AUTOMATED SYSTEM CONFIGURATION REPAIR

Aaron Weiss, Arjun Guha, Yuriy Brun
University of Massachusetts

I'm Aaron Weiss, and I'll be speaking today about Automated System Configuration Repair.

This work was completed while I was an undergrad at UMass, but *click* I've just started my Ph.D at Northeastern.

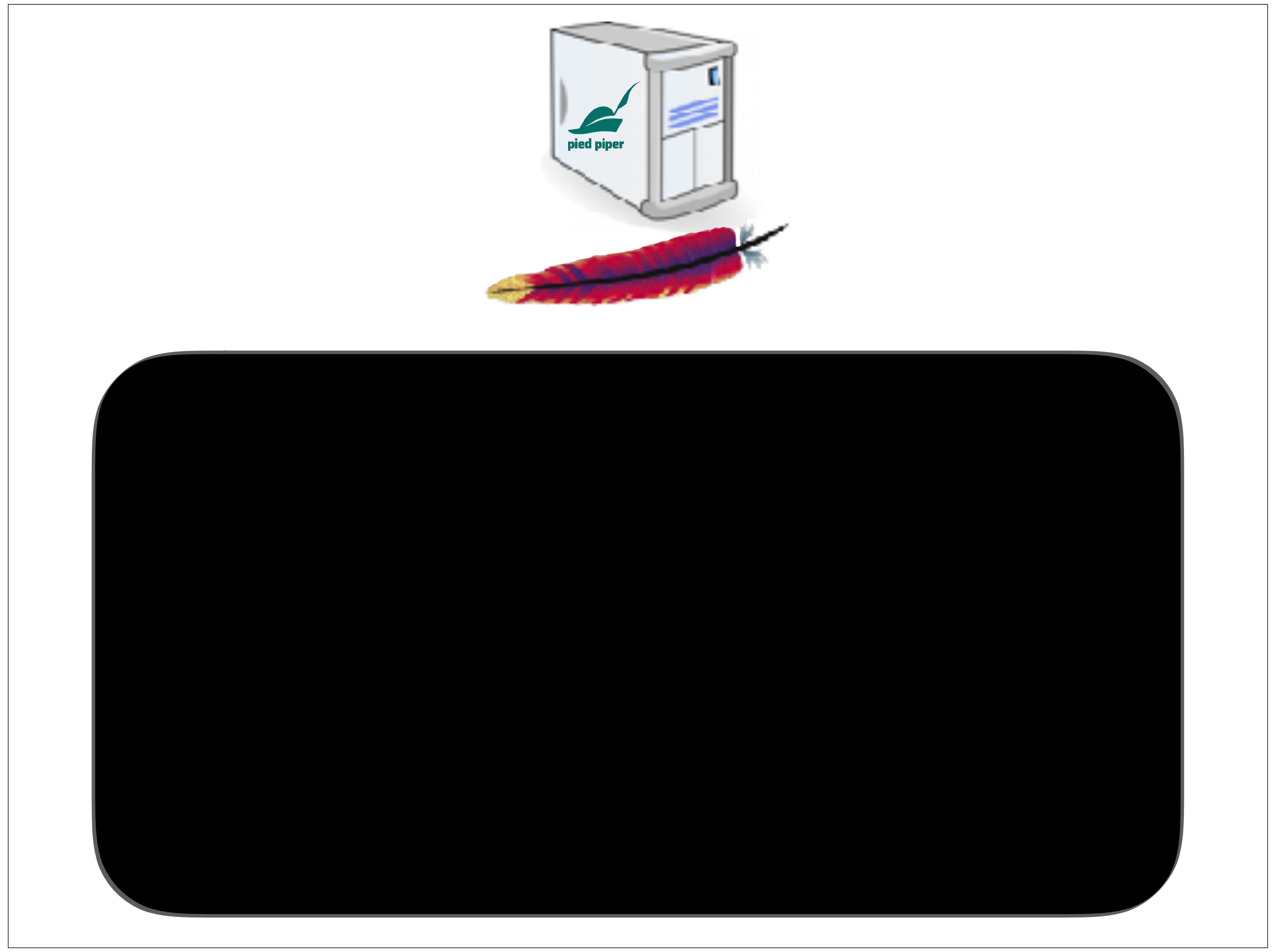
AUTOMATED SYSTEM CONFIGURATION REPAIR

Aaron Weiss, *Arjun Guha, Yuriy Brun*
University of Massachusetts

I’m Aaron Weiss, and I’ll be speaking today about Automated System Configuration Repair.

This work was completed while I was an undergrad at UMass, but *click* I’ve just started my Ph.D at Northeastern.

For the purpose of this talk, I’m an employee at the fictional startup Pied Piper which you may know from HBO’s Silicon Valley.



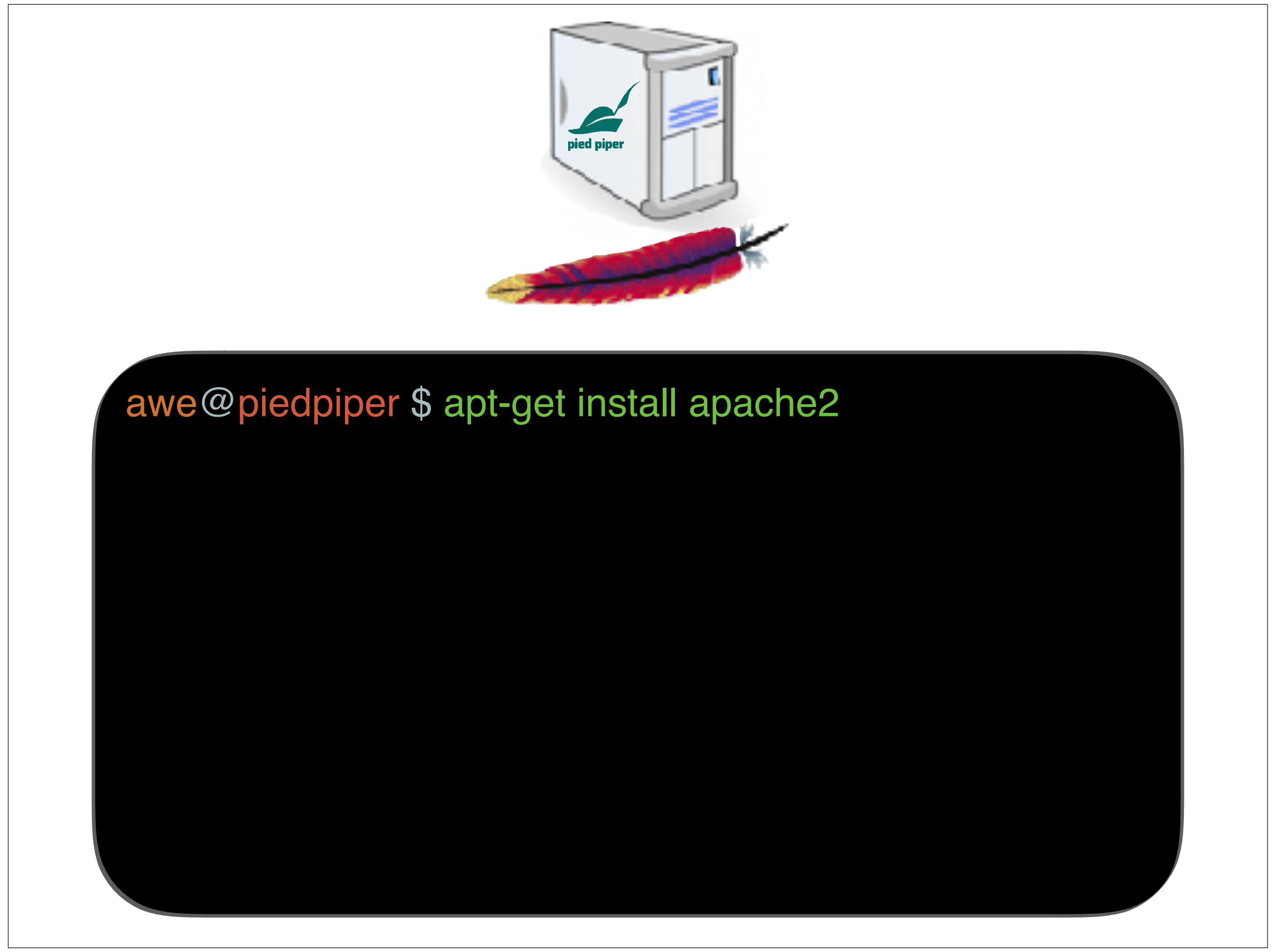
At Pied Piper, we use Apache to host our static web content. Let’s take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we’ll edit it again and restart.

We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click



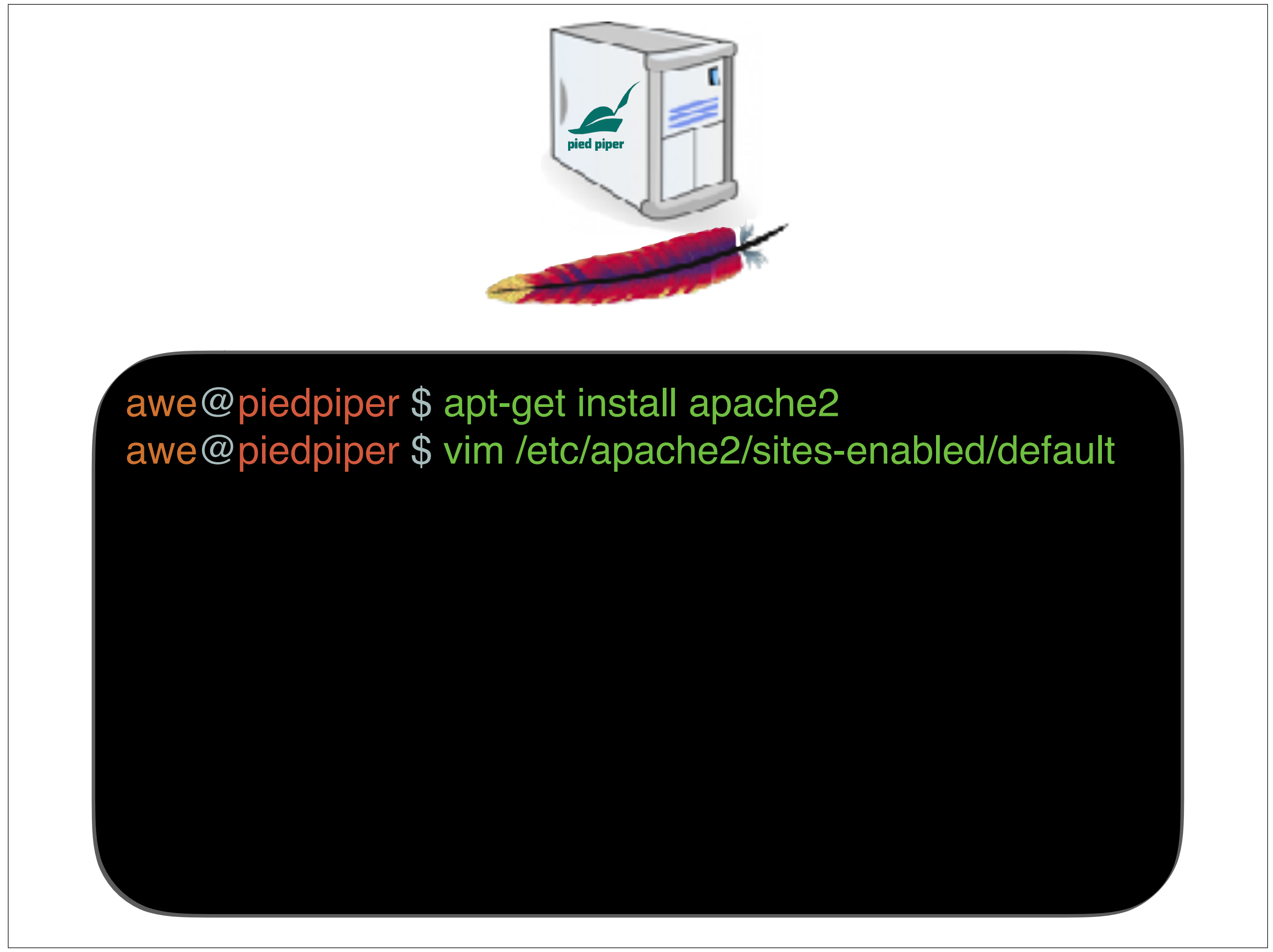
At Pied Piper, we use Apache to host our static web content. Let’s take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we’ll edit it again and restart.

We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click



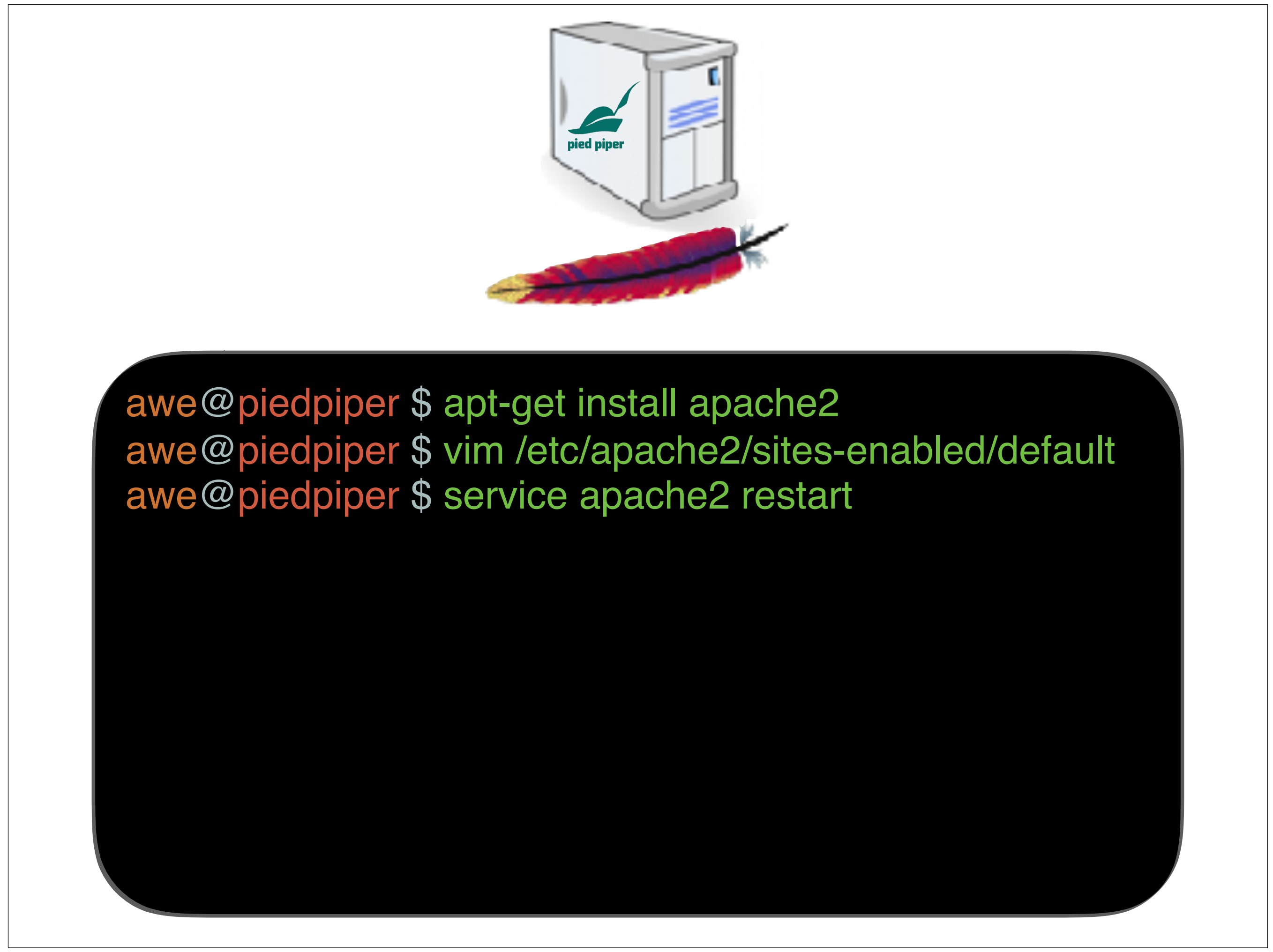
At Pied Piper, we use Apache to host our static web content. Let’s take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we’ll edit it again and restart.

We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click



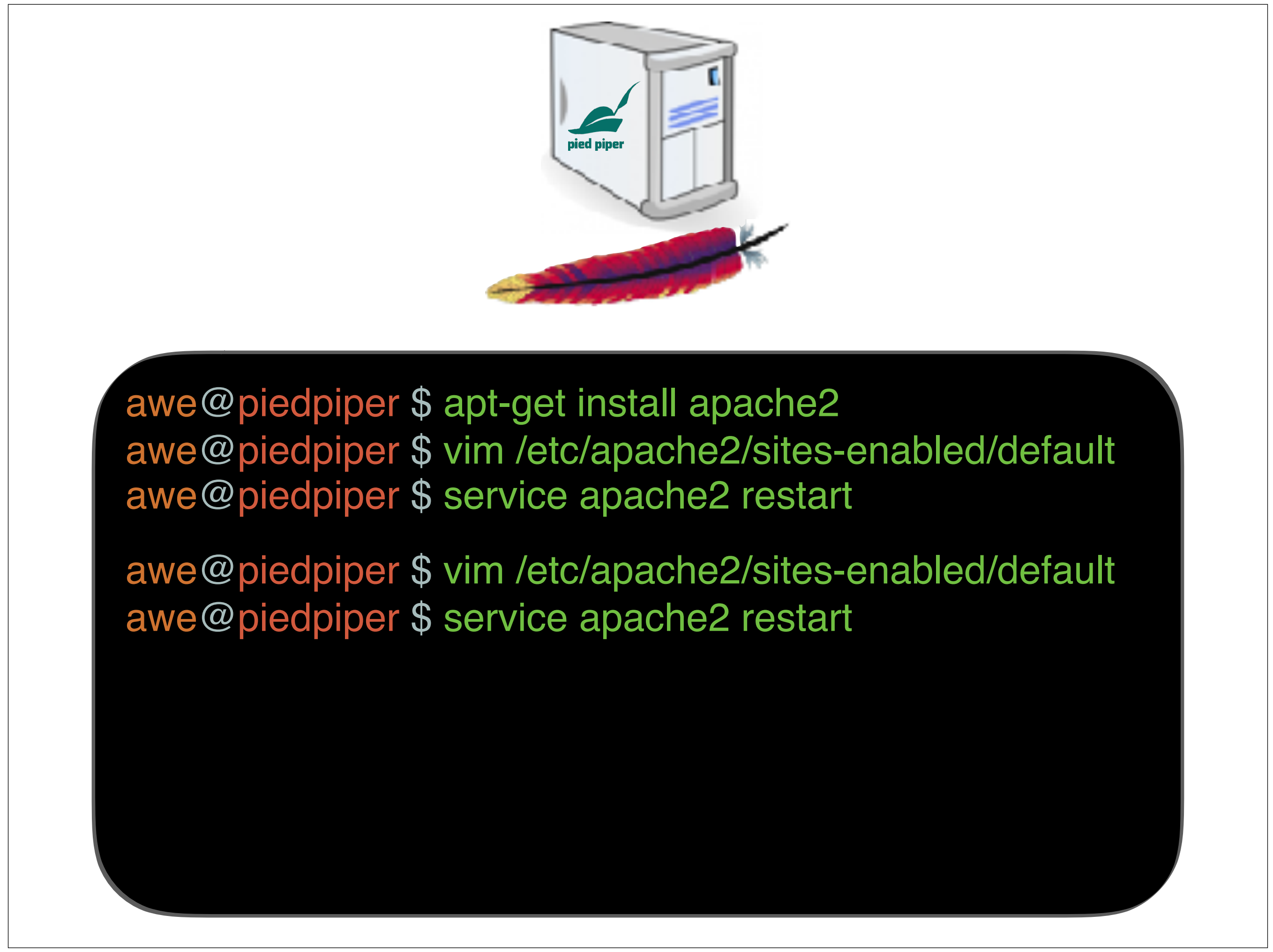
At Pied Piper, we use Apache to host our static web content. Let’s take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we’ll edit it again and restart.

We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click



At Pied Piper, we use Apache to host our static web content. Let’s take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we’ll edit it again and restart.

We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click

At Pied Piper, we use Apache to host our static web content. Let's take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we'll edit it again and restart.

We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click


```
awe@piedpiper $ apt-get install apache2
awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ iptables -dport ssh -j DROP

awe@piedpiper $ mount backup.local:/backup /mnt/backup
awe@piedpiper $ crontab -e
```

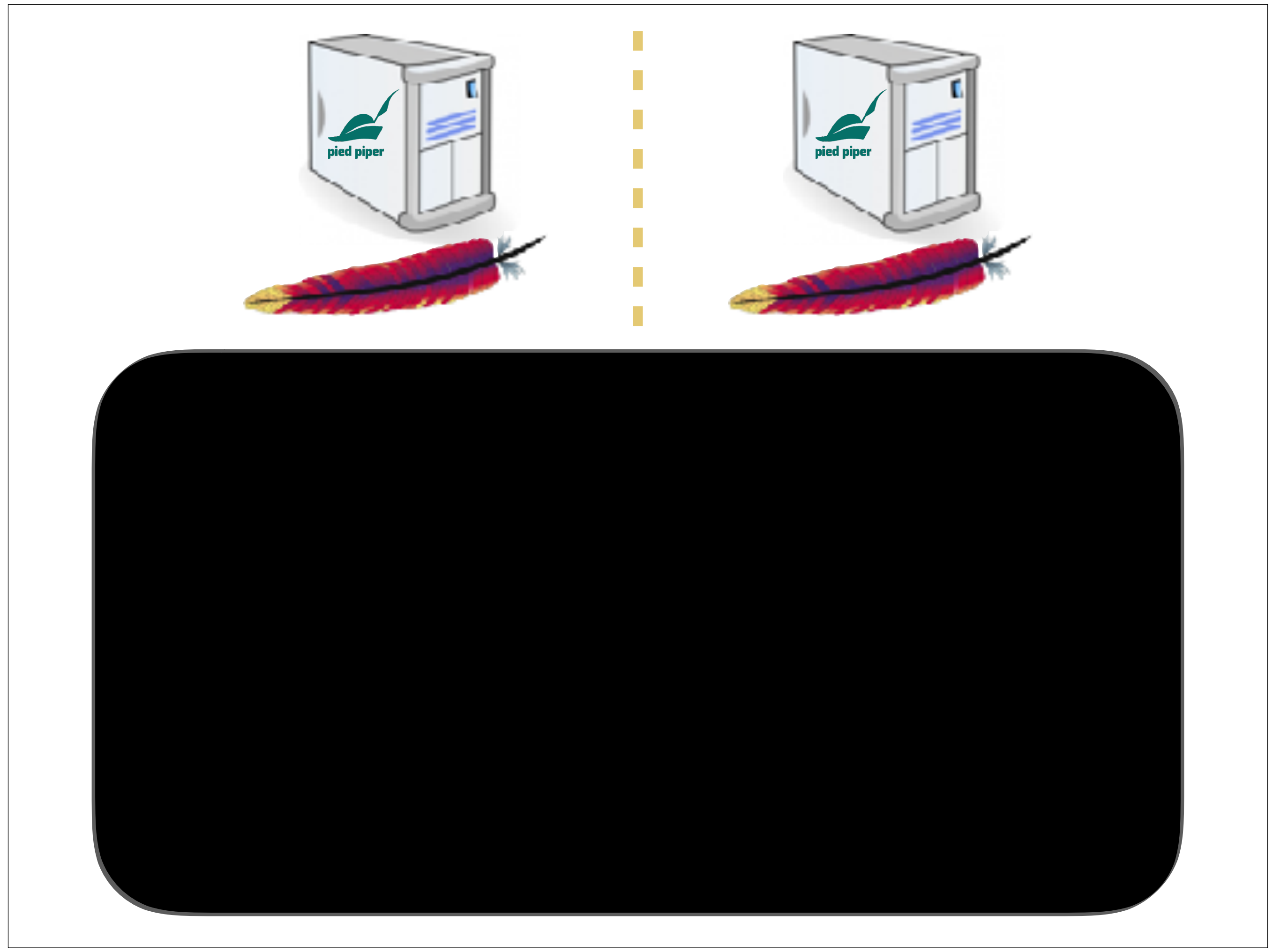
At Pied Piper, we use Apache to host our static web content. Let's take a look at setting up one of these servers.

click We install Apache
click Then, we edit the configuration
click And then we restart the service.

Of course, we made a mistake.
click So, we'll edit it again and restart.

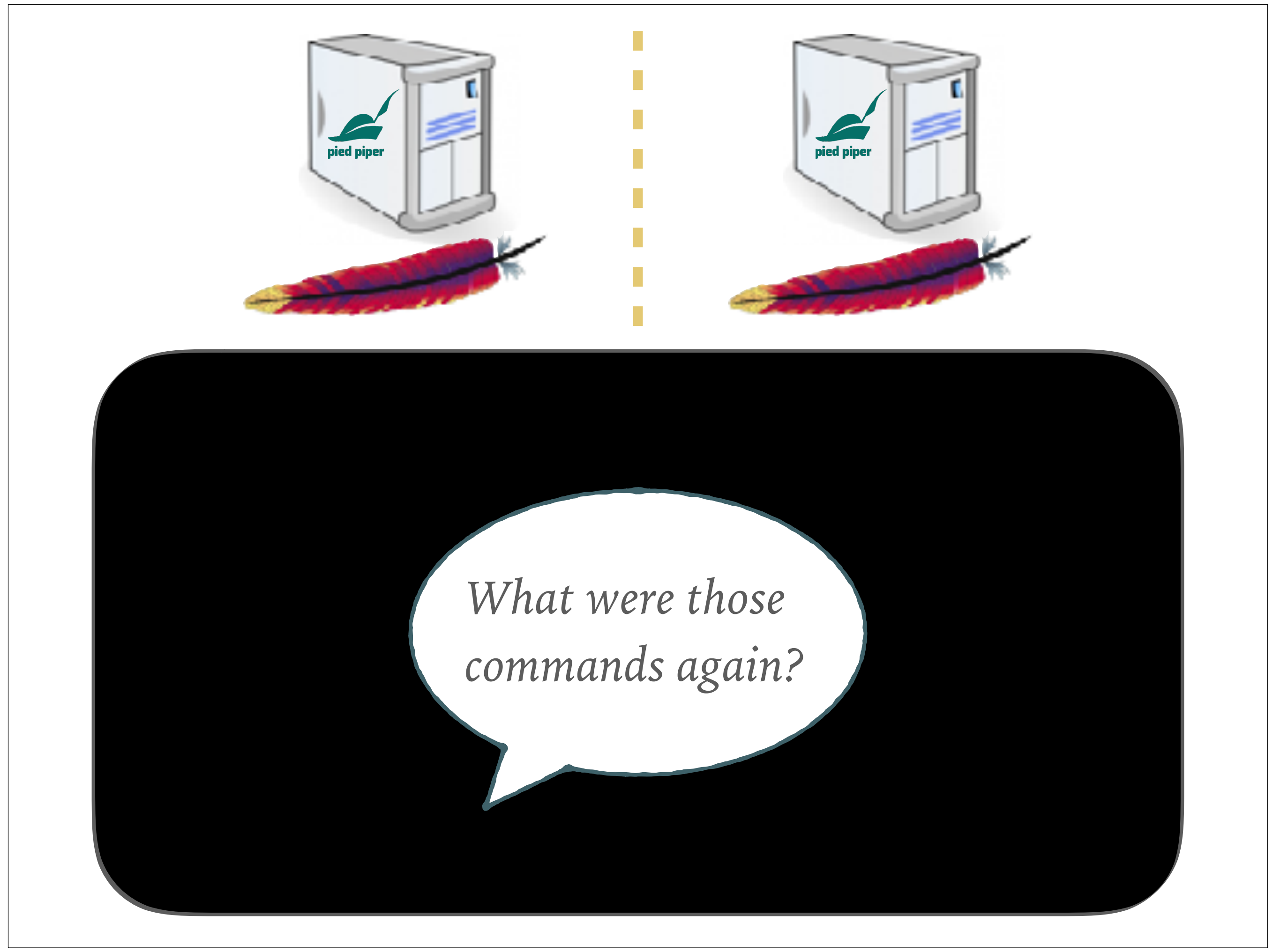
We also forgot to prevent SSH traffic for security.
click

And we probably want to set up our backup system too.
click



So, what happens when we need to add a new server?

We need to do all of that again, but *click* I don't remember what those commands were.



So, what happens when we need to add a new server?

We need to do all of that again, but *click* I don’t remember what those commands were.

And we have a lot more than just two servers. We need to configure all of them!

New York Stock Exchange:

"a software update went out [...] it returned an error. [...] There was clearly a difference in the configuration going into production [from the test environment]"

Of course, we can also make mistakes in configuring them.

The New York Stock Exchange had an outage resulting from a malformed configuration during a software update.

click Airbus had a military plane failure due to a configuration error.

click And most pressing of all, Facebook was down for over two hours because of a fault in configuration.

click
This issue is so pressing, even the police needed to comment.

New York Stock Exchange:

"a software update went out [...] it returned an error. [...] There was clearly a difference in the configuration going into production [from the test environment]"

Airbus military plane crash: "The error was in configuration settings programmed into the electronic control unit (ECU) of the engines and not in the code itself."

Of course, we can also make mistakes in configuring them.

The New York Stock Exchange had an outage resulting from a malformed configuration during a software update.

click Airbus had a military plane failure due to a configuration error.

click And most pressing of all, Facebook was down for over two hours because of a fault in configuration.

click
This issue is so pressing, even the police needed to comment.

New York Stock Exchange:

"a software update went out [...] it returned an error. [...] There was clearly a difference in the configuration going into production [from the test environment]"

Airbus military plane crash: "The error was in configuration settings programmed into the electronic control unit (ECU) of the engines and not in the code itself."

Facebook:

"Facebook was down or unreachable for many of you for approximately 2.5 hours. [...] An automated system for verifying configuration values ended up causing much more damage than it fixed."

Of course, we can also make mistakes in configuring them.

The New York Stock Exchange had an outage resulting from a malformed configuration during a software update.

click Airbus had a military plane failure due to a configuration error.

click And most pressing of all, Facebook was down for over two hours because of a fault in configuration.

click
This issue is so pressing, even the police needed to comment.

Of course, we can also make mistakes in configuring them.

The New York Stock Exchange had an outage resulting from a malformed configuration during a software update.

click Airbus had a military plane failure due to a configuration error.

click And most pressing of all, Facebook was down for over two hours because of a fault in configuration.

click
This issue is so pressing, even the police needed to comment.

CONFIGURATION MANAGEMENT TOOLS

To deal with the challenges of system configuration, there's been a number of tools for configuration management developed. These tools give high-level declarative abstractions.

In this talk, we'll look at Puppet. I'll give you an example of such an abstraction in a moment, but Puppet has roughly 5,000 modules that provide pre-made abstractions for setting up a variety of software.

```
user {“aaron”:
  ensure => present,
  managehome => true
}

file {“/home/
aaron/.spacemacs”:
  ensure => present,
  content => “…”
}
```

Here we have a simple Puppet program (called manifests).

- Makes a user account, aaron
- Creates a spacemacs configuration for the user

When we want to add another user, rather than copy and paste…
click
We can create an abstraction and instantiate it with different values.

```
user {“aaron”:
  ensure => present,
  managehome => true
}

file {“/home/
aaron/.spacemacs”:
  ensure => present,
  content => “…”
}
```

```
define account($name) {
  user {$name:
    ensure => present,
    managehome => true
  }

  file {“/home/$name/.spacemacs”:
    ensure => present,
    content => “…”
  }
}
account {“aaron”: }
account {“arjun”: }
```

Here we have a simple Puppet program (called manifests).

- Makes a user account, aaron
- Creates a spacemacs configuration for the user

When we want to add another user, rather than copy and paste…
click
We can create an abstraction and instantiate it with different values.

WHY ARE THESE TOOLS HARD TO USE?

But those configuration failures we saw still happen even with these tools. Why?

THEY'RE UNFAMILIAR

For one, tools like Puppet are unfamiliar to system administrators because they're still very new.

UNLIKE THE SHELL…

By contrast, most sysadmins have been using the shell since early on in their career.
click
It’s our home.

UNLIKE THE SHELL…

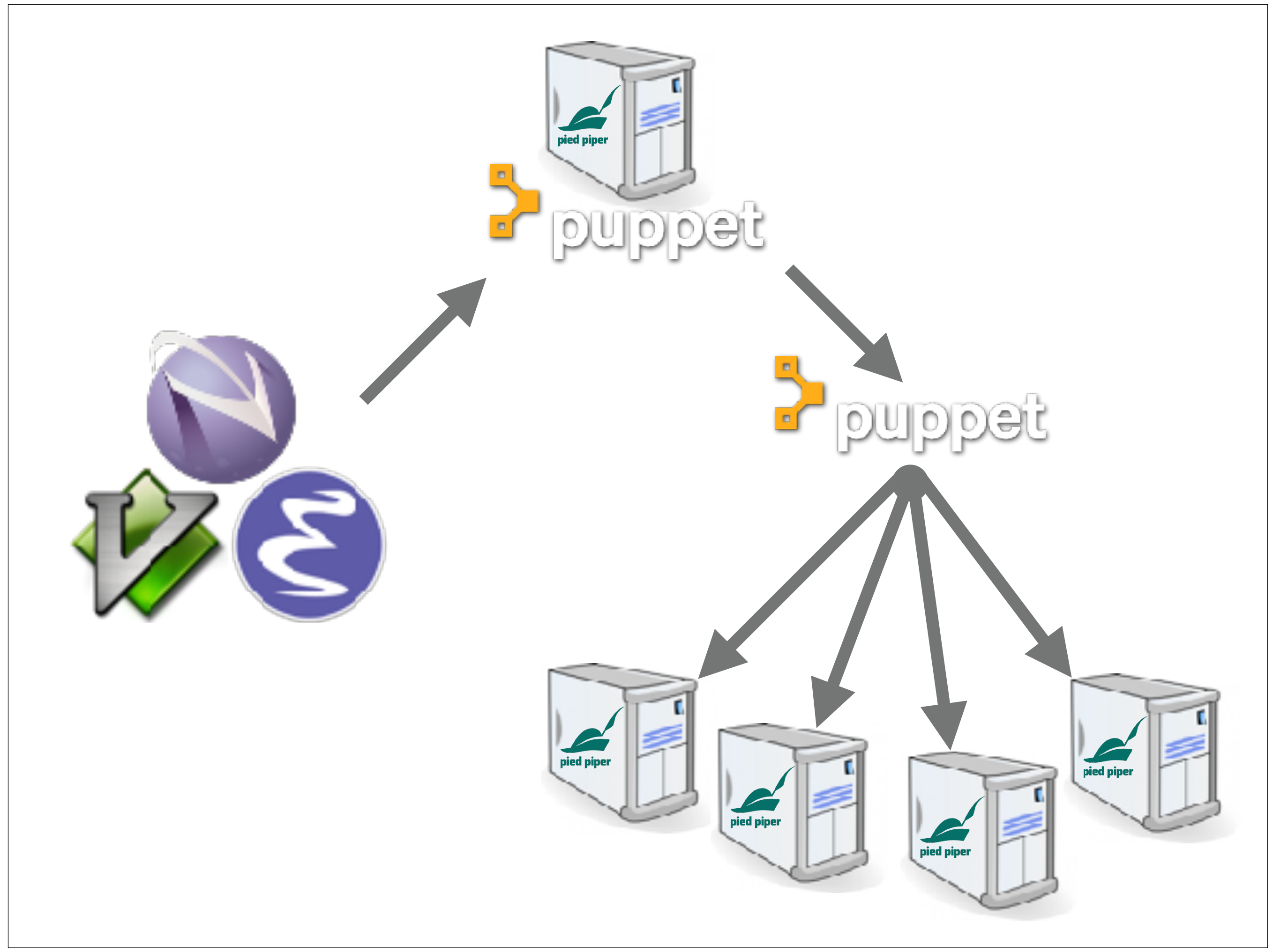
```
awe@piedpiper $ apt-get install apache2
awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ iptables -dport ssh -j DROP

awe@piedpiper $ mount backup.local:/backup /mnt/backup
awe@piedpiper $ crontab -e
```

Home! Sweet Home!

By contrast, most sysadmins have been using the shell since early on in their career.
click
It’s our home.

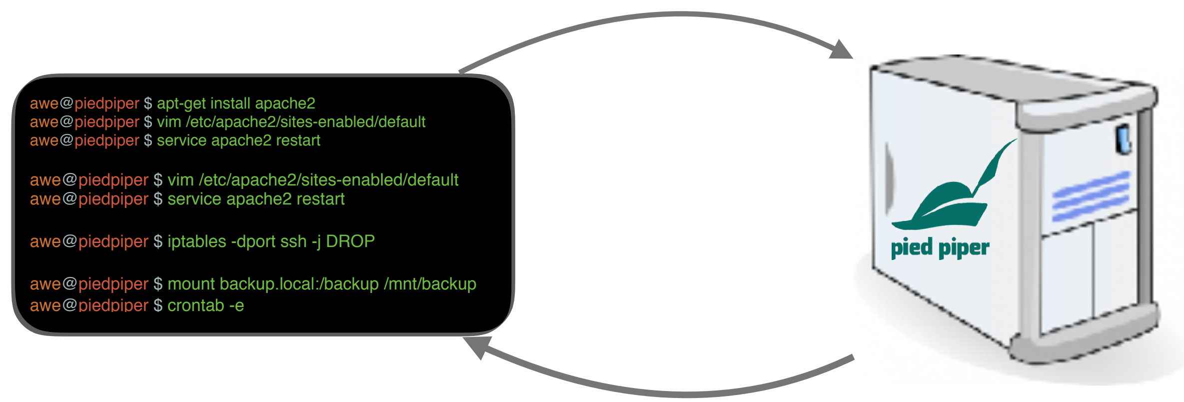
THEY'RE REMOVED

Configuration management tools are also removed from the systems they're configuring.



In particular, Puppet manifests are hard to test. To run a manifest, we have to reconfigure a system which can take minutes to hours depending on the manifest. Doing this repeatedly as one might do with an ordinary program is intractable.

UNLIKE THE SHELL…



This is again unlike the shell which runs atop a system and gives immediate feedback.

WHY NOT USE THE SHELL?

So, why not use the shell to fix these bugs?

By using the shell, we move the state of the machine away from the state specified by the manifest causing *configuration drift*. This can cause a number of problems with the manifest over time.

WHY NOT USE THE SHELL?

Configuration drift!

So, why not use the shell to fix these bugs?

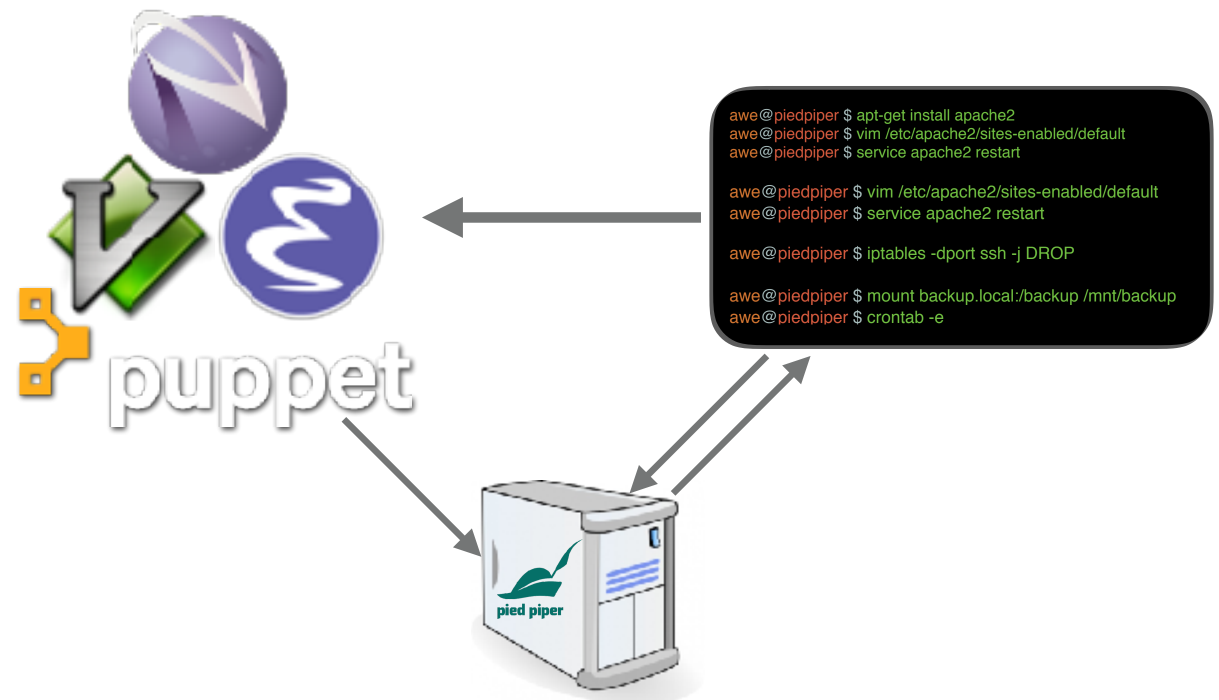
By using the shell, we move the state of the machine away from the state specified by the manifest causing *configuration drift*. This can cause a number of problems with the manifest over time.

CAN WE GET THE BEST OF BOTH WORLDS?

So, you’re probably wondering then: is there some way we can keep our abstractions but get the immediacy and familiarity of the shell?

Well, I probably wouldn’t be here if the answer was no…

IMPERATIVE CONFIGURATION REPAIR



What if we could fix configuration bugs directly from the shell and have the changes propagate automatically back to the Puppet manifest?

We call this technique “imperative configuration repair” or ICR for short.

PROPERTIES OF IMPERATIVE CONFIGURATION REPAIR

ICR has a number of important properties that make it useful for system administrators.

click Firstly, it’s sound meaning that it preserves all changes made from the shell

click Secondly, it protects maintainability by preserving the structure and abstraction of the original manifest.

click Thirdly, when multiple repairs are possible, it ranks them and presents them to the user in a logical fashion.

click Finally, it places no restrictions on what shell is used and should work with all existing shells.

PROPERTIES OF IMPERATIVE CONFIGURATION REPAIR

- Sound: All changes made via the shell are preserved

ICR has a number of important properties that make it useful for system administrators.

click Firstly, it’s sound meaning that it preserves all changes made from the shell

click Secondly, it protects maintainability by preserving the structure and abstraction of the original manifest.

click Thirdly, when multiple repairs are possible, it ranks them and presents them to the user in a logical fashion.

click Finally, it places no restrictions on what shell is used and should work with all existing shells.

PROPERTIES OF IMPERATIVE CONFIGURATION REPAIR

- Sound: All changes made via the shell are preserved
- Maintainable: Structure and abstraction is preserved

ICR has a number of important properties that make it useful for system administrators.

click Firstly, it’s sound meaning that it preserves all changes made from the shell

click Secondly, it protects maintainability by preserving the structure and abstraction of the original manifest.

click Thirdly, when multiple repairs are possible, it ranks them and presents them to the user in a logical fashion.

click Finally, it places no restrictions on what shell is used and should work with all existing shells.

PROPERTIES OF IMPERATIVE CONFIGURATION REPAIR

- Sound: All changes made via the shell are preserved
- Maintainable: Structure and abstraction is preserved
- Ranked: Multiple possible repairs are ranked

ICR has a number of important properties that make it useful for system administrators.

click Firstly, it's sound meaning that it preserves all changes made from the shell

click Secondly, it protects maintainability by preserving the structure and abstraction of the original manifest.

click Thirdly, when multiple repairs are possible, it ranks them and presents them to the user in a logical fashion.

click Finally, it places no restrictions on what shell is used and should work with all existing shells.

PROPERTIES OF IMPERATIVE CONFIGURATION REPAIR

- Sound: All changes made via the shell are preserved
- Maintainable: Structure and abstraction is preserved
- Ranked: Multiple possible repairs are ranked
- Unrestricted: Works with all existing shells

ICR has a number of important properties that make it useful for system administrators.

click Firstly, it’s sound meaning that it preserves all changes made from the shell

click Secondly, it protects maintainability by preserving the structure and abstraction of the original manifest.

click Thirdly, when multiple repairs are possible, it ranks them and presents them to the user in a logical fashion.

click Finally, it places no restrictions on what shell is used and should work with all existing shells.

WHAT DOES ICR LOOK LIKE?

So, what does ICR actually look like?

For that, we'll go back to our fictional startup, Pied Piper.

WHAT DOES ICR LOOK LIKE?

So, what does ICR actually look like?

For that, we'll go back to our fictional startup, Pied Piper.

```
package {“apache2”: ensure => present }
service {“apache2”: ensure => running }
define website($title, $root) {
  file {“/etc/apache2/sites-enabled/$title.conf”:
    content => “<VirtualHost $title:80>
    DocumentRoot /var/sites/$root
    </VirtualHost>” }
  file {“/var/sites/$root”:
    ensure => directory,
    source => “puppet://sites/$root”,
    owner => “root”,
    mode => 0700,
    recurse => “remote” }
}
website {“piedpiper.com": root => “piedpiper” }
website {“piperchat.com": root => “piperchat” }
```

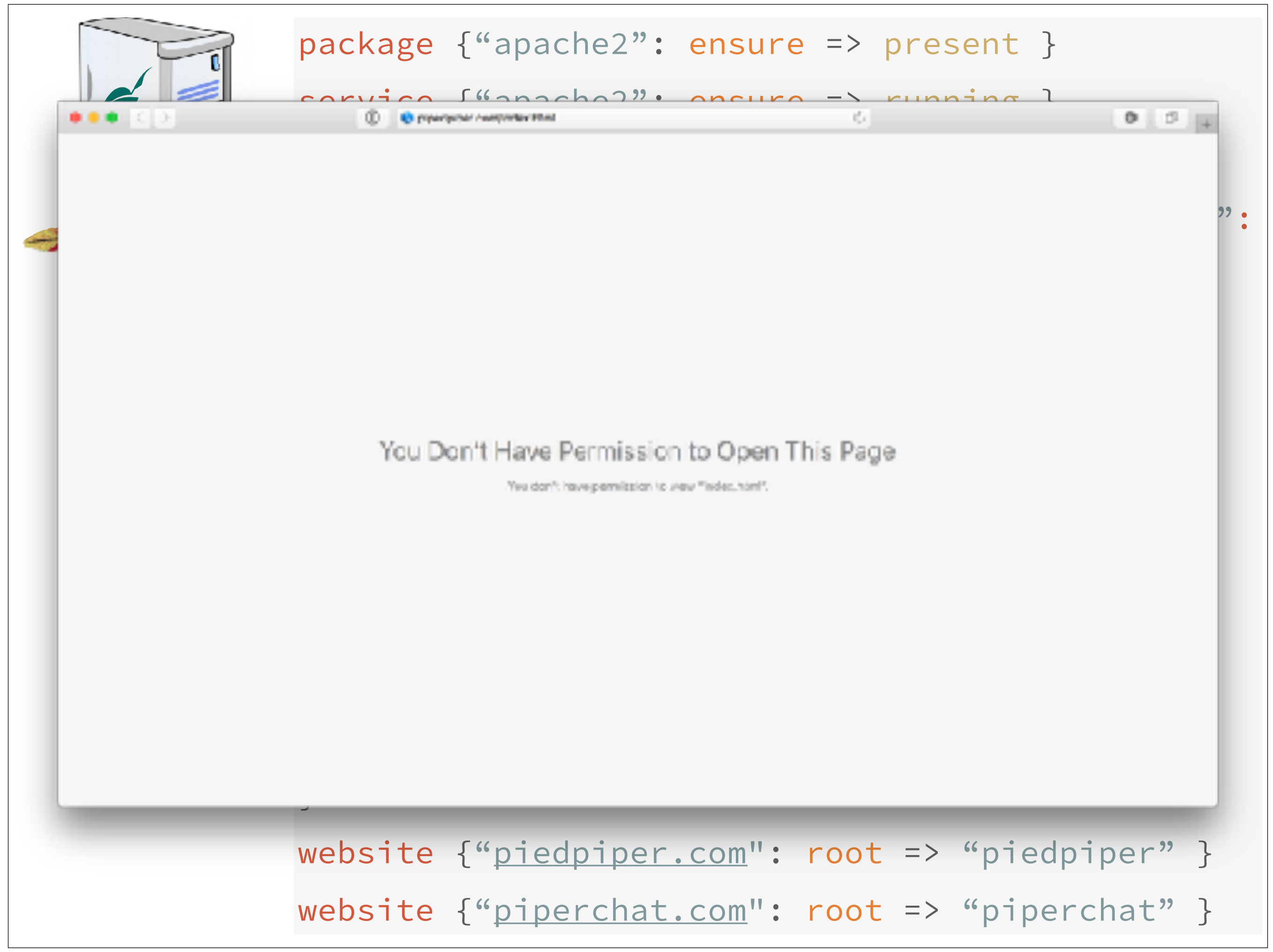
At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let’s go take a look at the site!
click
Oh, no. It looks like we’ve got some error: we’re getting a 403 Forbidden.

Let’s open the shell, and take a look at what’s going wrong.
click
First, we’ll want to look at the Apache log files.
click
Well, the log says permission denied. I’ve been using Apache for a while. So, I know that that means either there’s an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn’t set up an htaccess file. So, let’s check the permissions.

click
Ah, there’s the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let’s fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click



At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let’s go take a look at the site!
click
Oh, no. It looks like we’ve got some error: we’re getting a 403 Forbidden.

Let’s open the shell, and take a look at what’s going wrong.
click
First, we’ll want to look at the Apache log files.
click
Well, the log says permission denied. I’ve been using Apache for a while. So, I know that that means either there’s an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn’t set up an htaccess file. So, let’s check the permissions.

click
Ah, there’s the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let’s fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click

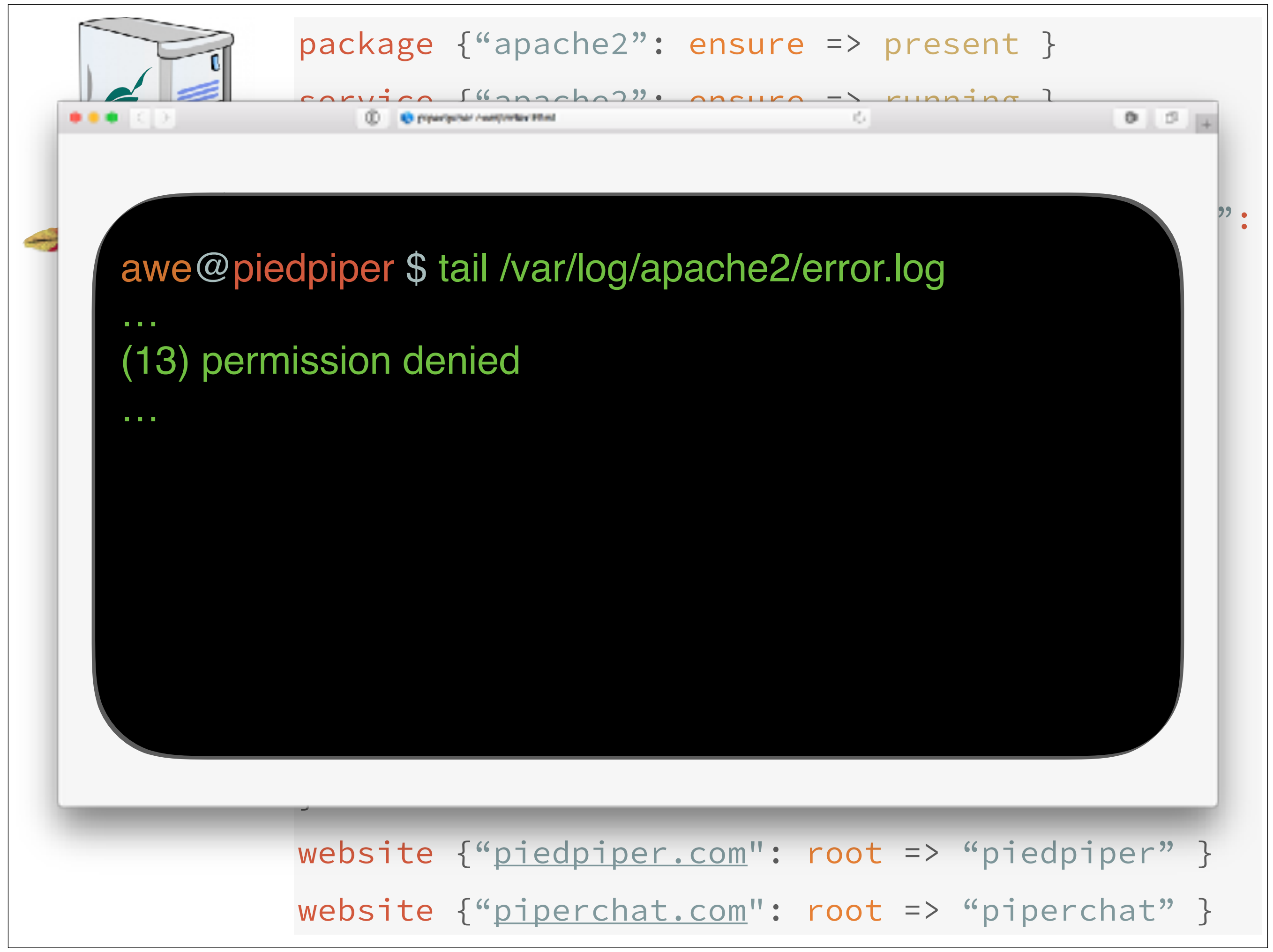
At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let’s go take a look at the site!
click
Oh, no. It looks like we’ve got some error: we’re getting a 403 Forbidden.

Let’s open the shell, and take a look at what’s going wrong.
click
First, we’ll want to look at the Apache log files.
click
Well, the log says permission denied. I’ve been using Apache for a while. So, I know that that means either there’s an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn’t set up an htaccess file. So, let’s check the permissions.

click
Ah, there’s the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let’s fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click



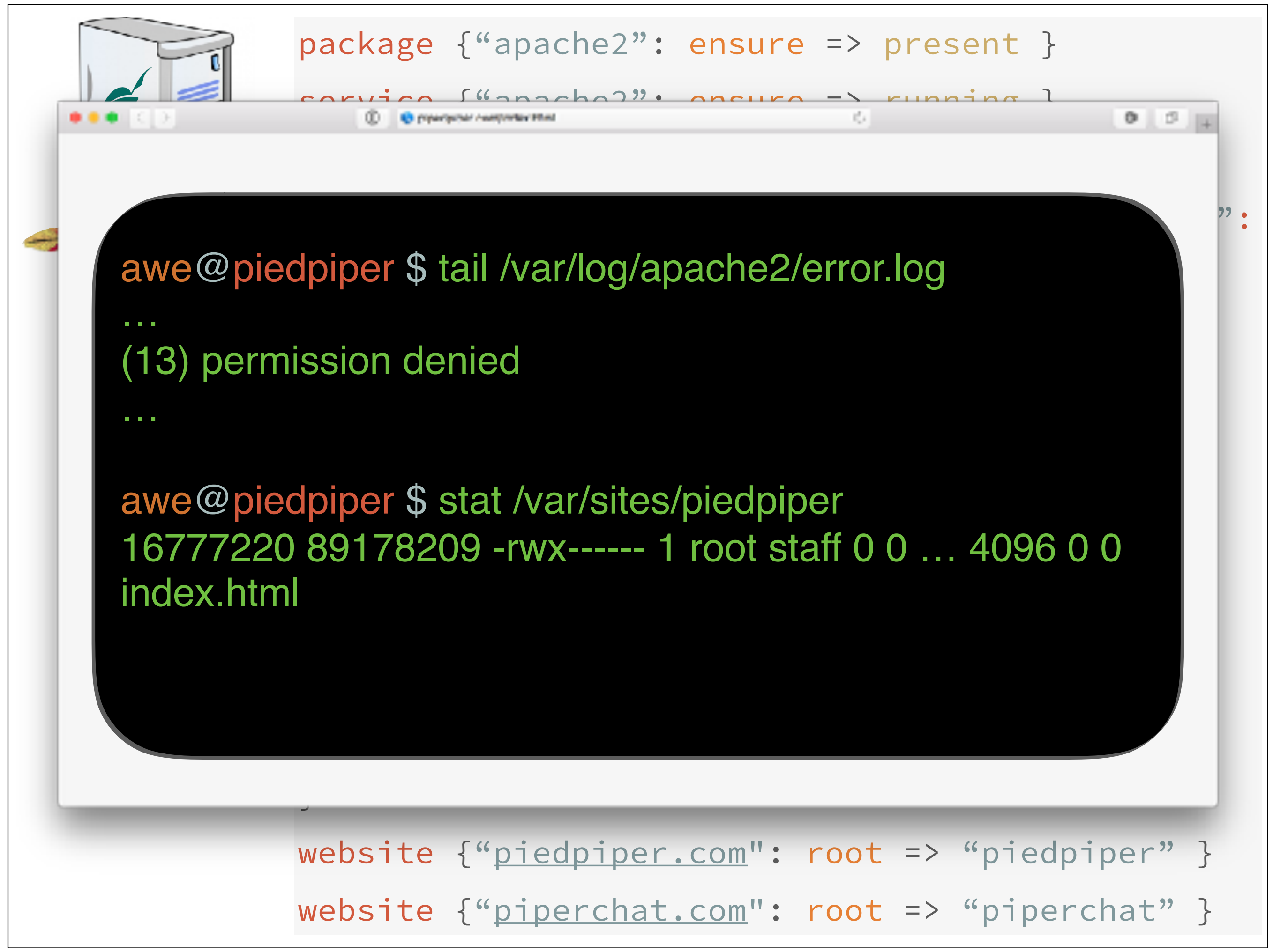
At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let's go take a look at the site!
click
Oh, no. It looks like we've got some error: we're getting a 403 Forbidden.

Let's open the shell, and take a look at what's going wrong.
click
First, we'll want to look at the Apache log files.
click
Well, the log says permission denied. I've been using Apache for a while. So, I know that that means either there's an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn't set up an htaccess file. So, let's check the permissions.

click
Ah, there's the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let's fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click



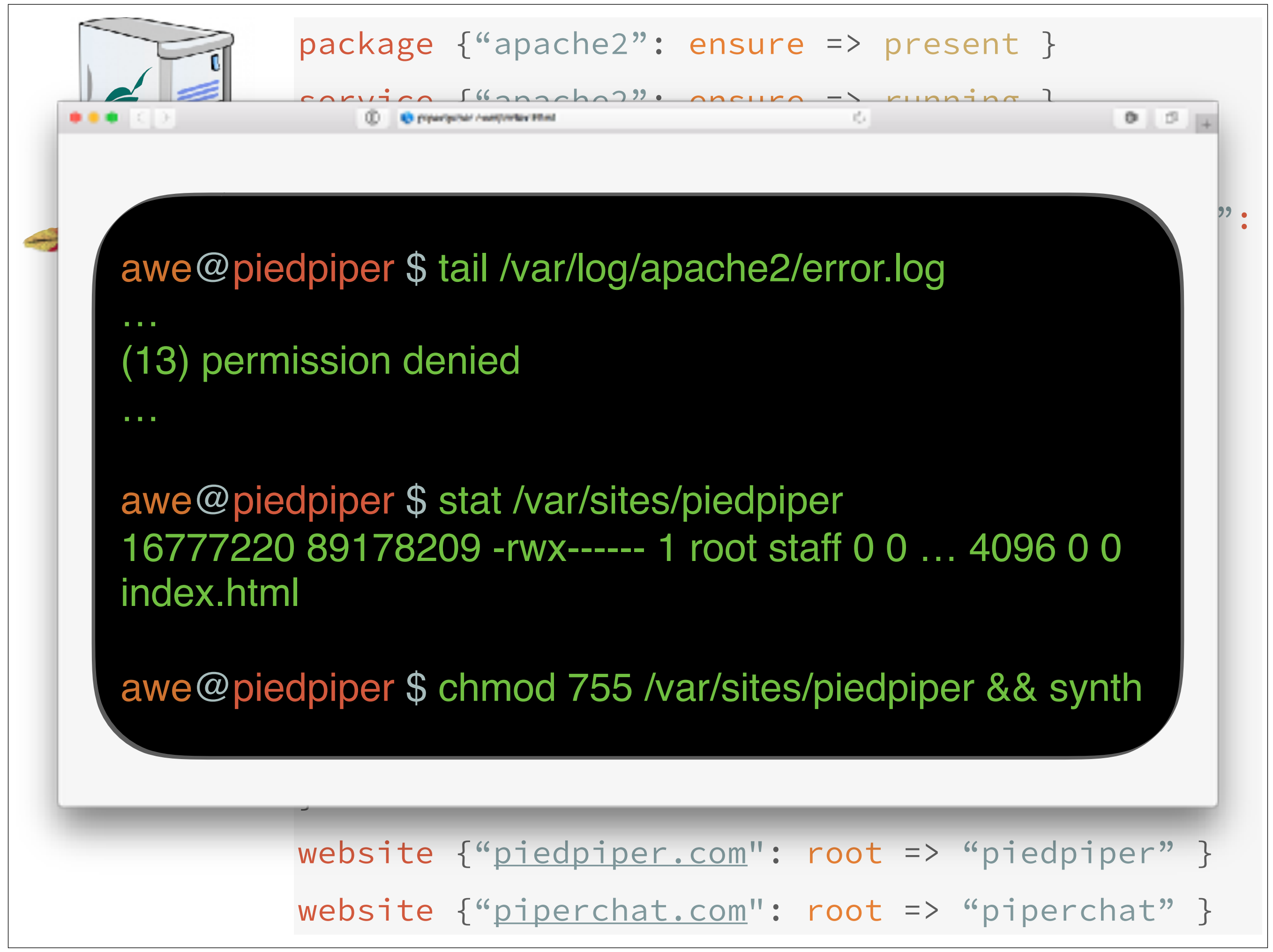
At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let’s go take a look at the site!
click
Oh, no. It looks like we’ve got some error: we’re getting a 403 Forbidden.

Let’s open the shell, and take a look at what’s going wrong.
click
First, we’ll want to look at the Apache log files.
click
Well, the log says permission denied. I’ve been using Apache for a while. So, I know that that means either there’s an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn’t set up an htaccess file. So, let’s check the permissions.

click
Ah, there’s the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let’s fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click



At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let's go take a look at the site!
click
Oh, no. It looks like we've got some error: we're getting a 403 Forbidden.

Let's open the shell, and take a look at what's going wrong.
click
First, we'll want to look at the Apache log files.
click
Well, the log says permission denied. I've been using Apache for a while. So, I know that that means either there's an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn't set up an htaccess file. So, let's check the permissions.

click
Ah, there's the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let's fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click


```
package {“apache2”: ensure => present }
service {“apache2”: ensure => running }
define website($title, $root) {
  file {“/etc/apache2/sites-enabled/$title.conf”:
    content => “<VirtualHost $title:80>
    DocumentRoot /var/sites/$root
    </VirtualHost>” }
  file {“/var/sites/$root”:
    ensure => directory,
    source => “puppet://sites/$root”,
    owner => “root”,
    mode => 0755,
    recurse => “remote” }
}
website {“piedpiper.com": root => “piedpiper” }
website {“piperchat.com": root => “piperchat” }
```

At Pied Piper, we wrote the following manifest to install Apache, and set up our websites, PipedPiper.com and PiperChat.com. We use a single abstraction to set up the website, and it copies the files for each site from our Puppet master.

We already deployed it to our web servers. So, let’s go take a look at the site!
click
Oh, no. It looks like we’ve got some error: we’re getting a 403 Forbidden.

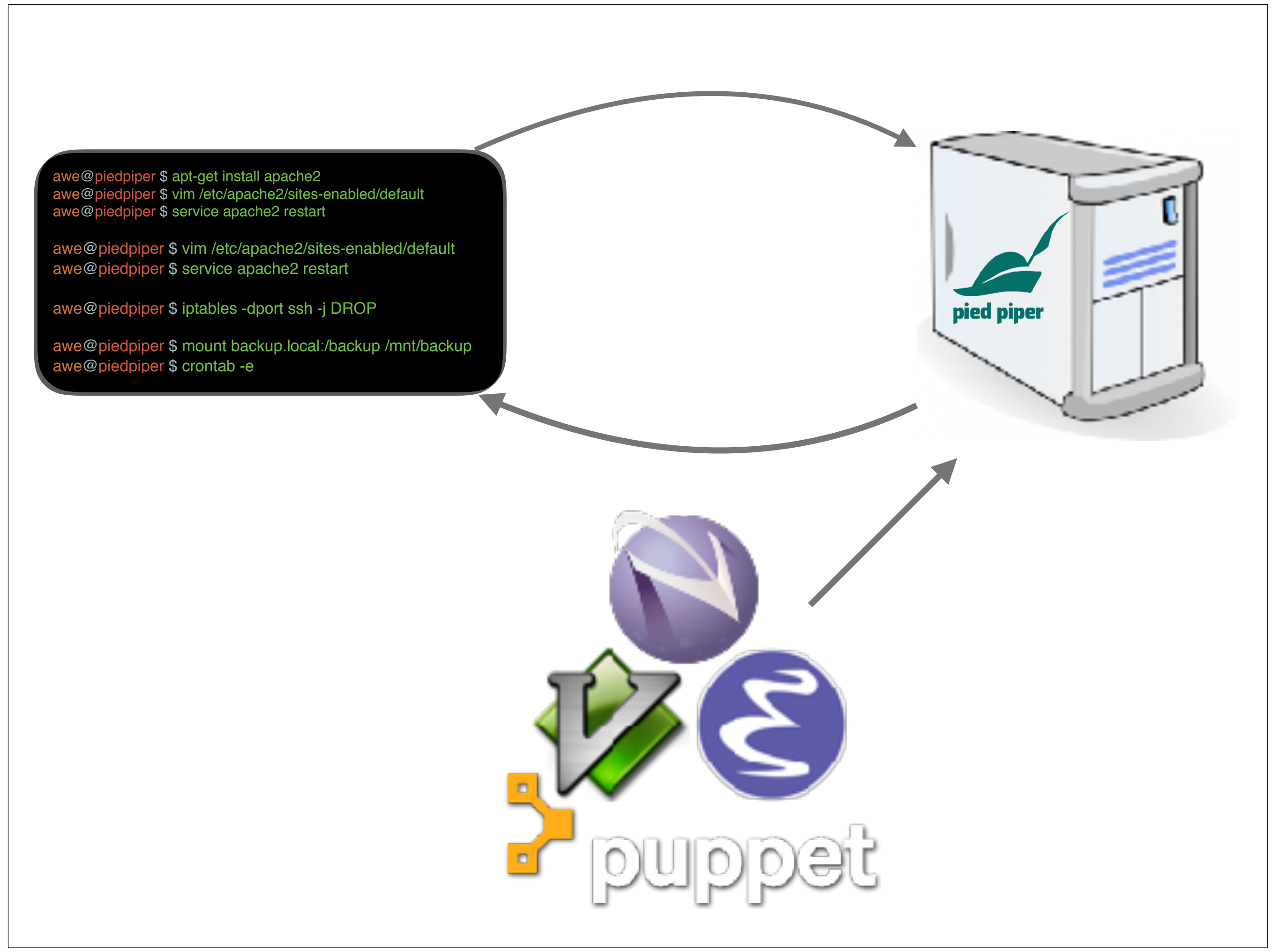
Let’s open the shell, and take a look at what’s going wrong.
click
First, we’ll want to look at the Apache log files.
click
Well, the log says permission denied. I’ve been using Apache for a while. So, I know that that means either there’s an htaccess file preventing us from viewing the page or the permissions are broken on the site directory. I didn’t set up an htaccess file. So, let’s check the permissions.

click
Ah, there’s the problem. Only root is able to view the files, but our www user needs to be able to see them for Apache to serve the content. So, let’s fix that.

click
And now, thanks to imperative configuration repair, the change can propagate back to the original manifest automatically.
click

TORTOISE: AN ICR PROTOTYPE

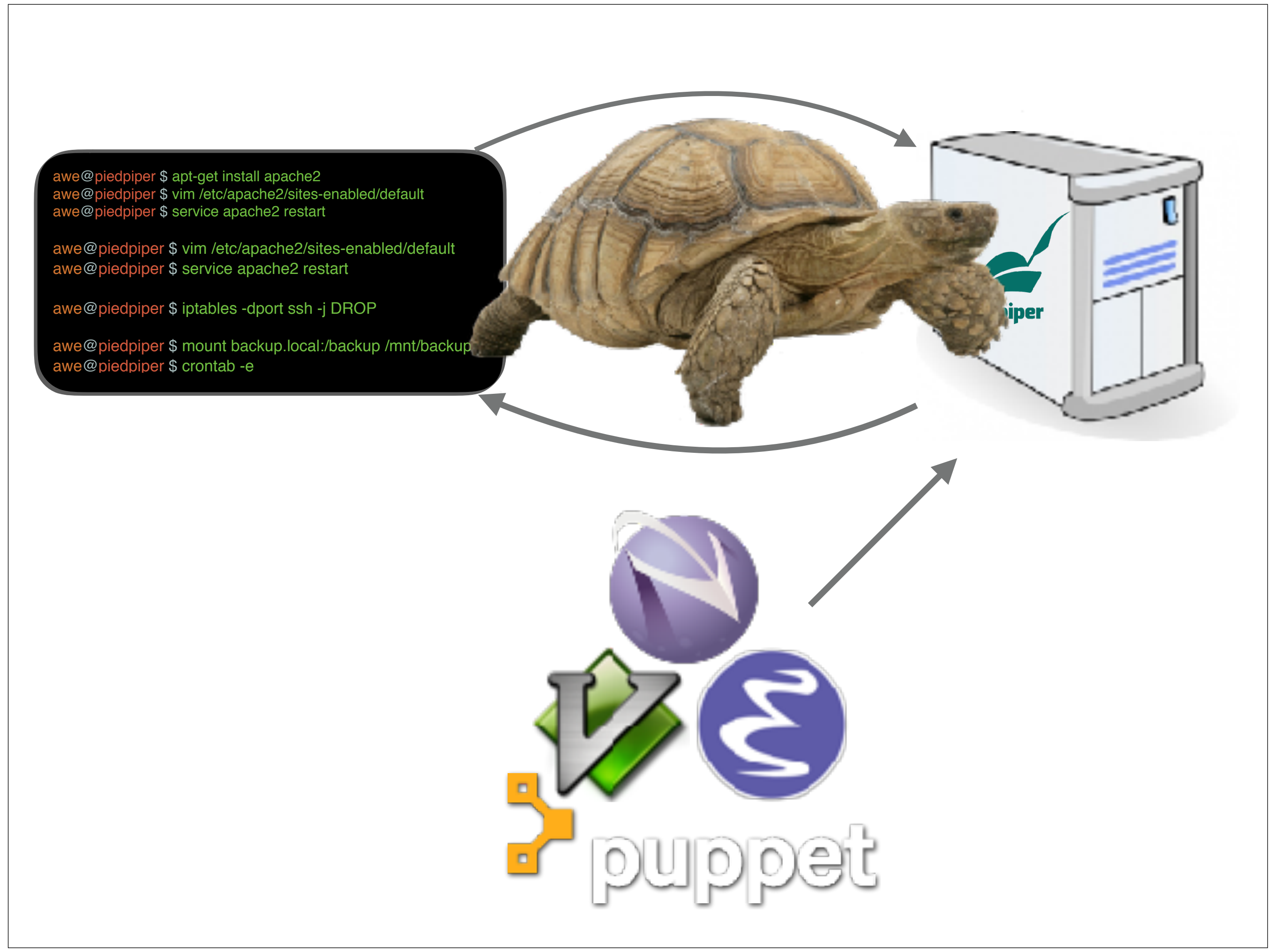
We implemented a prototype of this ICR technique called Tortoise that targets Puppet manifests.



Ordinarily, we'd deploy our manifest to a machine, make changes via the shell, and then go back and edit the manifest accordingly.

click
With ICR, Tortoise sits on the machine, monitoring the shell and the file system in the background. The user can make changes via the shell as usual.

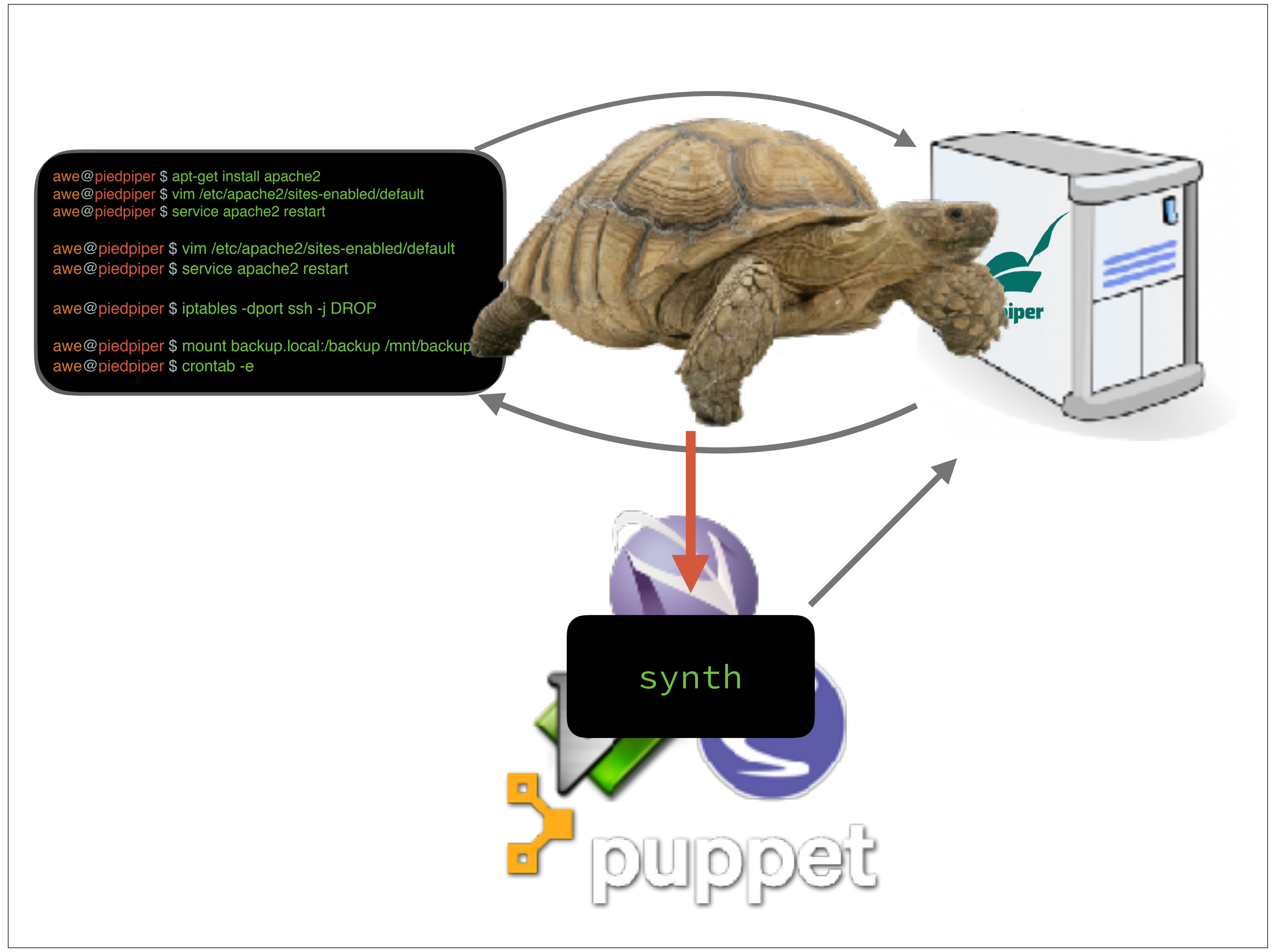
click
When they've fixed the bug, they can run a command to propagate the changes back to the manifest.



Ordinarily, we'd deploy our manifest to a machine, make changes via the shell, and then go back and edit the manifest accordingly.

click
With ICR, Tortoise sits on the machine, monitoring the shell and the file system in the background. The user can make changes via the shell as usual.

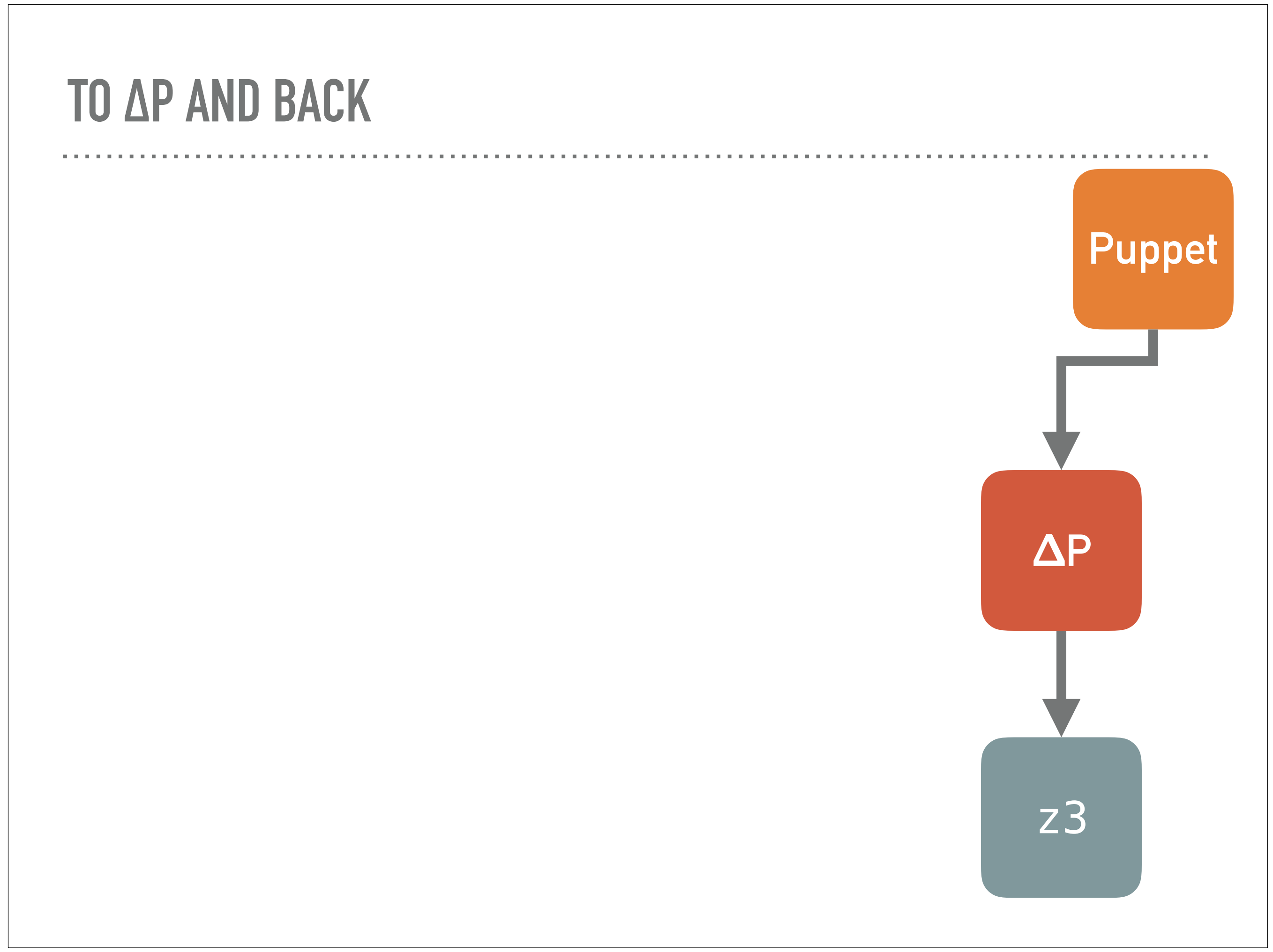
click
When they've fixed the bug, they can run a command to propagate the changes back to the manifest.



Ordinarily, we’d deploy our manifest to a machine, make changes via the shell, and then go back and edit the manifest accordingly.

click
With ICR, Tortoise sits on the machine, monitoring the shell and the file system in the background. The user can make changes via the shell as usual.

click
When they’ve fixed the bug, they can run a command to propagate the changes back to the manifest.



click
First, we compile the manifest into an imperative program in a language named Delta-P that models file system operations.

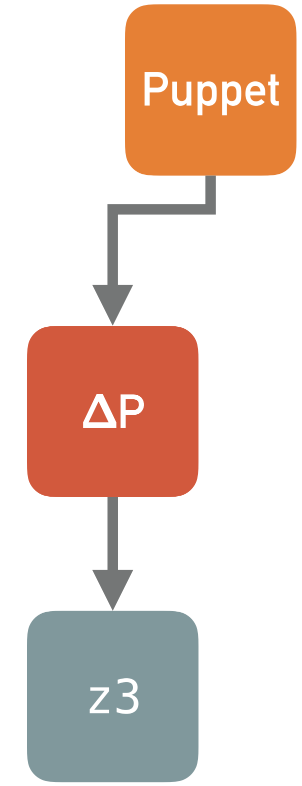
This compilation step introduces repairable let-bindings whose values can be updated during ICR.

click
Next, we convert the shell commands into assertions in ΔP that capture the desired final state of the file system.

Since these were changes made from the original manifest, these assertions may be currently false. We will use an SMT solver, namely z3, to repair the manifest to make them true.

TO ΔP AND BACK

- Compile manifest into ΔP specification
  - Imperative language + filesystem ops
  - Let bindings with z3-updatable values



click
First, we compile the manifest into an imperative program in a language named Delta-P that models file system operations.

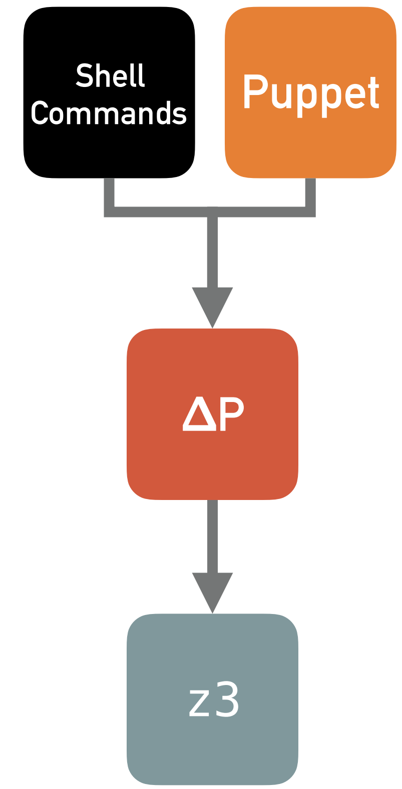
This compilation step introduces repairable let-bindings whose values can be updated during ICR.

click
Next, we convert the shell commands into assertions in ΔP that capture the desired final state of the file system.

Since these were changes made from the original manifest, these assertions may be currently false. We will use an SMT solver, namely z3, to repair the manifest to make them true.

TO ΔP AND BACK

- Compile manifest into ΔP specification
  - Imperative language + filesystem ops
  - Let bindings with z3-updatable values
- Convert shell commands into ΔP assertions
  - Changes via the shell mean these assertions are false
  - z3 will repair the manifest to make them true



click
First, we compile the manifest into an imperative program in a language named Delta-P that models file system operations.

This compilation step introduces repairable let-bindings whose values can be updated during ICR.

click
Next, we convert the shell commands into assertions in ΔP that capture the desired final state of the file system.

Since these were changes made from the original manifest, these assertions may be currently false. We will use an SMT solver, namely z3, to repair the manifest to make them true.

MULTIPLE REPAIRS

```
define dir($path) {
  file {$path:
    ensure => directory
  }
}
dir { path => “/foo” }
```

Of course, there are often multiple possible repairs that will make the assertions true. To deal with this, we developed a procedure for ranking repairs.

Consider the simple example shown here with a dir abstraction creating a directory, and a single instantiation for the path “/foo”.

If we use a command like *click* “mv /foo /bar”, there are actually two possible updates here.

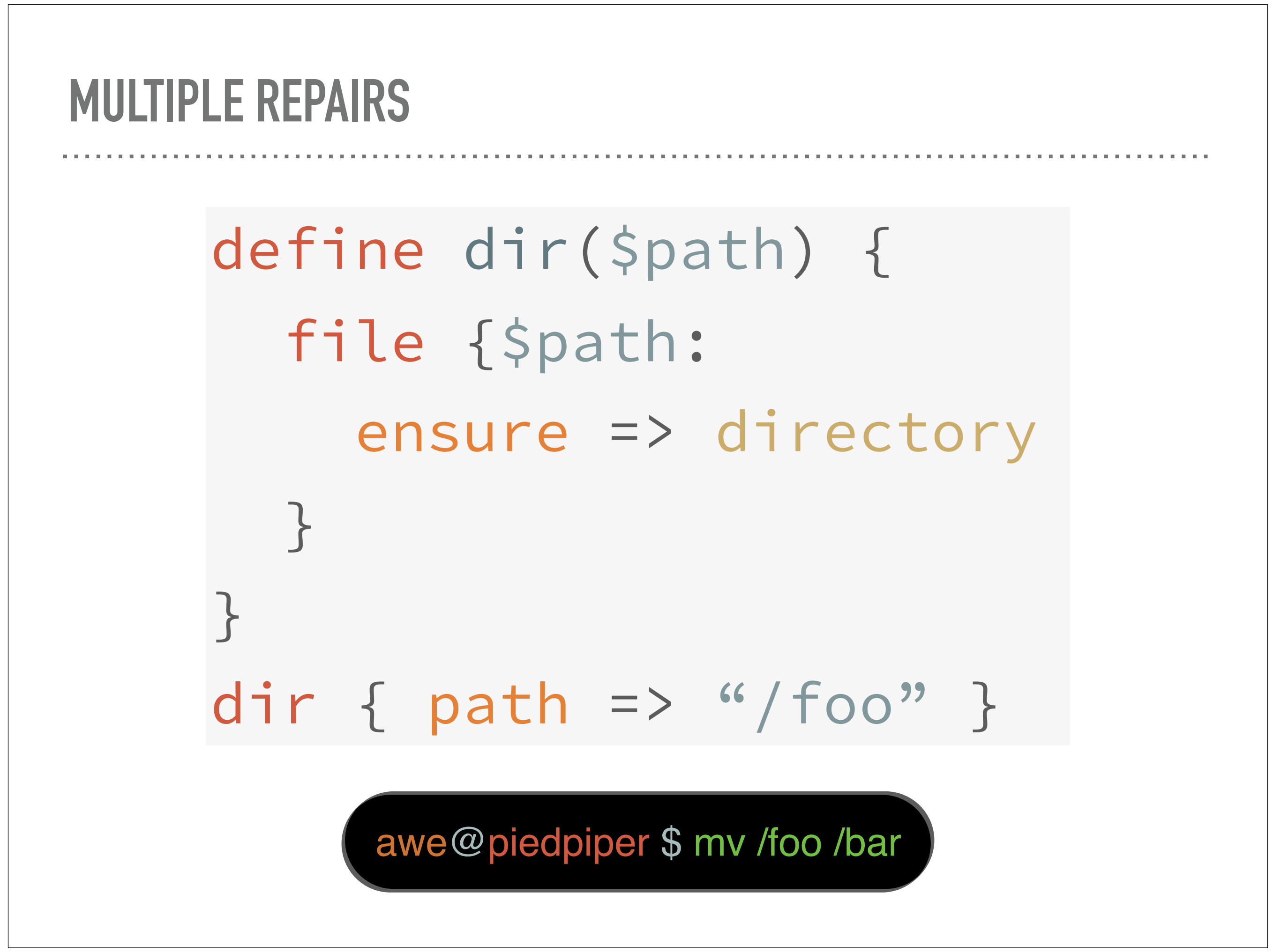
The obvious one is to update the path like so *click*
Here, we’ve changed the constant from “/foo” to “/bar”.

But we can also change the contents of the abstraction instead. *click*
Here, we’ve replaced an invocation of the variable $path with the constant “/foo”

Both changes are correct in that they preserve the change made from the shell, but the one that changes the constant is likely preferred.

We capture this via a ranking algorithm that assigns costs both to being a larger update and having more changes within an abstraction, and prefers the lowest cost updates.

click
In this case, our algorithm correctly ranks the obvious update as the better one.



Of course, there are often multiple possible repairs that will make the assertions true. To deal with this, we developed a procedure for ranking repairs.

Consider the simple example shown here with a dir abstraction creating a directory, and a single instantiation for the path “/foo”.

If we use a command like *click* “mv /foo /bar”, there are actually two possible updates here.

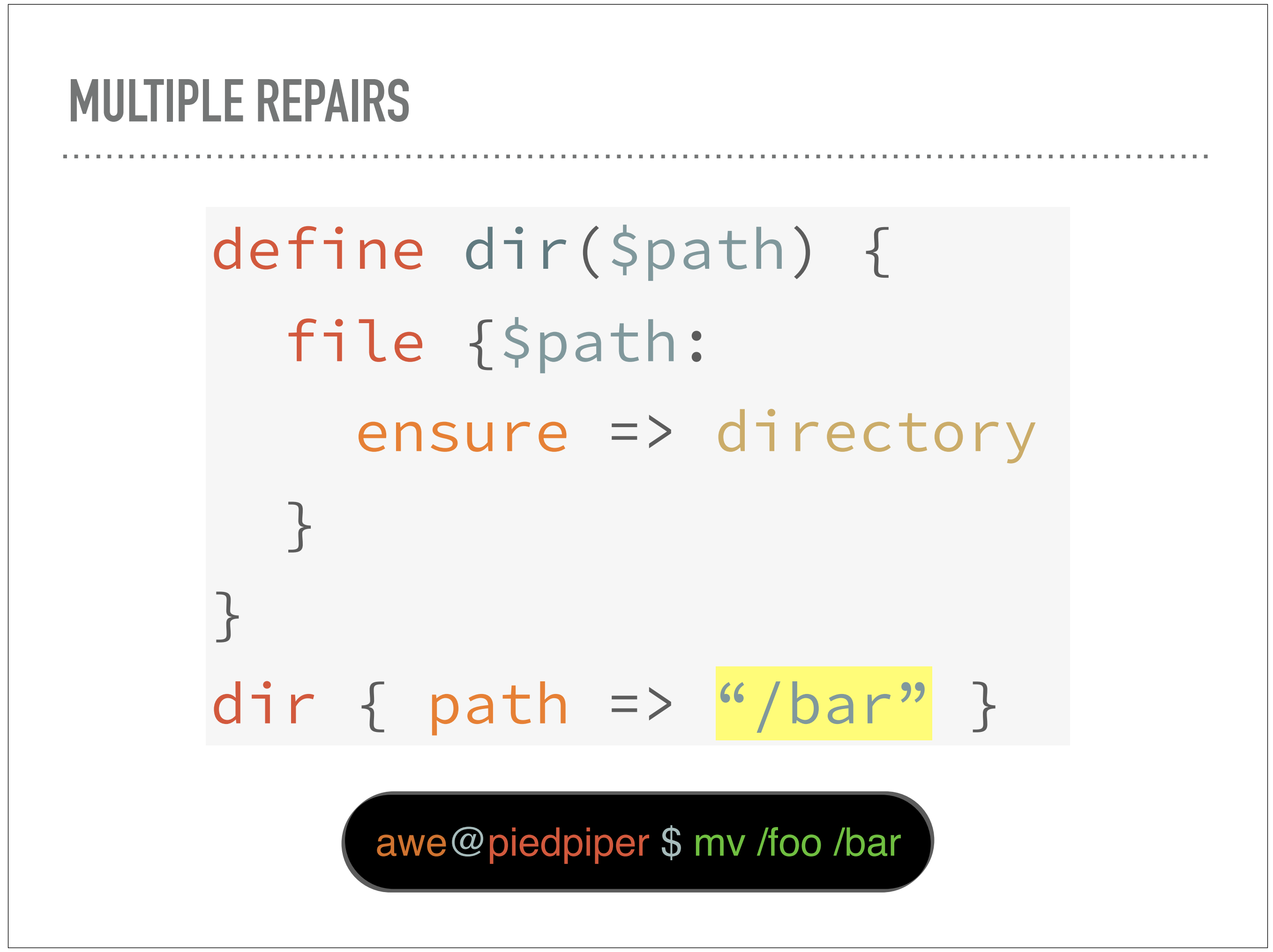
The obvious one is to update the path like so *click*
Here, we’ve changed the constant from “/foo” to “/bar”.

But we can also change the contents of the abstraction instead. *click*
Here, we’ve replaced an invocation of the variable $path with the constant “/foo”

Both changes are correct in that they preserve the change made from the shell, but the one that changes the constant is likely preferred.

We capture this via a ranking algorithm that assigns costs both to being a larger update and having more changes within an abstraction, and prefers the lowest cost updates.

click
In this case, our algorithm correctly ranks the obvious update as the better one.



Of course, there are often multiple possible repairs that will make the assertions true. To deal with this, we developed a procedure for ranking repairs.

Consider the simple example shown here with a dir abstraction creating a directory, and a single instantiation for the path “/foo”.

If we use a command like *click* “mv /foo /bar”, there are actually two possible updates here.

The obvious one is to update the path like so *click*
Here, we’ve changed the constant from “/foo” to “/bar”.

But we can also change the contents of the abstraction instead. *click*
Here, we’ve replaced an invocation of the variable $path with the constant “/foo”

Both changes are correct in that they preserve the change made from the shell, but the one that changes the constant is likely preferred.

We capture this via a ranking algorithm that assigns costs both to being a larger update and having more changes within an abstraction, and prefers the lowest cost updates.

click
In this case, our algorithm correctly ranks the obvious update as the better one.

MULTIPLE REPAIRS

```
define dir($path) {
  file {”/bar”:
    ensure => directory
  }
}
dir { path => “/foo” }
```

```
awe@piedpiper $ mv /foo /bar
```

Of course, there are often multiple possible repairs that will make the assertions true. To deal with this, we developed a procedure for ranking repairs.

Consider the simple example shown here with a dir abstraction creating a directory, and a single instantiation for the path “/foo”.

If we use a command like *click* “mv /foo /bar”, there are actually two possible updates here.

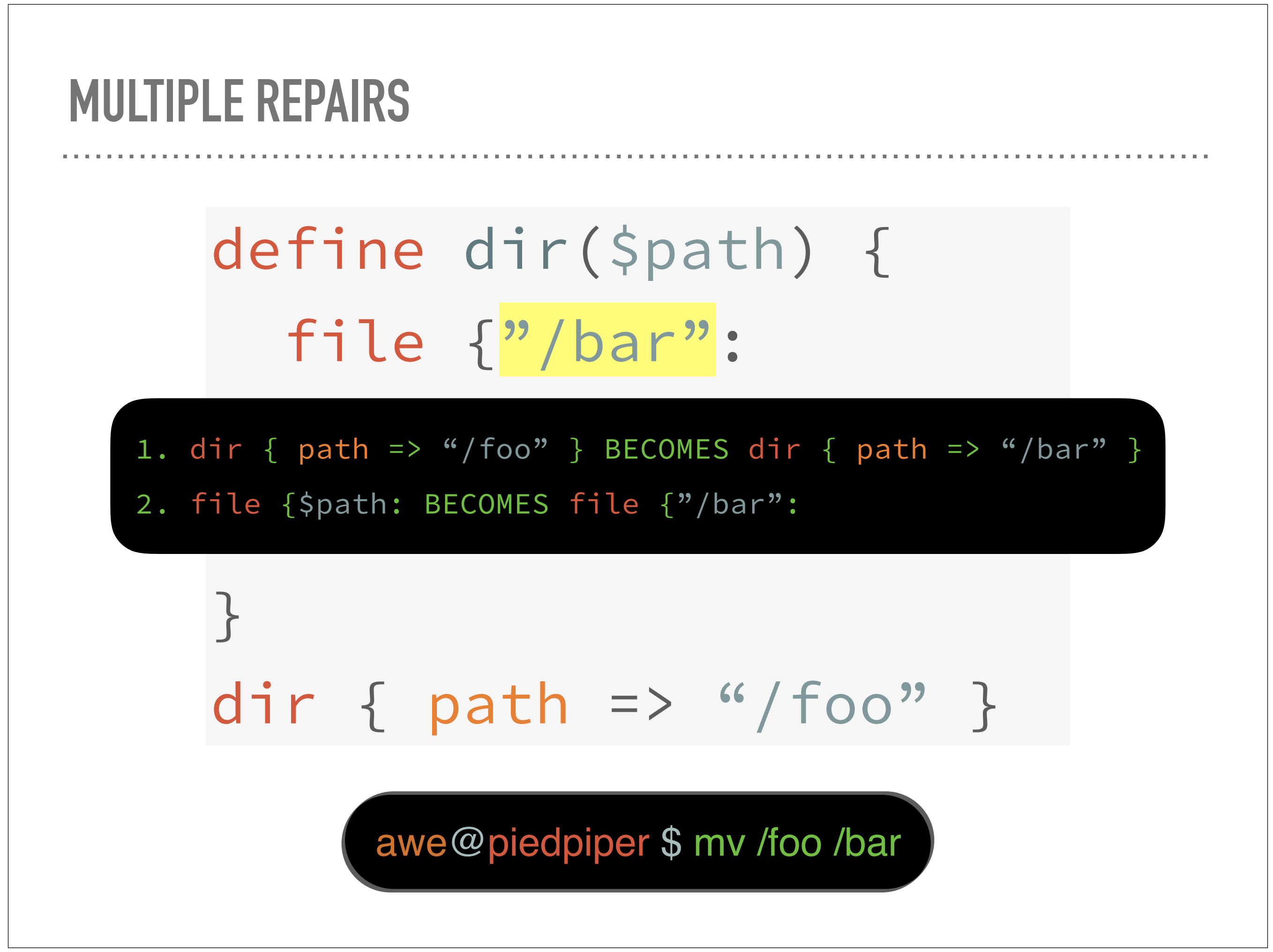
The obvious one is to update the path like so *click*
Here, we’ve changed the constant from “/foo” to “/bar”.

But we can also change the contents of the abstraction instead. *click*
Here, we’ve replaced an invocation of the variable $path with the constant “/foo”

Both changes are correct in that they preserve the change made from the shell, but the one that changes the constant is likely preferred.

We capture this via a ranking algorithm that assigns costs both to being a larger update and having more changes within an abstraction, and prefers the lowest cost updates.

click
In this case, our algorithm correctly ranks the obvious update as the better one.



Of course, there are often multiple possible repairs that will make the assertions true. To deal with this, we developed a procedure for ranking repairs.

Consider the simple example shown here with a dir abstraction creating a directory, and a single instantiation for the path "/foo".

If we use a command like *click* "mv /foo /bar", there are actually two possible updates here.

The obvious one is to update the path like so *click*
Here, we've changed the constant from "/foo" to "/bar".

But we can also change the contents of the abstraction instead. *click*
Here, we've replaced an invocation of the variable $path with the constant "/foo"

Both changes are correct in that they preserve the change made from the shell, but the one that changes the constant is likely preferred.

We capture this via a ranking algorithm that assigns costs both to being a larger update and having more changes within an abstraction, and prefers the lowest cost updates.

click
In this case, our algorithm correctly ranks the obvious update as the better one.

Atomic Expressions	
a ::= *str*	String
| *bool*	Boolean
| n	Integer
| **undef**	Undefined
| x	Variable reference
Expressions	
e ::= a	
| **file?**(a)	Test if a refers to a file
| **dir?**(a)	Test if a refers to a directory
| **exists?**(a)	Test if a refers to a file or directory
| **defined?**(a)	Test if not **undef**
| e_1 + e_2	String concatenation
| ...	Comparisons and boolean connectives
Statements	
c ::= **let** x = e	Variable declaration
| **if** (e) c_1 **else** c_2	Conditional
| { c_1; ⋯; c_n }	Block statement
| **chmod**(e_1, e_2)	Set permissions of e_1 to e_2
| **chown**(e_1, e_2)	Set owner of e_1 to e_2
| **mkdir**(e)	Create directory
| **write**(e_1, e_2)	Create file e_1 with contents e_2
| ...	Other file system operations
| **rlet** x = a **from** r	Let x be a, but can be repaired to r
| **assert**(e)	Assertion
Repair Spaces	
r ::= [a_1; ⋯; a_n]	Finite set of alternatives
| **str**	Any string or **undef**
| **int**(n)	Any n-bit number or **undef**

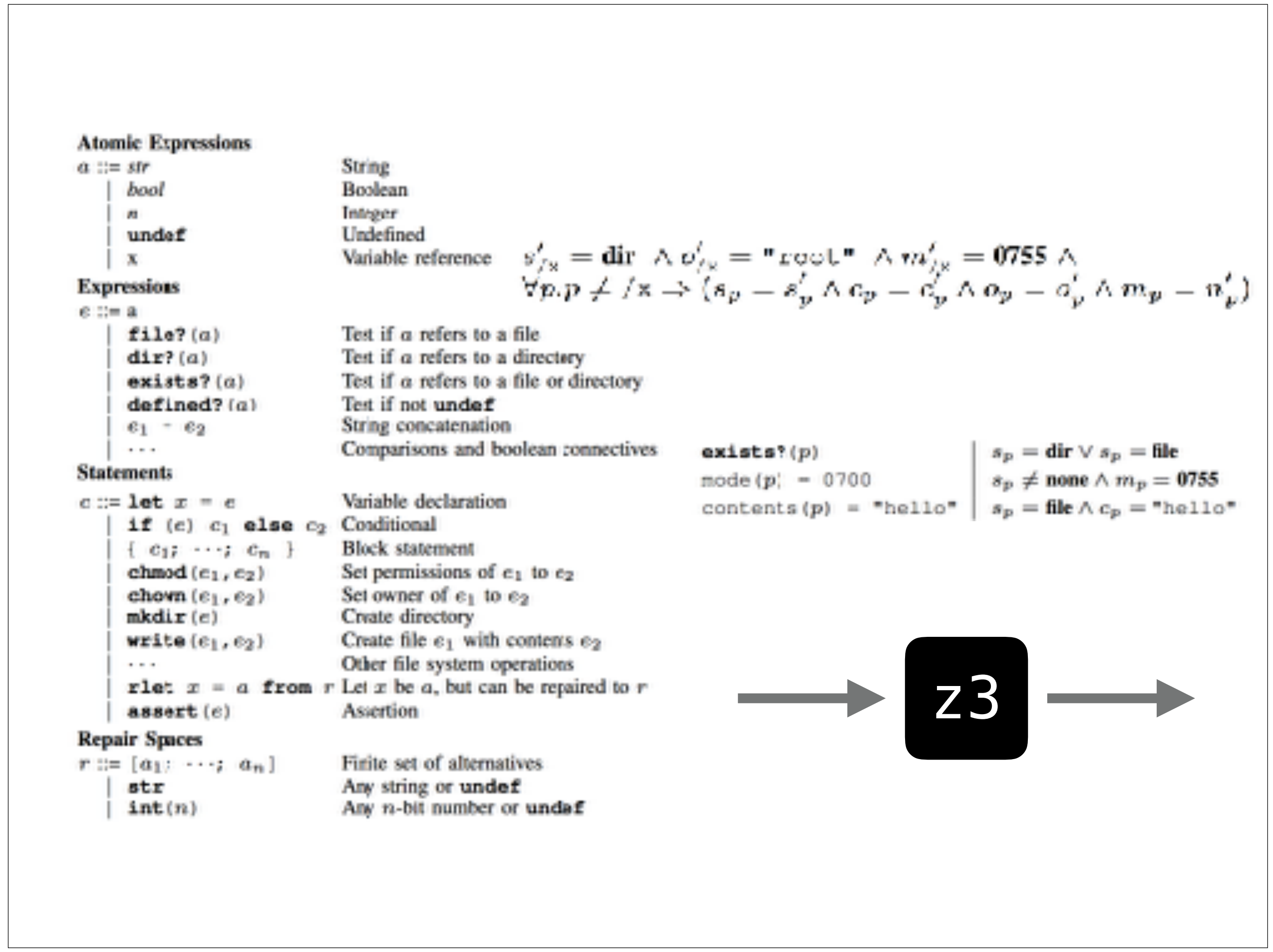
$$s'_{/x} = \mathbf{dir} \wedge o'_{/x} = \texttt{"root"} \wedge m'_{/x} = \mathbf{0755} \wedge$$
$$\forall p. p \neq /x \rightarrow (s_p = s'_p \wedge c_p = c'_p \wedge o_p = o'_p \wedge m_p = m'_p)$$

exists?(p)	$s_p = \mathbf{dir} \vee s_p = \mathbf{file}$
mode(p) = 0700	$s_p \neq \mathbf{none} \wedge m_p = \mathbf{0755}$
contents(p) = "hello"	$s_p = \mathbf{file} \wedge c_p = \texttt{"hello"}$

We have a paper currently in submission. You can ask me about a draft if you're interested.

In it, we show the syntax of Delta-P and describe the translation of Puppet to Delta-P. We also describe the process of compiling Delta-P to logical formulas for Z3 and how satisfying models from the solver are converted to repairs.

But we'll leave that to the paper.



We have a paper currently in submission. You can ask me about a draft if you're interested.

In it, we show the syntax of Delta-P and describe the translation of Puppet to Delta-P. We also describe the process of compiling Delta-P to logical formulas for Z3 and how satisfying models from the solver are converted to repairs.

But we'll leave that to the paper.

EVALUATING TORTOISE

Benchmark	# of resources	# of repair scenarios	Tortoise runtime (ms)	Average repair rank
amavis	6	1	25	1.00
bind	6	3	21	1.60
clamav	6	2	23	3.50
hosting	19	1	26	1.00
irc	18	1	292	1.00
jpa	10	1	21	1.00
logstash	14	6	48	1.00
monit	7	4	25	1.00
nginx	9	4	27	1.00
ntp	4	3	18	1.33
powerdns	5	7	39	1.43
rsyslog	7	4	129	1.25
xinetd	4	5	1,970	1.20
Total	115	42	205	1.31

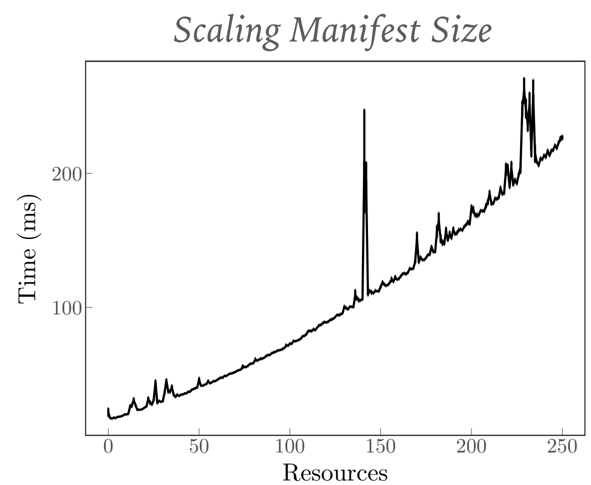
To get a sense of Tortoise's practicality, we evaluated it on a suite of real world manifests used in prior work with Puppet.

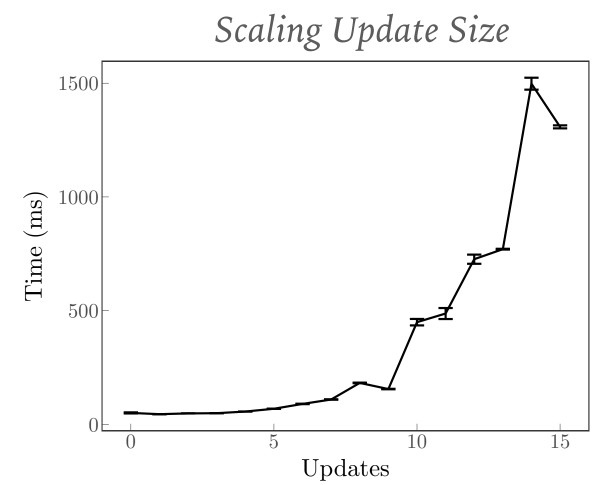
For each of the thirteen benchmarks, we came up with a number of possible repairs, totaling to 42. We then ran trials for each case, and presented the user with the set of Tortoise repairs in random order. We used this to compute the average rank of the correct repair by recording the rank Tortoise assigned to the repair selected by the user.

Average runtime: 205 ms
Average rank: 1.31

This means that Tortoise's highest rank repair is typically the correct one.

TORTOISE SCALABILITY





We also looked at how Tortoise works in the extremes using scalability benchmarks.

On the left, you can see runtime as we scale the number of resources in the manifest (leaving the update size constant). In practice, manifests tend not to grow beyond this size, and so performance is reasonable.

On the right, you can see runtime as we scale the size of the updates. This appears to be exponential. This is expected because of our use of SMT solving. However, it's not a problem because we can split updates into batches that are small enough to be solved quickly.

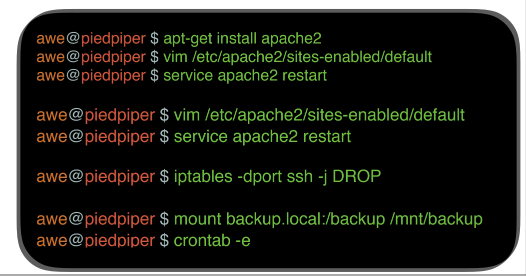
click In conclusion, we presented ICR

click It's sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we've bridged that gap.

SUMMARY

➤ Imperative Configuration Repair



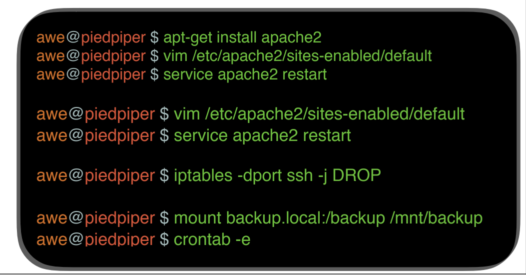
click In conclusion, we presented ICR

click It's sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we've bridged that gap.

SUMMARY

- Imperative Configuration Repair
    - Sound: All changes made via the shell are preserved



click In conclusion, we presented ICR

click It's sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we've bridged that gap.

SUMMARY

- Imperative Configuration Repair
  - Sound: All changes made via the shell are preserved
  - Maintainable: Structure and abstraction is preserved

```
awe@piedpiper $ apt-get install apache2
awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ iptables -dport ssh -j DROP

awe@piedpiper $ mount backup.local:/backup /mnt/backup
awe@piedpiper $ crontab -e
```

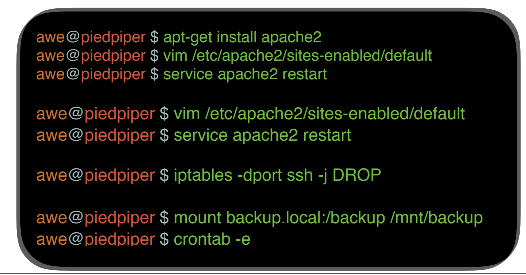
click In conclusion, we presented ICR

click It’s sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we’ve bridged that gap.

SUMMARY

- Imperative Configuration Repair
  - Sound: All changes made via the shell are preserved
  - Maintainable: Structure and abstraction is preserved
  - Ranked: Multiple possible repairs are ranked



click In conclusion, we presented ICR

click It's sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we've bridged that gap.

SUMMARY

- Imperative Configuration Repair
  - Sound: All changes made via the shell are preserved
  - Maintainable: Structure and abstraction is preserved
  - Ranked: Multiple possible repairs are ranked
  - Unrestricted: Works with all existing shells

puppet

```
awe@piedpiper $ apt-get install apache2
awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ iptables -dport ssh -j DROP

awe@piedpiper $ mount backup.local:/backup /mnt/backup
awe@piedpiper $ crontab -e
```

click In conclusion, we presented ICR

click It’s sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we’ve bridged that gap.

SUMMARY

- Imperative Configuration Repair
  - Sound: All changes made via the shell are preserved
  - Maintainable: Structure and abstraction is preserved
  - Ranked: Multiple possible repairs are ranked
  - Unrestricted: Works with all existing shells

```
awe@piedpiper $ apt-get install apache2
awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ vim /etc/apache2/sites-enabled/default
awe@piedpiper $ service apache2 restart

awe@piedpiper $ iptables -dport ssh -j DROP

awe@piedpiper $ mount backup.local:/backup /mnt/backup
awe@piedpiper $ crontab -e
```

click In conclusion, we presented ICR

click It’s sound meaning all changes made from the shell are preserved
click Maintainable meaning it preserves the structure and abstractions from the manifest
click Ranked meaning when multiple updates are possible, it ranks them for the user
click And unrestricted meaning it allows you to use all existing shells

There used to be a gap between Puppet manifests and the shell, but *click* thanks to Tortoise and ICR, we’ve bridged that gap.