Disclaimer
I will be describing what has worked for me, your milage may vary.

The problem
Automated testing brings us two main benefits: Describing the intended behaviour of our code, and preventing unexpected changes.
As with all things, it's not perfect; When we need to make behavioural changes to our code, the tests need refactoring -- and keeping hold of good structure and meaning become more and more difficult as the scale of the change increases.

The Challenge
My team at 7digital is tasked with replacing 121'466 lines of spaghetti VB and SQL mixed together, intricately tied to a dozen undocumented servers, a database that contains 535 tables, and 1296 stored procs, storing over 25 million tracks, and over a billion individual territory rights.

This is a system that is core to the company that has grown organically for 7 years -- and that no-one now fully understands.

Added to this, the company is growing and expanding into new business areas. And all of the major digital music suppliers are transitioning to a new global standard (that is of course being interpreted differently by everyone who uses it).

So, we knew that we were in for a big learning experience, and that things were changing fast enough that we could never complete a single transition.

We needed to cope with change in a major way.

How we think
However we were going to approach the problem, there were enough problems without making the tools difficult to use. We wanted something with very low *mental friction*.

So we had a look at how we describe complex systems, and how we describe changes to them. We drew a bunch of boxes, each of which was a small feature, described in a few dozen words. And we drew a bunch of arrows pointing between them.

Then we introduce some changes. Sometimes we move arrows, sometimes we scratch out a box or put another in.

So that's how we find it easy to describe what we're doing. Let's try and do it in code.

Making it happen
Boxes and lines

We designed a system where there is no central knowledge of all the little arrows in the diagram.

We use a broadcast+selective listener system that makes the receiver responsible for routing messages. The phase “sends a message to” is banned within the team. Any message can be sent from any service, and if no-one is listening it will be ignored and lost.

Each component is it’s own binary executable, and can be running on any machine in the network. Each component has it’s own history and we can release versions out-of-sync.

The routing of messages happens by a hierarchical system of message contracts. Services can listen for more generic versions of messages, but won’t receive more specific messages than the ones they have registered. This allows us a two-deploy method to re-route the system with no down-time and without affecting monitoring systems or any other part of the system.

Keeping things composable
Having worked with “saga” systems before, I feel that they work well to pin-down a well understood and tightly controlled flow -- which is exactly what we didn’t want.

There are also no transactions in the system -- nothing ever goes backwards, because if the system has changed there may no longer be anything to go back to.

Testing
From a testing viewpoint, we still want to take the benefits of test driven development: a good understanding of the code’s behaviour, and protection against unexpected change.

First we decide to that each component will have a very specific purpose, and well defined “edges” -- the place where this component start and stops. We make it as small as possible, and we strictly enforce that no code crosses that line.
This protects us from feature creep and accidental dependency within each component.

Next, we bring in the tiered testing that is standard in 7digital -- unit tests, integration tests and acceptance tests.

Unit tests are, as always, the way of ensuring that the code we have written does what we expect at the level of functions and classes.

Next up are integration tests, that check that external components and other systems are behaving as we expect. These tests we try to minimise; each component should be doing the bare minimum, and should interact with as few other systems as possible.
Where possible, there aren’t any integration test, only unit and acceptance tests.

Acceptance tests are at the outer edge of each component, and describe the visible behaviour, as close to a well written manual as possible.

Acceptance tests are where the meat of a service's behaviour is kept. We are trying to communicate the purpose and intent in our acceptance tests.

We have designed our messaging system to isolate code from the dirty business of actually sending and receiving messages -- and the messaging can be set to a 'loopback' test mode that causes all the real-world behaviour in an idempotent way.
This means that a service's behaviour is almost entirely defined as messages sent in response to messages received.

Smoke tests have been entirely removed, except for a quick check that a service is the correct version once deployed.
So how do we ensure that all the composed behaviours are working as expected? We have fused the roles of logging, monitoring and smoke tests into a single unit.

End-to-end monitoring is enabled by the broadcast nature of the monitoring -- every message sent can be received by any service without interfering with any other (in fact, without their knowledge). Each message needs to carry with it enough information for the system to progress, and this is available to any number of monitoring services in real time.