THE UNIVERSITY of EDINBURGH
informatics

Describing System Configurations with L3

Paul Anderson

dcspaul@ed.ac.uk

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

System configuration

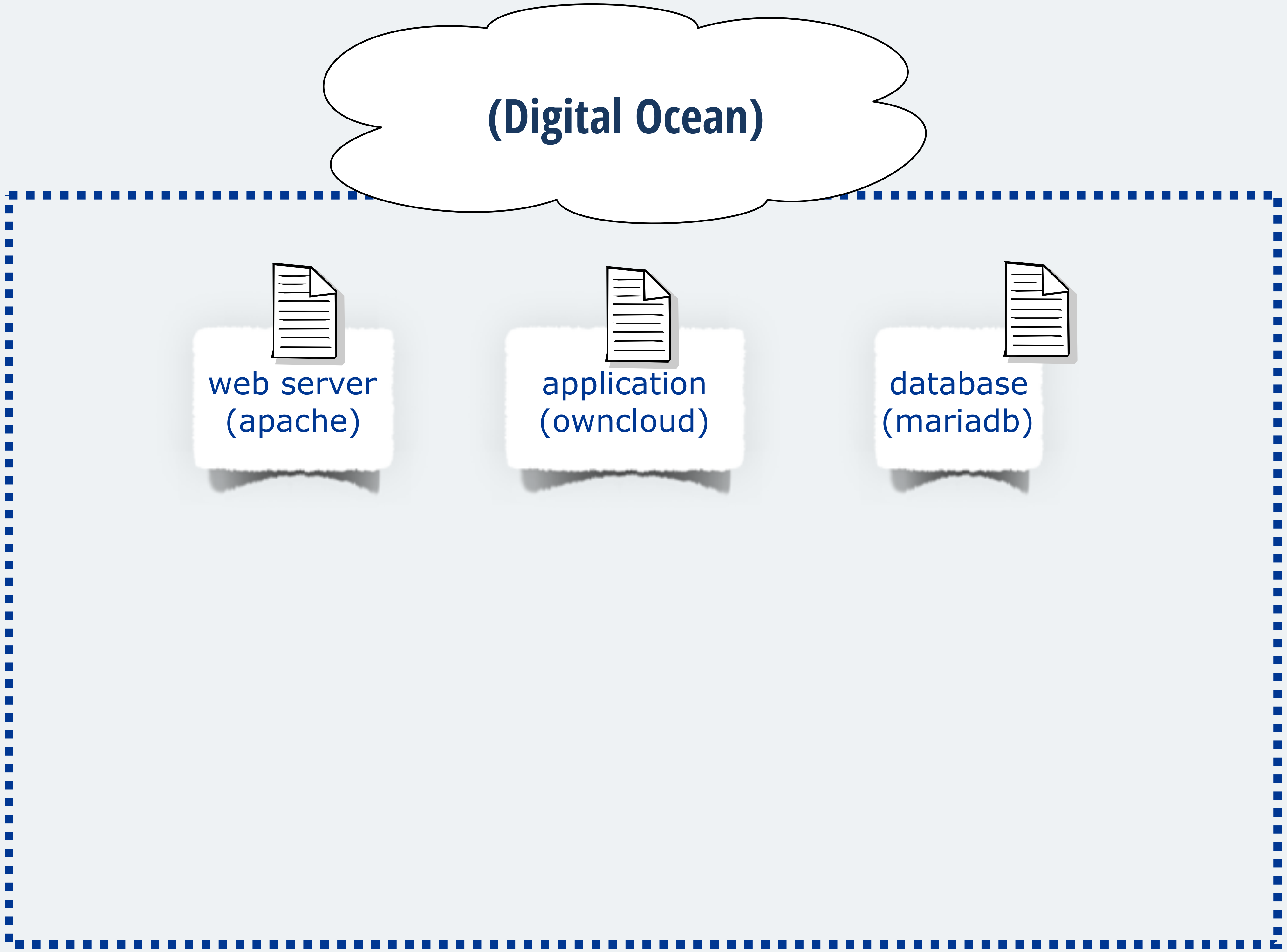
Is used in critical situations ...

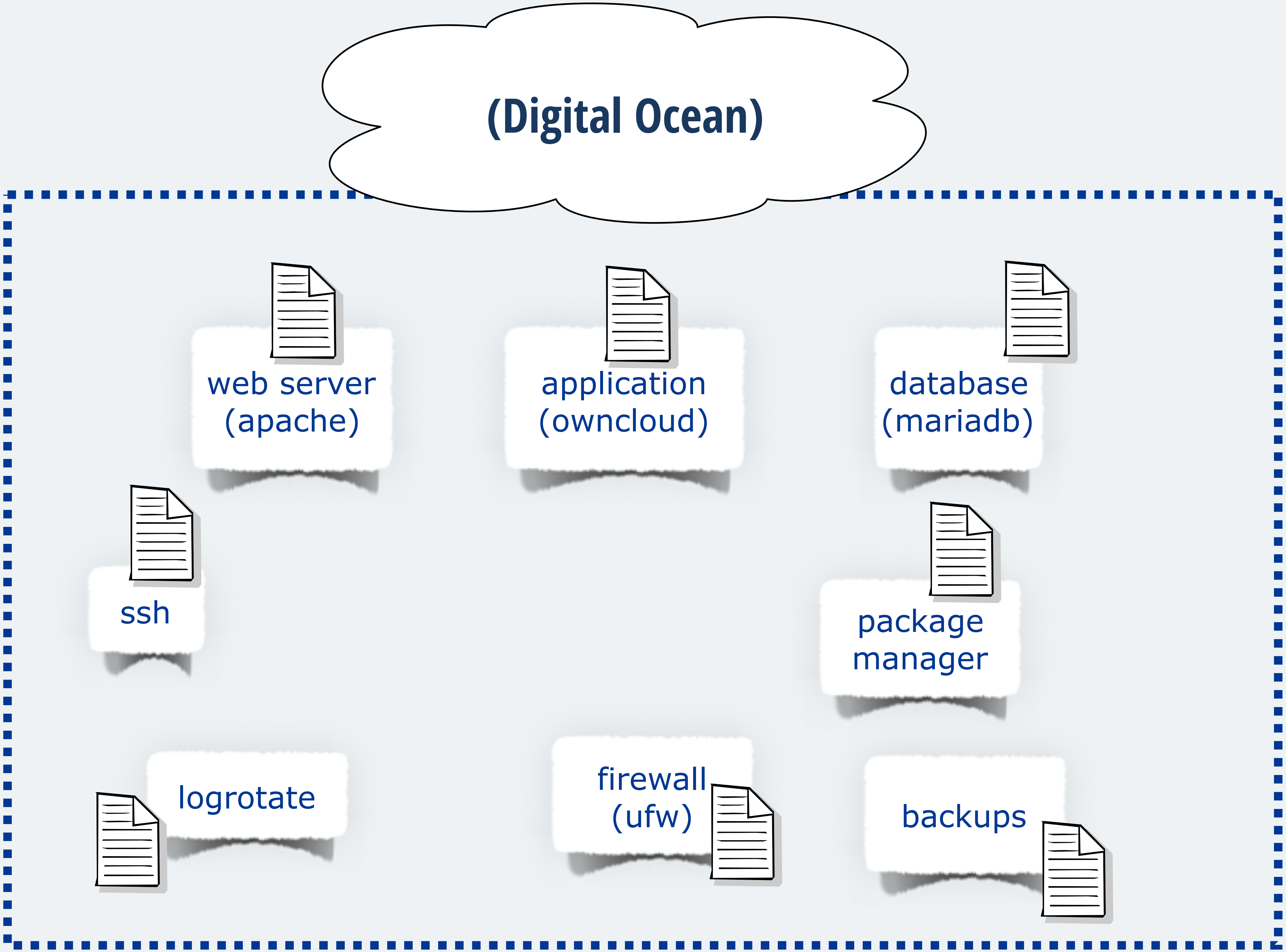
"``In his case study about Linux system engineering in air traffic control, Stefan Schimanski showed how scalable Puppet really is and how it can guarantee reliable mass deployment of the Linux-based, mission critical applications needed in air traffic control centers.''"

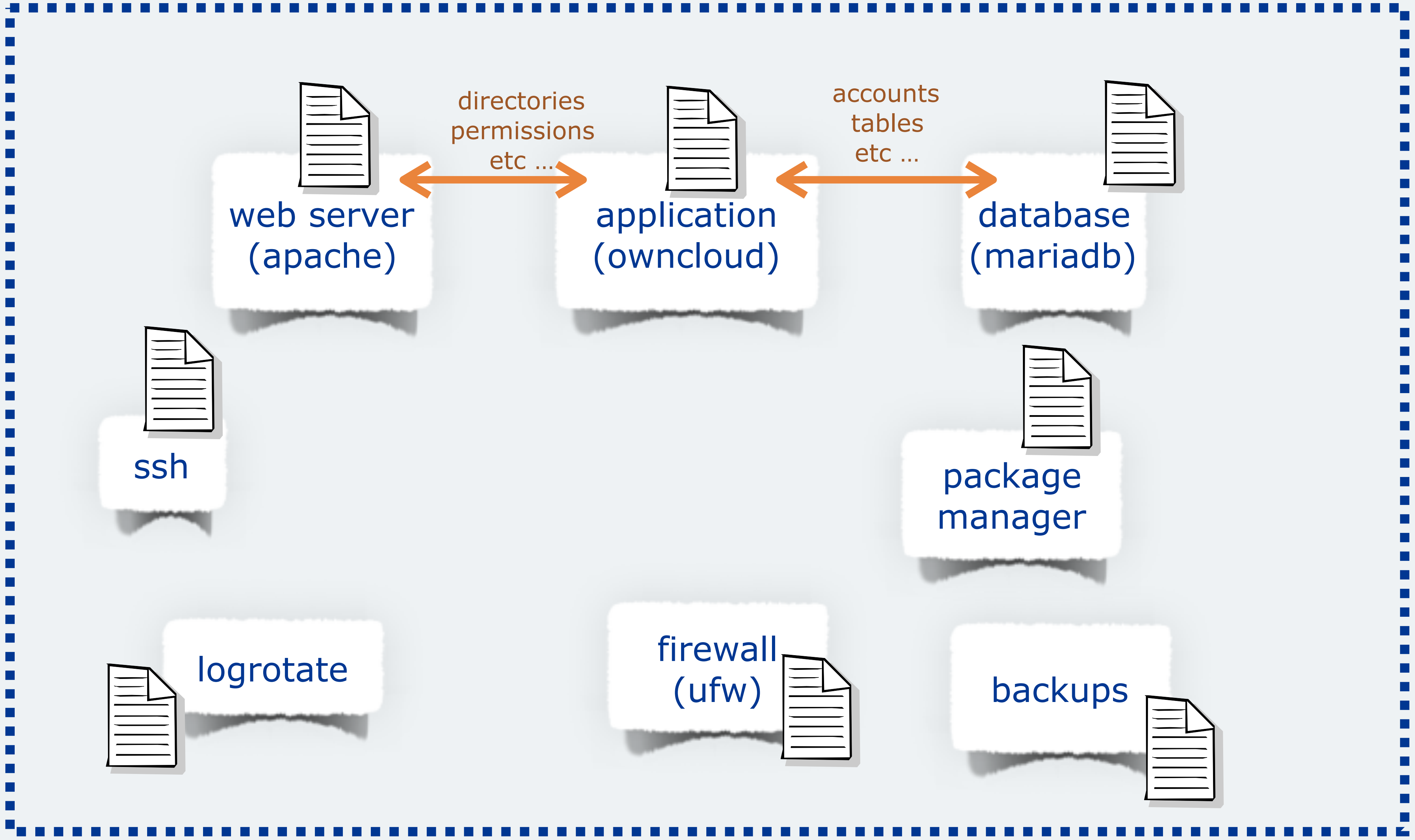
(Koen Vervloesem, FOSDEM Configuration Management Meeting Report, 2011)

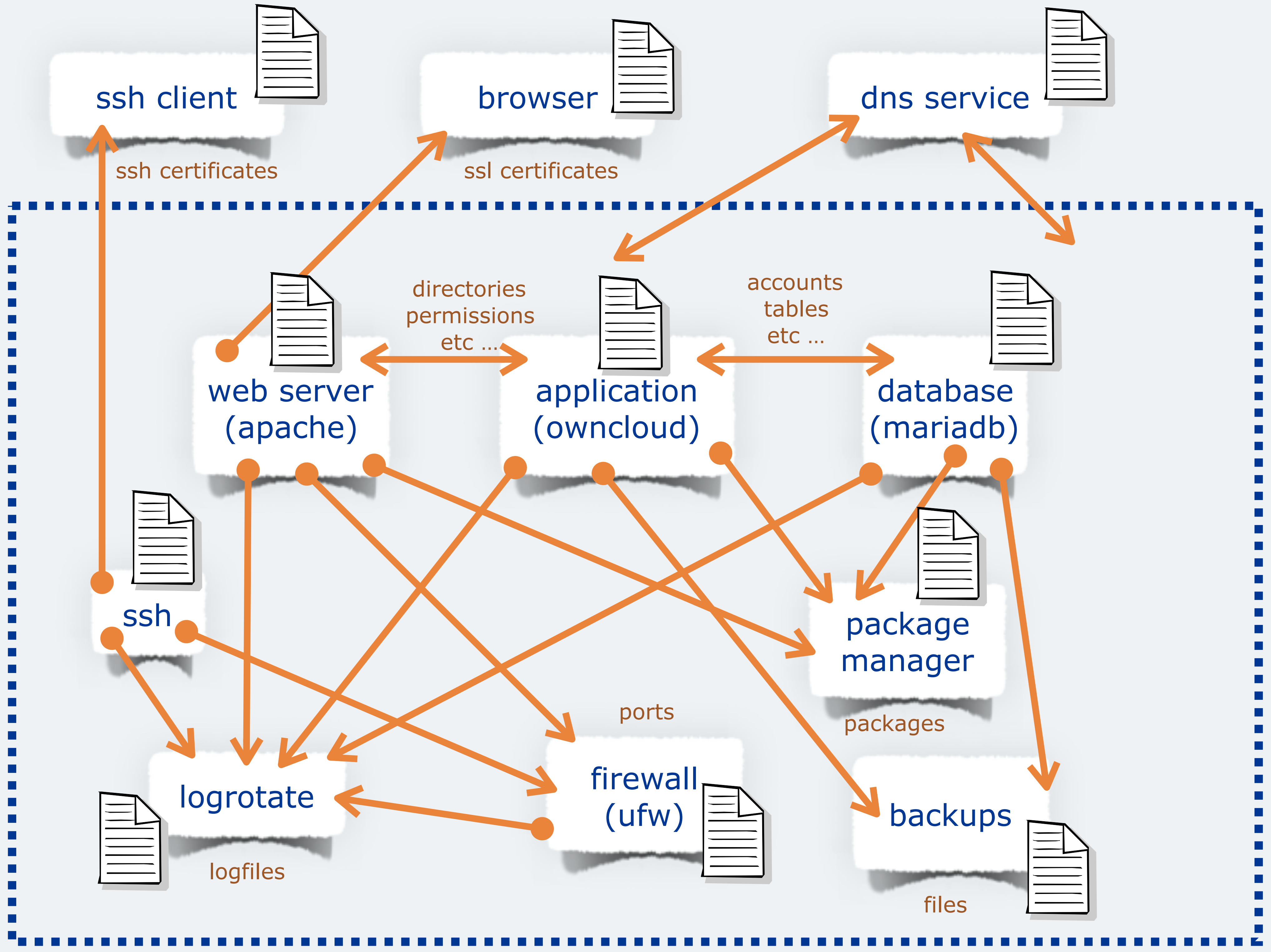
But the tools and languages are very informal

Unlike the languages used develop the applications


(Digital Ocean)
web server (apache)
application (owncloud)
database (mariadb)


(Digital Ocean)
web server (apache)
application (owncloud)
database (mariadb)
ssh
package manager
logrotate
firewall (ufw)
backups


web server (apache)
directories permissions etc …
application (owncloud)
accounts tables etc …
database (mariadb)
ssh
package manager
logrotate
firewall (ufw)
backups


ssh client
ssh certificates
browser
ssl certificates
dns service
directories
permissions
etc ...
accounts
tables
etc ...
web server
(apache)
application
(owncloud)
database
(mariadb)
ssh
package
manager
packages
ports
logrotate
logfiles
firewall
(ufw)
backups
files

I want a web server (in Ansible)

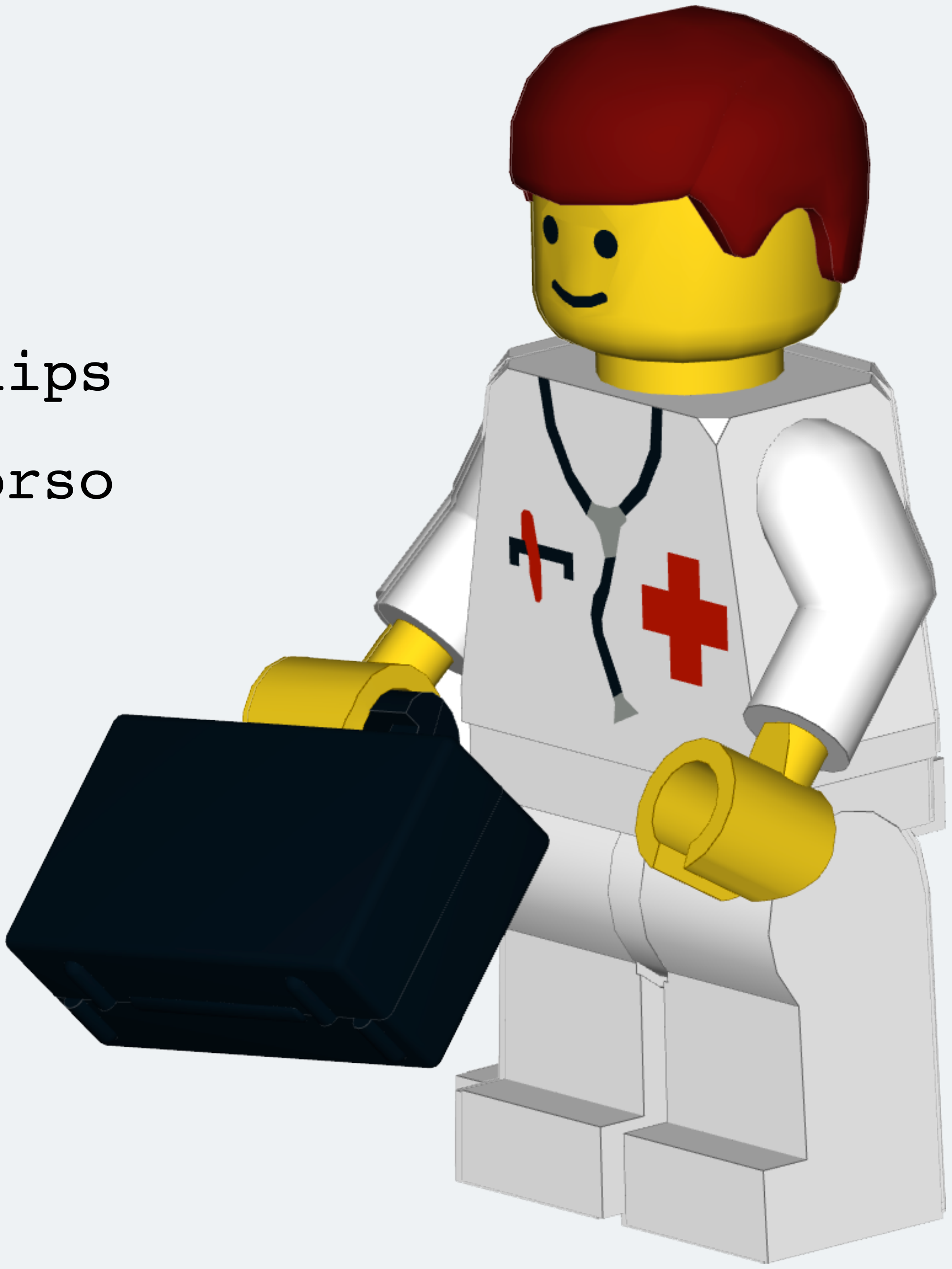
```
tasks:
 - name: ensure apache is at the latest version
   yum: name=httpd state=latest
 - name: write the apache config file
   template: src=/srv/httpd.j2 dest=/etc/httpd.conf
   notify:
   - restart apache
 - name: ensure apache is running
   service: name=httpd state=started enabled=yes
 handlers:
   - name: restart apache
     service: name=httpd state=restarted
```

http://docs.ansible.com/ansible/playbooks_intro.html

https://www.flickr.com/photos/foilman/15844421582

Imperative configuration

```
figure = hips
add legs under hips
add torso to top of hips
add head to top of torso
add arms to torso
add hair to head
add hands to arms
add bag to hand
```



Imperative configuration

Evolved from "scripting" the original manual procedures

- There is no explicit specification of the required state
  - this is simply a result of the deployment process
- The workflow requires a fixed (set of) starting state(s)
  - if the system starts in an unexpected state, there may be no appropriate workflow
- It is non-trivial to prove that the workflow produces a final state which meets the requirements
  - the "requirements" may not even be explicit
  - the workflow may not even terminate!
- The ordering implied by the workflow may be over-constrained

But ...

- This is still popular because system administrators can use familiar procedures and imperative scripting languages

Declarative configuration

```
figure: {
 head: {
  face: "male"
  hair: {
   style: "short"
   colour: “brown"
  }
  hat: none
 }
 clothing: {
  ...
 }
}
```

Declarative configuration

Specifies the desired state - not the workflow

- The specification is independent of the deployment
  - the deployment can be verified independently
- It is independent of the starting state
- There is an explicit specification of the “desired” state against which we can compare the “actual” state

Of course ...

- We do need to compute a workflow between the actual and desired states to implement the deployment
- I will not be discussing the deployment issues, but ..
  - many production tools assume that the ordering of this workflow is not important
  - however, we can use automated planning techniques to compute this from declarative constraints on the state
    - both final and intermediate states

Configuration languages

Declarative configuration languages ...

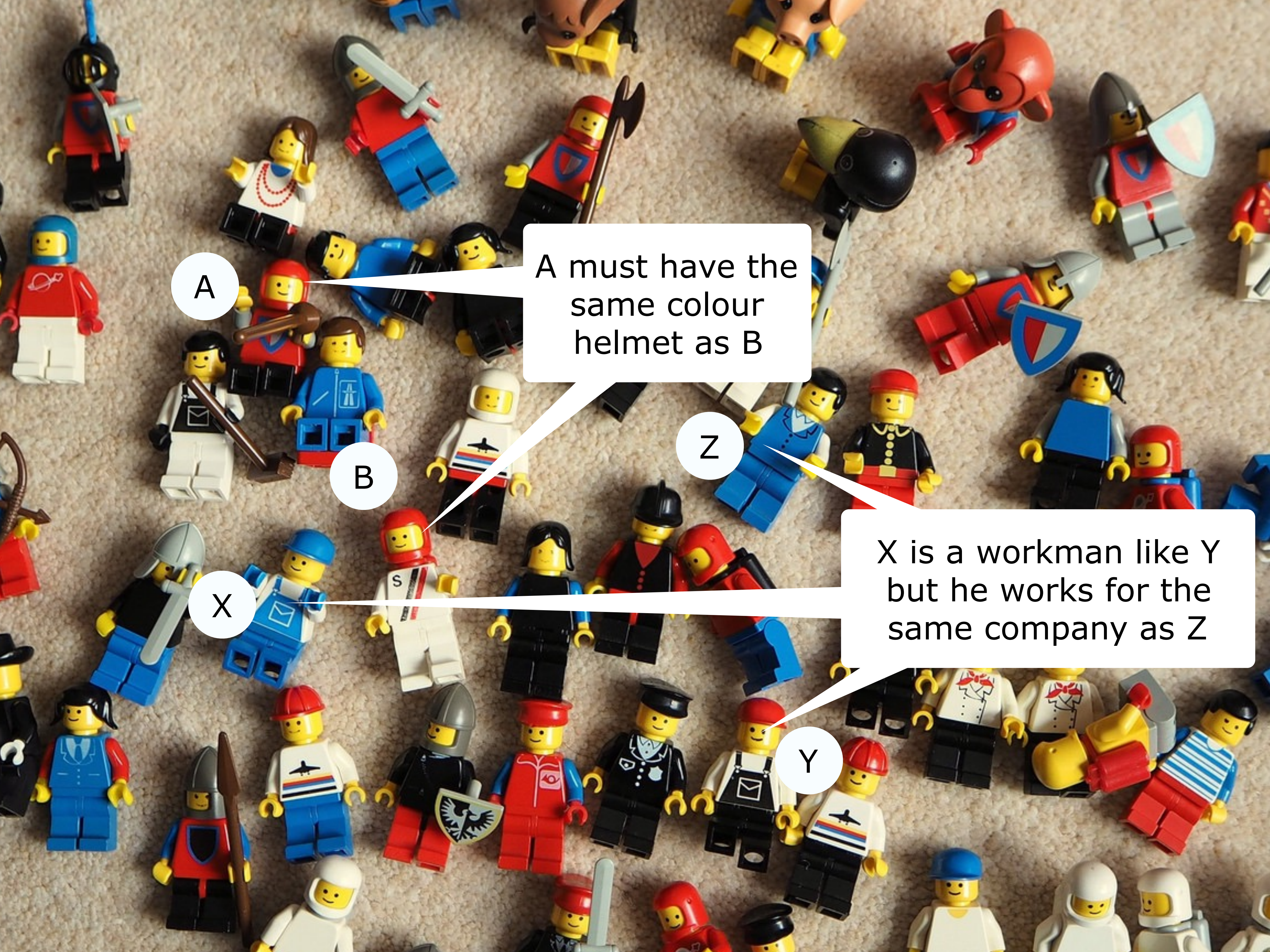
- Allow us to focus on the specification and structure of the desired configuration
- Without considering the deployment workflow

These are not programming languages

- They describe configurations rather than computations
- (Arbitrary) computation is not a core feature

So why do we need more than (say) JSON ?

- Many people are involved in specifying different, overlapping “aspects” of the configuration
  - we want them to be able to do this independently
  - these change rapidly
- (And other reasons …)


A
A must have the same colour helmet as B
B
Z
X
X is a workman like Y but he works for the same company as Z
Y

An example in SmartFrog

```
sfConfig extends {
  s1 extends Machine, {
    web extends Service
  }
  s2 extends s1, {
    web:running false;
  }
  pc1 extends Machine;
  pc2 extends Machine, {
    service s1:web;
  }
}
```

```
Machine extends {
  dns "ns.foo";
}

Service extends {
  running true;
  port 80;
}
```

Typical languages

SmartFrog

- Is not widely used outside of HP, but ..
- It is a small and well-defined language
- We created a formal semantics
- It can model arbitrary hierarchies

Other Languages ...

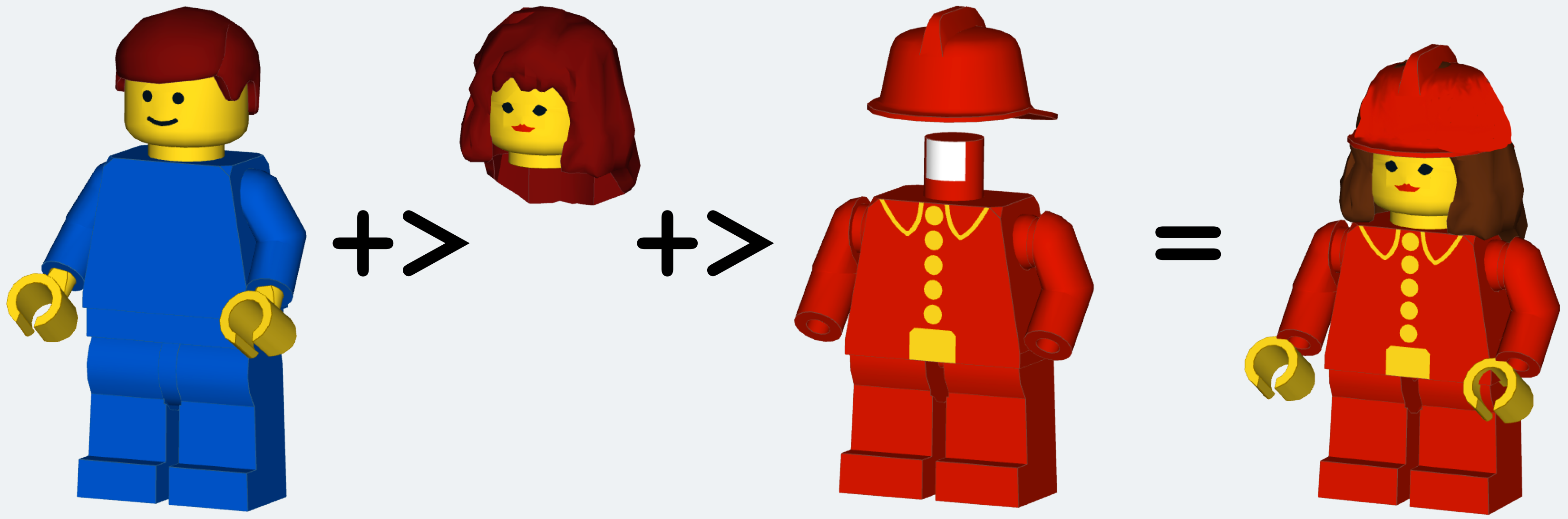
- LCFG
- Puppet
- Google cloud platform

Composition & Specialisation

With a bit about references ...

Specialisation (instance inheritance)

This is a typical operation ...



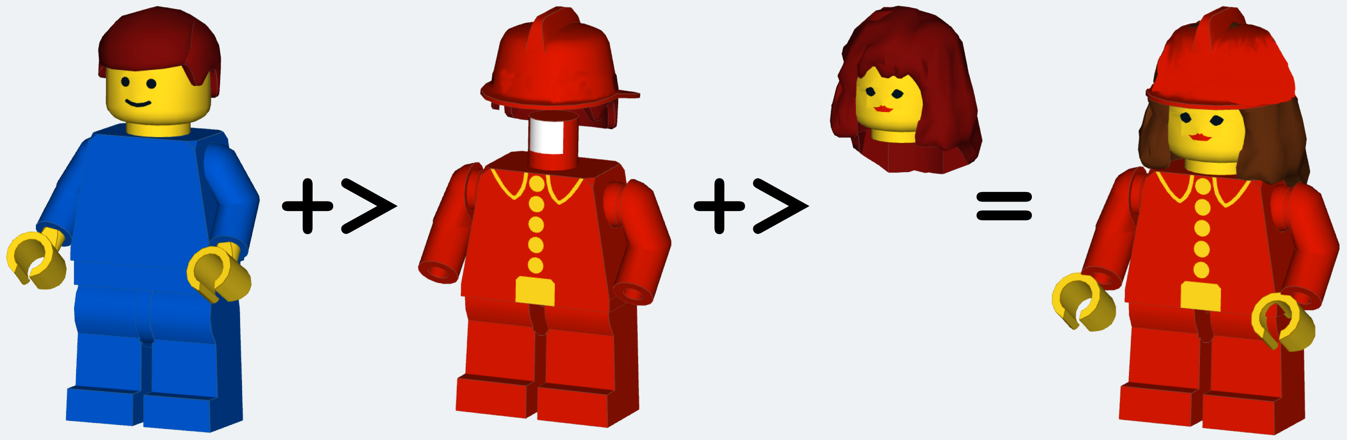
Or ...

"I want a Redhat Linux machine running Apache and Wordpress"

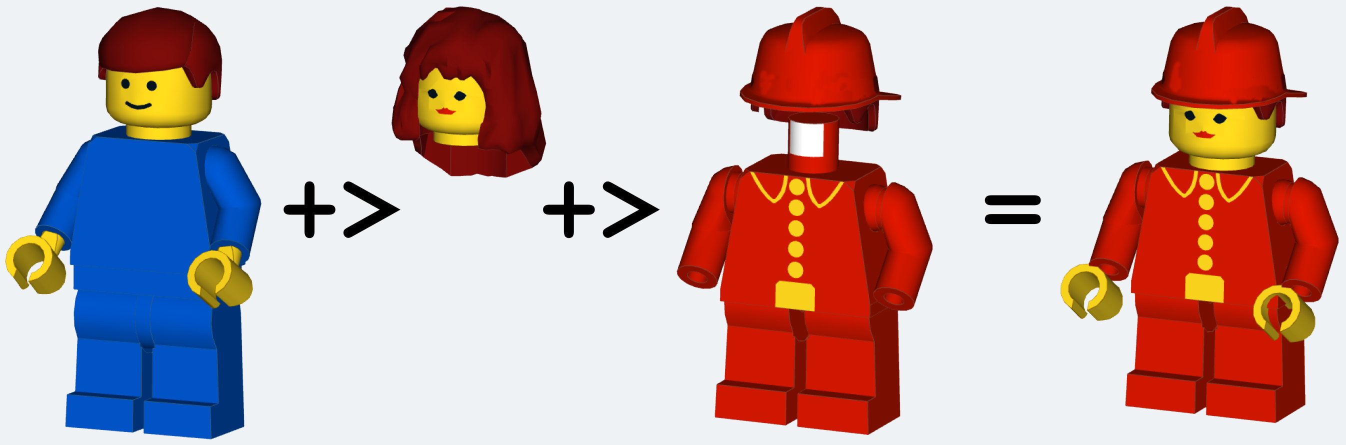
This works fine if the "aspects" are disjoint

Conflicts

"Hair will not extend beyond the bottom of the earlobe"
International Association of Women in Fire and Emergency Services
http://bit.ly/1Jt0Mz5



A female firefighter ?

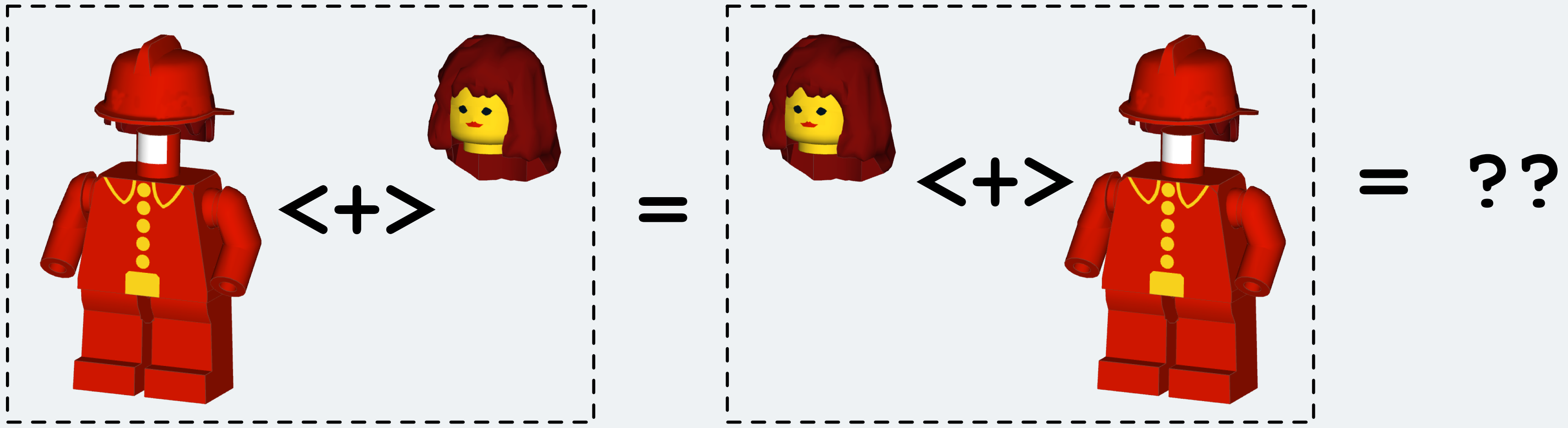


Or a firefighting female ?

Commutative composition

The "user" is forced to make this decision

- But they don't usually have the information to do this
- And neither order may be correct if there are multiple conflicts!



The user needs a commutative composition operation

- And the authors of the components need to specify how they should be composed

Resolving conflicts

What do we mean when we specify a value for a resource ?

- *"The value really must be 42".*
- *"I don't really care what the value is, but I can't leave it empty, so I'll give it the value 0".*
- *"36 would be a good value, but I don't care if someone else would rather have something different".*
- *"I think it should be 46, but if Jane thinks it should be different, then believe her".*
- *"The value must be between 100-200, but I can't specify a range, so I'll say 150".*

ConfSolve & constraints

“ConfSolve” supports very expressive constraints ...

- *I want 4 machines configured as database peers, which can be any machines except this one*:

```
    constraint
     forall (this in 1..4) (
      DatabaseServer_peer[this] != this
     );
```

These compose very well (commutative)

- They support all of the previous requirements & much more
- But, it requires thought to specify values which are not under- or over-constrained
- Understanding the consequences of the constraints is too difficult in most practical cases & the results can be unpredictable

L3

An experimental configuration language

- A small language with a clear, declarative semantics
- Features specifically design to support operations such as composition
- A balance between usability & expressiveness
- Not a programming language
- Output in JSON-like format which can easily be converted for deployment by other tools

Composition in L3

By default, conflicting values are composed as follows ...

- Blocks are composed recursively
- An Arbitrary (but deterministic) value is chosen for other conflicting values
  - this is more sensible than it sounds!

Values can be annotated with tags ...

- #final values take precedence over all others
  - composing multiple final values is an error
- #default values are only used if there are no non-default values
- Tags on blocks are inherited by the contained resources

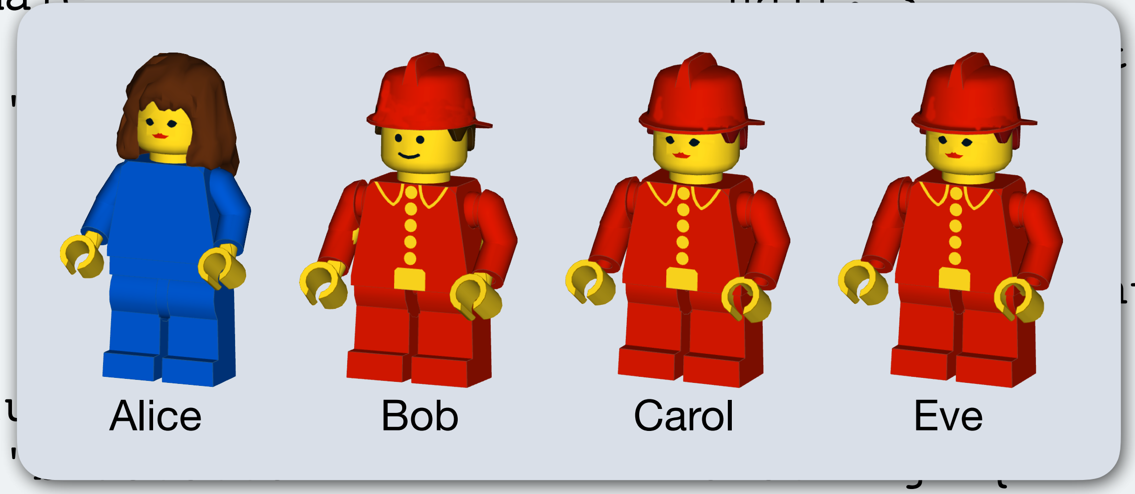
This is a very simple scheme (which is good!)

- We are trying to evaluate how well it works in practice
  - (it currently looks promising)

Composition example

```
figure: {
 head: {
  face: "male"
  hair: {
   style: '
   colour:
  }
 }
 clothing:
  top: "blu
  bottom: '
 }
} #default

female: $figure <+> {
 head: {
  face: "female" #final
  hair: {
   style: "long"
  }
 }
}
```

```
fireperson: $figure <+> {
 head: {
  hair: {
                               t"

                                 t"

  top: "firetop"
  bottom: "redbottom"
 }
} #final

alice:   $female
bob:     $fireperson
carol:   $female <+> ^fireperson
eve:     $fireperson <+> ^female
```



Arbitrary tags & constraints

Arbitrary tags can also be specified

```
a: { colour: "red" #aliceSays }
b: { colour: "blue" #bobSays }
```

And we can specify precedence between the tags

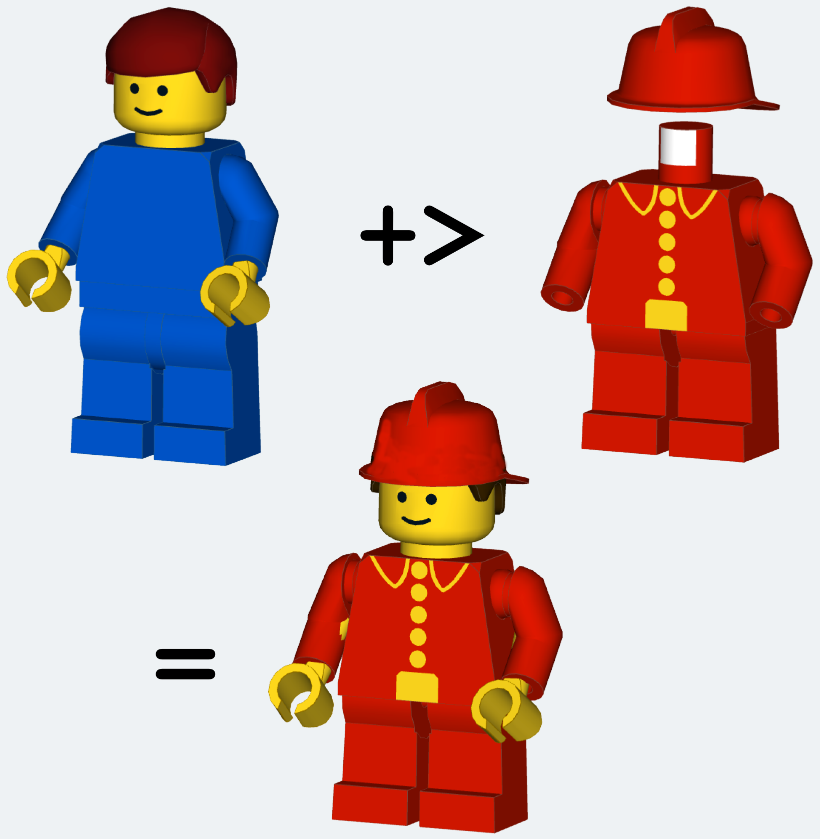
```
c: ( $a <+> $b ) #aliceSays >> #bobSays
d: ( $a <+> $b ) #aliceSays << #bobSays
```

This supports requirements such as ...

"I think it should be 46, but if Jane thinks it should be different, then believe her".
"Parameters specified at a departmental level should override those set at a corporate level".

Specialisation

```
fireperson:
$figure +> {
   head: {
     hat: "firehat"
   }
  }
  clothing: {
    top: "firetop"
    bottom: "redbottom"
  }
}
```



This can now be implemented in terms of composition ...

```
(X +> Y) ≡ (X #tag1 <+> Y #tag2) #tag1 << #tag2
```

References

References are used for ...

- "Cloning" prototypes (usually to be specialised)
- Ensuring consistency between related resources

```
bob:    $fireperson
carol:  $female <+> $fireperson
eve:    $fireperson <+> $female

service: { port: 45; … }
server:  { port: $service.port; … }
client:  { port: $service.port; … }
```

Absolute references are unambiguous

- But there are different possible semantics for relative references

Relative references

In this example, neither a purely “late” interpretation of the references, nor a purely “early” interpretation yields the “obviously” expected result:

```
service: {
  port: 25 #default
  client:  { port: ^^port, ... }
  server: { port: ^^port, ... }
}

myservice: ^service +> { port: 26 }

machineA: ^myservice.client +> { ... }
machineB: ^myservice.server +> { ... }
```

Disambiguating references

We could provide multiple types of reference

- LCFG has “early” and “late” references with different notations
- This is error-prone and very difficult for the user to get right

Humans are used to disambiguating references

"Divorcee and former air hostess Zsuzsi Starkloff talks on camera for the first time about her relationship with Prince William of Gloucester, the Queen's cousin and pageboy at her wedding”

(The Independent newspaper, Thursday 27th August 2015)

L3 currently has an experimental semantics

- Using composition to disambiguate multiple possible reference interpretations …

References in L3

```
service: {
  port: 25 #default
  client:  { port: ^^port, ... }
  server: { port: ^^port, ... }
}

myservice: ^service +> { port: 26 }

machineA: ^myservice.client +> { ... }
machineB: ^myservice.server +> { ... }
```

We compose all of the possible interpretations ...

`machineA.port = (25 #default) <+> 26 <+>` *`null`*

Other features

Some other current features …

- Partially-ordered collections
- No separate “variables” and “resources”
- Lazy evaluation
- Functions & operators (conditionals …)

Possible future work …

- Generating output for deployment by existing tools
- Provenance
- Higher-order functions (map)
- Visibility & modularisation
- Types
- Distributed specifications

But …

- We want to keep the semantics simple and avoid feature creep

Evaluation

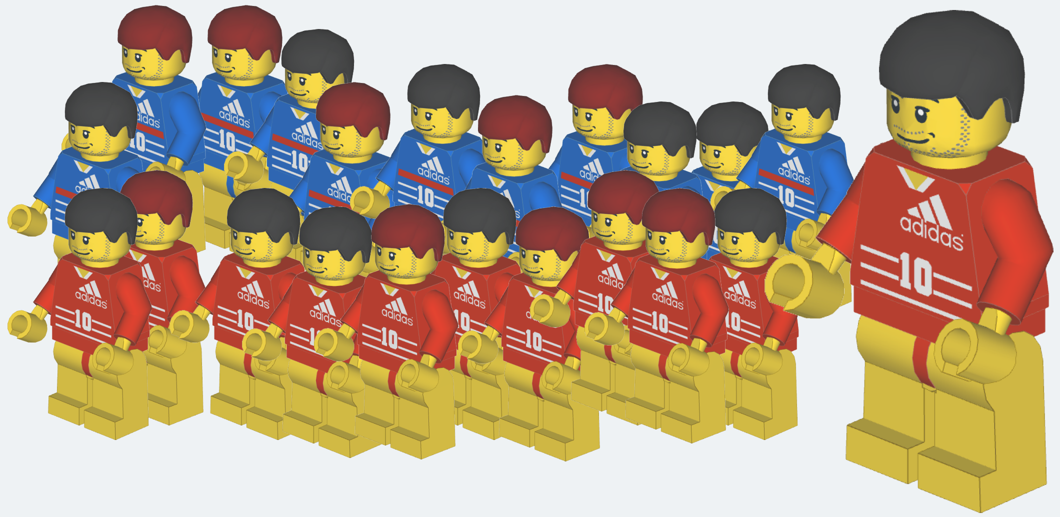
Usability is not easy to evaluate ...

- How easy is this to use in practice?
  - for an experienced administrator
  - for a novice?
- Asking system administrators about the language often involves them adopting an unfamiliar paradigm which takes time to absorb
- We have been manually translating configurations from other languages to identify issues and new paradigms

```
dave: ^figure +> {
  name: "dave",
  head: { hair: { colour: "black" }}}

bob: ...

team: {
  colour: "blue"
  player: ^^figure +> {
    clothing: {
      top: "shirt"
      colour: ^^^colour
    }
  }
  leftWinger: ^player +> ^^dave
  centreForward: ^player +> ^^bob
  ...
}

awayTeam: ^team +> { colour: "red" }
```

Paul Anderson

dcspaul@ed.ac.uk



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

(*publications, talks etc …*)