Management of Long-lived Servers:

The Missing Bits for Avoiding Config Drift with Declarative Configuration and Image-based Updates

Container Plumbing Days – 22 March 2023

I'm Kai,

one of the Flatcar Container Linux maintainers

Kai Lüke

Senior Software Engineer, Microsoft

Email: kailuke@microsoft.com

GitHub: pothos

flatcar.org

Agenda

Container OS and Immutable Infrastructure

Problem Statement: Declarative Reconfiguration

Existing Solutions and New Ideas

Demo with Flatcar Container Linux

Infra Recap: Three Principles

- Immutable Infrastructure
  - Avoid config drift by not managing servers but fully redeploying them for changes
  - Examples: Terraform (only first-boot SSH), Ignition, custom OS images, with caveats: Ansible or cloud-init if not used for reconfiguration
- Infra as Code
  - No manual actions, only automation, and ideally version-controlled
  - Examples: Terraform, Ansible, or 'kubectl apply' from Git folder or with GitOps
- Declarative Configuration
  - Guarantees described target state to be reached, cf. to using scripts
  - Examples: Ignition, Terraform, cloud-init, Ansible, all with the constraint to be careful with reconfiguration (old server state may be kept)

Goals

- Reproducibility/Reliable Operations: Easily get to the declared target state, apply and revert Infra changes in version control
  - Introspection: Ability to check if deployment matches declaration, or to identify a working version
- Security/Automatic Updates: Be on the latest version
- Low Maintenance: Automated deployment and updates
  - Low coupling of parts to facilitate easy updates without blocking dependencies
  - Health check for automatic rollback after breaking updates

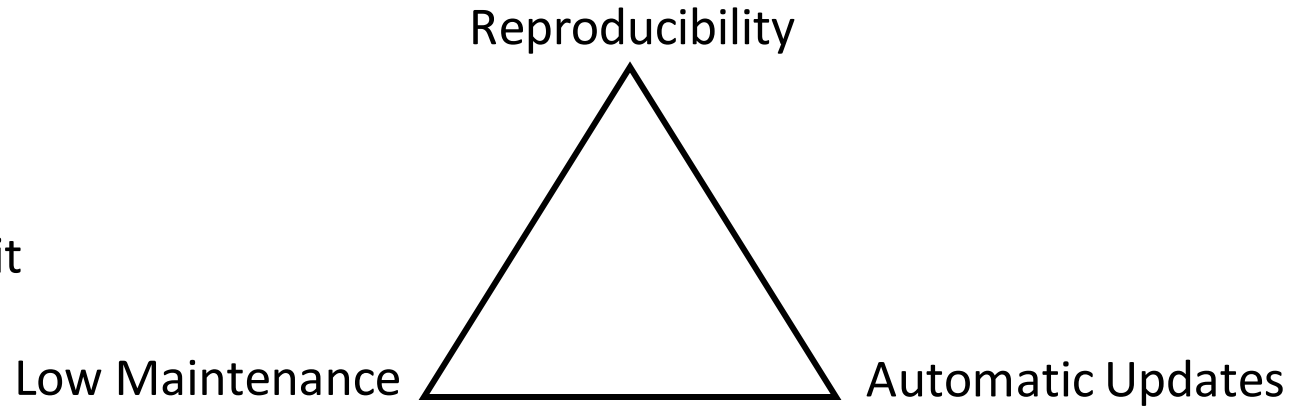
Example: Managing a Container (Image Reference)

- Immutable Infrastructure: We recreate the container and don't update it from the inside
- Declarative configuration: Specific container image hash or tag
- Reproducibility: Hash is stronger than tag
  - Introspection: Compare hashes, or find resolved hash for a tag
- Automatic Updates: Bump the tag (e.g., 'caddy:2' means latest version 2.x.y release), vs. update hash/tag in the configuration

Do we pick 2 or even 3?

E.g.: Automate the hash updates in Git

plus health check for revert



Container OS: Optimized for Modern Infra Principles

- Immutable “hermetic” /usr OS partition, image-based updates
  - Low coupling between OS container runtime and containers
  - Reproducibility (use version X), introspection (check OS version, image hash)
- Inbuilt first-boot provisioning with declarative configuration
  - Like a container, the OS may need additional configuration and data
- Automatic updates with rollback on failed health check
- Image-based OS in-place updates sidestep full redeployment to follow immutable infra and get a reproducible behaviour

Image-based OS: Control OS version

- Roll out of a specific version either client-side or update-server-side (cf. container image tag bump)
- Example: Flatcar asks the Nebraska update-server what version this machine should be on (benefit: rollout policies), but also has flatcar-update tool to switch to a specific version
- GitOps can be done both ways

Pain Points

- We still need a full redeployment for configuration changes
  - Takes long, local data gets lost, new IP addresses, SSH host keys…
- Escape hatch for Ignition users: Ansible
  - But reconfiguration does not work reliably when the declarative configuration can't clean up the side effects → Old server state creates “config drift” (Is it the reason why things are broken now, or do we rely on it and created a trap?)
- Rerunning Ignition is tricky and needs cleaning up the rootfs (Ignition rootfs reformat works in Flatcar)

Problem Statement: Declarative Reconfiguration

- Long-lived servers: reliable reconfiguration, keeping local data
- Server State: OS version, configuration, data, state
  - OS version (/usr) can be controlled client side or server side → Update in-pace
  - Config files under /etc from previous configuration (if allowed) → Discard
  - OS config files under /etc (legacy software) → Should be managed by OS
  - Local data can be somewhere on rootfs (if allowed) → Keep
  - Wanted state to keep like secrets or /etc SSH host keys → Keep
  - Unwanted state from side effects → Discard

Current Approaches: Minimize and Define State

- Disallow traditional Linux usage and its local state
- Define what local state is allowed and how it's managed
- Example: Single-purpose Container/Kubernetes OS
  - Talos Linux or Bottlerocket with read-only rootfs plus ephemeral and specific persistent storage
  - Talos reset tool, and even on updates Talos discards Kubernetes state by default
  - Aurae as another ”API-only“ OS design

Current Approaches: Traditional Linux with Managed /etc

- Both user and OS config files are under /etc and need to be managed
- Let's first look at OS config files
  - Modern software first looks under /usr and supports drop-in overwrites (see UAPI Group's Base Dir specification, libeconf)
  - But legacy software needs full config files in /etc instead of just drop-in for extension/overwrites
  - Package managers can update OS config files under /etc and can identify user changes → Need solution for image-based OS

Current Approaches: Update OS Config files in /etc

- CoreOS Container Linux approach was to use symlinks to /usr or downstream patches (systemd-tmpfiles files get outdated)
- Fedora CoreOS does 3-way merge of old/new OStree state and user changes in /etc (file-based, user-changed files win)
- New in Flatcar Container Linux: /etc as overlay mount with OS files from /usr/share/flatcar/etc (file-based, user-changed files win), can show changes with

  sudo git diff --no-index /usr/share/flatcar/etc/ /etc/

New Upstream Developments

- System extension images with systemd-sysext
  - Allows to extend /usr or /opt through overlay mounts
  - Can be in a single file for atomic updates
  - Verifiable for integrity, easy introspection through version metadata or file hash
- Soon: systemd-syscfg extensions for /etc - hopefully with mutable mode like Flatcar's /etc overlay
- User configuration can be updated almost reliably! Could bake such images from Butane for Ignition, … but side effects may still be there (opt-in vs opt-out discarding of old state)

New idea: Selective OS Reset

- To get opt-out discarding of old state we can let the user define what data to keep on reconfiguration
- New in Flatcar: flatcar-reset tool to stage an OS reset on next boot (rootfs cleanup)
  - Can specify regex for paths to keep, e.g., /var/lib/containerd, /etc/ssh/ssh_host_.* or /var/log
  - Will rerun Ignition, optionally with a different config source
- Last missing bit to reliably reconfigure a server instead of having to fully redeploy

Demo

Terraform Setup

- Ignition config in cloud instance userdata, updated in-place, triggers flatcar-reset while keeping machine-id, SSH host keys and certain app data (Terraform workaround: skip first run of helper script, Azure workaround: avoid outdated userdata)
- Observe that state file created by old test service definition is cleaned up and that new configuration is there

Flatcar Demo

Recording:

https://youtu.be/1_QcY1ic7mA

Terraform code and demo script under:

https://github.com/flatcar/flatcar-terraform/tree/main/azure-without-instance-replacement – for Azure

https://github.com/flatcar/flatcar-terraform/tree/main/equinix-metal-aka-packet-without-instance-replacement – for Equinix Metal

Thank you

Kai Lüke

Senior Software Engineer, Microsoft

Email: kailuke@microsoft.com

GitHub: pothos