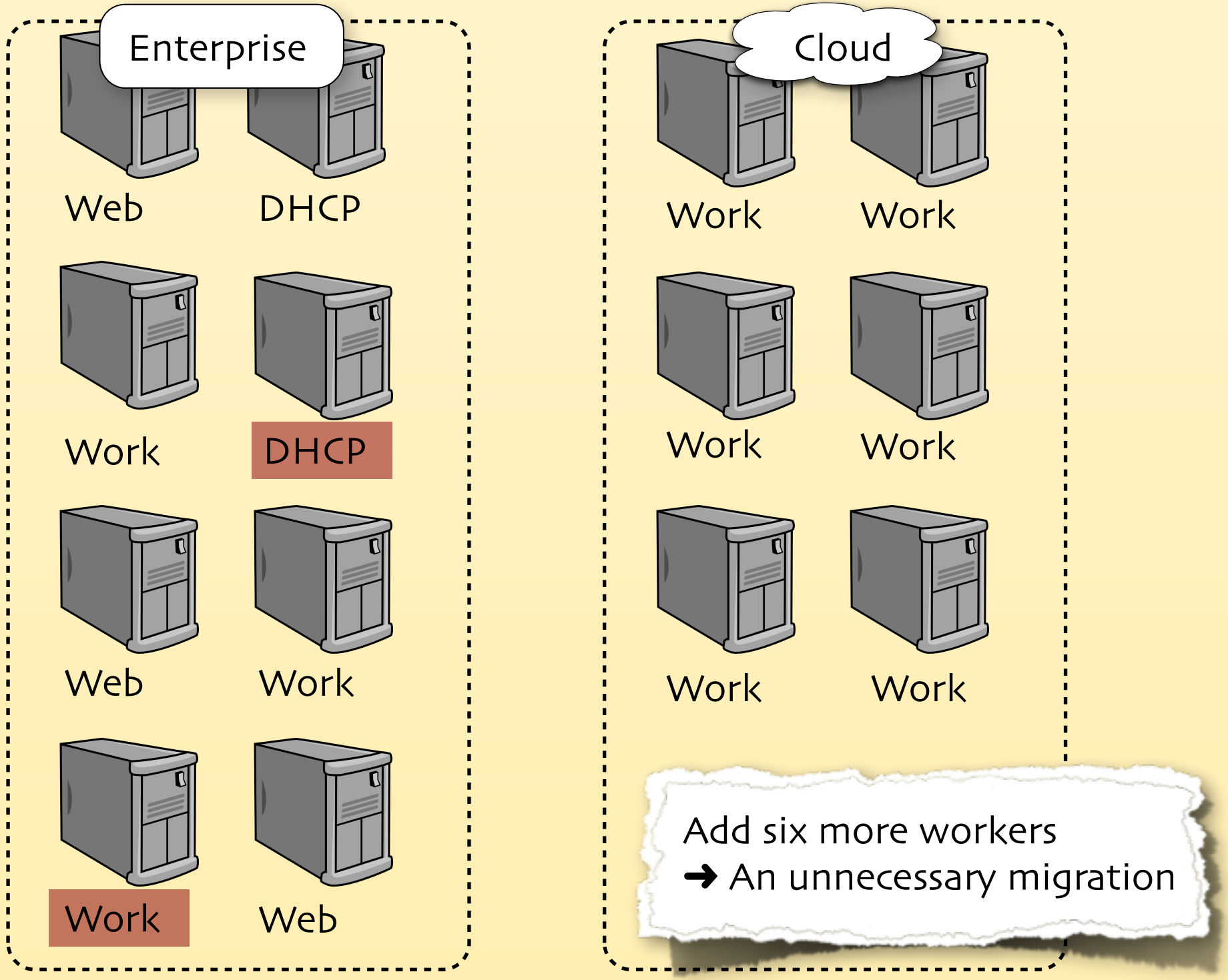
- this policy can be defined completely independently


Enterprise
Web
DHCP
Work
Work
Web
Work
DHCP
Web
Cloud
A much better solution

Add Six More Workers

```
var cloud as Datacenter
var enterprise as Datacenter

var dhcp as DHCP_Service[2]
var worker as Worker_Service[3]
var worker as Worker_Service[9]
var web as Web_Service[3]
```

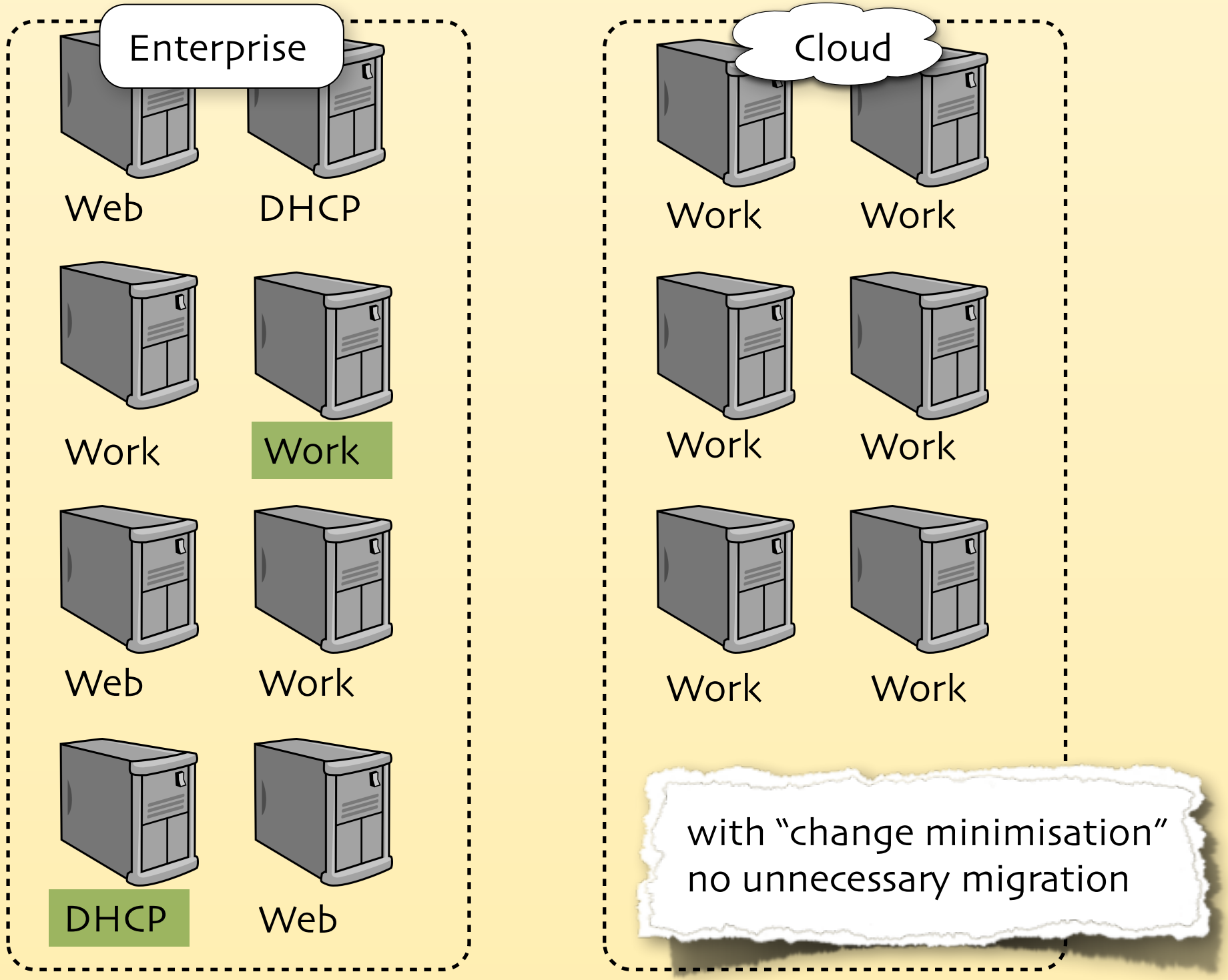

Enterprise
Web
DHCP
Work
DHCP
Web
Work
Work
Web
Cloud
Work
Work
Work
Work
Work
Work
Add six more workers
➜ An unnecessary migration

Minimising Changes

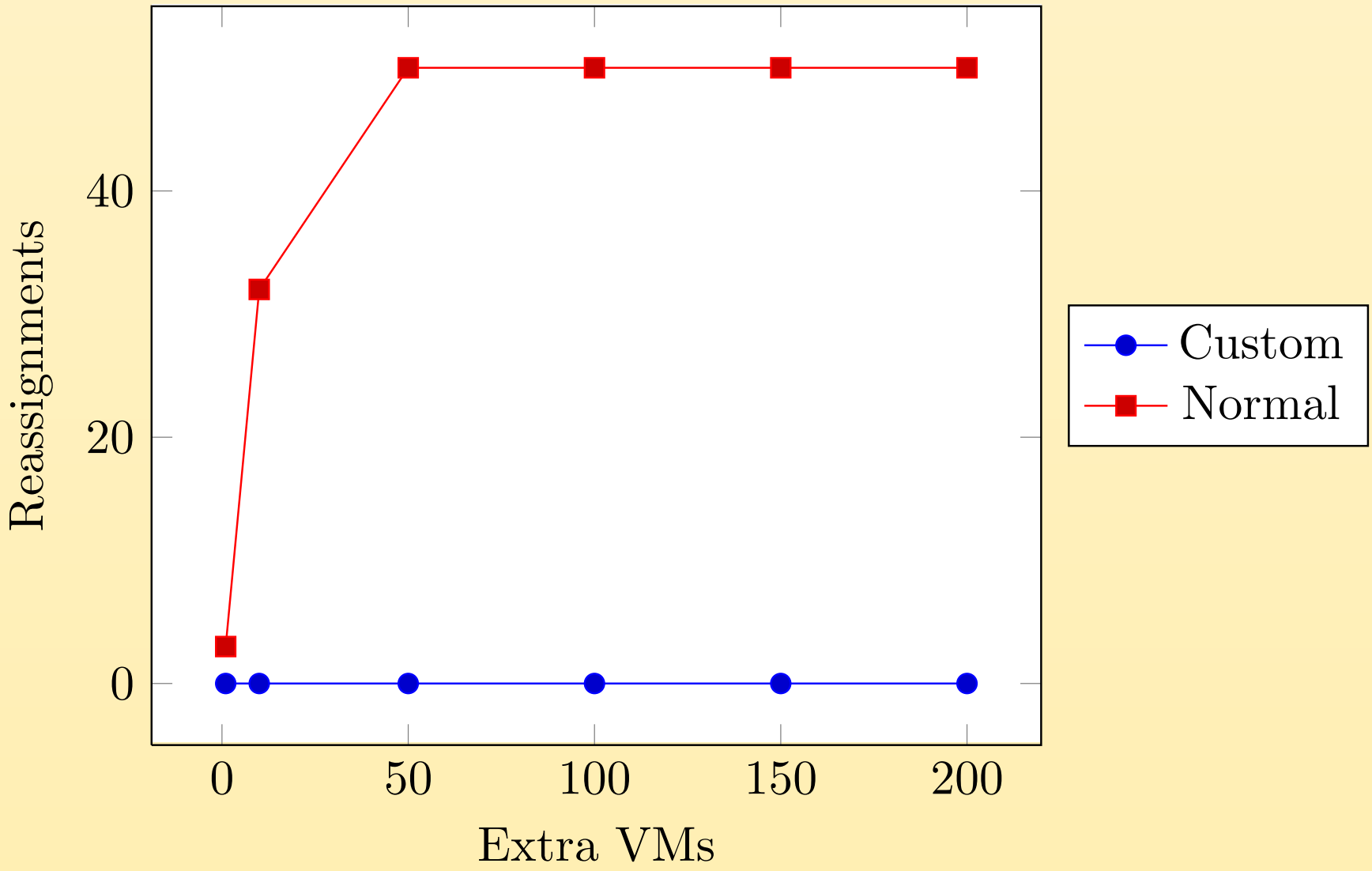
```
change {
  forall s in services {
    s.host = ~s.host;
  };
}
```

"Don't move machines once they have been allocated"

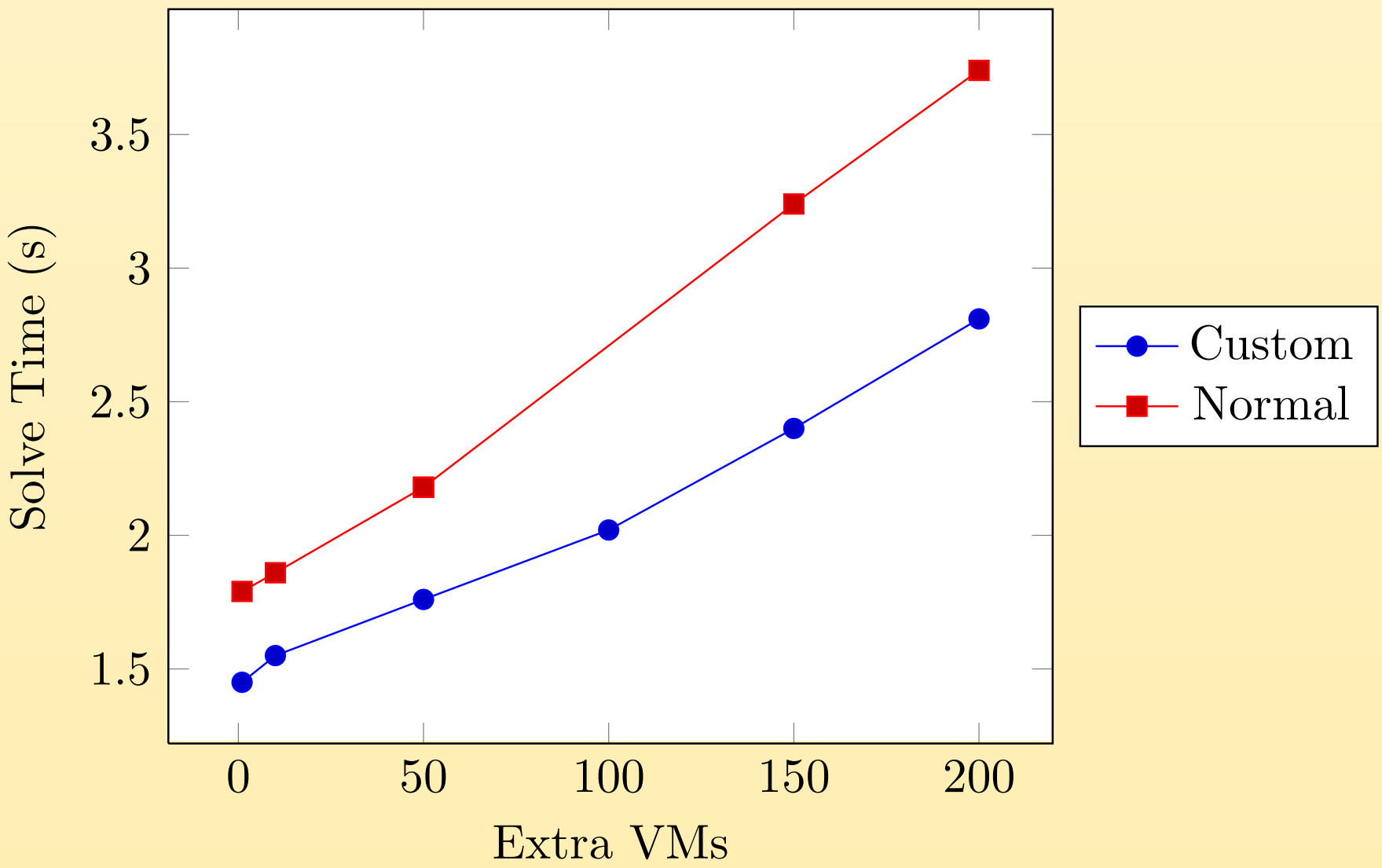
- "change" block is only valid when we have a previous configuration
- **~s** is the "previous" value
- this is a "hard constraint"
  - it could also have been a maximise/minimise constraint


Enterprise
Web
DHCP
Work
Work
Web
Work
DHCP
Web
Cloud
Work
Work
Work
Work
Work
Work
with "change minimisation"
no unnecessary migration

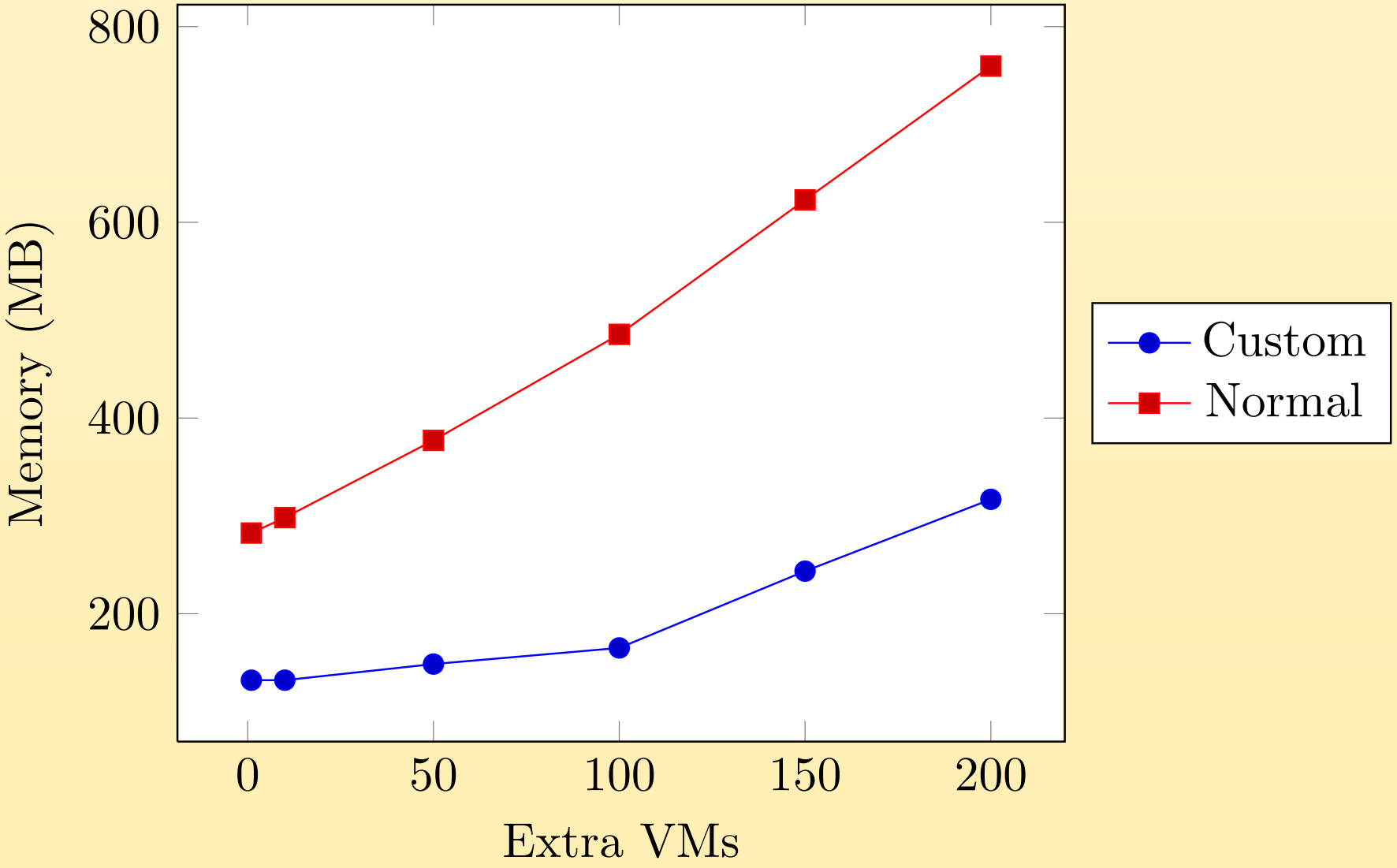
Reassignments



Time


Solve Time (s)
3.5
3
2.5
2
1.5
0
50
100
150
200
Extra VMs
Custom
Normal

Memory


800
600
400
200
Memory (MB)
0
50
100
150
200
Extra VMs
Custom
Normal

What's Good ?

Users can specify and change their own requirements completely independently

- and the resulting configurations are guaranteed to match the requirements

If some constraint changes, the system can automatically generate a new valid configuration (if one exists)

- things may change because of requirement changes
- or, for example, failures
- the deployment of the new configuration can be scheduled with automated planning tools

When the system reconfigures, it can do so with the minimum disruption necessary to meet the final requirements

What's Not So Good ?

It is very hard to specify comparative "costs"

- I could leave one service unnecessarily in the cloud, or I could move it back into the datacenter, but I would need to shuffle ten other servers to do so - which is best?

It is quite hard to avoid over-specifying or under-specifying constraints

- we either miss good solutions, or deploy bad ones

It can be hard for humans to predict the effects

- sysadmins are very nervous with this degree of automation

Sometimes there may be no solution

- and it is difficult to understand why

Performance can be unpredictable

- it is not always obvious what is computationally expensive

Some Conclusions

Constraint-based (declarative) configuration languages seem promising

- they are capable of supporting the automatic composition of intersecting aspects
- but a fully-general constraint-solver is probably not appropriate for production use
- some human-factors research would be very useful to determine typical usage patterns which could be incorporated into a production language in a more usable way

We need better configuration languages & implementations

- which support higher-level modelling
- and have clearer semantics
- and extensible implementations

Current Work

Some things I am interested in …

- configuration specification languages and semantics
  - making it clearer (and less error prone) for users to specify their requirements
- "provenance" and security
  - who is responsible for what?
- automated planning and deployment
  - distributed planning and agent-based negotiation of configurations

cisa | Centre for Intelligent Systems and their Applications

THE UNIVERSITY of EDINBURGH informatics

Declarative System Configurations with Constraints

Paul Anderson

dcspaul@ed.ac.uk

http://homepages.inf.ed.ac.uk/dcspaul

http://homepages.inf.ed.ac.uk/dcspaul/publications/paris-2014.pdf