Cloud Best Practices: Containerized Development

By Dillon Lees

Designing for the Cloud

- Microservices
- API Contract
- Continuous Integration/Continuous Delivery

Microservices

- Highly modular
- Easier to understand, develop and test
- Resistant to crossing domain boundaries
- Idependently scalable

Conway's Law

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
> — Melvin E. Conway

API Contract

- Know your audience
- Create an API style guide and be consistent
- The API is a promise; treat it that way
- Design for your clients, not for your organization
- Separate API design from implementation details
- Santize all inputs/never expose passthrough functionality

Continuous Integration/Continuous Delivery

- Invest time to automate
- Invest in good tooling
- Review and change the tools that get in the way
- Use the testing pyramid
  - 70% Unit
  - 20% Integration
  - 10% End-to-end
- Continuously tighten the feedback loop

Challenges

- Coordination
- Reproducability

Coordinating Microservices is Tricky

- Components A, B and C are independently maintained, deployable microservices of a cloud offering
- All components maintain an exhaustive set of integration tests for their dependencies
- Component A has a dependency on B
- Component B has a dependency on C
- Component A is changed, passes all its integration tests and is deployed to production
- Component B is changed, passes all its integration tests and is deployed to production
- Component A is unaware that B has changed and is now broken

Works on my Machine ¯_( )_/¯

- Microservices introduce reproducability and portability challenges
- Reproducibility is necessary for root cause analysis
- Portability is necessary for healthy development process
    - Peer review
    - Pair programming

Solution

- Monorepo
- Containers
- Kubernetes

Monorepo

- All components share the same feedback loop
- Testing and deployment coordination is simple
- Greatly facilitates cross-training
- Encourages shared ownership
- Enables feature complete pull requests

Containers

- Maximizes portability
- Consistent Operation
- The building block for any modern Cloud provider

Kubernetes

- Runs everywhere
  - On-prem
  - Hybrid Cloud
  - Public Cloud
- Designed to minimize operational overhead
- Runs highly flexible workloads

Best-in-class Tools for Containerized Development

- Bazel
- Kubernetes
- Skaffold

Bazel

- Extensible
- Reproducable
- Portable
- Highly Scalable
- Fast

Kubernetes

- Service Discovery
- Automated Rollouts
- Automated Rollbacks
- Self-healing
- Secret and configuration management
- Open Source
- Maintained by Cloud Native Computing Foundation (CNCF)
- Widespread adoption

Skaffold

- Local Kubernetes Development
- Reproducible
  - `git clone`
  - `skaffold run`
- Tight feedback loop
- Only redeploys what's changed

Getting Started

Install Skaffold

Linux

```
curl -Lo skaffold https://storage.googleapis.com/skaffold/releases/latest/skaffold-linux-amd64 && \
sudo install skaffold /usr/local/bin/
```

Mac OS

```
brew install skaffold

# Alternatively
curl -Lo skaffold https://storage.googleapis.com/skaffold/releases/latest/skaffold-darwin-amd64 && \
sudo install skaffold /usr/local/bin/
```

Windows

```
choco install -y skaffold
```

Install Bazel

Technically we're installing Bazelisk which will pick the right version of Bazel to run.

Linux

```
curl -Lo bazel https://github.com/bazelbuild/bazelisk/releases/download/v1.10.1/bazelisk-linux-amd64 && \
sudo install bazel /usr/local/bin/
```

Mac OS

```
brew install bazelisk

# Alternatively
curl -Lo bazel https://github.com/bazelbuild/bazelisk/releases/download/v1.10.1/bazelisk-darwin-amd64 && \
sudo install bazel /usr/local/bin/
```

Windows

```
choco install bazelisk
```

Note: Consider creating a symlink to bazelisk.exe as bazel.exe if not already done for you.

Install kubectl

Linux

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl" && \
sudo install kubectl /usr/local/bin/
```

Mac OS

```
brew install kubectl

# Alternatively
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl" && \
sudo install kubectl /usr/local/bin/
```

Windows

```
choco install -y kubernetes-cli
```

Docker

Get Docker

Docker Desktop

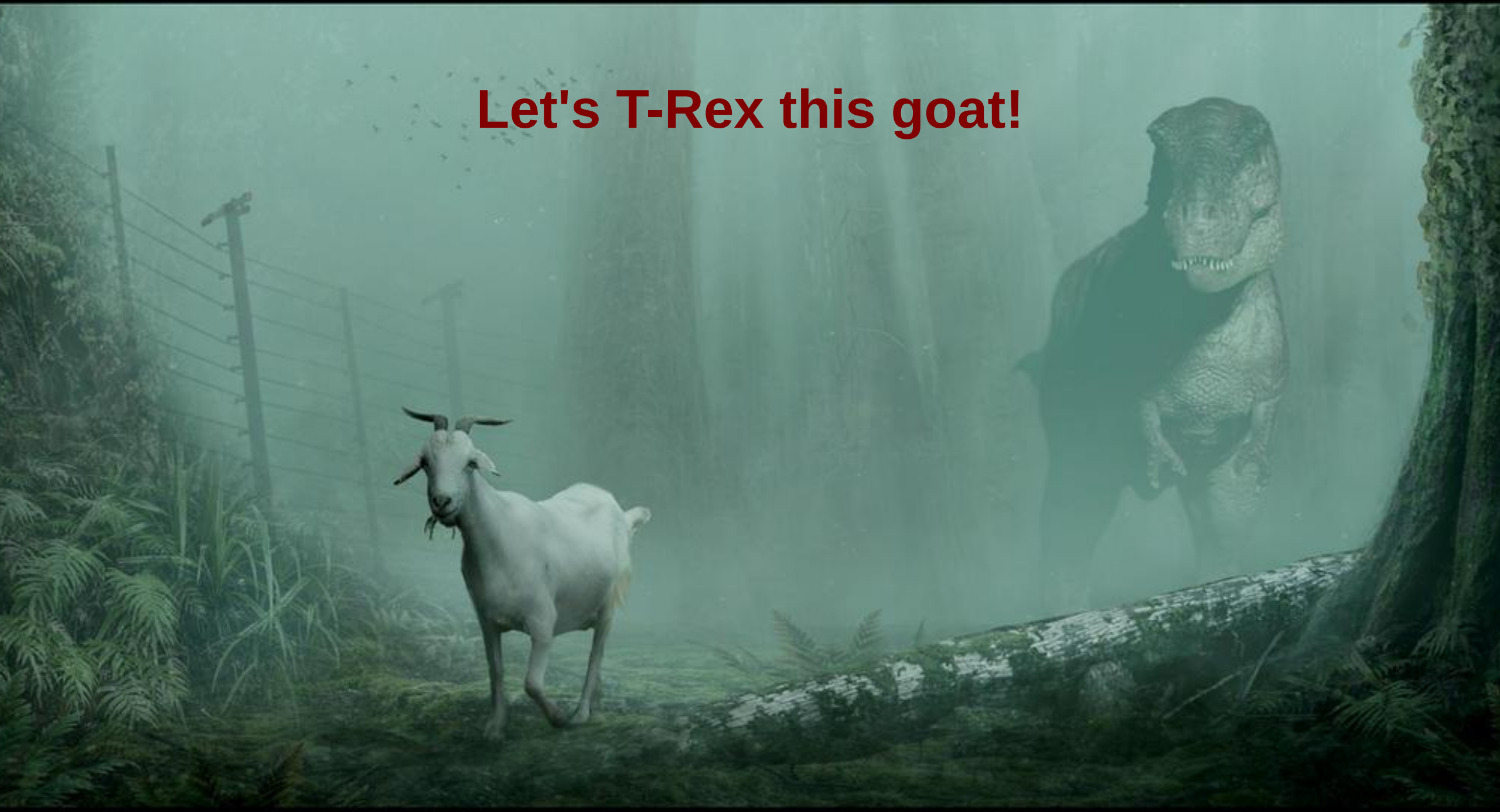
`Preferences > Docker Engine`

```
{
  "features": {
    "buildkit": true
  },
  "experimental": true
}
```

`Preferences > Kubernetes > Enable Kubernetes`

Other Kubernetes Distros

- kind
- microk8s
- minikube
- k3s
- k3d
- kubeadm


Let's T-Rex this goat!

The Example Repository

The following command will clone a repository containing a webapp with a microservice architecture built and deployed with Bazel and Skaffold.
The backend API is written in Go and the frontend is written using NextJS.

```
git clone git@github.com:ddlees/microservices.git
```

The Magic!

If everything is installed correctly, the following command will build and deploy the api and ui container images continuously. Make a change to `api/api.go` or `ui/pages/index.tsx` and watch how Bazel and Skaffold work together to propogate your change out to your cluster.

```
skaffold dev
```

Note: Changes to `ui/pages/index.tsx` will appear automatically. To see the ui consume any changes made to `api/api.go` be sure to refresh your browser.

How does it all work?

The `skaffold.yaml` file

Here we tell skaffold how each artifact in the project is built. The `api` image is built using Skffold's integrated Bazel support so we only need to give it the Bazel target. The `ui` image is a little special; here we're relying on Bazel to pack our existing tooling into the container and letting skaffold sync the files. This allows teams to gradually migrate building their artifacts with Bazel if they wish to do so or not.

```
artifacts:
  - image: api
    bazel:
      target: //api:image.tar
      args:
        - '--platforms'
        - '@io_bazel_rules_go//go/toolchain:linux_amd64'
  - image: ui
    custom:
      buildCommand: 'bazel run ui:latest --platforms @build_bazel_rules_nodejs//toolchains/node:linux_amd64'
      dependencies:
        paths:
          - ui/**/*
```

The `skaffold.yaml` file

Skaffold supports syncing changed files to a deployed container to avoid the need to rebuild, redeploy, and restart the corresponding pod. Since we want to use our existing tooling to rebuild the ui, here we're telling skaffold which files it should sync to leverage the hot-reloading feature in our existing tooling.

```
sync:
  manual:
    - src: ui/pages/**/*
      dest: /app/ui/image.binary.runfiles/microservices/ui/pages/
      strip: ui/pages/
    - src: ui/public/**/*
      dest: /app/ui/image.binary.runfiles/microservices/ui/public/
      strip: ui/public/
    - src: ui/styles/**/*
      dest: /app/ui/image.binary.runfiles/microservices/ui/styles/
      strip: ui/styles/
```

The `skaffold.yaml` file

Now that Skaffold knows how to build our container images we need to tell it how to deploy our images and which ports to forward from the cluster to the local machine.
In this case we're telling Skaffold to deploy our native Kubernetes manifests located in the `k8s` directory using `kubectl`.

```
deploy:
  kubectl:
    manifests:
      - k8s/*
portForward:
  - resourceType: deployment
    resourceName: api
    port: 8080
    localPort: 8080
  - resourceType: deployment
    resourceName: ui
    port: 8080
    localPort: 3000
```

The `api/BUILD.bazel` file

The `rules_go` and `rules_docker` Bazel rule sets give us declarative APIs for telling Bazel what we want to build. Here we're telling Bazel to build a go binary for our host machine and a container image to run on our cluster.

```
load("@io_bazel_rules_docker//go:image.bzl", "go_image")
load("@io_bazel_rules_go//go:def.bzl", "go_binary")

go_binary(
    name = "api",
    srcs = ["api.go"],
)

go_image(
    name = "image",
    srcs = ["api.go"],
    goarch = "amd64",
    goos = "linux",
    static = "on",
)
```

The `ui/BUILD.bazel` file

The `rules_nodejs` Bazel rule set gives us API to leverage existing tooling for frontend development. In this case, we're having Bazel leverage existing tooling to build artifacts for production and development suitable for the host machine. Additionally, we're having Bazel wrap the tooling in a NodeJS container image for us to use on our cluster.

```
next(
    name = "ui",
    args = ["dev", "ui"],
    data = _RUNTIME_DEPS + ["//:node_modules"],
)

next(
    name = "dist",
    args = ["build", "ui", "$(@D)"a],
    data = _RUNTIME_DEPS + ["@npm//:node_modules"],
    output_dir = True,
)

nodejs_image(
    name = "image",
    args = ["dev", "ui"],
    data = _RUNTIME_DEPS + _DEPENDENCIES,
    entry_point = "@npm//:node_modules/next/dist/bin/next",
)
```

The `WORKSPACE` file

The `WORKSPACE` file is Bazel's way of declaring what rules and/or external dependencies required to build the artifacts
in your project. In lieu of explaining each `WORKSPACE` function, the gist of what this is doing in the example repository is that it's pulling the required ruleset for building `NodeJS/Javascript/Typescript` projects, the ruleset for building `Golang` projects and the ruleset for building container images.

The `WORKSPACE` file

```
workspace(
  name = "microservices"
  managed_directories = {"@npm": ["node_modules"]},
)

load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")


########
# NodeJS
################

http_archive(
    name = "build_bazel_rules_nodejs",
    sha256 = "16fc00ab0d1e538e88f084272316c0693a2e9007d64f45529b82f6230aedb073",
    urls = ["https://github.com/bazelbuild/rules_nodejs/releases/download/0.42.2/rules_nodejs-0.42.2.tar.gz"],
)

load("@build_bazel_rules_nodejs//:index.bzl", "yarn_install")

yarn_install(
    name = "npm",
    package_json = "//myjsstuff:package.json",
    yarn_lock = "//myjsstuff:yarn.lock",
)

load("@npm//:install_bazel_dependencies.bzl", "install_bazel_dependencies")

install_bazel_dependencies()

load("@npm_bazel_typescript//:index.bzl", "ts_setup_workspace")

ts_setup_workspace()


########
# Go
################

http_archive(
    name = "io_bazel_rules_go",
    sha256 = "a8d6b1b354d371a646d2f7927319974e0f9e52f73a2452d2b3877118169eb6bb",
    urls = [
        "https://mirror.bazel.build/github.com/bazelbuild/rules_go/releases/download/v0.23.3/rules_go-v0.23.3.tar.gz",
        "https://github.com/bazelbuild/rules_go/releases/download/v0.23.3/rules_go-v0.23.3.tar.gz",
    ],
)

load("@io_bazel_rules_go//go:deps.bzl", "go_rules_dependencies", "go_register_toolchains")

go_rules_dependencies()
go_register_toolchains(
    go_version = "1.14.4",
)

########
# Docker
################

http_archive(
    name = "io_bazel_rules_docker",
    sha256 = "4521794f0fba2e20f3bf15846ab5e01d5332e587e9ce81629c7f96c793bb7036",
    strip_prefix = "rules_docker-0.14.4",
    urls = ["https://github.com/bazelbuild/rules_docker/releases/download/v0.14.4/rules_docker-v0.14.4.tar.gz"],
)
load(
    "@io_bazel_rules_docker//repositories:repositories.bzl",
    container_repositories = "repositories",
)

container_repositories()

load(
    "@io_bazel_rules_docker//repositories:deps.bzl",
    container_deps = "deps",
)

container_deps()

load("@io_bazel_rules_docker//repositories:pip_repositories.bzl", "pip_deps")

pip_deps()

load(
    "@io_bazel_rules_docker//go:image.bzl",
    _go_image_repos = "repositories",
)

_go_image_repos()

load(
    "@io_bazel_rules_docker//node:image.bzl",
    _nodejs_image_repos = "repositories",
)

_nodejs_image_repos()
```

Questions?