Software Architecture Patterns for Deployability

Featuring Rick Kazman as Interviewed by Suzanne Miller

Welcome to the SEI Podcast Series, a production of the Carnegie Mellon University Software Engineering Institute. The SEI is a federally funded research and development center sponsored by the U.S. Department of Defense. A transcript of today's podcast is posted on the SEI website at sei.cmu.edu/podcasts.

Suzanne Miller: Welcome to the SEI podcast series. My name is Suzanne Miller, and I'm a principal investigator in the SEI Software Solutions Division. Today, I am joined by my friend and colleague Rick Kazman, a visiting scientist in the SEI Software Solutions Division. We're here today to discuss two software architecture patterns for the category of deployability, and we'll explain what that is. These patterns were recently explored in the fourth edition of *Software Architecture in Practice*, which Rick co-authored with Len Bass and Paul Clements.

Welcome, Rick. It's good to talk to you again.

Rick Kazman: Good to see you again, Suzie.

Suzanne: So, let's start by having you tell our audience a little bit about yourself, why you came to the SEI, and the work that you do here. And in particular, you're a visiting scientist, [which] is a little different role, so maybe explain a little bit about that, and what other things you do besides work with us.

Rick: Sure, so I'm a professor at the University of Hawaii. I have been with the SEI and wearing various hats for over 20 years now, originally as a member of the technical staff, and then since 2000, as a visiting scientist where I have been involved in a whole bunch of different projects, worked on a bunch of books in the SEI series with various co-authors. Most of my work centers around architecture analysis, architecture design, requirements gathering, and tooling for all of the above, and empirical studies on all of the above.

I think I mentioned this in the last podcast, but most of our books have the word *practice* somewhere in the title. We take that really seriously, that we don't want to be preaching anything in our books, in our blog posts, in our podcasts that we have not road-tested and practiced, that we have not worked with real practicing architects on real-world systems. I feel comfortable that when I preach something, I have got some reasonable empirical foundation behind it.

Suzanne: That is one of the things that I think people appreciate about the work of the SEI in general, is that to the extent possible, we want to prove things in practice, at least in an initial way. I know we can't always do the classic controlled experiments, which is why it is in practice and not in the lab. I think our audience understands that and if not, sometime we will have a discussion about the difference between different kinds of empirical studies in classical hypothesis-based. Wouldn't that be fun?

Rick: One of my favorite sayings is, *In theory, there is no difference between theory and practice. In practice, there is*.

Suzanne: OK.

Rick: You got that?

Suzanne: I like that. OK, you are making me spin around. All right, so let's move on to talking about architecture patterns and deployability. First, how about just explain a little bit about what do we mean when we say an architecture pattern, because not everybody understands that concept?

Rick: Sure, so the whole field of patterns has been around for about three decades, depending on who you want to trace it to, but most people would go back at least as far as the 1994 book, by the Gang of Four, on design patterns. Shortly thereafter, people, starting in around 2000, people started publishing books of architectural patterns. There are many, many of these books of architectural patterns. A pattern is simply a collection of architectural elements, some kind of components, some kind of connections between them, organized in some sort of topology with some constraints on that organization. If you think of the layers pattern, the most common pattern ever, you have got these components that are layers, you have got these connections between them, some sort of calling or allowed-to-use relationship, and you have got some constraints on topology, like some versions of layering. Say, you can't have cycles between layers, if layer one calls layer two, layer two can't call layer one. Or you can't skip a layer; you can't go directly from layer one to layer three. Then, typically the way these are documented, there is a context in which you would apply a particular pattern. There are pros and cons, the strengths and weaknesses of the patterns, and the tradeoffs of the patterns. Together, this represents a package of time-tested design decisions with proven outcomes, things that they are

good for, things that they are not so good for, the context in which you should or should not use it. So it is this nice, prepackaged wisdom so that you, the architect, don't have to reinvent this wheel that's been reinvented a thousand or 100,000 times over, for decades.

Suzanne: This is important. I think, software engineering is a newer discipline than, say, civil engineering. The patterns movement, I see as an analog to some of the standard practices that you see in guidebooks for bridge building or things like that, that were built hundreds of years ago. Those patterns are sometimes hundreds and thousands of years old, but I see it as an evolution of our discipline that we actually are capturing that same kind of wisdom. We don't have to reinvent from theory every single thing, because we have this experience base that has been tested in contexts that we can be explicit about. That to me is one of the great values of patterns is, it is giving us some of that engineering maturity that has been present in other disciplines for much longer.

Rick: Yes, and patterns are not a guarantee of success. A pattern can be undermined by poor realization or by allowing the pattern to erode through inadequate maintenance. A very large and important bridge near the SEI in Pittsburgh just collapsed in March, I think it was.

Suzanne: Yes, Fern Hollow Bridge.

Rick: Yes, and that was an example of a reasonable pattern, a reasonable engineering decision that was undermined by poor maintenance over decades.

Suzanne: That's a good point.

Rick: So a pattern doesn't guarantee success. It is a precursor to or an enabler of success.

Suzanne: Yes. Excellent. OK. So architecture patterns, what is your definition of deployability? Before we got on camera, we were talking about different contexts of deployment, and it's important for people to understand what is the context of deployability with this set of patterns.

Rick: I am restricting this discussion to deployment of software. Here we are talking about deploying of software onto an environment in which that software can be executed. That could be a cloud environment, that could be onto your laptop, that could be onto a peer-to-peer network or a set of edge devices, whatever your platform is for deployment, with a predictable and acceptable amount of time, effort, and risk. So it could be that every time you deploy, it's handcrafted, and you hand-carry your fragile little software component, and you lovingly place it in its environment. While that might work for small and infrequent deployments, low-risk deployments, it is probably not a very strong engineering approach to deployability. We tend to want to automate and regularize deployability as much as possible so that it is predictable, and it

is cost-controlled. It's efficient. It's testable, and so forth. We treat deployment scripts just like any other piece of software, in fact.

Suzanne: Right, and that is an important point. I think as we get into the era of DevSecOps, containerization, etcetera, etcetera. Those aspects, deployability where in the past, I might have deployed to a server once a month. I may now be deploying into this or that container environment once a day, and the effort required once a month might be acceptable to be handcrafted or hand-scripted each time, but not when we are doing once a day. And the error-proneness of those handcrafted scripts versus the regular scripts becomes an issue. Those two things alone are going to make deployability something that should be considered by people that are using modern environments and modern methods, right?

Rick: Yes.

Suzanne: So let's talk about the patterns that you found are useful for improving deployability.

Rick: OK, yes, but before I do that, let me just add one thing to what you just said. We talk about three aspects of deployability, and these patterns are going to address these three aspects. One is granularity. What is the granularity of the thing that you are deploying? Is it monolith? We have got to deploy the whole thing, all or nothing? Can we deploy little bits and pieces of it? *We just want to change the authentication part of the system. We just want to change the edge-detection algorithm* to some little, tiny piece of the whole, so the granularity of deployment. How controllable that is, so how easy is it to roll things out and roll them back if we decide that maybe it passed all the tests in the test environment, but once we got into production, things have gone off the rails? And how efficient, which gets back to what we were just talking about. What is the level of effort required to do that? If you are Netflix, and you are rolling out many deployments per day, or you are Google and you are doing many deployments a day, then you want that to be extremely efficient and as automated as possible. So keep those three characteristics in mind when we talk about the patterns.

Suzanne: All right.

Rick: The first one is not really a deployability pattern, but just a way of thinking about granularity, which is microservice architecture. The more that you granularize your architecture, the more that you make each service its own little independent thing, managing its own data and talking through some well-defined interfaces or perhaps through a broker of some sort, the easier it is to deploy that little piece, that granule of the system. That is orthogonal to how you are actually doing the deployment. I just make that as a sort of side comment to say, the more granular your architecture, like microservices, the easier it makes some things. It is not a panacea. Many organizations are actually pulling back from microservice architectures these

days, because they realize that, *Well, we are always deploying microservices 1 through 73 together, and so why are they even separate microservices?* They are paying all the execution overhead and packaging overhead for no gain.

Suzanne: Right, right.

Rick: In fact, they are giving something up. So just want to say that granularity isn't a, *More granular-is-always-better kind of lever that you could always push*.

Suzanne: The fact that we are able to measure that is important, that the granularity that we are looking at that. I mean, the people that notice that 1 to 73 are always deployed at the same time, those are people that are actually using the attributes of their architecture to measure its performance. That is also…If you start out one way, it doesn't mean you have to stay there, right?

Rick: Yes.

Suzanne: And that's the point there.

Rick: There was actually an article in IEEE Software a few months ago called The Monolith Strikes Back. We can put that in the notes of this blog. I can give you the citation for it, but it was exactly on this topic, how monoliths are actually starting to have a little bit of a renaissance these days as people pull back from their unbridled enthusiasm for microservice architectures.

Suzanne: OK. All right.

Rick: Let's talk about deployment patterns. One category of deployment patterns is patterns for complete replacement of services. So I have *n* services, and I want to replace those with *n* new versions of those services. One pattern for that is called blue/green. The idea of the blue/green pattern is that I would have two deployment environments, two environments that are as identical to each other as I could possibly make them. I have the green, which is the running version. Green light means go, and I have the blue, which is where I deploy my new latest, greatest versions of these services. Then what I can do is, I can do a cut over to the blue. I can monitor them. Once it is determined that the new instances are working properly, up to spec, then I can remove the old versions of the services, or I simply rename blue to green, and I rename green to blue, and I start over with the next deployment. The nice thing about that is that there is ideally just a single switch, right? We point to blue. *Ooh, things are going off the rails. Boop, back to green before too much damage has been done*.

Suzanne: This pattern implies a lot of measurement going under the covers because for me to know that I need to go from blue back to green, I have to know that it's going off the rails. I need

to have baseline measures, and I have got to be able to compare my blue performance to my green baseline.

Rick: Yes, and so that means you need to think about monitorability.

Suzanne: Ah! Another quality attribute.

Rick: Another quality attribute when you are architecting that system is, *How am I going to monitor for system resources?* Like, *How much memory is this using? How close am I to reaching the maximum CPU limit? How much am I using my buffers or my bandwidth or whatever, database accesses*, whatever I care about measuring? But you may also have application-specific measures like, *How many customers abandon their shopping cart without checking out? We don't know why. We just know that when we switched over to the new version, we're getting a lot more, you know, a lot higher bounce rate than we did before*. So it could be, as I say, the system-specific measures, or they could be application-specific measures.

Suzanne: OK. All right. So that pattern, what is the downside to that pattern? Every pattern has got a context it works well in. Where would you not want to use a blue/green pattern?

Rick: Absolutely. So of course it is expensive to maintain two identical versions of your environment, especially if your environment is big and costly. This might, in fact, siphon resources away from your testing infrastructure. If you only have so many machines, so many instances, you may be making a tradeoff there between, *how much infrastructure I can put in my blue/green* versus *how much I have in my test environment*. It means that you really have to pay careful attention to these environments to ensure that they are in fact truly identical or as identical as you can make them. That is trivial in some environments, but almost impossible in other environments where they have really extensive and messy deployment infrastructures. When I say messy, like, you have connections to 130 different partners. So how do you manage all those 130 connections in both environments?

Suzanne: And identical connections in both environments.

Rick: Right.

Suzanne: And you have got to get your partners to agree to having two connections always in play and things like that, so there are business aspects to this that could come into play.

Rick: We worked years ago with an energy broker that was operating in the Midwest, and they had all these connections to all these different utilities, power companies. They just said, *We do not have a test environment that replicates our production environment, impossible for us.* So, in some domains, easy. In some domains, this just may not be possible.

Suzanne: Am I correct in thinking that this is one of the areas where a cloud environment is advantageous? Because I do have more elasticity of resources and the ability to containerize and, have more environment parity among multiple…

Rick: Yes. Then it's only about money.

Suzanne: Yes. Well, ah, it's just money! It's just money, right? OK, all right, so blue/green. I want to hit risk just a minute, because you talked about a little bit about business risk, but is this a deployment pattern that I would need to be careful about if I was like, say, in a healthcare environment where I was in a high risk if this fails?

Rick: Yes, absolutely.

Suzanne: Then blue/green may be a little not my choice.

Rick: There are a number of patterns that then can bring that risk down. So one is called rolling upgrade. That is the idea that rather than replacing all instances of your service or services at once, you replace some slice. You may choose, like, to replace only one instance at a time, or you may choose a few instances, but in each case, it would typically be a small fraction of the instances that you would replace at any one time. And it is rolling. So you are rolling out service after service after service until, at some point, all versions of the old service have been replaced. And again, you need to monitor those to ensure that the new instances are behaving as expected, but that does lower the risk.

Suzanne: OK, good. All right, because I know there are people out there that think I can possibly think about this. All right, so second pattern.

Rick: OK, so kind of building on this idea of rolling upgrade, but taking a different dimension, is what is called canary testing. The idea there is the canary in the coal mine, is the metaphor that this is borrowing from. Your canary, in this case is going to be some subset of your target market. Maybe you are going to roll it out to just your internal users first, or you have a dedicated group of beta testers or something like this. So they are your canaries. If they go *urk*, then you know that this was a bad rollout, and you haven't affected any of your actual, real, high-value…

Suzanne: Larger, yes.

Rick: So these could be, as I say, purely low-risk users, like internal users, but you might also cultivate groups of users who exercise different paths, different kinds of functionality in different ways. So you can be quite strategic about this before you decide to go live with the release. The canary testing can work in conjunction with the blue/green or the rolling upgrade.

Suzanne: So, again, when would I not want to use canary testing or canary deployment?

Rick: Well, certainly, all of the tradeoffs in the blue/green pattern apply here, right? You are still going to have to have identical or as near as possible identical infrastructures, but now you are going to have additional complexity for managing configurations.

Suzanne: OK.

Rick: Who gets routed to which version of what service at what time? Then maybe they have to be rolled back, so all of that really needs to be carefully orchestrated. If not, then you might miss important signals about what has gone right or what has gone wrong.

Suzanne: It sounds like in this pattern, I actually have to do more of a, I would call it an experimental design, in terms of what parameters are—is each canary population seeing. Does this group give me enough confidence that I am going to go with a higher volume rollout, or do I need to have rolling canaries?

Rick: Right, exactly.

Suzanne: It is OK, so now I can go to this larger group.

Rick: I have a group of power users that exercise some rarely used paths in the system, or I have another group that is really going to hammer on the database because they are loading and retrieving huge image files or whatever it makes sense for your domain. You might have different kinds of canaries for different system conditions that you want to probe.

Suzanne: No, I get it. I am a canary on for a couple of sort of odd pieces of software that I like. I have specific aspects of the software that I care about, and I don't want them to screw those up. I volunteer anytime they ask for somebody to do that. It sounds like the canary deployment pattern is…Am I correct in that it is probably one that is for user-centered, user-facing kinds of things, or do you use a canary pattern when it is more machine-to-machine?

Rick: In fact when it is more user functionality, that is a special variant of this pattern that is called A/B testing. So canary testing is often about the functionality of the system. A/B testing you may be…Marketers are really keen on using A/B testing, because they want to test sometimes tiny differences in the user experience or the user interface.

Suzanne: Human preferences.

Rick: Yes. So, the anecdote, I don't know if it's true, but they said that Google did A/B testing on 41 different shades of blue…

Suzanne: Wow.

Rick: …on its search results before they arrived at the one that they finally settled on. And you think like, *Oh, blue. Pick one*. Nope, they did the systematic testing. So things like, *What color do I make the Proceed and the Cancel buttons?* Things like that, that you don't even think about, somebody has to make those design decisions, and they will often use A/B testing with a set of handpicked users or just with a very tiny subset of the users, real users to determine…

Suzanne: Yes, I can see with that particular one, using…There are a lot of variations in colorblindness. I have a friend who, for example, everything looks to him from some gradation of pink to brown. It is a very unusual one, but it is out there. When I would do slides, when he worked with us, I would always send them to him and say, *Is there anything that looks totally weird here*? Because I can't see that spectrum the way he does, but I could do tonal differences and things like that that would make it more readable for him. So there are things like that that if you don't have an awareness of some of the variance in human perception, you are just not going to hit it. I get the 41 shades of blue.

Rick: Yes. Those are the major patterns that I wanted to talk about today. I think that the point of the blog post, and the point of the tactics, the patterns, and the tactics that we have provided, is to raise awareness of this as an area of software engineering, of software architecture that people need to and people are increasingly becoming aware of and paying attention to. Because more and more of our software is being deployed into complex environments into cloud environments and edge environments, onto microservices, onto IoT. So deployment is, well, as I said, you have to treat your deployment scripts as software.

Suzanne: Right. It can't be an afterthought. I think the real message is, *This cannot be an afterthought, or else it's going to be problematic*. A good piece of software may never get to its intended use because it's not deployable on a regular basis.

Rick: Yes, if you messed up and, and you deploy the wrong version and there is a security hole, or there is a performance risk, you would start crashing left and right.

Suzanne: And you can't bring it back.

Rick: Right. If rolling back to the old version is costly or risky or damaging to your reputation, these things can have huge consequences.

Suzanne: Yes.

Rick: Yes, that was why we wrote the technical report where these patterns were are included, the blog post, and again, wrote about it in the book.

Suzanne: Right. That is why they made it into the book. Only the only the really good patterns make it into the book, right?

Rick: That is right. The rest are left on the cutting room floor.

Suzanne: There you go. Rick, as always, it is a pleasure to talk with you and to explore some of these ideas that we may take for granted, or else we may not be thinking about, and we should. I hope our audience appreciates that as much as I do. We did mention several things during the talk, and we will make sure that those links are included in the transcript. I do want to remind our audience that this podcast will be available wherever you get your podcasts. Our favorite is the SEI YouTube channel, but you can get it wherever you like. I do want to thank all of you for joining us today. Go forth, and make deployable software.

Rick: All right, always a pleasure to talk to you, Suzie.

Suzanne: Thanks.

Thanks for joining us. This episode is available where you download podcasts, including SoundCloud, Stitcher, TuneIn Radio, Google Podcasts, and Apple Podcasts. It is also available on the SEI website at sei.cmu.edu/podcasts and the SEI's YouTube channel. This copyrighted work is made available through the Software Engineering Institute, a federally funded research and development center sponsored by the U.S. Department of Defense. For more information about the SEI and this work, please visit www.sei.cmu.edu. As always, if you have any questions, please don't hesitate to email us at info@sei.cmu.edu. Thank you.