Configuration Management

Disclaimer

- The information and opinions expressed in this presentation and on the following slides are solely those of the author.

- The following slides are based on:
  - Christopher Parnin's used in the course "DevOps: Modern Software Engineering Practices" at NC State University.
  - Jez Humble; David Farley: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010), Chapter 2.

Facts [5, Chapter 25]

- Software systems are constantly changing
  - New requirements emerge when the software is used
  - The business environment changes
  - Bugs are discovered and have to be fixed
  - New hardware and system platforms are released
  - The performance or reliability may have to be improved

- Changes are made => a new version of the **Software System** is created
  - Software System = < Product*, Prod Environment>

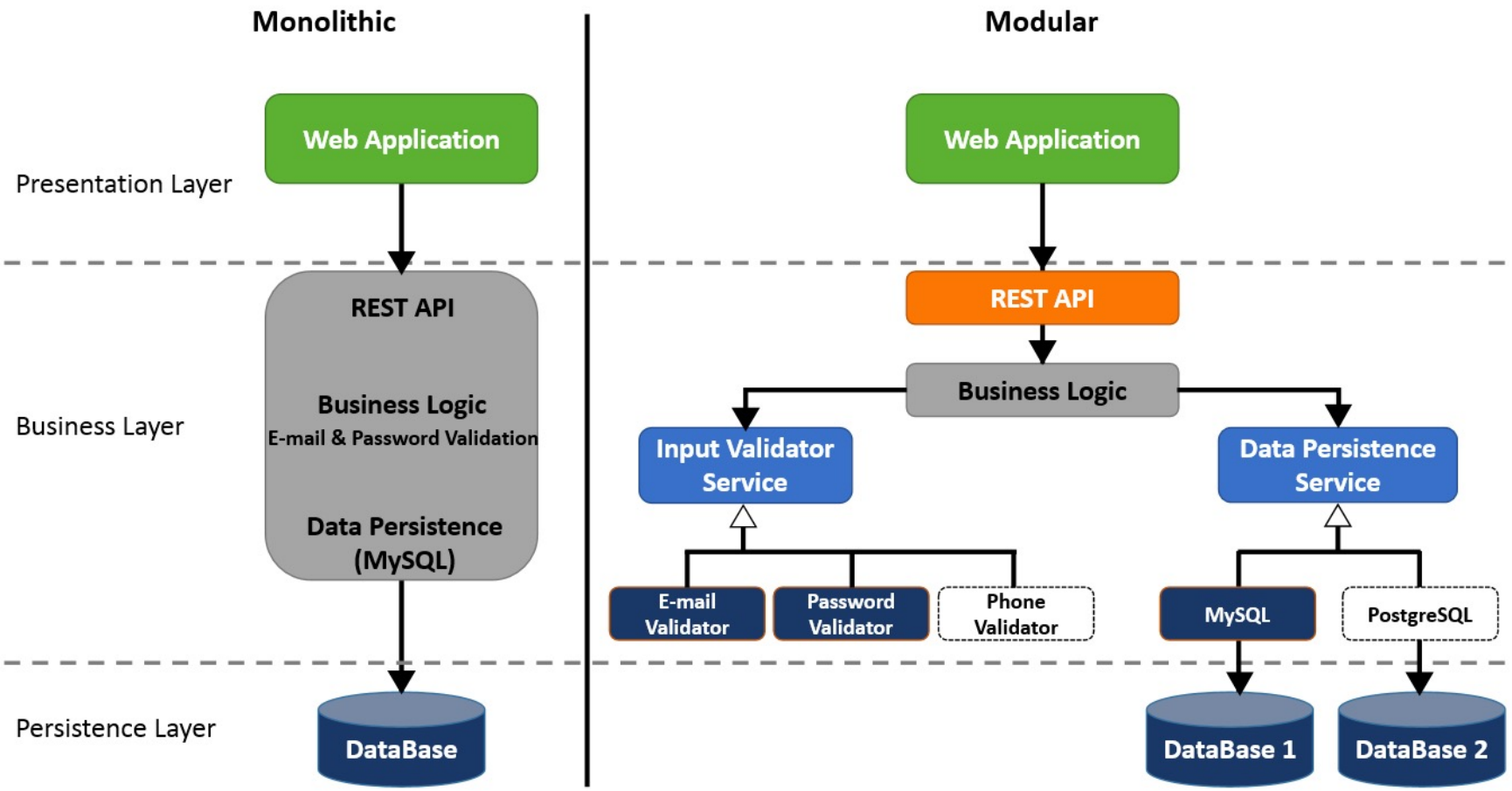
(*) “Product” is synonymous of “Application”.

In our context

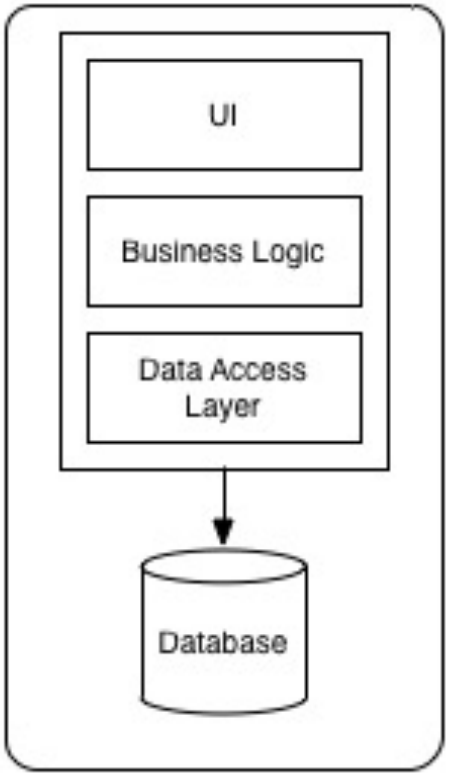
- The Product:
  - “Binaries”
  - Configuration
  - Data
- Environments:
  - Dev
  - Integration
  - Stage(s)
  - Production

What it can be changed “often”

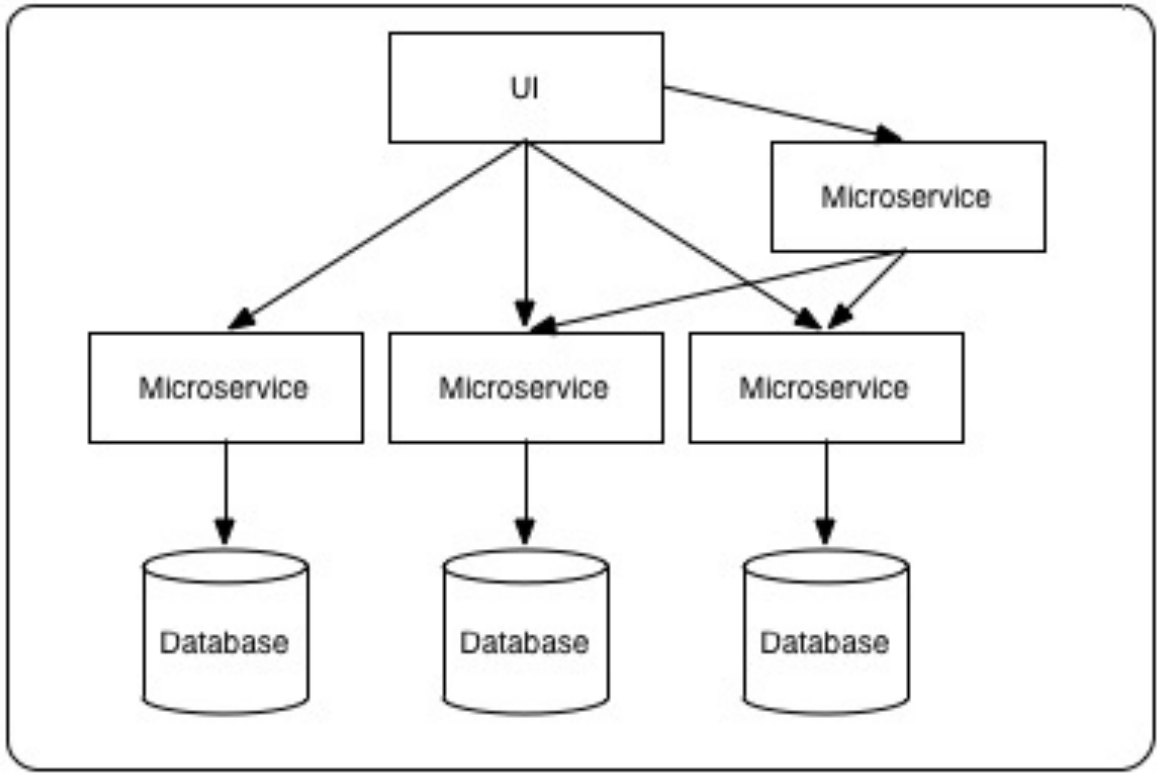
Encapsulated software development



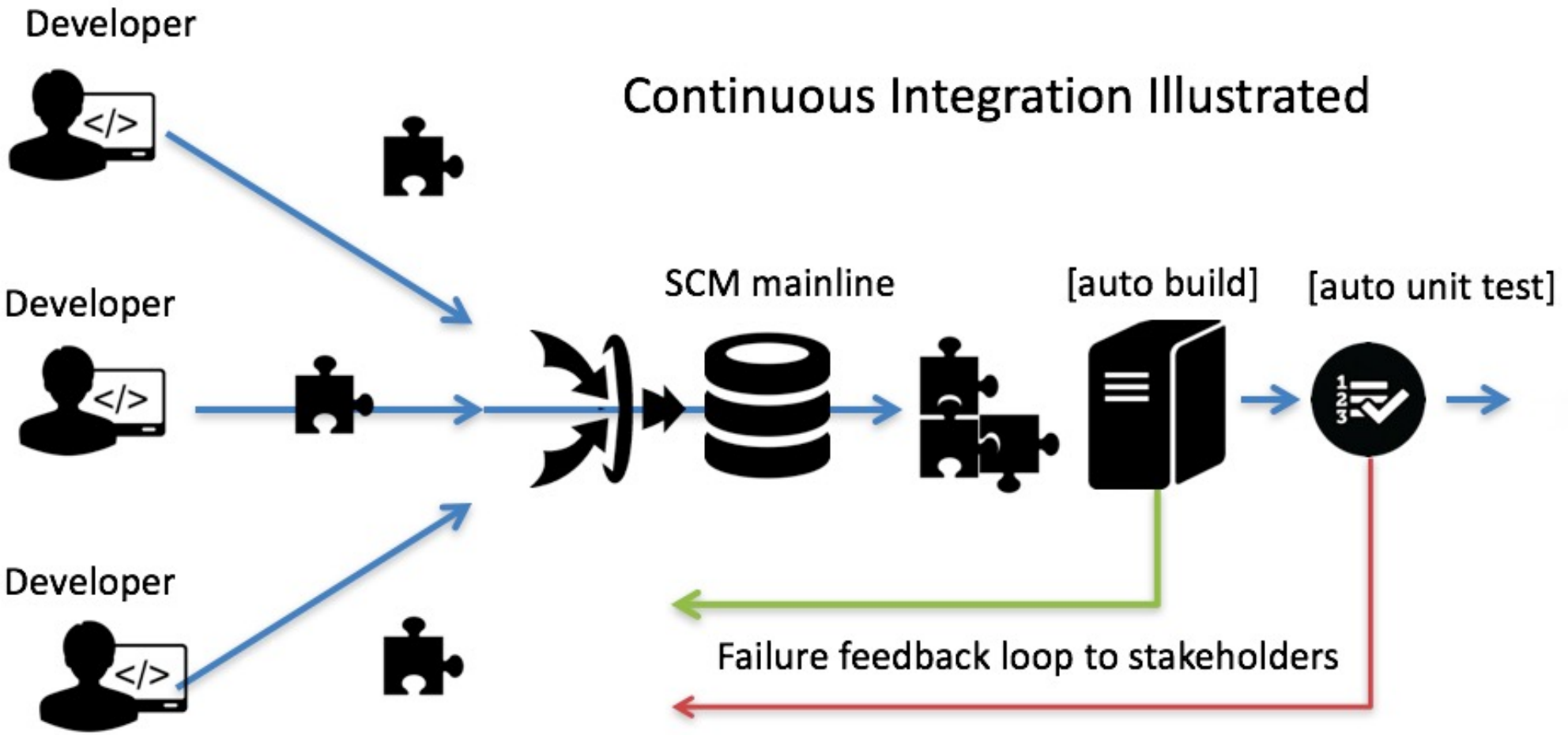
Microservices



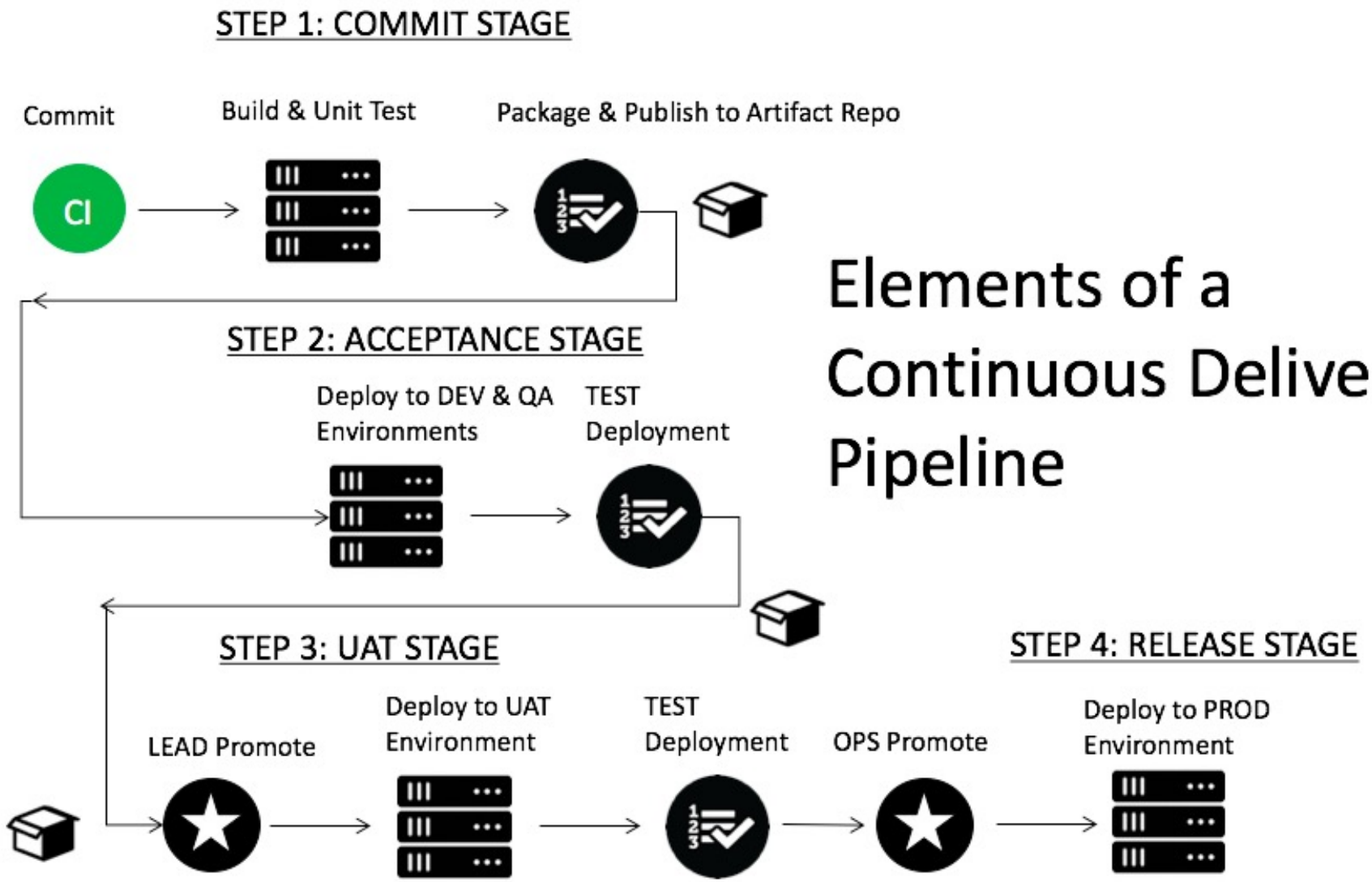
Monolithic Architecture



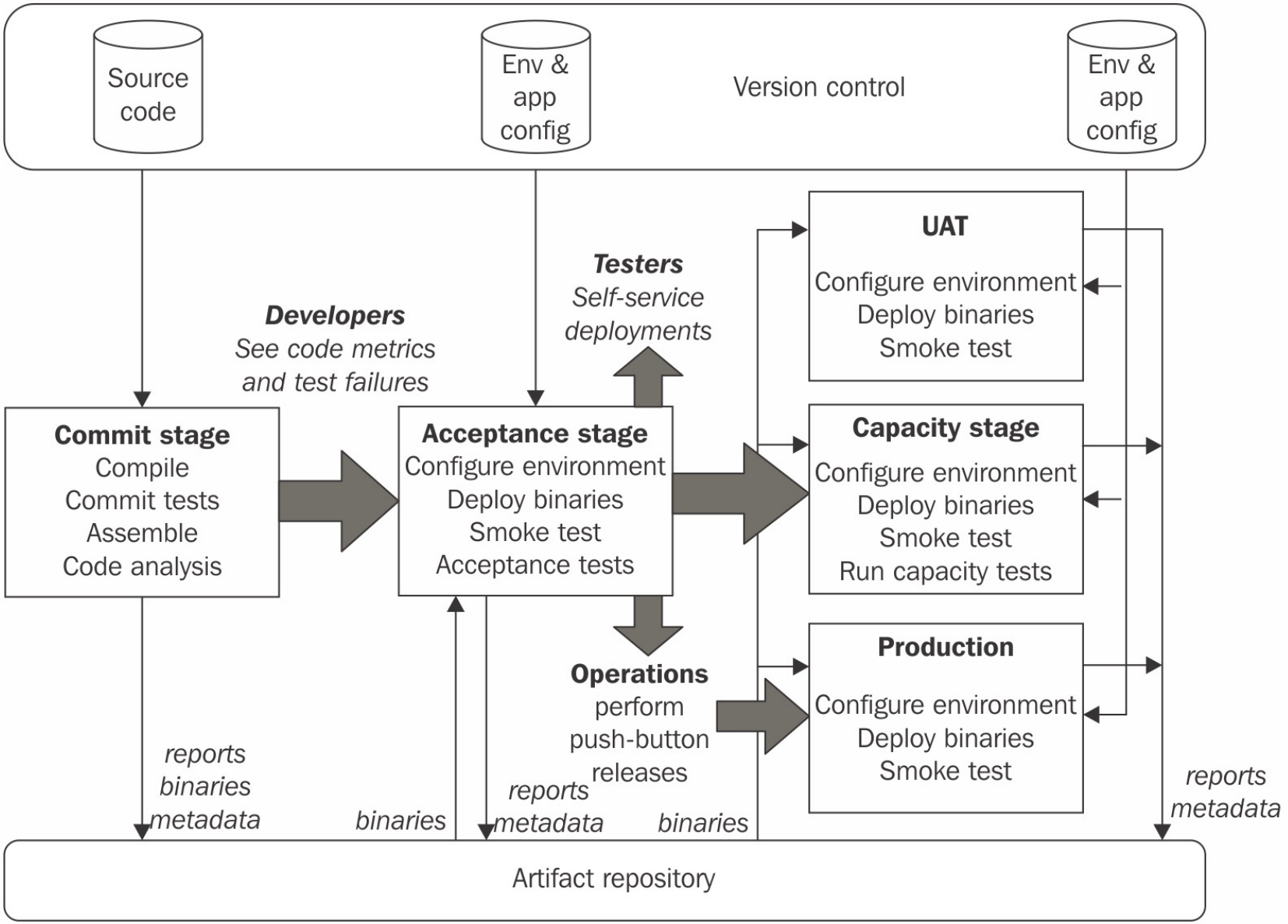
Microservices Architecture


Developer
Continuous Integration Illustrated
Developer
SCM mainline
[auto build]
[auto unit test]
Developer
Failure feedback loop to stakeholders

Elements of a Continuous Delivery Pipeline



Components of continuous delivery

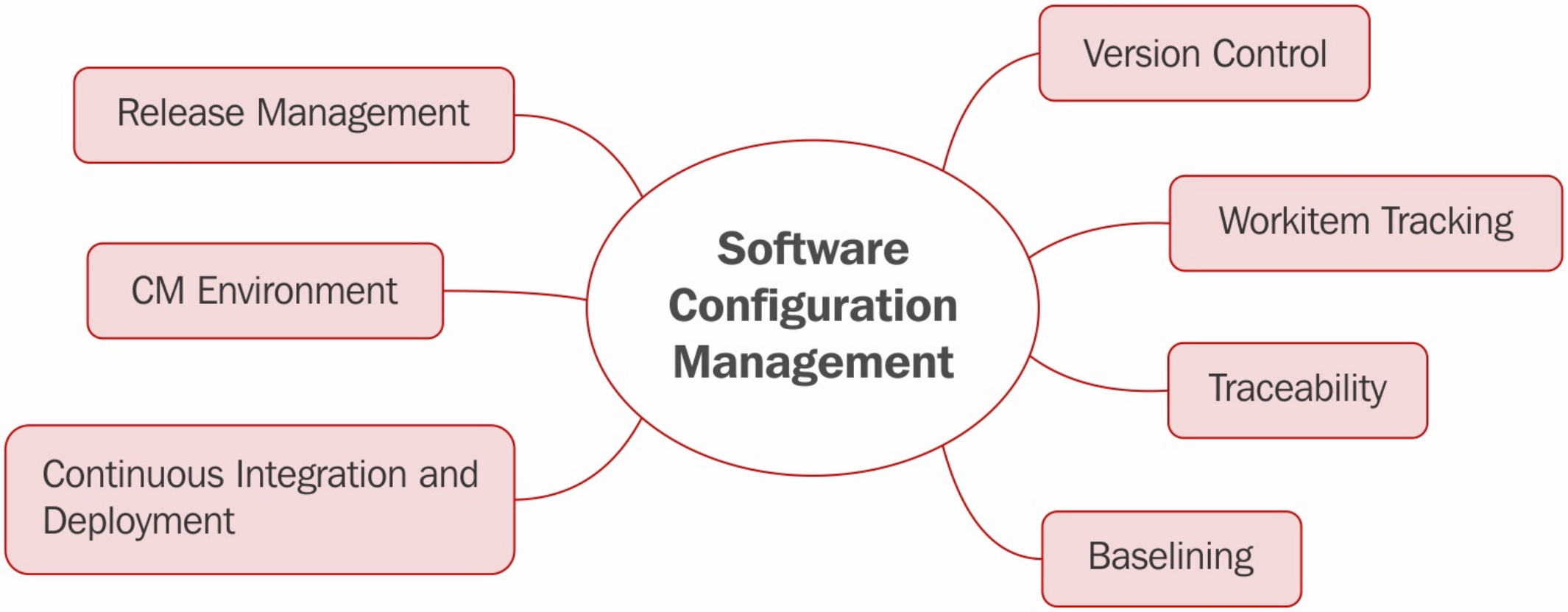


Motivation

These are the goals of good Configuration Management (CM)

- I want to “easily”*:
  - know what constitutes a particular version of the **product**.
  - reproduce a particular version of the **product** that existed in production
  - reproduce a particular **environment** (including OS version, network configuration, …)
  - see each change that occurred to a particular **environment** and trace it back to see exactly what the change was, who made it, when and for what reason.
  - create a local **environment** to make a small “bugfix” without any chance of facing with “code regression” issues.
  - make my changes **available** to every team member

(*) Easily : without having to resort to “heroic” efforts.
Software regression: it is a software bug that makes a feature stop functioning as intended after a certain event (for example, a system upgrade, system patching or a change to daylight saving time)


Release Management
CM Environment
Continuous Integration and Deployment
Software Configuration Management
Version Control
Workitem Tracking
Traceability
Baselining

Goals of configuration management

- To track changes made to a given system or set of systems
- To provide traceability and auditability for defects that may arise as part of a set of changes made to a given system
- To help reduce the amount of manual effort made by developers, QA, and operations folks by maintaining a set of automated solutions that can aid in the provisioning and configuration of a given system
- To provide a level of repeatability to the organization by clearly defining (in automation form) the steps required to build out a given system

Horror Story

We had to reproduce a System (i.e. Product + Environment) which was released in 2004.

Note: no Configuration Management tools available!

Challenge:

- reproduce a particular version of the product that existed in production
- reproduce the environment (OS version, network configuration, …)

Configuration Management (CM)

- From “Systems and software engineering — Vocabulary”, ISO/IEC/IEEE 24765:2017(E):
  - *“**1**. discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, **control changes** to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1]*
  - ***2**. technical and organizational activities, comprising configuration identification, **control**, status accounting and auditing [IEEE 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering, 2.1]*
  - ***3**. Coordinated activities to direct and **control** the configuration [ISO/IEC TR 18018:2010 Information technology — Systems and software engineering — Guide for configuration management tool capabilities, 3.7]”*

Other definitions for CM

- *“It refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified.”* [1, pp 31]

- *“It is the task of tracking and* ***controlling changes*** *in the software.”* [2]

- “*It is concerned with the policies, processes and tools for* ***managing changing*** *software systems.*” [3]

Functional areas of CM [3]

Note: **modern** configuration management => these areas are integrated throughout the process

- Source code management
  - Safeguard all the projects resources
  - Baselining the code => record specific milestones
  - Create code variants => parallel dev, bugfixes
- Build engineering
  - Compile, link, package code components (for a specific variant)

Component : smallest part of the Product

- Environment configuration
  - Manage code dependencies (e.g. libraries, services) and the environments themselves
- Change control
  - Grant permission for changes, control promotions from one stage to another one
- Deployment/Release engineering
  - Identification of the components built through the build engineering process
  - Promotion of packaged components
  - Monitor environments => ensure that there are not unauthorised changes

Components of Modern CM

- Configuration tools
- Version control systems (git, branches)
- Package managers
- Build managers
- Environment handling (dev stacks, containers)
- Data migration

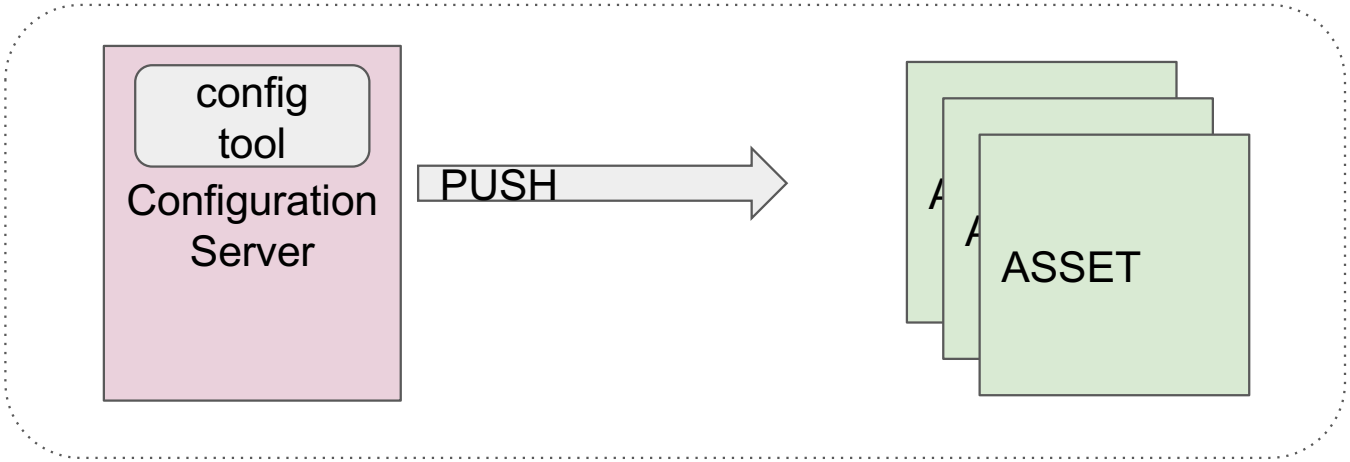
Configuration tools

- Ansible
  - “Playbook” scripts run over ssh
- Puppet
  - Less hands on, agents run on server.
- Vagrant
  - An interface for interacting with virtual machines.
  - Works with a VM provider (e.g. VirtualBox)
  - Public repository of images
  - Vagrantfile is a specification that configures image.
- … and many more. See the Ultimate List of Provisioning/CM Tools here:
  - https://xebialabs.com/the-ultimate-devops-tool-chest/provisioning-config-management/

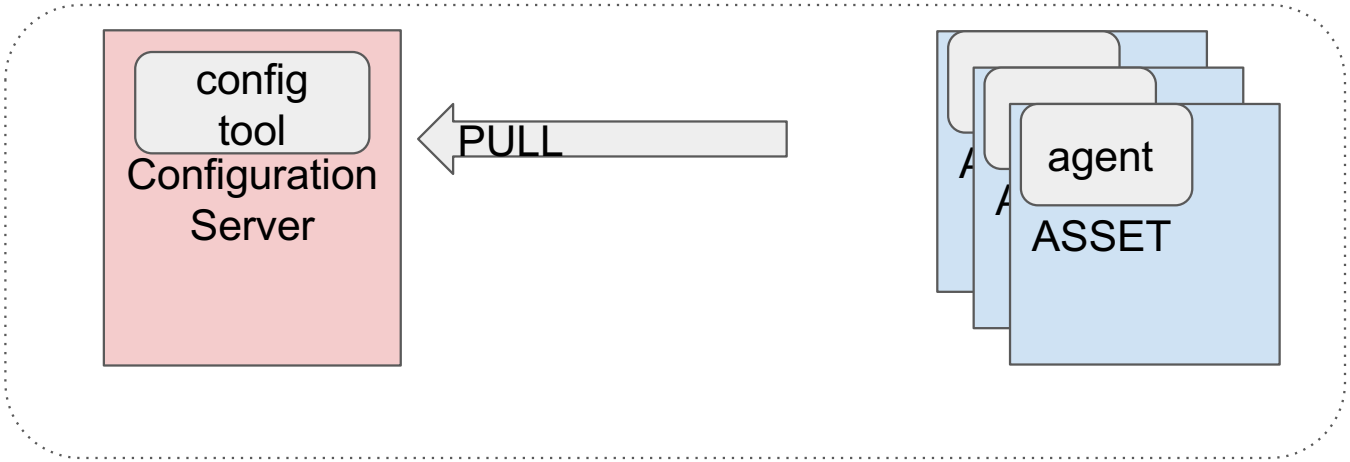
Infrastructure as Code

- SCM Tools are mostly open source options that provide ways to keep and maintain infrastructure in code form, or IaC (Infrastructure as Code).
- DEFINITION: Infrastructure as Code is the process of managing and provisioning computing infrastructure (processes, bare-metal servers, virtual servers, and so on) and their configuration through machine-processable definition files, rather than physical hardware configuration or the use of interactive configuration tools.
- Infrastructure as Code (IaC) plays an important role in Configuration Management.

Configuration models



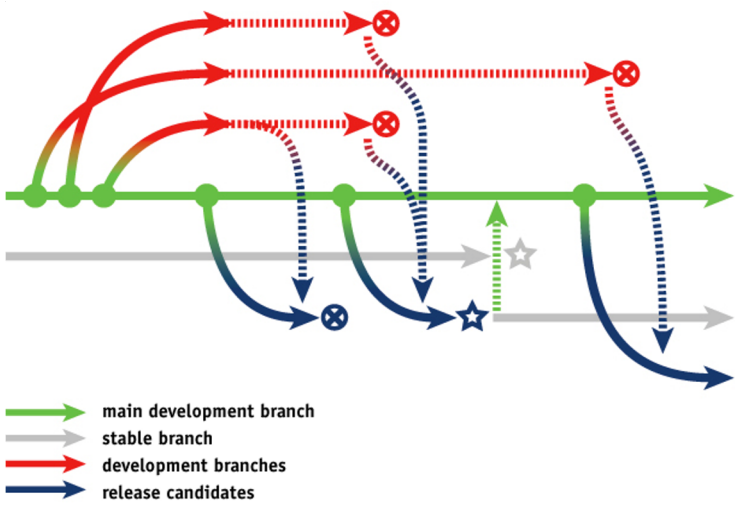
- Easier to manage.
- Less enforcement of state (asset can drift from config).



- Better at ensuring assets stay in sync with config. I.e, agent can enforce state.
- More complex and adds overhead

Version Control Systems

- Any that has the ability to create branches
  - Today's most popular: git
- A branch is a mechanism for allowing concurrent changes to be made to a source repository.
  - Empirical evidence shows that branches can be effective, but **devs** can create ineffective branches
    - See: Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis.

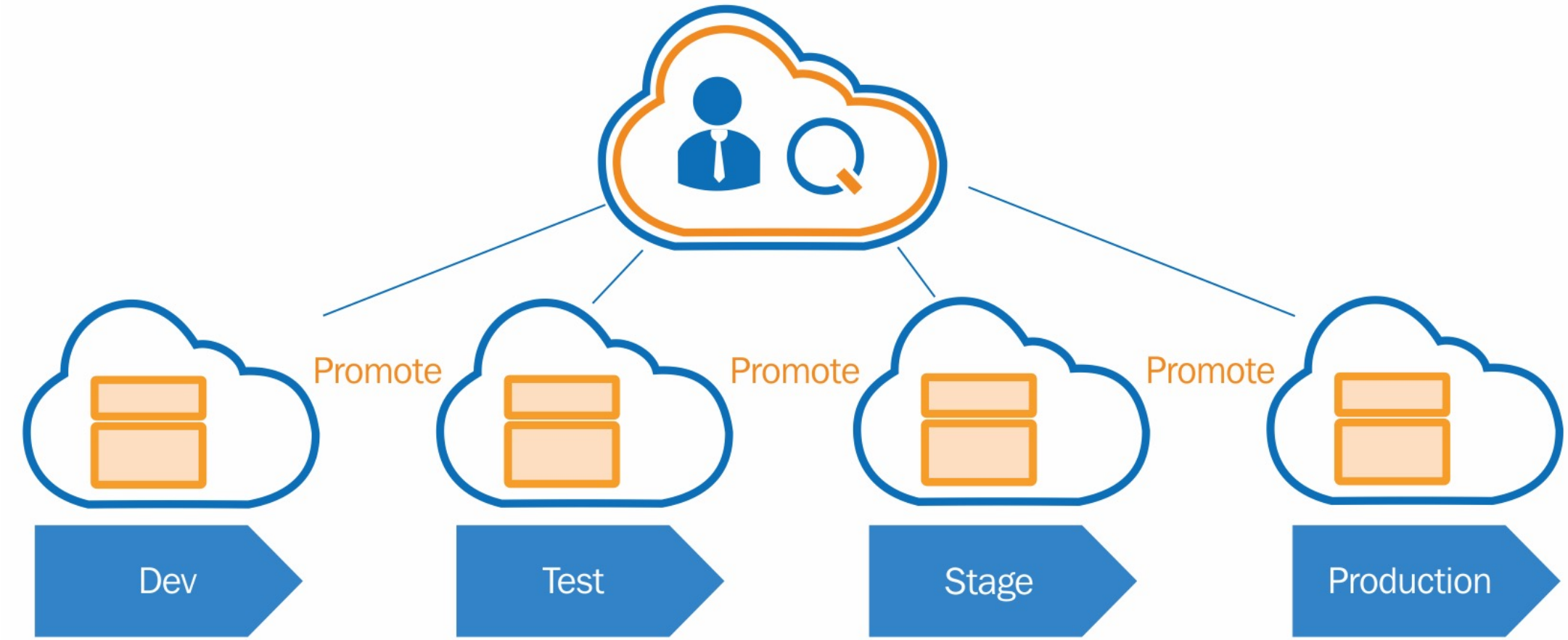


Notes:

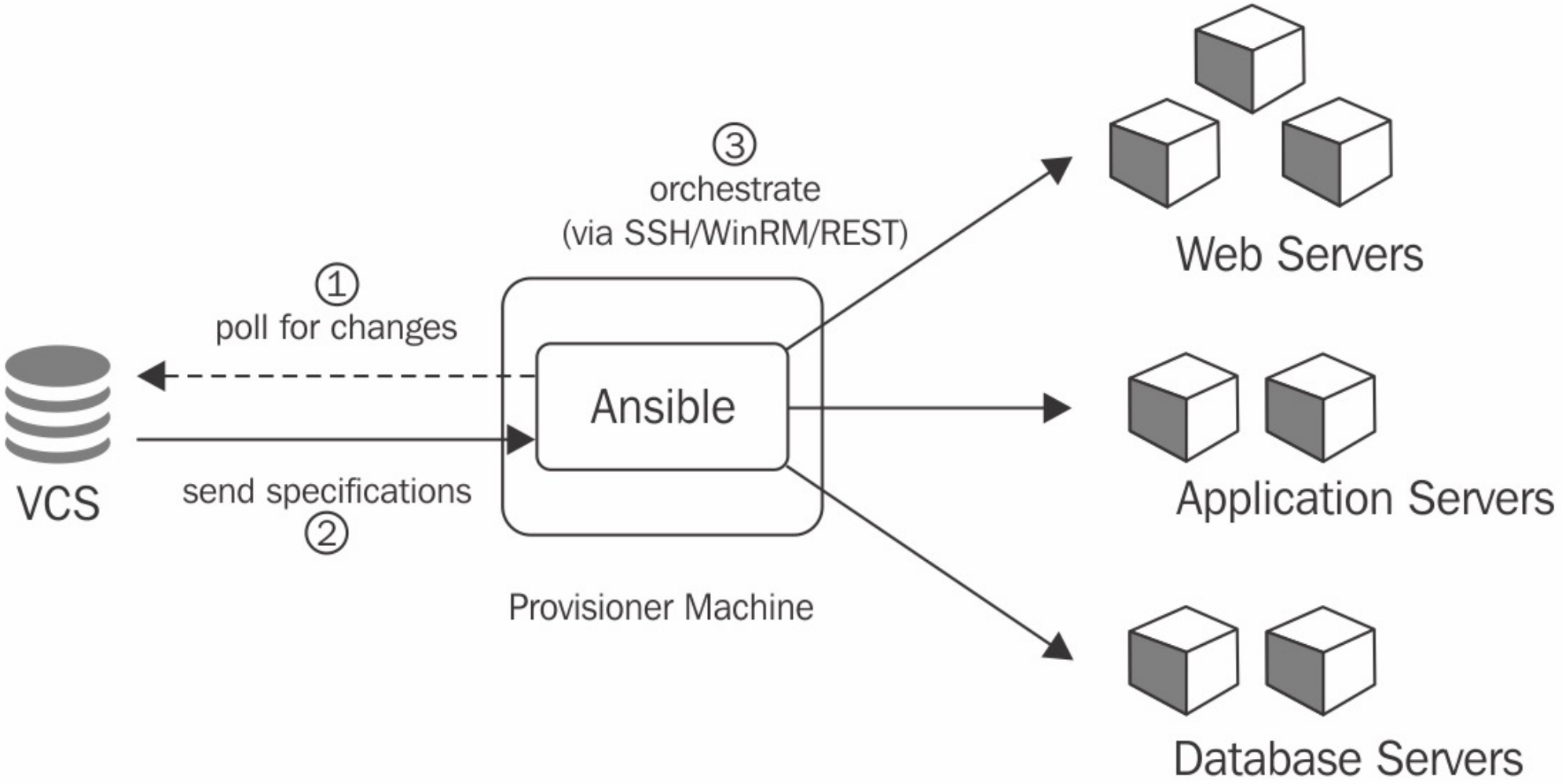
- the branching policy is defined by each dev team.
- the VCS tool selection (and installation, configuration, and maintenance) is made by the DevOps team (i.e. DevOps engineers).

A typical DevOps situation

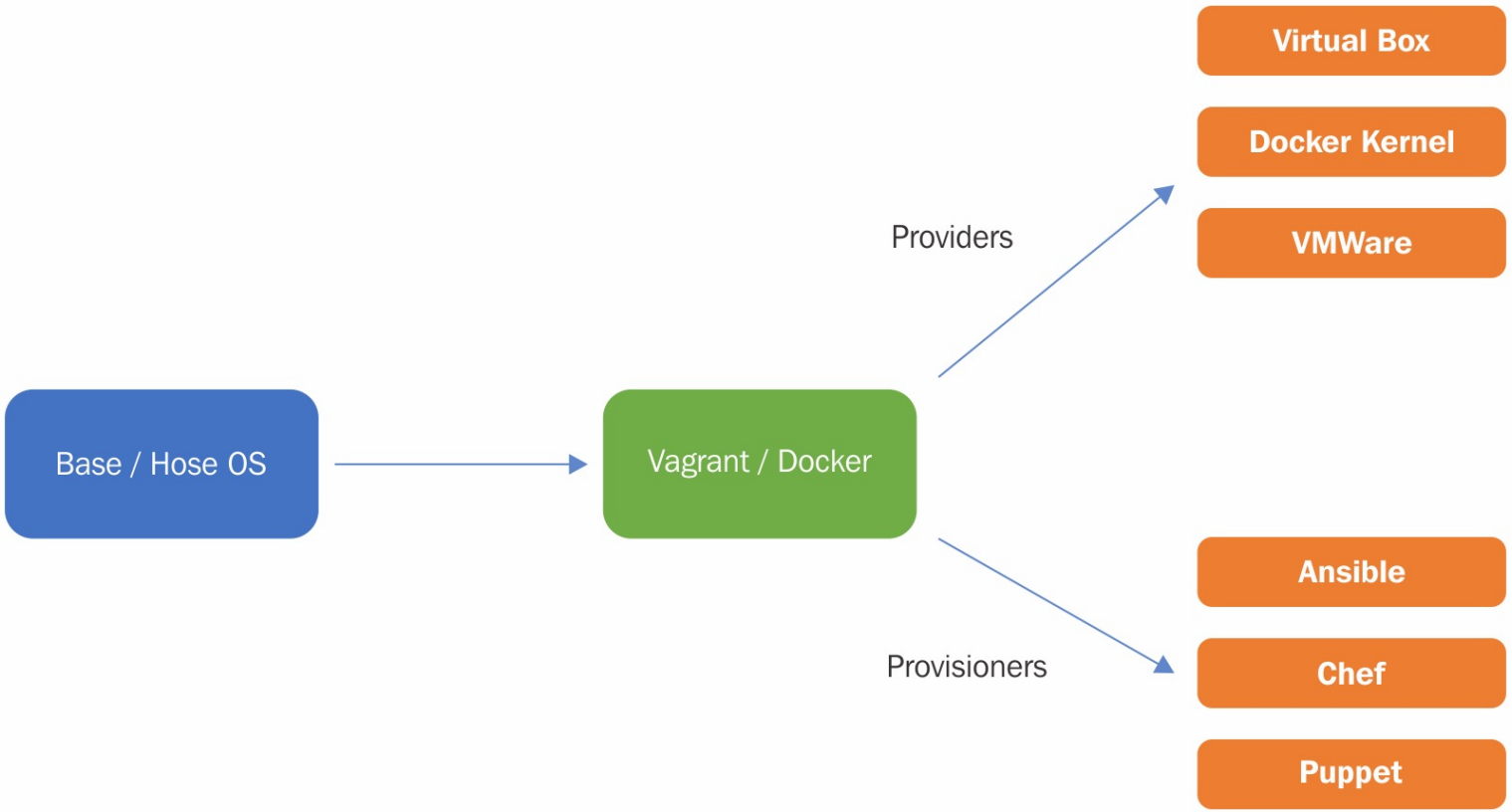
within the devops community, terms are frequently interchanged. Some of the terms that are used when referring to the movement between environments are **deploy**, **propagate**, **move**, **promote**, and **release**.



Ansible's Agentless Architecture



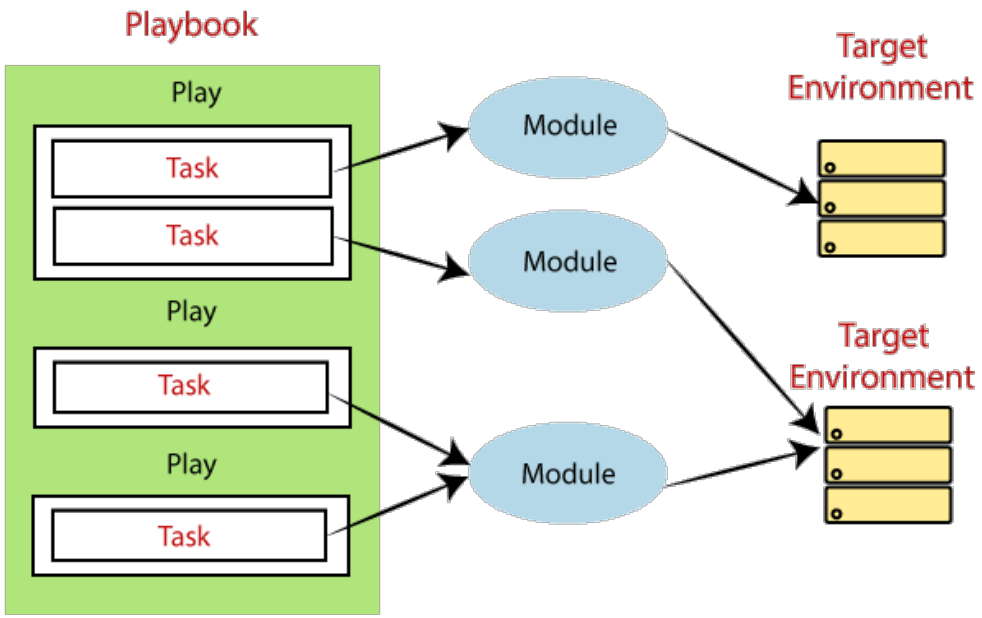
Container oriented automation



Ansible playbook

Playbooks are the files where the Ansible code is written. Playbooks are written in YAML format. **YAML** means "Yet Another Markup Language," so there is not much syntax needed.
Playbooks are one of the core features of Ansible and tell Ansible what to execute.
Playbooks contain the steps which the user wants to execute on a particular machine. playbooks are run sequentially.

Ansible playbooks tend to be more configuration language than a programming language.
Through a playbook, you can designate specific roles to some of the hosts and other roles to other hosts. By doing this, you can orchestrate multiple servers in different scenarios, all in one playbook.



Branch Anti-Patterns

- Merge Paranoia: avoiding merging at all cost, usually because of a fear of the consequences.
- Merge Mania: spending too much time merging code instead of developing it.
- Big Bang Merge: deferring branch merging and attempting to merge all branches simultaneously.
- Never-Ending Merge: continuous merging activity because there is always more to merge.
- Branch Mania: creating too many branches.
- Cascading Branches: branching but never merging back to the main line.
- Volatile Branches: branching with unstable files merged into other branches.
- Development Freeze: stopping all development activities while branching and merging.
- Integration Wall: using branches to divide the development team members, instead of dividing work.
- Spaghetti Branching: integrating changes between unrelated branches.

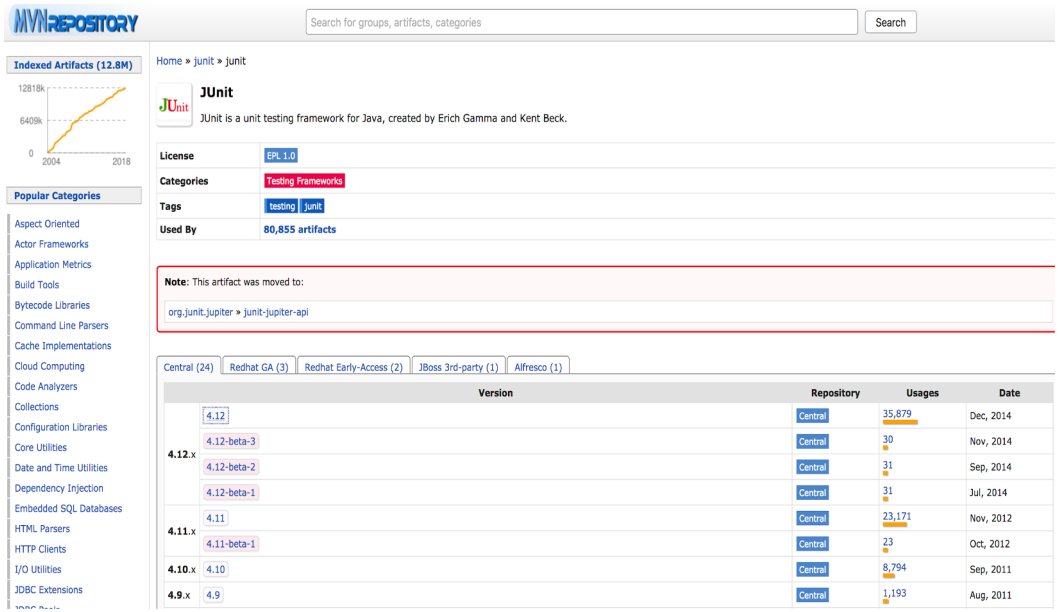
Effective use of Version Control Systems

- Keep everything in version control
  - Source code
  - Scripts (database, build, deploy)
  - Documentation
  - Configuration files
- Check in regularly to “Trunk”
  - Run your test suite before you check changes into version control
  - Introduce changes incrementally
- Use meaningful commit messages
  - Multiparagraph message
  - Include link to the identifier (e.g. Ticket) in your project management tool (e.g. Jira) for the feature/bug/improvement you worked on.

Package managers

- Avoid problems related to platform configuration.
- Binary
  - dpkg, rpm, apt-get, brew (mac)
- Source:
  - pip (python), npm (node.js), **maven (Java)**

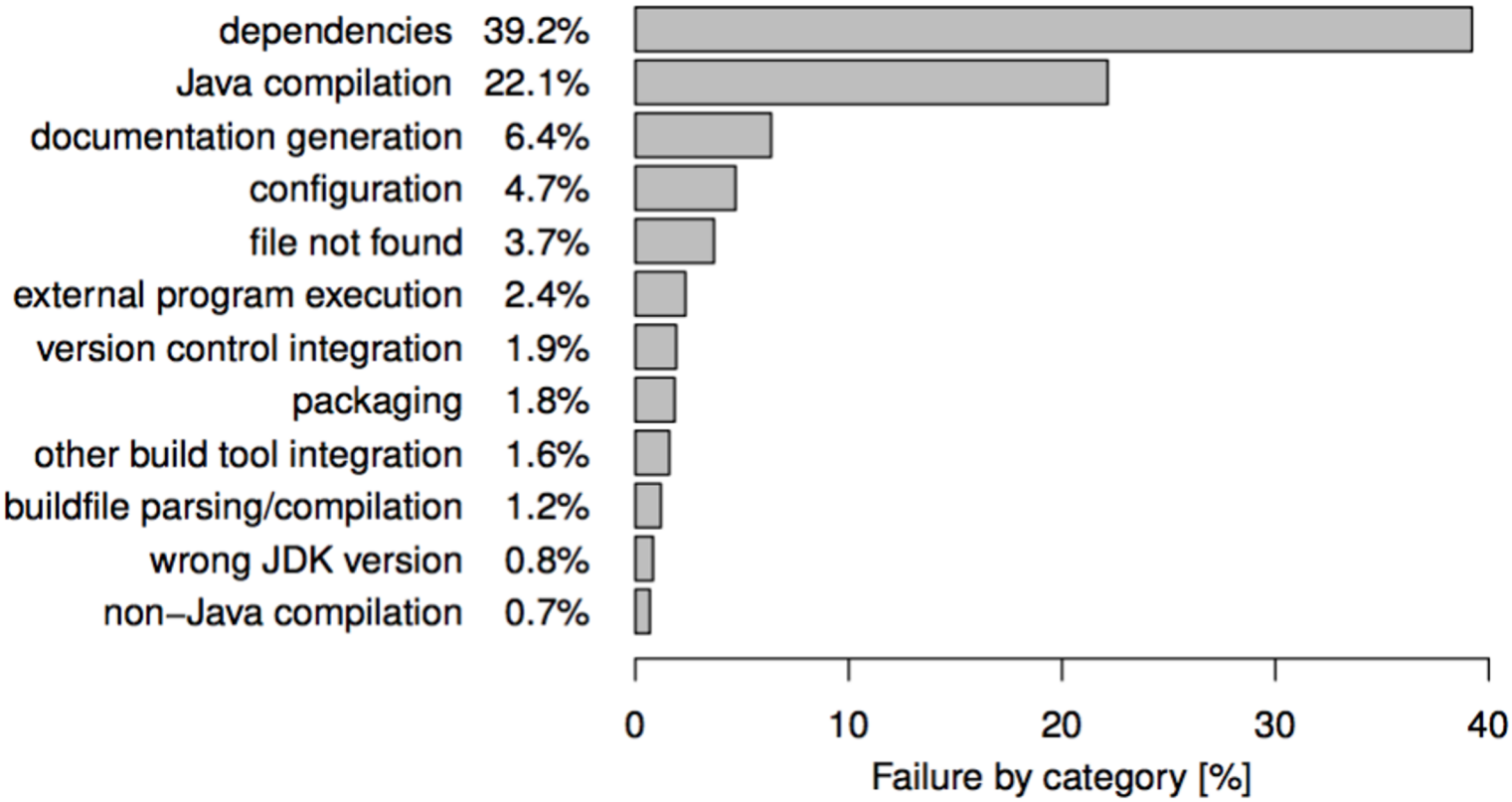
Note: some tools coordinate both the integration of “external libraries” and components => Dependency managers (e.g. maven)




```
<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Build managers

- Avoid problems related to build steps and dependencies.
  - Tools: make, ivy, ant, gradle, maven



A Quantitative Study of Java Software Buildability -> https://sulir.github.io/papers/Sulir16quantitative.pdf

Managing dependencies

- Libraries
  - Binary form
  - Licence compliance
- Components
  - Source code
  - Evolve along with other project's components
    - Build dependency

Note 2: keep a local copy of external libraries.
Question: Where?

Note 1: a component may become a library

Environment handling

- Key to success: make the creation of an Environment a fully automated process
- Relevant information to include into the process:
  - OS (including version, updates, and configuration files)
  - Dependency packages (including version and configuration files)
  - Networking information
  - External required services the product depends on (including version and configuration files)
  - Data → Data migration:
    - Inter and Intra environments
    - Solution: migration scripts
- Tools
  - Virtualization, Provisioning/Management Configuration Tools (see slide 11)

Note: keep configuration files in one place (VCS is the best place)

Product configuration

- Aim: achieve flexibility
- When to inject the configuration information?
  - Build
  - Packaging
  - Deployment
  - Startup/Runtime
- Track the configuration information
  - Storing vs. Product access
- Test the configuration settings
  - Service is up and running (while deploying the product)
  - Product has access to the service as expected (smoke tests)

References

1. Humble; Farley: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010).
2. Pressman: Software Engineering: A Practitioner's Approach (7th ed.). McGraw-Hill (2009).
3. Aiello; Sachs: Configuration Management Best Practices: Practical Methods that Work in the Real World. Addison-Wesley Professional (2010).
4. Bass, Weber, Zhu: DevOps : a software architect’s perspective. Addison-Wesley Professional (2015).
5. Sommerville: Software Engineering (9th ed.). Addison-Wesley (2010).
6. McAllister, Implementing DevOps with Ansible 2, Pakt (2017)

Questions?

Exercise 1

- Objective: Managing Product configuration
- Activity: for the product you have selected to use in your project
  - Find examples (at least 2) of injecting configuration at different times (i.e. build, packaging, deployment, startup/runtime)
  - Create a presentation (2 slides max) where you show for each of the found examples:
    - its pros and cons
    - implementation feasibility **under the given project constraints**
      - (already done, easy, hard, imposible)

Tip: this exercise helps for Checkpoint 1.

Exercise 2

- Objective: Managing Environment configuration
- Activity: provisioning with Ansible
  a. Create a new VM using vagrant (named “devops-vm”)
  b. Connect to the VM using ssh
     - Create priv/pub keys
       - $ ssh-keygen
     - Add public key into VM
  c. Provision with ansible
     - $ ansible-playbook -i inventory main.yml

Hint: https://github.com/acapozucca/devops/tree/master/provision