Google Prodcast Season Three Episode Thirteen

[JAVI BELTRAN, "TELEBOT"]

STEVE MCGHEE: Welcome to season three of The Prodcast, Google's podcast about site reliability engineering and production software. I'm your host, Steve McGhee. This season, we're going to focus on designing and building software in SRE. Our guests come from a variety of roles, both inside and outside of Google. Happy listening, and remember, hope is not a strategy.

—

STEVE MCGHEE: Welcome back to the broadcast, Google's podcast on SRE and production software. I'm your host, Steve McGhee, and I'm joined again by Jordan Greenberg. Hi, Jordan.

JORDAN GREENBERG: Hi.

STEVE MCGHEE: Hi. This season is all about software engineering and SRE. And today's episode is about something maybe we don't always think about as traditional software engineering, but it is dear to SRE's hearts, configuration.

You might have heard about imperative versus declarative configuration and config as code. But I like to say config is code. And today, we'll dig into the ramifications of that. OK, let's bring in our guests. Today, we have Dominic and Nicc. Guys, please introduce yourselves.

DOMINIC: Sure. I'm Dominic. I'm a Staff SRE at HashiCorp. Before that, I spent most of my time in smaller early stage engineering organizations, where I've been lucky to work on a mix of stuff, like from satellites and IoT through to SaaS in the finance and HR space. I don't know. They were foolish enough to give me a chance, and I had a bunch of fun and opportunity to learn. So that's me. Nicc?

NICCOLO': Hey, thanks. I'm Nicc. I have been a Google SRE since the very beginning of my career. I just came to Google straight off my university. I started in Google in 2011, and I've been working in the last 10 years for the internal continuous delivery system, which is based on a declarative approach, which is going to be part of the topic that we talk about today.

JORDAN GREENBERG: Nice. Another person who started at Google as their first job out of college. We are kindred spirits.

STEVE MCGHEE: Me too, kind of. I had a job in a different college. Does that count? I don't think that counts. I think it was--

JORDAN GREENBERG: I don't think that counts.

STEVE MCGHEE: No? OK.

JORDAN GREENBERG: I think you're good. We're twins.

DOMINIC: You've been ousted. You're not part of the club, unfortunately. I'm sorry.

JORDAN GREENBERG: You know what? You are now. You made it. You're here. Glad to have you.

STEVE MCGHEE: There are many clubs, it turns out.

JORDAN GREENBERG: Yes. So I would like to know, as an SRE TPM, what does declarative and imperative mean in this context of SREism? Did you make these words up? Were they adapted from something else? Math is different than SRE apparently. So what does it mean in this context?

DOMINIC: I don't think we made them up. Pretty sure we took them from a bunch of other domains. All stuff's derivative, right?

JORDAN GREENBERG: Yes. Absolutely.

DOMINIC: Obviously, computer science and programming.

JORDAN GREENBERG: Yeah.

STEVE MCGHEE: All words are made up, it turns out.

JORDAN GREENBERG: So what are the terms declarative and imperative in this context? They do come from other places like math and whatever. And I guess math is SRE. But in the SRE context, what does declarative and imperative mean in this way?

DOMINIC: We definitely stole them from somewhere, that's for sure. I don't know. I guess I have some formal study background, like I work a lot on Vibe and things I've learned over the years. To me, the main difference is that declarative normally involves like there's a DSL that describes something. And you give that DSL to something, which decides how to arrive at the point you described.

STEVE MCGHEE: That's a domain-specific language, right? The DSL.

DOMINIC: Yeah. Sorry. I'm bad at the acronyms thing as well. I'll try to cut back on that. And then imperative is like most people will be used to in like traditional programming. It's like you write some instructions, and they work through top to bottom. Maybe there's some control flow and like loops that do this thing 10 times. But yeah, that's very broad brush strokes. But the practicalities, when it actually comes to config management, there's some subtle differences. I don't know. Nicc, what do you reckon?

NICCOLO': Yeah, I agree with you that the things that I explain, I used to explain what declarative and imperative really means is I always say if you are giving me a piece of code, a script, a snippet of Python, bash script that runs specific instructions one after the other, that to me is imperative. It can be a very complicated script that is embedded into a larger system like Spinnaker, which by itself has a declarative-based API because you declare your workflow in Spinnaker through the Cloud API. But the end result is an imperative workflow that executes their operations.

If you are telling me instead that here is the data, a piece of data, which is normally a protocol buffer, a JSON file, or literally data, no program, and that defines the intended state that you want to reach, whatever the intended state represents, then to me, that is declarative system. So as you said correctly, you must have a domain-specific language which abstracts the right atom for you to operate in order to perform a declarative-based operation.

DOMINIC: I guess there's another part I'd add into that, which is like something I commonly see in declarative, change control systems, is like the concept of modules or things you can import that other people have authored. And you can use them in your declarative stuff. And you kind of compose it all together to achieve the end result you wanted.

Whereas like, yeah, you do do that in scripts. Well, sorry, I don't want to reduce imperative to scripts straight away. But in the imperative sort of change control world, you do do that. It just doesn't seem to be such a focus on it, which we might touch on a bit later.

NICCOLO': Yeah.

JORDAN GREENBERG: OK. So summing up, an imperative system, you have to produce the code for and in a declarative system, you produce the data for. Can an imperative system have a declarative interface?

NICCOLO': Yes. As I said before, Spinnaker, it's the example that I brought before, which is itself, Spinnaker is a workflow engine, which means that you're going to write a workflow, which is just a script on steroids, if you want, which does many more things, and it's much more elaborated.

And you produce the scripts by using declarative APIs like create, update. So you are going to create a node, you're going to create a step. You're going to create an action, whatever are the atoms that the Spinnaker API proposes. But in the end, your result is a workflow, which is an imperative script. That's how I would describe it.

If I want to give you a contrast, then I'll also let Dominic tell his own opinion on this. Kubernetes instead is declarative because if you look at a YAML file in Kubernetes, you say, I want to have this deployment with this image and this number of replicas. You don't tell it what to do. You just give to it. This is the intended state. Please reach it based on your own internal logic on how to enforce the intended state.

JORDAN GREENBERG: When would you use an imperative versus a declarative?

DOMINIC: I don't know about Nicc, but for me, speed is like something that comes in quickly. I can knock together a crappy script pretty quickly. But then I need to put on my engineering hat and be like, what is the longevity I expect out of this thing? Like do I expect it to be around for a couple of years? Like who's going to maintain it? Am I going to maintain it? Am I going to have to bring other people up to speed on it? Am I going to be able to satisfy the constraints that I need to, like security, or cost control, or whatever it is. Once it gets, I'm going to say more advanced, I start to lean towards declarative approaches definitely, like imperatives for quick stuff, quick and dirty in my books. But that's just my opinion.

NICCOLO': Well, I share your opinion, to be very honest, because I was going to answer something along the same lines. So the advantages of imperative is that normally programmers are very used to writing imperative languages. So you execute the order. You execute the instruction in order and you know what is going to happen.

Then you introduce multi-threading, and then it's where things start to become interesting. And it's the same for imperative because think about it. You start small. You start with your script and it becomes a Python program. Then this Python program that's responsible, maybe to push the version of your Docker images or whatever, is responsible for you need to push, not only to cooperate with another script that your colleague has written, that is the one that adjusts the dimensions of your replica sets

or whatever capacity-management related things.

And now you have two scripts that both contribute to the intended state of the same resource. And they need to cooperate because they do not share the intended state, the final intended state, because they just know a piece of it. So when you start to go in that direction. And then add more processes, and it becomes a quadratic problem because every other script needs to know the logic of everything else in order to synchronize, or you do it naively and with the mental synchronization, and then it becomes an interesting debugging exercise.

Or you put everything under a declarative hood, which says the intended state is this and different components, different processes produce the specific part of the intent that they are responsible for, one for the version, one for the capacity, one for the other things, the experiment, flags, or whatever. They get combined into an intended state, which is then rolled out by the same system which takes care of rolling out the intended state across the fleet, following policies, following your predefined rules that you put as part of your configuration for, in this case, a continuous delivery system that is based on declarative engine.

JORDAN GREENBERG: OK. So it's like the system is coming up with the imperative instructions to achieve declared intent.

NICCOLO': Yes. The process described is for that can be written in any form you want. But in the end, they produce a part of the intended state, which is data. So it's like they are all contributing to the same piece of data, which is the intended state of the overall fleet, the overall system, or whatever they are controlling through the declarative engine.

JORDAN GREENBERG: Nice.

DOMINIC: In addition to the synchronization issues you touched on, something else that I want to bring up is like at least when you're in smaller orgs and you don't have a bunch of resources, you write this script, and it begins small. And then you need to add some things, and you add some other things. And all of a sudden, the script is big and depended on by many people and many things.

And then one day, like change is risk, right? You need to change the script to do a thing, and it breaks. And the blast radius of that script has grown as the length of it has grown as well. I find with the declarative systems, you can decompose and pull stuff apart into smaller chunks, so that you sort of have like a blast radius or like a failure domain around like, oh, this is the module that controls the

database, or this is the module that controls, I wouldn't even go that big, like x step in setting the database up, as opposed to like, oh, I changed the script, and we have a half deployed set of infrastructure, and nothing works. What do we do now? Rally the troops, I guess.

JORDAN GREENBERG: So in an emergency, if we need to make some changes really fast, are declarative workflows slower than imperative ones?

NICCOLO': I don't think the engine per se is slower. It depends how you configure it. So if you need to do something ad hoc because your specific emergency requires you to do one off data migration. If your declarative engine is not already supporting that thing, you're not going to be able to quickly implement it in the declarative system. So probably, it's better that someone puts down some good imperative workflows, or scripts, or whatever.

Although I would say, normally, if you are in a declarative system where you can prepare your own emergency operations in advance, you are not in a rush to implement them quickly because that's the moment you make the outage worse to try to make it better. You make a very quick script that tries to go over everything, but then you type something wrong, and then you go over everything and everything else that you are not supposed to touch. And then the outage is suddenly worse. So it's always a balance to understand how quick you want to patch something. But remember, the quicker you go, the faster you make a mistake. The blast radius is bigger.

DOMINIC: Yeah, I agree with that. I think an interesting thing I think about is like the breadth of how wide the number of systems you have to touch with this modification is. I think a lot of smaller orgs, I'll use the data migration example, they need to touch one database that backs their entire production system.

And then like you get up to Google mothership scale, and it's like oh, well, globally replicated and shuttered, and there's definitely not one database, that's for sure. So we need to apply it across this wide set of infrastructure.

I think operational teams find it hard to figure out when it's worth like approaching the buy-in to this declarative thing that allows me to roll out across a fleet, versus like what I commonly see in most orgs that are smaller, which is like, oh, I'll just open a console, or I'll have someone review my script and then run it against this thing.

JORDAN GREENBERG: I know patching is really common in SRE. You're trying to do that gradual

change that SREs are so gung ho about. You don't want to change everything at once because that's tough. But a patch sometimes can make that very big change when you don't want it to. Can you tell me about a time where a patch made something go sideways?

DOMINIC: Yeah. I don't know if I want to get into the details.

JORDAN GREENBERG: We won't share this with anybody, we swear.

DOMINIC: Something that stands out to me, I guess with patching is like when you are trying to control the rollout of a patch with infrastructure as code, either declarative or imperative, I question if that's the right way to get that level of control you want, sort of contained the potential fallout.

I start to look towards progressive rollout techniques, like feature flag enablement, or like perhaps a small segment of the fleet gets that first, and you observe it, referred to as a canary by most people before you let it roll broader.

You can affect that progressive change through declarative or imperative change control flows. But like I don't see that as like infrastructure as code's job. Infrastructure as code is just what you use to achieve that gradual rollout. Like it's no silver bullet, right? Yeah. Nicc, do you want to share a spooky patch outage story?

NICCOLO': Yeah. It's a very complex and multifaceted question because the first thing I would say is that, well, it depends what kind of change it is, right. Is it- are you introducing a new feature? Are you removing a load feature? Are you writing a program to do a one off data migration from database A structure to database B structure?

So each of these is his own answer. I would say that in general, the majority of the changes gets packaged into a release of an artifact. Normally is the new compiled binary of your backend or the new set of JavaScript files that you need to roll out to production.

I think the union of microservice architecture and micropatching is the strategy that is probably the most winning if you can do it and if you can afford it. Because not all situations can be done like that. If you are uploading firmware on on-premises devices of people, including the heating system that runs in your home, those get updated sometimes. You want to be very sure that those are always working. You don't want to shut down the heating system of the people in the middle of the winter because you made a mistake.

So probably those kinds of situations, you go on a much lower development because the risk of making a mistake is bigger. So it boils down to the risk management. So what risk are you managing? Are you risking mission critical systems that if they fail, they cannot work? Think about the Rover on Mars or Voyager 1. They are going to be very careful what they push to those things. Because you can't just go and fix them.

If instead, it's a microservice architecture, which is more fluid by nature and by architecture, and you can easily redirect traffic from non-working instances because you have set up canarying in the proper way, and you of course, you're doing the right AB, blue green approach to do comparison and doing regression analysis. The risk is much lower. So it depends a lot on what you are running to know if you need to deal with big patches or not.

DOMINIC: I see patches as definitely something different to a code deploy, like a release of a new software version of software that we ship. That patch doesn't conjure that image up in my head. Like a patch to me is like, I don't know, an OS system dependency on the VMs that you like pull together to run your scheduler or what have you. And you increase it because you're like, yeah, faster JSON parsing. And then you're like, no, everything is on fire and nothing can talk to anything all of a sudden. I don't know what we did. Yeah, I guess I definitely answered for that.

NICCOLO': Cool. Yeah, the first question I would ask in that situation would be what process triggered that? Was it a global sudden change? Because most of the time, you realize that you have a system that is global unknowingly. And you may all rely on a specific file in a specific distributed file system, just to coordinate the lock.

And that one file is a global file. And if you change the name of the file because you are doing a migration, despite your system being well distributed and released in a gradual fashion and everything, then you have this single point of failure that the moment you touch it, everything falls apart.

So this is always very good. Now we're going probably more in a postmortem kind of culture and understanding what is going on. But that's exactly where you analyze what happened in the architecture to remove this point of failure when possible. Or you protect them with additional mechanisms that would prevent the next time a bad catastrophic failure to happen, probably.

DOMINIC: I think I can tie it back to declarative pretty well. An exercise I'd recommend any team does is like think about your failure domains. And like maybe go through an exercise like fault tree analysis

for your systems. Personally, as an engineer, like I find that decomposing stuff into declarative modules that I like pull in, and I manage, that helps me reason about these boundaries of failure and like break things up into components, and these things that interact with one another.

It's not like just doing that. It's not analyzing your failure domains, but it really helps, and it helps also get the team on the same page about what to refer to the bits and pieces as and how do they interact? Yeah.

STEVE MCGHEE: Totally. One kind of issue that arises sometimes with teams that maybe have been around for a while, or let's just call them enterprises. Like large companies that have lots of people who have been doing different jobs for a while. Often we'll find, at least when I talk with customers, we'll find that there's a cultural like issue at play here when it comes to the team who writes the code and the team who manages the configuration. And often, we'll just call this the dev team and the ops team just for shorthand, but it can be lots of stuff.

And so I think that one thing that we can we're kind of getting at here is the ability for the team who wrote the code and kind of owns the product and the complexity of the system, to be able to also then be the ones who tinker with the config and push the config or change the config or whatever. Because traditionally, it was not that. There was this set of config files that this other team just kind of either blindly or by following playbooks, would flip switches. Maybe not considering failure domains and things like that.

So I'm eventually getting to a question here. Like A, do you think this is a true story, or is Steve just making this up? But B, like is there something that can be done here for the industry and like within platform teams, within enterprises, where you build up a system that allows teams to manage their own fate? To not have to pass the ball over the wall, as we say. Where we can say like you can actually manage your own configuration through this new thing, and it has all these features and has these control structures.

I think the trick here is alluding to all the problems we just talked through and all the options that we just talked through, like it's a big hairy beast. And so making people who are developing software choose through all that is yet another burden. And this is why we see platform engineering teams arise and saying like, here's how we configure the stuff. Like use this from now on. That was a long question. Like what do you guys think of this worldview? Am I way off base, or is this a real thing?

DOMINIC: I'm on board with it. I think there'll be a couple of answers to this question. We might go back and forth over it a bit. When I work with the notion of platform engineering, that's what everyone's trying to do when they're working with infrastructure as code, I think, whether they recognize it or not.

The thing I circle around a lot is like what abstraction are you exposing to your consumers? Whether it's like a module that they can import, or I know I've seen like GitHub pull request-based workflows, where it's like open PR and this repo. And you'll get a bunch of stuff spun up, and you put the YAML block in the PR, and it configures what you get.

Like whatever the interface is, the abstraction, like not the paradigm of like declarative versus imperative, but the abstraction that you expose is really where the meat of the problem is. Something that I struggle with like in the smaller orgs is the needs of the business evolve really quickly. So like you spend this time to build this awesome abstraction for the infrastructure needs of today.

And by the time you ship it, they're like, no, we don't need that anymore. We do thing x now, and you're like, damn. I guess we'll chase that dream. So it doesn't solve the problem, but it does help. It gives you, especially in places where no product market fit and what you're shipping is more well figured out and stable, static, I think it's more attainable.

But I would caution against investing in it too early because it's maybe a waste of effort at the very beginning. I'm saying like early stage business, it's like maybe not the best idea to pursue. I don't know. Nicc?

NICCOLO': Yeah. That makes sense. In the end, on one side, you must have the right incentives in the right moment in order to create the culture necessary to go from very tactical reactions and follow the things very quickly to a more strategic, long term maintainable system.

And different products go and switch to the more mature phase at different points in time. And this also means that the company will end up having a system that is very quick in development, that it doesn't have time to look on the long term things, and they'd rather do it quickly. Plus there is the long term that is not fully fleshed out. So you're always in a little bit in this sea of uncertainty of what's the right thing to do.

But I have to say, in general, if you are a company that has already developed a set of policies around gradual rollouts or continuous delivery and deployment, which are mature because you already have a

set of products that are mature, and they need to be reliable because now, people rely on them.

The new products should be shifted toward looking into that direction quickly, which means that this long term durable thing should, over time, lower the bar, to make greenfields more accessible and directly jump into it right away. It's not always going to be possible, but that's the kind of transformation that you need to do in order to make it sustainable.

You make it good for the hard thing first or the most important thing first. And then you gradually expand the scope, of course, if there are the possibilities. Once you are in this situation, any type of change can then become part of the same rollout system. So it's not becoming, oh, I just do version releases with the full fledged thing, and then all the config releases remain forgotten and done by hand by the poor chap that knows the things in and out and knows how to do the right ordering.

So once you have the system that can declaratively push the things to production following a policy, it should be normally easy to adopt it to also perform other types of changes following the same logic. Because you give an example. You deploy your firewall backend across all your cloud regions or your data centers.

You release the binary following a certain qualification process, which is probably going to be regionalized and take care of the traffic pattern and all of that. And then you also have the firewall rules. The firewall rules also are deployed together with the rest of your production. And probably, they follow very similar locations in logic as well as the binary.

So why don't you use the same system or technology or at least logic that you use to push the versions across the entire fleet of binary of your firewall to also push the config rules for your firewalls? It's not easily done because people start to think about the firewalls is a global rule. I need to always have the same global rule everywhere.

But if you start to break it down, you realize that it's not really true that it's global because the firewall itself is already distributed. So it's not going to be atomically using the firewall rule everywhere at the same time unless it's engineered on that purpose. But if it's an engineering that purpose, it means that you have an actual global point of failure, which is the configuration. So again, it depends what is the risk that you're managing, going back to what I was answering before.

DOMINIC: That resonates with me, something that jumps out in my head. My head does this. Sometimes, it just bubbles up a thought. Sequencing is the magical word. And I think every team

should talk about sequencing when they look at how these different components roll out.

The obvious one is like, is it code then infra, infra as code, or is it infra as code, then code? Do they go together? But then I think an often overlooked one is where does the database fit into that? Like database migration, is it database migration, then code, then infra-- like you should all have the same understanding of what's going out when. Because if you design your change with a different understanding of when that's going to get affected in mind, that's a recipe for like unforeseen circumstance or consequence.

I think that's kind of what you're touching on, right, Nicc? It's like what system affects the change isn't so much it. But I really hope that people understand it consistently because I've been in a lot of incidents where it's like, ah, yeah, we didn't know that would happen when this hit. We thought this would land before that landed. And that's why x interacted with y, and we got potatoes. And I'm like, OK, well.

NICCOLO': If I may follow up on that. You also touched one thing that is one of the most expensive thing that a team needs to do in order to go really declarative. So if you buy the declarative approach, the first thing you notice is that if you're lucky, half of your system is modeled. The rest is maybe written in some document, and then 10% of it is just in people's heads.

So the very first problem that you're solving is you realize that you do not even have the full specification of your system in place. So the very first thing is to admit that, recognize the problem, and then start to insert into the system model of your declarative continuous delivery system all the dependency of your system in order to be able to manage it.

DOMINIC: I think the thing that really gets to me is like once you have this declarative infrastructure as code, I don't need docs. I read the infrastructure as code to understand what's deployed where. When a new person joins the team, I'm like, hey, read this stuff. And they're like, ah, cool. I understand all the bits and pieces.

Where I'm like, hey, welcome to the team. Please read these 400 lines of bash that talk to various GCP APIs. They're like, yeah, I read it. I'm like, no, you didn't. You definitely didn't. But that's OK. We can learn as we go along. I've definitely seen that happen, like in orgs I've been in the past.

NICCOLO': So what would be nice, so when you have configured your system model properly, since the machine is following the system model in order to do the right order of operations, it should also

be able to render the graph. And all of a sudden, instead of the TL of the project coming and doing the usual thing that all TLs do, draw boxes and arrows on a whiteboard that says, well, the machine thinks that our architecture is like this. Actually, the model dictates that the architecture looks like this. And then you get the pregenerated understanding of the model.

Once you have the model in place, it means that your version release qualification, your capacity management, your experiment control system, they are all contributing to the intended state, and that is the relationship between the various components. Because all of a sudden, if you want to release the binary version of your backend that talks to your database, you can start to connect things together, and it's no longer given to the human the task to make sure that when you update the version of the binary, make sure that is never going to be to version above the version of the database schema, whatever is the policy that you must have.

Instead, all the system's descriptions is part of the model, and the model allows you to implement policies and invariants, which will prevent accidental outages just because the new person didn't know that rule that was just documented in the people's brain.

DOMINIC: Can I jump one more?

NICCOLO': Yeah, please.

DOMINIC: That prompted something for me. It's like I'm reminded of the word hermetic or hermeticism. It's like everything you need to run, prod or whatever code, infra, the config of the infra, the data migrations to get it set up, you contain it all in one thing, and that's like your hermetic artifact to arrive you at the state you want to be. I think NixOS and like the Nix Project is a good example of hermetic, an intriguing example of achieving hermeticism or enforcing it rather. Like it's pretty hard not to break its paradigm.

When the system gets bigger, this becomes a very useful tool for diagnostic and debugging and also reasoning about. Like what am I shipping? Oh, I'm shipping this thing that contains all the possible bits and pieces, as opposed to like I'm shipping this thing which has this dependency on this thing outside of my control, and I don't know what that's doing at this point in time.

STEVE MCGHEE: At the end of the day, I feel like we've hit a bunch of high points here, which are all inextricably interconnected in a very complex, confusing way. But essentially, like let's throw a few terms that came up today. Hermiticity is a good one. Hermetic builds is a thing that people have heard

of.

But the idea of hermetic in general is something that can be applied beyond just CI. It can go further than that. Another one that came up was sequencing. That was a good one. Making sure that you understand or can understand by reading a script or the actual code to determine what is the process by which things happen in our system.

And then I think the last thing that popped up in my head in terms of what is the outcome of all of this is really what we're trying to avoid is unintended side effects of change. And often, when we just have people out there “yoloing” and installing the new JSON optimizer, which accidentally removes all semicolons, like this is the problem is that we're not keeping track of what the change is, and we're not applying gradual change effectively. We're just expecting expert humans to be experts and never make mistakes and remember every single thing they did and predict the future. And none of that is good.

So instead, we have these other things. And whether it's imperative or declarative, some have benefits. Some have detractors. It's all just code because kind of I alluded to at the top, like config still is just code. It's just telling other code what to do or what to not do. So you really just making changes to the system.

So make sure you focus on understanding your failure domains through your entire system and look out for those unintended side effects. Thanks for the awesome discussion, both of you guys, Dominic and Nicc. That was awesome. Before we go, is there anything that you want to leave our audience with in terms of how to hear more from your voice or just like one last shout out to your favorite thing?

NICCOLO': Actually, I invite people if they're curious to know more into the details, everything that I based my conversation on is the system that we developed in the last 10 years at Google, you can find a USENIX paper that we publish in 2021 that is called Prodspec and Annealing. So if you just Google Prospect and Annealing, you should find it very easily. And I think we can add the resource to the podcast metadata, so you can find it from there.

DOMINIC: Yeah, somewhat boldly, I'm going to commit myself to do something which I haven't yet done, which is I'm working on a blog.

JORDAN GREENBERG: Oh, wow.

DOMINIC: That's one where I practice my terrible drawing. And maybe I make the concepts and principles of reliability engineering more accessible to people. It's early days, as in I haven't written anything on it yet. But maybe by the time that this goes out, there might be something on it.

JORDAN GREENBERG: Nice.

DOMINIC: If you want to check it out, it'll be available at sketchreliability.engineering.

STEVE MCGHEE: Nice.

NICCOLO': I'll make sure to check it out. Good luck.

JORDAN GREENBERG: It's in the ether now. You have to do it.

DOMINIC: Yeah. I really put myself on the spot. Now I might red line this like right before it goes out and be like, please, no, I have not written any posts.

STEVE MCGHEE: No problem. No pressure.

JORDAN GREENBERG: Yeah. Well, thank you to both of our guests, Dominic and Nicc. Thanks so much for coming here. And thanks to my co-host, Steve. I hope everybody has an awesome day and we'll see you soon.

DOMINIC: Thanks for having us.

—

[JAVI BELTRAN, "TELEBOT"]

JORDAN GREENBERG: You've been listening to Podcast, Google's podcast on site reliability engineering.

Visit us on the web at sre.google, where you can find papers, workshops, videos, and more about SRE. This season's host is Steve McGhee with contributions from Jordan Greenberg and Florian Rathgeber. The podcast is produced by Paul Guglielmino, Sunny Hsiao, and Salim Virji.

The podcast theme is “Telebot” by Javi Beltran.

Special thanks to MP English and Jenn Petoff.