PAUL ANDERSON

dcspaul@inf.ed.ac.uk

Alva Couch

couch@cs.tufts.edu

What is This Thing Called "System Configuration"?

School of informatics

The University of Edinburgh

Tufts University
Computer Science

Overview

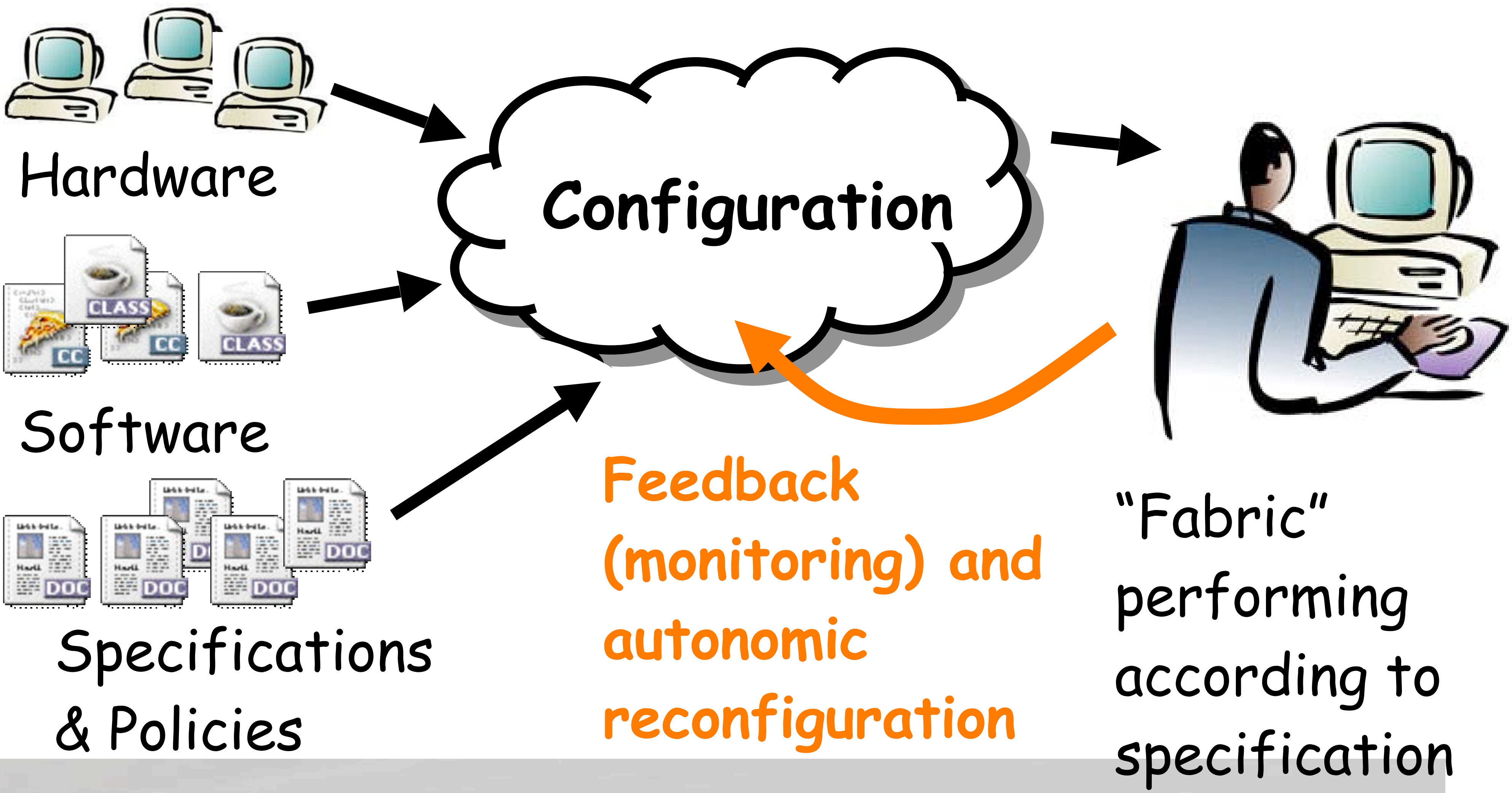
Paul says:

- The configuration problem
- Configuration specification
  - Types of specification
- Some language issues
  - Federated configurations
  - Autonomics
  - The role of theory
- Non-language issues
  - Decentralization, …
- Conclusions

If we have no clear way of stating the required configuration, then we can't create a tool to implement it!

The configuration problem

Paul says:



The configuration problem

Paul says:

- Starting with:
  - Several hundred new PCs with empty disks
  - A Repository of all the necessary software packages
  - A specification of the required service
- Load the software and configure the machines to provide the required functionality
- This involves many internal services – DNS, LDAP, DHCP, NFS, NIS, SMTP, Web …
- Reconfigure the machines as the required service specification changes
- Reconfigure as the environment changes

Some context on configuration management

Alva says:

- "So easy that it's hard."
- "Set the same bits on every disk." – **NOT**.
- Very dynamic research community: annual LISA workshop, technical papers, etc.
- Perhaps *too* dynamic: "religious" controversies about tools; "**Infrastructure Mafia**".
- Goal in this talk: get beyond religion and tools; understand nature of good practice.
- Key question: what is "good enough practice?"

Good enough?

Alva says:

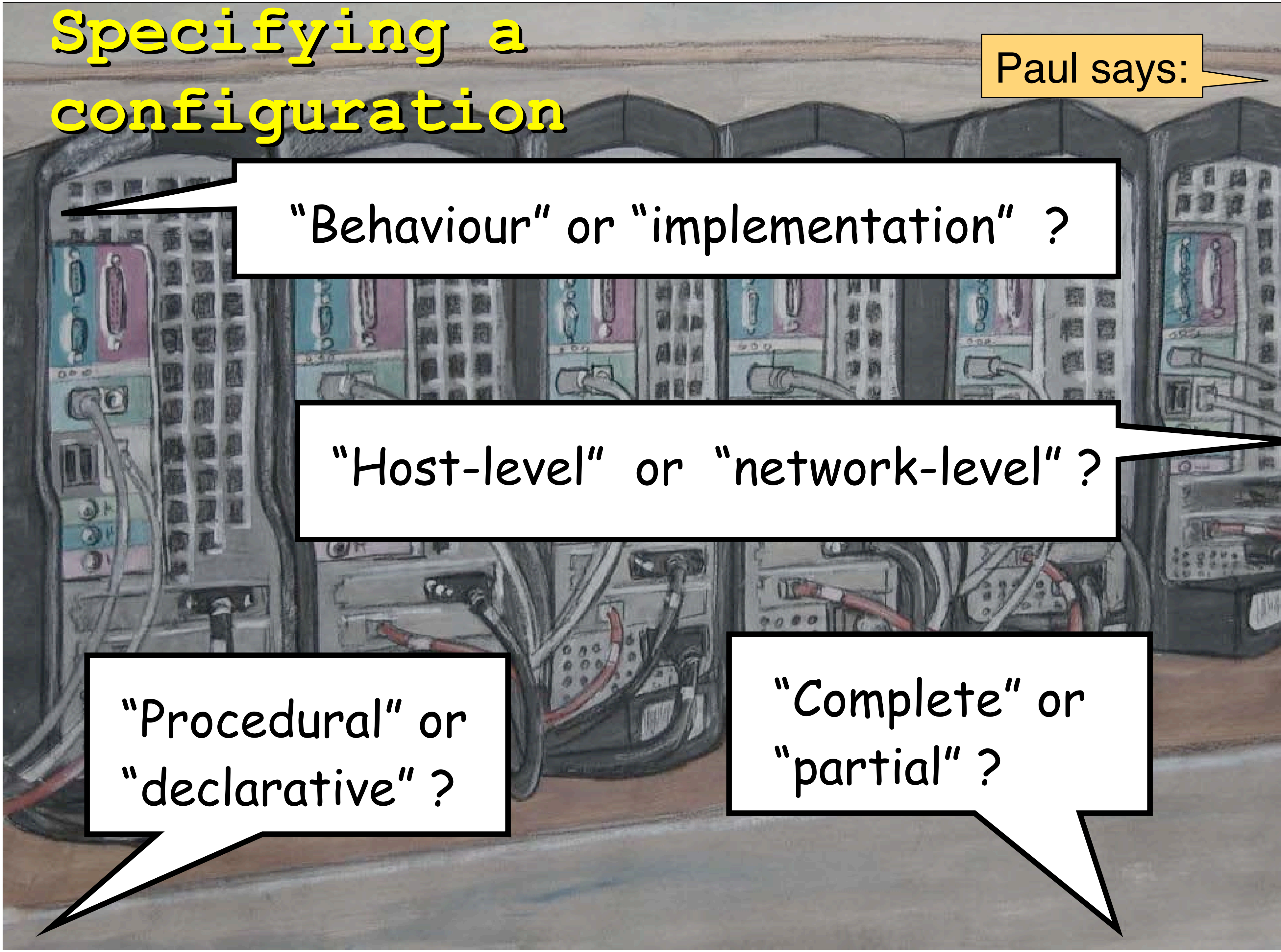
- What is “good enough?”
- Inside every hard computer science problem, there’s an easy one straining to get out.
- Key: “best” _ “good enough”.
- It’s “good enough” if its cost is reasonable given its value…

Are you already doing configuration management?

Alva says:

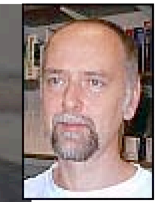
- Common occurrence: “closet” configuration management
  - Provide base services
  - Insure consistency
  - Cope with scale
  - Cope with change
  - Automate common algorithms
- Are you doing this and don’t realize it?
- All too common: SAs approach Configuration Management “through the back door”.

Specifying a configuration



"Behaviour" or "implementation"

Paul says:



- At the highest-level we want to be able to specify the desired behaviour of the system:
  - I want an SMTP service on port 25 of mail.foo.com
  - I want a response time of 1sec from my web service
- At present, this is normally translated manually into an implementation specification:
  - I want sendmail installed on some machine, etc …
- The correspondence between the behaviour and the implementation can only be validated by monitoring and feedback
  - Behaviour depends heavily on external events

Implementing behaviour

Paul says:

- All current tools really take implementation specifications
- The translation from the required behaviour is nearly always manual
  - *Although validation may be automatic*
- Automatic tools can use rules to implement limited variations of behaviour:
  - *Add an extra web server if the response is too slow*
- Could we have something more general?
  - *Would we want it ?*

"Host-level" or "network-level"

Paul says:

- Configuring services often requires cooperating configurations on many different hosts:
  - Configure host X as a web server
  - Configure the DNS to alias www.foo.com to X
  - Configure the firewall to pass http to host X
- A network-level specification allows us to model the service as an entity and automatically generate the host-level configuration data
  - There is no scope for mismatch between cooperating hosts parameters
- Note that network-level specifications are essential for autonomic fault-tolerance

"Procedural" or "declarative"

Paul says:

- "Procedural" configurations specify a set of actions to perform
- Procedural configurations do not capture the "intent" of the action and cannot be validated
  - If the environment changes, the same actions may have very different consequences
- "Declarative" configurations specify the desired final state
- Of course, action are required at some point to physically change a configuration
  - Tools can compute the required actions from declarative specifications of intent

A subtle distinction

Alva says:

- **Declarative**: implementation of directives might be ordered, but order is somehow “obvious” or “implied” by context.
- **Procedural**: specific ordering is the only way to get it to work; no “obvious” ordering other than the one given.
- Example: RPMs: Implicit order determined by dependencies _ list is **declarative**.
- Example: scripts: must keep lines in order _ script is **procedural**.

A declarative example

Paul says:

- **Declarative** (requirement)
  - Host X uses host M as the mail server
- **Non-declarative** (implementation)
  - "Run this script on host X to edit the sendmail.cf file"
- If we have <u>only</u> the implementation, then the intent is not clear
  - We cannot reason about the desired configuration
  - We cannot validate security policy, for example
  - And many other problems, such as order-sensitivity!

Why declarative?

Alva says:

- Make specifications simpler.
- Leave implementation to a tool.
- More portable.
- Allows flexible response.
- Easier to compose differing requirements.

Why procedural?

Alva says:

- Closer to normal manual configuration.
- Short learning curve for automating procedure.
- Intuitive mechanism for specifying what to do.
- Interoperable with many existing management tools (rpm, make, rdist, rsync, etc)

Evolution of management strategies

Alva says:

Unstructured changes	Manual commands
↓	
Scripting/documentation	Perl, bash
↓	
Script management systems	Pikt, isconf
↓	
Declarative recipes for one host	Cfengine,lcfg,bcfg2,psgconf,…
↓	
Declarative recipes for a fabric	Lcfg,pan,tivoli,…

A common myth dispelled

Alva says:

- Many people seem to believe that the choice of **tool** determines ease of configuration management.
- In fact, it’s the **practice** of using the tool that determines how well the tool works.
- Choice of tool makes little difference; **discipline** of use is everything.

"Complete" or "partial"

- A "**complete**" specification ties down all the parameters <u>about which we are interested</u>
- A "**partial**" specification assumes that some of the configuration parameters are controlled from elsewhere
  - *Sometimes, this is necessary – e.g. DHCP*
- There is a great danger with partial specifications of creating configurations with unpredictable values for important parameters
  - *If we don't specify it, then we have to be sure that someone else is managing it, or that we don't care!*

Perhaps better nomenclature: proscriptive or incremental

- **Proscriptive**: somehow **specify everything** about the configuration of a host or network.
- **Incremental**: specify some aspects of systems; leave others to other management processes.
- Example: build from bare metal: **proscriptive**
- Example: take over a legacy machine without a rebuild: **incremental**.

Common beginners' mistake: not being proscriptive enough

- Game of configuration management: make a lot of stations and/or servers cooperate and work similarly.
- Enemy of configuration management: "**latent preconditions**" differ among hosts, and are unmanaged by any process.
- Example: half the hosts don't contain an entry in /etc/hosts for foo.bar.com
  - OK if you don't need services from that host.
  - Bad when it somehow becomes your master fileserver!

Evolution of proscription

Alva says:

Ad-hoc: control whatever's convenient

Incremental: control a few things — "abuse of cfengine"

Bare metal: rebuild from scratch — "deterministic"

Can repeat a build
with exact same effect — "reproducible"

Can recover from
unforeseen developments. — "convergent"

Typical current practice

Paul says:

- Behavioural specifications are translated manually into implementations
  - *Apart from a few limited special cases*
- Most configuration specifications are host-level, rather than fabric-level
  - *The best tools are capable of some fabric-level specification*
- Complete configuration specifications are possible (and desirable!)
  - *But not used widely, due to the learning curve of the tools*
- Declarative (to some degree) specifications are common and widely accepted as a “good thing”

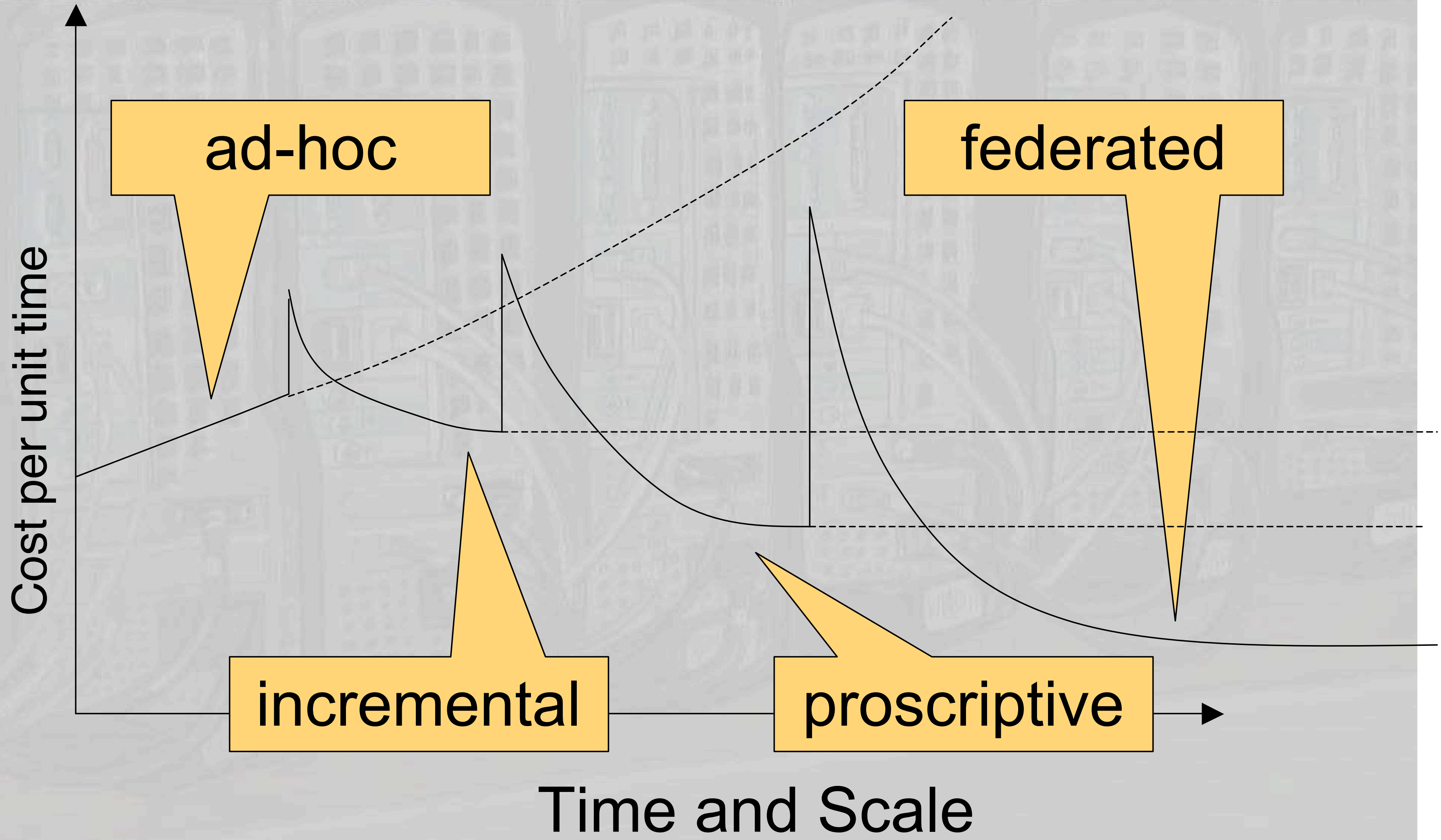
A little mystery

Alva says:

- Paul:
  - uses “fabric” management.
  - Declarative language.
  - Autonomic reconfiguration.
  - Rather complex learning curve.
- Alva:
  - uses “host” management.
  - RPM-based solution (non-declarative).
  - Scheduled wipe-and-rebuild.
  - Very simple tools.
- Why?

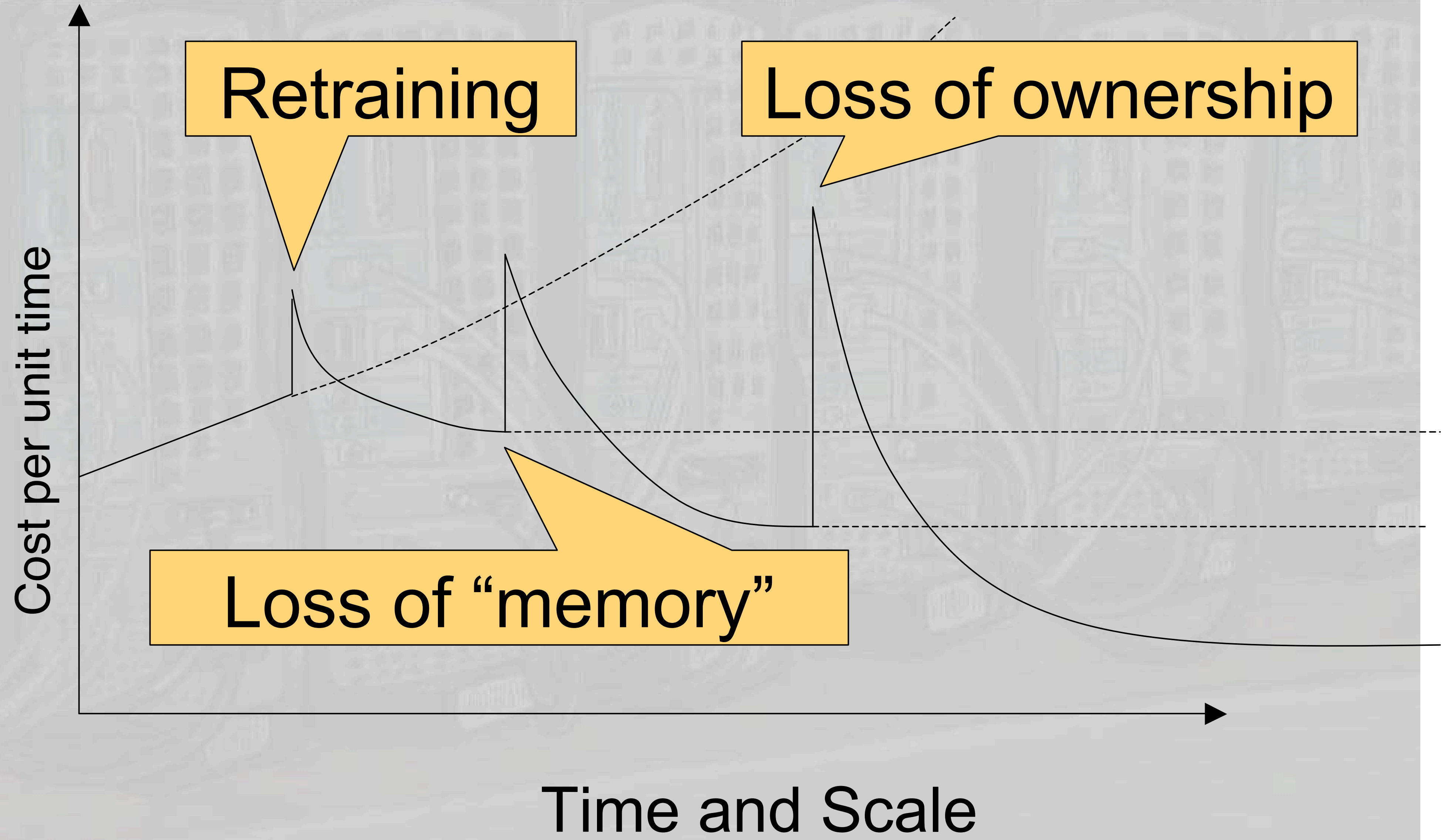
Backing into configuration management

Alva says:



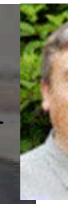
Slamming into cost and implementation barriers

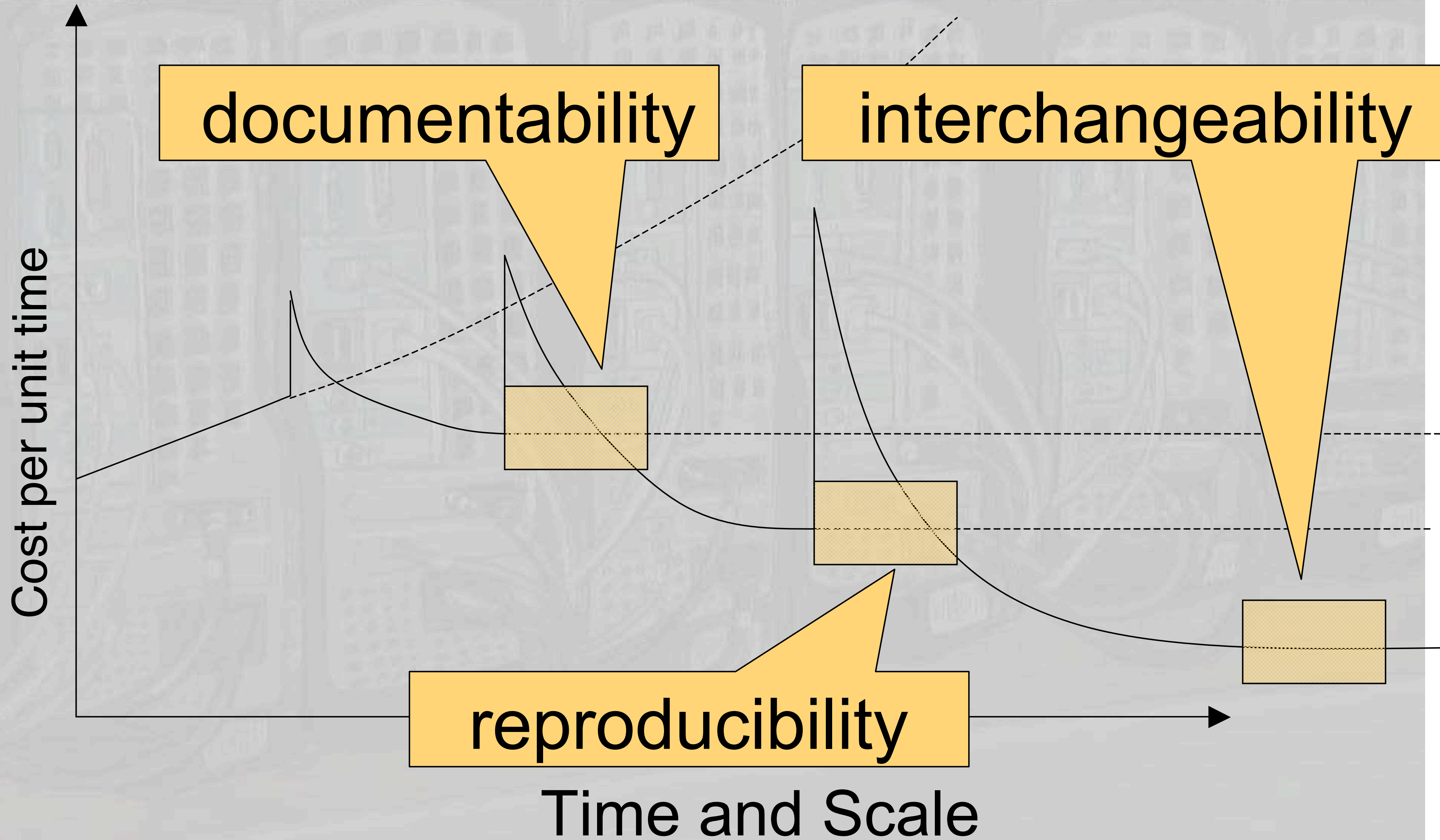
Alva says:



Backing into process maturity

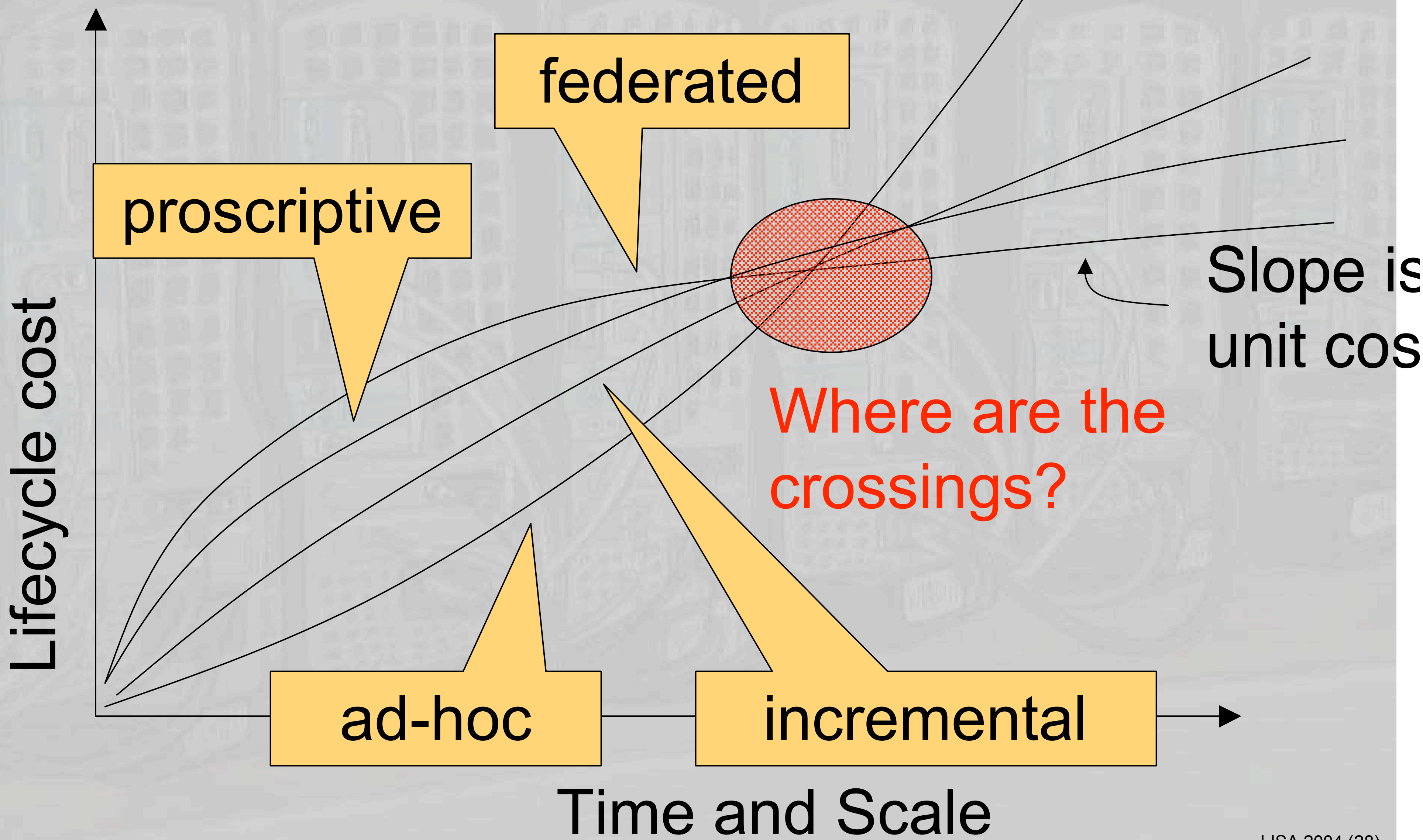
Alva says:





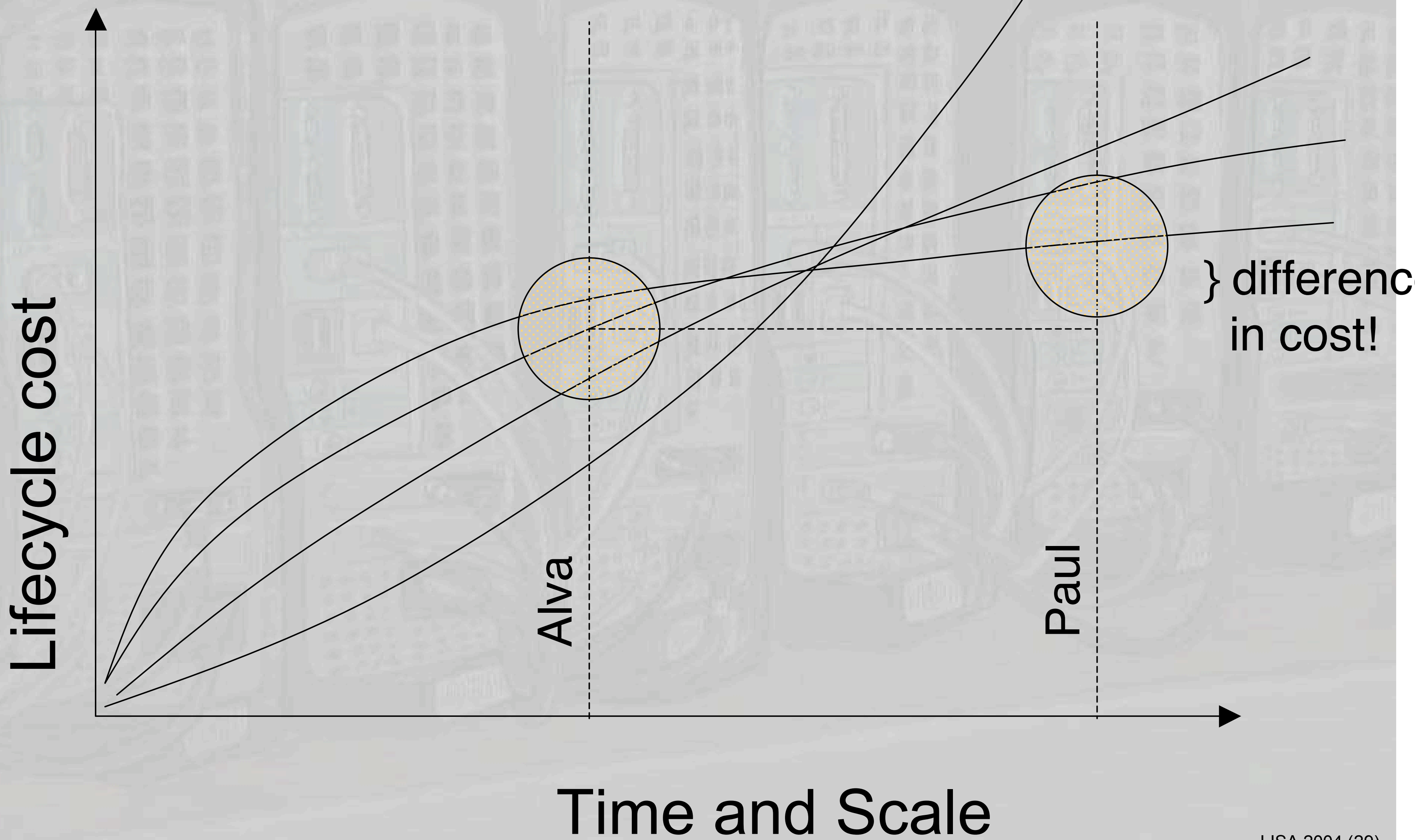
Lifecycle cost is a sum of unit costs

Alva says:



A little mystery solved

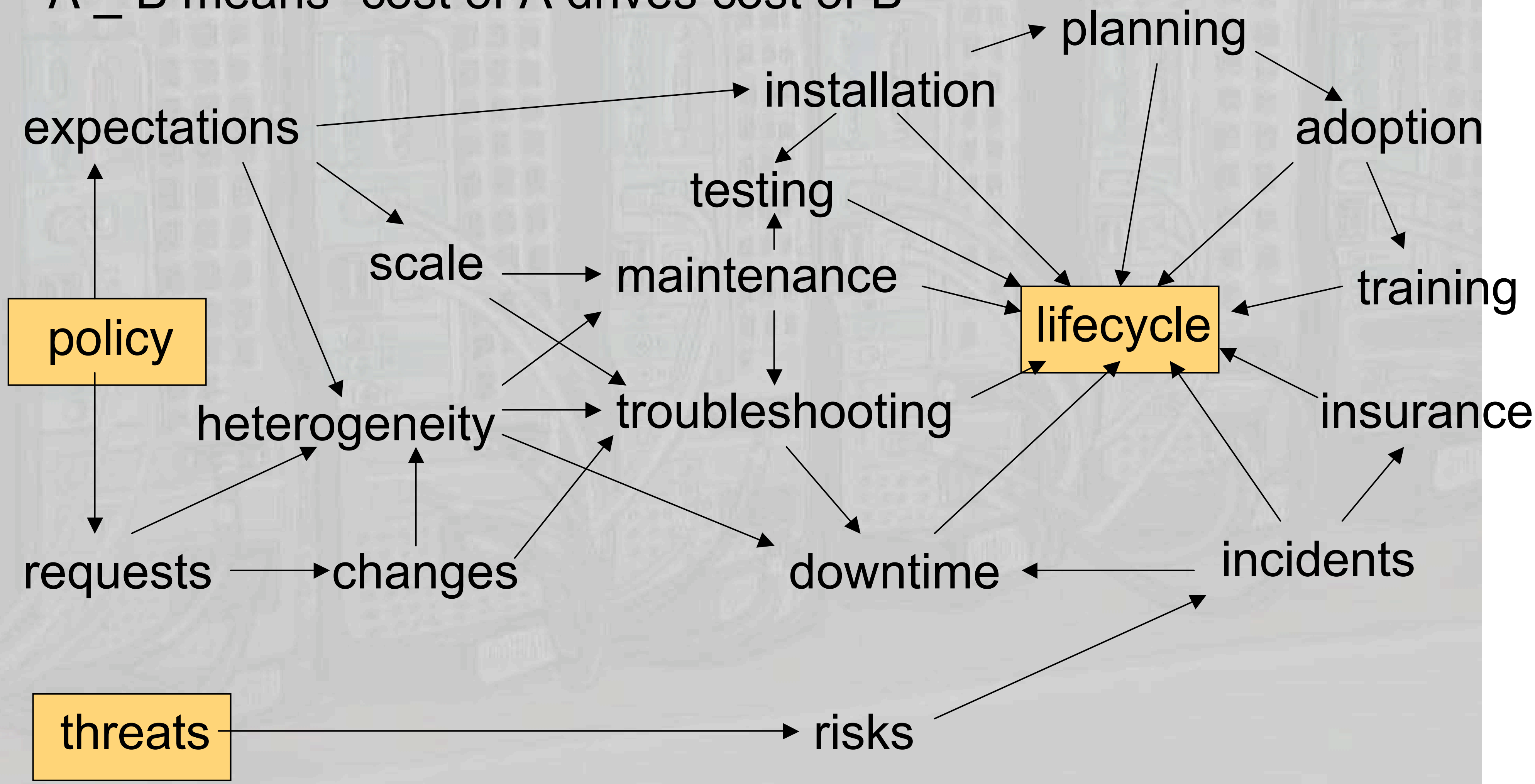
Alva says:



From whence come costs?

Alva says:

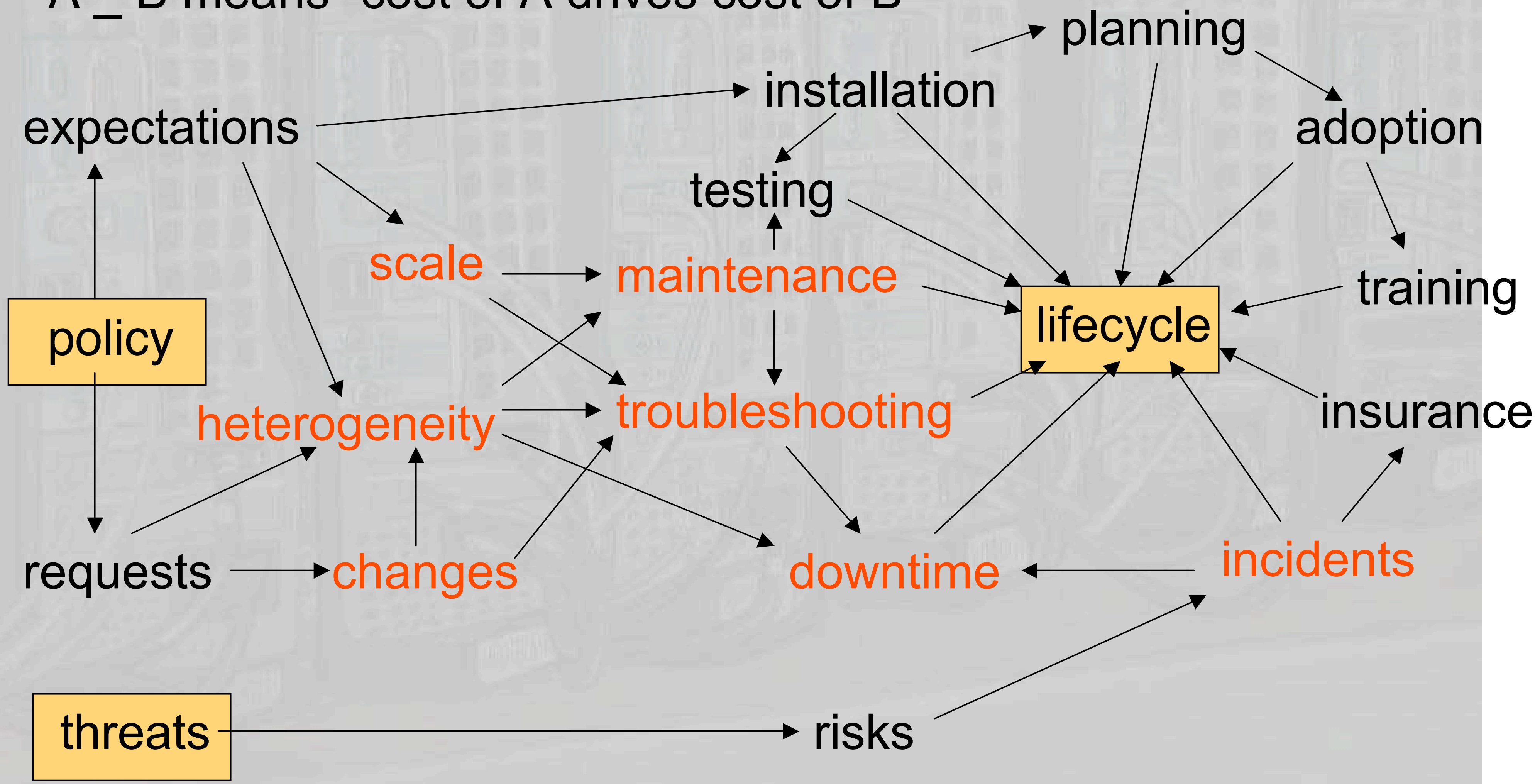
A _ B means “cost of A drives cost of B”



Manual management

Alva says:

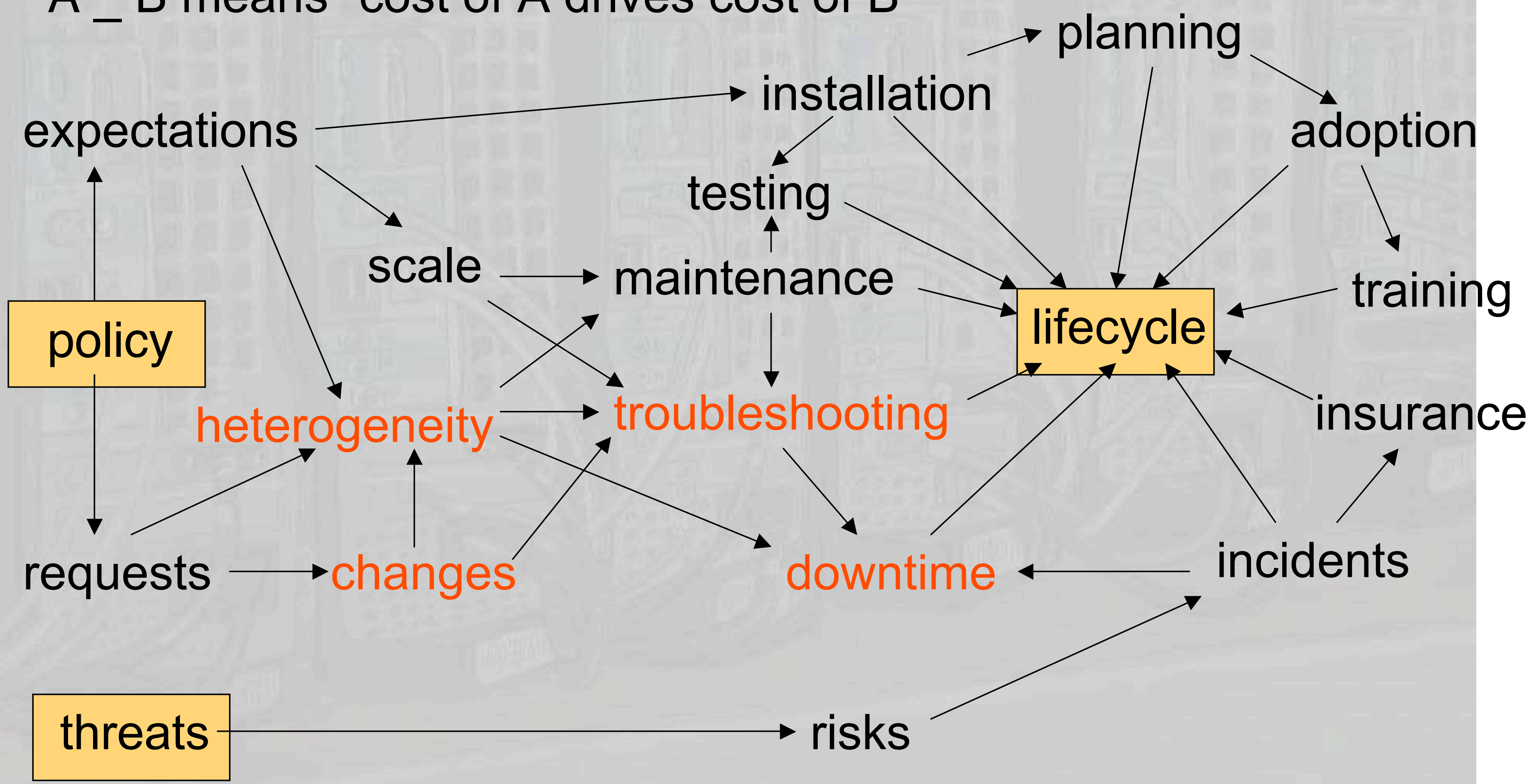
A _ B means “cost of A drives cost of B”



Incremental management

Alva says:

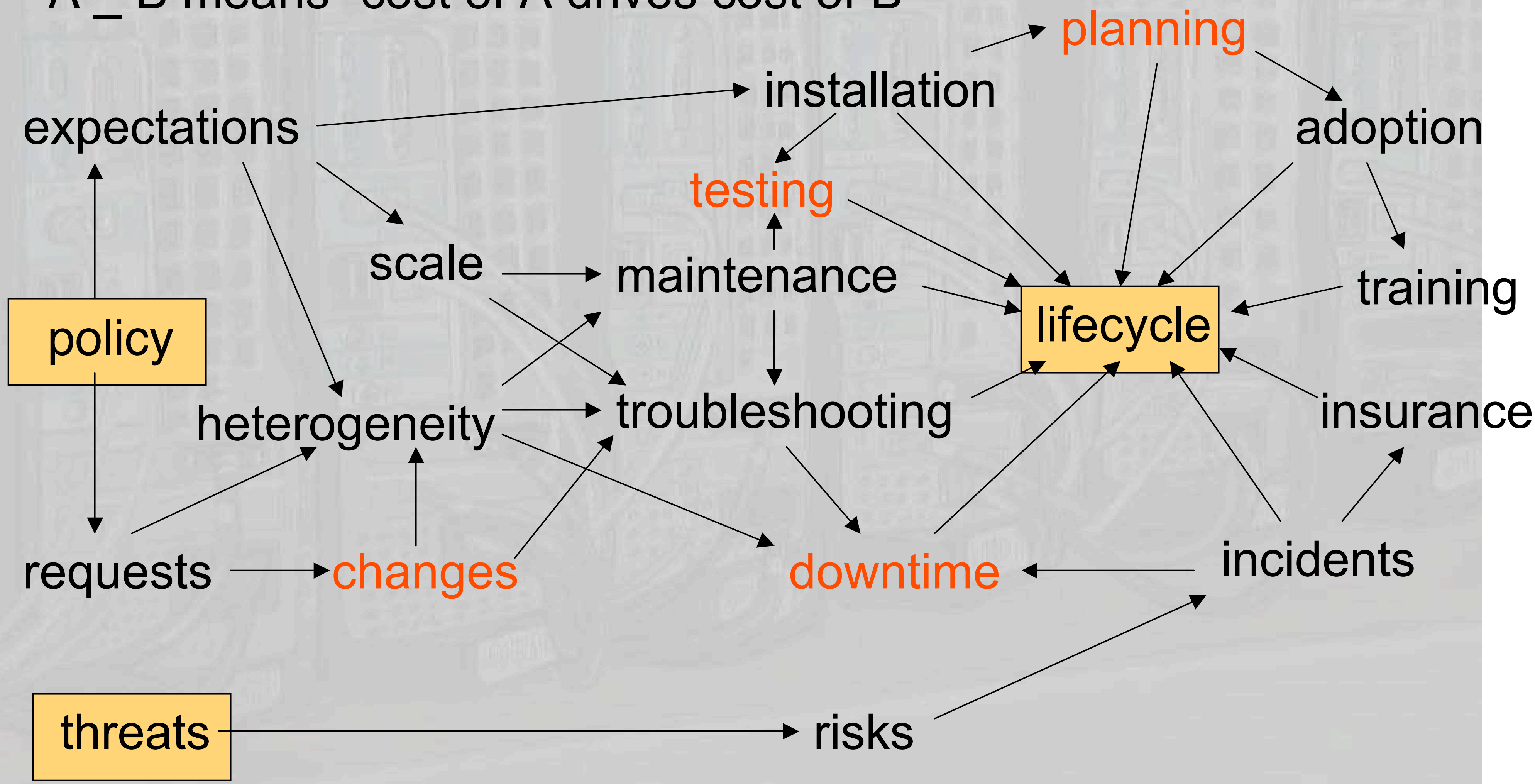
A _ B means “cost of A drives cost of B”



Proscriptive management

Alva says:

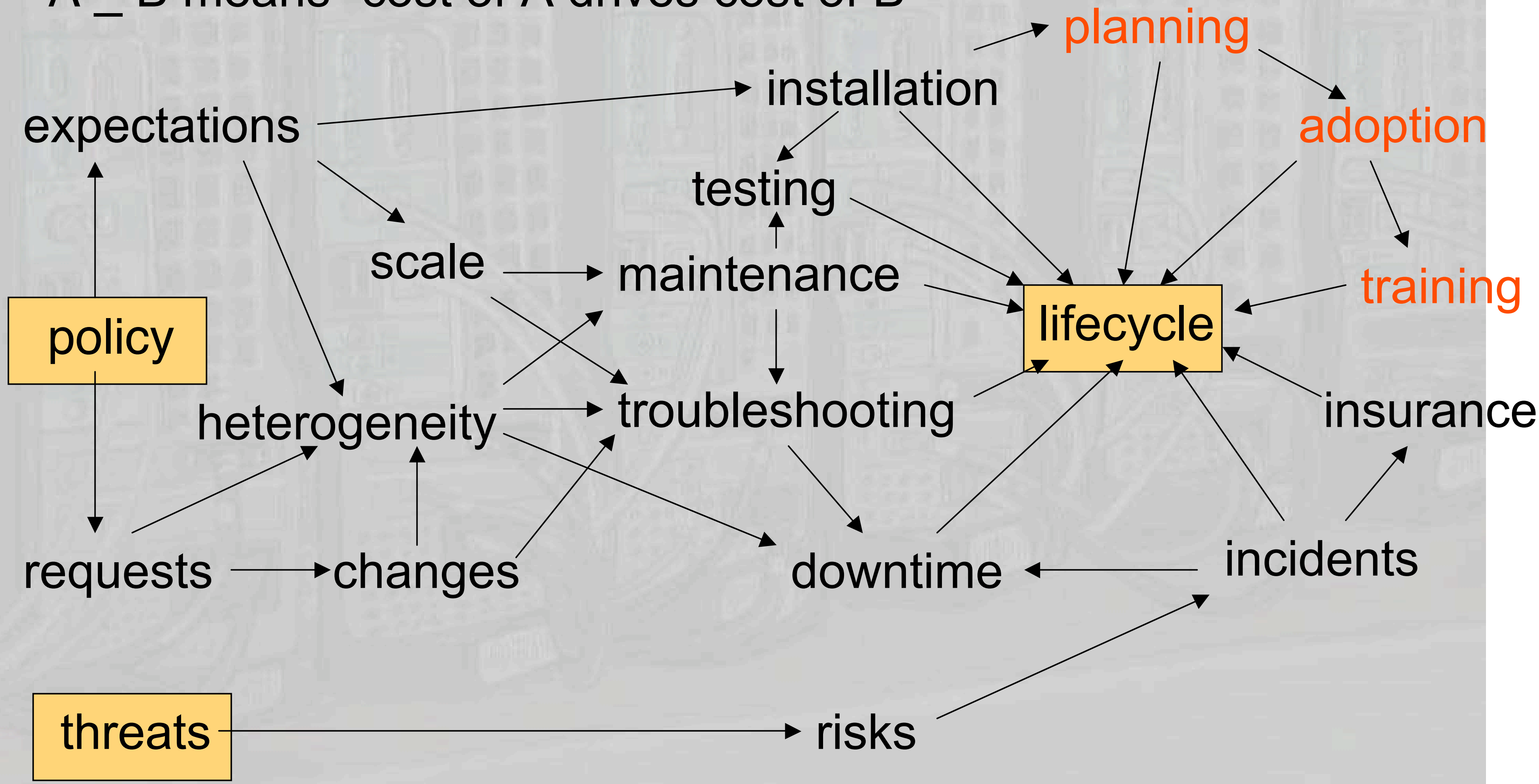
A _ B means “cost of A drives cost of B”



Federated management

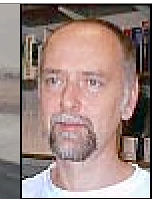
Alva says:

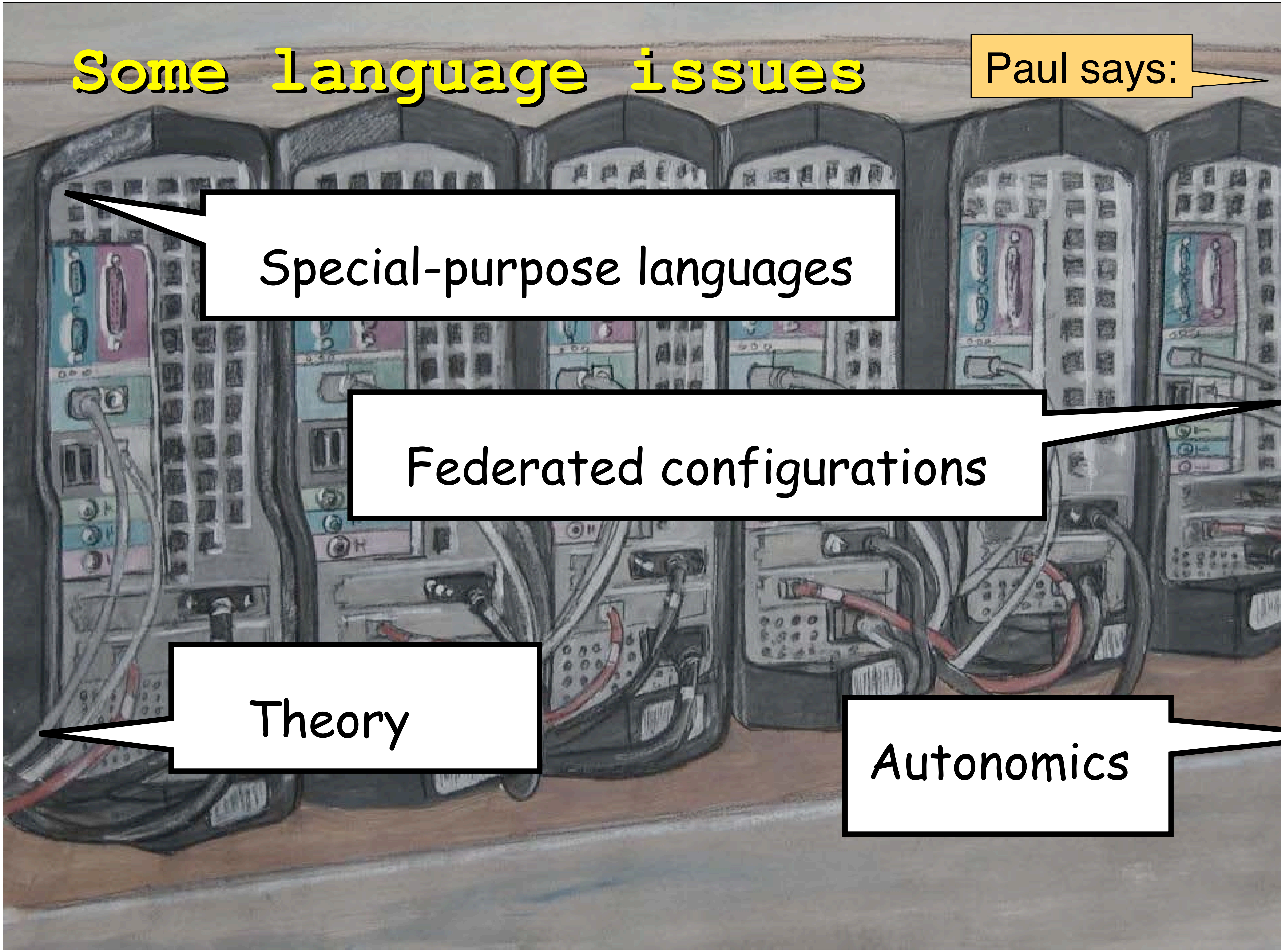
A _ B means “cost of A drives cost of B”



Some language issues

Paul says:





Configuration languages

- Configuration languages are essentially "data description" languages
  - I.e. declarative languages which determine the contents of the configuration files
- Configuration languages are different from programming languages
  - Which usually describe algorithms (as well as data)
- Structuring and managing the configuration information is one of the major current problems
  - We have 1000 hosts x 5000 parameters
- Some example problems follow ...

Federated configurations

- Existing configuration languages provide mechanisms such as hierarchical prototypes, or host “classes” for structuring the configuration data
- These are insufficient for modern “federated” installations where many people are responsible for different “aspects” of the same system
  - *Classes (aspects) overlap*
  - *Real, or apparent, conflicts arise frequently*
- Languages need better features to support this

Aspect composition

Paul says:

- The language forces explicit values to be specified:
- Aspect A
  - Use server Y
- Aspect B
  - Use server X
- This conflict is irreconcilable without human intervention because we don't know the intention

- The user really only wants to say …
- Aspect A
  - Use any server on my Ethernet segment
- Aspect B
  - Use one of the servers X,Y or Z
- These constraints can be satisfied to
  - Use server Y (assuming Y is on the right segment)

Autonomics

- To create systems from higher-level specifications, we need “autonomic” behaviour
  - Add more web servers if the response is slow
  - Configure a new DNS server if an existing one dies
- To do this in a declarative way, the language needs to support much “looser” specifications
  - I.e. The user should specify no more than is necessary, so that the system has freedom to assign other values
  - E.g. “I want two DHCP servers on each Ethernet segment”
- This is a similar requirement to the loose constraints required for aspect composition

A fault tolerance example

- Traditional "fault-tolerance" systems are usually based on event-action rules. For example:
- A declarative configuration:
  - Hosts X, Y and Z are my web servers
- An event-action rule:
  - If a web server goes down …
  - Then configure the backup server S as a web server
- Note that the procedural rule has broken the declarative nature of the original specification
  - This is no longer true

The role of theory

Paul says:

- Basic CS theory has helped to develop better programming languages which are easier to use and more likely to produce “correct” programs
- Corresponding theories for configuration languages are only in their infancy
  - What is a “configuration” ?
  - What is the effect of some fragment of configuration specification in some language?
  - We can look at the formal semantics of configuration languages
- The two previous problems suggest that constraint-based languages may be useful
  - But general-purpose constraint solvers are not viable at every level

Programming language development

Paul says:

- Unstructured programming is very hard to relate to the outcome of the program:
  - 1: blah blah
  - …
  - 2: if X then goto 4
  - …
  - 3: if Y then goto 1
  - …
- Most current configuration specifications are comparable to this level!

- The structured equivalent relates more closely to the declarative purpose of the code:
  - While (condition) do
  - …
  - End
- Providing that the loop terminates, we can be sure that the condition is false at the end

Non-language issues

Paul says:

- Decentralization
  - Centralized generation and distribution of configurations is becoming less feasible
  - Centralized control of the specification seems likely to become an unreasonable assumption
  - Decentralization complicates all the following issues
- Autonomics
  - Dealing with uncertainty
  - Monitoring and feedback
  - Recovery strategies
- Security and trust are major unsolved problems
- Planning and sequencing of complex, related configuration changes
- Lack of standards for configuration APIs and models
  - Is a problem for tool development and collaboration

Conclusions

Paul says:

- Increases in scale and complexity require more formal, higher-level approaches to system configuration
  - *Autonomics, federation, decentralization, …*
- Best current practice involves fabric-level, complete, declarative specifications
  - *Behavioural specifications cannot yet be translated automatically into implementations*
- For many people, this involves a significant change in practice, complicated because …
  - *Current tools involve steep learning curves*
  - *It must be possible to trust the tool to make significant decisions automatically*
  - *There are no widely useful standards*

Conclusions (cont’d)

Alva says:

- Concentrate on appropriate practice, not appropriate tools:
  - Avoid “closet” configuration management: face the problem and take control.
  - Be proscriptive rather than incremental.
  - Evolve toward declarative specification.
  - Evolve toward federated management.
  - Plan based upon lifecycle cost rather than unit cost.
- Consider the cost of *not* applying configuration management.

References

Paul says:

- Lssconf - An informal research collaboration
  - Annual LISA workshops & mailing list
  - **http://homepages.informatics.ed.ac.uk/group/lssconf/**
- The LCFG Project
  - The configuration tool developed in the School of Informatics at Edinburgh University
  - **http://www.lcfg.org**

PAUL ANDERSON
dcspaul@inf.ed.ac.uk

Alva Couch
couch@cs.tufts.edu

What is This Thing Called "System Configuration"?

School of informatics
THE UNIVERSITY OF EDINBURGH

Tufts University
Computer Science