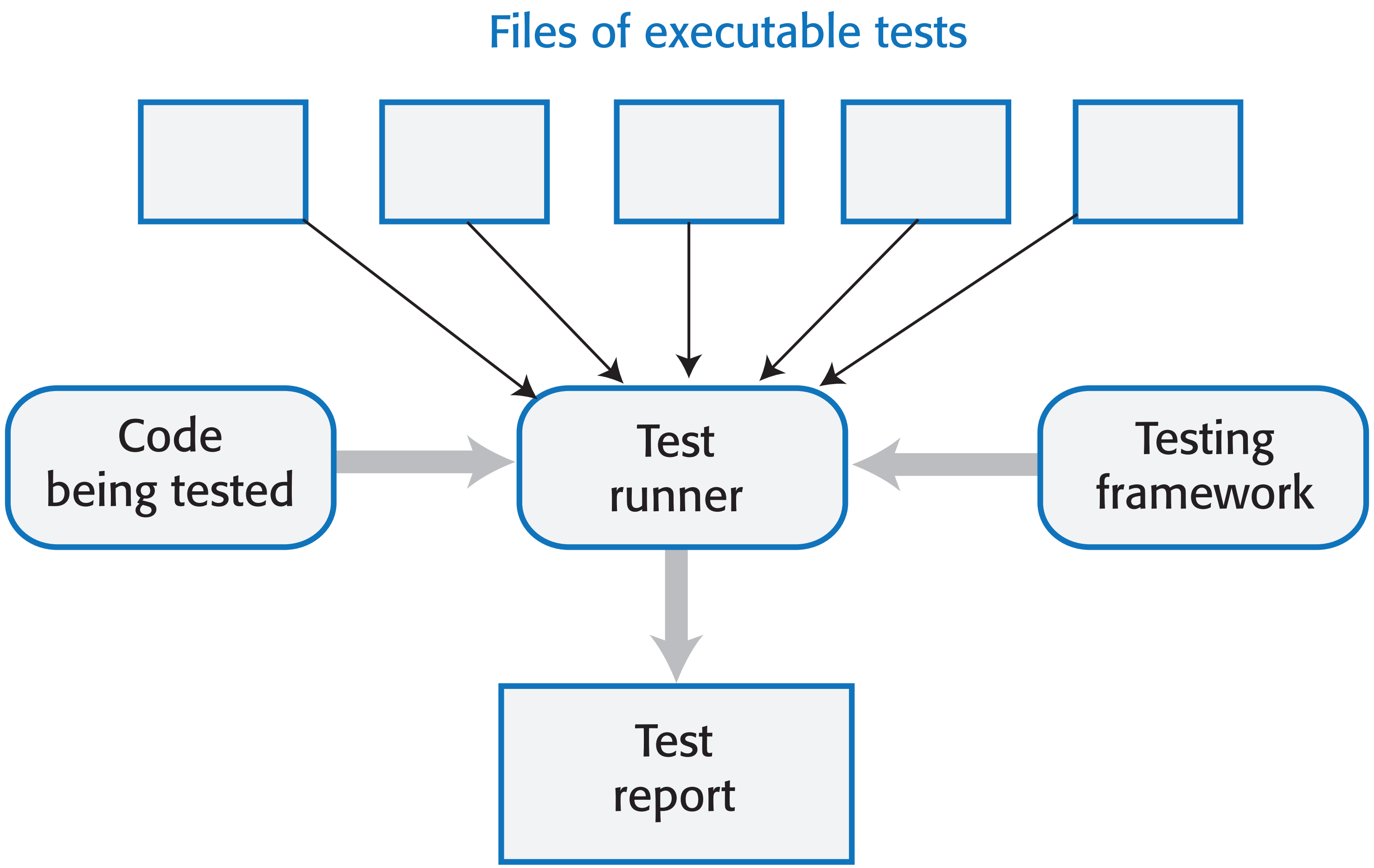
Test Automation and Code Review

Based on slides developed by Ian Sommerville and on GitLab CI

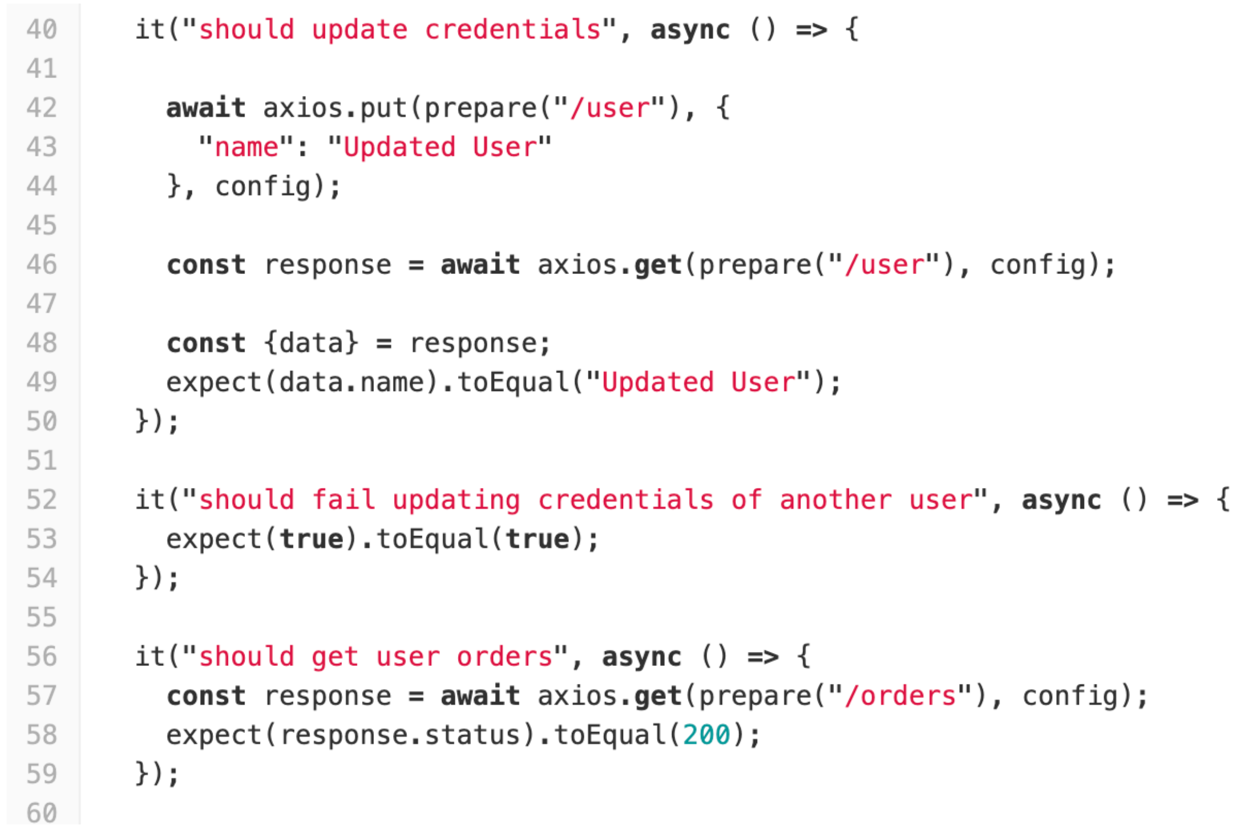
Test Automation

- Automated testing is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- You run the test and the test passes if the unit returns the expected result.
- Normally, you should develop hundreds or thousands of executable tests for a software product (or at least automate the production of such tests)

Automated Testing



Example



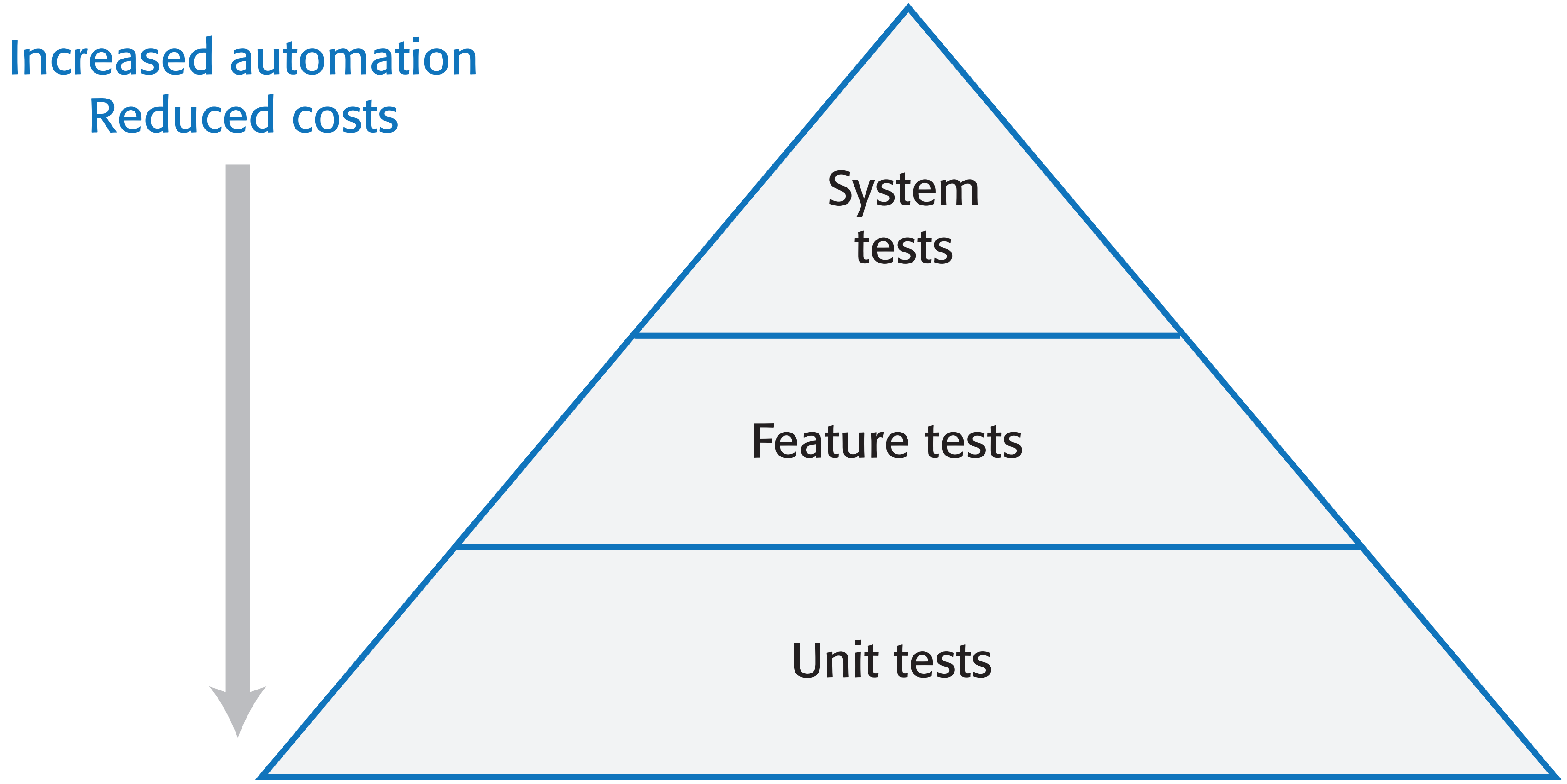
```
it("should update credentials", async () => {

  await axios.put(prepare("/user"), {
    "name": "Updated User"
  }, config);

  const response = await axios.get(prepare("/user"), config);

  const {data} = response;
  expect(data.name).toEqual("Updated User");
});

it("should fail updating credentials of another user", async () => {
  expect(true).toEqual(true);
});

it("should get user orders", async () => {
  const response = await axios.get(prepare("/orders"), config);
  expect(response.status).toEqual(200);
});
```

- The environment is already set up, e.g. the database with orders and users.
- The automated test sets up the parameters,
- Executes the function under test.
- Expects a particular return value.

Automated Tests

- It is good practice to structure automated functional tests into three parts:
  - **Arrange** You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed (some of this is in the environment spanning many tests).
  - **Action** You call the unit that is being tested with the test parameters.
  - **Assert** You make an assertion about what should hold if the unit being tested has executed successfully.
- If you use category partition to specify tests, ideally you should have several automated tests based on correct and incorrect inputs from each partition.
- A test runner will run all of the tests (based on the scripts in the package description in the case of the JavaScript project). For a tutorial on the basic details for Jest, see: https://www.valentinog.com/blog/jest/

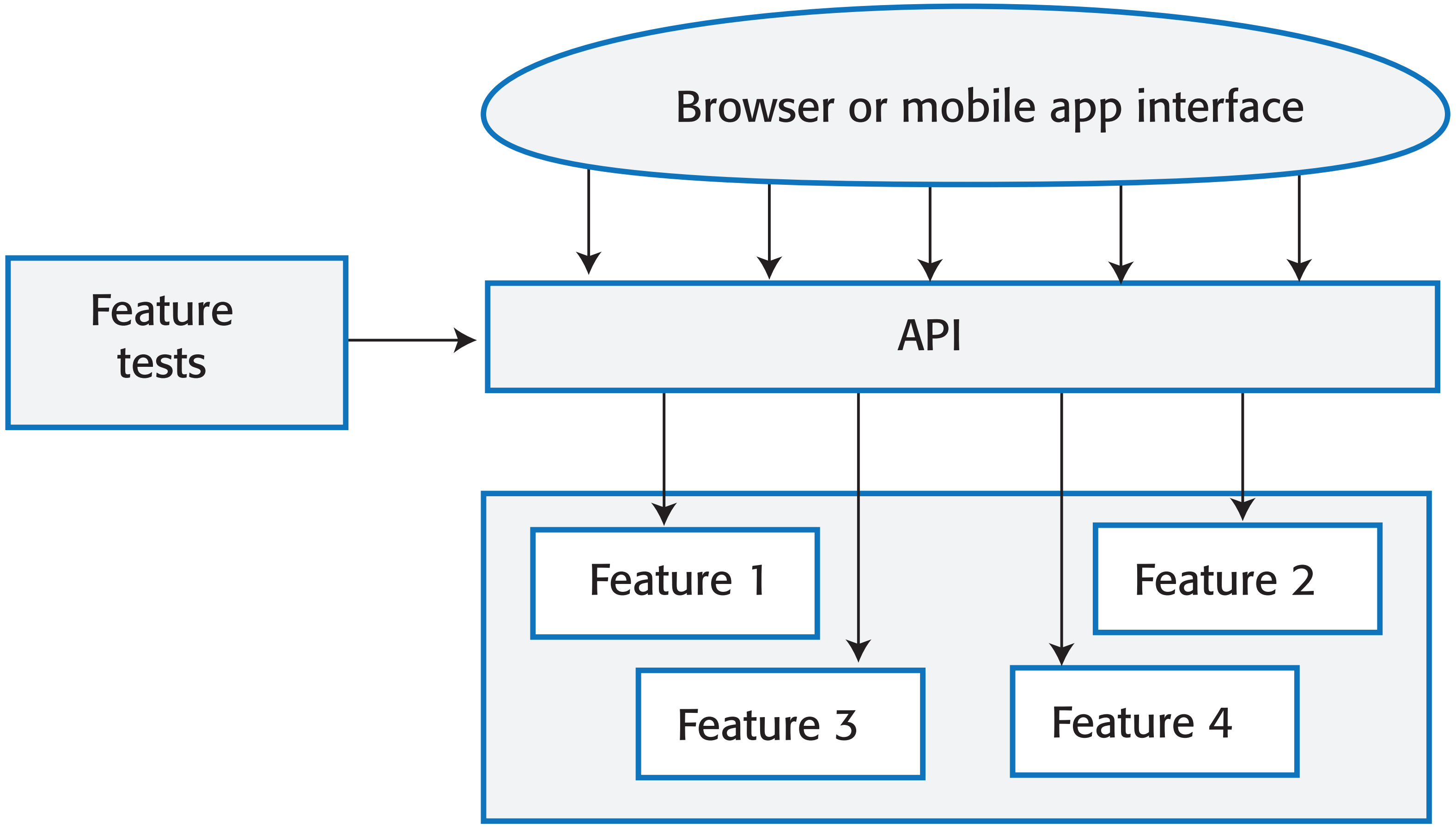
Test Pyramid



Automated system testing

- Often, users access features through the system’s graphical user interface (GUI).
- However, GUI-based testing is expensive to automate so it is best to design your system so that its features can be directly accessed through an API and not just from the user interface.
- Tests can then access features directly through the API without the need for direct user interaction through the system’s GUI.
- Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software (separating concerns).

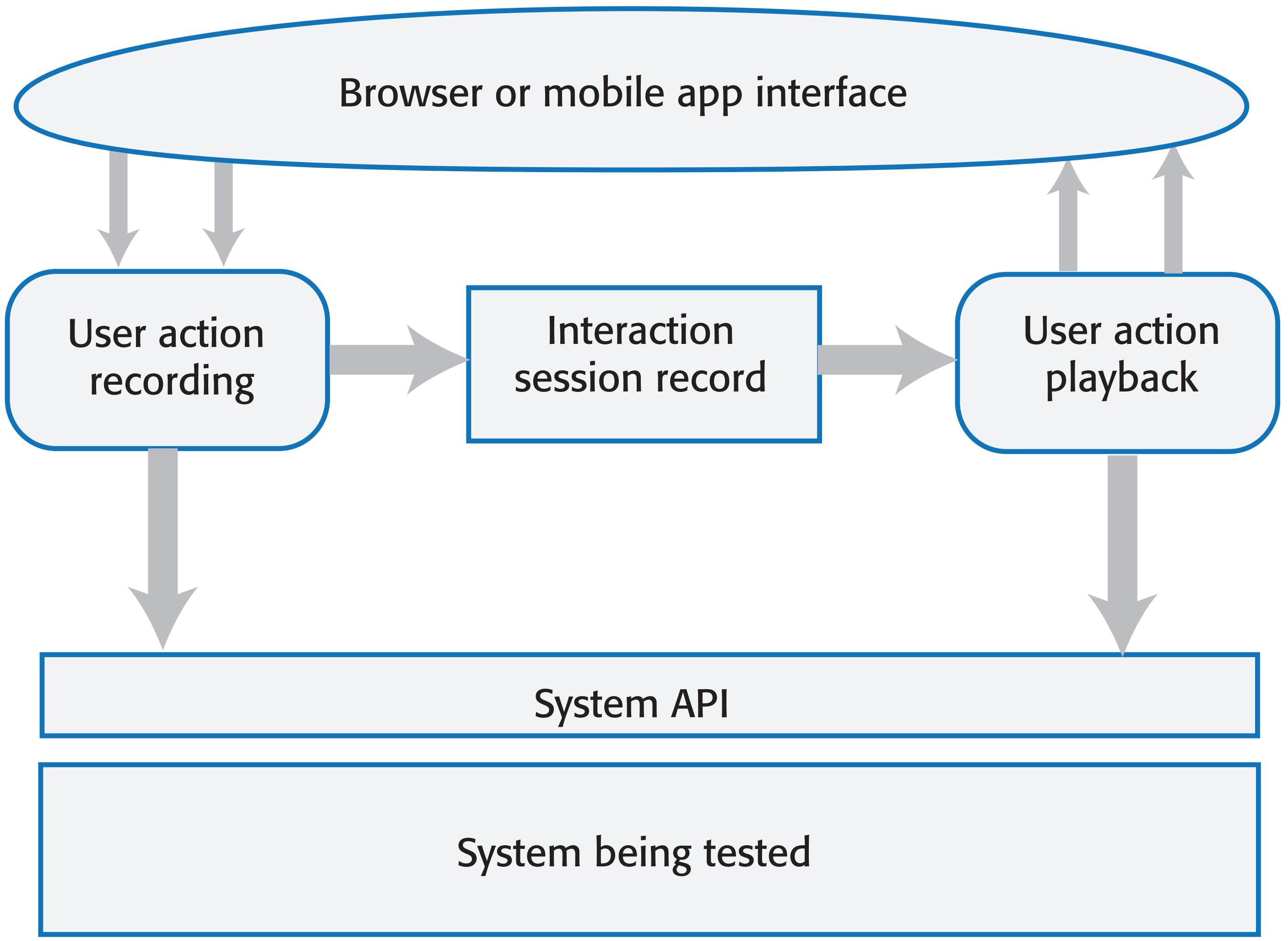
Using an API for Testing Features



System Testing

- System testing involves verifying system requirements that may involve simulating user behaviour.
- System tests, involve some sort of UI as the way users will access the features of the system
- You are looking for interactions between features that cause problems, sequences of actions that lead to system crashes and so on.
- Manual system testing, when testers have to repeat sequences of actions, is boring and error-prone. In some cases, the timing of actions is important and is practically impossible to repeat consistently.
    - Testing tools have been developed that record UI gestures and automatically replay these when a system is required – this can take the repetition out of system test

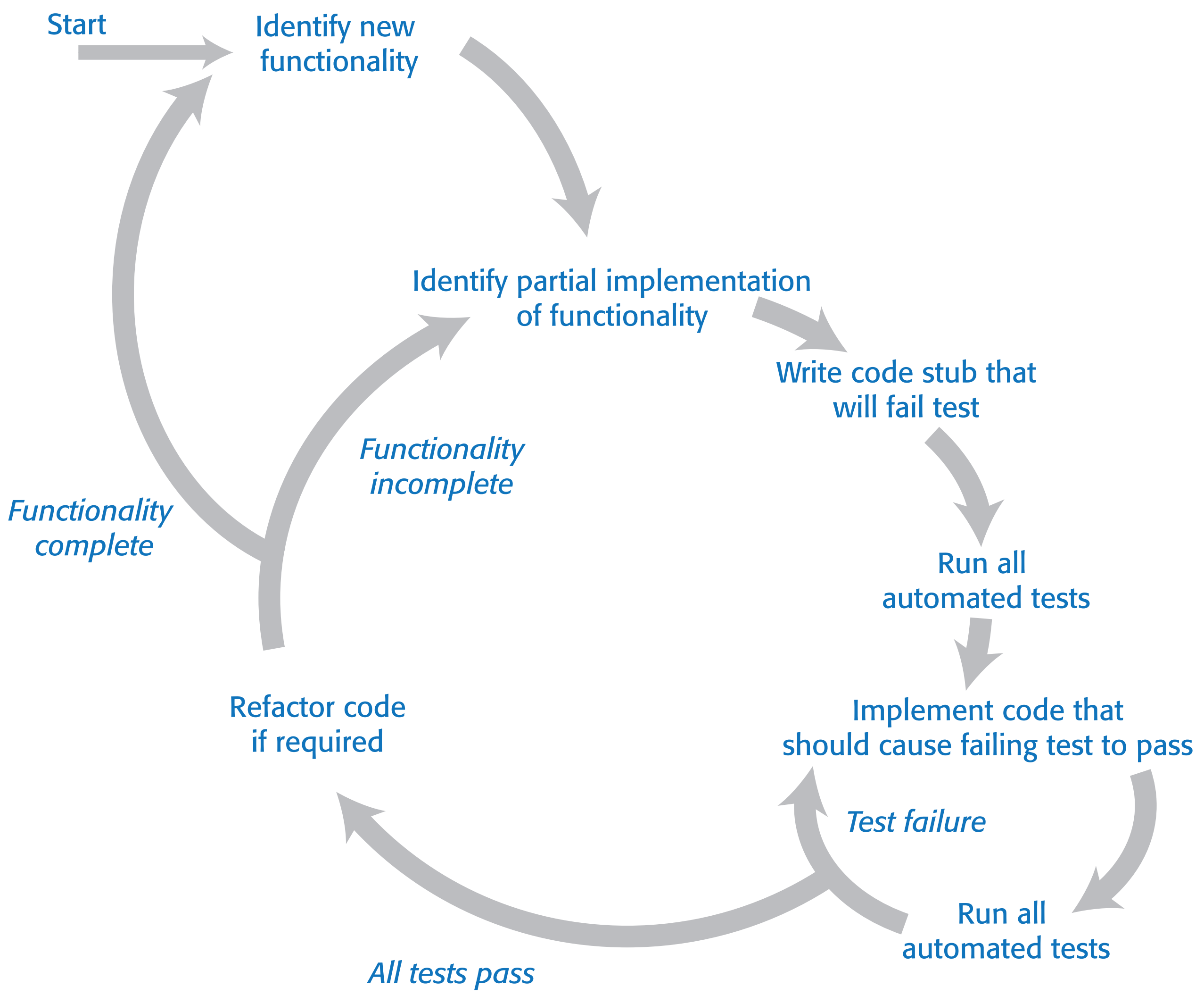
Gesture Capture for System Test



Test-driven development

- Test-driven development (TDD) advocates writing a executable tests for code before you write the code. The test is a sort of specification.
- Early users of the Extreme Programming advocated TDD, but it can be used with any incremental development approach.
- Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.
- Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

Test Driven Development


Start
Identify new functionality
Identify partial implementation of functionality
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing test to pass
Run all automated tests
Test failure
All tests pass
Refactor code if required
Functionality incomplete
Functionality complete

TDD Stages

- ***Identify partial implementation***
  Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
- ***Write mini-unit tests***
  Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
- **Write a code stub that will fail test**
  Write incomplete code that will be called to implement the mini-unit. You know this will fail.
- ***Run all existing automated tests***
  All previous tests should pass. The test for the incomplete code should fail.

- ***Identify partial implementation***
  Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
- ***Write mini-unit tests***
  Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
- **Write a code stub that will fail test**
  Write incomplete code that will be called to implement the mini-unit. You know this will fail.
- ***Run all existing automated tests***
  All previous tests should pass. The test for the incomplete code should fail.

TDD Benefits

- A systematic approach to testing, tests are clearly linked to sections in the program code.
  - Good confidence that tests cover all of the code and that there are no untested code sections.
- The tests act as a written specification for the program code. It should be possible to understand what the program does by reading the tests.
- Fault location may be simplified, when a program fails, you can immediately link this to the last increment of code that you added to the system (this failure could be uncovering a fault that needs fixing in the already developed code).
- TDD can encourage simpler code. Programmers write code to be testable because often they also construct the unit tests.

TDD Problems

- **TDD can discourages radical program change**
  Significant change means more failed tests. If the number of passed tests motivates the developer they will avoid generating lots of failures
- **Tests are often insufficiently expressive**
  If we see tests as the specification then this can restrict what we spcify because some things are hard to test
- **If everything is "code" it's hard to get a good overview**
  TDD can encourage a focus on details that might cause tests to pass or fail and discourages large-scale program revisions.
- **It is hard to write 'bad data' tests**
  Many problems involving dealing with messy and incomplete data. It is practically impossible to anticipate all of the data problems that might arise and write tests for these in advance.

Security testing

- Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.
- The tests should demonstrate that the system can resist attacks on its availability, attacks that try to inject malware and attacks that try to corrupt or steal users’ data and identity.
- Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities (but there are still unknown unknowns).

Typical Security Hazards

- Unauthorized attacker gains access to a system using authorized credentials
- Authorized individual accesses resources that are forbidden to them
- Authorized individual accesses resources they are authorized to see but then releases them to unauthorised individuals.
- Authentication system fails to detect unauthorized attacker
- Attacker gains access to database using SQL poisoning attack
- Improper management of HTTP session
- HTTP session cookies revealed to attacker
- Confidential data are unencrypted
- Encryption keys are leaked to potential attackers

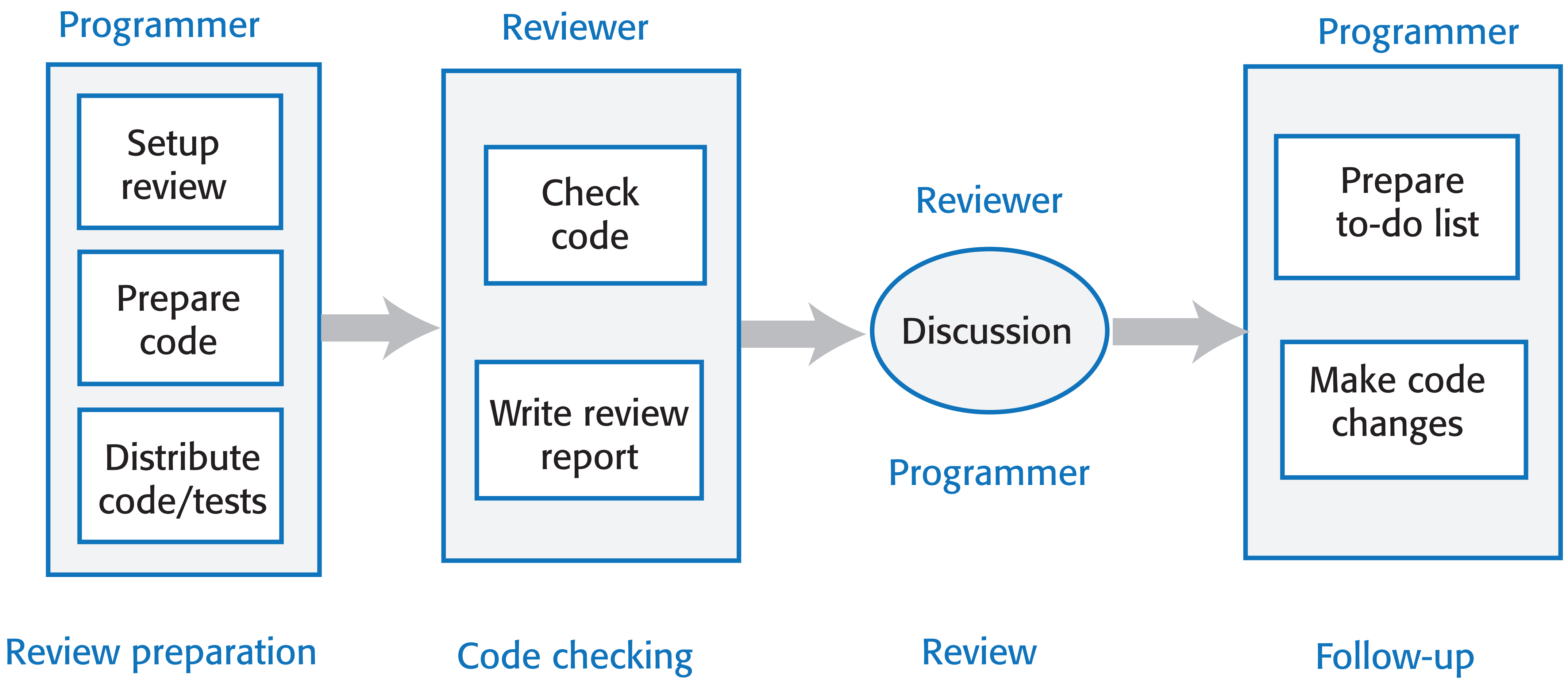
Hazard Analysis

- Once you have identified hazards, you then analyze them to assess how they might arise. For example, an attacker gaining credentials there are several possibilities:
  - Weak passwords that can be guessed are in use.
  - The user has not set up two-factor authentication.
  - An attacker has discovered credentials of a legitimate user through social engineering techniques.
- Once you have done this you can estimate **likelihood and severity** and you have risks.
- You can then prioritise what to mitigate and consider test to further characterise the risk.
  - For example, you might include biometrics as part of the authorisation in mitigation.

Code reviews

- Code reviews involve one or more people examining the code to check for errors and anomalies and discussing issues with the developer.
- If problems are identified, it is the developer’s responsibility to change the code to fix the problems.
- Code reviews complement testing. They are effective in finding bugs that arise through misunderstandings and bugs that may only arise when unusual sequences of code are executed.
- Many software companies insist that all code has to go through a process of code review before it is integrated into the product codebase.

Review Process



Review Activities

- ***Setup review***
  The programmer contacts a reviewer and arranges a review date.
- ***Prepare code***
  The programmer collects the code and tests for review and annotates them with information for the reviewer about the intended purpose of the code and tests.
- ***Distribute code/tests***
  The programmer sends code and tests to the reviewer.
- ***Check code***
  The reviewer systematically checks the code and tests against their understanding of what they are supposed to do.
- ***Write review report***
  The reviewer annotates the code and tests with a report of the issues to be discussed at the review meeting.
- ***Discussion***
  The reviewer and programmer discuss the issues and agree on the actions to resolve these.
- ***Make to-do list***
  The programmer documents the outcome of the review as a to-do list and shares this with the reviewer.
- ***Make code changes***
  The programmer modifies their code and tests to address the issues raised in the review.

Checklist for Review

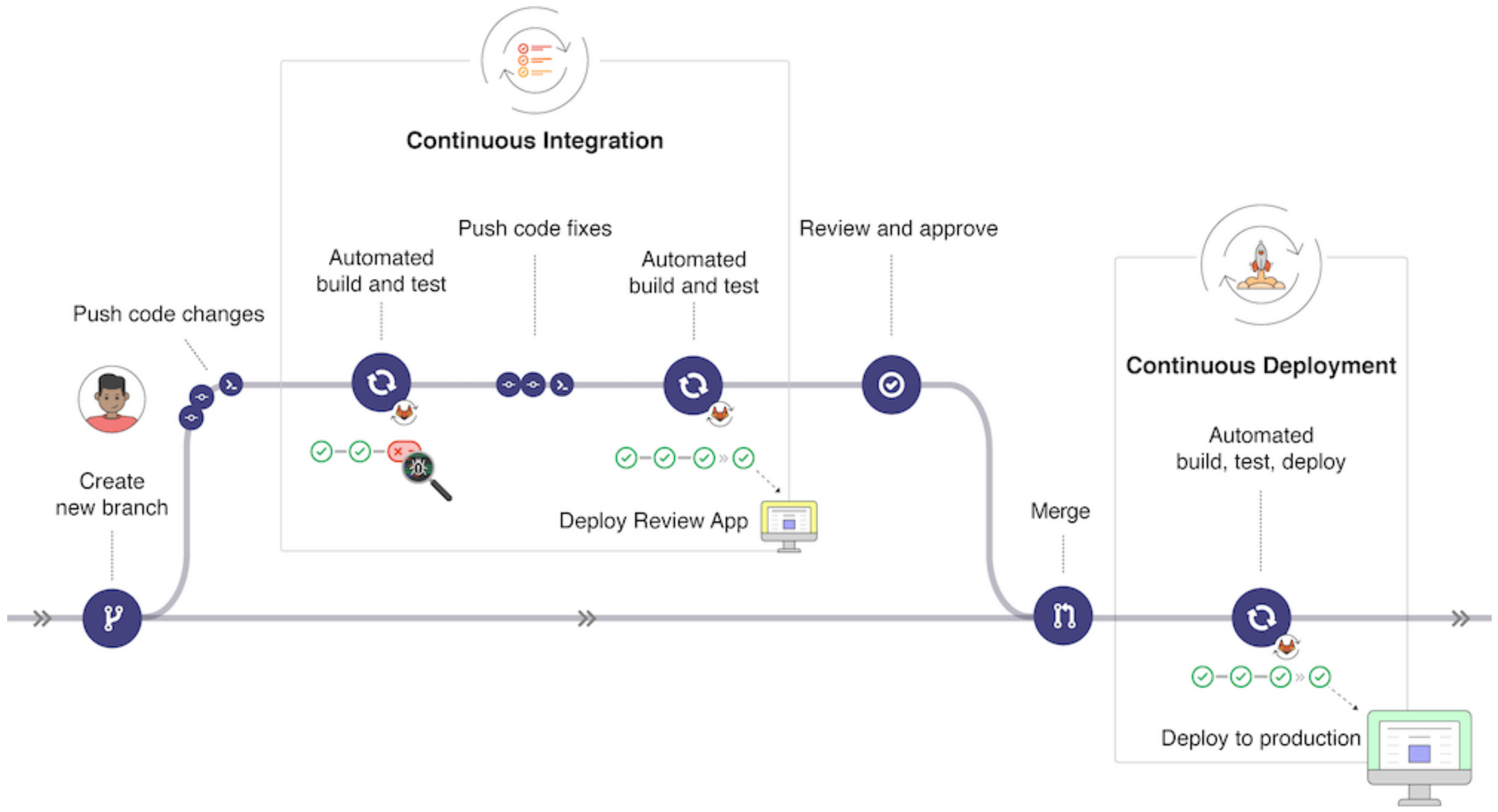
- **Are meaningful variable and function names used?**
  Meaningful names make a program easier to read and understand.
- **Have all data errors been considered and tests written for them?**
  It is easy to write tests for the most common cases but it is equally important to check that the program won’t fail when presented with incorrect data.
- **Are all exceptions explicitly handled?**
  Unhandled exceptions may cause a system to crash.
- **Are types used consistently?**
  If the language has a type system there should be conventions for its use that are observed e.g. naming
- **Is the code properly formatted?**
  There will be rules on how code should be displayed to avoid common difficulties cause by jumbled presentation of code.

Continuous Integration

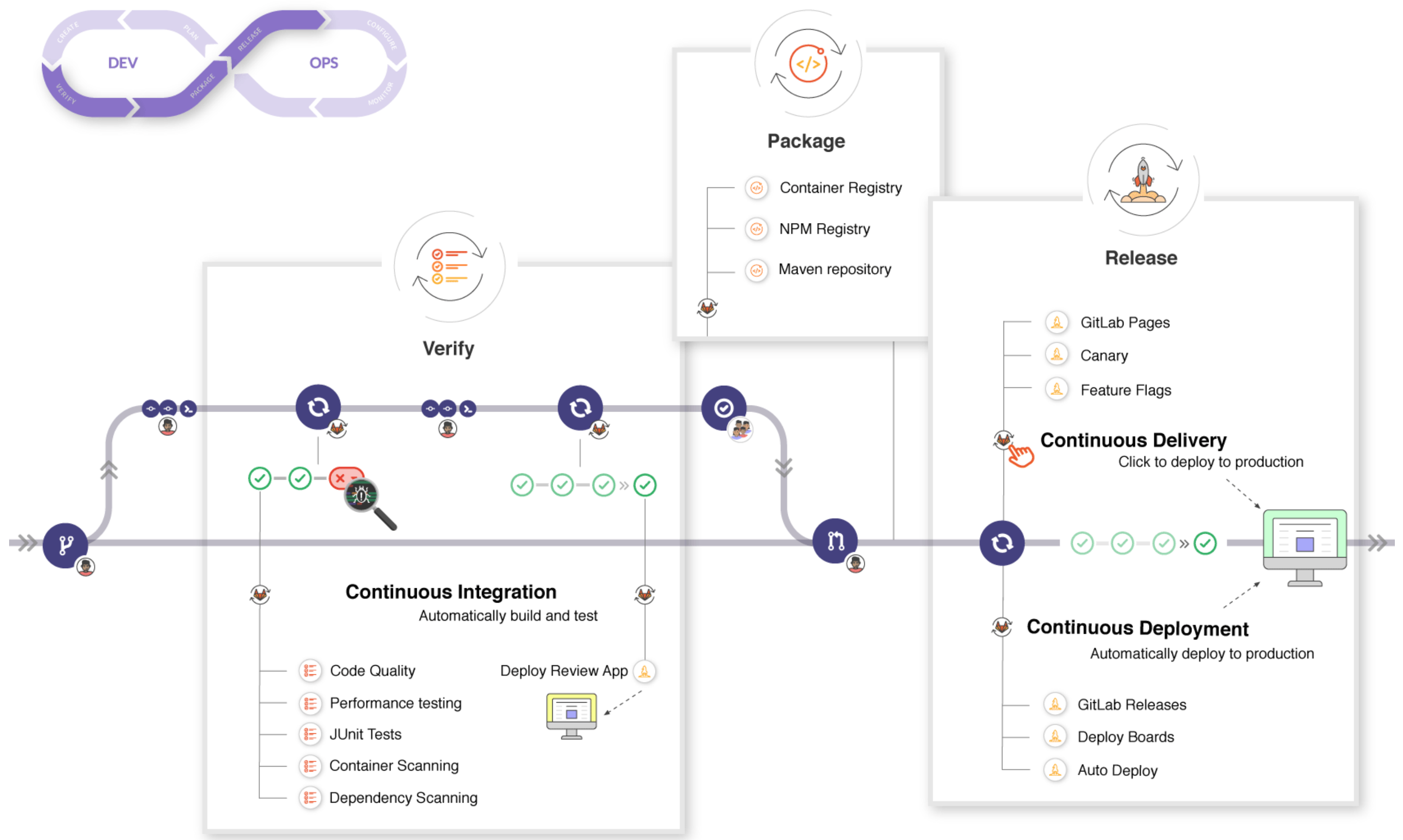
I vividly remember one of my first sightings of a large software project. I was taking a summer internship at a large English electronics company. My manager, part of the QA group, gave me a tour of a site and we entered a huge depressing warehouse stacked full with cubes. I was told that this project had been in development for a couple of years and was currently integrating, and had been integrating for several months. My guide told me that nobody really knew how long it would take to finish integrating. From this I learned a common story of software projects: integration is a long and unpredictable process.

Martin Fowler on “Integration”

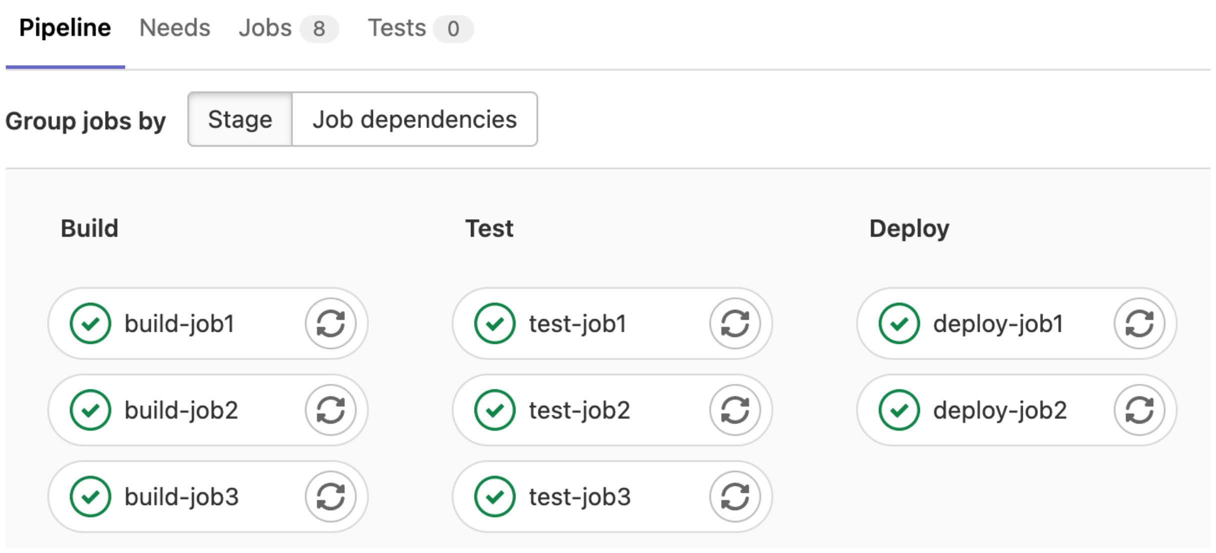
Continuous Integration in GitLab

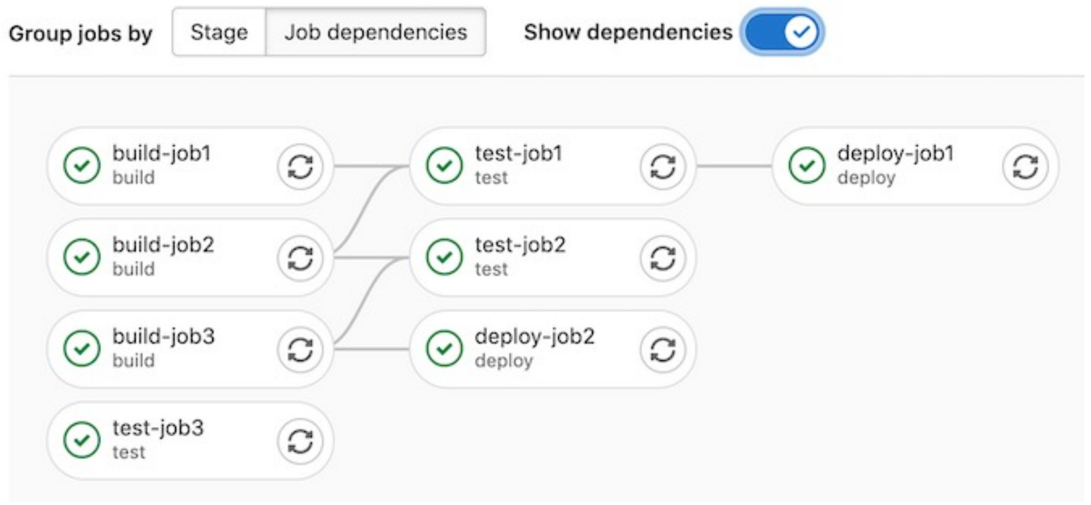


Continuous Integration in DevOps



CI Pipelines





Summary

- Testing should be automated as far as possible where there is low value from human interaction.
- A risk-based approach can be helpful e.g. in security testing.
- Review complements test.
- There are quite complete automation tools now (e.g. GitLab).