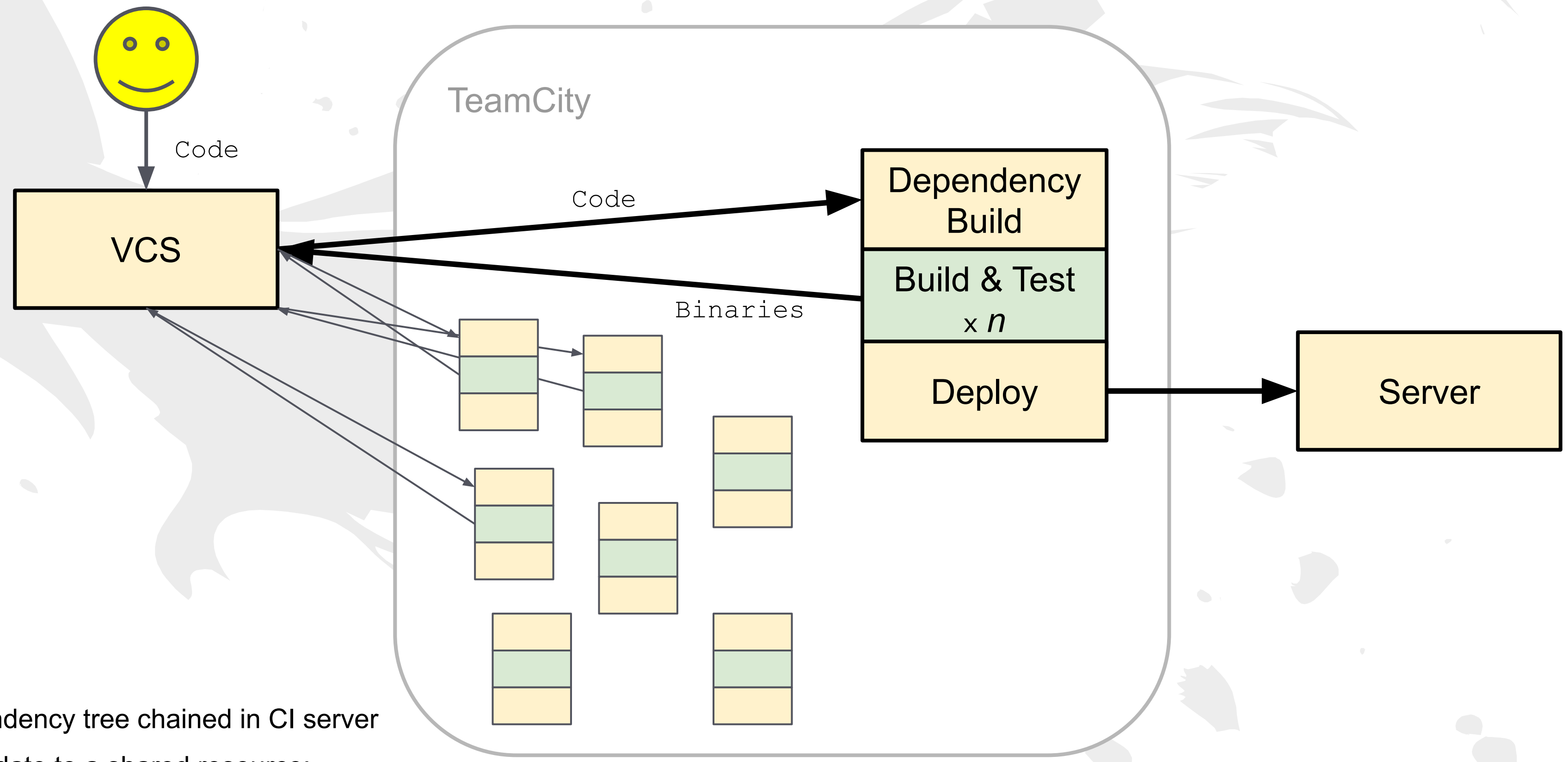
Distributed Dependency Management

fault tolerant and fast build processes

The Status Quo



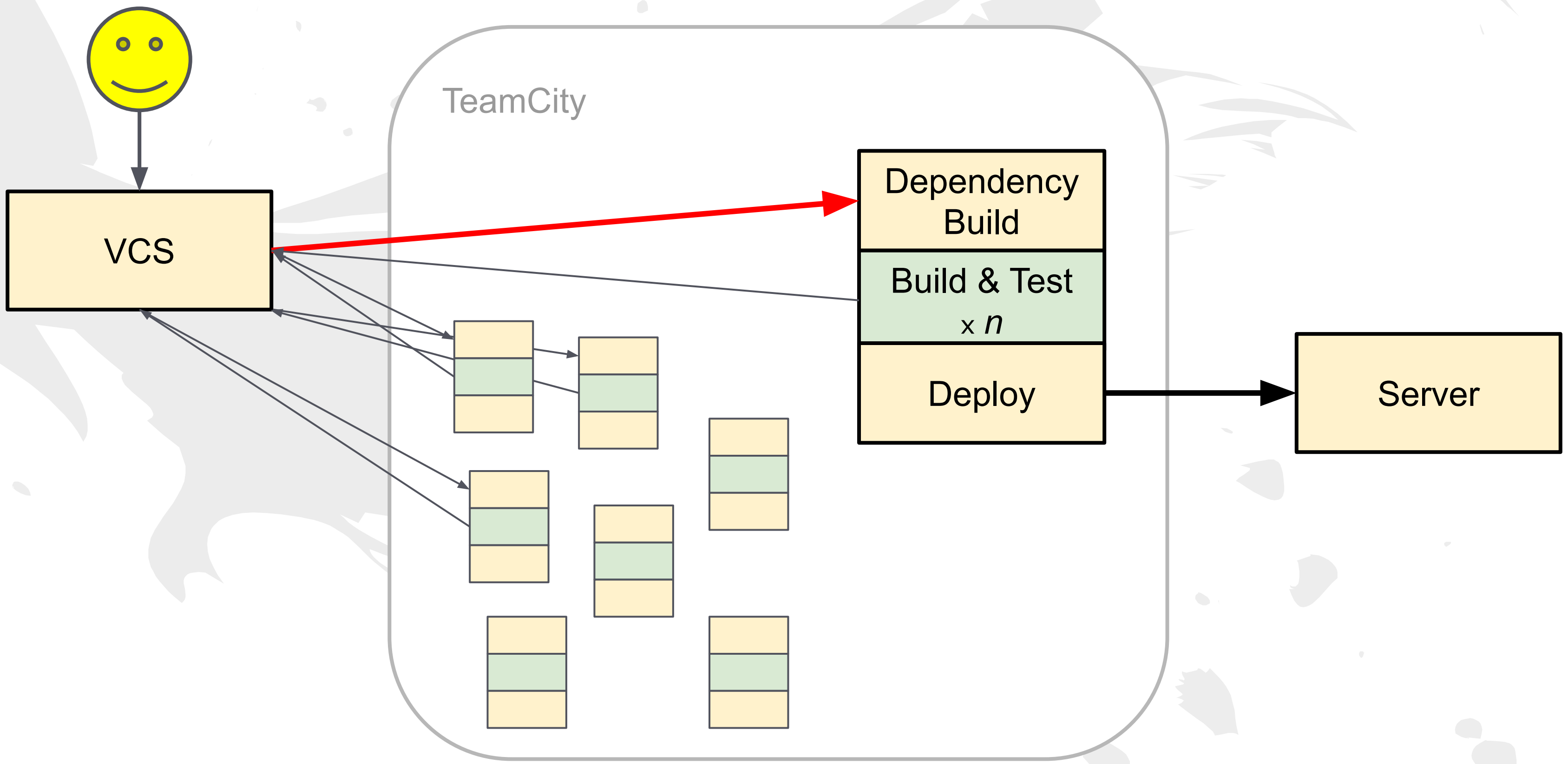
Dependency tree chained in CI server

An update to a shared resource:

1. Push resource to VCS
2. *n* build triggers watching resource fire
3. Solutions build and create new shared resources
4. These push to VCS
5. *m* build triggers fire

And so on until things settle down!

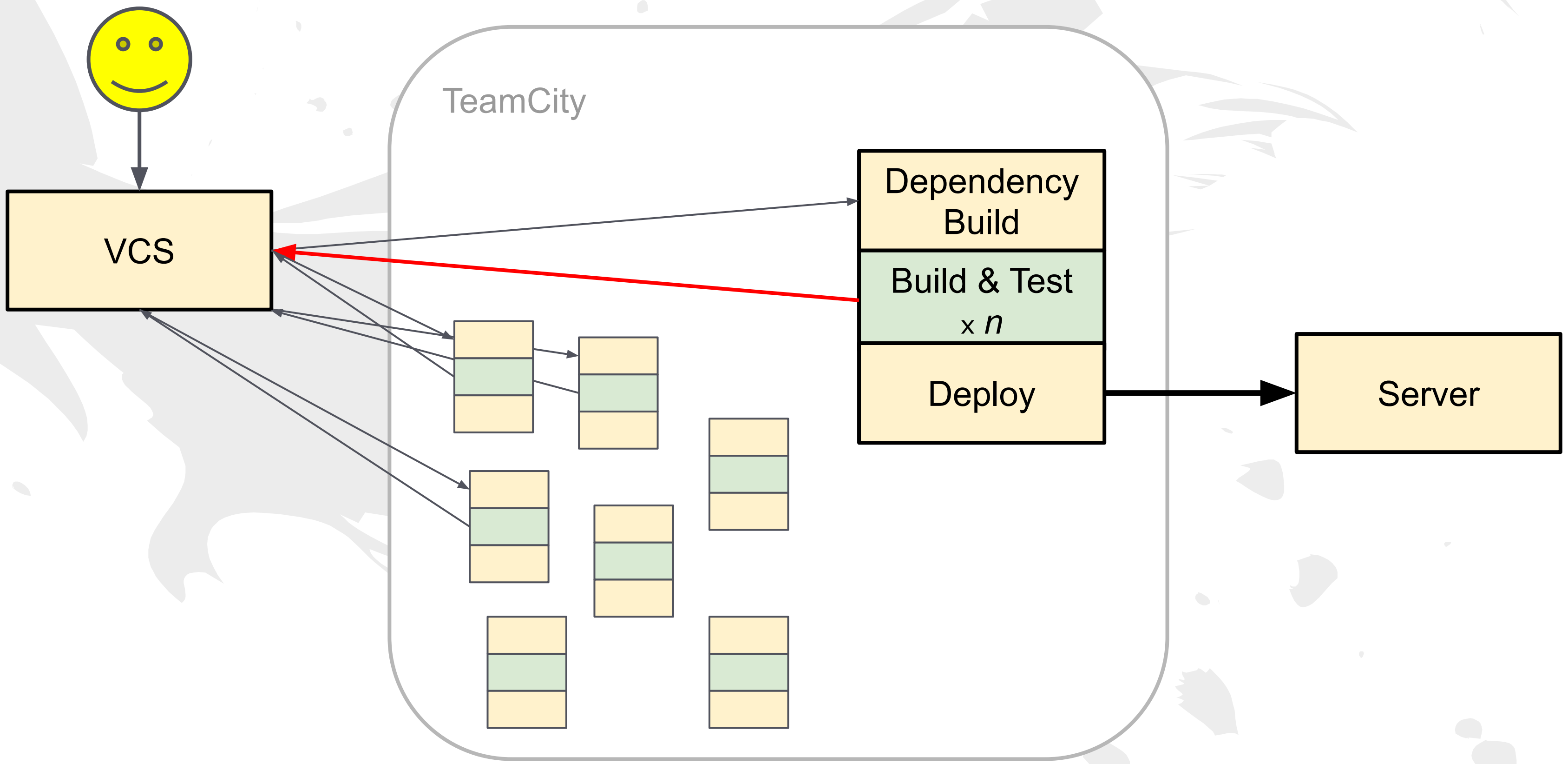
Waiting



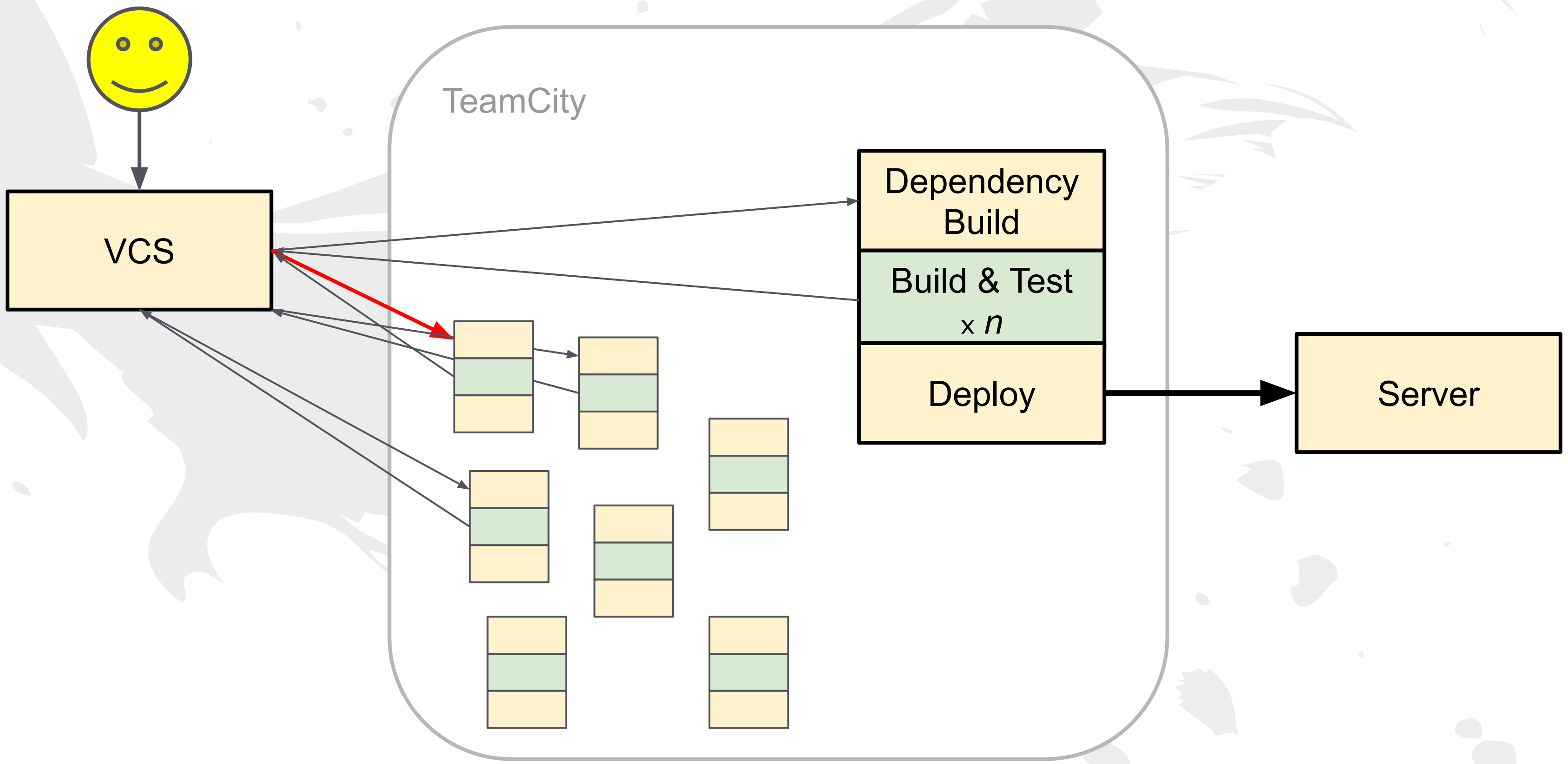
If we want the result of something down the chain, we have to wait for several triggers (or fire them ourselves, waiting for each in turn).

Each of these has to be queued, and uses resource shared with all the other teams

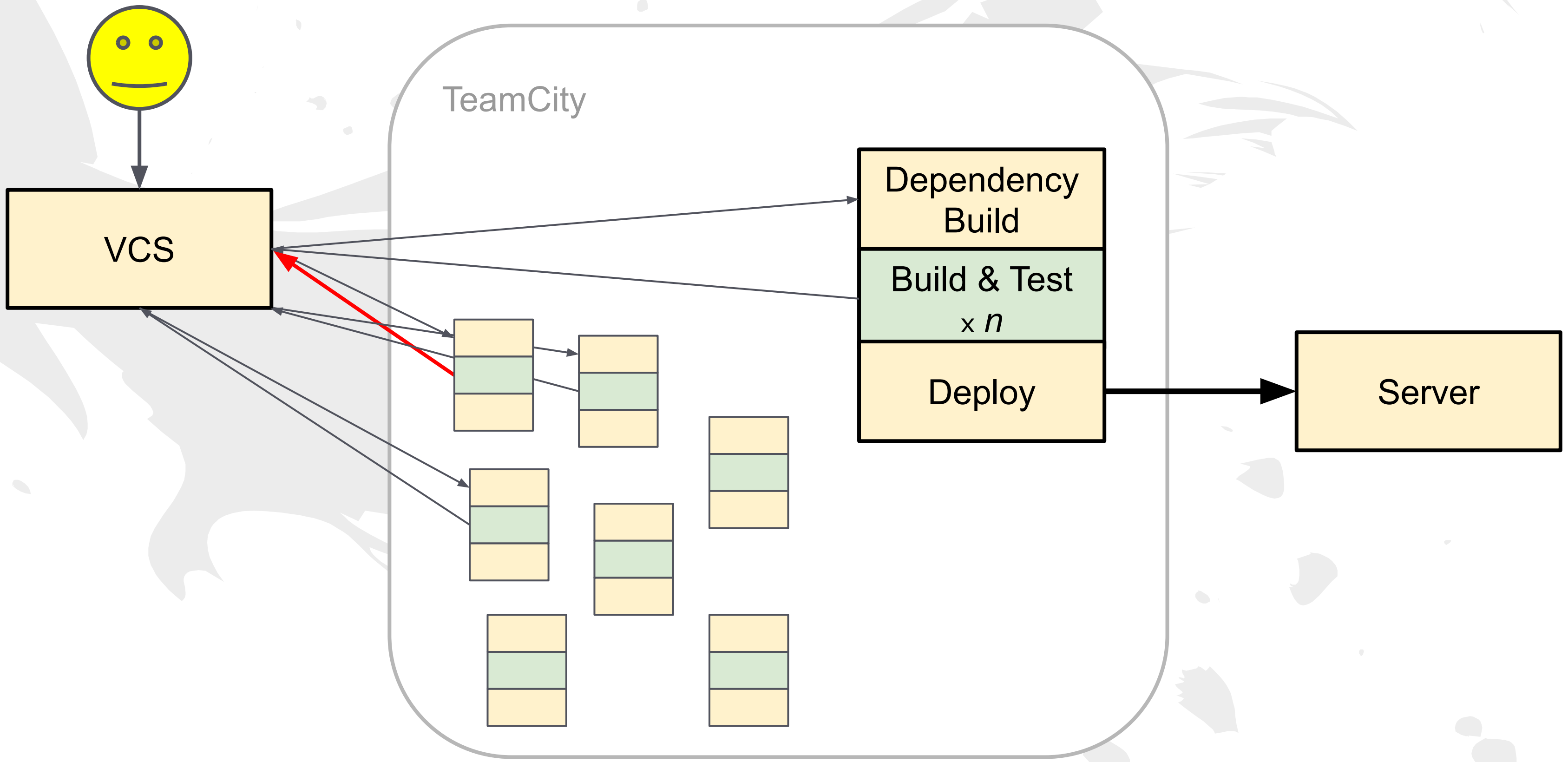
Waiting



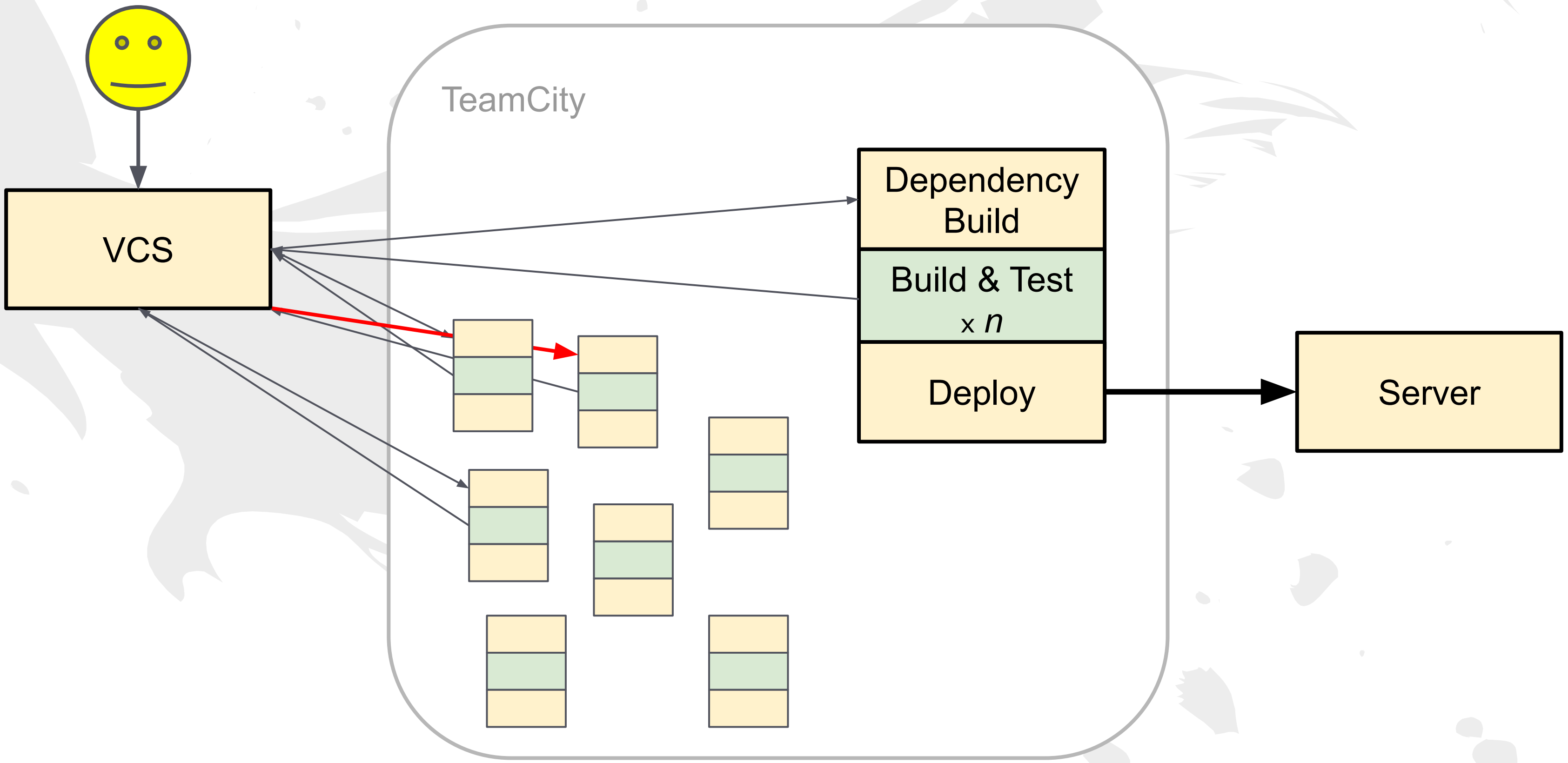
Waiting



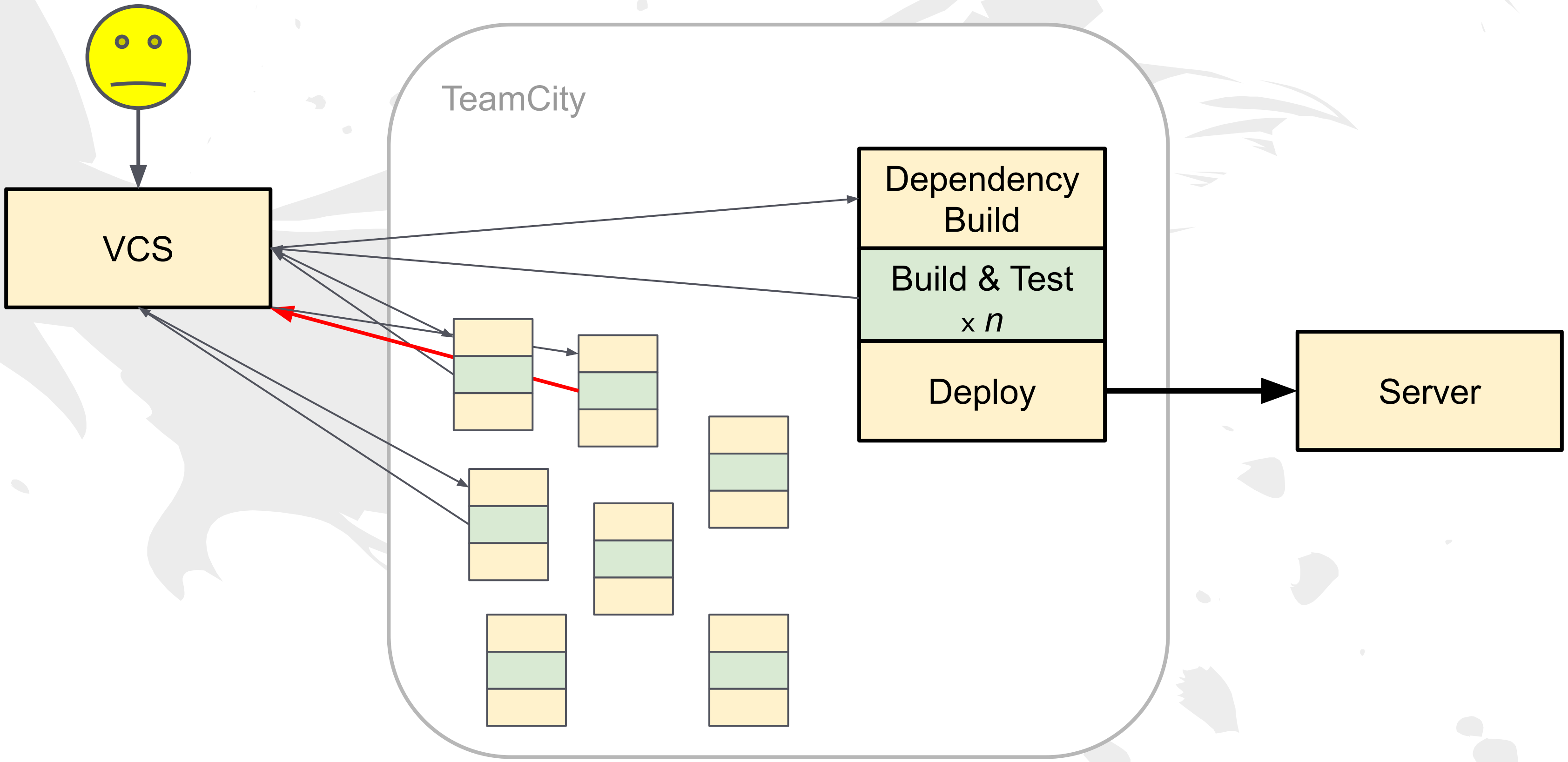
Waiting



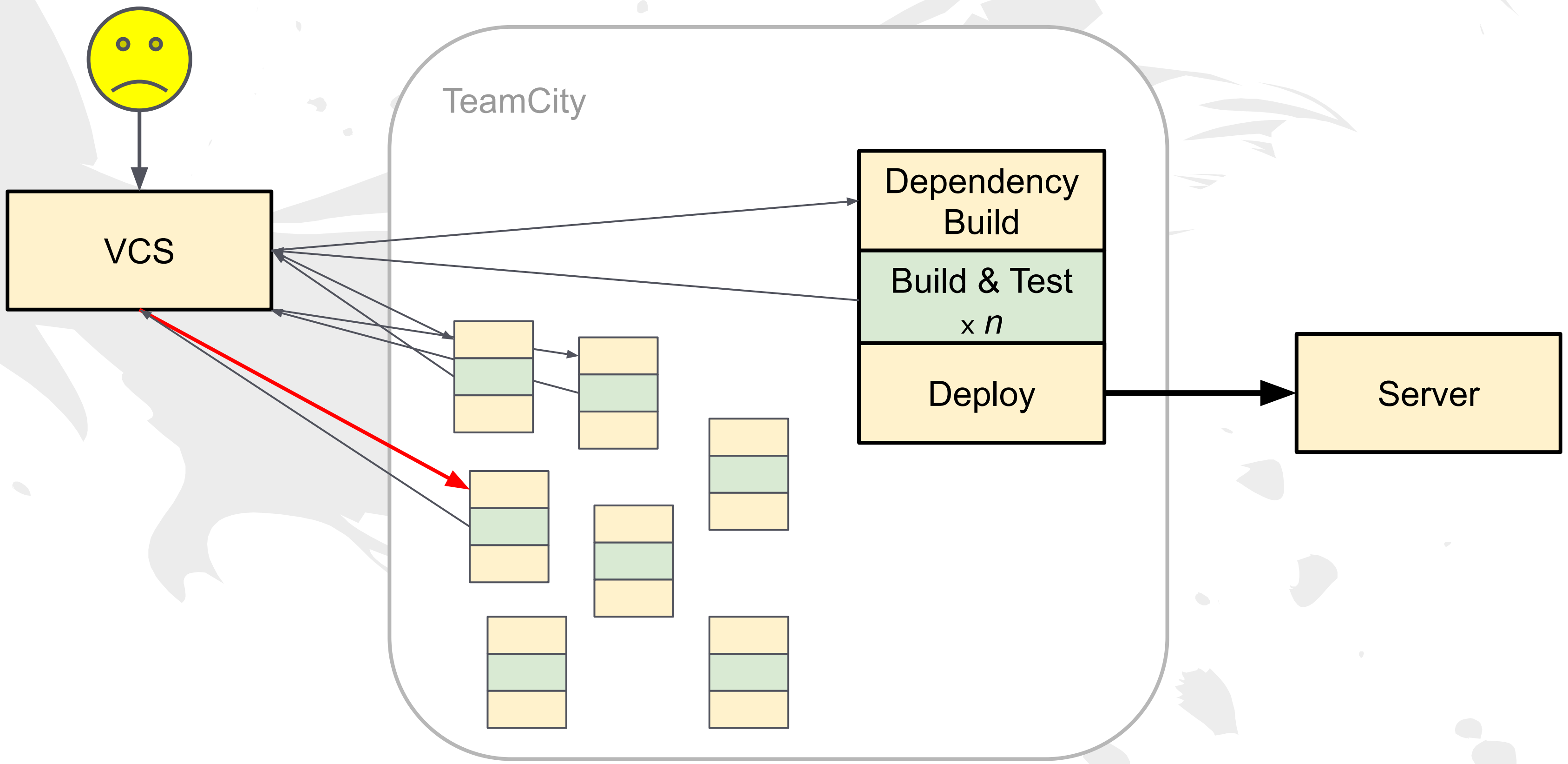
Waiting



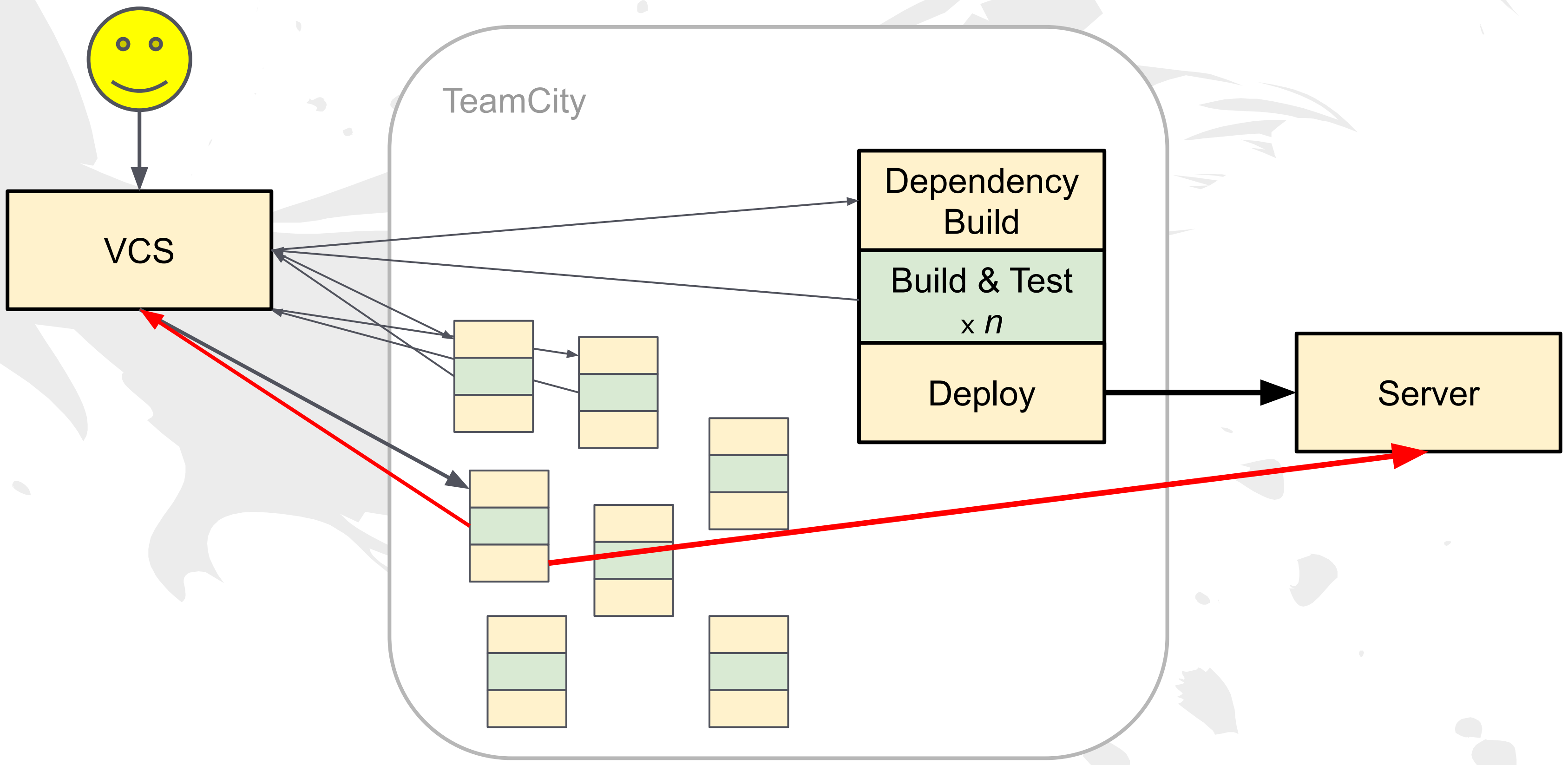
Waiting



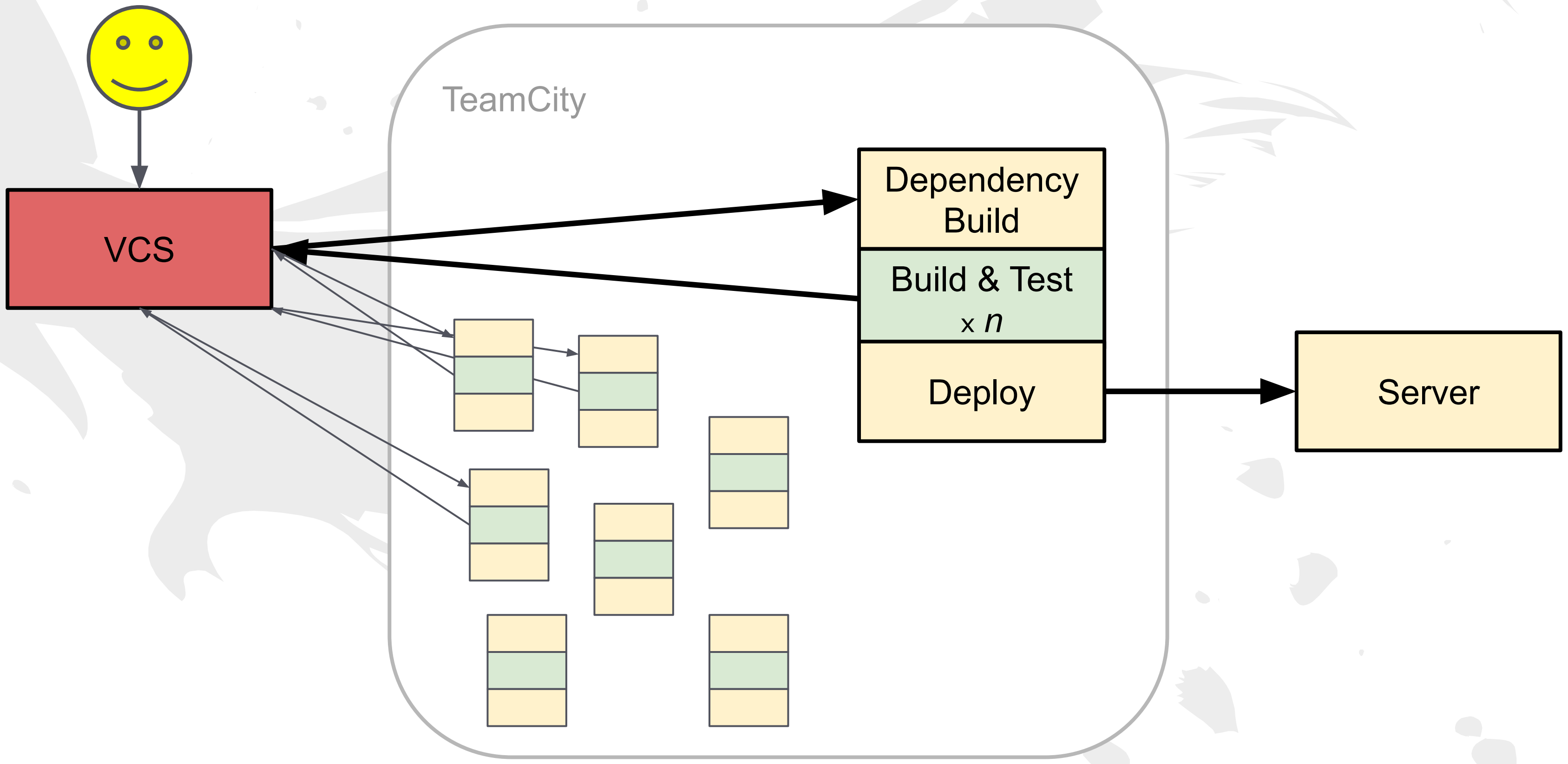
Waiting



Waiting

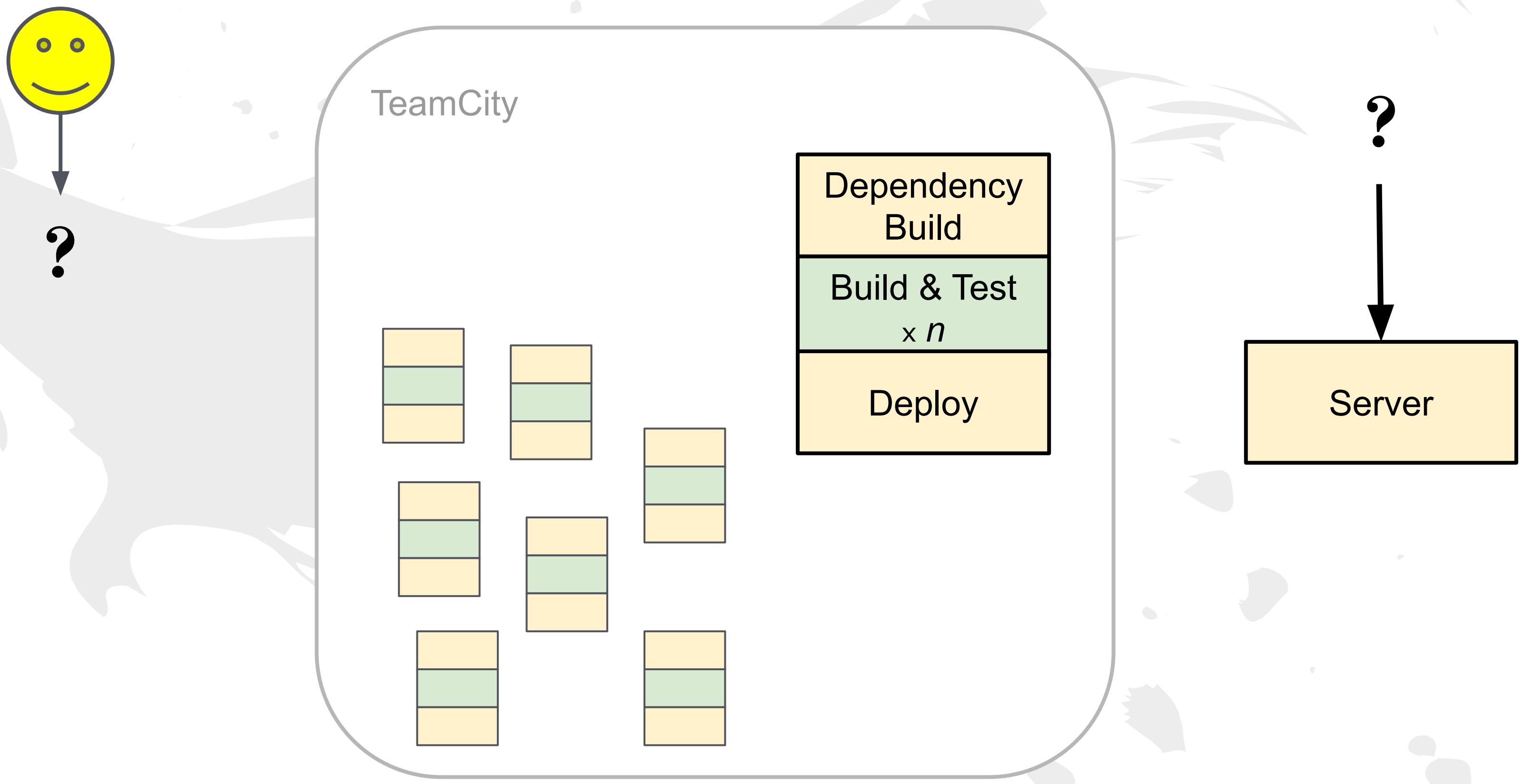


Failure cases: VCS overload



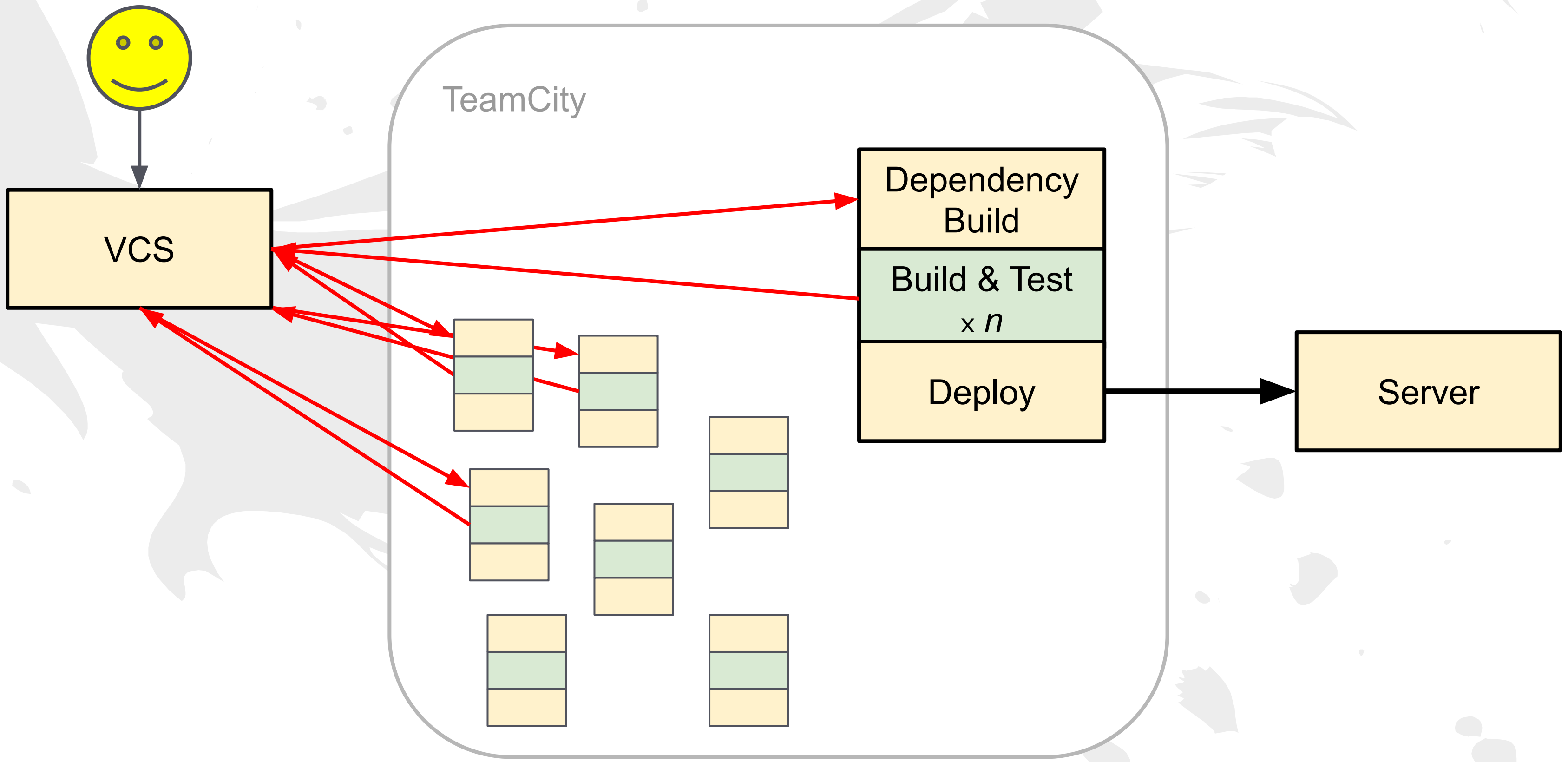
The version control server gets hammered with every dependency tree update, and ends up with a lot of commit history which is non-descriptive "Auto commit from dependency build. Build number *n*."

Failure cases: VCS dead



Working across dependency lines with no VCS? You're outta luck pal!

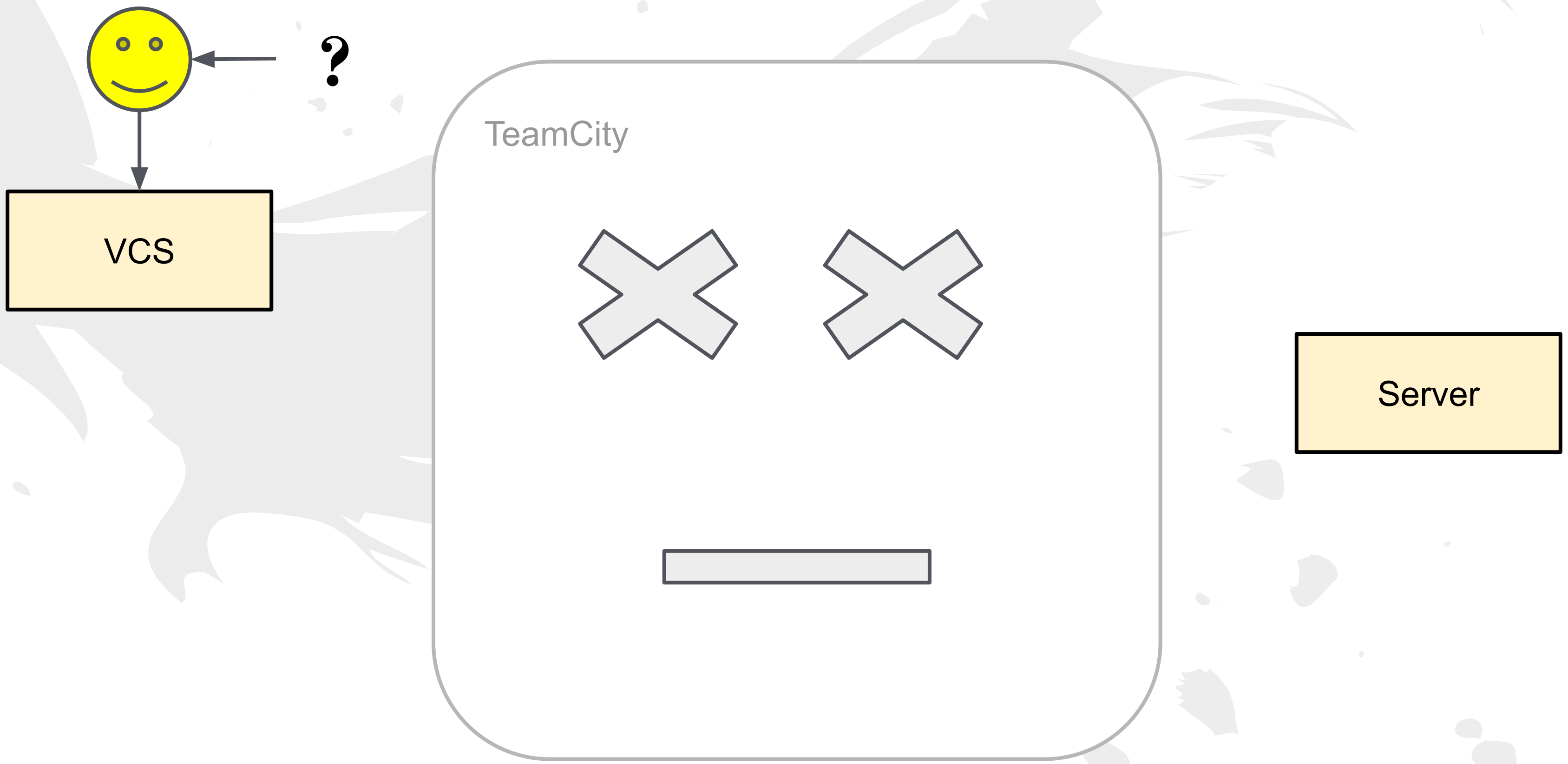
Failure cases: TC Build Storm



The TeamCity server gets cluttered with loads of builds, each of which is kicking off a dependency build. The queue gets full, the VCS server slows down (delaying the queue).

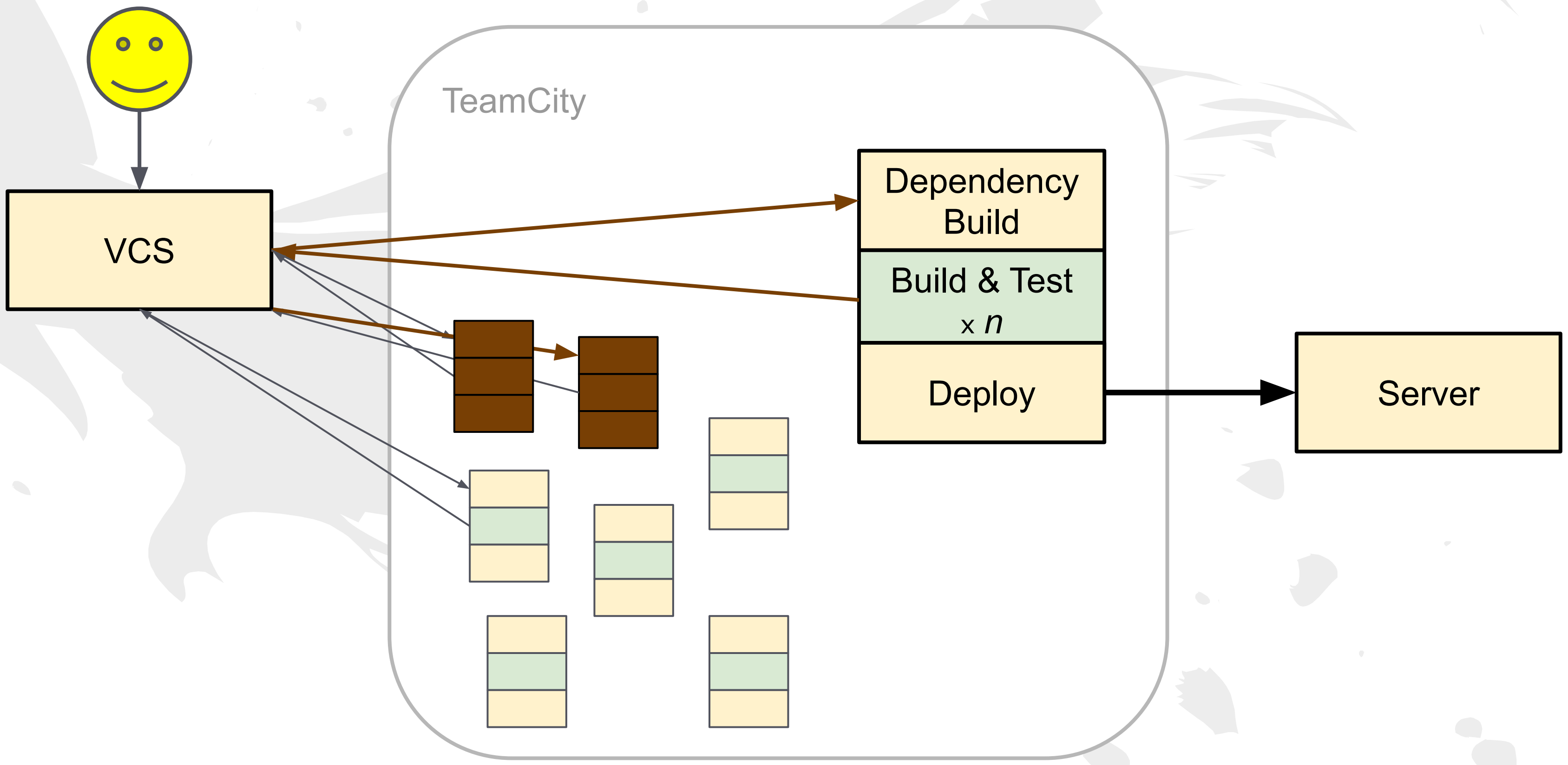
Result: long lunch (yay), no work done (boo!)

Failure cases: TC dead



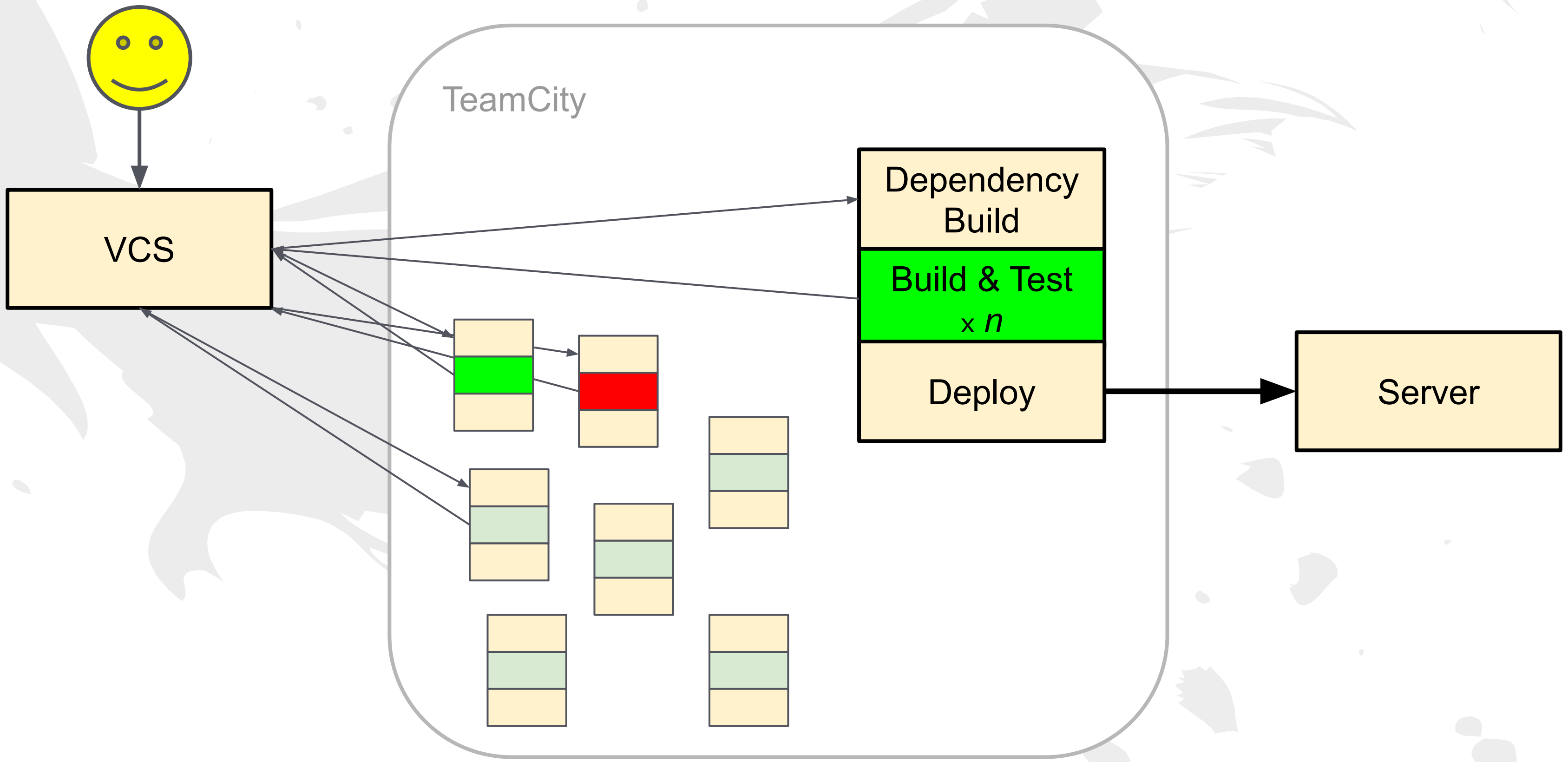
The TeamCity server experiences a failure, configuration change or blows up... can't update dependencies, can't work across solution boundaries

Failure cases: Poison Build



A spike code change or edge-case break gets into the VCS, The dependency builds kick off, products break and we all have to down tools to find and fix the problem. People are smart enough not to do this twice, but it makes people timid of trying far-reaching changes -- because it's a pain in the ass.

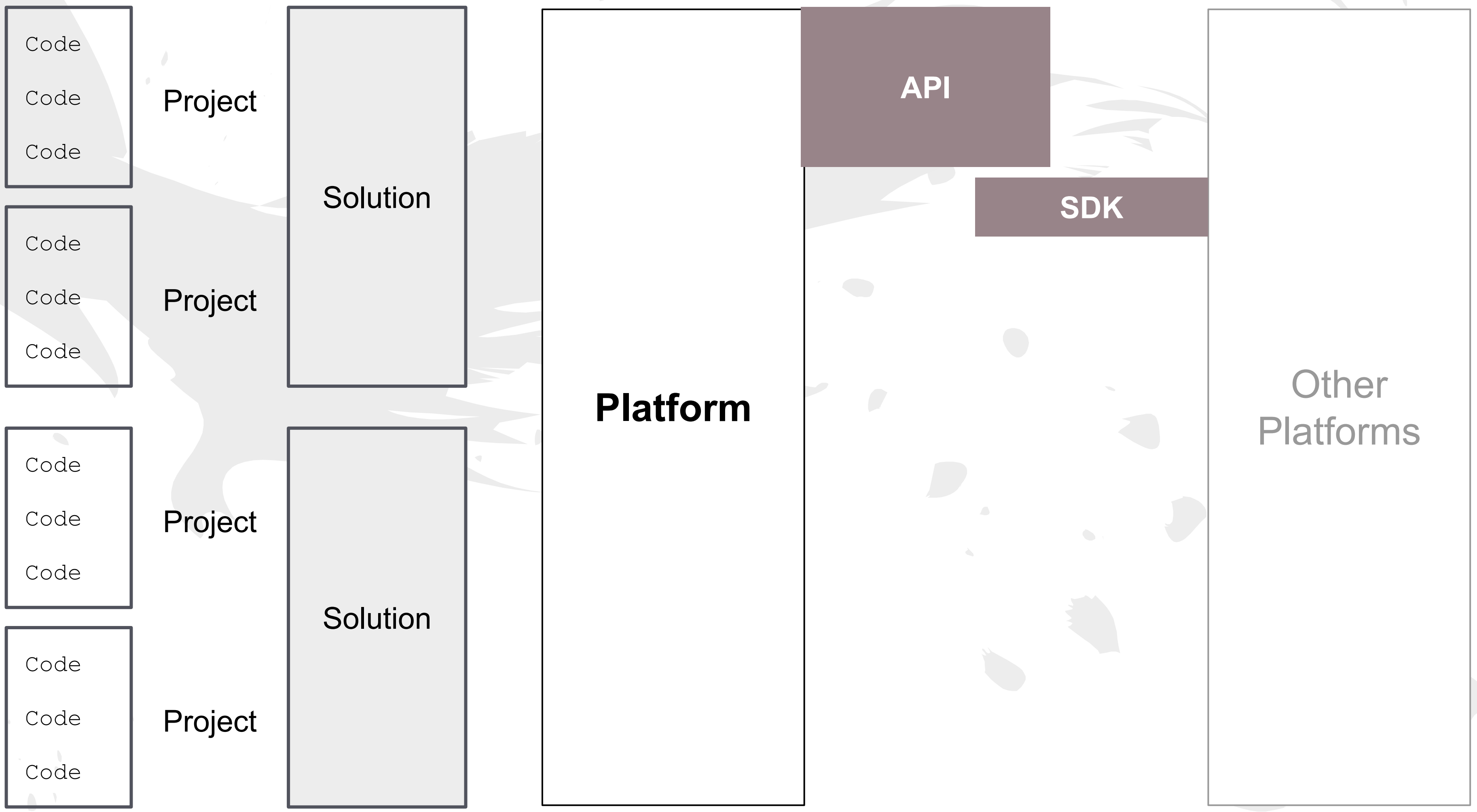
Failure cases: Dependency Rash



We make a change that is *green* inside the project, but causes a failure down the line. Our Integration System spends a lot of time red, meaning a lot of time we can't deploy.

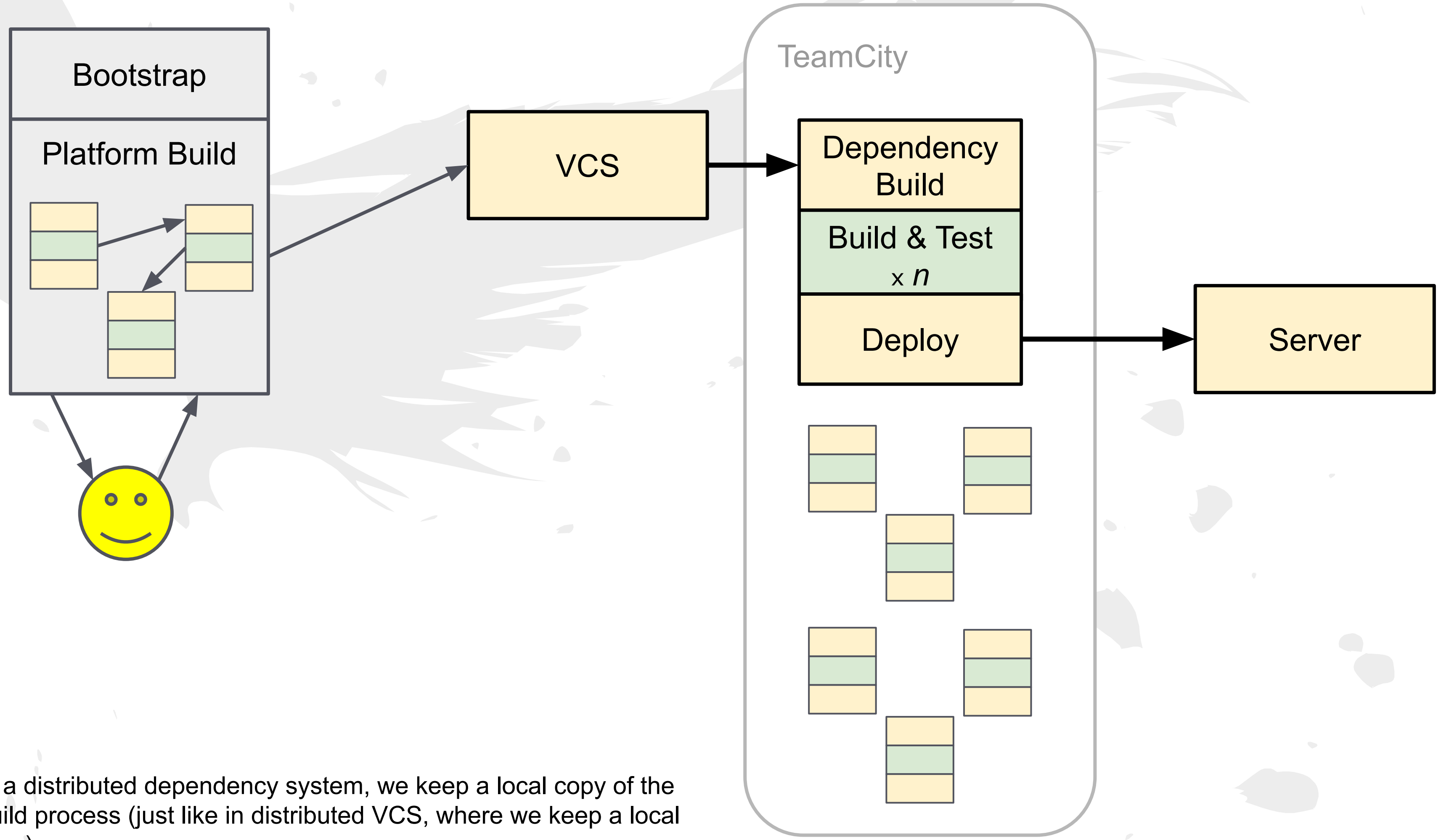
Distributed Build System

Some naming conventions



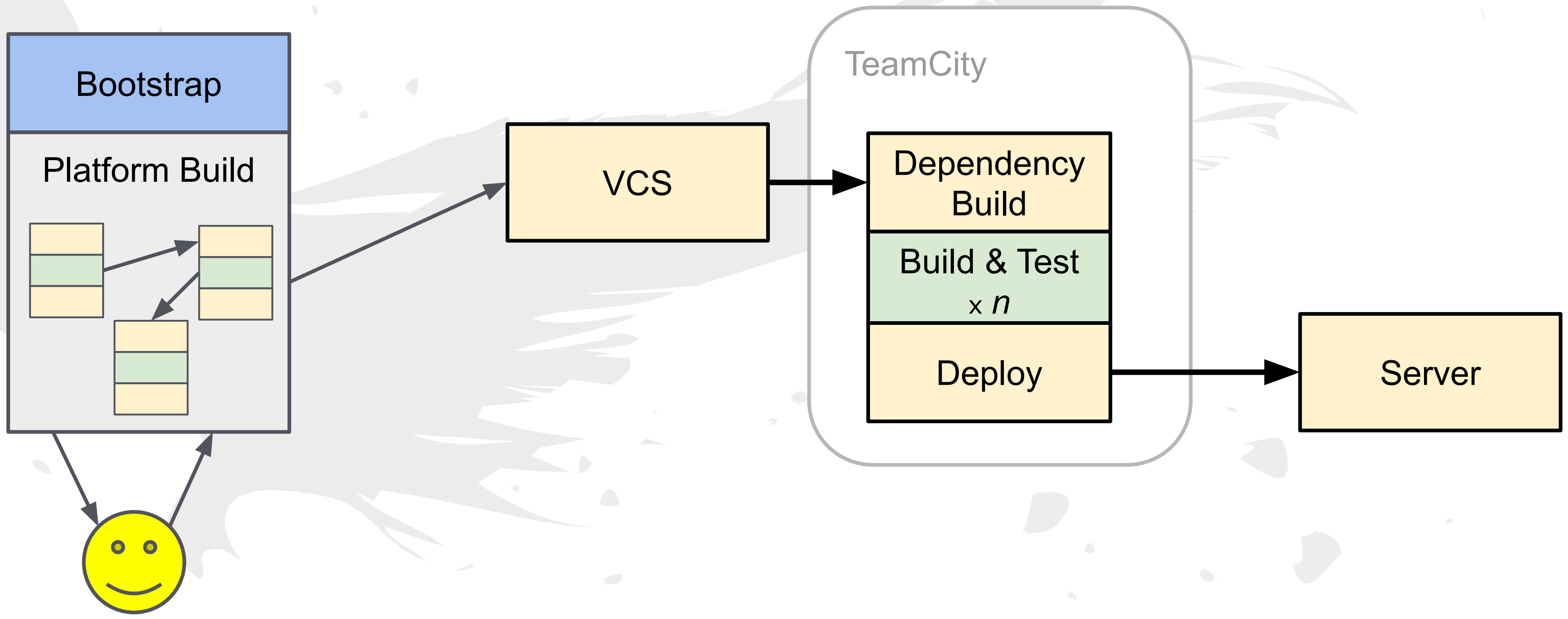
We have the familiar 'collection of code is a project' and collection of projects is a solution.
We introduce the 'collection of solutions is a platform'.
Next level up, a collection of platforms is a company.
Platforms only every communicate through APIs and SDKs. They never share components -- no binaries, no databases.

Distributed build platform



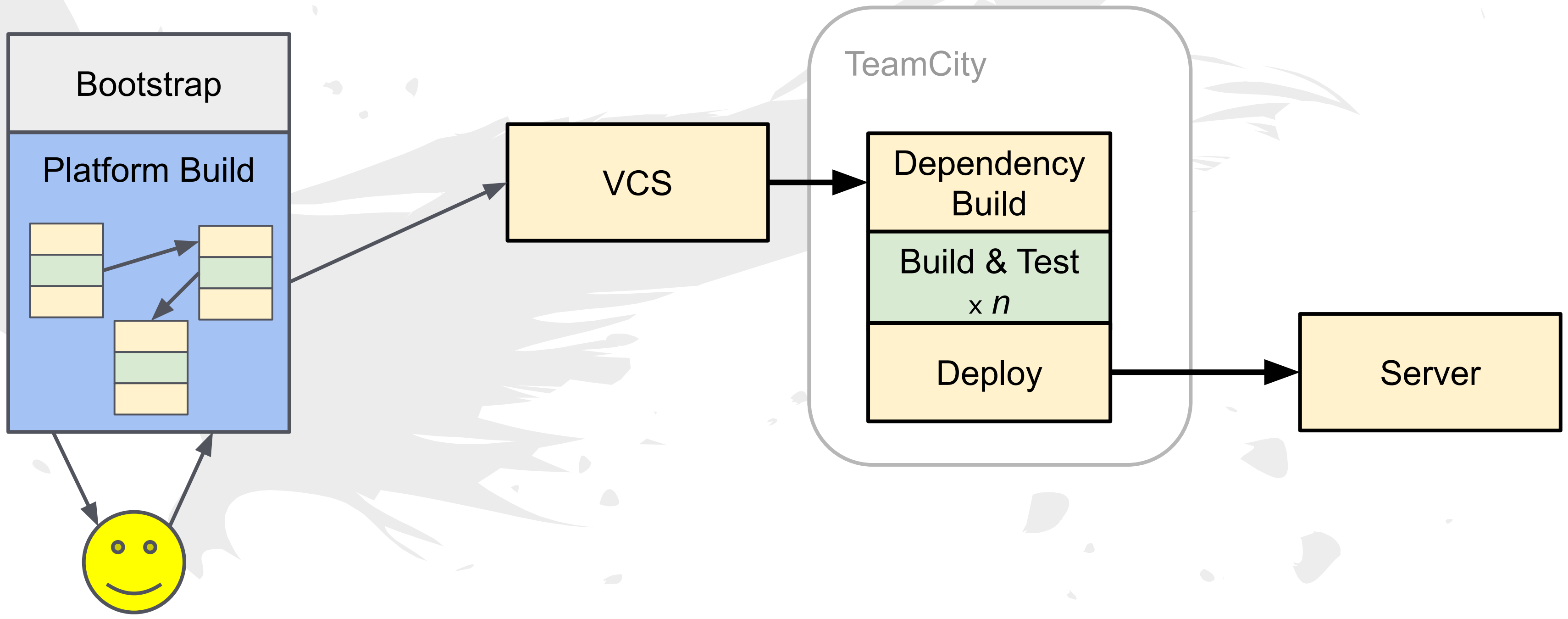
In a distributed dependency system, we keep a local copy of the build process (just like in distributed VCS, where we keep a local repo)

Distributed build platform



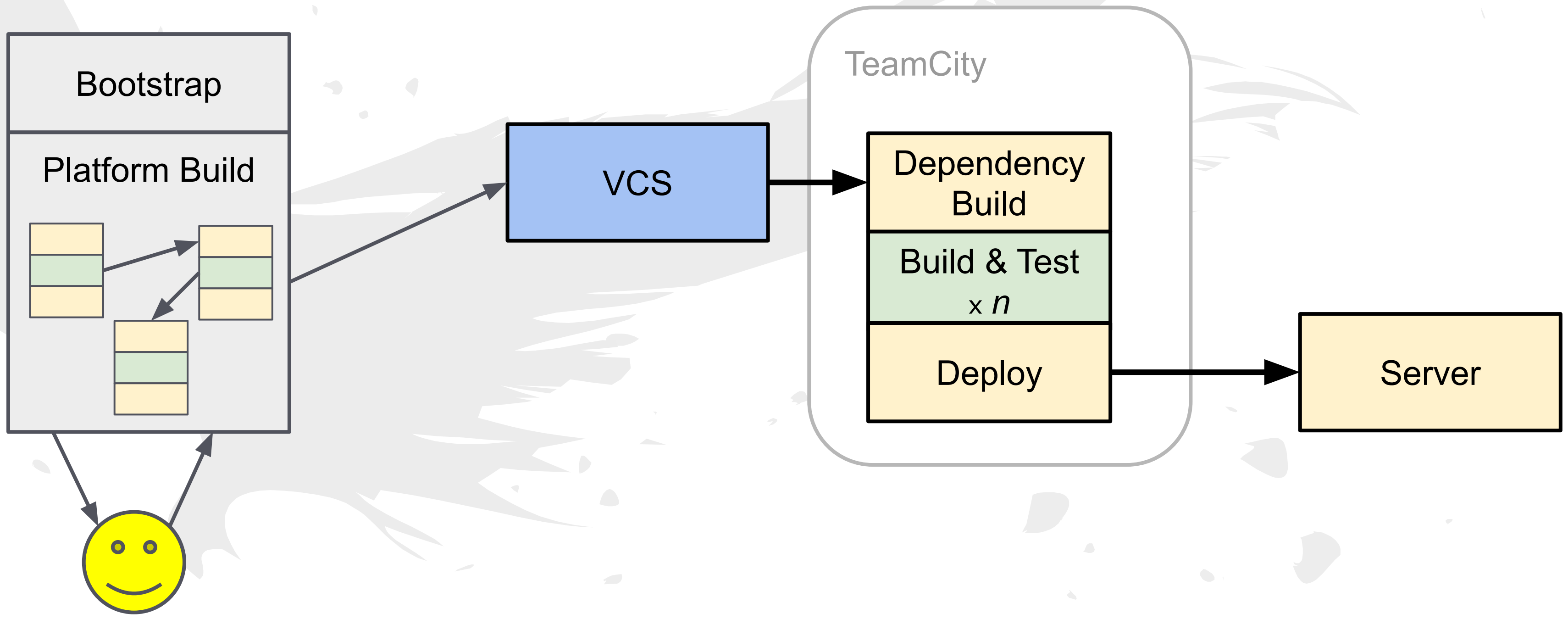
The bootstrap ensures that we have all the necessary components to start working on the platform (all the code, frameworks and tools)

Distributed build platform



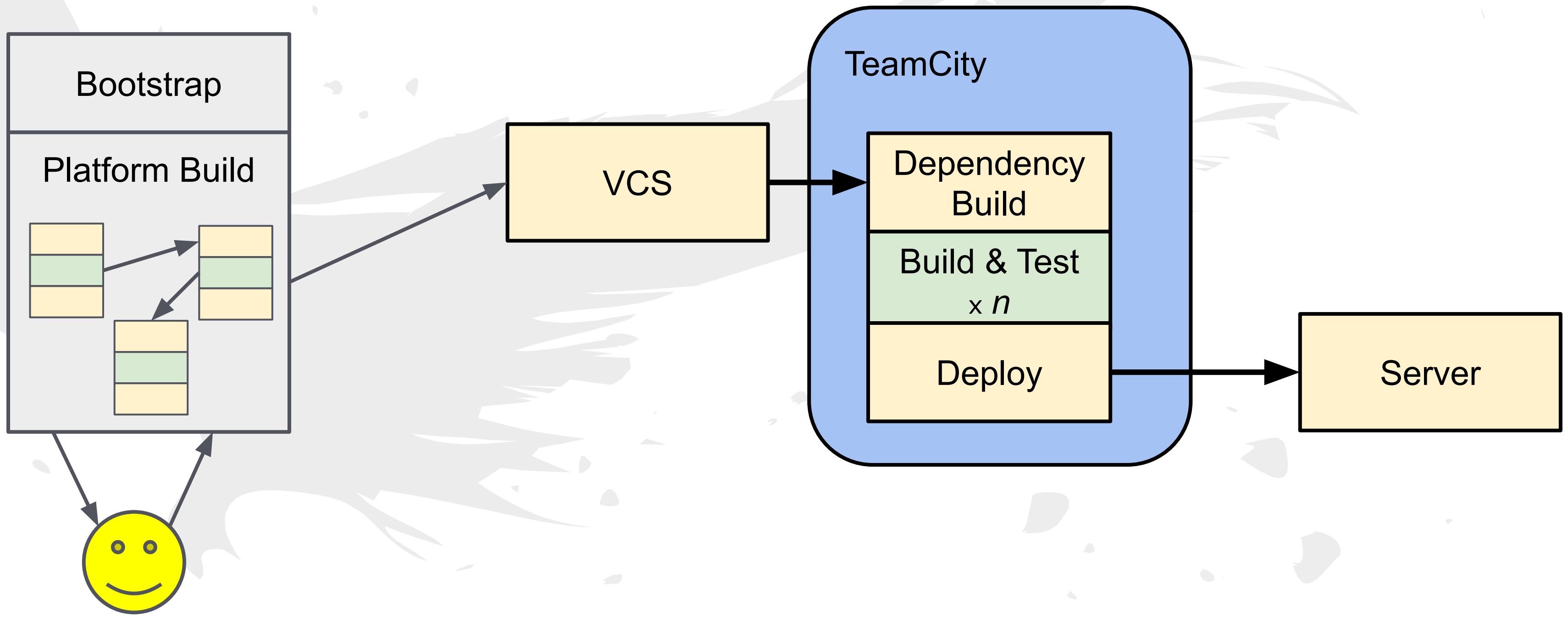
The platform build does all the chain-build and dependency management that has been done with TeamCity in the past

Distributed build platform



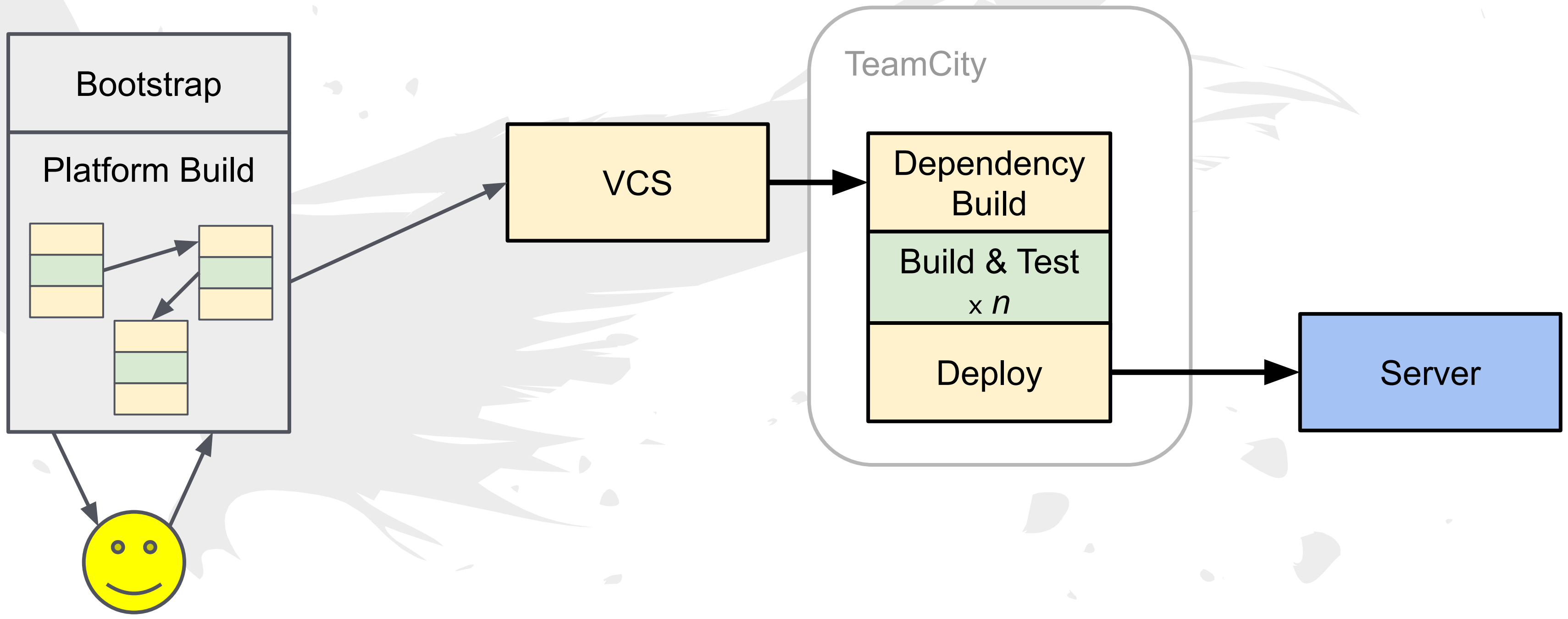
Version control is used exactly as before, it's just plain ol' Git

Distributed build platform



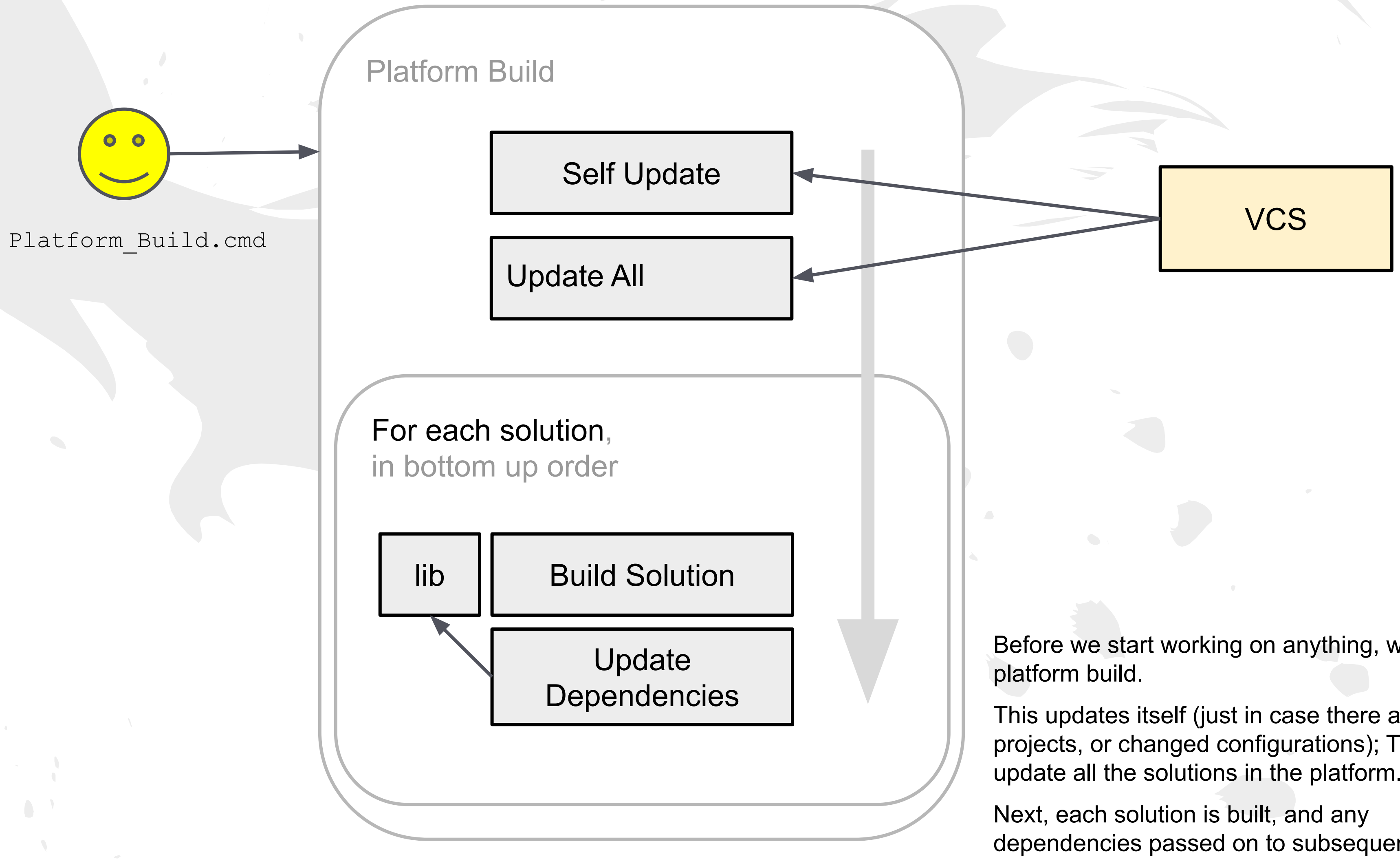
TeamCity handles just two jobs -- doing Continuous Integration tests and deployment tasks
(the things it's best at!)

Distributed build platform



Servers and the live system are exactly as before.

Inside Platform build

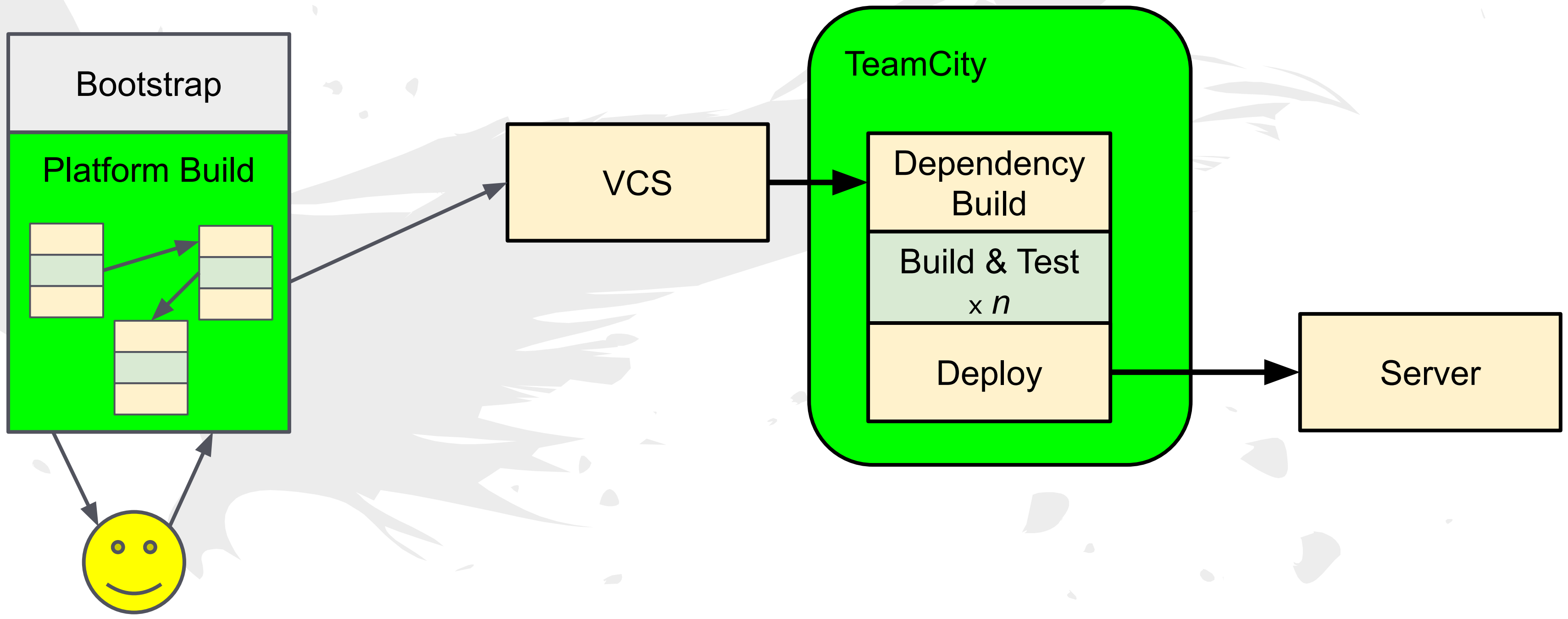


Before we start working on anything, we platform build.

This updates itself (just in case there are new projects, or changed configurations); Then we update all the solutions in the platform.

Next, each solution is built, and any dependencies passed on to subsequent solutions (through the 'lib' folder or similar)

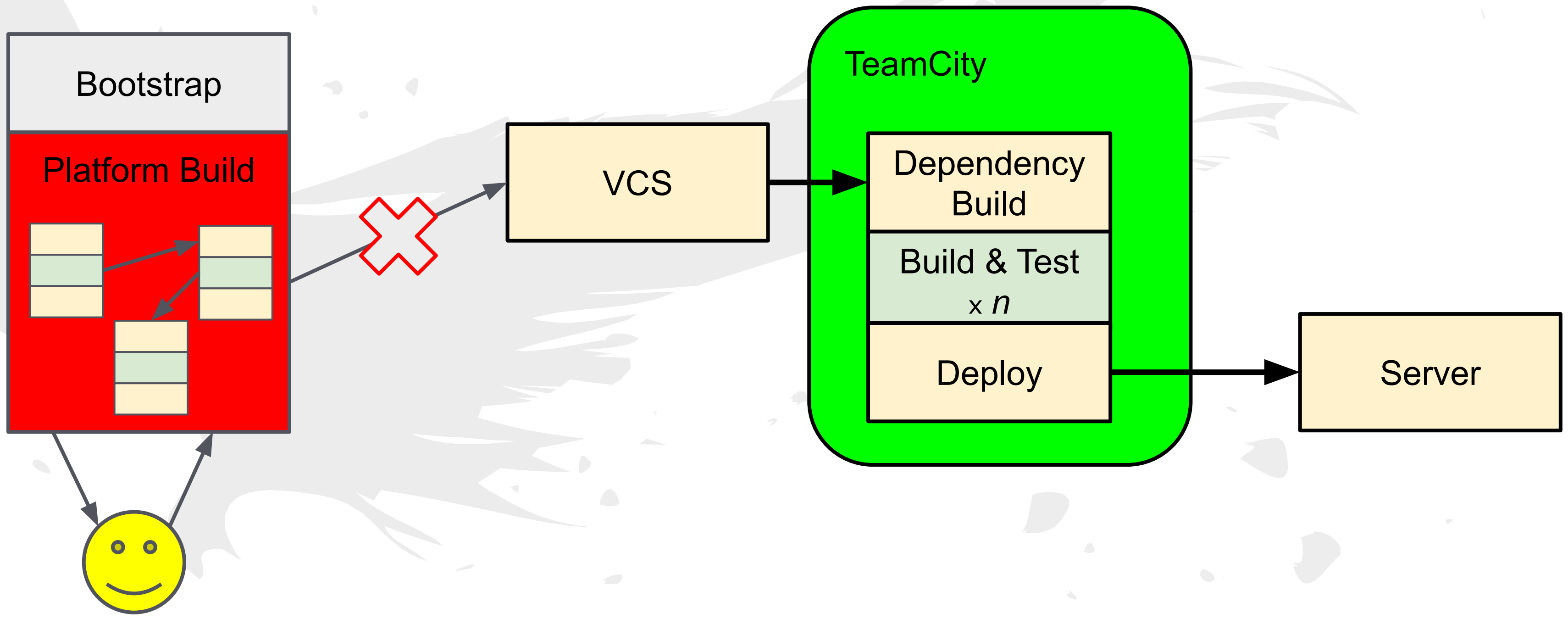
Always green



The local platform is the same as the integration platform, and all tests that are run can be run on both.

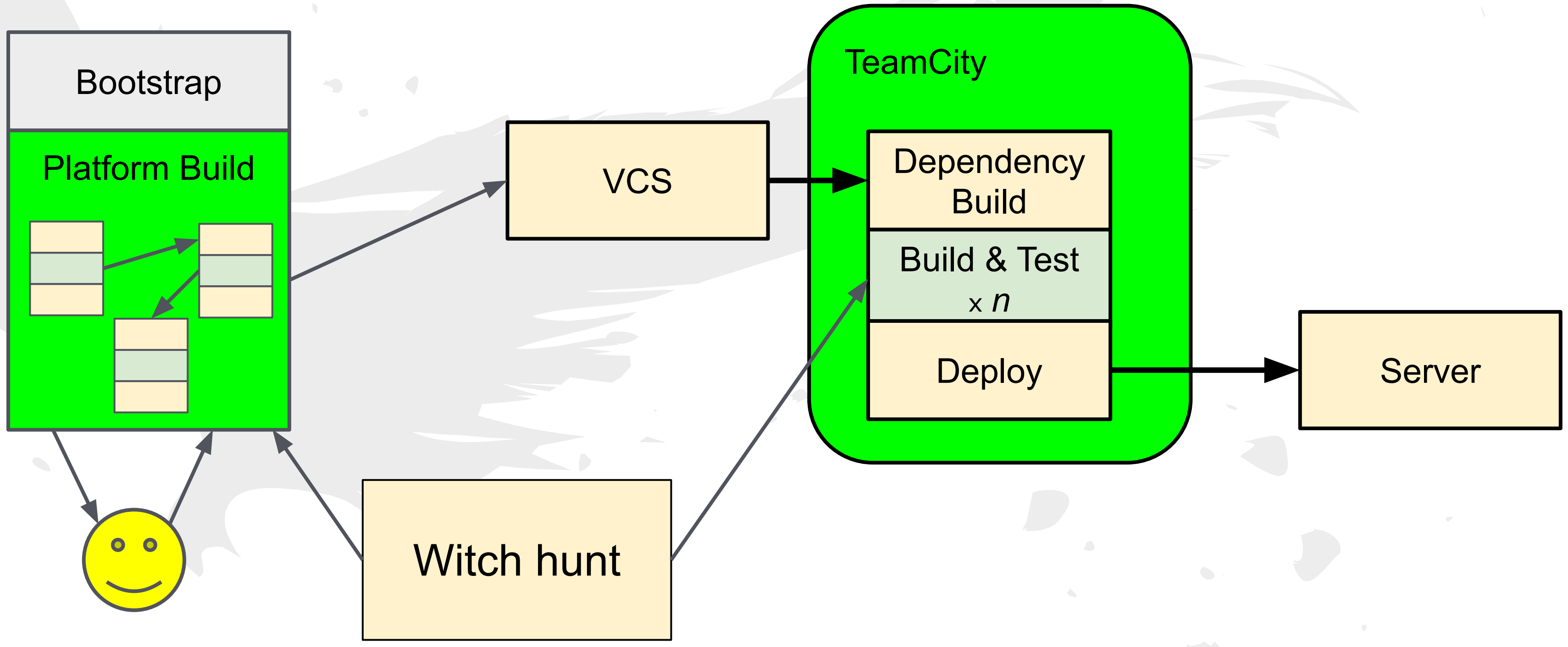
The entire platform gets committed green, so the build server stays green as much as possible.

Safe spike



We can keep building and rebuilding the entire platform locally without pushing our changes, so we can try big changes without stopping anyone from working or deploying.

All tests!



External test sets can be run locally, greatly reducing cycle time in case of failure.

github.com/i-e-b/GitBuildPlatform

Links