W4156

The Software Factory Floor

Agenda

We are going to talk about the low level mechanics of software via an analogy of a ‘classical factory floor’ / ‘production line’.

Through this analogy we will cover some core topics

- ❏ Environments
- ❏ Version Control
- ❏ Build
- ❏ Dependency Management
- ❏ Artifact Management

Factory

Let's say we are building car:

1. Designer/automotive engineer are constantly evolving design (materials, parts, wiring, etc)
2. On the factory floor new cars are being built all the time
3. The designer/engineer print design changes and send to factory floor
4. Each car is manually assembled to whatever design is attached to the notice board each morning
5. There is no real script/checklist to assemble cars. Everyone does it in different order/steps
6. Each car uses components out of a 'parts bin' but we don't keep track of which parts go into each car
7. A car takes 10 days to build then goes out on the test track (design changes during this time)
8. After passing the test track car is given to a customer

- If there is a problem on the test track which design change caused the issue?
  - Which other cars are impacted by this design flaw?
  - Which cars given to customers are affected?

- If a part was found to be faulty
  - Which cars have that part?

- A design flaw was added then later fixed. Which cars have that flaw?

Factory / Production Process

Clearly a shambolic and uncontrolled process which would have negative outcomes

We wouldn’t build something ‘physical’ this way

Software is not ‘physical’ but many similar fundamental challenges

(I could even argue the non-physical issue of software makes these issues even more impactful)

Let’s explore the software engineering equivalents within this context/analogy

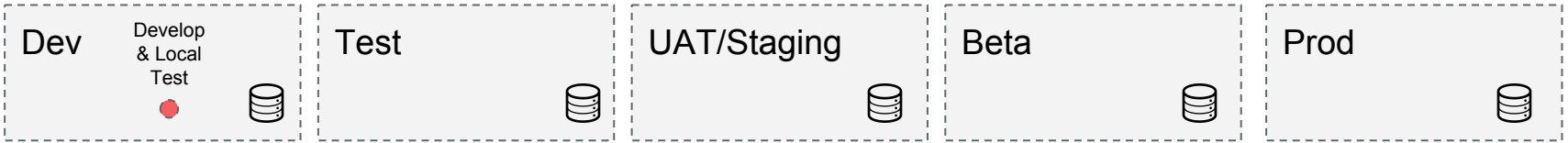
Environments

Environment:

- A set of hardware and software resources upon which a version of the application is deployed.
- Each environment has a specific purpose within the development lifecycle
- In many of the environments the transactions conducted have no external/monetary impact (i.e. its paper money)
- Environments are ideally segregated from each other physically or logically.

Most teams are likely running multiple versions of software in different environments at any given time
(in our car analogy it can be in construction, testing on the track or in ‘production’ with the user)

Environments



Database specific to environment

Environments

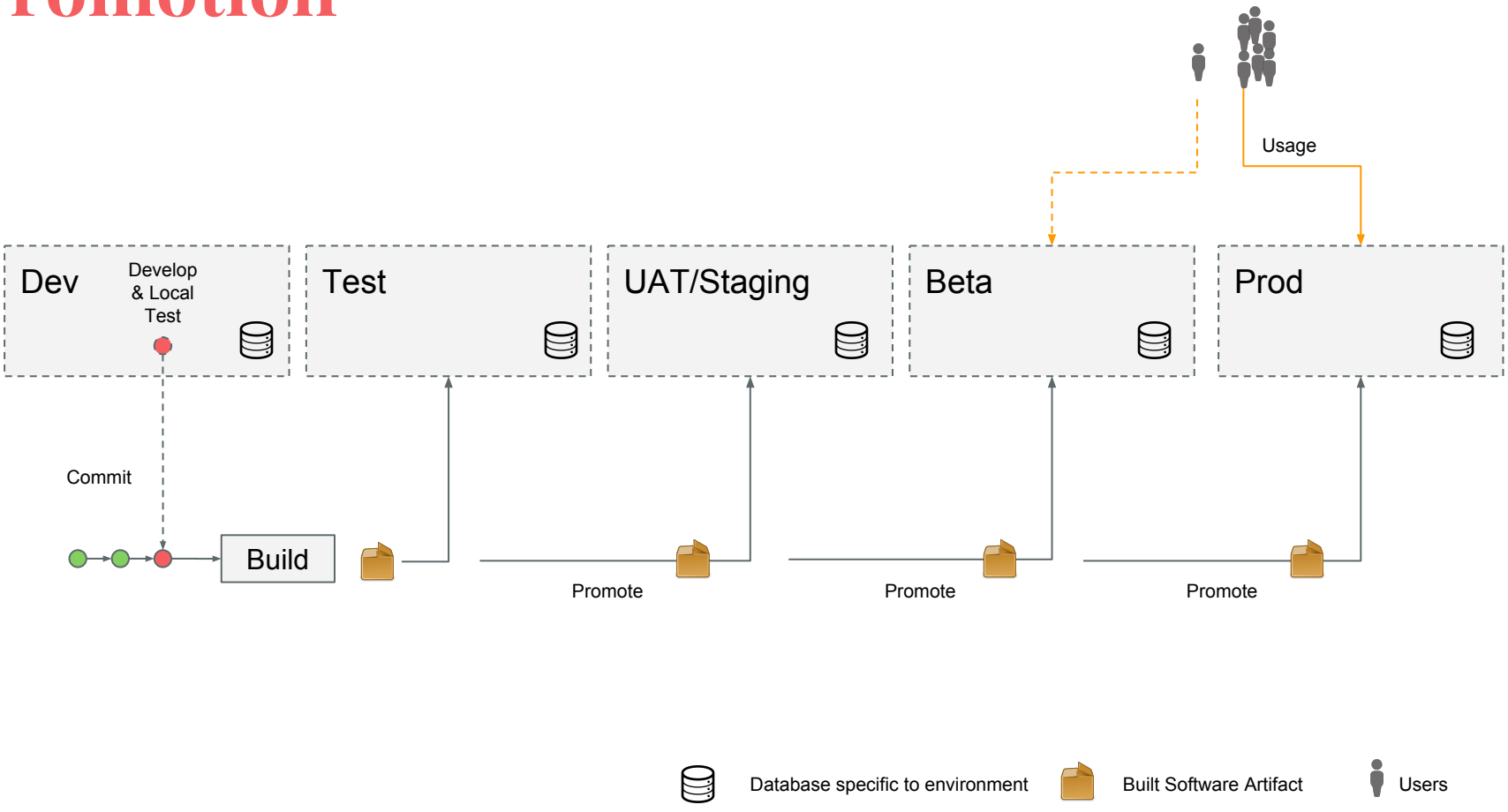
Environment	Purpose	‘Transactions Real’
Dev	Where each developer performs new development	No
Test	Contains candidate new version. All code / components integrated	No
UAT/Staging	Show to users for them to sign off ‘ accept’ (user acceptance testing)	No
Beta	Small flow rollout	Yes, % of all users
Production	All users	Yes, All users

(Many teams have different set of environments. There may be dedicated performance testing environment or multiple test or staging environments)

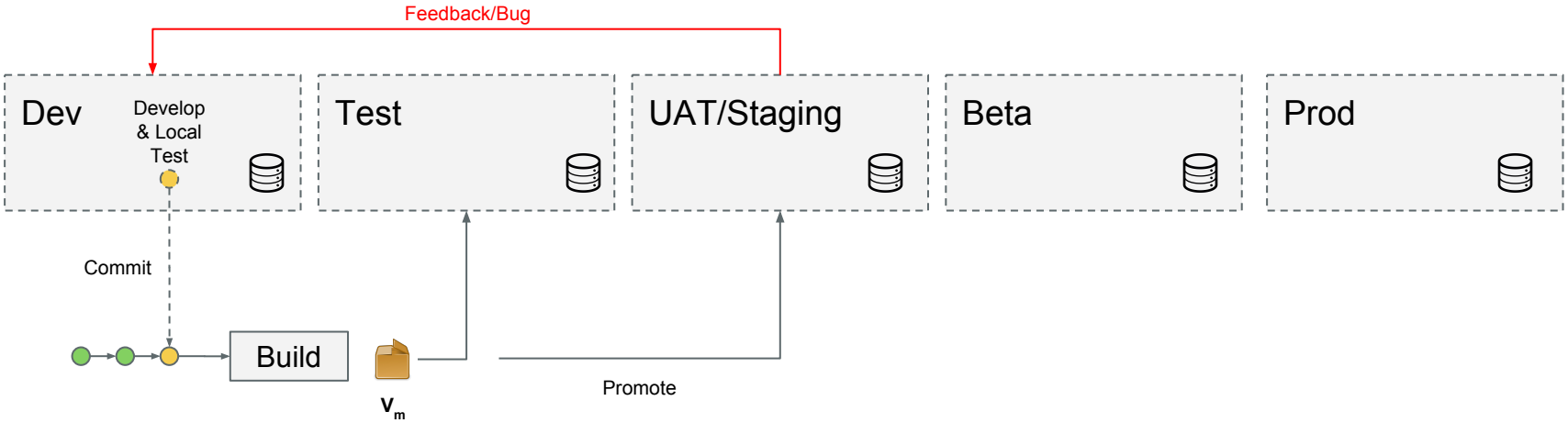
Promotion

Promotion: The act of taking a particular version of software - code, configuration and deploying in a higher environment

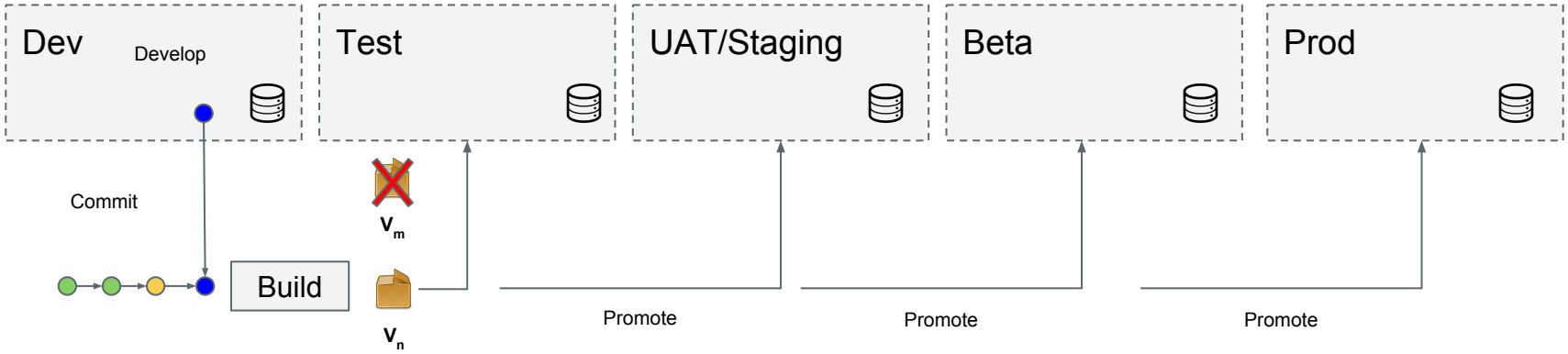
Promotion


Usage
Dev
Develop & Local Test
Test
UAT/Staging
Beta
Prod
Commit
Build
Promote
Promote
Promote
Database specific to environment
Built Software Artifact
Users

Feedback 1


Feedback/Bug
Dev
Develop
& Local
Test
Test
UAT/Staging
Beta
Prod
Commit
Build
Promote
V_m

Feedback 2


Dev
Develop
Test
UAT/Staging
Beta
Prod
Commit
Build
v_m
v_n
Promote
Promote
Promote

Symmetry

Symmetry: Although environments serve different purposes and may even be different deployment architecture (number of servers, region, etc) we want the environments to be as *symmetrical* as possible. That means that the hardware, operating system, drivers, JDK/SDK, compiler, libraries are identical. If they are not then we may not be able to either:

- Be confident that if something works in a lower environment we can ***promote*** safely (left to right)
- Bugs in higher environment we can easily ***replicated*** in a lower environment (right to left)
- If you are not in control of your environments it becomes a complete and frustrating nightmare

Software Factory

With a better understanding of the complexity of the factory floor / production process we can better relate to some of the discipline, processes and tools we need to be productive and controlled

- ✓ Environments
- ❑ Version Control
- ❑ Build
- ❑ Dependency Management
- ❑ Artifact Management
- ❑ Configuration Management

Version Control: Git

Git is (notoriously?) confusing to learn

I think this is because the *commands* are derived from the model

Once we have the mental framework the commands make sense.

Let’s spend lecture time on model and augment with labs on commands

From First Principles

1. Starting with a single developer saving code in folders on a machine
2. They need to go back 2 days (we took a wrong turn, diff old vs new)
3. We release a ‘cut’ and need to keep track
4. A second developer joins the project (‘concurrent copies’, ‘merges’)
5. Developer(s) working on new feature(s) not ready for release
6. Want to take an existing project and extend/customize
7. Want to contribute back changes to the original project

Git Concepts

1. Software artifacts are a graph with branches
2. Git is distributed
3. On a dev machine you have local, Index and HEAD
4. There is no one *way* to use git (there are different workflows)

If you understand the above then the commands (should) make (more) sense

Lets work through a simple example

Version Control

VCS: A special ‘database’ that tracks every version of the files (be they code, sql, documentation or configuration*). Generally provides constructs to support support of concurrent versions of software in addition to developers working concurrently.

* subject to best practices

Build

When we had a single file we compiled with javac or just ran it for interpreted languages.

However, for non-trivial systems we will have

- Compile steps
- Generate files from configuration (GPB, etc)
- Package files into the artifact we want to run

Manually executing is boring, slow and error prone. The steps also *evolve* with the software.

These steps are

1. Turned into executable script (build file)
2. Often leveraging a build tool (provides value added features)
3. ***Committed*** to the repository (so any version is buildable)

Dependency Management

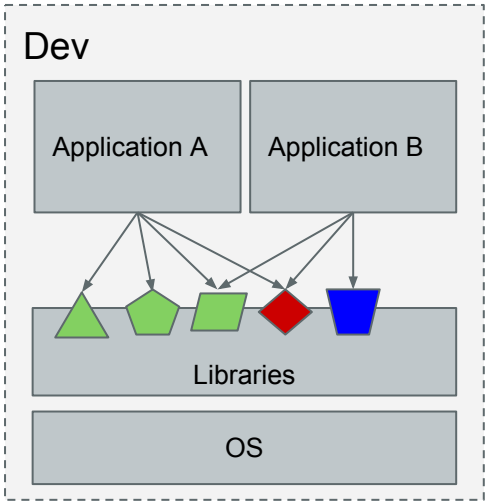
Shared by both

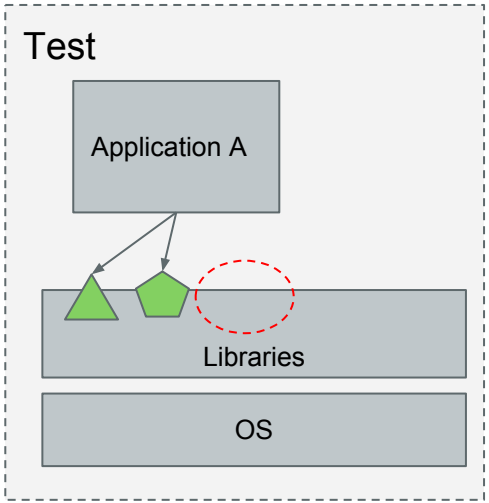
A but not B

Version Conflict

Non-trivial applications have a surprising number of dependencies

- SDK/Interpreter/Compiler: Python 2.7 or 3.6.4 or 3.6.3?
- Libraries: flask 0.13 or 0.12?, connexion, requests, etc (can often be 10s .. 100s)





Dependency Management

All major languages come with some form of application dependency manager

- ‘Requirements File’: specifying required packages (requirements.txt, package.json, pom.xml)
- Dependency Manager: to resolve, download and managed dependency conflicts (pip, npm)
- Isolation: a way to isolate projects from machine installations (virtualenv / npm local)

The *process* is then

1. Developers add any dependencies to the requirements file
2. This file is ***committed*** into the repository
3. The resolution of dependencies is part of the build process
4. Often incorporated into an pre-baked ***artifact***

Build Artifact Management

When you build your code you will generate an **output/build artifact.**

- This may be a jar, war, zip file, docker image
- Artifacts often *stored* in some artifact repository (generally not SCM)
- It must be traceable to a version of source code and repeatable
- With a stored build artifact we can push it between environments without rebuilding
  (though we need to manage config between environments)

Configuration Management

Your application will also have configuration.

- Is a feature enabled or disabled?
- What is the setting of a configurable option?
- What is the memory size of the JVM?
- What is the IP address of a service we need?
- What is the IP of the database?*
- What is the password for the database?*

Configuration Management

Does configuration differ between environments?

12 Factor Configuration

****DO NOT COMMIT PASSWORDS TO THE REPOSITORY. NEFARIOUS PEOPLE SCAN PUBLIC GITHUB REPOS LOOKING FOR AWS KEYS TO HACK IN AND MINE BITCOIN ETC***

*** DO NOT COMMIT DATABASE URLS OR PASSWORDS TO REPOSITORIES. YOU CAN ACCIDENTALLY CONNECT AND DELETE PRODUCTION. IF CONFIG IS SEGREGATED YOU CAN'T HAVE THAT ACCIDENT. STORE THAT CONFIG IN EACH ENVIRONMENT ***

Wrap Up

- People often try to the ***wider*** software development process physical metaphors (house building)
  - These metaphors are almost universally bad (software != house building)
  - However, in terms of the ***limited*** scope of source control -> production the ‘factory’ or ‘assembly line’ analogy apt. We are engineers and must create a precise, controlled, repeatable and traceable process.

- Yes - a lot of time goes into writing and configuring the ‘development infrastructure’. We will close the loop on this in the next lecture on continuous integration. A few years ago it was even worse as each team configured their own build pipeline. These days there are many plug and integrate managed services. However, it still takes time.

- There is no singular process. Teams often start simple and *evolve* the process as they scale in complexity.

Project Recommendation

1. Create a common github project for the project
2. Commit code, sql DDL, build script into git
3. Pick a git workflow (either mainline/trunk based development or github wf)
4. Follow basic practices
    a. Don’t commit broken code
    b. Write meaningful commit comments

Pop Quiz

Question	Answer
Git is a tool or a workflow?	
Dependency management is required to prevent?	
What needs to be committed to source control?	
What should not be committed to source control?	

Reading

Material	Optionality
Learn Git (Github)	Required (1-25)
Git guide	Required
Trunk vs Flow	Required
Git Cheat Sheet	Optional
Beginning Software Engineering Chap 9	Required

Appendix A: Immature Development

Let's say a hypothetical development setup is:

1. Develop code on your machine
2. Saving 'good' versions using 'cp -r myproject myproject_goodbackup1'
3. Using the version of <python/Java> on your machine but not quite sure which one
4. You installed a bunch of libraries during development
    a. You are developing two projects on your machine (some libraries not for this project)
    b. Some of the dependencies you removed during development
    c. Not quite sure which ones are still required
5. You test on your developer machine
6. SQL Schema files are on google drive. Manually drop and create tables when required
7. When you like the version on your machine you compile and ssh the binary to production
8. If another developer joins you email files back and forward
9. All the production and development config on your machine with plaintext passwords

* and quite possibly by the end of the course you will have produced enough code/configuration that the issue becomes real

(I feel anxiety typing/reading that slide)

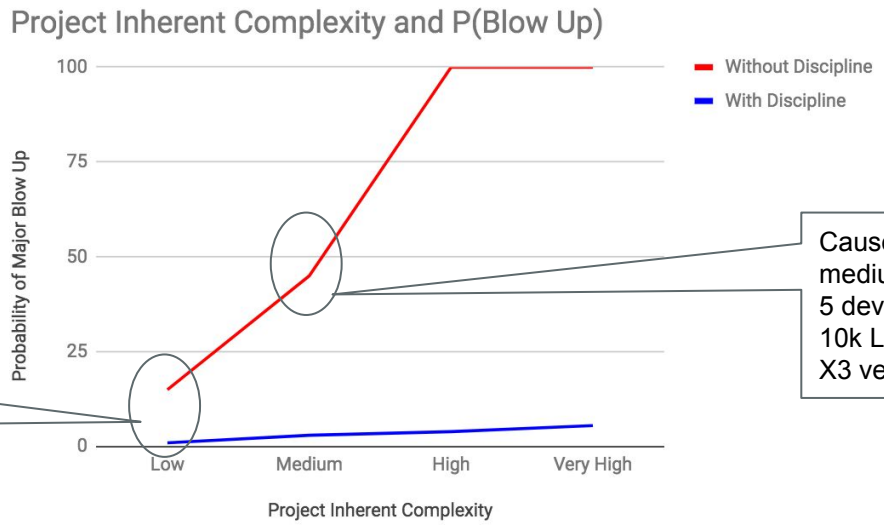
Sensible things you want to do	But can’t	Because
Have multiple developers working on a code base productively	Because emailing back and forth is time consuming, confusing and error prone	Lack of source code management (no ‘truth’)
Make a minor change to the version running in production	Erm …. What version of code is in production?* (The version you are working on is newer but has a bunch of changes and is not ready)	Lack of artifact management and traceability to source code
Push to production quickly and reliably	You push out the code and find that machine is missing libraries you need OR the code was dependent on a schema change	Lack of dependency management OR lack of source control of all assets
Repeatability of bugs from production	Production is running a different JDK	Inconsistent environments

* occasionally you will meet a team that doesn’t know which version is running in production and it causes an almighty PITA

Pause - Complexity

- When learning to program it was you on your own, working with a small set of files and one running version of the code.
- Major shock when transitioning to industry we are working with others, on many files, 10s-100s of libraries, compiler versions, target operating systemes, many versions of the software (being developed, tested and running)
- Ad-hoc processes (emailing files, copy pasting binaries, building on desktops, don’t scale)

Pause - Complexity



Practices that 'survive' here
1-2 developers
1-2k LOC
X1 version running

Cause projects to implode at medium complexity.
5 developers
10k LOC
X3 versions running