Experience Report: Driving the Adoption of Exploratory Testing on Cloud Foundry

jalford@pivotal.io
Jesse Alford

I have spent the last year and as an “Exploratory Tester” at Pivotal. At times I have been the only person in the organization with that title. Pivotal has an opinionated and established approach to software development, iterated on and validated by Pivotal Labs and their clients for over twenty years. Pivotal Labs was founded as a consultancy in 1989, and has been a proponent and practitioner of Extreme Programming since that movement’s infancy. In 2013, Pivotal Software was spun out of Pivotal Labs, EMC and VMware to produce software products the Pivotal Labs way. Testers as such have not, until recently, been a significant part of the Labs-derived development process Pivotal now uses on its own software products. We’ve learned a lot by adding testers to the mix. This is a snapshot of how it’s gone so far, what we’ve learned, and what we’re thinking of trying next.

Cloud Foundry

The product I work on, Cloud Foundry, is *complex.* It’s an open-source Platform-as-a-Service (PaaS) project composed of dozens of repositories and governed by an open foundation with a contribution-based governance model. It’s deployable across many possible infrastructures and configurations – there are tools to create resources for tools to configure tools. There are commercial offerings and an open-source core.

About a hundred and fifty people work on Cloud Foundry product teams at Pivotal. All of them test. My co-presenter Karen Wang is a Product Manager (PM). Previously, she was an engineer. I am called a “tester,” she is not; she has done as much testing as I in the last year. We may have few testers *qua* testers, but we test a *lot*. Our approach also integrates an aggressive slate of other practices aimed at consistently delivering high-quality software.

Developers pair as close to 100% of the time as is reasonably practicable. Test Driven Development (TDD) is normal and constantly refined, debated and *practiced*. My own anecdotal experience of walking around the office suggests that pairs are roughly as likely to be discussing a testing problem as an implementation issue at any given time. Continuous Integration status monitors and production performance charts loom large. Product Managers write and prioritize feature stories, getting feedback from designers and programmers both on the fly and in frequent iteration planning meetings. Once features have made their way through the automatic checks and extensive specs employed in the CI pipelines, PMs perform acceptance testing, pulling in designers or programmers for support as necessary. Designers do UX interviews and usability testing. Pre-release builds are routinely circulated to field engineers and support, who provide feedback to product teams. Products themselves are sometimes considered tests, with a generalized belief that releasing something valuable as soon as possible and finding out how customers relate to it is essential to learning to build the right thing.

So that's where we were before we had anyone explicitly dedicated to testing at all. Everyone tests. But we wanted more.

Why "Exploratory Testers?"

The way Pivotal does things is born of experience. Much of that experience is from contexts dissimilar to Cloud Foundry. Labs projects are smaller, less distributed over time and space and stakeholders. They tend not to have as much of an operations element, and labs customers often have their own testing approaches and resources.

The Directors of Engineering wanted additional approaches to handling complexity and the risks it introduces and conceals. Elisabeth Hendrickson, the Director of Quality Engineering at the time (and not coincidentally, the author of the very useful book Explore It!) thought adding exploratory testing to our slate of practices would help equip teams to deal with the additional complexity. She set out to hire a few exploratory testers to rotate amongst product teams. She provided a strong emphasis on the notion that the explorers were to be "just another engineer," albeit one hired for their specialist skill in testing instead of their competency in writing software. The directors have a preference for testers with some development capability, but this is substantially because our product is for developers – though it is also helpful to be able to pair on feature stories.

I was the second exploratory tester, and had only the barest minimum of development skills. The first, David Liebreich, was also a competent developer, so I was also the first who started without development chops. The team I started on hadn't had a tester before. Elisabeth set me up with a simple formula for introducing testing sessions to the Pivotal workflow. I would work with the PM to write charters, then pair with team members to perform sessions based on the charters. Reports, bugs and features were to be the deliverables of that work, and it would be up the PM to accept or reject them, as with feature stories.

The intention was that this would lead to writing and performing charters becoming part of the engineering skillset of the team, and once I rotated to another team in a few months, the skills and practices would remain. Testing specialists would seed the organization with the practices and skills to organize information-seeking activities and expand opportunities to notice problems, tighten feedback loops and improve the quality of the product.

A note, too, on terminology. With "TDD" and "Unit Tests" and "Integration Tests, Usability Tests" the term "test" is overloaded in our environment, and the term "tester" unclear. This helps explain the use of the pleonastic term "exploratory tester" in my job title. "Exploration" became a clear shorthand for the distinct approach to testing software we were adding alongside existing practices.

Charters: the Crucial Adapter

Product teams work out of prioritized Pivotal Tracker backlogs. Pairs should be able to pull and work the top story. This solves what I call "problem zero" most of the time for most of the team. Problem zero is: "what should I be working on *right now?*" Problem *one* is *actually doing whatever you should be working on*, but problem zero has to stay solved to get anywhere on

problem one. Keeping problem zero solved with a groomed backlog is a PM's responsibility to the team.

If engineers regularly need to do a type of task, we have to be able to coordinate it in Tracker, our solution for problem zero. Charters are how we do that for testing sessions, in the same way that feature stories are how we do that for implementation work. Charters are typically stand-alone stories, though the framework can be used for exploration that needs to be done as part of another story, as well.

Our charters use the template from *Explore It!* They specify three things:

i. The target of the session.
ii. The resources to be used in the session.
iii. The informational objectives of the session.

These are strung into a single statement: **Explore** #{target} **with** #{resources} **to discover** #{information}.

When I roll onto a team that hasn't had a tester, charters are the first thing I introduce. I explain that the deliverable of charters is *information*, in the form of a charter report and stories filed in the appropriate backlogs. I've had success with an initial meeting with the whole team, an hour long, where we discuss where mystery, complexity and risk might lurk in the codebase, how we might go about investigating that, and collaboratively writing our first few charters on a whiteboard. I emphasize simplicity with these first charters, and stress that follow-up stories from a charter typically include additional charters. Here's a set of examples from a team that had experienced unpleasant surprises (for instance: unintended emails to a large number of users) due to the complexity of data in their database.

i. **Explore** account state **with** a Prod DB dump to **discover** how many records are in each state (and what states are actually in use)
ii. **Explore** sponsored accounts **with** a Prod DB dump **to discover** inconsistencies in sponsorship events
iii. **Explore** cancelled/suspended accounts in the prod DB with **with** org state in the CC API **to discover** if there are active orgs that should have been suspended

I think it's notable that these initial charters barely need the relatively heavy support of a structured template at all. You could rephrase #1 as "get a count of accounts in each state from the db" and any engineer on the team could turn that around for you with a couple of minutes of SQL work. Given that, some might reasonably ask why we'd write a charter around it at all.

The first answer is that the purpose of that initial meeting is to write some charters as a group. If I ask "how many registration states are in the database, and roughly how many records are in each of them" and the team can't readily answer, my end of the exchange goes like this:
"It sounds like maybe we have an **informational objective**! What **resources** would we need to fulfill our objective? Okay, great. What, ultimately, are we trying to get information *about*, here? Yeah, okay, that's our **target**. Nice work, you just wrote your first charter! It's very simple, but if we do it in an exploratory mindset, we'll find ourselves with enough information to write more charters very quickly."

The second answer is that a testing session is *different* from looking up some values in a database, even if the latter is how the information is ultimately obtained. Developers pairing for their first testing session are in for a lot of novelty; a *simple* charter can help.

These initial sessions are about achieving the informational objective, sure. But they're also about learning test execution skills: being reminded to *notice* things, to ask if each thing noticed could be a problem (and why and how did we notice it, anyway?). They're about being asked to articulate how we might be able to tell if that was a problem, and if that would be worth learning about right now, or noting for later. We use the charter to make decisions about if a test idea is on target or off, if it serves our current explicit informational objective or a different one, if it's related to the resources we've allocated ourselves or not. And then we write a charter report, and we talk about what should go in it and what should be left out - again, turning to the charter for guidance in how we make these inherently heuristic decisions. If at all possible, we try and make sure we write some follow-up charters as one of the outcomes for these initial sessions, so we get more practice in charter writing, too.

As the more experienced practitioner in this, the tester has an opportunity to reinforce that good testers have *skills*, that there are patterns and tools to be learned, that while testing can be approachable, it is not an unskilled task that people automatically know how to do well. Besides which, this is likely the *tester's* onboarding to the team, too.

We don't do a lot of explicit onboarding when people rotate between teams; pairing all the time takes care of a lot of that. But doing simple charters leaves the tester especially free to ask project-familiarity questions like "where are our ssh keys stored" or "how does this repo's directory structure even work" while keeping the high-status mantle of the mentor in regard to other parts of the conversation.

Once we've got a steady flow of charters being written and sessions being performed, we review and refine (and sometimes estimate) charters in weekly iteration planning meetings alongside the other stories. It has been occasionally helpful to have weekly charter results review meetings, especially if the informational objectives are of general interest to all developers and followup work is being aggressively pursued – this was the case with the database-related investigation above. This gives the team a chance to discuss and consider general implications of problems that might have far-reaching consequences, or might be challenging to address. As a general practice, though, I have found it effective to avoid whole-team meetings dedicated to working with and discussing charters. Integrating discussion of testing activities into the existing framework of IPMs, retrospectives, extemporaneous story-writing and planning workshops cements the status of testing sessions as a normal, if new, part of routine development work.

Business Value

I have been in my share of conversations about the difficulty of arguing the relative business value and priority of testing. I've attended talks about it, I've read articles about it, I've sat with experienced testers as they gripe about it, it's been covered in classes I've taken.

But I haven't had to *do* much of it.

Elisabeth Hendrickson was one of three directors of engineering for most of the time I've been on Cloud Foundry. She is very well regarded at Pivotal, and exploratory testing is known to be kind of her *thing*. She's given repeated talks and demos at labs over the years prior to joining Cloud Foundry, in addition to working on labs projects. So when exploratory testing comes up as a possibility in the parts of the organization where she's had the most influence, people assume it's a good and useful thing, and want to do it. This leads to problems of its own, (shallow agreement and "best practice"-type thinking) which we'll discuss in a bit. The point is, it seems like in a lot of contexts testers have to fight tooth and nail for resources, recognition, respect and access, constantly reminding the organization of the value they provide.

At Pivotal, we've held retrospectives on the practice, and people have told *me* about the business value we've realized out of the practice, which by all accounts is a nice reversal.

Product Managers enjoy having a way to get more depth of understanding about the state and workings of their product, not to mention earlier bug reports. That kind of information is useful when considering what the team should be working on, seeking feedback from users, and adjusting their product roadmaps. In the database examples above, the PM (Karen, my co-presenter) found that having information about the state of the database was exceptionally useful in prioritizing work and deciding what features to commit to, and in gaining confidence that deploying changes that had profiles similar to changes that had caused problems in the past wouldn't cause the same kind problems again. It was also powerful to have information about what parts of the product could benefit from sustained focus – there were times when testing revealed enough small issues in an area that redesigning it got moved up on the schedule. Finally, there were times where even when issues weren't prioritized for an immediate fix, when customers *did* encounter them, the PM was equipped to discuss the issue, having already seen the situation described and fixes consciously deferred.

Engineers have expressed that they appreciate having a way to organize initial investigation around areas that will be impacted by new features. They also like that when they notice something that might be worth following up on, the charter format gives them a structure that can help ensure they capture enough information to allow a different pair to investigate it effectively later. There have been times to when programmers approach me and say something like "hey, Jesse, I just wrote some maybe risky code. I wrote the specs I can think of, and it seems okay, but could you help me explore it?"

With our relatively frequent rotations and long-running projects, engineers are often working on parts of a codebase they had no hand in and would otherwise be unfamiliar with; performing charters gives them a chance to pick up a lot of context around the target of the charter, regardless of the particular informational objective.

Designers appreciate testing sessions' ability to reveal edge-cases, interactions and quirks in visual elements that acceptance testing and automated checks can very easily ignore or miss. My personal favorite in this category was a case where the login button was hidden behind a "learn more" icon in an older version of IE - a version that didn't need to be fully supported (thank all that is good and right), but *did* need to allow users to login.

All of these things directly support balanced teams in delivering high-quality stories in ways they are willing and able to articulate.

Missteps, Surprises, Provisos and Future Victories

Performing the Right Testing Remains Difficult

Once when rolled onto a team, a senior programmer asked me how I would avoid “testing things we’re already covering with automated tests?” I did not laugh much, and told him it wouldn’t be a problem. In a sense, it wasn’t; we found worthwhile bugs with chartered testing on that project.

In retrospect the question addresses half of a difficult problem. Now I tell people:
“There will be overlap. We’ll be interested in some of the same things, and so we’ll try things, and look for things, that are being checked automatically as well. That’s okay. We can, in looking, notice problems the checks aren’t checking for. Direct human observation is a more powerful oracle then coded expect statements. But we will try to allocate our effort to questions that specs and CI are not well situated to answer, to the non-obvious, to scenarios unlikely to be constructed by the specs, to failures unlikely to be watched for, to finding information we know we aren’t already getting from elsewhere, or that we don’t even know we’re missing, yet.”

This is a better answer. But it’s a recipe for testing that potentially focuses on information that’s of dubious value to the project. In the worst case, testing becomes an episode of “dumb tester tricks,” amusing but frivolous. On the project where I downplayed the question of overlap, I filed a number of bugs that were briefly discussed, then deleted. For instance, a product’s description on a download site would fail to wrap if the description went on for a very unlikely number of characters without the use of spaces. I found this using one of my *favorite* potential dumb tester tricks, a counterstring generator.

For those of you who are unfamiliar with this simple, wonderful tool, it generates a string that indexes its own length. This is a great way to score some easy bugs (and get familiar with a system’s limits and make some guesses about how it’s interacting with its data persistence layer, if you don’t already have information about that, but I digress) on a product. However, no one cares that prose descriptions written and double-checked by trusted product owners won’t wrap if they don’t contain the single most common character in the english language. Or rather, to the extent that they care, they feel great when they delete the bug, because they *must be getting some really thorough testing and not have any problems if those crazy testers are hitting it with all they’ve got and this is all that’s turning up*, right? Well, no, it’s just an impressive trick. (I love counterstrings. They’re very useful. I have nothing against them. But, like many clever and useful things, they can lead you astray.)

The problem is that in focusing on areas and issues unlikely to have been covered by TDD, BDD, CI, Acceptance, and Usability Testing, the chance that you are focusing on something that doesn’t matter goes up. These practices are intended to give people many chances to consider how they want the product to work and how they want it to fail, to experiment and check and notice and validate. So the original question might’ve better been phrased “how will we avoid

putting our effort into testing things we've already put a lot of thought into without wandering into the woods and testing things we haven't considered *because* we don't care about them?"

This has been a problem. I have had some success addressing it with negotiation and collaboration with PMs about which charters are prioritized. It is more likely that charters produced in collaboration with PMs will turn up information they actually care about. While this is not a surprise, one interesting side-effect is that charters emerging from dedicated charter-writing meetings with PMs who come into it unsure what they want tested can tend toward questions about the product (or an integration or dependency) that someone other than the PM already knows the answer to. These charters are taking the place of a conversation, and while they might reveal more, they're expensive and may reveal *less*.

If things begin to feel this way, it can be useful to have a programmer in the room. While our developers' tendency to favor checking their mental model or reading code in order to answer questions can sometimes lead to resistance towards performing empirical tests, it can also quickly answer questions that otherwise would turn into expensive information-seeking charters. Navigating the difference seems to be a matter of experience, in that I am not yet experienced enough with it to articulate my heuristics.

Getting these discussions right is crucial. It's the difference between writing charters that turn up a bunch of low-priority bugs but happen to be easy to think of and execute and writing charters that lead to important discoveries, though they might've been more difficult to think of, put into words, and perform.

Problems and Plans

- Testing of features was requested less than I expected. I intend to adjust how I coach PMs in writing stories, so that I encourage them to think of risks in connection to features, as it has emerged that this is testing they want but don't know how to put into charters, yet. One thing we're going to try to mitigate this is to encourage exploration of feature stories by the pair that did the implementation prior to delivery.
- People often demonstrated a few false starts when learning what to use charters for and how to write them, and people trying to use charters in good faith while suffering shallow agreements sometimes expressed confusion about why charters were being used instead of story patterns they already knew. I address this by demonstrating what sort of decisions each part of the charter is helpful for deciding, and encouraging people to think of that sort of question when applying the template.
- We have had more success spreading the coordination, communication, and planning skills around the use of charters than in spreading test execution and reporting skills, though the appreciation of these skills has certainly grown. I intend to address this by explicitly naming heuristics and mnemonics as I use them, and putting a name to such when I notice my pair applying something I have a name for. There is precedent for this among pairs in various named implementation patterns.
- Testers that can be expected to succeed in this environment are difficult to find. The requirement that they be able to introduce the testing practices to teams who have no experience with them, the requirement that they be unafraid of command line tools and distributed systems, and the requirement that they be unafraid of pairing with programmers on feature stories have all been challenging. Our plan for addressing this

relies on growing testing skills in our existing engineers instead of expanding the role of dedicated testers on teams.
- Allocating testers as supernumerary to teams that are generally even (to facilitate pairing) leads to a lot of logistical issues, but is hard to avoid if testers are to rotate more frequently than typical engineers do and/or can't productively pair on implementation work. We're making an extra effort to allocate testers in a way that leaves teams even, and to ensure that testers are prepared to be useful pairs on implementation work (which doesn't always mean being able to program).
- Use of charters can fall off once a dedicated tester rolls off a team, which leaves the team with the same blind-spots to potential problems they had before developing the exploratory practice. I suspect this is because while they understand the chartering/ organizational tools well, the test-execution skills necessary to realize value from charters haven't taken deep root, and in the absence of a skilled tester, charters begin to feel less worthwhile. Our plan here is additional focus on developing testing skills (and the ability to transmit said skills) among interested programmers by way of experiential workshops, community-of-practice discussion groups, and adding exploration as an area developers might expect to be rewarded for growing in.

Miscellaneous Additional Learnings

- Teams new to testing sessions take time to learn how to handle mid-session hand-offs if a pair rotation switches who's working on a charter before that charter can be completed. We had an upgrade bug that we *knew we were looking for* make it into a release because the cycle time for the group of experiments was long, and after I tagged out of the pair, they didn't finish that particular test. (It's not really a matter of knowing what to do: write your intentions as tasks in the tracker story. It's knowing how to tell which things are important to write down, an acquired skill pairs already have around implementation and refactor stories but may lack around test execution.)
- Director-level support and a culture that sees testers as high-status specialists have been very important.
- Pairing testers with developers on feature stories worked better than I expected it would, and there is a set of pattern-and-theory type interactional expertise that can make non-programming testers surprisingly effective pairs.
- Our reliance on the words "explore" and "exploratory" to mitigate the overloading of the word "test" in our environment has had benefits and drawbacks, and I am considering following RST's lead and deprecating the term "exploratory testing."
- Testers are encouraged to adopt the identity of an expert – and to adopt the humility toward that expertise necessary when one is merely a local, specialized expert in a highly competent technical team. Both are excellent for personal and professional growth.
- Frequent rotation between product teams with interdependencies, plus the practice of "dog-fooding," plus tester specialization in rapid learning and interactional expertise can very rapidly render testers experts in the operation of the whole system, who are assumed to have a good chance of being able to answer esoteric questions about failure conditions and complicated behavior rules, which helps maintain expert status.