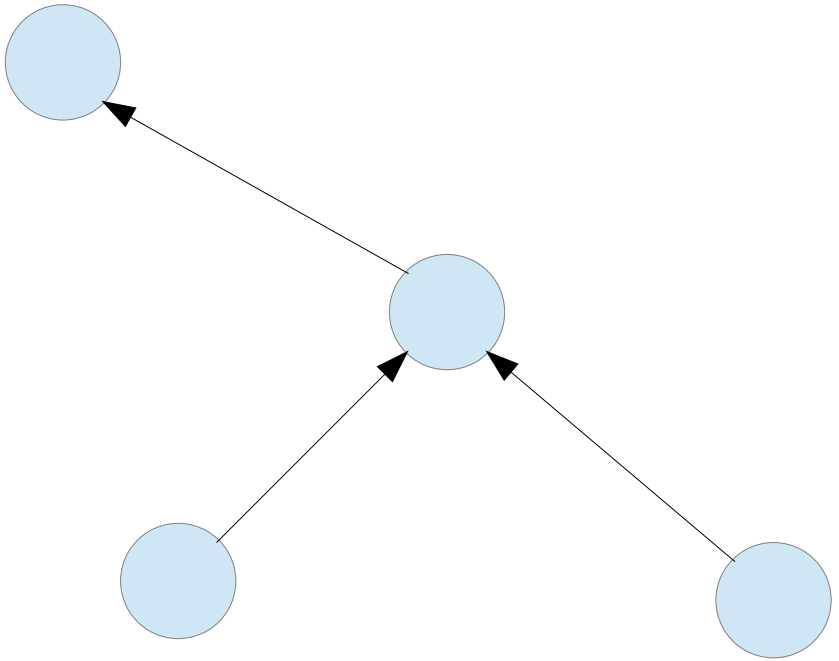
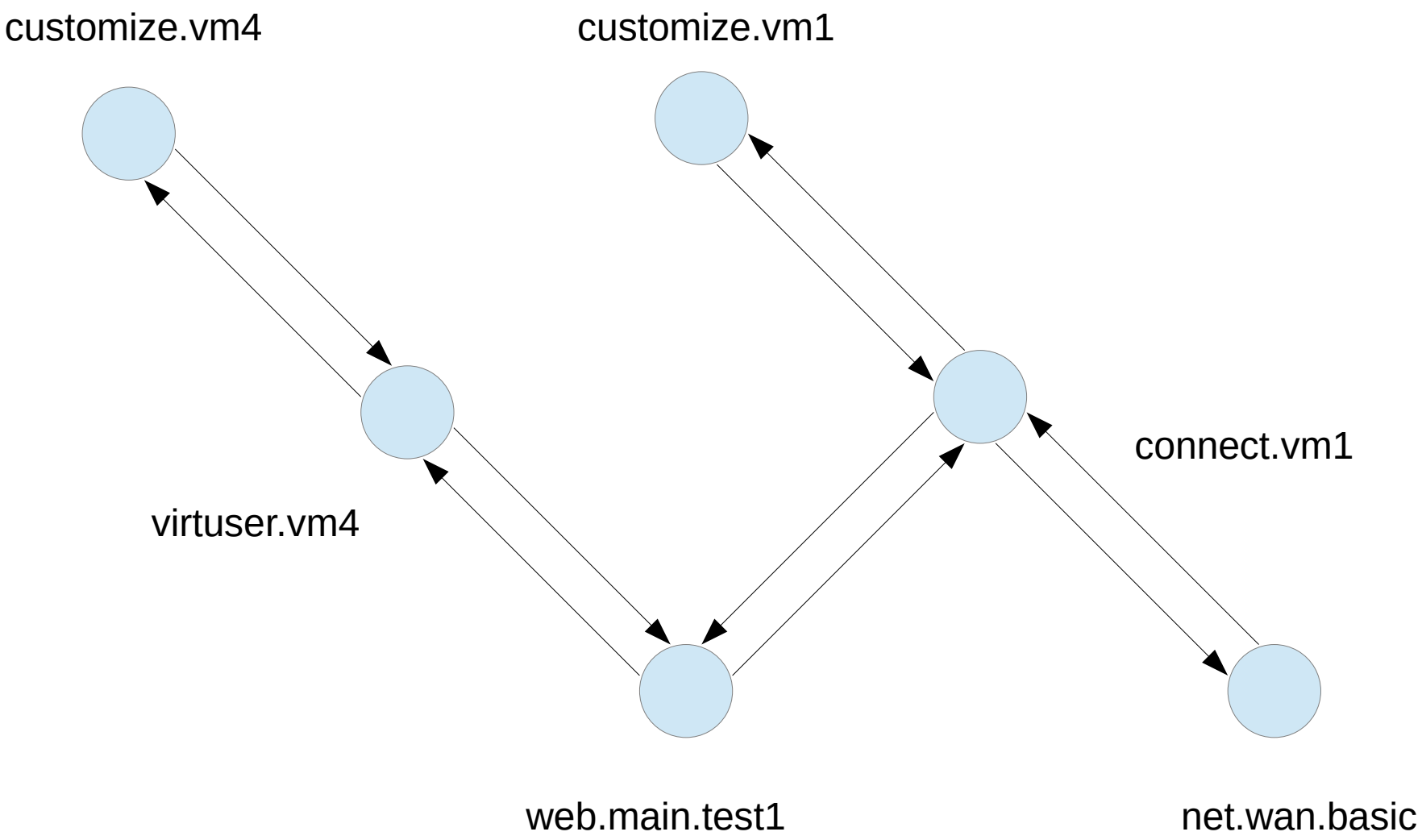
Cartesian graph traversal algorithm



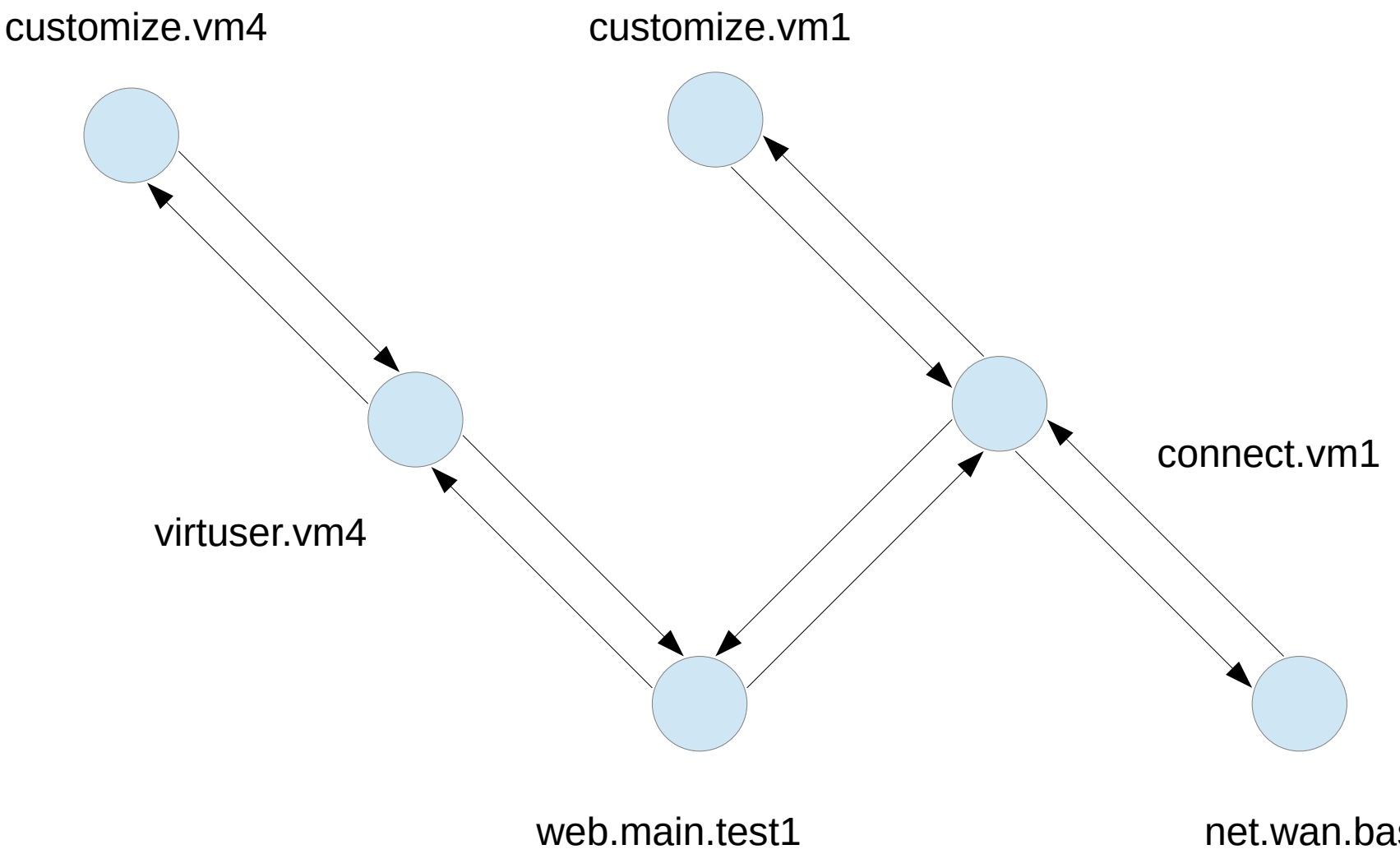
Let's take a brief look of how the algorithm used for automated setup and resolution of test dependencies works. It is the core algorithm of our test suite.

**** Please keep in mind that in order to understand the algorithm you need to understand the graph structure behind it which is explained in detail in the document accompanying this one.*
If you do so then I promise this will be a lot of fun.



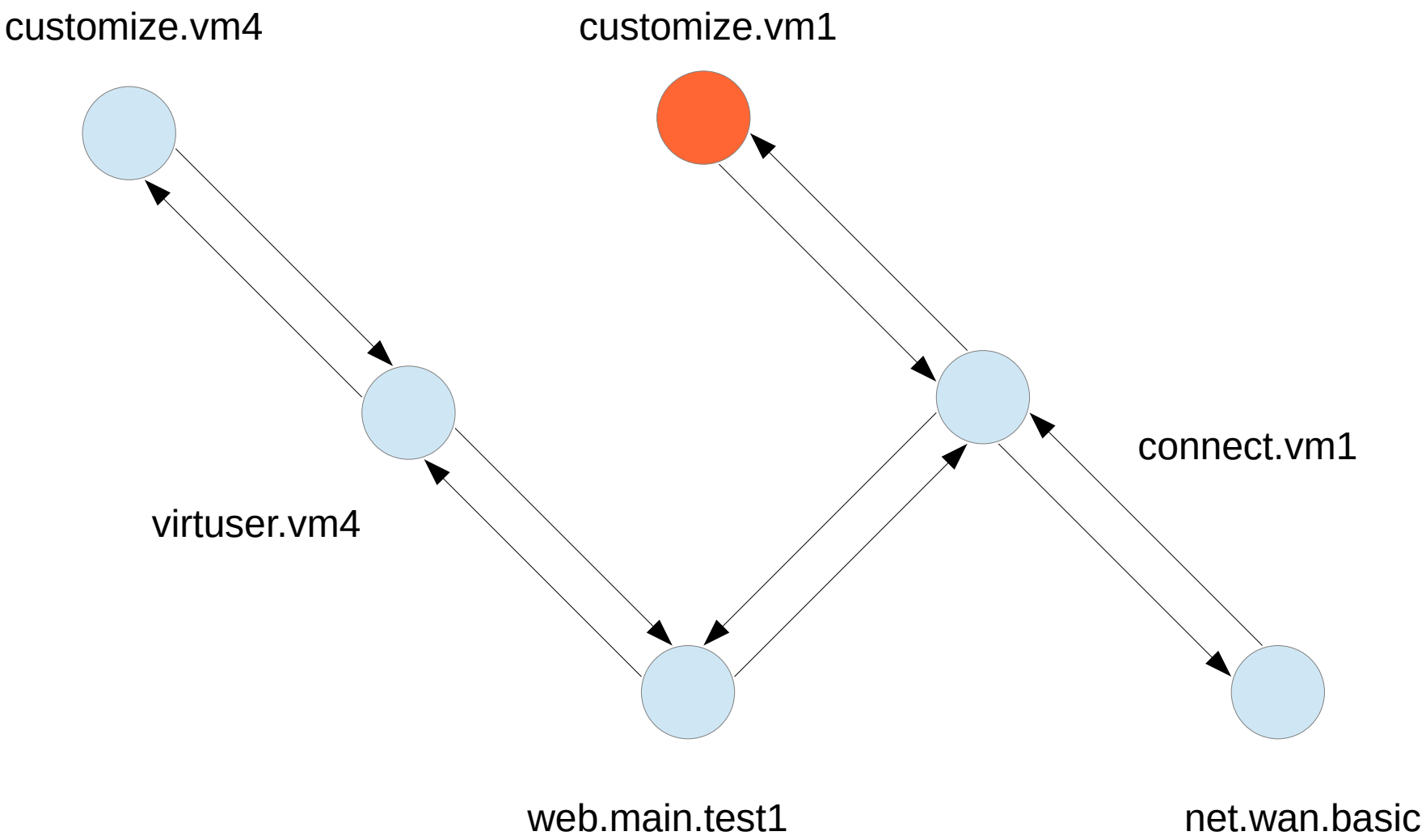
Firstly, let's draw a circle for each test node.
Secondly, let's connect each test with an upward arrow to a test it depends on.
Lastly, let's connect each test with a downward arrow to a test depending on it.

Each test is then aware of the arrows that originate from it. No test is aware of the arrows that point to it. This is important for the implementation of the algorithm where we use only pointers to other nodes divided as pointers to parents and pointers to children.



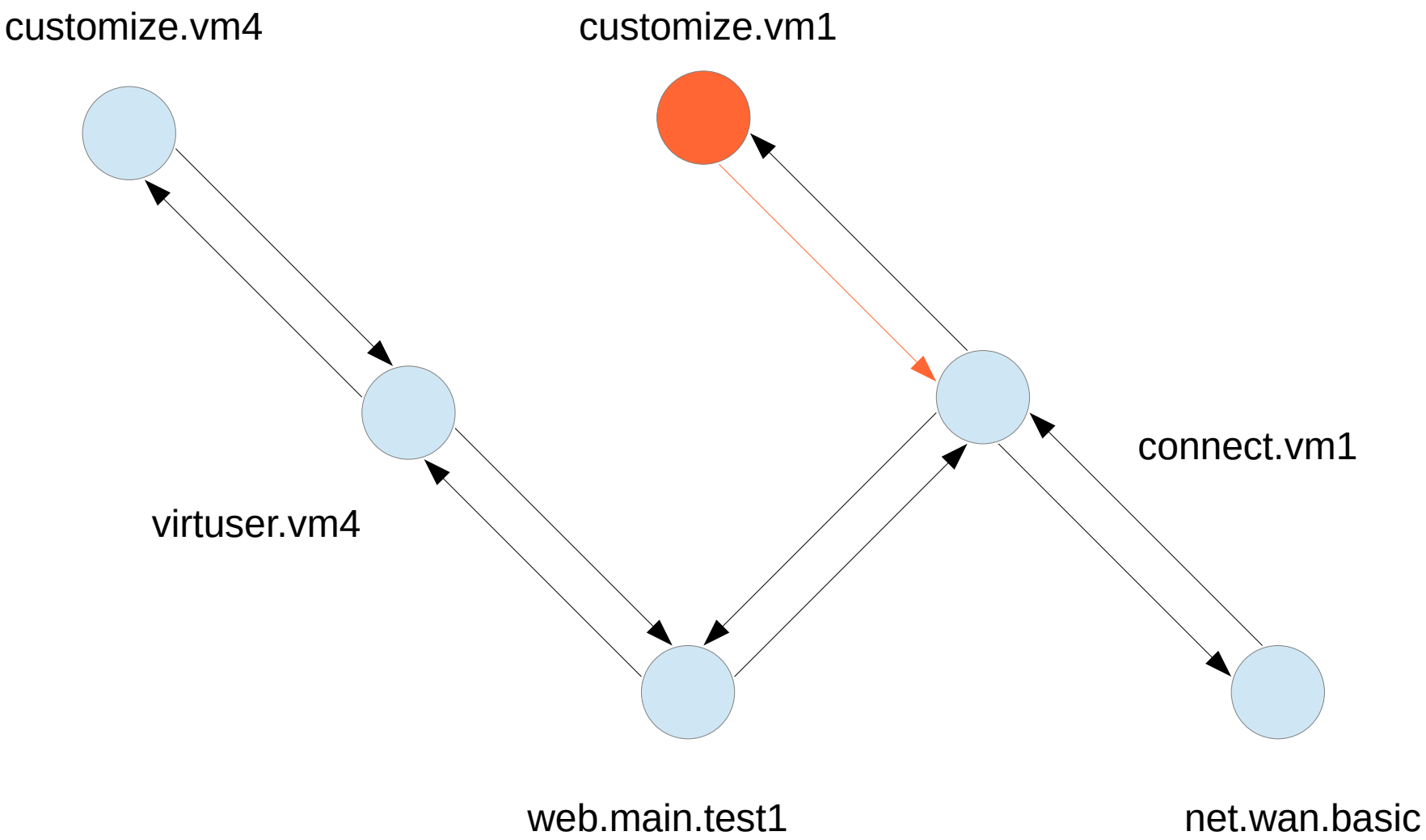
Here we take a few tests interconnected in this way giving them some example names. To differentiate among repeating tests but acting on different test objects (vms), we will append ".vmX". The ones that don't have this appended are using multiple vms, e.g. *web.main.test1* uses *vm1* as well as *vm4* which can be seen by the dependencies.

Let's start from customize.*vm1* as our root test node...
//to simplify the default start from an actual root node which is analogical (in the last slides)



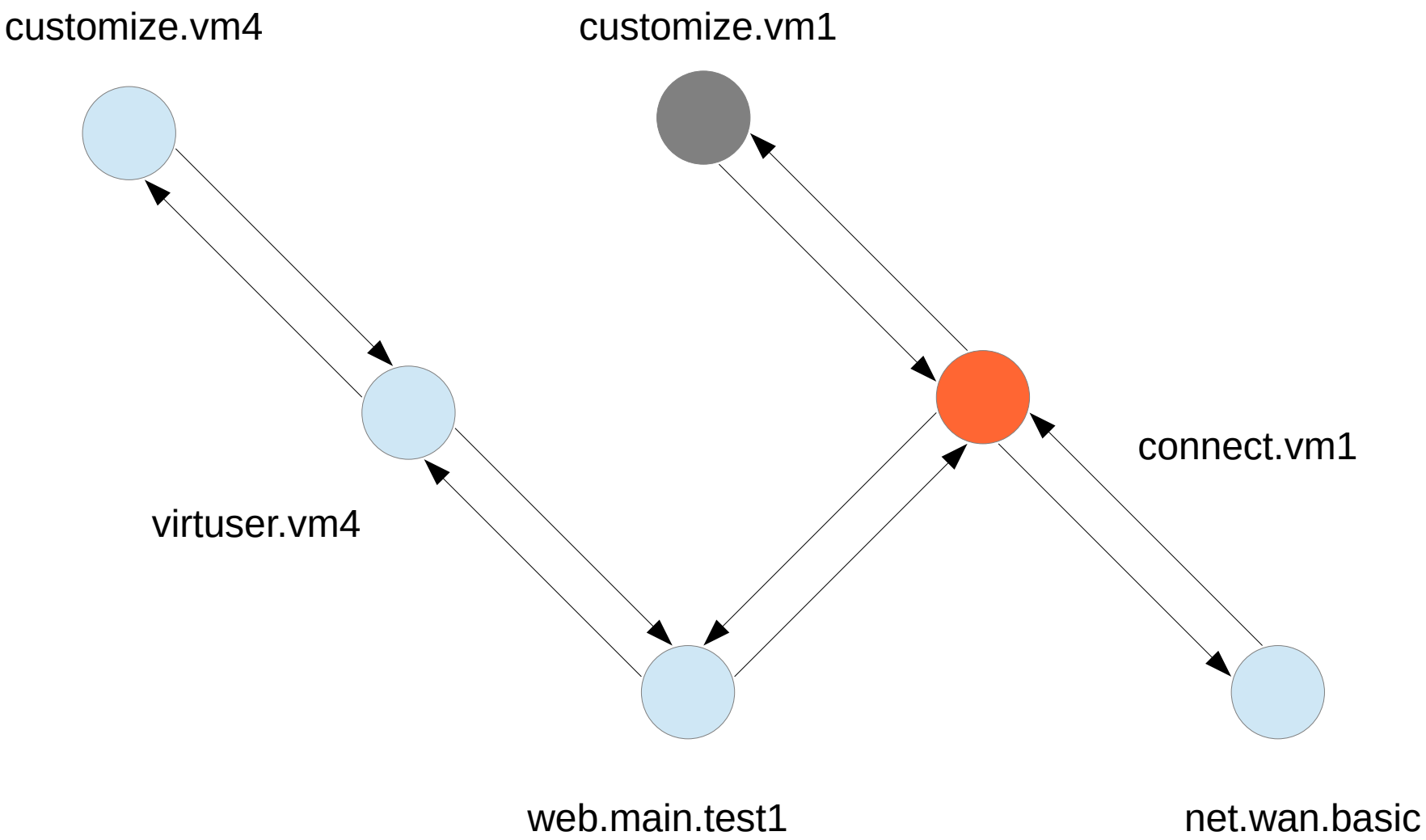
There are three qualities that affect our decision about customize.vm1:

- I**s it setup ready** = is the number of parent connections equal to 0
- **Is it cleanup ready** = is number of child connections equal to 0
- **Is it reusable** = does it create end states that can be reused so that we don't need to run it each time and if so was it run at least once so that the states are already available



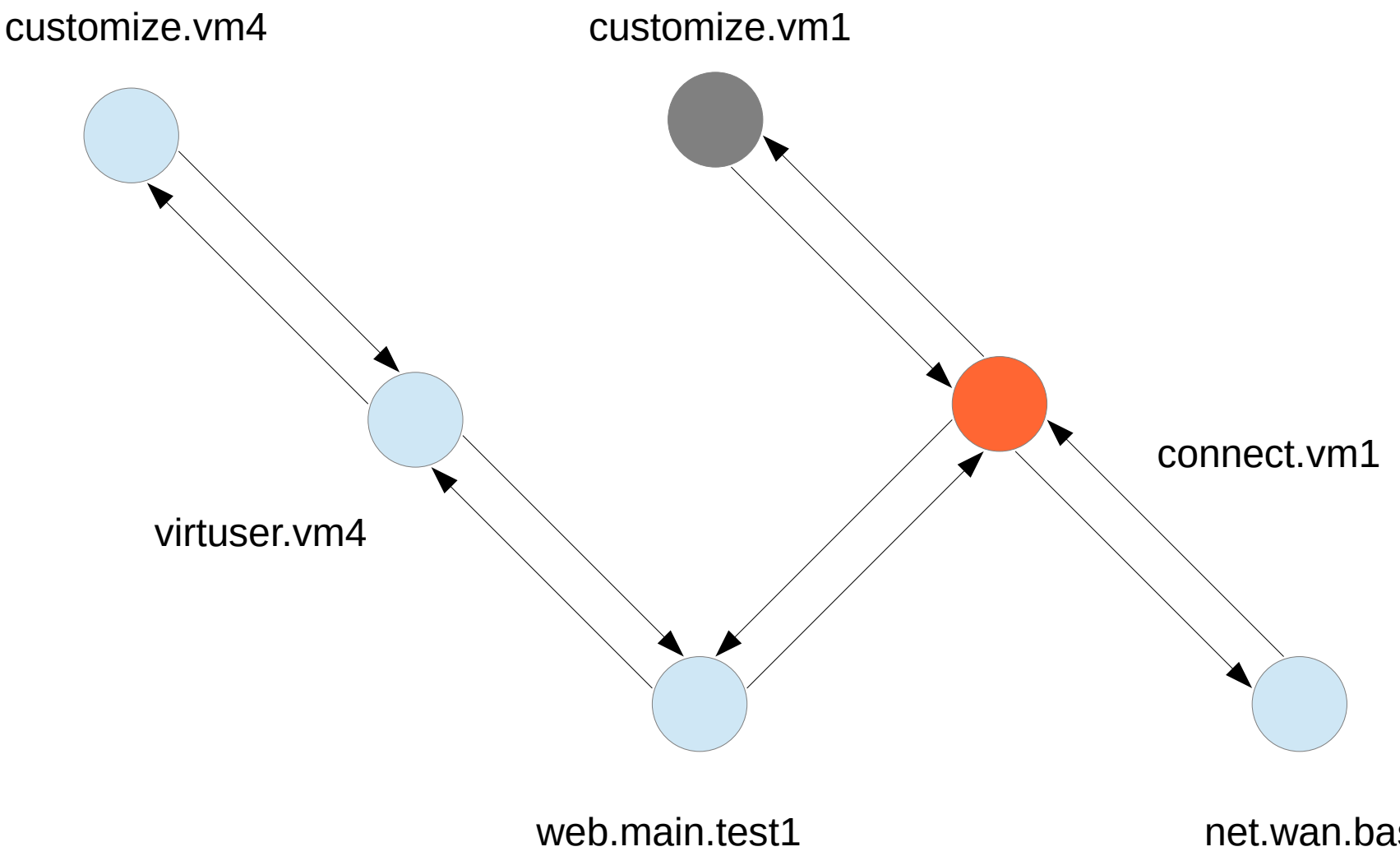
The conclusions are:

- I**s it setup ready** – True (always true for root state since it doesn't need setup by definition)
- **Is it cleanup ready** – False (one connection pending so no cleanup yet)
- **Is it reusable** – that depends on scanning for availability after previous runs, let's say it was run (True)



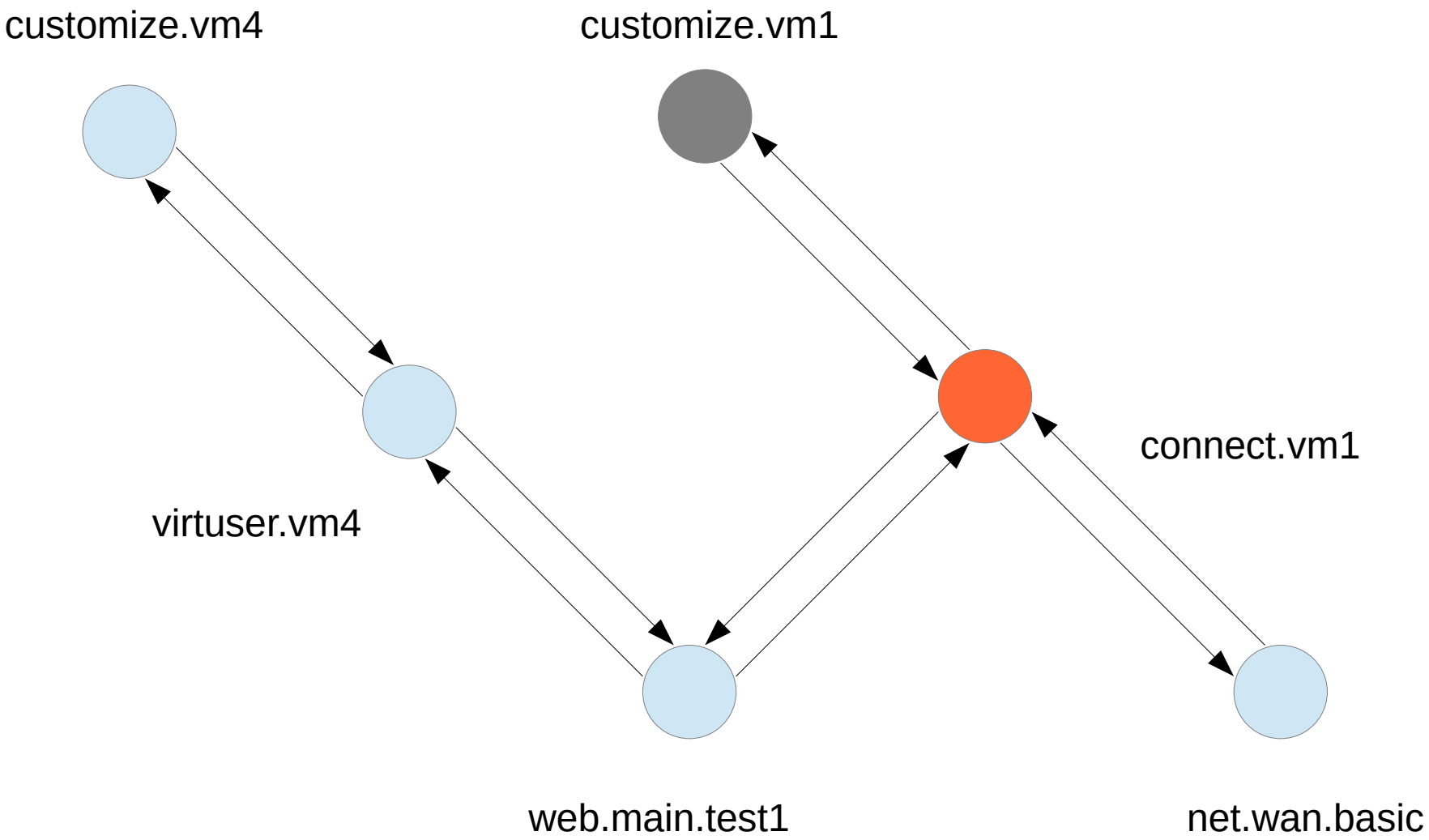
Moving to the next test node consists of two steps:

1) Push the node that is left behind to a stack of all visited nodes that are not cleanup ready
2) Use the path to the root that this stack represents to determine whether you are moving down or up



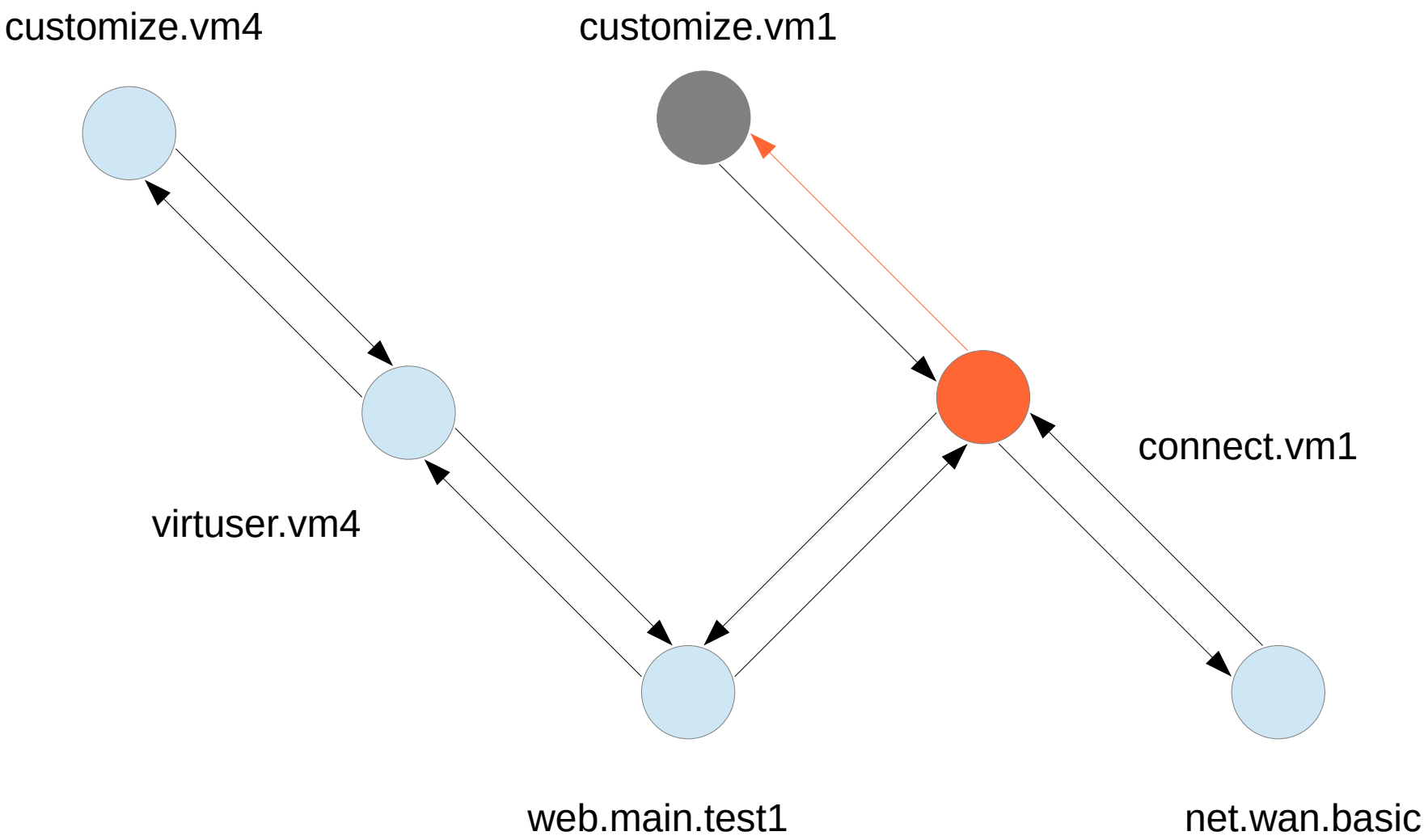
The direction to determine the next action (moving down or up) as is a matter of a simple check:

1) If the previous node (in the stack) is one of this node's parents, direction is down
2) If it is one of this node's children, direction is up (inverse traversal looking for missing setup)

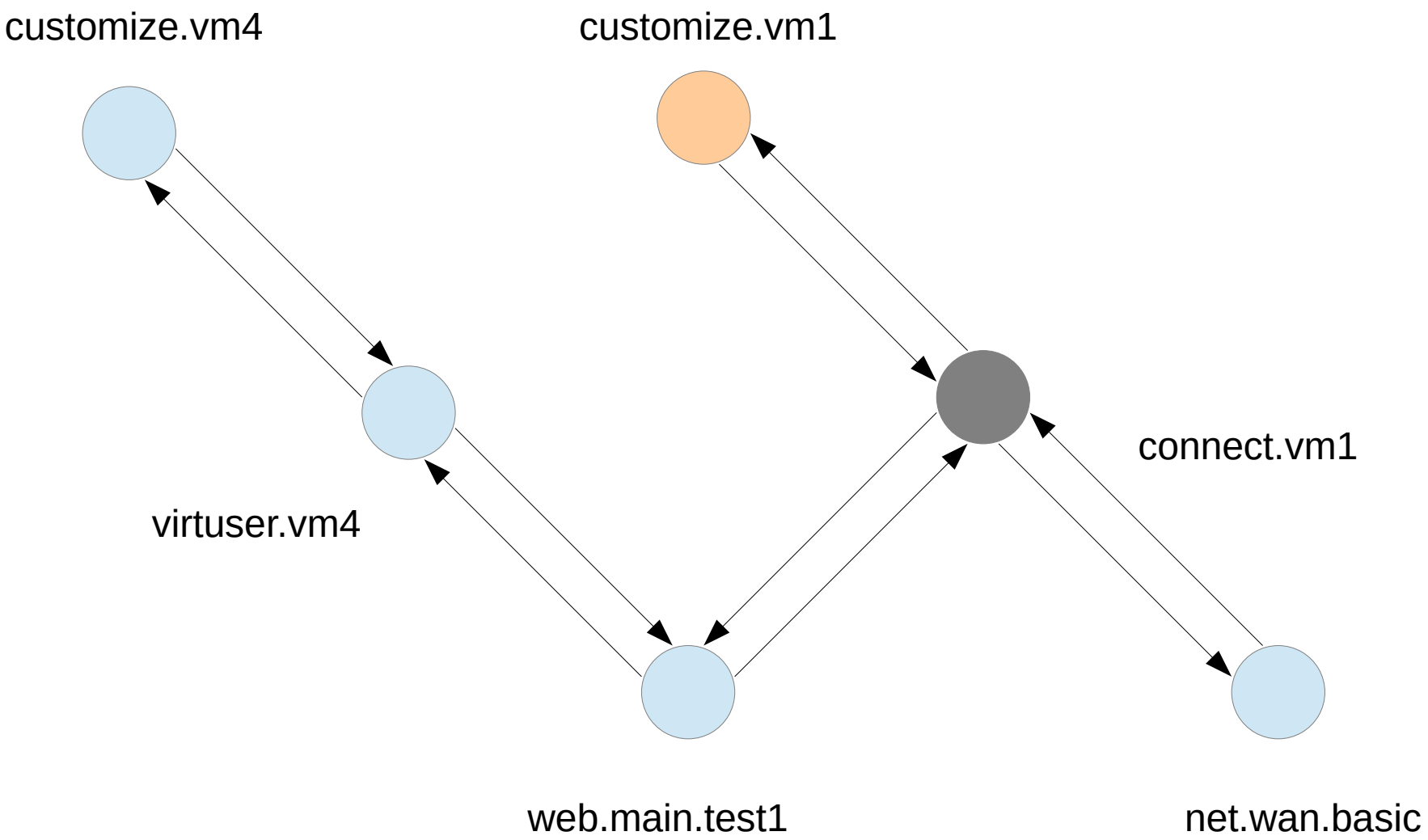


We can see above or even test with the previous condition that the direction is down, so we continue with the previously showed decision process:

- I**s it setup ready** – False (there is one arrow to parent, i.e. setup to check for availability)
- **Is it cleanup ready** – False (two child connections pending so other tests wait to use this one)
- **Is it reusable** – depends on reusable state scanning, but let's say it wasn't run (False)

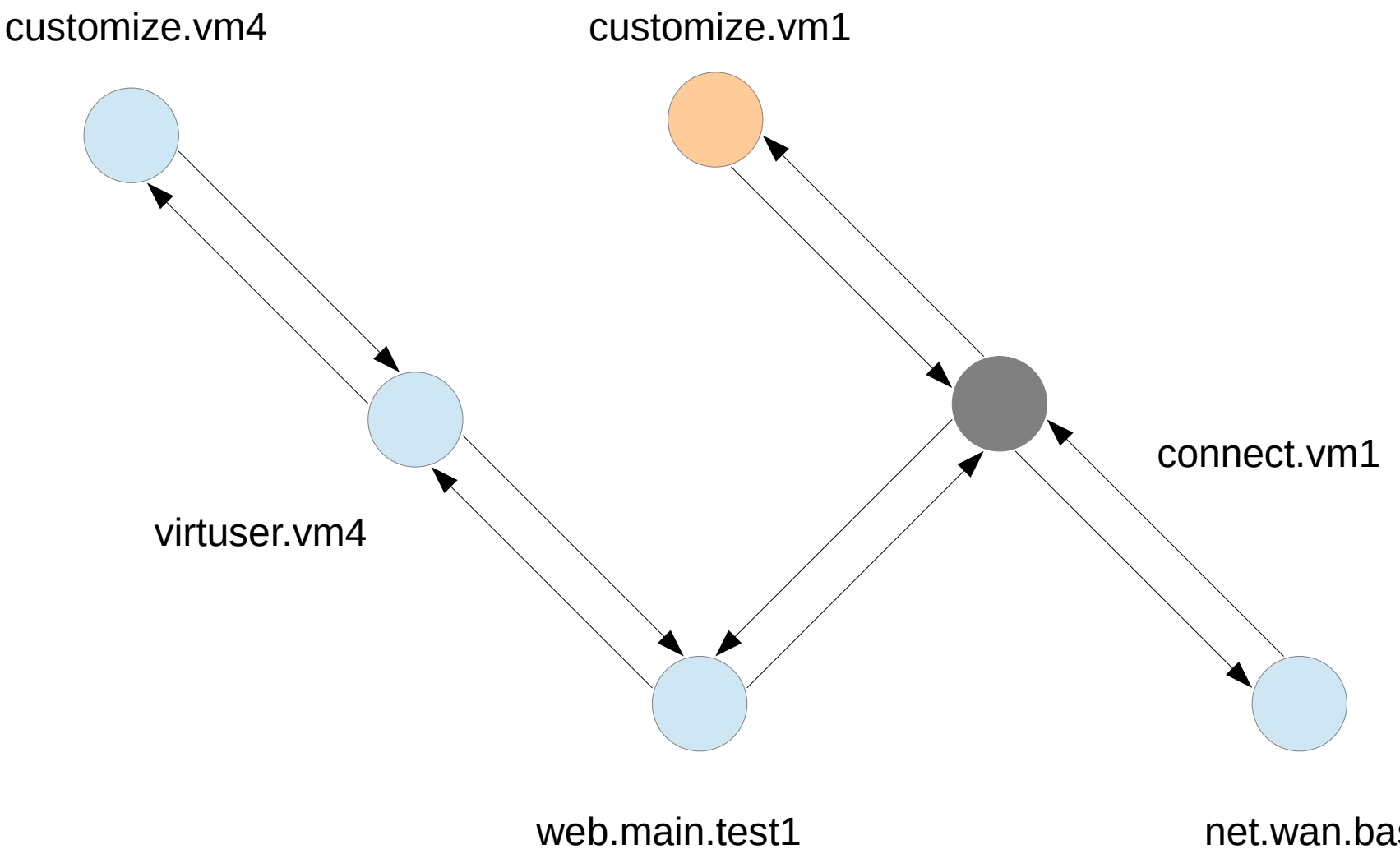


What is different now it that the node is not setup ready, so we can't run it or continue down. Instead, we prioritize the setup and reverse the direction to up.



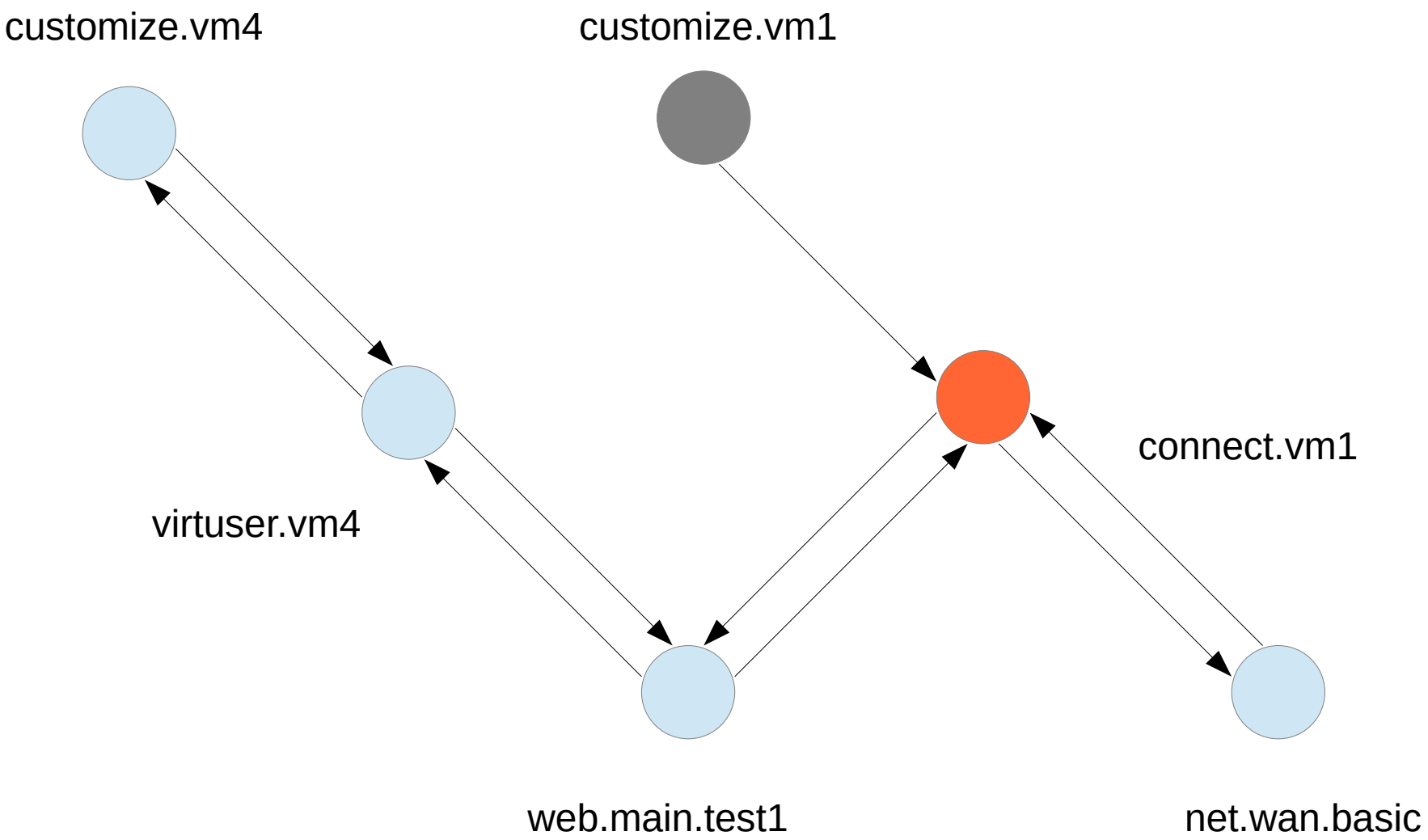
I picked light orange instead of orange only to remind you that the stack now contains three nodes (the current path) with repeating *customize.vm1*:

customize.vm1 -> connect.vm1 -> customize.vm1



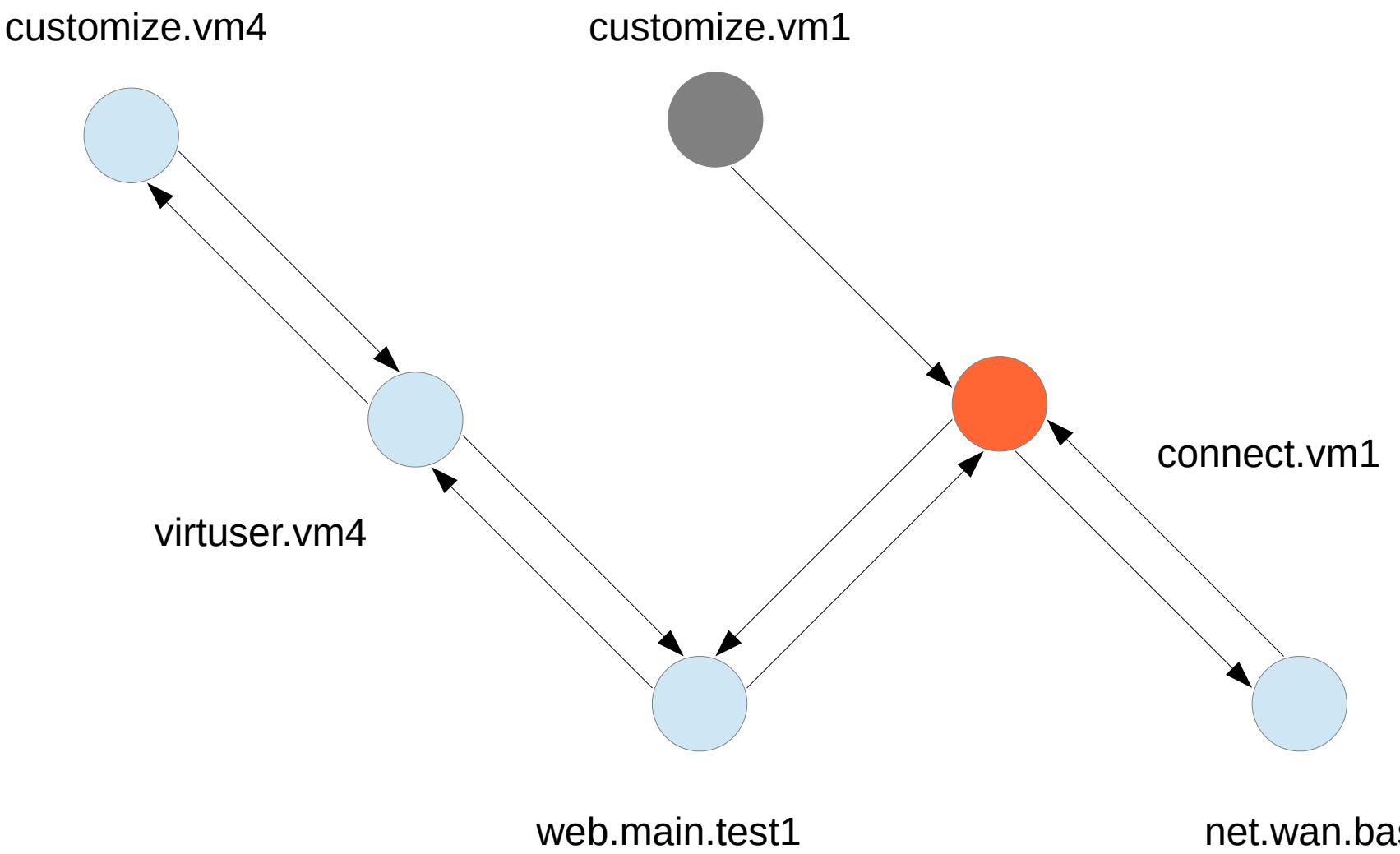
Of course we use this to detect the we have reversed direction and it is now up, so we make a different set of decisions based on the same three criteria:

- **If node is not setup ready** –> go to its next parent (just one such here) - inverse DFS
- **If node is setup ready** –> run it if not reusable or skip it if reusable, pop the stack and remove the arrow that led to here
- **Since customize.vm1 node is setup ready** –> let's observe the second case!



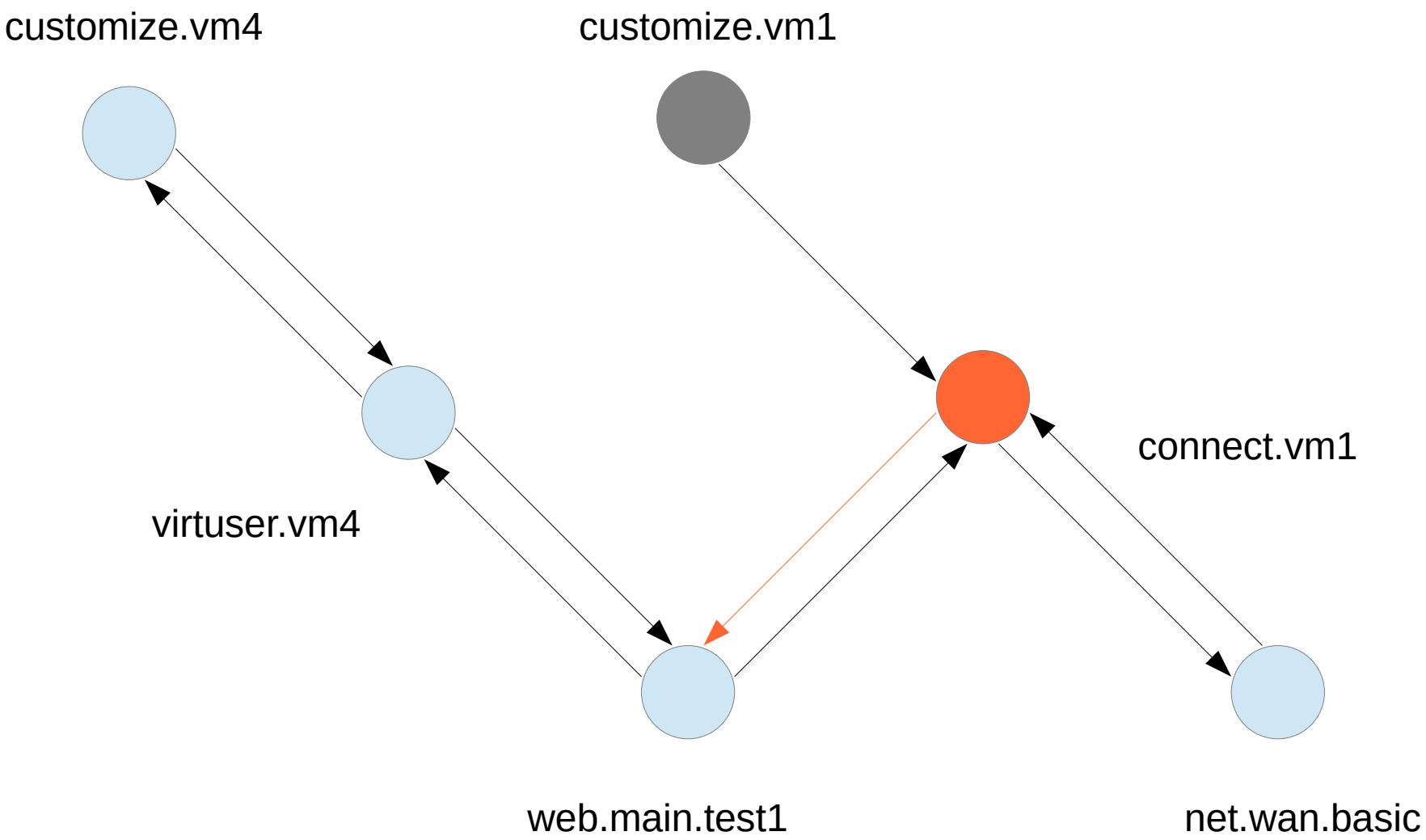
So as we just said, three things happen:

1) Run the test, however rather skip it since we said that it is *reusable*
2) Pop the stack, i.e. go to the last test before this one and make a step backward in the path
3) Remove the arrow from the previous test node to this one (or rather consider it visited)

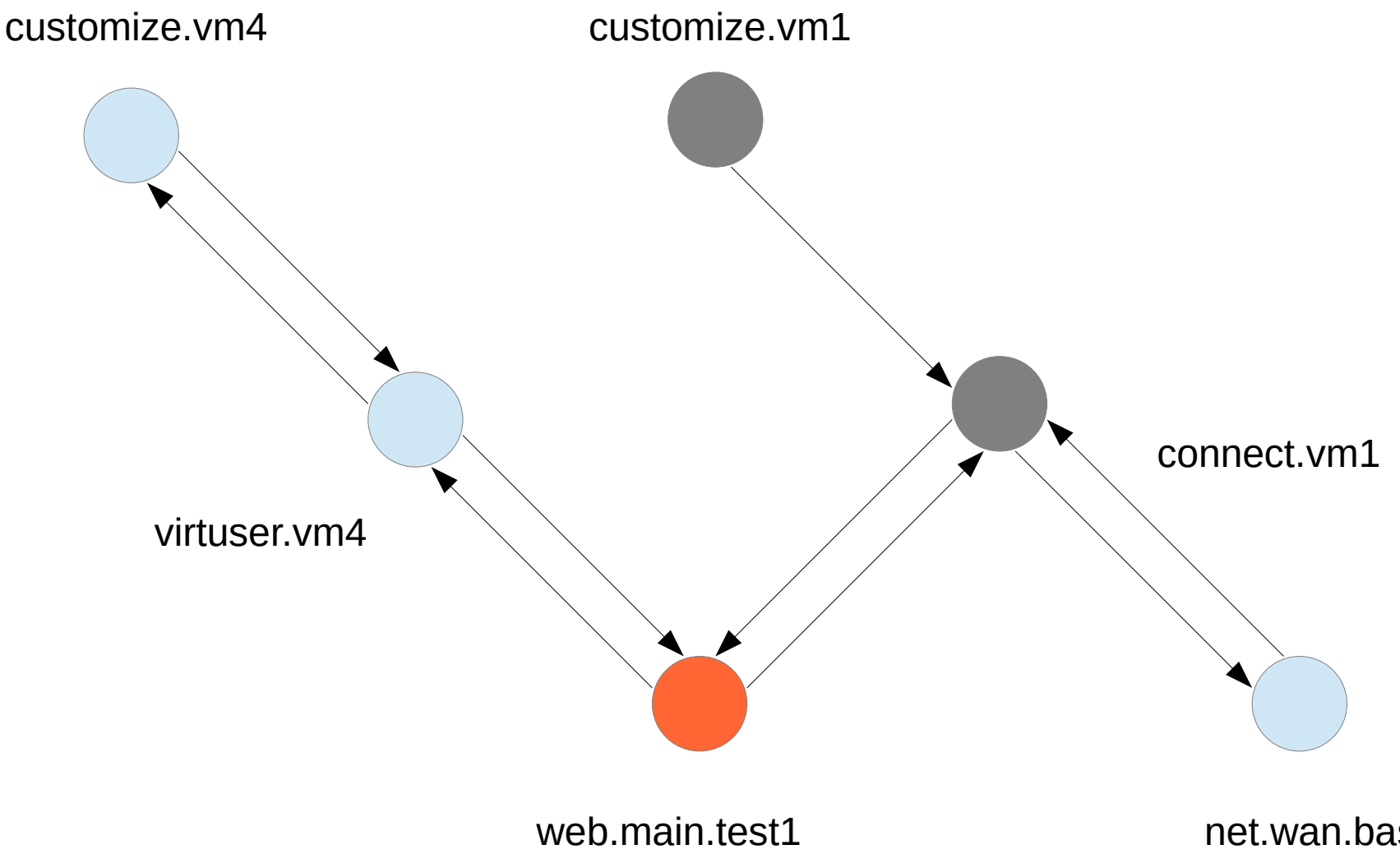


What now? Well, checking the direction now it seems we are going down, so our check for these three gives us updated results:

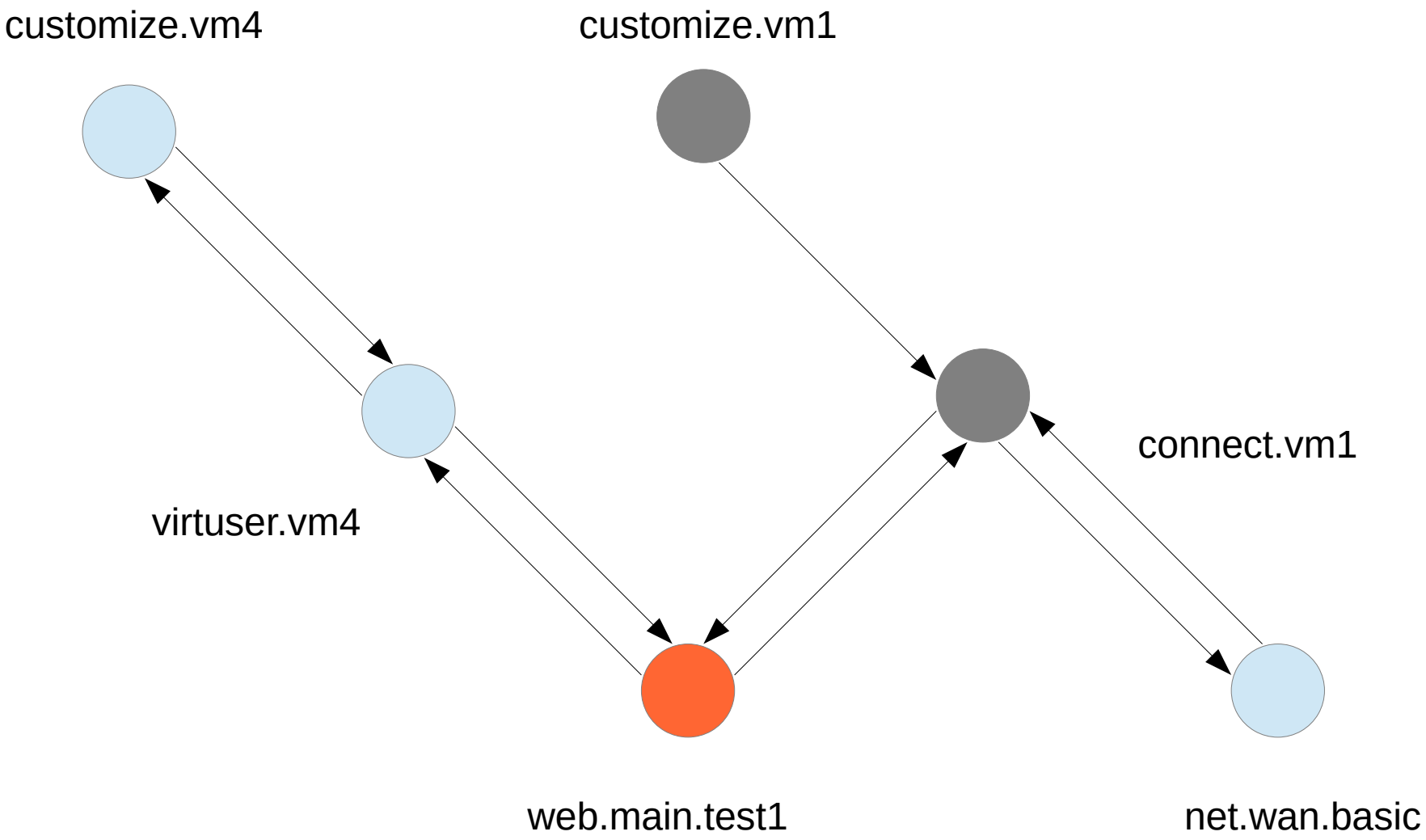
- I**s it setup ready** – True (there are no arrows to parent, i.e. all setup is available and can be run)
- **Is it cleanup ready** – False (two connections are still pending so other tests keep waiting)
- **Is it reusable** – we said it wasn't run (False) so it's time to run it!



Decisions are as expected – we run the connect.vm1 and go to the child with the lowest id count or any other scheduling criterion among same level children

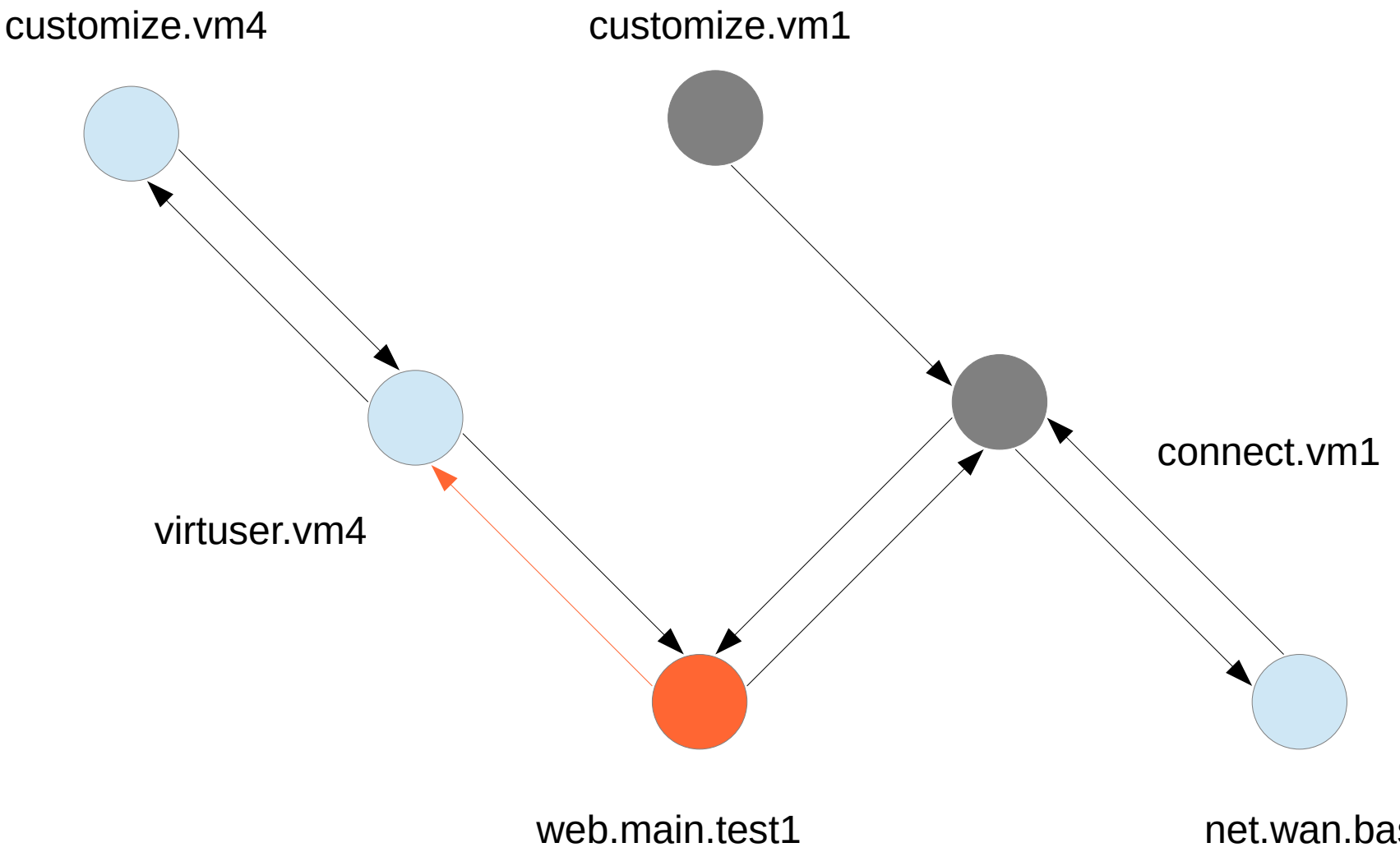


It's just about to get boring, but not yet. I promise it it will become a rather tedious process in a few slides ;)

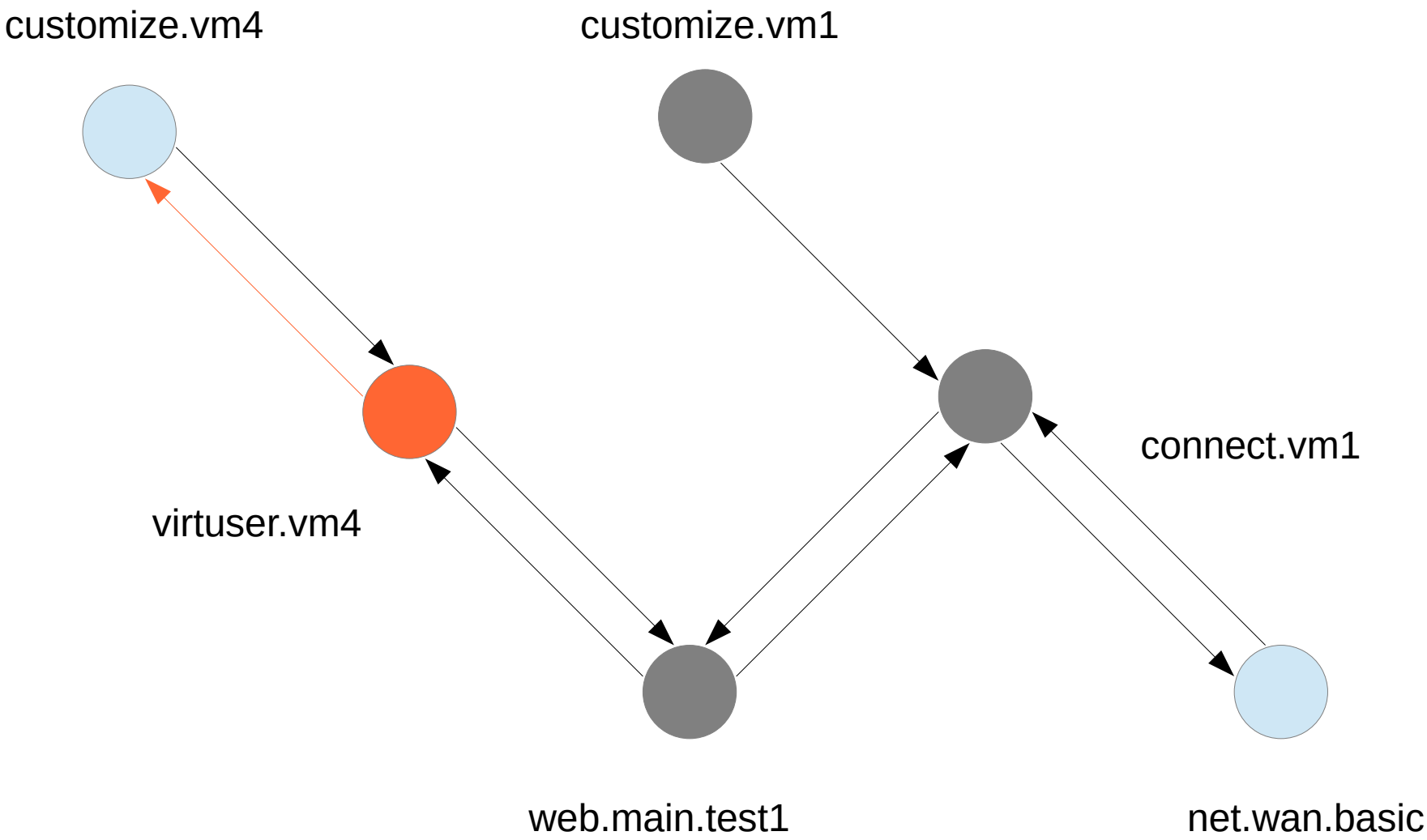


Checking the conditions tells us what to do next:

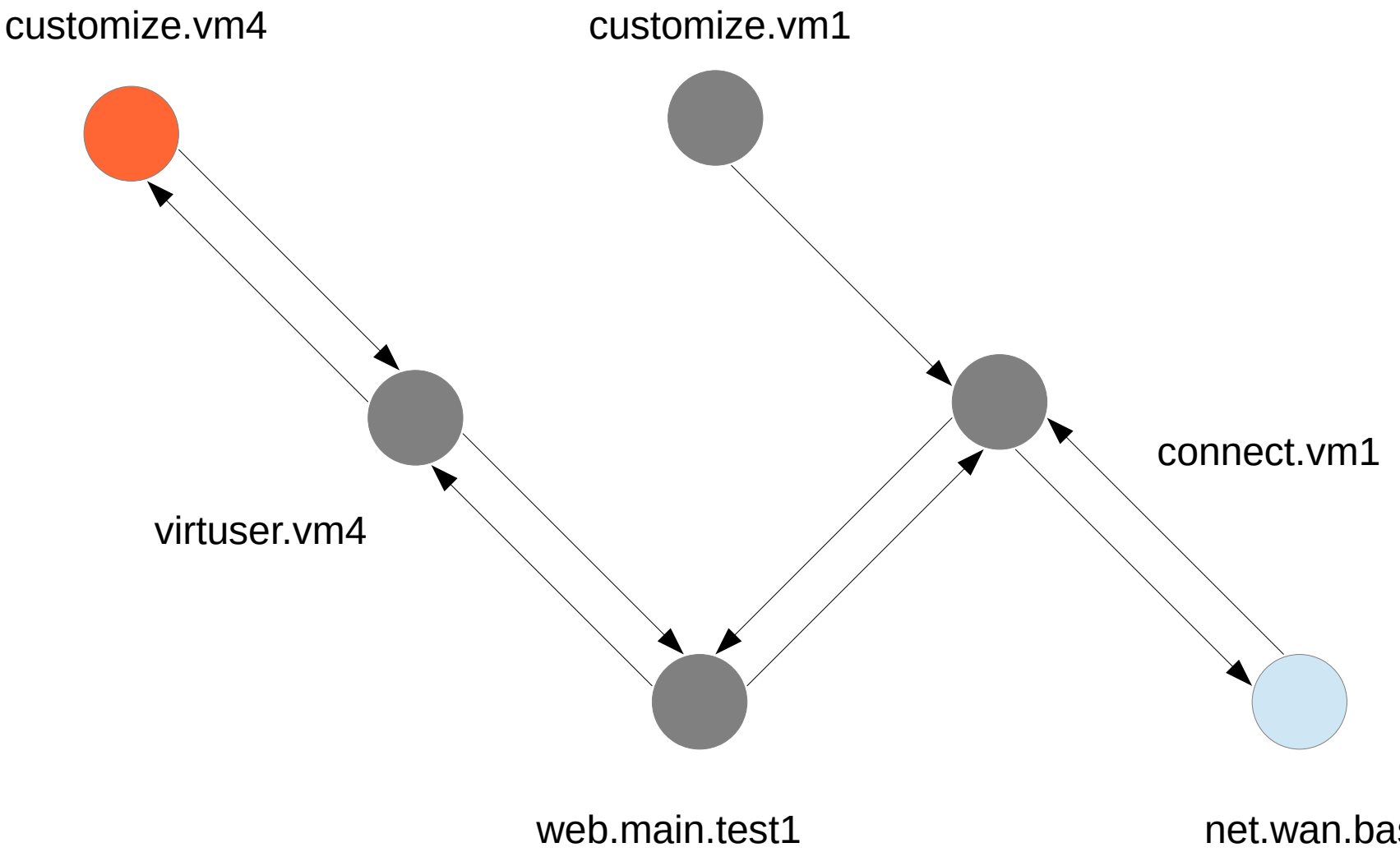
- I**s it setup ready** – False (there are two arrows to parent, so multiple objects have to be prepared)
- **Is it cleanup ready** – True (no other tests wait and depend on this one, it is a “leaf” test node)
- **Is it reusable** – as a leaf node, this is expected to leave no end states, i.e. it is not reusable (False)



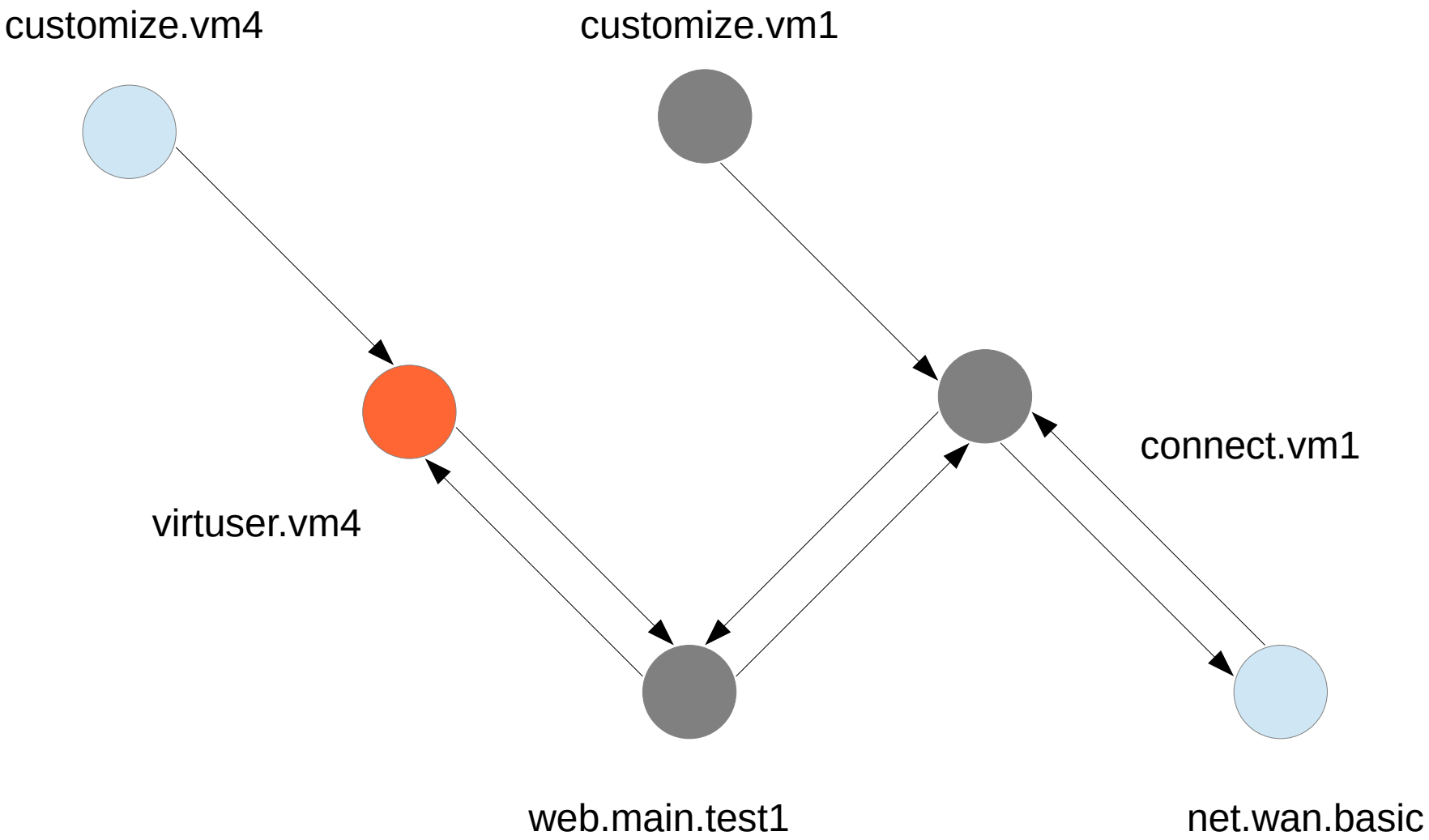
The test node not being reusable ever means that no matter how many times we run it, we should run it again. However, not just yet - just like before we should inverse the direction and go to the next parent based on id count or other priority.



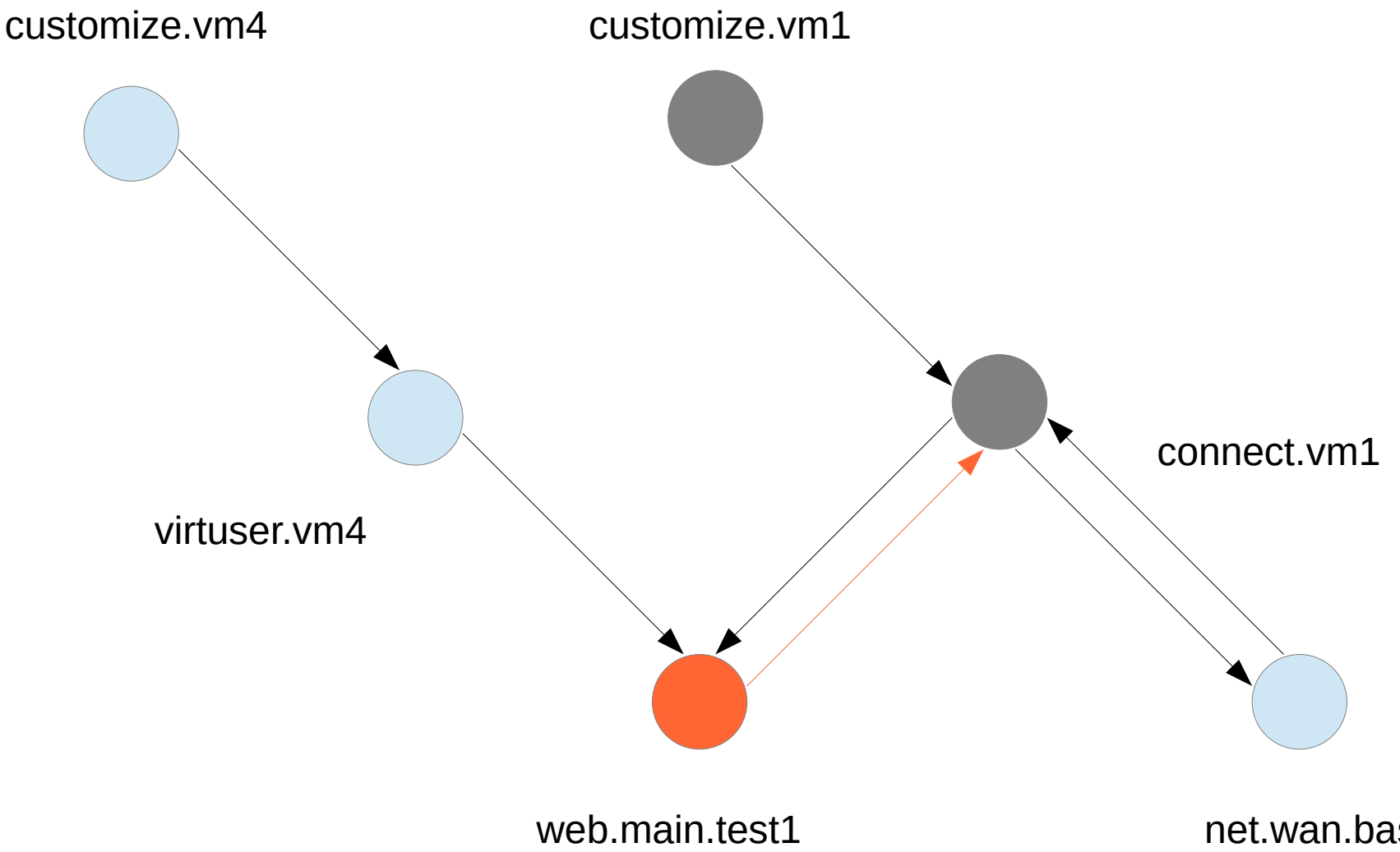
Inverse direction always makes us check only if the node is setup ready. It is not so we go to the next parent based on the criterion of choice (inverse DFS as promised).



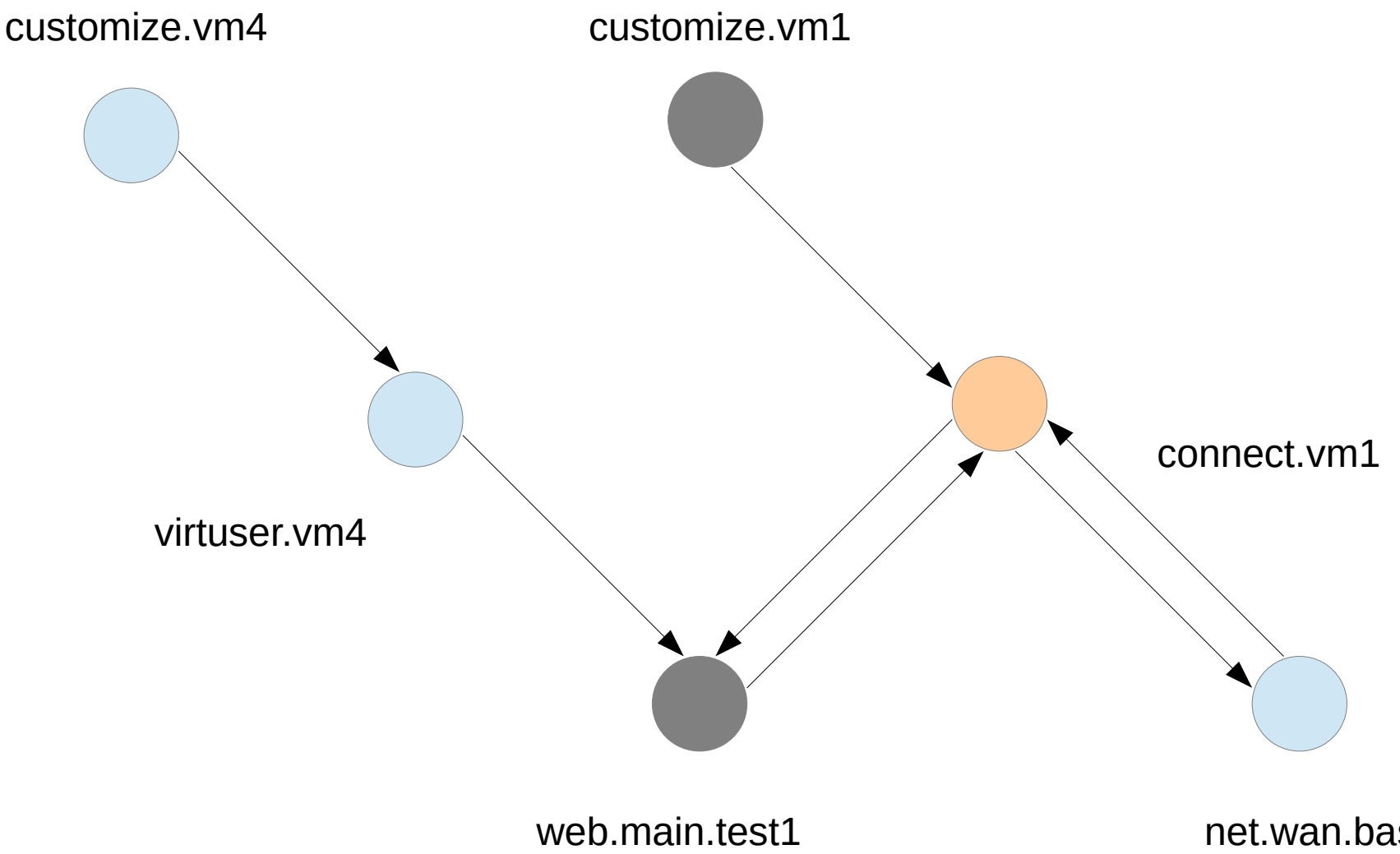
As this seems to be another root state (not really but for the sake of simplicity yes yes yes), we repeat the same thing like *customize.vm1* but for vm4 as *customize.vm4* so it should lead to...



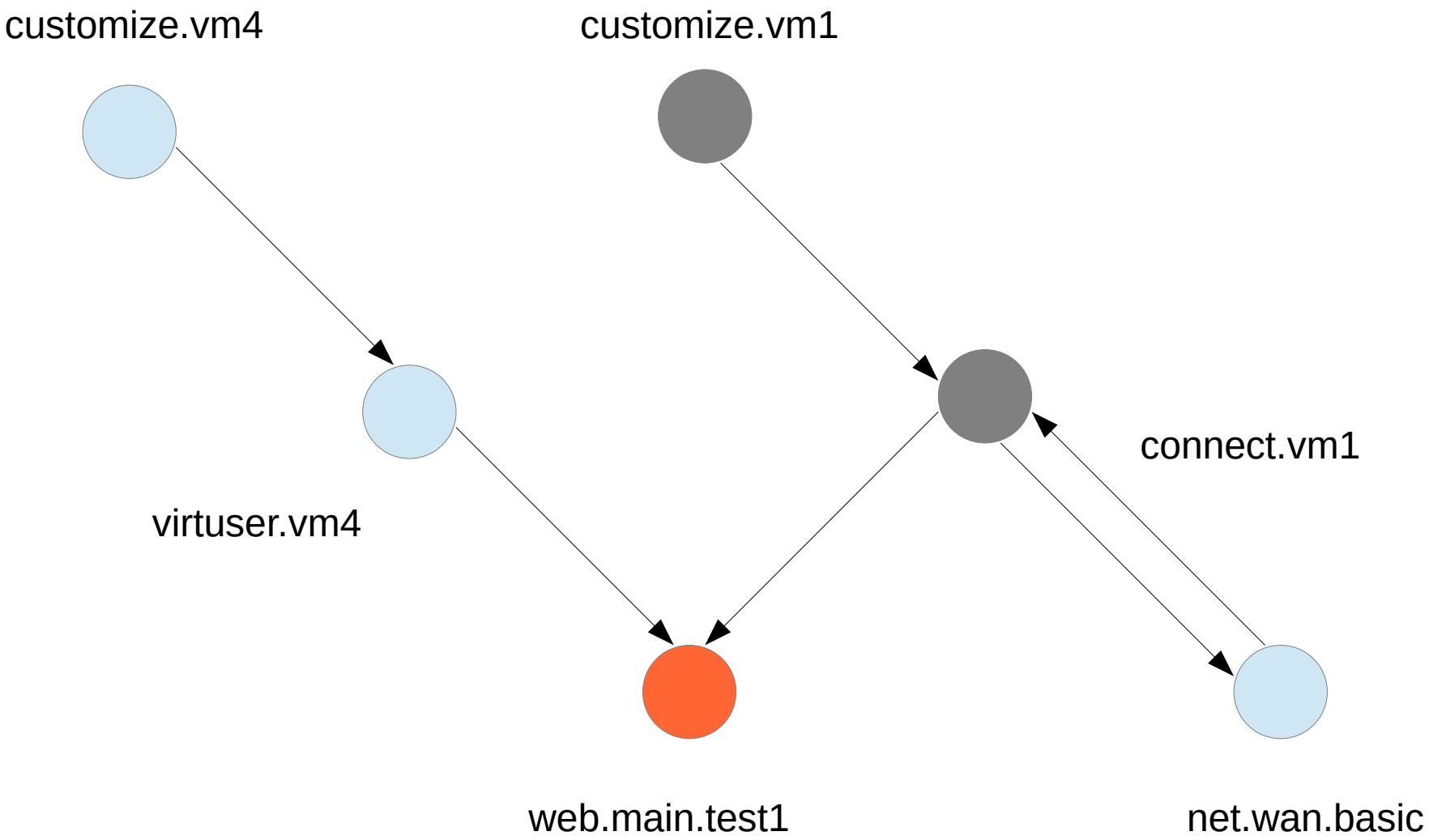
...this. Now the virtuser.vm4 is setup ready so we make another step back running yet another test (I forgot to say we consider it not reusable).



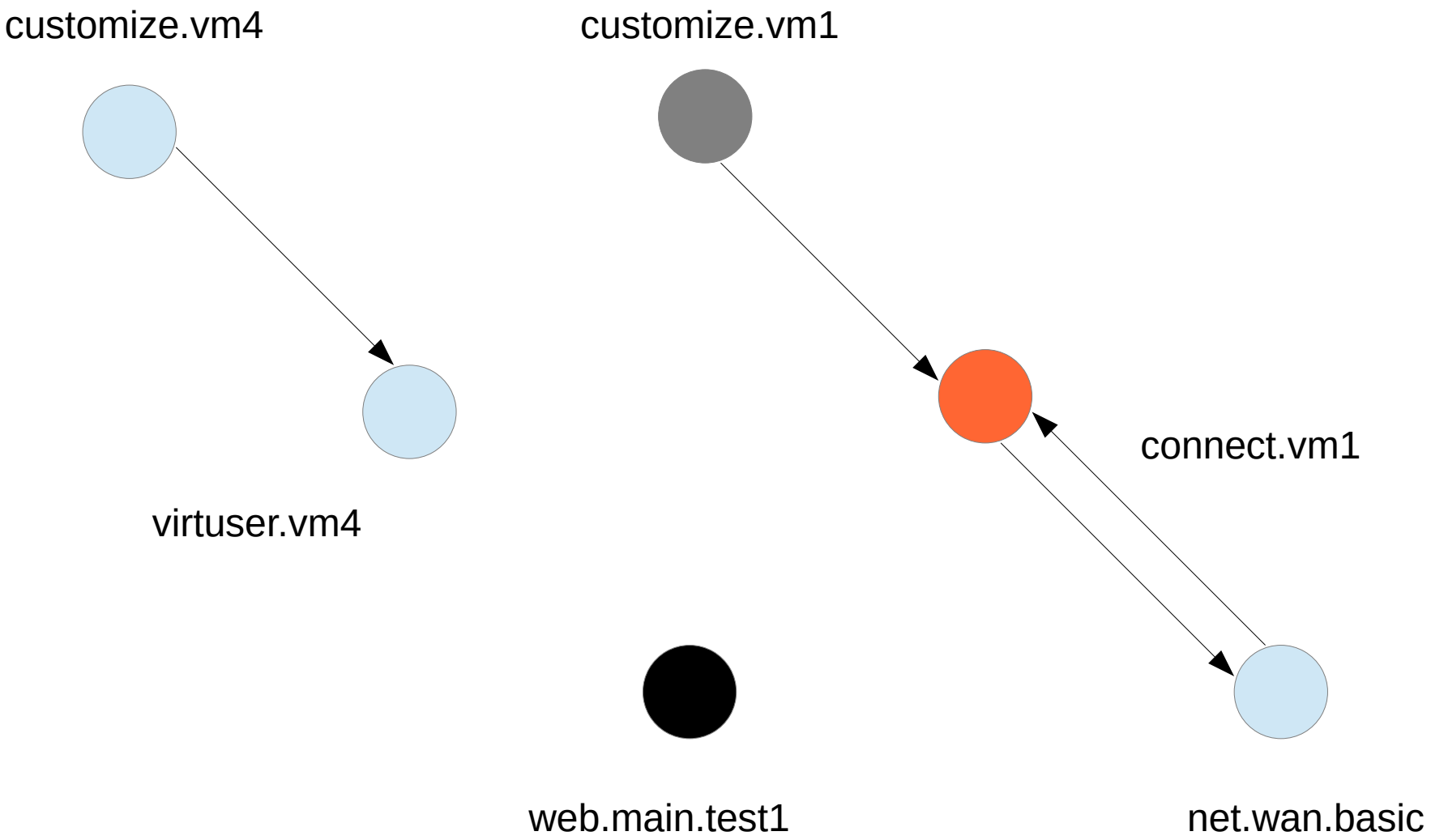
Is web.main.test1 setup ready now? Not yet - one more object needs a setup but this one should be easier, as connect.vm1 was run and left behind an end state to be reused.



Since connect.vm1 is reusable (True) we skip it. Good but a question here would be where now? The direction is up, so the three steps apply: run it (done), remove the arrow (done in next slide), and pop the node. Oh, that should be it then – we pop the node and go to the previous node, i.e. web.main.test1.

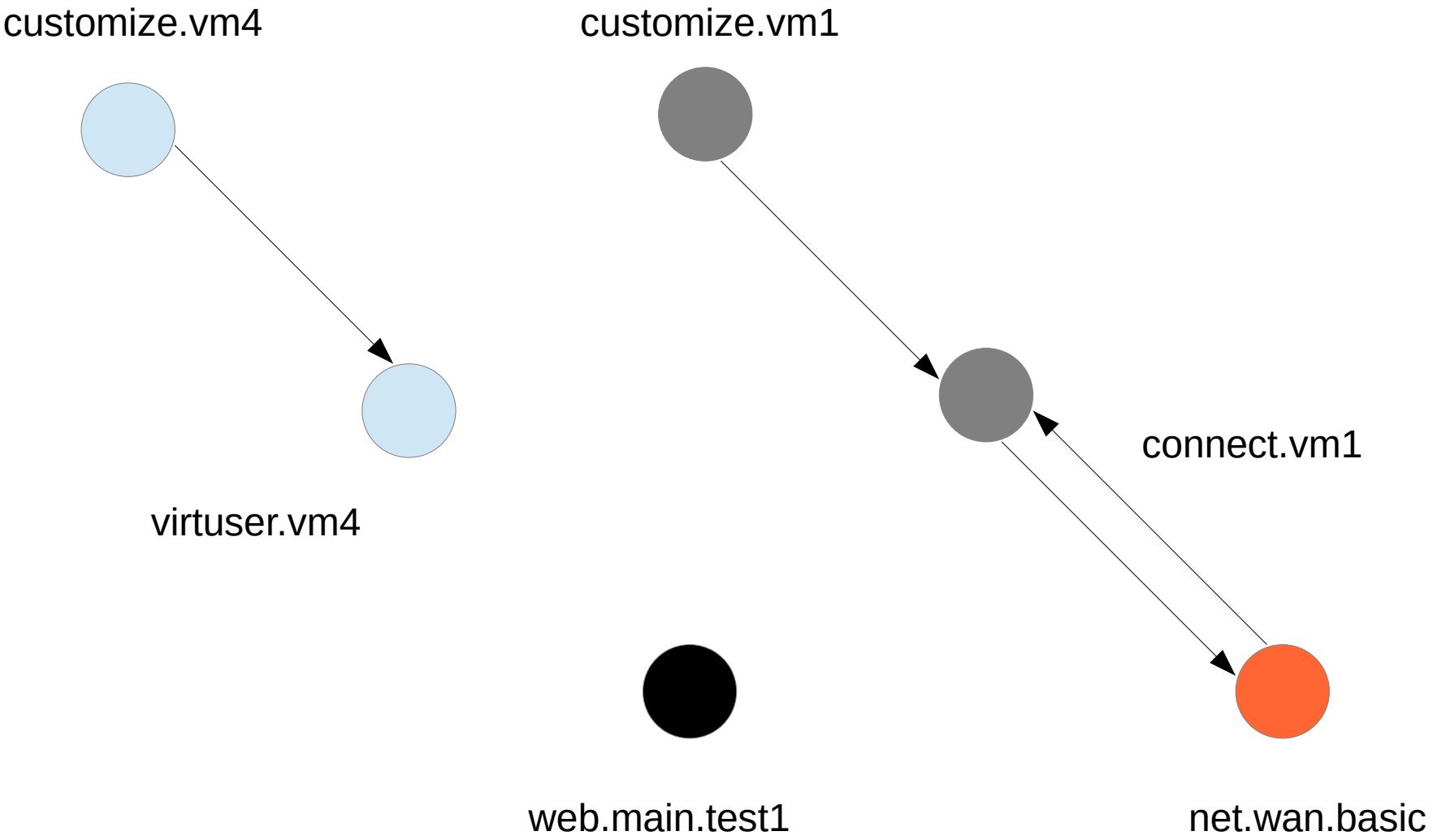


Is web.main.test1 setup ready now? Finally it is. So we run the test. What after we run it? Now we get the chance to finally handle cases when the node is cleanup ready. And how to do it? Well, it is almost the same like handling the setup ready case in upward direction, but mirrored for the down direction.

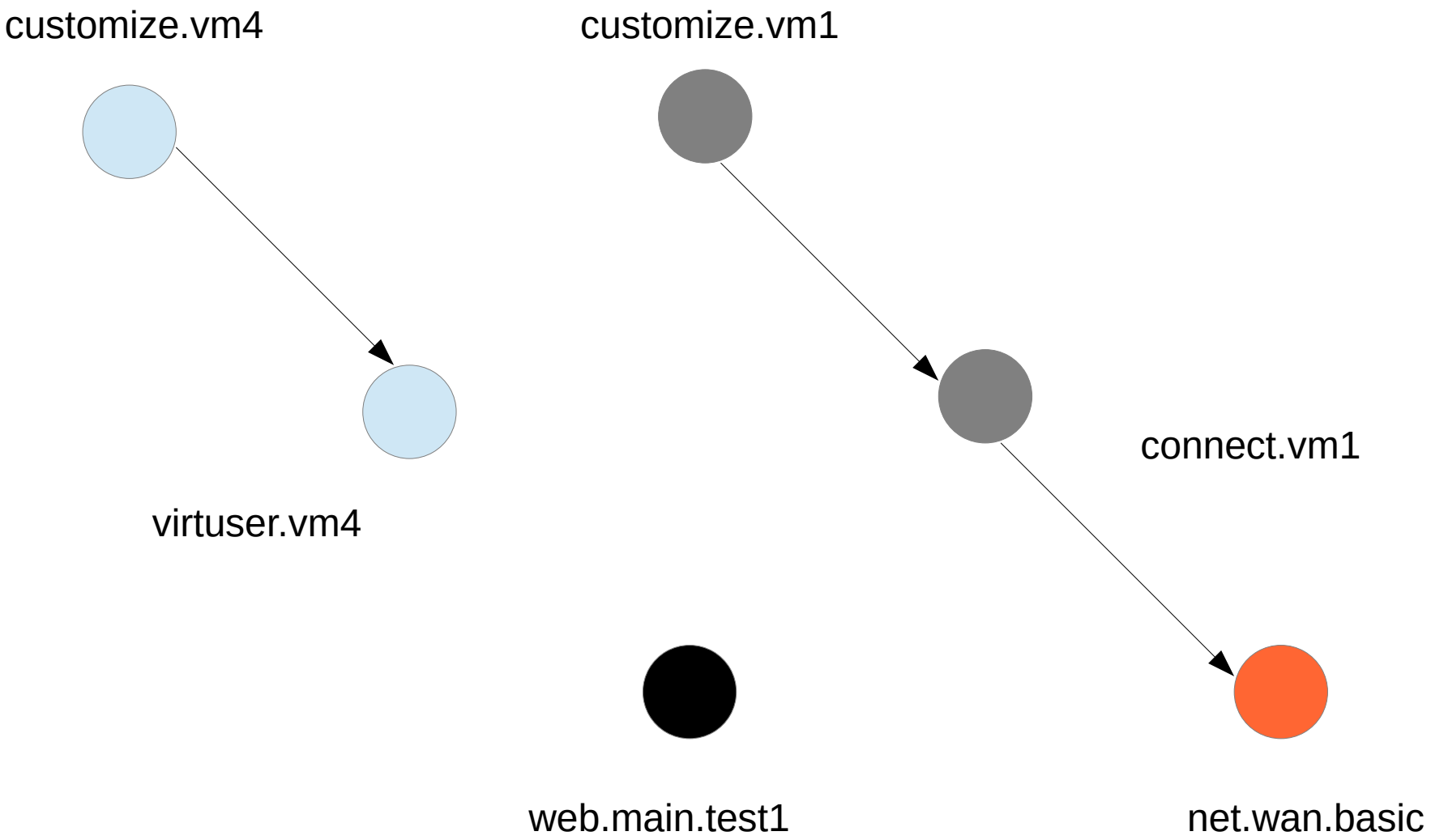


So as we just said, three things happen:

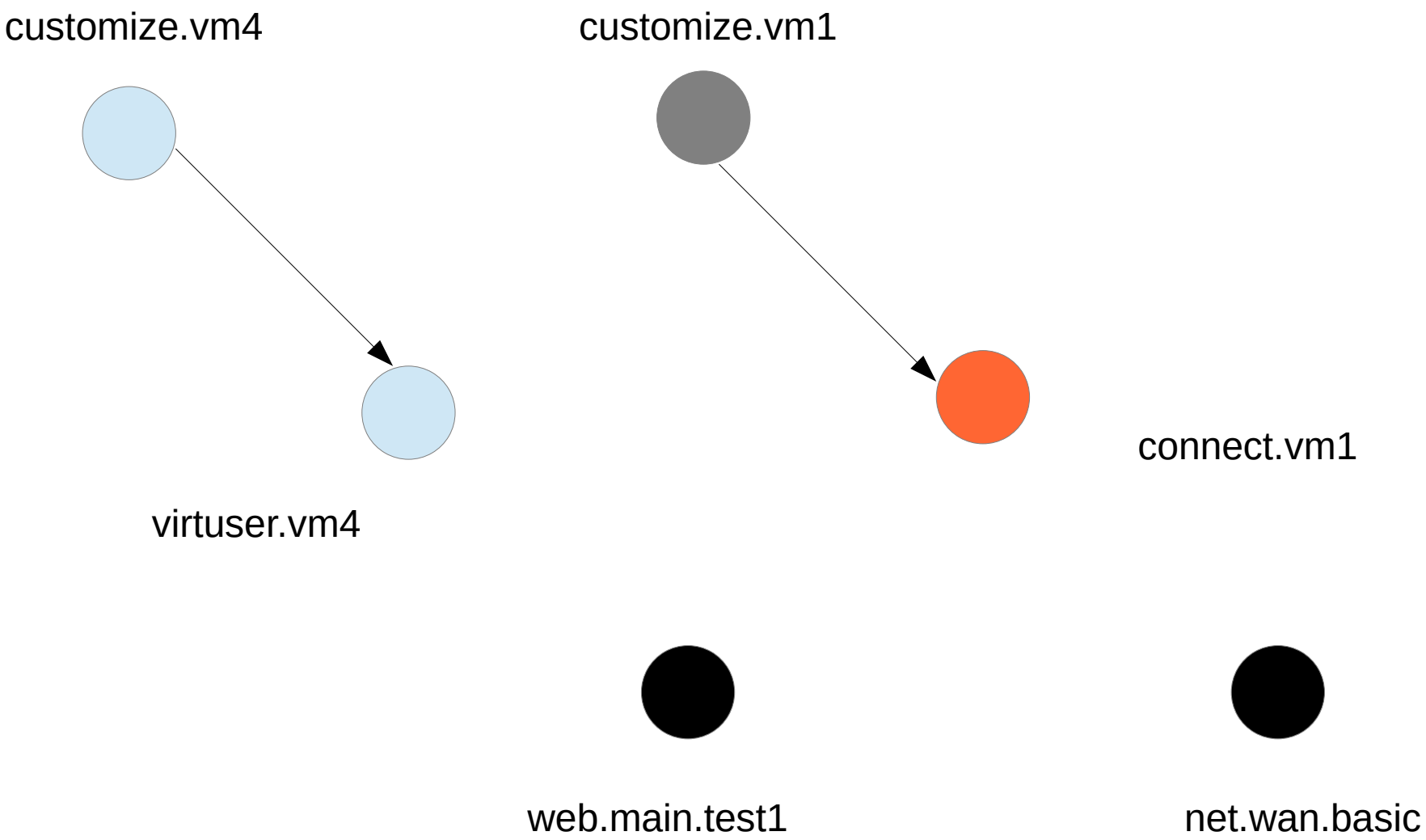
1) Run the test, we said that already
2) Pop the stack, i.e. go to the last test before this one and make a step backward in the path
3) Remove all arrows from the parent nodes to this one (or rather consider it visited child for all)



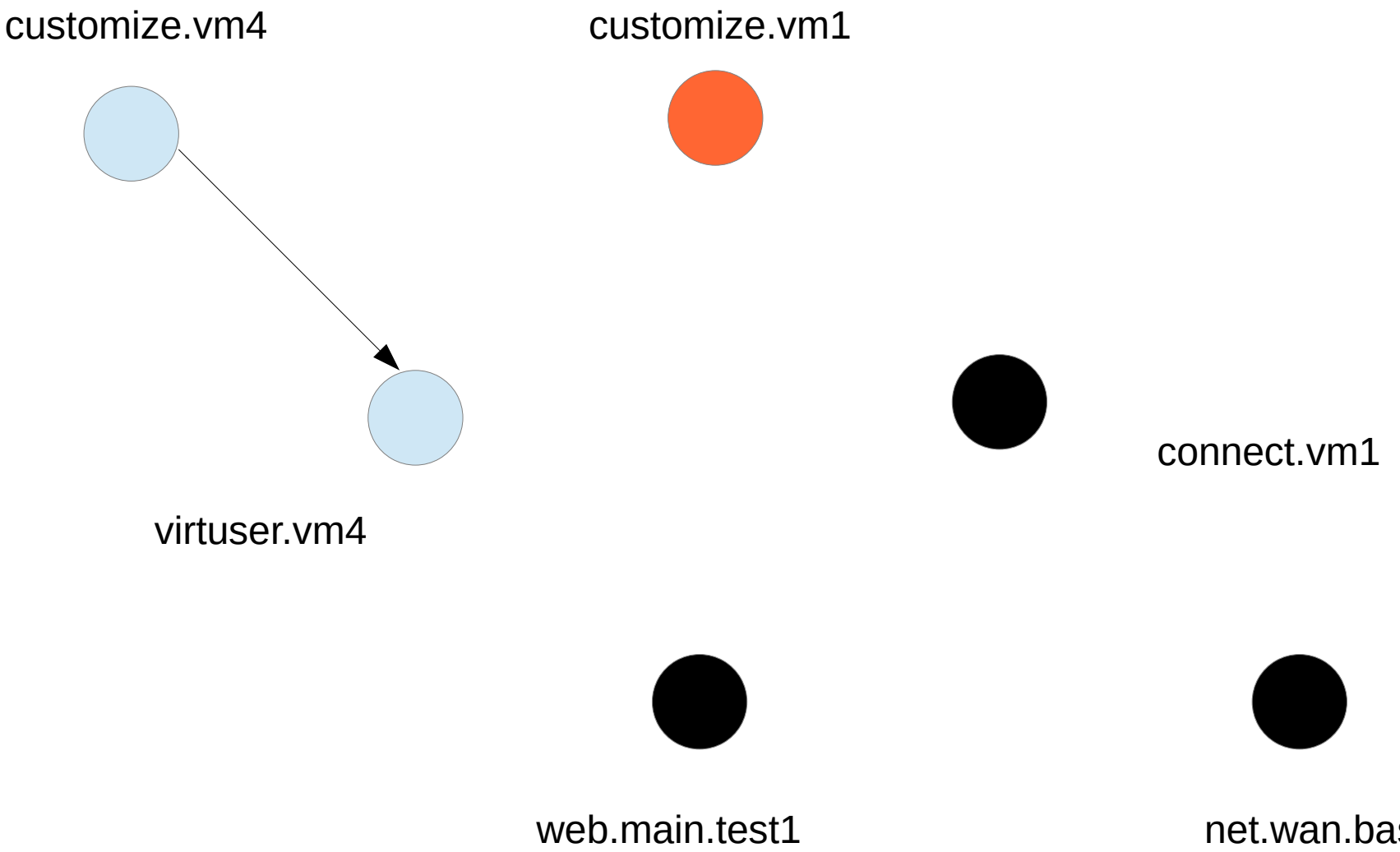
Hold on, we are almost there! We now continue our regular DFS with the next child which turns to be not setup ready, cleanup ready, and not reusable (leaf).



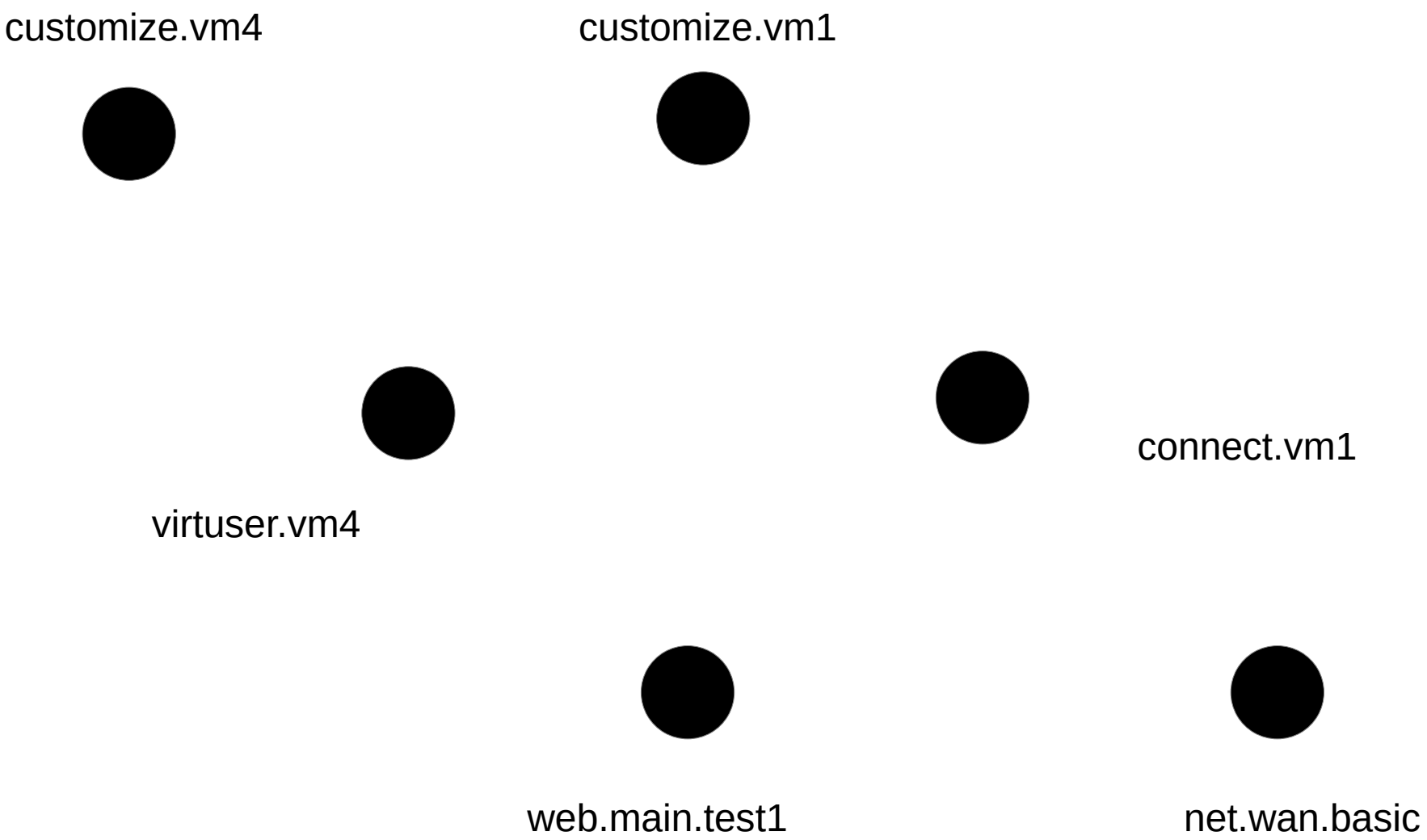
I promised it will become tedious although I am a few slides down that promise. The setup for this node was reusable/skipped, the inwards arrow was removed and now we are about to make the three cleanup ready steps again...



Finally the connect.vm1 node is cleanup ready! As you can see the test is no longer needed for any other test node so this would be the perfect place to perform other cleanup steps like removing object states, etc. and not just popping.

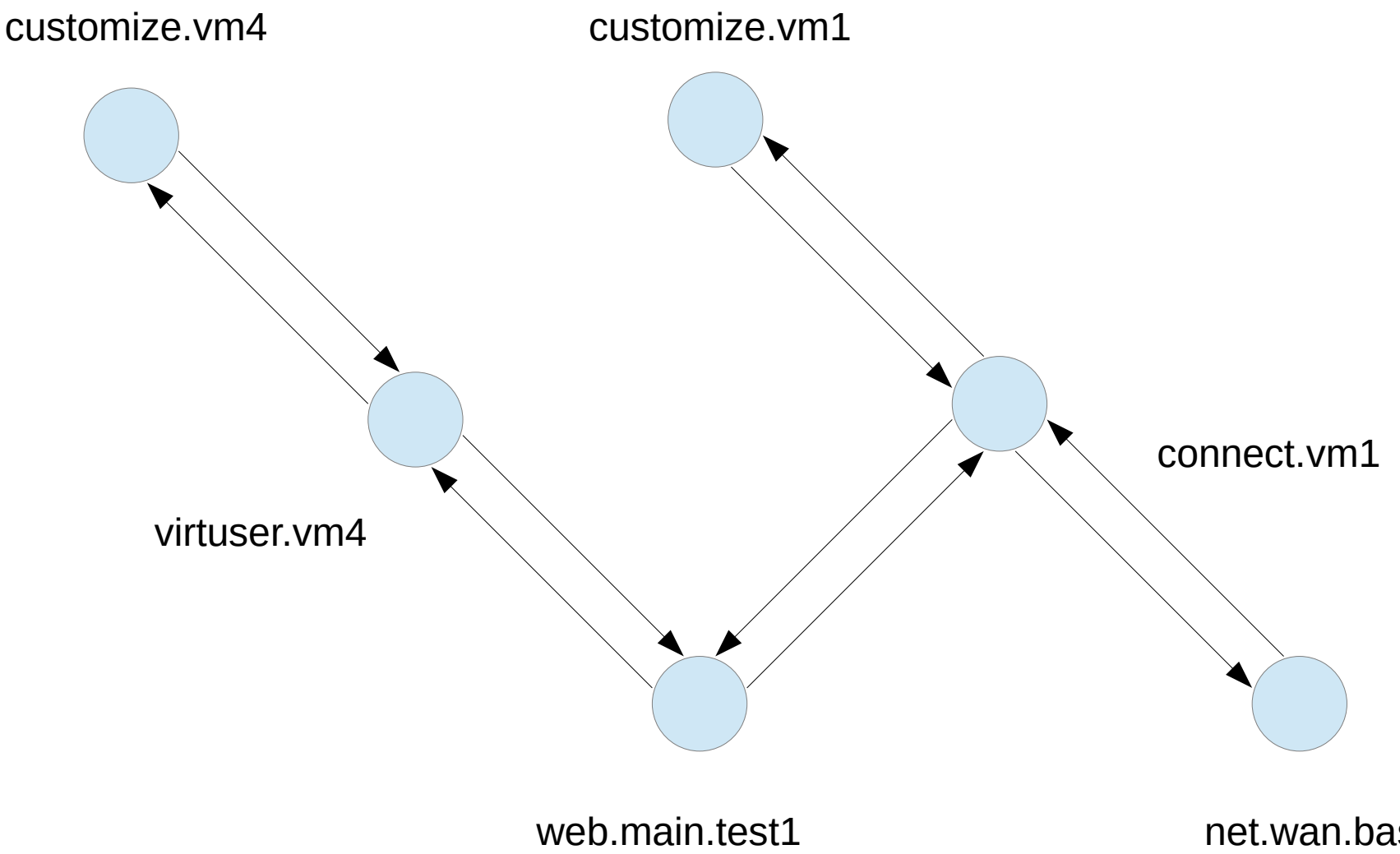


The size of the stack has reduced to one... I leave the nodes incinerated for visual effects, all to show that there is a branch that will never be traversed and it's child tests never to be run. How can we set it on fire? Well, with a single real root test node which we skipped here for simplicity. If our current node was that root, it would have continued down its children, reaching these nodes.

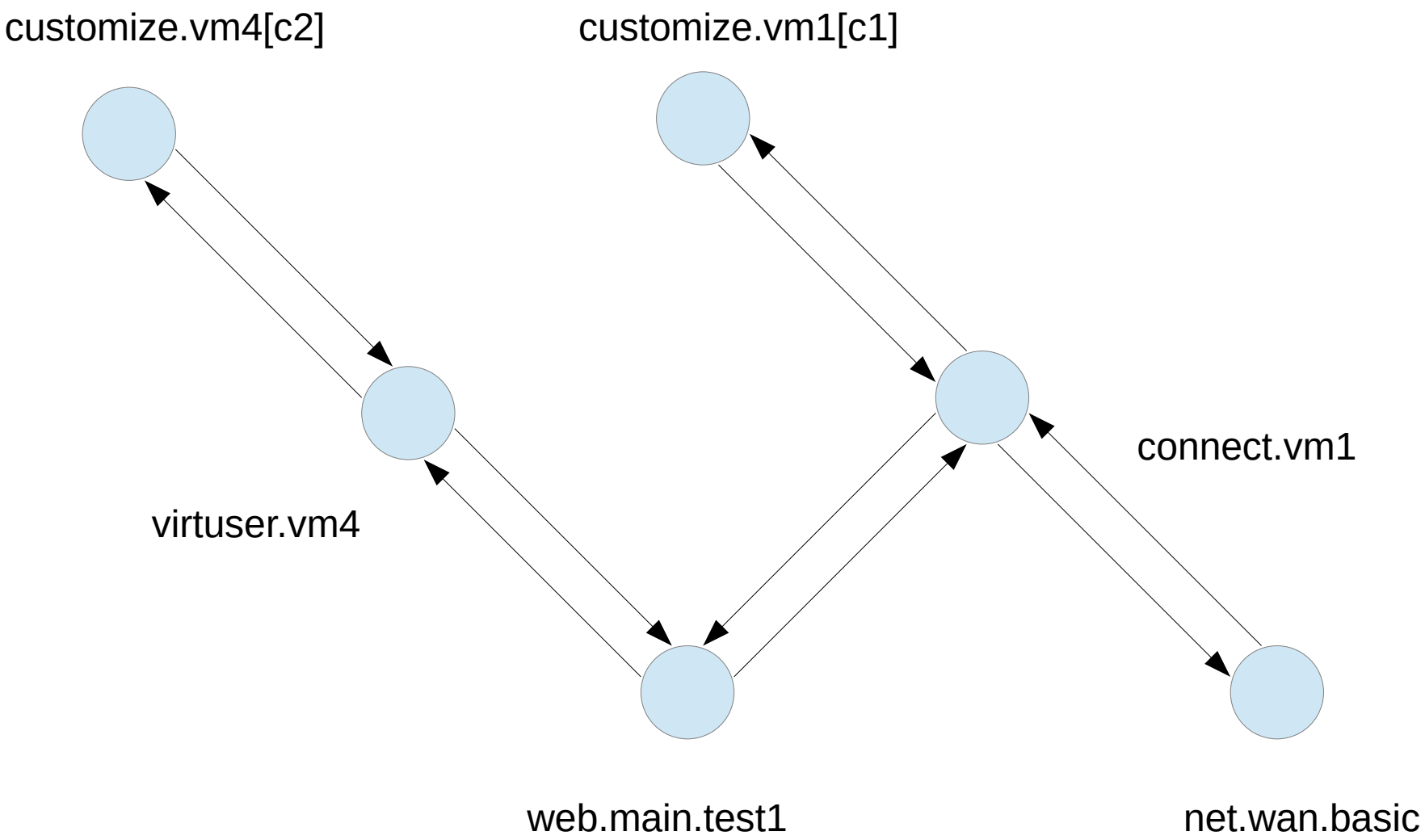


As you might have already guessed the exit condition is:

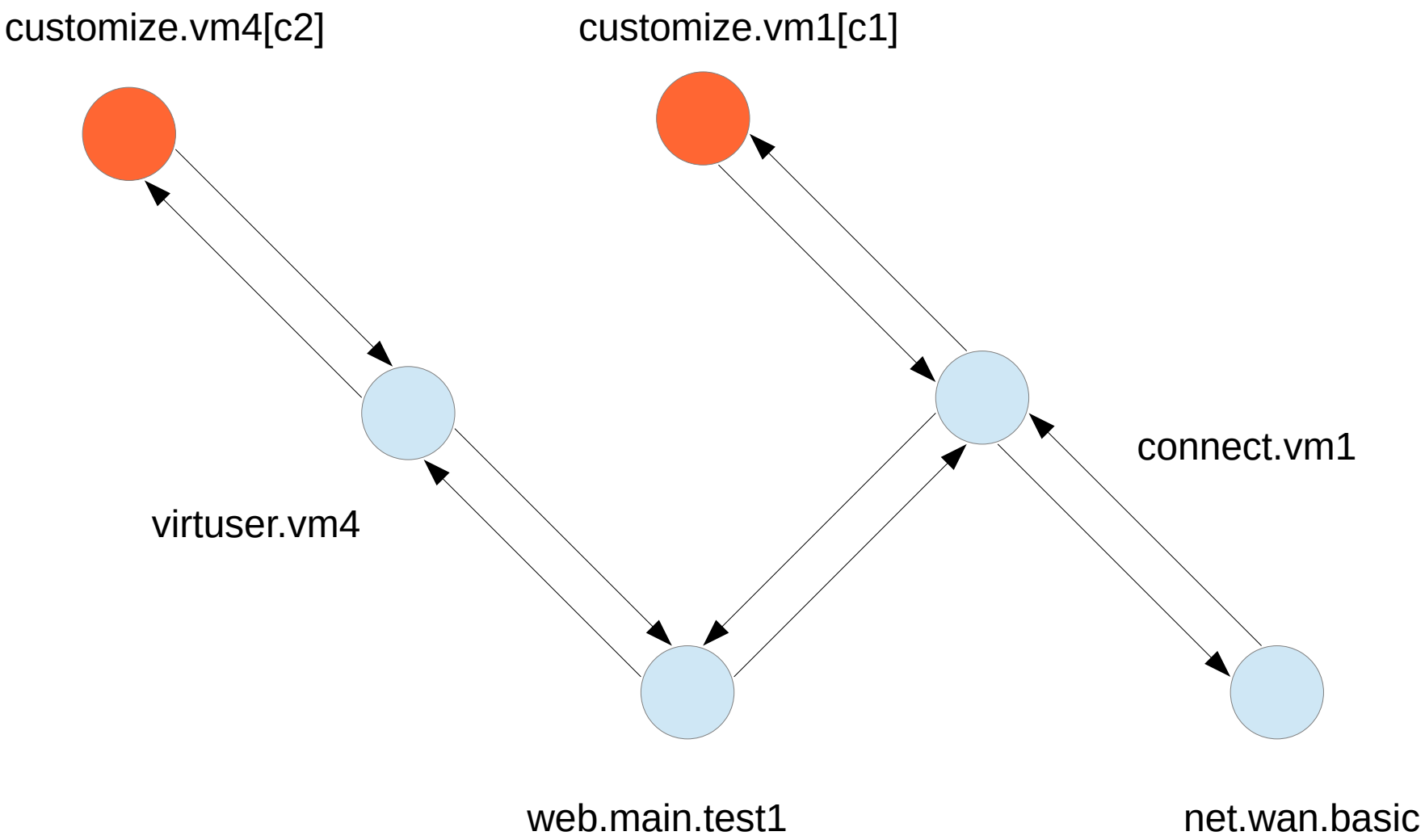
root_node.is_cleanup_ready() == True.



Instead of the classical test-defined “setup” and “cleanup” procedures, it is so much nicer for a fellow test to just put all its mess in one object state and allow other to reuse it or simply forget it. This structure is trying to make the development of each test easier, taking care of both parts but also of scheduling of the tests so that maximum mess is reused :) But wait, it could be faster...

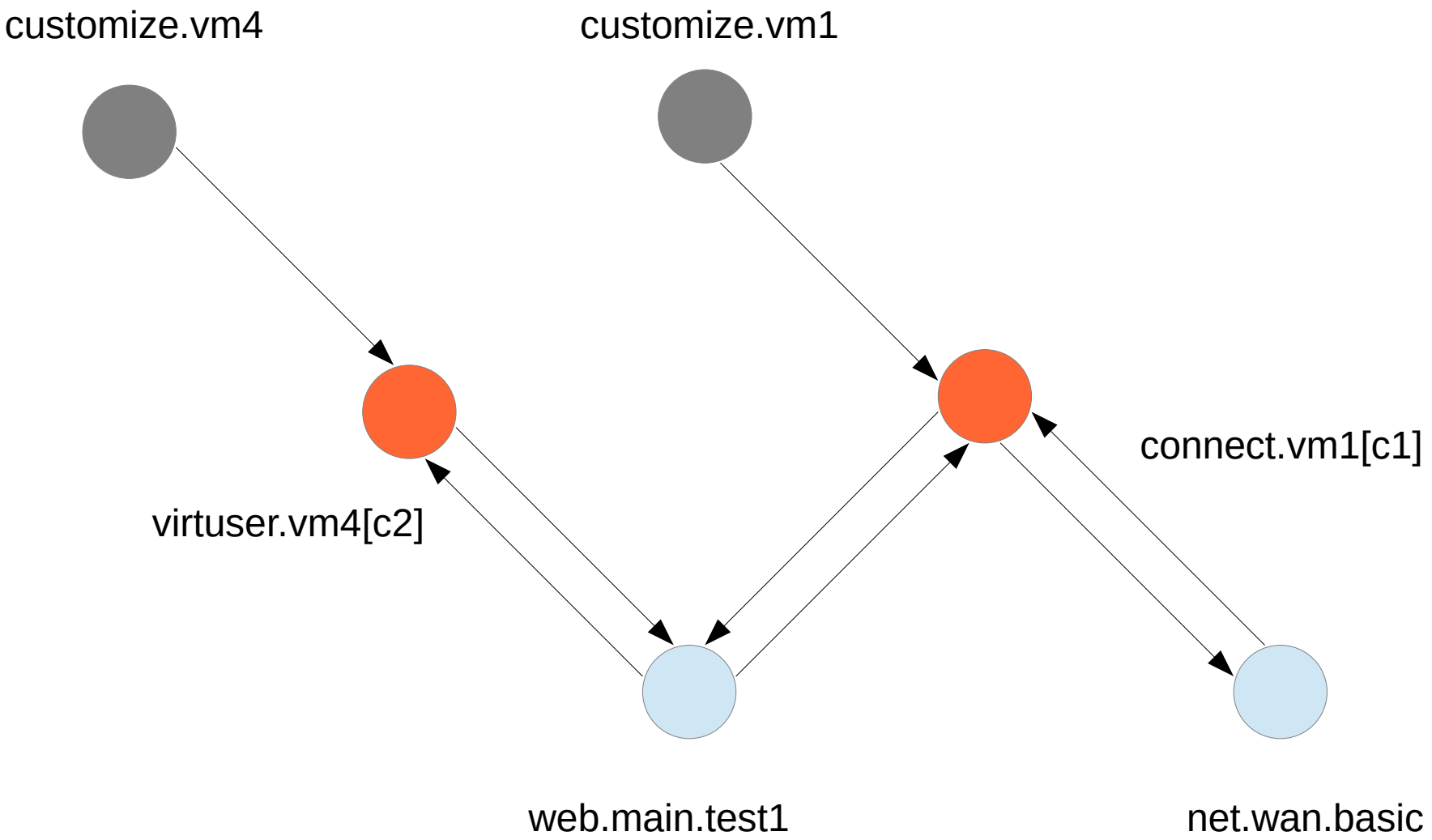


Instead of just one such traversal happening at once, imagine multiple workers running tests in their predefined isolated environments, all traversing the same graph. Then take this a notch further, like ants sensing their chemical trails when exploring around, workers could also diverge into maximally different directions, covering different corners of the graph faster.

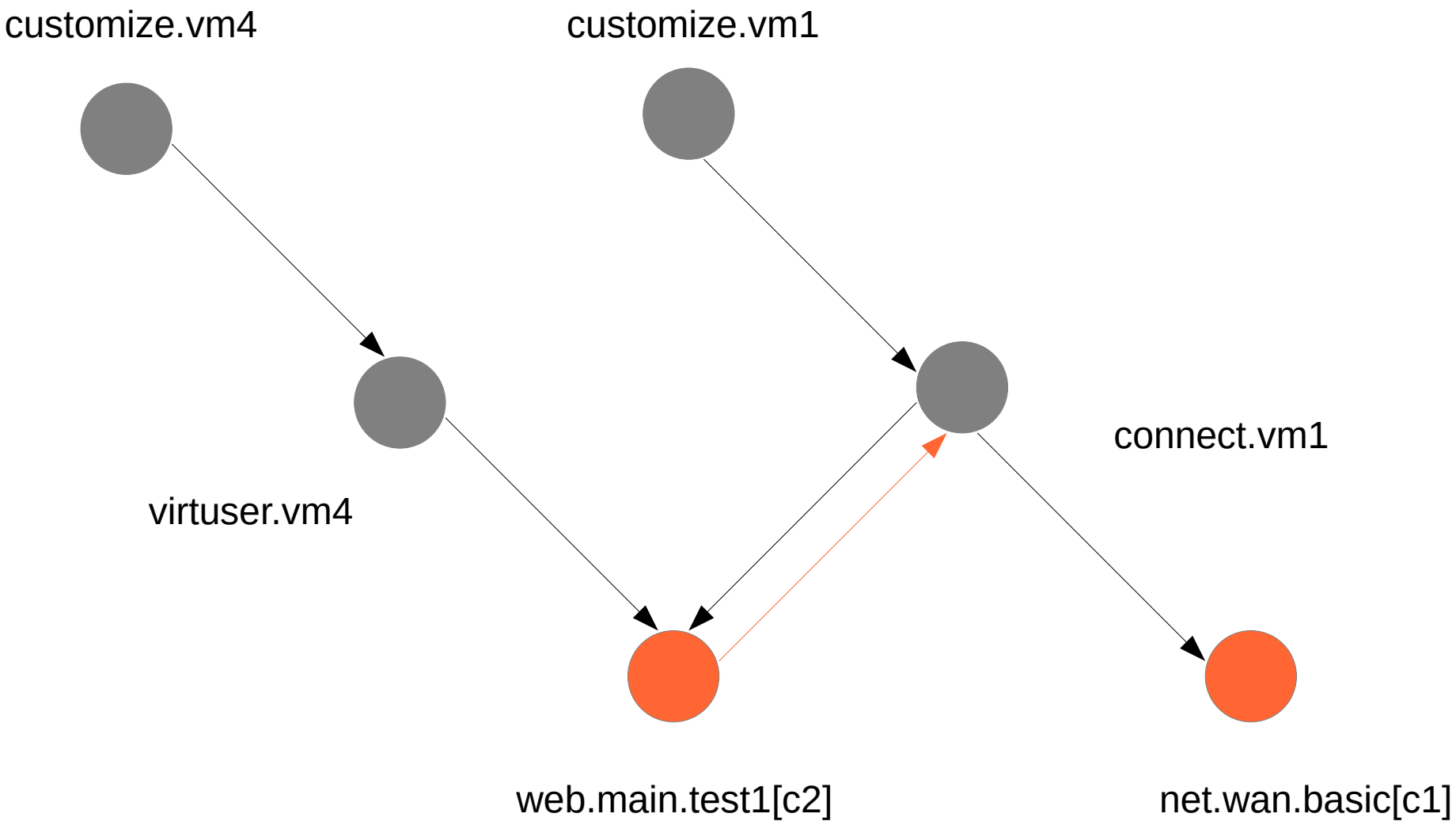


There are at least a few advantages of such a decentralized scheduling policy which could maximize test coverage in the shortest amount of time:

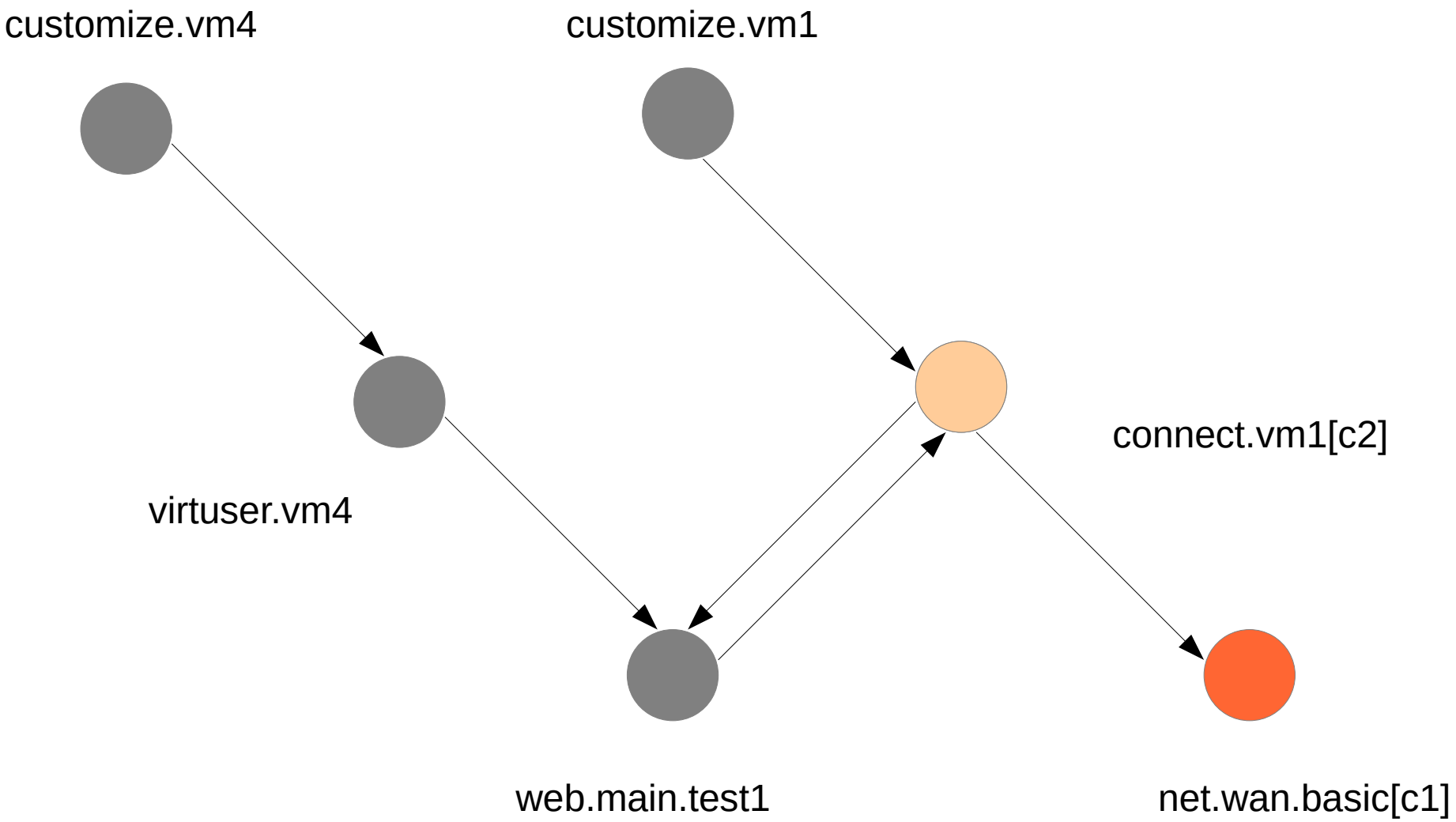
1) soft early fail - the more descendants a test has (the earlier and more crucial setup it happens to be), the faster it has to be run and fail for a “soft” or fuzzy early fail of sorts where more critical problems will be reported ASAP



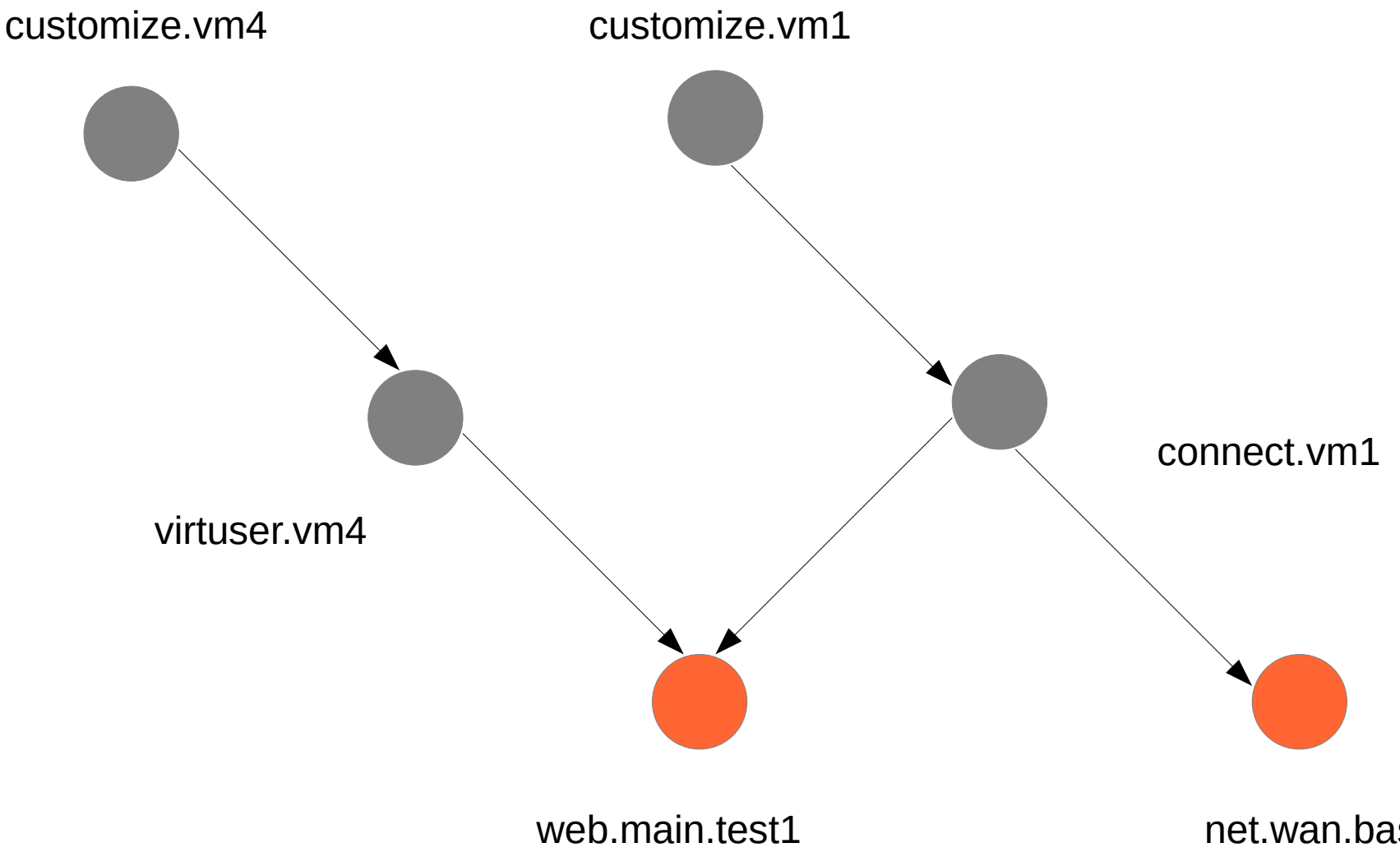
2) gradual coverage of increased resolution - imagine loading a picture in lower than higher resolution to paint a picture of the product state, e.g. what we used to do with predefined "minimal" vs "normal" test sets but now also auto-generating everything in between for an arbitrary time period allotted for the test run



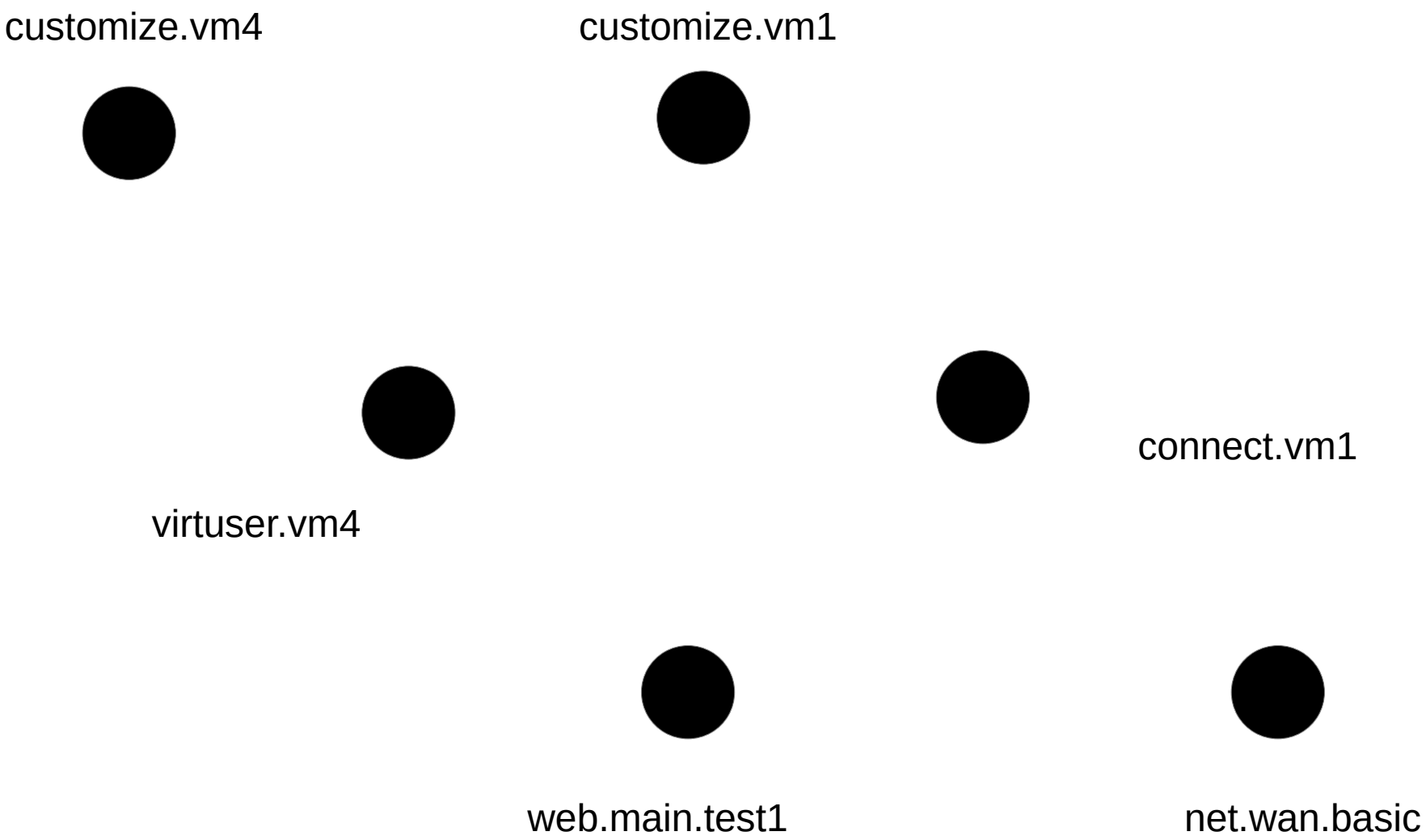
3) greater reuse and localization of resources and produced setup like downloaded or cooked vm states which will "cluster" around the same worker environment slots - the graph scopes happen to be very identical in terms of resource demands, e.g. the tests inheriting from a given setup node usually require similar vms and states so diverging traversal paths will make it less likely that another worker needs similar setup



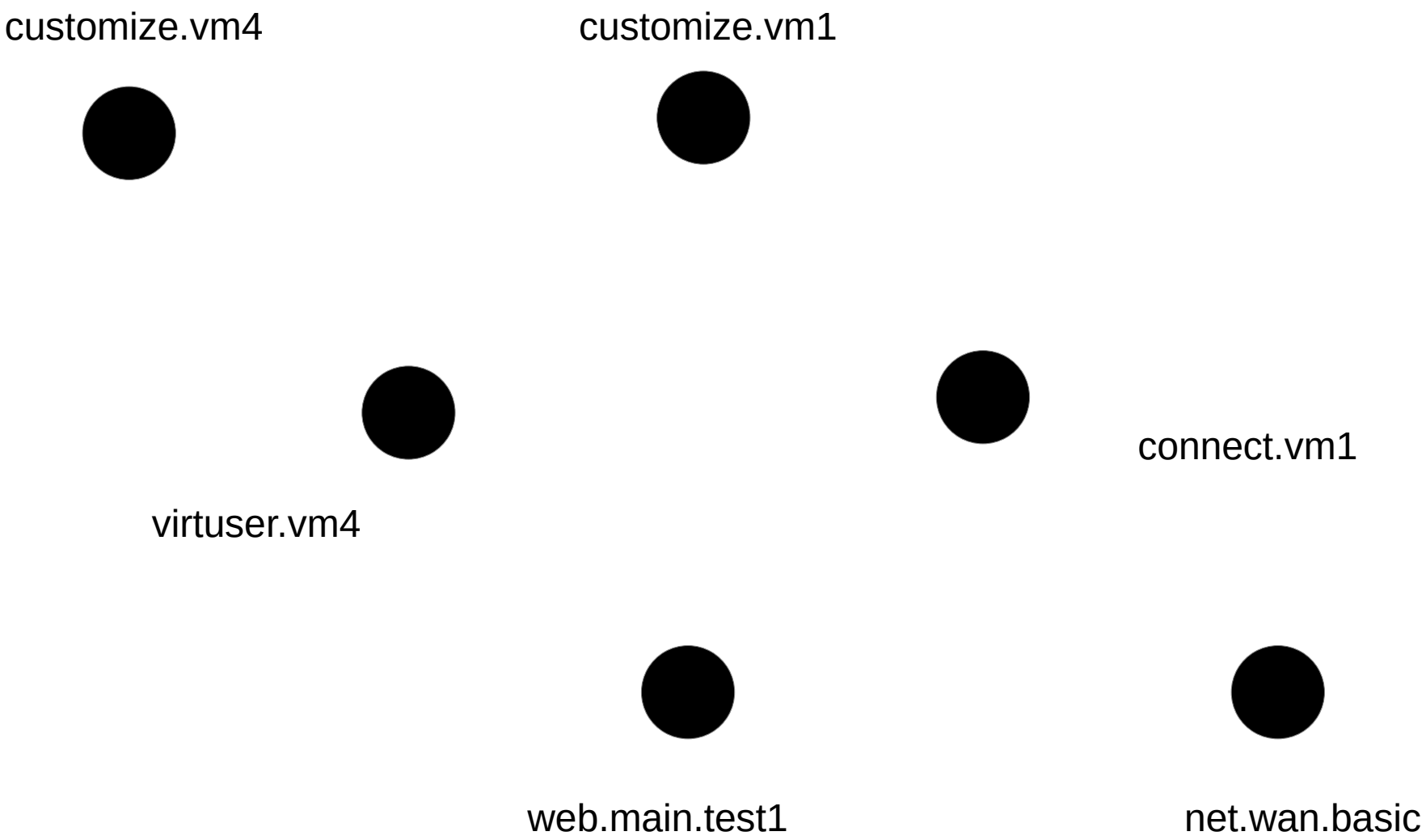
And worker c2 can simply rerun the setup test node connect.vm1 in case it failed and setup is still not available or download and reuse the setup from worker c1 (the two workers thus forming a decentralized pool of reusable setup). As c2 “scents the correct chemical trail” of which worker (c1) connect.vm1 was traversed by it can request the reusable setup states from that worker’s environment.



Due to the inherent setup/cleanup symmetry if both workers end up in the same web.main.test1 and it is not ready with setup, the same parallelization strategy works also in reverse – the workers will distribute the setup load among each other and cut down the setup provision time with just one worker running the final test.



Reversing the graph works identically too – each worker will make sure to clean up where needed or required (with correct state unset policy) or sync reusable setup provided by others it hasn’t downloaded yet. This setup could also be distributed in clusters of multiple swarms but the traversal policy remains the same.



The trail scent approach goes a bit further: there could be more sources in addition to workers for the sync (shared pools across the intranet), filters on which sources could no longer be synced from, and ultimately policies for syncing worker setup from its closest compatible source. As multiple workers could produce the same setup due to virtualization differences in cluster hardware each worker should rely on its closest sources to guarantee setup compatibility.