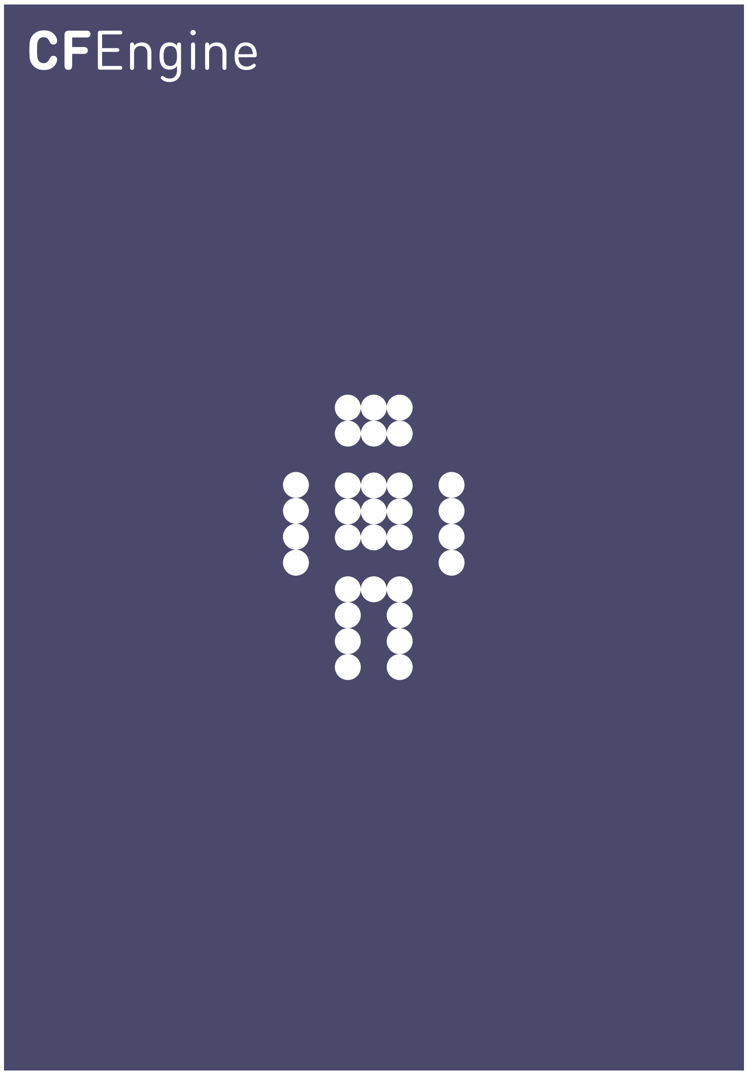


Agility in Infrastructure Engineering

A CFEngine Special Topics Handbook

CFEngine AS

Agility is a widely used term in today's fast moving IT industry. It reflects a need and a desire to respond quickly to changes in the environment. This document explains the management factors that affect speed, agility and scale in common scenarios, and what CFEngine can do to help you be agile.

Last updated December 2011

Table of Contents

1 Understanding Agility

We intuitively recognize agility as the capability to respond rapidly enough and flexibly enough to a difficult challenge. If we imagine an animal surviving in the wild, a climber on a rock-face or a wrestler engaged in combat, we identify the skills of *anticipation*, *speed* of response and the ability to *adapt* or bend without breaking to meet the challenges.

- Anticipate.
- Act.
- Adapt.

In infrastructure management, agility represents the need to handle changing demand for service, to repair an absence of service, and to improve and deploy new services in response to changes from users and market requirements. It is tied to economic, business or work-related imperatives by the need to maintain a competitive lead.

The compelling event that our system must respond to might represent danger, or merely a self-imposed deadline. In either case, there is generally a penalty associated with a lack of agility: a blow, a fall or a loss.

1.1 What make agility possible?

To understand agility, we have to understand time and the *capacity* for change. Agility is a relative concept: it's about adapting quickly enough, in the right context, with the right measure and in the right way. Below, we'll try to gain an *engineering* perspective on agility to see what enables it and what throttles it.

To respond to a challenge there are four stages that need attention:

- To comprehend the challenge.
- To solve the challenge.
- To respond to the challenge.
- To confirm or verify the response.

Each of these phases takes actual clock-time and requires a certain flexibility. Our goal is to keep these phases simple and therefore cheap for the long-term. Affording the time and flexibility needed is the key to being agile. Technology can help with this, if we adopt sound practices.

Intuitively, we understand agility to be related to our capacity to respond to a situation. Let's try to pin this idea down more precisely.

1.2 The capacity of a system

The capacity of a system is defined to be its maximum rate of change. Most often, this refers to speed of the system response to a single request[1].

In engineering, capacity is measured in *changes per second*, so it represents the maximum speed of a system within a single thread of activity[2].

1.3 Speed

Speed is the rate at which change takes place. For a configuration tool like CFEngine, speed can be measured either as

Clock speed

The actual elapsed wall-clock time-rate at which work gets done, including any breaks and pauses in the schedule.

This depends on how often checks are made, or the interval between them, e.g. in CFEngine, the default schedule is to verify promises every five minutes.

System speed

The average speed of the system when it is actually busy working on a problem, excluding breaks and pauses. For example, once CFEngine has been scheduled at the end of a five minute interval, it might take a few seconds to make necessary changes.

Engineers can try to define an engineering scale of agility as the ratio of available speed to required speed and ratio of number ways a system can be changed to the number of ways imperatives require us to change.

Agility is proportional to both how much speed we can muster compared to what is required, and the number of change-capabilities we possess, compared to what we need to meet a challenge. In other words: how well equipped are we? As engineers, we could write something like this:

```
                Available speed under control        Changes available
    Agility =~  -----------------------------  *  -----------------------
                      Required speed                 Changes Required
```

Although such a scale might be hard to measure and follow in practice, the definition makes simple engineering sense[3], and brings some insight into what we need to think about. What it suggests is that agility is a combination of *speed* and *precision*.

[1] Capacity is often loosely referred to as ‘bandwidth’ because of its connection to signal propagation in communication science, but this is not strictly correct, as bandwidth refers to parallel channels.

[2] For example, for a single coding frequency, the capacity of a communications channel is measured in bits per second, and the bandwidth is the number multiplied by the number of parallel frequencies.

[3] If available speed matches need, and we have the capability to make all required changes, then we can claim exactly 100% agility. If we have less than required, then we get a smaller number, and if we have excess speed or changeability then we can even claim a super-efficiency.

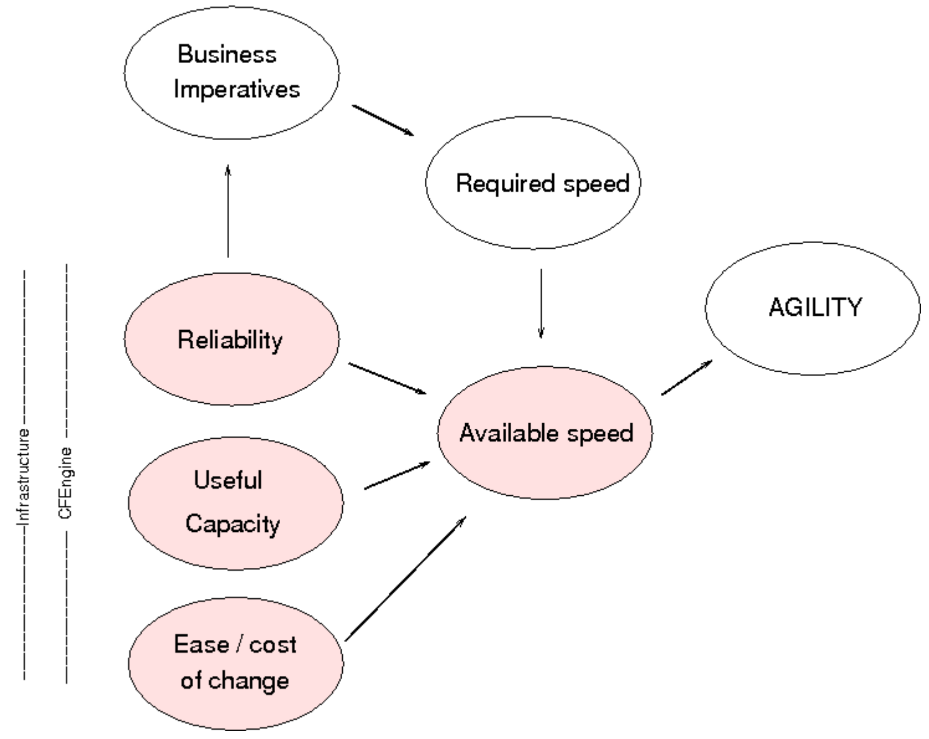
What is required speed? It is is the rate of change we have to be able to enact in order to achieve and maintain a state (keep a promise) that is aligned with our intent. This requires a dependence on technology and human processes.

The weakest link in any chain of dependencies limits the speed. The weakest link might be a human who doesn't understand what to do, or a broken wire or a misconfigured switch, so there are many possible failure modes for agility. An organization is an information rich *society* with complex interplays between Man and Machine; agility challenges us to think about these weakest links and try to bolster them with CFEngine's technology.

For example:

- If we think in terms of services, it is the Service Level you have to achieve in order to comply with a Service Level Agreement.
- If we think of a support ticket, it is the speed we have to work at in order to keep the impact of an unpredicted change within acceptable levels.

What we call 'acceptable' is a subjective judgement, i.e. a matter for policy to decide. So there are many uncertainties and relativisms in such definitions. It would be inconceivable to claim any kind of industry standard for these.



How agility depends on technology measures.

We can write some scaling laws for the dependencies of agility to see where the failure modes might arise.

The speed available to meet a challenge is (on average) the maximum speed we can reliably maintain over time divided by the number of challenges we have to share between.

```
                              Expected capacity * reliability
    Average available speed =~  ------------------------------
                                    Consumers or challenges
```

This expression says that the rate at which we get work done on average depends no only on how we share maximum capacity amongst a number of different consumers, clients, processes, etc, but also on how much of the time this capacity is fully available, perhaps because systems are down or unavailable.

The appearance of reliability in this expression therefore tells us that maintenance of the system, and anticipation of failure will play a key role in agility. Remarkably this is usually unexpected for most practitioners, and most of system planning goes into first time deployment, rather than maintaining operational state.

1.4 Precision

Acting quickly is not enough: we also need to be accurate in responding to change[4]. We need to be able to:

- Model the desired outcome accurately in terms of universal policy coordinates: Why, When, Where, What, How.
- Maximize the chance that the promised outcome will be achieved.

Precision is maximized when:

- Changes are 'precise', i.e. they can be made at a highly granular level, without disturbing areas that are not relevant (few side-effects).
- Policy is able to model or describe the desired state accurately, i.e. within the relevant area, the state is within acceptable tolerances.
- If any assumptions are hidden, they are describable in terms of the model, not determined by the limitations of the software[5].
- The agent executes the details of the model quickly and verifiably, in a partially unpredictable environment, i.e. it should be fault tolerant.
- If the model cannot be implemented, it is possible to determine why and decide whether the problem lies in an incorrect assumption or a flaw in the implementation.

[4] In the 20th century, science learned that there is no such thing as determinism – the idea that you can guarantee an outcome absolutely. If you still think in such terms, you will be quickly disappointed. The best we can accomplish is to maximize the likelihood of a predictable result, relative to the kind of environment in which we work.

[5] In some other configuration software, assumptions are hard-coded into the tools themselves, making the outcome undocumented.

CFEngine is a fault tolerant system – it continues to work on what it can even when some parts of its model don't work out as expected[6].

1.5 Comprehension

The next challenge is concerns a human limitation. One of the greatest challenges in any organization lies in comprehending the system.

Comprehensibility increases if something is predictable, or steady in its behaviour, but it decreases in proportion to the number of things we need to think about – which includes the many different contexts such as environments, or groups of machines with different purposes or profiles.

```
                        Predictability (Reliability)    Predictability
  Comprehensibility =~  ---------------------------- = ----------------
                                 Contexts                  Diversity
```

Our ability to comprehend behaviour depends on how predictable it is, i.e. how well it meets our expectations. For technology, we expect behaviour to be as close as possible on our intentions. CFEngine's maintenance of promises ensures that this is done with best possible effort and a rapid cycle of checking.

To keep the number of contexts to a minimum, CFEngine avoids mixing up *what* policy is being expressed with *how* the promises are kept. It uses a declarative language to separate the what from the how. This allows ordinary users to see what was intended without having to know the meaning of how, as was the case when scripting was used to configure systems.

1.6 Efficiency

Finally, if we think about the efficiency of a configuration, which is another way of estimating its simplicity, we are interested in how much work it takes to represent our intentions. There are two ways we can think about efficiency: the efficiency of the automated process and the human efficiency in deploying it.

If the technology has a high overhead, the cost of maintaining change is high and efficiency is low:

The efficiency of the technology decreases with the more resources it uses, e.g. like memory and CPU. Resources used to run the technology itself are pure overhead and take away from the real work of your system.

```
                                  Resources used
         Resource Efficiency =~  1 -  ---------------
                                  Total resources
```

[6] Other systems that claim to be deterministic simply stop with error messages. What is the correct behaviour? Clearly, this is a subjective choice. The important thing is that your system for change behaves in a predictable way.

It is a design goal of CFEngine to maintain minimal overhead in all situations. The second aspect of efficiency is how much planning or rule-making is needed to manage the relevant issues.

The efficiency of a model decreases when you put more effort into managing a certain number of things. If you can manage a large number of things with a few simple constraints, that is efficient.

```
                             Number of objects affected
      Model Efficiency =~  -------------------------------
                          Number of rules and constraints
```

General patterns play a role too in simplifying, because the reduce the number of special rules and constraints down to fewer more generic rules. If we make good use of patterns, we can make few rules that cover many cases. If there are no discernible patterns, every special case is a costly exception. This affects not just the technology cost, but also the cognitive cost (i.e. the comprehensibility).

Efficiency therefore plays a role in agility, because it affects the cost of change. Greater efficiency generally means greater speed, and more greater likelihood for precision.

2 Aspects of CFEngine that bring agility

We can now summarize some qualities of CFEngine that favour agility:

1. Ability to express clear intentions about desired outcome (comprehension).
2. Availability of insight into system performance and state (comprehension).
3. Ability to manage large numbers of hosts and resources with a few generic patterns (efficiency).
4. Ability to bundle related details into simple containers (comprehension without loss of adaptability).
5. Ability to accurately customize policy down to a low level without programming (adaptability).
6. Ability to recover quickly from faults and failures. The default, parallelized execution framework verifies promises every 5 minutes for rapid fault detection and change deployment (clock speed)[1] .
7. A quick system monitoring/sampling rate – every 2.5 minutes (Nyquist frequency), for automated hands-free response to errors.
8. Ability to recover cheaply. The lightweight resource footprint of CFEngine that consumes few system resources required for actual business (system speed – low overhead, maximum capacity).
9. Ability to increase number of clients without significant penalty (scalability and easy increase of capacity).
10. A single framework for all devices and operating systems (ease of migrating from one platform to another).

2.1 What agility means in different environments

Let's examine some example cases where agility plays a role. Agility only has meaning relative to an environment, so in the following sections, we cite the remarks of CFEngine users (in quotes), and their example environments.

Users' expectations for agility can differ dramatically in the present; but if we think just a few years down the line, and follow the trends, it seems clear that limber systems must prevail in IT's evolutionary jungle.

2.1.1 Desktop management

"The desktop space can be a very volatile environment, with multiple platforms."

[1] Two related concepts that are frequently referred to are the classic reliability measures: Mean Time Before Failure (MTBF) or proactive health and Mean Time To Repair (MTTR), speed of recovery: (i) If we are proactive or quick at recovering from minor problems, larger outages can be avoided. Recovery agility plays a role in avoiding cascade failure.

(ii) If the time to repair is long, or the repair is inaccurate, this could result if more widespread problems. Inaccurate change or repair often leads to attempts to 'roll-back', causing further problems.

Speed:

Speed is essential when there is a need to respond to a security threat that affects all of the desktop systems; e.g. when dealing with malware that requires the distribution of updated '`virus.dat`' files, etc. CFEngine can be very helpful by automating the process of distributing and restarting the application responsible for virus detection and mitigation. Systems that have been breached, need to be returned to known and secure state quickly to avoid loss. CFEngine can quickly detect and correct host based intrusions using file-scanning techniques and can secure hosts for examination, or just repair them quickly.

Another case for agility lies in user request processing. For example, when a new user joins a workplace and needs resources such as desktop, laptop, phone, Internet connection, VPN connection, VM instances, etc. Speed is of the essence to minimize employee downtime.

Precision:

Desktop environments can involve many different platforms: Windows, multiple flavours of Linux and Macintosh, etc. A uniform low-cost way of 'provisioning' and maintaining all of these, as well as responding to common threats is of significant value.

Precision is important to ensure that the resources made available are indeed the correct ones. Inaccuracy can be a potential security issue, or merely a productivity question.

Precision also comes into play when an enterprise rolls out new patches or productivity upgrades. These upgrades need to be uniformly and precisely distributed to all of the desktop systems during a given change window. By design, desktop clients running CFEngine automatically check for changes in system state and can precisely propagate desired state. In the case of system restoration due to corruption or hardware failure, CFEngine can greatly reduce the time needed to return to the most current enterprise build.

2.1.2 Web shops

Modern web-based companies often base their entire financial operations around an active web site. Down-time of the web service is mission critical.

Speed:

The frequency of maintenance is not usually critical in web shops, since configuration changes can be planned to occur over hours rather than minutes. During software updates and system repairs, however, speed and orchestration are issues, as time lost during upgrades is often revenue lost, and a lack of coordination of multiple parts could cause effective downtime.

It is therefore easy to scale the management of a web service, as change is rarely considered to be time-critical.

Resource availability for the web service is an issue on busy web servers, however web services are typically quite slow already and it is easy to load balance a web service, so resource efficiency of the management software is not usually considered a high priority, until the possible savings become significant with thousands of hosts.

Credit card information is subject to PCI-DSS regulation and requires a continuous verification for auditing purposes, but these systems are often separated from the main web service. Speed of execution can be seen as an advantage by some auditors where repairs to security matters and detection of breaches are carefully monitored.

Precision:

The level of customization in a web shop could be quite high, as there is a stack of interdependent services including databases and name services that have to work seamlessly, and the rate of deployment of new versions of the software might be relatively high.

Customization and individuality is a large part of a website's business competitiveness. Maintaining precise

2.1.3 Cloud providers

Speed: The cloud was designed for shorter time-scales, and relatively quick turnover of needs. That suggests that configuration will change quite often. For Infrastructure-as-a-Service providers and consumers, set up and tear-down rates are quite high so efficient and speedy configuration is imperative.

Precision:

For Software and Platform as a service providers, stability, high performance and regulation are key issues, and scaling up and down for demand is probably the fastest rate of change.

2.1.4 High Performance Computing

High Performance clusters are typically found in the oil and gas industry, in movie, financial, weather and aviation industries, and any other modelling applications where raw computation is used to crunch numbers.

Speed:

The lightweight footprint of CFEngine is a major benefit here, as every CPU cycle and megabyte of memory is precious, so workflow is not disrupted.

"A single node in the compute grid being out of sync with the others can cause the entire grid to cause failed jobs or otherwise introduce unpredictability into the environment, as it may produce results that differ from its peers. Thus it is imperative that repairs to an incorrect state happen as soon as possible, to minimize the impact of these issues."

Precision: "Precision is exquisitely important in an HPC grid. When making a configuration change, due to the homogeneity of the environment, small changes can have enormous impacts due to the quantity of affected systems. I liken this to the "monoculture" problem in replanted forests – everything is the same, so what would ordinarily be a small, easily-contained problem like a fungus outbreak, quickly spreads into an uncontrollable disaster. Thus, with HPC systems it is imperative that any changes deployed are precise, to ensure that no unintended consequences will occur. This is clearly directly related to comprehensibility of the environment – it is difficult or impossible to make a precise change when you don't fully comprehend the environment."

2.1.5 Government

Speed:

Government is not known for speed.

Precision:

"Government systems are reviewed and audited under FISMA and so one has often thought in terms of the ability to reduce complexity to make the problem manageable. Government typically wants the one-size-fits-all solution to system management and could benefit from something that can manage complexity and diversity while providing some central control (I bet you hate that word). The only thing we might have in common with finance is auditing but I'm sure the methods and goals are completely different. Finance is big money trying to make more big money. Government is focused more on compliance with its own regulations."

2.1.6 Finance

Speed:

One of the key factors in finance is liability. Fear of error, has led to very slow processing of change.

High availability in CFEngine is used for continuous auditing and security. Passing regulatory frameworks like SOX, SAS-70, ISO 20k, etc can depend on this non-intrusive availability. Liability is a major concern and significant levels of approval are generally required to make changes, with tracking of individual responsibility. Out-of-hours change windows are common for making upgrades and making intended changes. Scalability of reporting is a key here, but change happens slowly.

Precision:

Security and hence tight control of approved software are major challenges in government regulated institutions. Agility has been a low priority in the past, but this will have to change as the rest of the world's IT services accelerate.

2.1.7 Manufacturing

SCADA (supervisory control and data acquisition) generally refers to industrial control systems (ICS): computer systems that monitor and control industrial, infrastructure, or facility-based processes, as described below.

Speed:

Manufacturing is a curious mix of all the mention areas and more. In addition to the above, there is an tool component. The tools can design, build, test, track, and ship a physical unit. Downtime of any component is measured in missed revenue, so speed of detection and repair is crucial.

Precision:

"We need to ensure agility and accuracy of reporting. We need to know what is going on at any microsecond of the day. One faulty tool can throw a wrench in the whole works. The digital equivalent of the steam whistle to stop the line.

> From there, all the tool information is fed upstream to servers, from there to databases, then reports, that statistical analysis, and so on. Each piece needs to move with the product and incorporate it. It is a steady chain of events where are all information is liquid and relevant.
>
> Not only do you have the security requirements, from virus updates to top secret classification, but these tools need to never stop, ever. Also, these tools need constant reconfiguration depending on the product they are working on: e.g. you can't use the same set of procedures on XBox chip as a cellphone memory module. And all the tools are different too: one may be a probe to detect microscopic fractures in the layers, one tool may just track it's position in line. Supply and demand, cost and revenue."

2.2 Separating What from How (DevOps)

If you have to designs a programmatic solution to a challenge, it will cost you highly in terms of cognitive investment, testing and clarity of purpose to future users. Thinking *process* (how) instead of *knowledge* (what) is a classic carry-over from the era of 2nd Wave industrialization[2].

Think of CFEngine as an active knowledge management system, rather than as a relatively passive programming framework.

For 'DevOps': programming is for your application, consider its deployment to be part of the documentation.

Many programmatic systems and 'APIs' force you to explain how something will be accomplished and the statement about 'what' the outcome will be is left to an implicit assumption. Such systems are called imperative systems.

CFEngine is a declarative system. In a declarative system, the reverse is true. You begin by writing down What you are trying to accomplish and the How is more implicit. The way this is done is by separating data from algorithm in the model. CFEngine encourages this with its language, but you can go even further by using the tools optimally.

CFEngine allows you to represent raw data as variables, or as strings within your policy. For example:

```
bundle agent name
{
vars:

  "main_server" string => "abc.123.com";

  "package_source[ubuntu]" string => "repository.ubuntu.com";
  "package_source[suse]"   string => "repository.suse.com";

 # Promises that use these data
 #
 # packages:
```

[2] See http://www.cfengine.com/blog/sysadmin-3.0-and-the-third-wave

```
       # processes:
       # files:
       # services: , etc

      }
```

By separating 'what' data like this out of the details of how they are used, it becomes easier to comprehend and locate, and it becomes fast to change, and the accuracy of the change is easily perceived. Moreover, CFEngine can track the impact of such a change by seeing where the data are used.

CFEngine's knowledge management can tell you which system promises depend on which data in a clear manner, so you will know the impact of a change to the data.

You can also keep data outside your policy in databases, or sources like:

- LDAP
- NIS
- DNS
- System files

For example, reading in data from a system file is very convenient. This is what Unix-like system do for passwords and user management.

What you might lose when making an input matrix is the *why*. Is there an explanation that fits all these cases, or does each case need a special explanation? We recommend that you include as much information as possible about 'why'.

2.3 Packaging limits agility

Atomicity enables agility. Atomicity, or the avoidance of dependency, is a key approach to simplicity. Today this is often used to argue to packaging of software.

Handling software and system configuration as packages of data makes certain processes appear superficially easy, because you get a single object to deal with, that has a name and a version number. However, to maintain flexibility we should not bundle too many features into a package.

A tin of soup or a microwave meal might be a superficially easy way to make dinner, for many scenarios, but the day you get a visitor with special dietary requirements (vegetarian or allergic etc) then the prepackaging is a major obstacle to adapting: the recipe cannot be changed and repackaged without going back to the factory that made it. Thus oversimplification generally tends to end up sending up back to work around the technology.

CFEngine's modelling language gives you control over the smallest ingredients, but also allows you to package your own containers or work with other suppliers' packages. This ensures that adaptability is not sacrificed for superficial ease.

For example: your system's package model can cooperate with CFEngine make asking CFEngine to promise to work with the package manager:

```
packages:

  "apache2";
  "php5";
  "opera";
```

If you need to change what happens under the covers, it is very simple to do this in CFEngine. You can copy the details of the existing methods, because the details are not hard-coded, and you can make your own custom version quickly.

```
packages:

  "apache2"

    package_method => my_special_package_manager_interface;
```

2.4 How abstraction improves agility

Abstraction allows us to turn special cases into general patterns. This leads to a compression of information, as we can make defaults for the general patterns, which do not have to be repeated each time.

Service promises are good example of this[3], for example:

```
services:

  "www";
```

In this promise, all of the details of what happens to turn on the web service have been hidden behind this simple identifier '`www`'. This looks easy, but is it simple?

In this case, it is both easy and simple. Let's check why. We have to ask the question: how does this abstraction improve speed and precision in the long run?

Obviously, it provides short term ease by allowing many complex operations to take place with the utterance of just a single word[4]. But any software can pull that smoke and mirrors trick. To be agile, it must be possible to understand and change the details of what happens when this services is promised. Some tools hard-code processes for this kind of statement, requiring an understanding of programming in a development language to alter the result. In CFEngine, the definitions underlying this are written in the high-level declarative CFEngine language, using the same paradigm, and can therefore be altered by the users who need the promise, with only a small amount of work.

[3] Service promises, as described here, were introduced into version 3.3.0 of CFEngine in 2012.
[4] All good magic stories begin like this.

Thus, simplicity is assured by having consistency of interface and low cost barrier to changing the meaning of the definition.

2.5 Increasing system capacity (by scaling)

Capacity in IT infrastructure is increased by increasing machine power. Today, at the limit of hardware capacity, this typically means increasing the number of machines serving a task. Cloud services have increased the speed agility with which resources can be deployed – whether public or private cloud – but they do not usually provide any customization tools. This is where CFEngine brings significant value.

The rapid deployment of new services is assisted by:

- Virtualization hypervisor control or private cloud management (libvirt integration).
- Rapid, massively-parallelized custom configuration.
- Avoidance of network dependencies.

Related to capacity is the issue of scaling services for massive available capacity.

By scalability we mean the intrinsic capacity of a system to handle growth. Growth in a system can occur in three ways: by the volume of input the system must handle, in the total size of its infrastructure, and by the complexity of the processes within it.

For a system to be called scalable, growth should proceed unhindered, i.e. the size and volume of processing may expand without significantly affecting the average service level per node.

Although most of us have an intuitive notion of what scalability means, a full understanding of it is a very complex issue, mainly because there are so many factors to take into account. One factor that is often forgotten in considering scalability, is the human ability to *comprehend* the system as it grows. Limitations of comprehension often lead to over-simplification and lowest-common-denominator standardization.

Scalability is addressed in a separate document: *Scale and Scalability*, so we shall not discuss it further here.

3 Agility in your work

3.1 Easy versus simple

Just as we separate goals from actions, and strategy from tactics, so we can separate what is easy from what is simple. Easy brings short-term gratification, but simple makes the future cost less.

Easy is about barriers to adoption. If there is a cost associated with moving ahead that makes it hard:

- A psychological cost
- A cognitive cost
- It takes too long
- It costs too much money

Simple is about what happens next. Once you have started, what happens if you want to change something?

Total cost of ownership is reduced if a design is simple, as there are only a few things to learn in total. Even if those things are hard to learn, it is a one-off investment and everything that follows will be easy.

Unlike some tools, with CFEngine, you do not need to program 'how' to do things, only what you want to happen. This is always done by using the same kinds of declarations, based on the same model. You don't need to learn new principles and ideas, just more of the same.

3.2 How does complexity affect agility?

In the past[1], it was common to manage change by making everything the same. Today, the individualized *custom experience* is what today's information-society craves. Being forced into a single mold is a hindrance to adaptability and therefore to agility. To put it another way, in the modern world of commerce, consumers rule the roost, and agility is competitive edge in a market of many more players than before.

Of course, it is not quite that simple. Today, we live in a culture of 'ease', and we focus on what can be done easily (low initial investment) rather than worrying about long term simplicity (Total Cost of Ownership).

At CFEngine, we believe that 'easy' answers often suffer from the sin of over-simplification, and can lead to risky practices. After all, anyone can make something appear superficially easy by papering over a mess, or applying raw effort, but this will not necessarily scale up cheaply over time. Moreover, making a risky process 'too easy' can encourage haste and carelessness.

Any problem has an intrinsic complexity, which can be measured by the smallest amount of information required to manage it, without loss of control.

[1] Perhaps not just in the past. We are emerging from an industrial era of management where mass producing everything the same was the cheapest approach to scaling up services. However, today personal freedom demands variety and will not tolerate such oversimplification.

- *Ease* is the absence of a barrier or cost to action.
- *Simplicity* is a strategy for minimizing Total Cost of Ownership.

Making something truly simple is a very hard problem, but it is an investment in future change. What is easy today might be expensive to make easy tomorrow. But if something is truly simple, then the work is all up front in learning the basics, and does not come as an unexpected surprise down the line.

At CFEngine, we believe in agility through simplicity, and so we invest continuous research into making our technology genuinely simple for trained users. We know that a novice user will not necessarily find CFEngine easy, but after a small amount of training, CFEngine will be a tool for life, not just a hurried deployment.

Simplicity in CFEngine is addressed in the following ways:

- The software has few dependencies that complicate installation and upgrading.
- Changes made are atomic and minimize dependencies.
- Each host works as an independent entity, reducing communication fragility.
- The configuration model is based on Promise Theory – a very consistent and simple approach to modelling autonomous cooperative systems.
- All hosts run the same software agents on all operating platforms (from mobile phones to mainframes), and understand a single common language of intent, which they can translate into native system calls. So there are few exceptions to deal with.
- Comprehensive facilities are allowed for making use of patterns and other total-information-reducing tactics.

A certain level of complexity might be necessary and desirable – complexity is relative. Some organizations still try to remain agile by avoiding complexity. However, the ability to respond to complex scenarios often requires us to dabble with diversity. Avoiding it merely creates a lack of agility, as one is held back by the need to over-simplify.

3.3 An effective understanding helps agility

All configuration issues, including fitness for purpose, boil down to three things: why, what and how. Knowing why we do something is the most important way of avoiding error and risk of failure. Simplicity then comes from keeping the 'what' and the 'how' separate, and reducing the how to a predictable, repairable transaction. This is what CFEngine's *convergent promise* technology does.

Knowledge is an antidote to uncertainty. Insight into patterns, brings simplicity to the information management, and insight into behaviour allows us to estimate impact of change, thus avoiding the risk associated with agility.

In configuration 'what' represents transitory knowledge, while 'how' is often more lasting and can be absorbed into the infrastructure. The consistency and repairability of 'how' makes it simpler to change what without risk.

3.4 Maximizing business imperatives

Agility allows companies and public services to compete and address the needs of continuous service improvement. This requires insight into IT operations from business and vice versa. Recently, the 'DevOps' movement in web arenas has emphasized the need for a more streamlined approach to integrating business-driven change and IT operations. Whatever we choose to call this, and in whatever arena, 'connecting the dots between business and IT' is a major enabler for agility to business imperatives.

Some business issues are inherently complex, e.g. software customization and security, because they introduce multifaceted conflicts of interest that need to resolved with clear documentation about *why*.

Be careful about choosing a solution because it has a low initial outlay cost. Look to the long term cost, or the Total Cost of Ownership over the next 5 years.

Many businesses have used the argument: *everything is getting cheaper so it doesn't matter if my software is inefficient – I can brute force it in a year's time with more memory and a faster CPU.* The flaw in this argument is that complexity and scale are also increasing, and you will need those savings down the line even more than you do now.

The ability to model our intentions in a clearly understandable way enables insight and understanding; this, in turn, allows us to anticipate and comprehend challenges. CFEngine's knowledge management features help to make the configuration itself a part of the documentation of the system. Instead of relying on command line tools to interact, the user documents intentions (as 'promises to be kept'). These promises, and how well they have been kept, can be examined either from the original specification or in the Mission Portal.

In the industrial age, the strategy was to supply sufficient force to a small problem in order to 'control' it by brute force. In systems today the scale and complexity are such that no such brute force approach can seriously be expected to work. Thus one is reduced to a more even state of affairs: learning to work with the environment 'as is', with clear expectations of what is possible and controlling only certain parts on which crucial things depend.

3.5 What does agility cost?

CFEngine is designed to have a low Total Cost of Ownership, by being exceptionally lightweight and conceptually simple. The investment in CFEngine is a 'learning curve' that some find daunting. Indeed, at CFEngine, we work on reducing this initial learning curve all the time – but what really saves you in the end is simplicity without over-simplification.

At a recent deployment in the banking sector, CFEngine replaced an incumbent software solution where 200 machines were required to make the management infrastructure scale to the task.

CFEngine replaced this with 3 machines, and a reduced workforce. After the replacement the clock-time required for system updates went from 45 minutes to 16 seconds.

The total cost of providing for agility can be costly or it can be cheap. By design, CFEngine aims to make scale and agility inexpensive in the long run.

3.6 Who is responsible for agility?

The bottom line is: you are! Diversity and customization are basic freedoms that user-driven services demand in today's world, and having the agility to meet changing desires is going to be an increasingly important and prominent feature of IT, as we delve further into the information-based society.

Competitive edge, response to demands, in both private sector and research, makes agility the actual product of a not-too-distant tomorrow.

Who or what makes agility a reality? The simple answer to this question is everyone and everything. Change is a chain of dependent activities and the weakest link in the chain is the limiting factor. Often, that is human knowledge, since it is the part of the chain that we take most for granted.

CFEngine has been carefully designed to support agile operations for the long term, by investing in knowledge management, speed and efficiency.