CS 4530: Fundamentals of Software Engineering

Module 12.2: Beyond Unit Testing

Adeel Bhutta, Mitch Wand

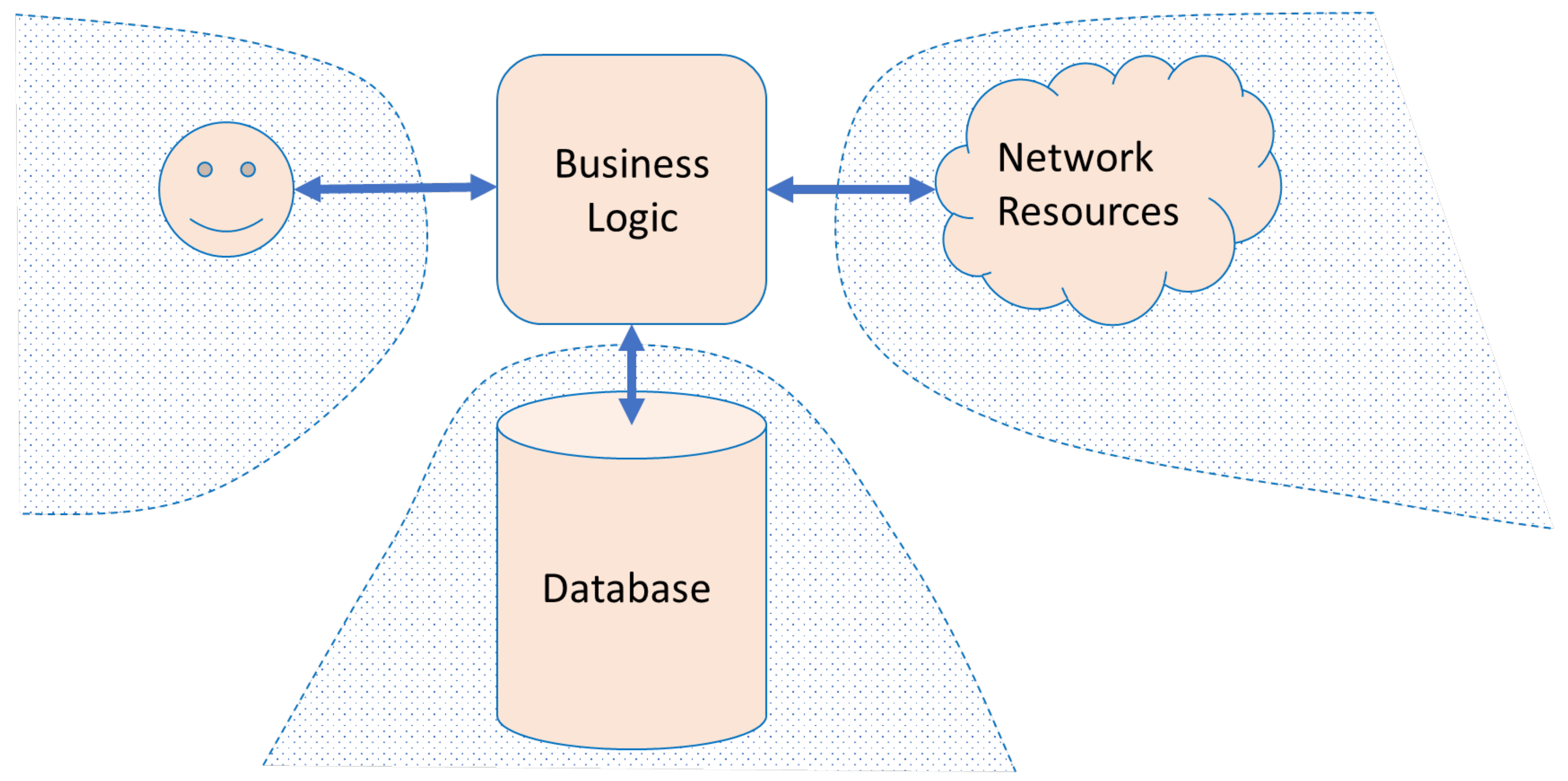
Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be prepared to:
  - Explain why you might need tests that are larger than unit tests
  - Explain why you should or shouldn’t use a mock in conjunction with these larger tests.
  - Explain how large, deployed systems lead to additional testing challenges

Large Systems are Hard to Test

- Database component
  - Contents may need to reflect/simulate real-world;
  - Data may be expensive/proprietary/confidential.
- Network connections
  - "Real" connections may be slow/flaky/disrupted;
  - Resources may have changed since test was written.
- Environment
  - Interactions with OS, locale or other software.
- Human actors
  - Ultimately unpredictable.
- Specifications are incomplete, and may change
  - Large systems -> many behaviors/interactions to consider
  - Specifications may evolve over time



Test doubles can help.

- To create "small" tests that are faster and less flaky
  - Example: Testing a unit that processes result of an external API call; only interested in testing what happens *after* the external call returns
- When the real thing is unavailable
  - Example: Integrating with external vendors
- When testing for unusual or exceptional cases that are hard to make happen in practice
  - Example: when external service fails in the middle of a transaction

Mocks and fakes can sometimes help.

- Sometimes called a *fake*, these mocks have an implementation of the object being replaced
  - A *low-fidelity* fake implements things partially
    - Enough to work for the test.
  - A *high-fidelity* fake implements most aspects:
    - Usually all functional aspects;
    - Usually not as efficiently or as scalable.
- The purpose of this mock is to avoid processes/network/cost, but still perform some activities
- Create fakes in Jest with *mock.mockImplementation(…)*

Fake has "semi-real implementation"

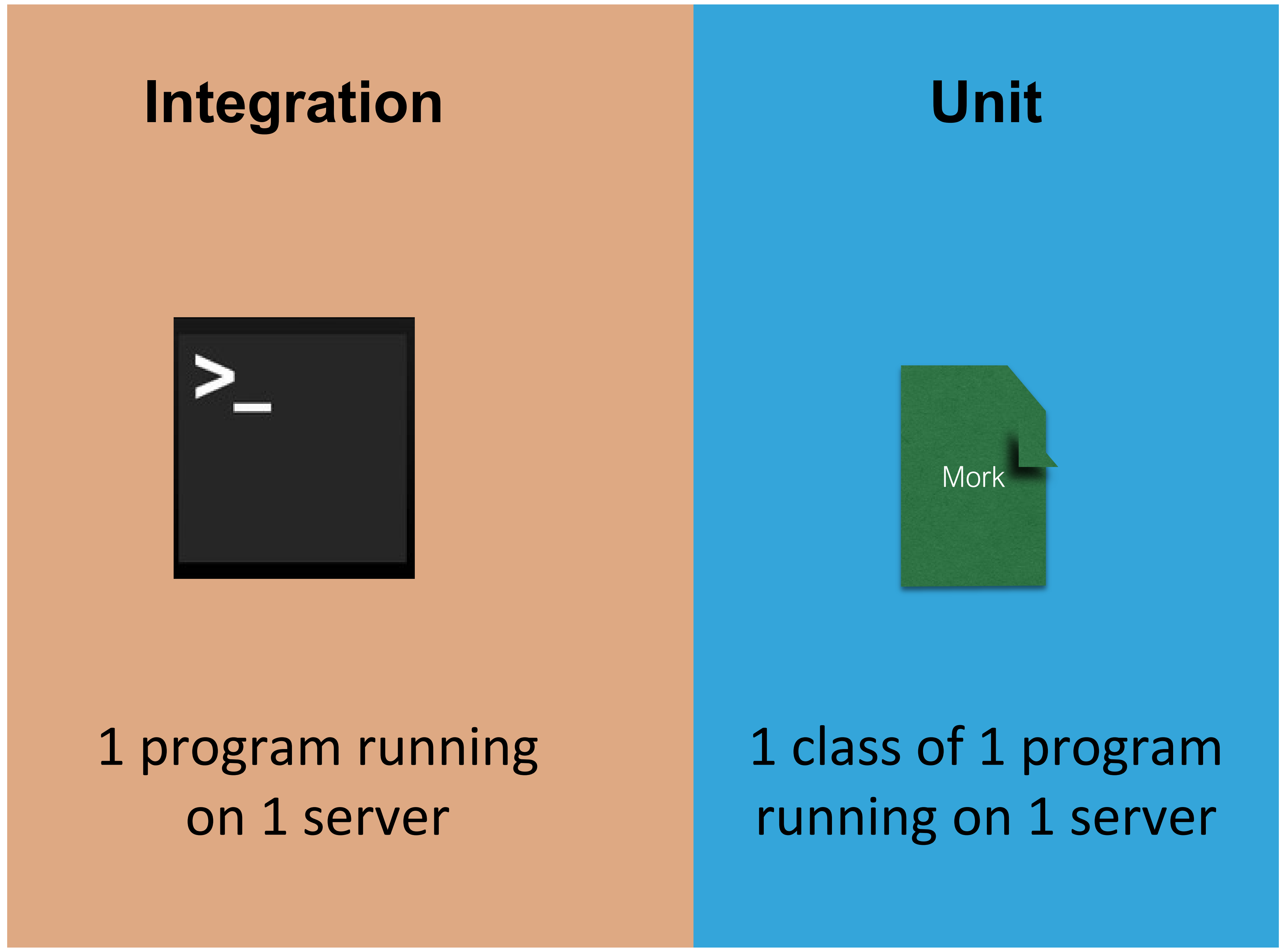
Test Doubles Have Weaknesses

- Some failures may occur purely at the integration between components:
  - The test may assume wrong behavior (wrongly encoded by mock)
  - Higher fidelity mocks can help, but still just a snapshot of the real world
- Test doubles can be brittle:
  - Spies expect a particular usage of the test double;
  - The test is "brittle" because it depends on internal behavior of SUT;
- Potential maintenance burden: as SUT evolves, mocks must evolve.

We already saw this in the preceding lesson

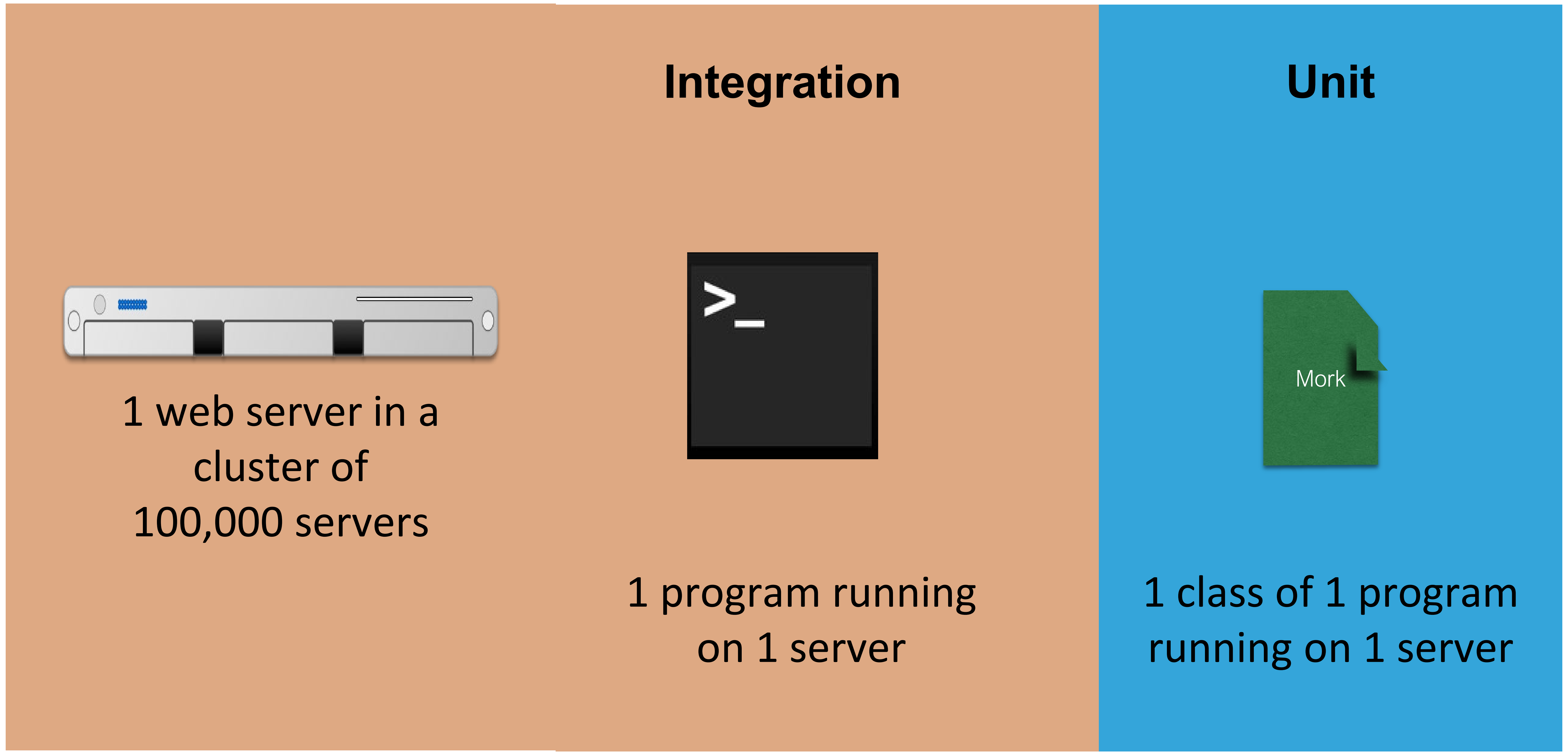
But some bugs are observable only when multiple components interact.

- These are usually because one module has made incorrect assumptions about some other module
- Unit tests won't reveal such bugs
- Mocks won't help, either (since they may incorporate our incorrect assumptions)
- So you really need integration tests

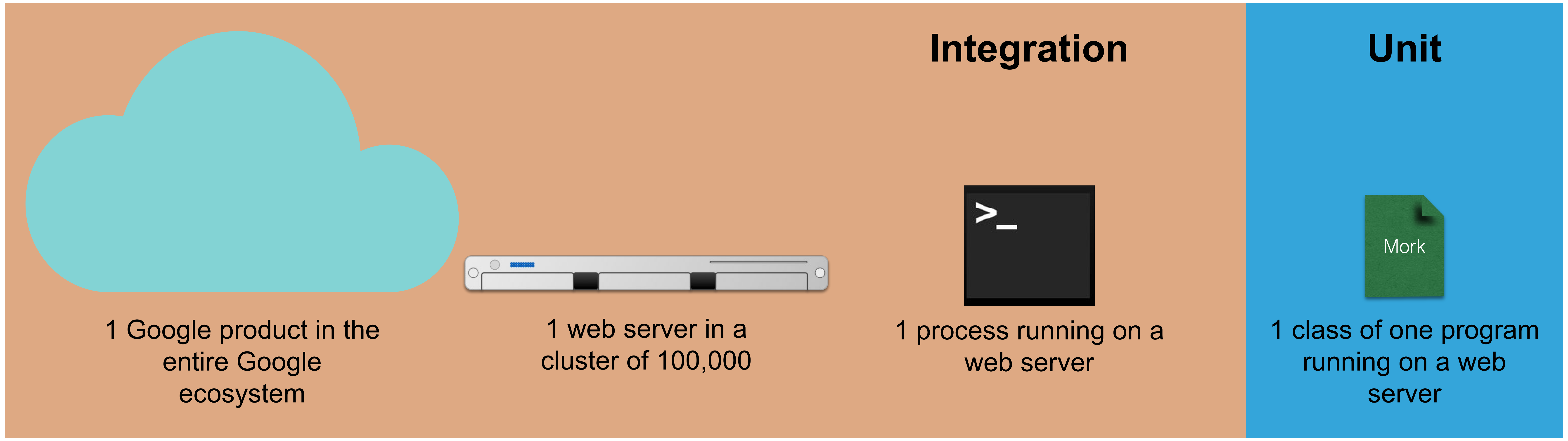

Integration
Unit
>_
Mork
1 program running on 1 server
1 class of 1 program running on 1 server

Integration tests may be larger

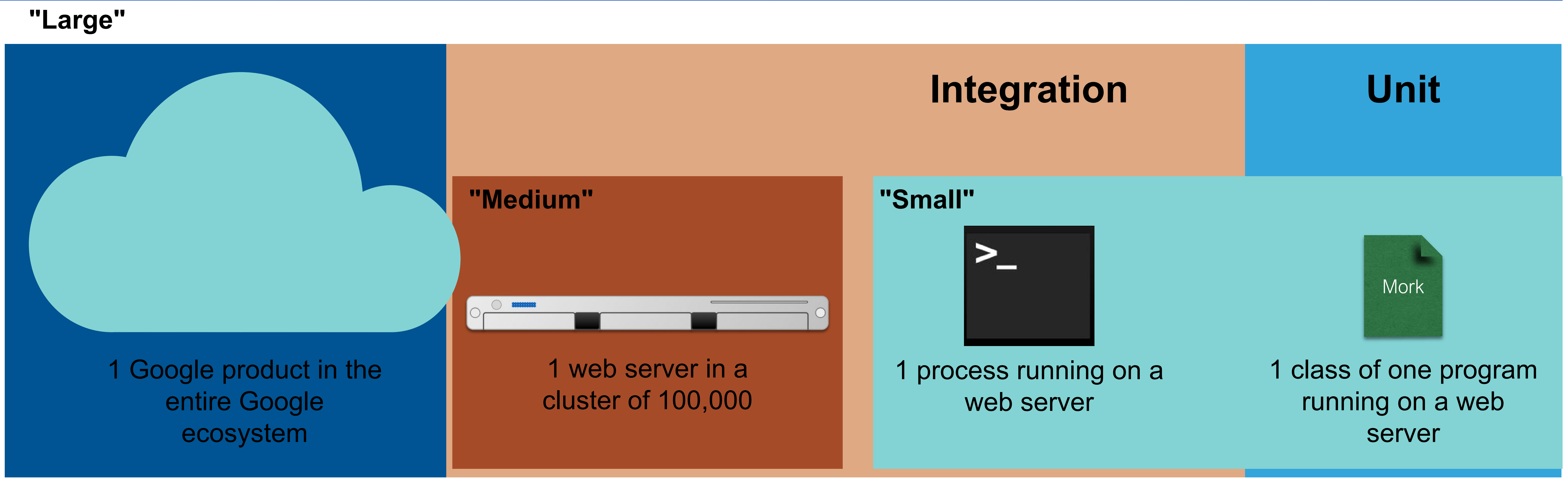
- Does the presence of other jobs on our server change the behavior of our program?
- Does the presence of the other servers change the behavior of our program?


Integration
Unit
Mork
1 web server in a cluster of 100,000 servers
1 program running on 1 server
1 class of 1 program running on 1 server

Some Tests are Enormous


Integration
Unit
>_
Mork
1 Google product in the entire Google ecosystem
1 web server in a cluster of 100,000
1 process running on a web server
1 class of one program running on a web server

Google classifies tests by "size"

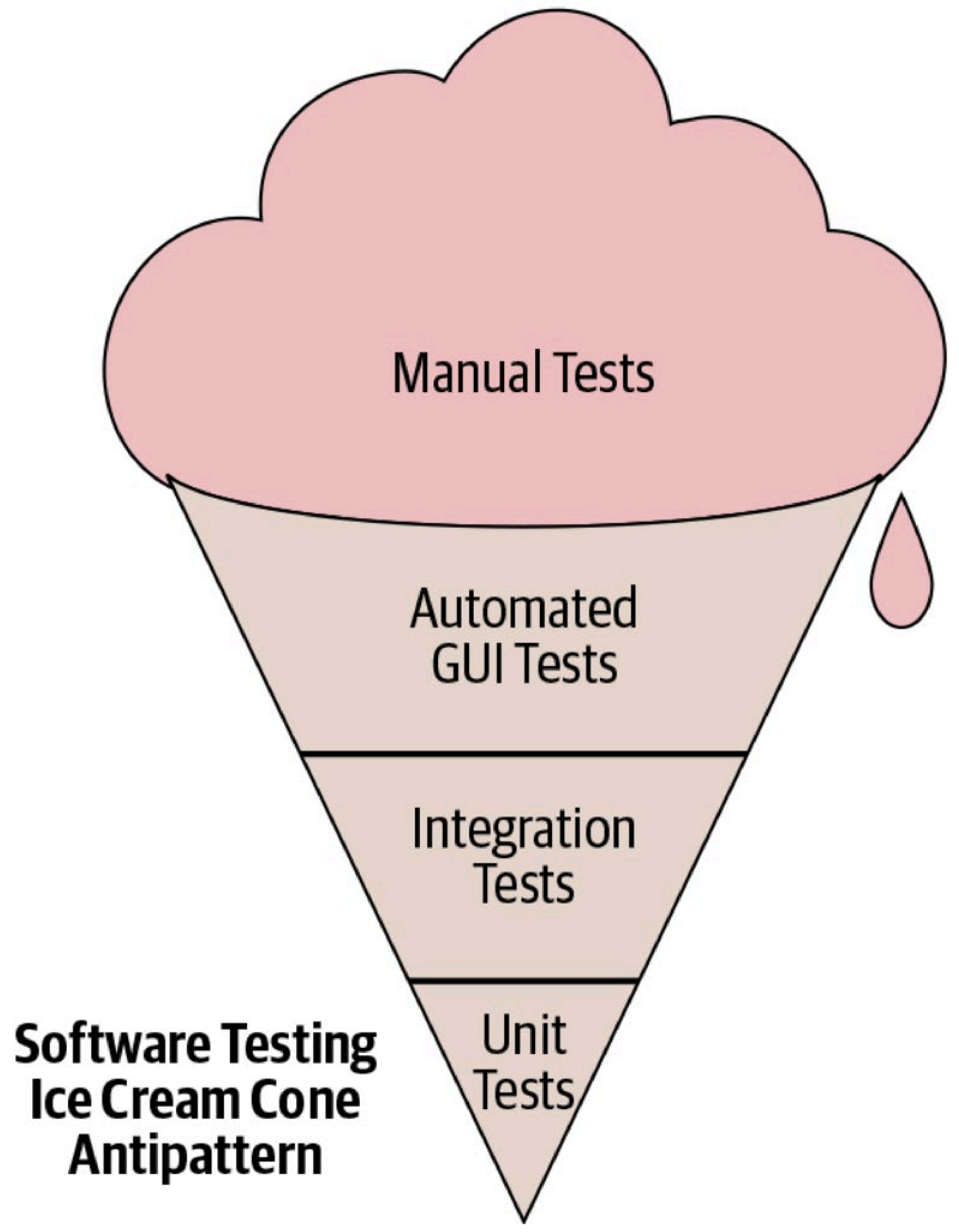

"Large"
Integration
Unit
"Medium"
"Small"
>_
Mork
1 Google product in the entire Google ecosystem
1 web server in a cluster of 100,000
1 process running on a web server
1 class of one program running on a web server

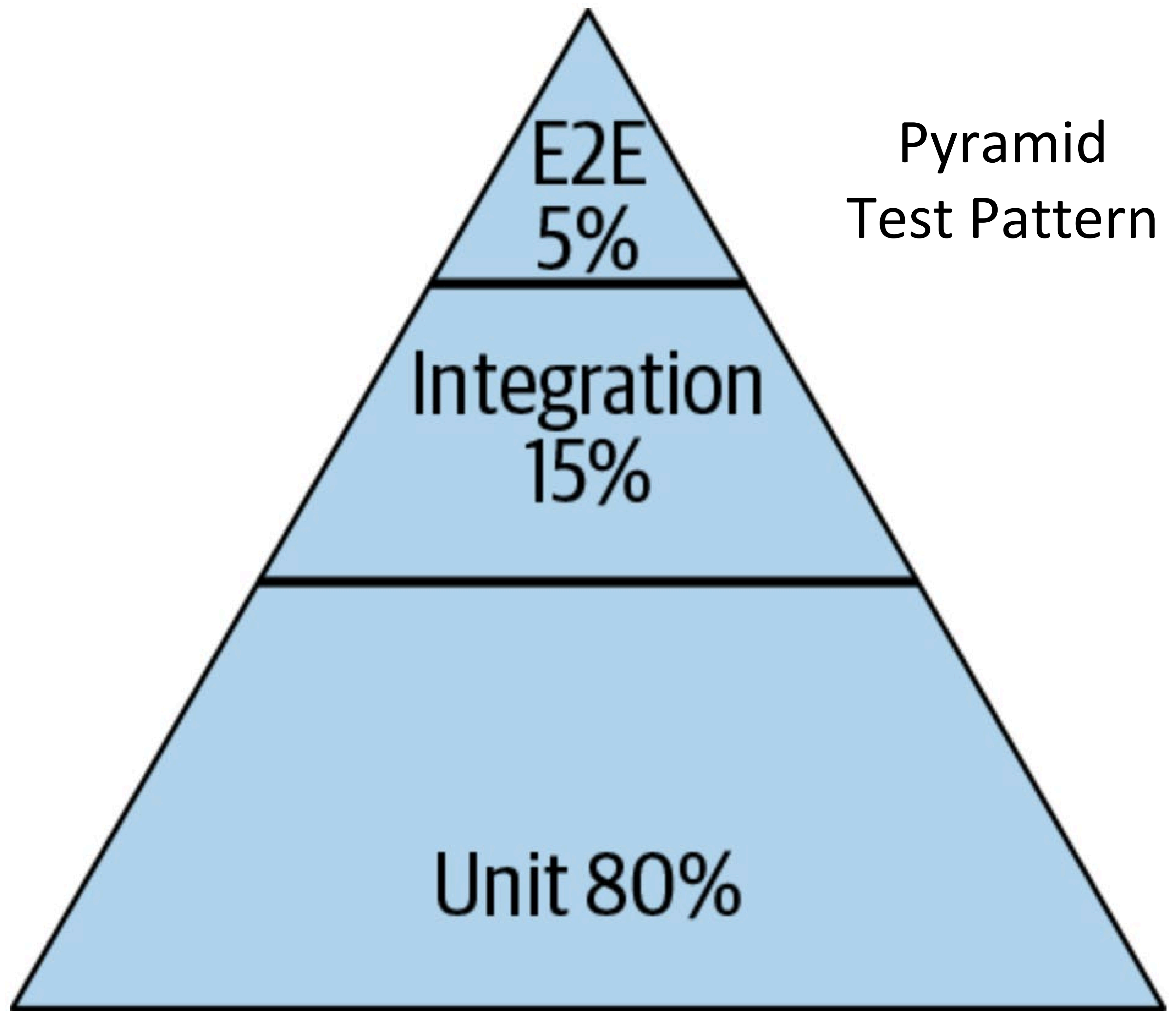
- "small" = single process
- "medium" = single machine
- "large" = bigger than that.

How big is my test?

- Small: run in a single thread, can’t sleep, perform I/O or make blocking calls
- Medium: run on single computer, can use processes/threads, perform I/O, but only contact localhost
- Large: Everything else

Testing Distribution (How much of each kind of testing we should do?)





Pyramid
Test Pattern

From SoftEng @ Google Chapter 11

- https://learning.oreilly.com/library/view/software-engineering-at/9781492082781/ch11.html#testing_overview

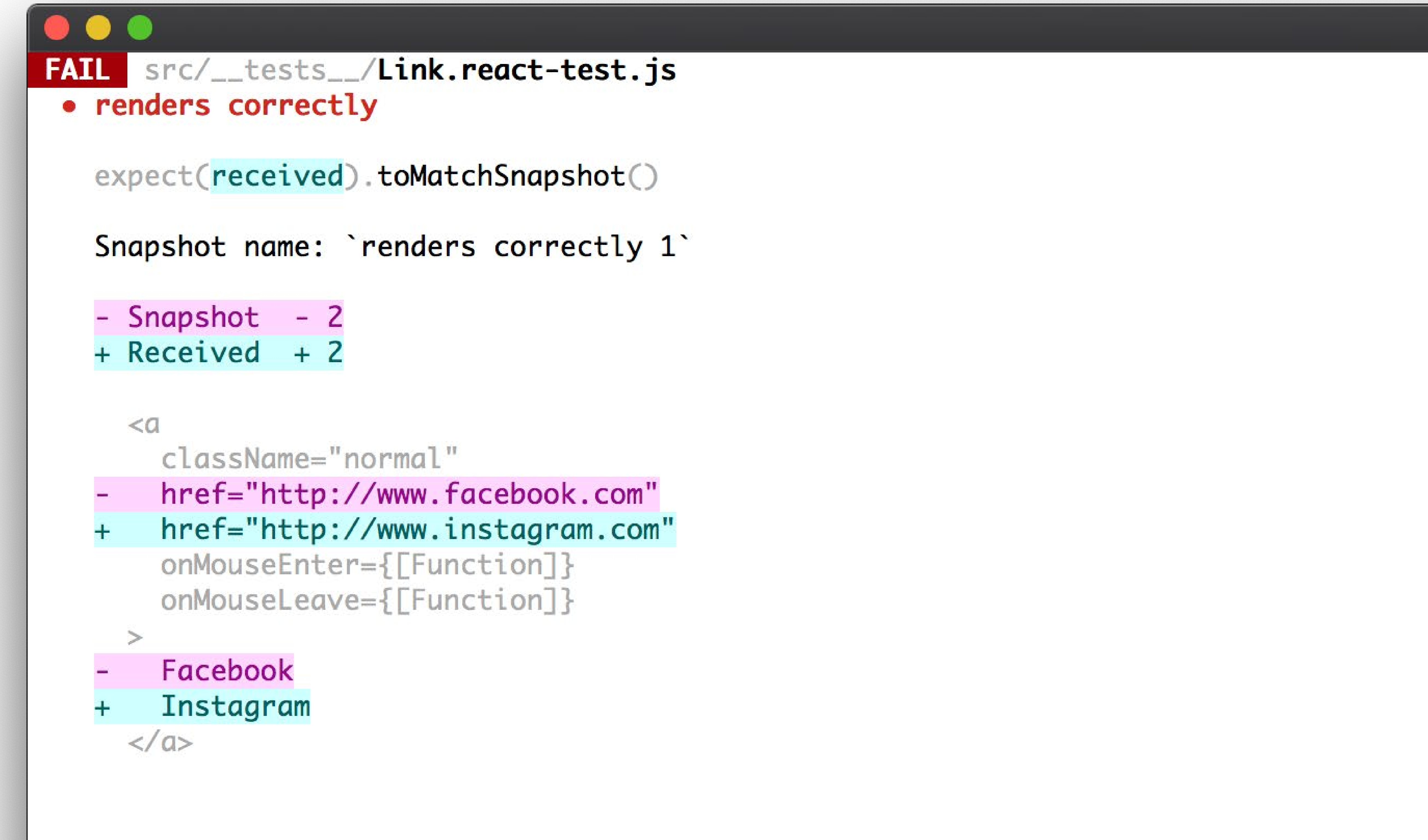
Deployed systems create even more testing challenges

- Clients believe “how it is now is right”,
  - Not “how the API intended it to be is right”
  - Writing thorough test suite is even harder, less useful
  - What is a “breaking change”?
- Still: vital to detect breaking changes
- Examples:
  - Detailed layout of GUIs
  - Side-effects of APIs, particularly under corner-cases

Snapshot Tests Can Detect GUI Changes

- The first time the test runs, it saves a "snapshot" of the rendered GUI
- Subsequent runs will fail if the snapshot changes

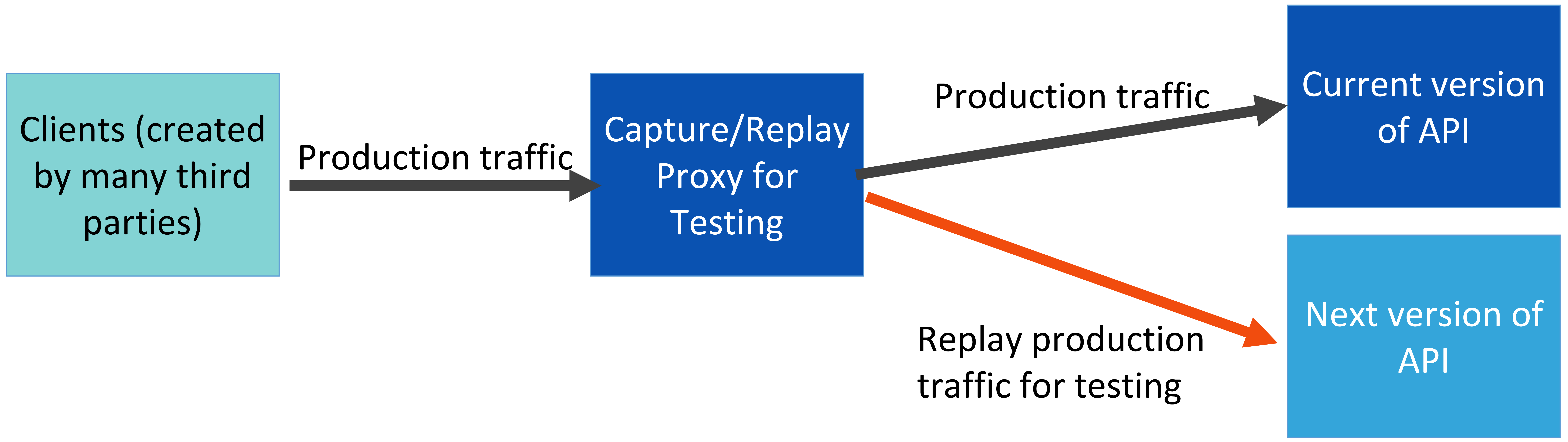
```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link
page="http://www.facebook.com">Facebook</Li
nk>)
      .toJSON();
  expect(tree).toMatchSnapshot();
});
```



```
FAIL  src/__tests__/Link.react-test.js
 ● renders correctly

   expect(received).toMatchSnapshot()

   Snapshot name: `renders correctly 1`

   - Snapshot  - 2
   + Received  + 2

     <a
       className="normal"
   -   href="http://www.facebook.com"
   +   href="http://www.instagram.com"
       onMouseEnter={[Function]}
       onMouseLeave={[Function]}
     >
   -   Facebook
   +   Instagram
     </a>
```

Capture/replay can detect breaking changes in API endpoints

- Record the API requests and responses that clients make
- Test new versions of the API by identifying requests that result in different responses ("breaking changes")


Clients (created by many third parties)
Production traffic
Capture/Replay Proxy for Testing
Production traffic
Current version of API
Replay production traffic for testing
Next version of API

https://www.tradeweb.com/our-markets/data--reporting/replay-service/

Review: Learning Objectives for this Lesson

- You should now be able to:
  - Explain why you might need tests that are larger than unit tests
  - Explain why you should or shouldn’t use a mock in conjunction with these larger tests.
  - Explain how large, deployed systems lead to additional testing challenges