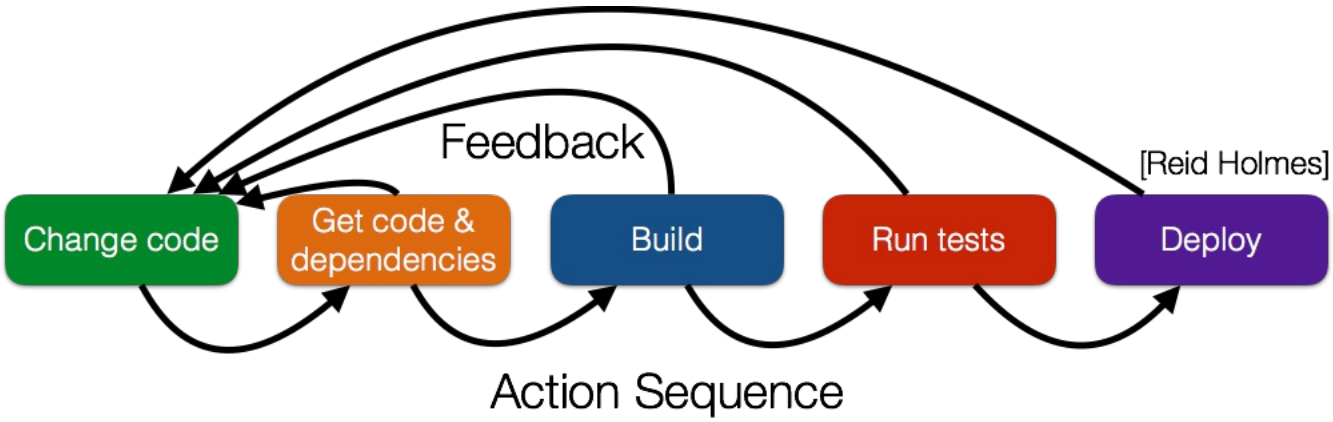
Automation

Our expectations for modern software systems demand delivering a level of velocity that was not possible in the past. Gone are the days where delivering once every 3-5 years was considered acceptable. Being able to deliver changes on a weekly, daily, or minute-by-minute basis allows software teams to more quickly deliver new products to market or provide more timely bug fixes.

The figure below shows a coarse-grained view of the broad steps required to get a code change deployed in front of a user. While human intervention is required to make every code change, it should be possible to automatically transition from the change in front of a user. The less time this sequence of actions takes (aka the less friction there is in the build process), the quicker the developer can get feedback to refine or roll back their change. Also, the quicker and easier this process is, the more likely developers are to take advantage of it (e.g., if running tests is hard, developers will try not to run them or if deploying is hard developers will avoid deploying fixes that could benefit users unless they are significant 'enough').



Software systems are increasingly large, distributed, and heterogeneous. Successfully building, testing, and deploying them requires coordinating a huge stack of tools, environments, and systems. This requires a high

degree of craftsmanship and has historically involved many manual, laborious, time consuming, and error prone steps to translate source code into a shippable executable. The goal of automation is to transform this unsustainable way of building software into a process that is:

- **Repeatable**: The process of building a product should not vary between different versions of the software (modulo minor improvements over time). A unified building process provides a mechanism for building the system the same way, every time with minimal (prefereably no) human intervention required.
- **Reliable**: If the build process was repeatable but was subject to many non-determinstic failures it would not have value. Being able to trust that the repeatable process will run to its successful completion is crucial, although naturally test failures are sometimes to be expected and are not considered a build automation reliability failure (unless the tests themselves are flaky).
- **Revertible**: The build process should make it possible to quickly and transparently revert out of any change. If a build is deployed and is found to have an error in production, it should be possible to automatically revert to a prior build in a quick and transparent way.

The gains from automation can be thought of analogously to the gains made when books could be printed with a printing press instead of being written by hand: the setup time is considerable, but the benefits far outweigh these costs in the long run.

Automateable units

There are many software building tasks that can be automated:

1. **Source control**: Every software team uses version control to track their source code resources (and other assets including any

automation tools themselves). `git` and `hg` are two commonly used version control systems that provide a reliable way to access any past development state.

2. **Dependency management**: Code is not developed in isolation; all systems are built upon a host of existing libraries and frameworks. Dependency management solutions provide a means to reliably procure software required for the build process. Common tools include `ant`, `mvn`, `npm`, and `yarn`.
3. **Build tools**: Compiling the code into shippable units (be they libraries or actual executables) is the next step. While these tasks are sometimes easy, they often involve many independent steps that are joined together by build tools like `make`, `ant`, or `gulp`.
4. **Test tools**: Tests are code too, so enabling them to be written with as little boilerplate code as possible is important for reducing the costs of automated testing. Many unit test tools in particular have been developed to help software engineers write tests and form them into coherent test suites quickly. These tools include `jUnit`, `NUnit`, or `mocha`.
5. **Test runners**: Once test suites have been developed they must be run in a repeatable way. While this can take place on the engineer's development machine, tests are often run on a common infrastructure to increase the consistency between test runs for all developers. Tools like `Jenkins`, `Bamboo`, and `TravisCI` all work to execute test suites remotely. These first five steps are often called *continuous integration*.
6. **Deployment**: In the online realm software must be deployed once it is built; this field (often referred to as *continuous deployment*) is beyond the scope of this reading.

Ultimately effective automation will allow a software system to be checked out, configured with the correct dependencies and tools, built,

and unit tested with a single command (or a short sequence of commands that can be placed in a simple script). For example, for the project we can simply run:

```
yarn install
yarn run configure
yarn run build
yarn run test
```

By enforcing each step to be executable on the command line without developer intervention, the entire process can happen automatically in the background. For example, after every push to a project's VCS the whole process can be automatically performed and the results sent to the engineer if any problems are detected.

References

- A humorous discussion of automation can be found in the Phoenix Project.
- Nice set of build automation slides.
