pavedroad.io

Treating our Service as Code (SaC)

Why we need to challenge our thinking

I want to lay the groundwork for this piece by giving a couple of recent examples.

In my prior position as a group CTO in a large enterprise, my group received much attention as the companies "unicorn" initiative. At one point, a group was dispatched from the corporate CTO office to ensure we were adhering to company policies. One requirement was to provide a full lab build of our network, servers, storage, and software for testing to their facilities. I received a puzzled look when I ask him how they were going to get Amazon and a few other cloud providers to cooperate with that request?

Around the same timeframe, our CEO appointed a new head of engineering. His first request was for a full bottom-up set of documentation and architectural diagrams. I explained that our entire service was based on Infrastructure as Code (IaC) and wasn't static. I could write him a program that diagrammed the current state and have it pull the current build documentation, but operations could and did make changes that would make that snapshot quickly obsolete. Another puzzled look ensued.

Both of these individuals have read all the DevOps books, but they failed at making the mental shift required for placing the lessons learned into action. Further, they continued to view applications as a static definition.

- Modern applications are a dynamic living thing
- Our methodologies and tooling must adapt to that fact

I pose the question, have we done that as a DevOps community?

Best practice for developing business applications

When developing applications, we've learned to decompose them into independently deployable services, aka microservices. Functions as a Service, or serverless, allows breaking breakdown applications into individually deployable functions. Then using the Event Emitter pattern as the communication vehicle for both functions and microservices.

When evaluating DevOps practices, we've chosen to focus on software implementation and apply some essential criteria. Is it:

- A Single Responsibility Principle (SRP)?
- Loosely coupled?
- Follow the Event Emitter and Consumer patterns?

These three are foundational patterns in microservices, but many more exist.

The state of current DevOps solutions

Let me provide two more examples of why I think we have not made the mental transition like my former coworkers.

The Software-Defined Delivery Manifesto lays out a set of principles intended to make the applications driving the CI/CD pipeline equally as essential as the business applications it delivers. It rightly points out, "Scripts don't scale." This is key since most of us grew up writing scripts to automate workflow and then later transitioned to using tools like SaltStack, Chef, Puppet, and others.

However, another tenent makes you question if we have moved on. The "**Among software:** We use best-of-breed tools, but how we combine them is *unique." O*r better said, every combination is a one-off, aka a snowflake. That is not a good thing.

For example, think of the amount of code required to integrate different CI/CD tools, and when one of those tools changes, it can break the entire chain. In Mik Kersten's book, Project to Product, he uses the fact it takes an engineer six months to learn pieces of their code as justification for not moving resources inside of an organization. From a business perspective, the complexity described limits flexibility and resource allocation.

If, on the other hand, those integration points had well-defined interfaces, aka Bounded Contexts, integration would be simpler and less error-prone. Such an approach also supports a much higher level of reuse.

As a second example, let's contrast the CI/CD models used by two popular tools; Tekton and Skaffold.

Tekton

Tekton's basic unit of execution is a task. A pipeline is used to order tasks. And tasks can depend on or produce resources. The Tekton controller uses the pipeline definition plus a task required resources or/or explicitly declared execution orders to instantiate tasks into Kubernetes Pods. The deterministic ordering of tasks is a concept that developers can easily understand.

Does it meet our criteria?

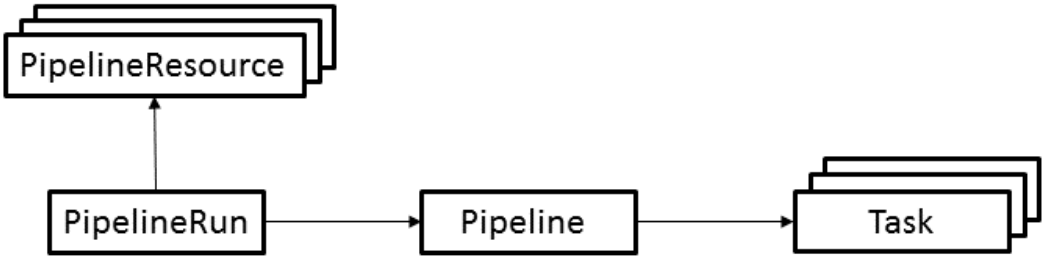
Is it SRP compliant?

It does decompose the CI/CD tool network components; pipelines, tasks, resources, steps. So if you consider executing a task as a single responsibility, the answer is yes. However, tasks include steps that can execute any number of programs or scripts. Also, the existing task catalog,

https://github.com/tektoncd/catalog, has a wide array of functions performed such as; building, deploying, and testing they meet the SRP definition.

Is it event-driven? No

Is it loosely coupled? While the pipeline specification is not, the ability to use Kubernetes specifications and selectors provides some level of loose coupling.

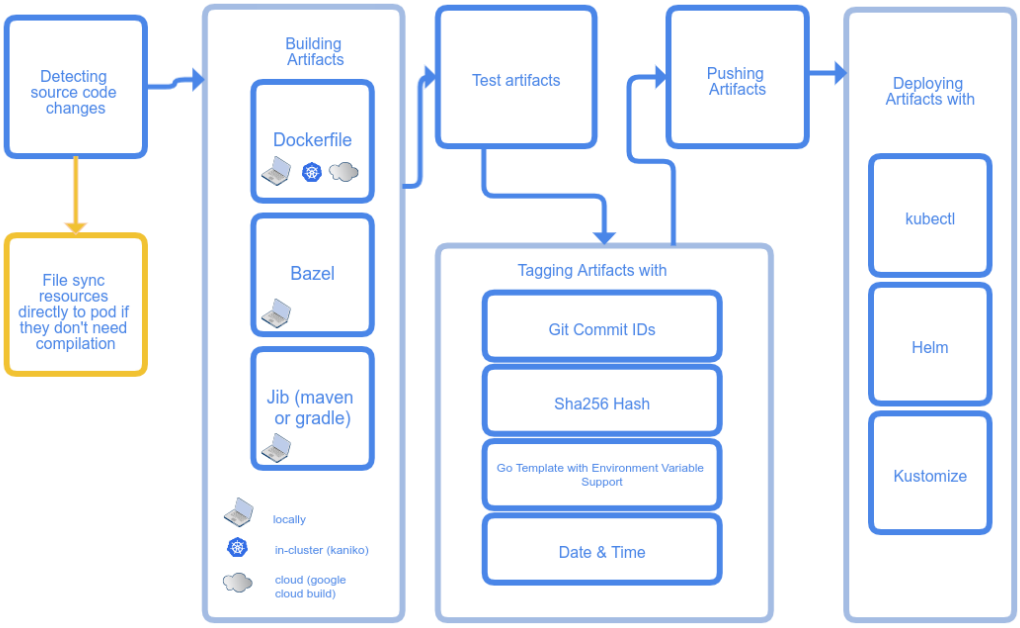


The CI/CD pipeline and constructs provided by Tekton can easily accomplish this using other infrastructure automation tools like SaltStack. Salt states can be used to execute tasks or construct pipelines, the order can be explicitly stated or inferred by requisites. SaltStack grains/pillars can act like labels/selectors.

So what did we gain with Tekton? Two key elements; the abstraction of resources and the use of Kubernetes labels and selectors to specify the tool network as code.

Skaffold

In contrast, Skaffold has a very different model. It decomposes, logically, into functional domains, build, test, artifacts, tag, and publishing. Or thinking of it another way, it takes Tekton tasks and breaks them down into the functions they perform. Unfortunately, its implementation is monolithic.



Does it meet our criteria?

Is it SRP compliant?

In some cases, yes. For example, a tagging service is decomposed well. However, it is possible to decompose other tasks like the "building artifacts" task further. Building artifacts include; compiling, execution of tests on the compiled code, manifest generation, packaging the code, testing the package code, and pushing all the build artifacts to a repository.

Is it event-driven? No, but it can produce gRPC events to trigger other build functions.

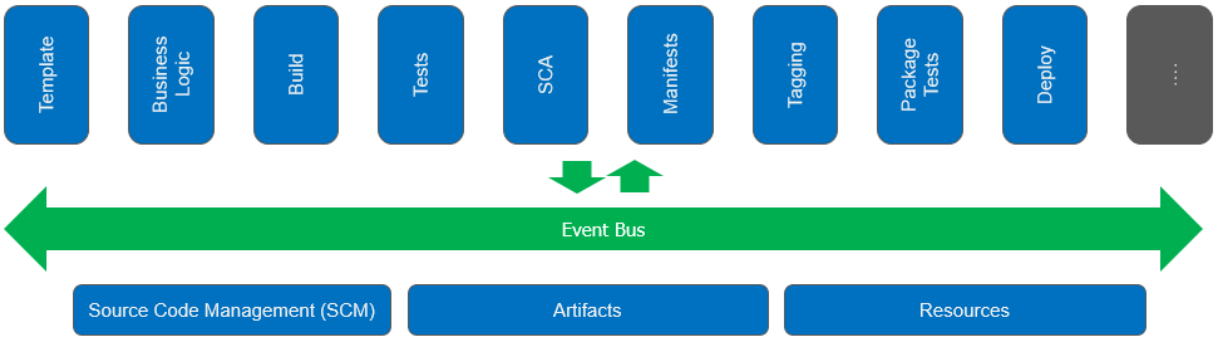
Is it loosely coupled? No

So what did we gain with Skaffold? A better abstraction.

Towards a composable tool network

What if we took the approach of Skaffold and Tekton a bit further? At PavedRoad, we are developing an open-source, low-code platform for writing microservices, serverless functions, and Kubernetes custom resource controllers. As part of that, we integrate a tool network that covers the major components of delivering code.

We started by applying the principle of SRP using the Skaffold model as a baseline. In our initial model, we came up with the following microservices briefly described below. Then we connected them with a Kafka message bus. We then applied the Kubernetes reconciliation pattern wherein an application specifies its desired state and controllers work to make its status match that spec.



Templates domain: Templates read from a set of code templates and apply code generation to construct a microservice, serverless function, or CRD. Each template implements a well-established design pattern, such as the tri-lateral pattern. Templates also generate unit, functional, and benchmark test suites. Each template includes entry points for users to add their business logic external to the template files.

Business Logic: Enables automation or notification to developers that a template has been created, updated, or deleted. For example, if a new version of a template is available, it can automate the reintegration of business logic.

Build domain: Generation of binaries or other elements need to package code for deployment. For Go, it also provides cross-compilation for different platforms.

Tests domain: Execution of unit, function, and benchmark tests used to validate build artifacts.

Static Code Analysis domain: Executes static code analysis, aka "linting," for security, programing standards, complexity, portability, code duplication, and more.

Manifest generation: Generates the manifests require for packaging a build for deployment. Currently, it supports generating Docker, Docker-compose, and Kubernetes manifests.

Tagging domain: Enables tagging for any artifact produced. Artifacts can be tagged individually, or all artifacts of a tool network invocation are tagged.

Packaging domain: Construction container images.

Package testing domain: Verifies a package image

Deployment domain: Supports deploying to a given environment. It supports applying deployment policies such as rolling, canary, Blue/Green, and A/B testing. Deploying to Kubernetes or other cloud services such as an EC2 instance.

Environment domain: Declaring and managing environments like development, test, staging, and production. (Not shown)

Source Code Management (SCM) domain: Integrates SCM systems like git for tracking source code, application configuration, IaC specifications, and the tool network specification itself.

Artifacts domain: Managing artifacts produced via microservices in the tool network.

Resource domain: Managing resources available to the tool network.

So what does this buy us?

Loosely coupling delivers flexibility and composability

A loosely coupled architecture brings us the standard microservices benefits:

- Easier to build and maintain applications
- Improved productivity
- Support for best technology use
- Scalability
- Support for autonomous cross-functional teams
- Technology experimentation
- Higher quality

Also, when we combine Kubernetes style labels and selectors we get:

- Composability or what we also call dynamic construction
- The ability to express our tool network as code

By Dynamic construction, we mean the ability of the tool network to adapt the availability of currently available resources. As an example, imagine a developer is on a flight or camping in the woods and only has access to the resources on his computer. With Dynamic constructions, steps can be omitted to save resources or reduce processing time.

It also provides ease of extension. Skaffold supports three methods of tagging and only tags images it builds. What if you wanted to add more methods or tag different assets like artifacts? My choices are to fork the repository and add my desired functionality and then maintain it, or create an issue, open a pull request, and hope the community accepts my contribution. If my feature only has value in my use case, it likely will be rejected.

In comparison, when we construct the tool network as microservices, you can create, package, and publish your new tagger via a service like; DockerHub, GitHub, or Google Container Registry. Now your new tagger is available for inclusion in your pipeline.

By creating independently deployable services, others can reuse just the components they want without taking an entire source distribution. Ultimately decreasing duplicate code, improving quality, and driving collaboration.

Note: The ability to discover contributions is critical for reuse and collaboration.

Events are valuable

Introducing an event bus like Kafka that persists all messages as immutable objects enables several vital capabilities.

Quickly understanding how a good or bad outcome occurred when responding to incidents or general questions is extremely valuable. This capability becomes more valuable as system complexity grows. In our approach, the producer of an event signs it using Blockchain. Each consumer of that event adds its signature, producing a secure record of the event flow and consumers and subsequent producers.

- Traceability: given an event source, be it the origin or a processing point in the tool network, you can track all preceding or proceeding consumers of that event.
- Visibility and insights: Standard event interfaces and the use of a message bus allows tools to determine the past or current state of a system and drive insights.
- Repeatability: Immutable event busses enable recreating the exacts sequence of any build or deployment.
- Experimentation: Allows that same message history to be played using new microservices and those results to be contrasted, determining the positive or negative effect achieved.
- Auditability: The combination of immutable events plus Blockchain provides the most robust possible audit trail.

Other Enhancements

In the ideal world, applications would already be well-architected, adequately packaged, and cloud providers would provide a standard API. In the real world, most applications are still monolithic, and cloud providers each have their API. Today, this requires developers to understand the nuances between cloud providers and develop expertise in each system. Given that the average enterprise uses 4.8 clouds, this places a heavy burden on engineers.

Similarly, engineers need to write code that accounts for the availability of resources in the tool network required by their application. Those required resources can become restricted for many reasons, including resource starvation, network partitions, or policy.

The cost of dealing with these complexities comes at the expense of other business functionality. To minimize that impact, we introduce several abstractions and patterns.

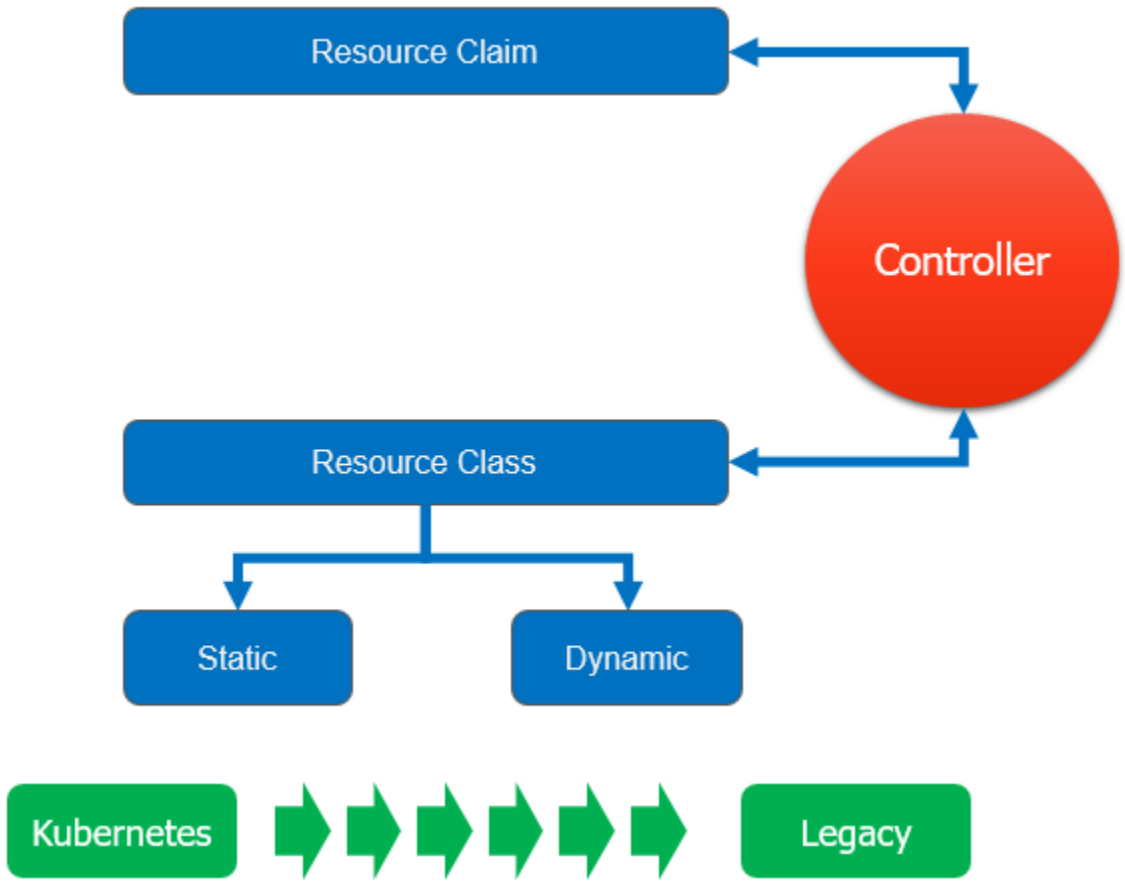
Resource abstraction

A resource represents an external piece of infrastructure ranging from low-level services like clusters and servers to higher-level infrastructures like databases, message queues, buckets, and more.

To achieve this, we leverage work by the Crossplane team on resource abstraction.

A resource claim is a persistent object that captures the desired configuration of a resource from the perspective of a workload or application. Its configuration is cloud-provider and cloud-offering independent, and it's free of implementation or environmental details. A developer or application owner is responsible for creating a resource claim. The claim is a request for an actual resource.

A resource class is a configuration that contains implementation details specific to a particular environment or deployment and policies related to a kind of resource. ResourceClass acts as a template with implementation details and policy for resources that get provisioned at deployment time. An administrator or infrastructure owner typically creates resource classes.



Each resource type implements a resource controller. The job of that controller is to reconcile the desired state specified by the specification with the actual state of the provisioned resource provided by the resource class.

References

1. https://www.cio.com/article/3267571/it-governance-critical-as-cloud-adoption-soars-to-96-percent-in-2018.html
2. https://12factor.net/
3. SRP https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view
4. https://crossplane.io/docs/v0.2/concepts.html