Reactive Nix

Enabling Functional Reactive Configuration with **mgmt**

Jannik Höfler

https://github.com/karpfediem/rx.nix

Contents

Motivation

Why this talk?

Comparison of secret managing schemes[2]

Scheme	Pre-build	Build time	In the store	System activation	Runtime	Encryption technology	Usable project	Templating support	Additional notes
`deployment.keys.<name>` option in NixOps	Plaintext value in a Nix expression.	N/A	Not stored in the Nix Store.	N/A[note 1]	Unencrypted in `/run/keys` or configured path.	N/A	Yes	No	Secret management happens outside of `nixos-rebuild`
Agenix	Encrypted raw files, `agenix` CLI encrypts with the user and host ssh key	N/A	Encrypted	Decryption with the host SSH key.	Unencrypted in `/run/secrets` or configured path.	Uses `age` with SSH user and host keys; does not support `ssh-agent`.	Yes	No	
`agenix-rekey`	Extended `agenix`.	N/A	Encrypted	Decryption with the host SSH key.	Unencrypted in `/run/secrets` or configured path.	Use with `agenix`; provides more convenience.	Yes	No	
ragenix	Encrypted raw files, `ragenix` CLI encrypts with the user and host SSH keys.	N/A	Encrypted	Decryption with the host SSH key.	Unencrypted in `/run/secrets` or configured path.	Drop-in replacement of `agenix`, written in Rust and based on the `age` crate.	Yes	No	
sops-nix	Encrypted file with `age`, PGP or SSH key, support yubikey when gnupg is used, can be stored in a Git repository.	N/A	Encrypted	Decryption with GPG or `age` keys.	Stored in `/run/secrets` with configurable permissions.	Uses SOPS	Yes	Yes	Can be used with NixOps, `nixos-rebuild`, krops, morph, nixus and possibly other deployment tools.
krops	Stored in the password store.					Uses the password store (aka `pass`) which uses GPG.	Yes	No	
terraform-nixos	Plaintext value in a Nix expression.				Stored in `/var/keys` owned by the `keys` Unix group.		Yes	No	See the `terraform-nixos` documentation.
secrix	Encrypted raw files, like `agenix`.		Encrypted	Decryption with the host SSH key.	Unencrypted in configured path in `/run`.	Uses `age` by default with SSH user and host keys; does not support `ssh-agent`.	Yes	No	Focuses on trying to keep secrets decrypted for a minimal amount of time.
vaultix	Encrypted raw files like agenix		Encrypted	Decryption with the host SSH key.	Unencrypted in specific paths.	Powered by the age Rust crate.	Yes	Yes	
brizzbuzz/opnix						1password			
Blog Entry: wrapper around `pass` based on `nix-plugins`.	Stored in the password store.	Data is retrieved/decrypted with `pass` during evaluation time.	Unencrypted in the store.			Uses the password store (aka `pass`) which uses GPG.	No	No	
`builtins.readfile` `builtins.exec`[6]	`builtins.readfile` can read any file, `builtins.exec` can execute commands and thus query any kind of database or password manager.	These functions return values in a Nix expression; it is up to the user what happens to these values in the NixOS configuration.	See "build time"	See "build time"	See "build time"	These functions just read files or execute commands, they do not provide anything inherently "secure" or "cryptographic".	No	No	The referenced NixOS Discourse discussion is about a signing key that is only needed during build time and should not be stored in the nix store at all.
Scheme	Pre-build	Build time	In the store	System activation	Runtime	Encryption technology	Usable project	Templating support	Additional notes

Figure 1: Comparison of secret managing schemes [1]

- Took issue with how secrets management is currently done in NixOS
  - https://wiki.nixos.org/wiki/Comparison_of_secret_managing_schemes
- Investigated why **exactly** I was unhappy
- How can it be improved?

Goal for today

- Understand reactive systems and functional **reactive** programming
- What **can** and what **can't** we solve with *nix* today?
- Motivation to view existing problems through a reactive lens

- NixOS is excellent at building **correct snapshots**.
- Some properties we care about are **not snapshot properties**:
  - time windows
  - drift/tampering
  - delayed readiness (ACME certificates for services)
  - distributed dependencies (Wireguard, …)

Two mental models

Snapshot (NixOS)

config as input ⟶ system as a closure

$$\text{system} = f(\text{config})$$

Trajectory (Goal)

config as input ⟶ reactive function over time ⟶ system

$$\text{system}(t) = f(\text{config}, \text{environment}(t))$$

Evolution of Config Management

Alice & Bob build a “house”

Alice & Bob build a "house"

Figure 2: A familiar cast [2]

- They somehow **really** like SSH :)

Handcrafted (manual, local state)

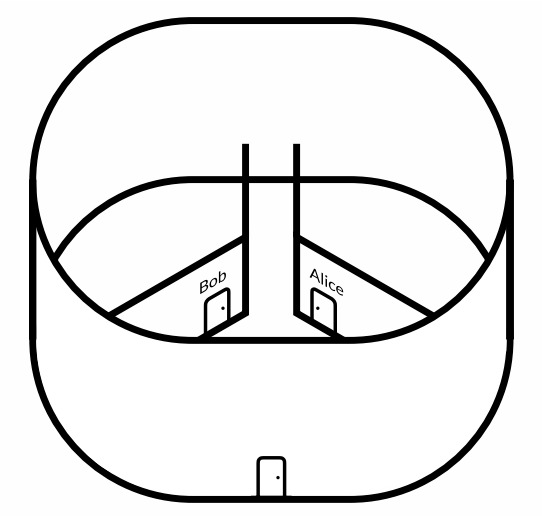

Figure 3: One door, one shared key: a simple, slow-changing world.

Problem	• low scale • correctness is “works at all”
Change frequency	months
Acceptable delay	days
Complexity	trivial

	Technique	• manual edits • shared context • tribal knowledge
✓	NixOS	unnecessary overhead

Repeatability pressure

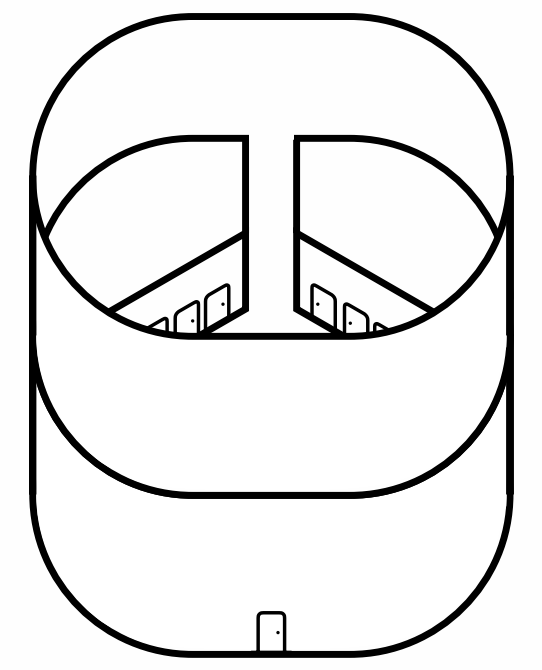

Figure 4: As people/doors grow, execution consistency becomes the limiting factor.

	Problem	onboarding/offboarding repeats; humans miss steps.
	Change frequency	weekly
	Acceptable delay	hours
📈	**Complexity**	trivial→medium

📉	**Technique**	runbooks → scripts → templates
✓	**NixOS**	strong: intent as code; reviewable; reproducible.

Standardize and batch apply

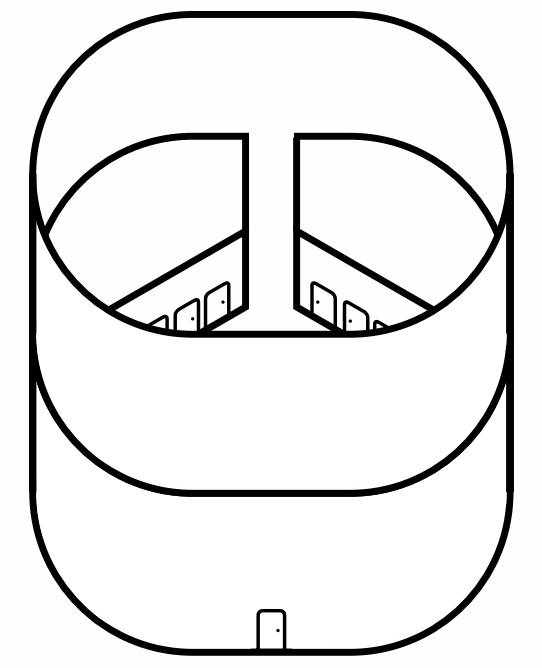

Figure 5: “Do the same thing everywhere” becomes the goal.

Problem	many similar targets | reduce variance
Change frequency	weekly⟶daily
Acceptable delay	hours
Complexity	medium

📈	**Technique**	a batch “apply” (scripts/playbooks)
✓	**NixOS**	strong: produces consistent snapshots

Spec becomes the center

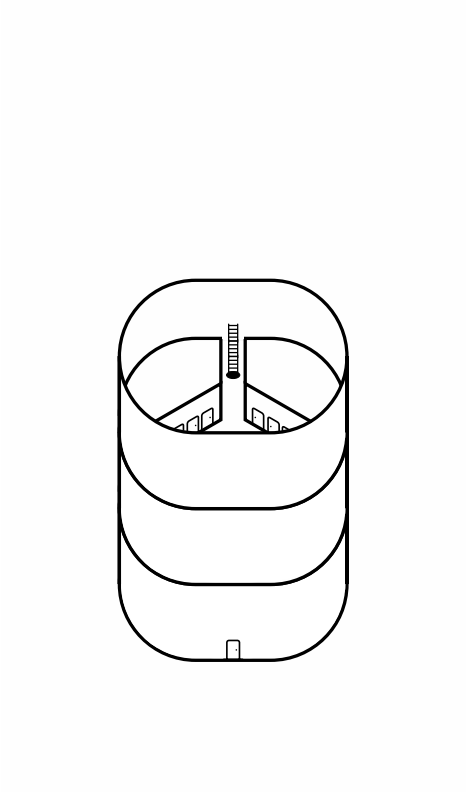

Figure 6: Intent moves from procedure toward specification.

	Problem	scripts get brittle (ordering, partial failure)
	Change frequency	daily
	Acceptable delay	hours
📈	**Complexity**	medium⟶high

📉	Technique	describe **what should be true** (resources/invariants)
✓	NixOS	excellent snapshot spec | still point-in-time

Drift appears (periodic enforcement)

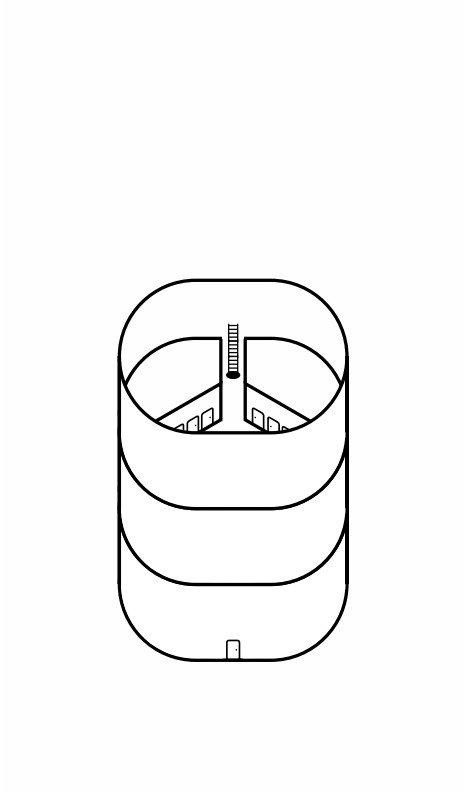

Figure 7: Reality changes between applies.

Problem	out-of-band changes and churn cause drift
Change frequency	daily + random drift
Acceptable delay	hours⟶10–60 min
Complexity	high

📉	**Technique**	periodic re-apply (cron/timers) | idempotency
✓	**NixOS**	immutability guarantees | otherwise: rebuild still doable, but expensive

The drift window is the pain

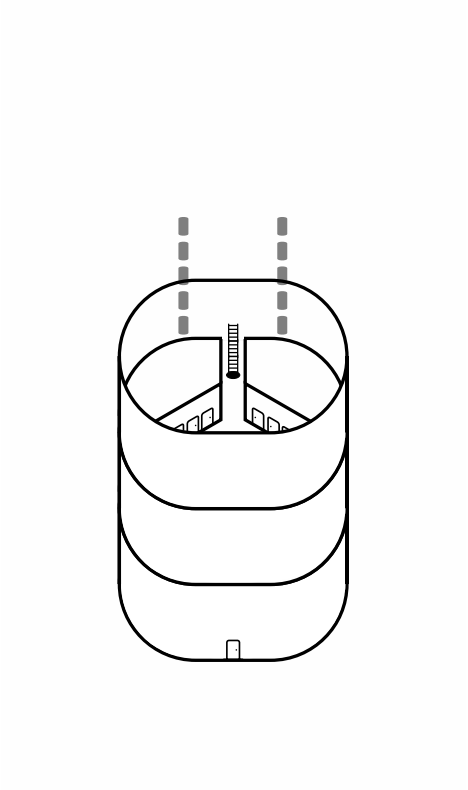

Figure 8: More frequency reduces drift windows, but raises cost and side effects.

Problem	“apply succeeded” ≠ “correct over time”.
Change frequency	minutes if you chase drift
Acceptable delay	10–60 min⟶minutes
Complexity	high

📈	**Technique**	crank cadence ⟶ more compute + more failure surface.
✗	**NixOS**	evaluation cost is a hard limit for fast signals.

Policy constraints (time windows)

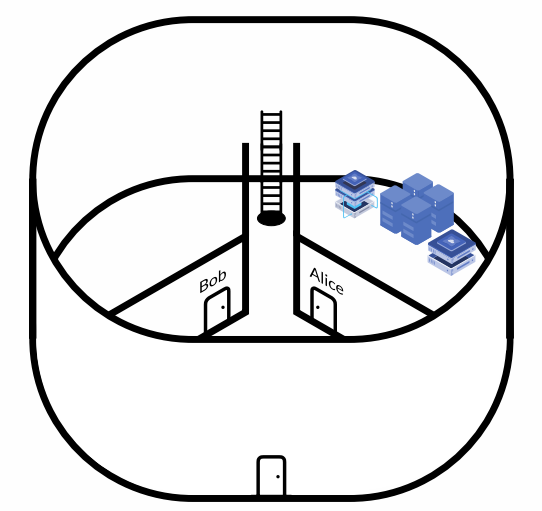

Figure 9: A server room introduces conditional access: policy, not just membership.

	Problem	Access depends on time or context: • weekdays, work-hours • short-lived access • lockdown mode
	Change frequency	hours
	Acceptable delay	minutes
📈	**Complexity**	high⟶very high

📈	**Technique**	re-compute + deploy from policy inputs
✗	**NixOS**	rebuilding on clock edges is semantically awkward and operationally heavy

Tampering and sensors (event-driven remediation)

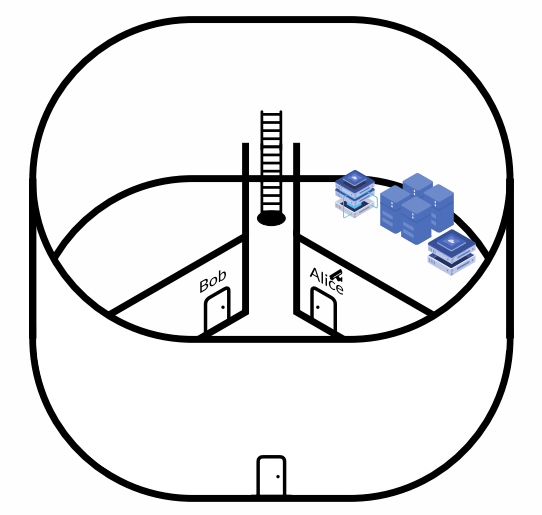

Figure 10: When drift is adversarial or accidental, latency dominates.

Problem	revoke/fix now; not “next round”.
Change frequency	continuous drift events
Acceptable delay	seconds/immediate
Complexity	very high

📈	**Technique**	alarms/watchers trigger remediation immediately.
✗	**NixOS**	systemd.path/inotify + oneshots work, but policy becomes glue.

Controllers as the default runtime

	Problem	many subsystems interact; “a job ran” is not a success condition.
	Change frequency	continuous small updates
	Acceptable delay	bounded seconds
	**Complexity**	very high⟶extreme

	Technique	• always-on reconciliation loops (controllers): ▸ observe desired + observed state ▸ reconcile continuously ▸ report “converged/healthy”
✗	NixOS	per task custom tools/scripts/services | Controller sprawl

Complexity Overload

Complexity Overload

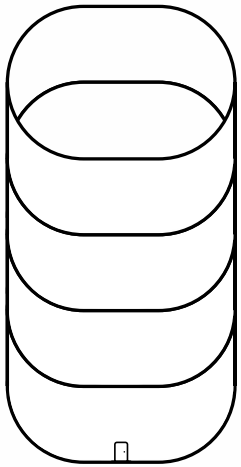

Figure 11: Bob & Alice like to build

As it turns out,...

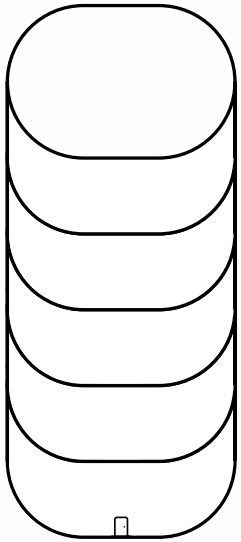

Figure 12: Bob & Alice like to build

…what Alice and Bob were building…

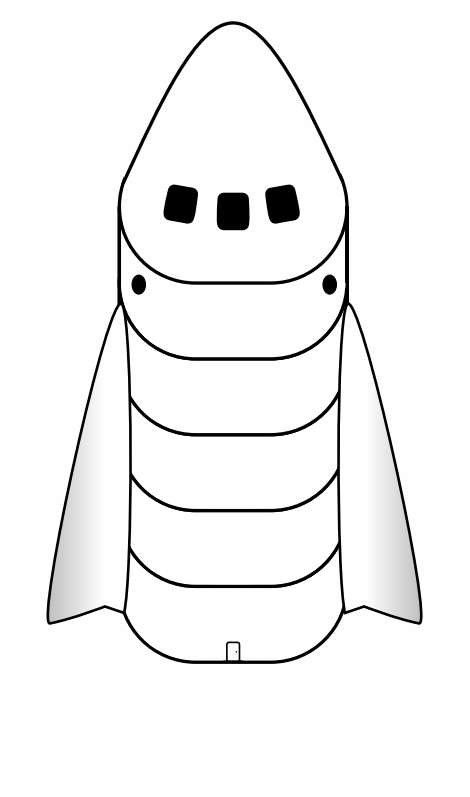

Figure 13: Bob & Alice like to build

…what Alice and Bob were building…

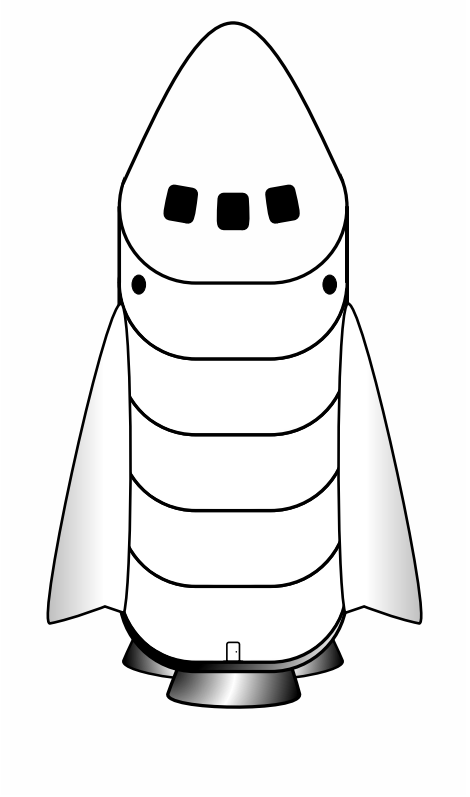

Figure 14: Bob & Alice like to build

...was a spacecraft after all!

A Moving Target

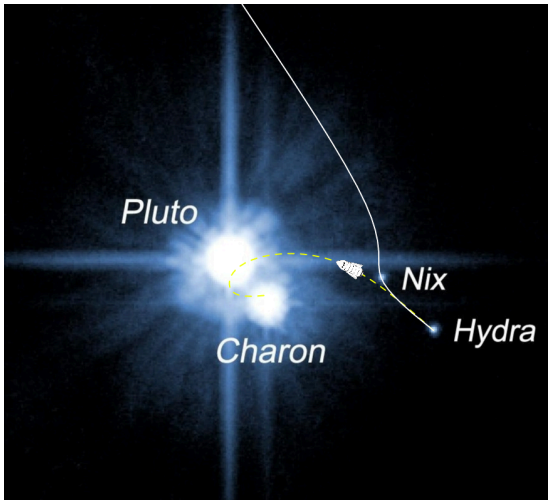

Figure 15: A moving target: desired course vs. actual course (adapted from [3])

Problem

- the desired course changes continuously (policy + environment)
- the actual course deviates (disturbances, drift, partial failures)
- sensors are noisy/delayed; actuators are discrete

Change frequency	continuous
Acceptable delay	bounded seconds
🚀 **Complexity**	very high⟶extreme

A Moving Target

🚀	**Technique**	Continuous-Time Control Theory
✗	**NixOS**	???

Can we still do it?

From Snapshots to Trajectories

- The target itself moves: the desired state is a function of time-varying signals.
- We want the spec to denote a value at any time:

Signal-defined desired state

```
system(t) = f(config, environment(t))
```

Figure 16: mgmt Logo [4]

Goals:

- keep NixOS for deterministic construction
- add a runtime for correctness **over time**.

If `system(t)` is a moving target, we need a runtime that:

- subscribes to signals (time, events, health)
- propagates changes through dependencies
- converges with bounded reaction time

> mgmt provides such a runtime (FRP + resource graph + convergence)[5]

Figure 17: mgmt Logo [4]

mgmt

- closed loop controller:
  - observe ⟶ compute error ⟶ actuate
- stability constraints (hysteresis/backoff/rate limits)
- choreography (distributed state, leader-election)
- propagate changes through dependencies; converge continuously

rx.nix: Integrating mgmt with NixOS

What do we need to do?

- Describe the desired state as a total function **over time**:
  - via mgmt's functional-reactive language **MCL**
- NixOS builds the scaffold: packages, services, wiring
- Activate the reactive engine
  - mgmt executes the control loop in flight

What rx.nix does today

- Embed MCL code inside Nix.
- Bundle it into a mgmt module.
- Include a systemd service that runs mgmt in the resulting NixOS output.

Division of responsibility

- NixOS builds the **snapshot skeleton**.
- mgmt maintains **correctness over time**.

Where this becomes interesting in NixOS

- Fast-changing inputs that are expensive to evaluate into the Nix snapshot:
  - secrets and cert readiness
  - health-driven restarts
  - distributed dependencies
- The hand-off point:
  - Nix decides **what is possible** (packages, services, wiring).
  - The reactive engine decides when to act (signals and bounded reaction).

Demo

Reactive Nix

https://github.com/karpfediem/rx.nix
https://codeberg.org/karpfen/cfgmgmtcamp2026
@karpfen:matrix.org

mgmt

https://github.com/purpleidea/mgmt
https://mgmtconfig.com/
+ Workshops

References

[1] NixOS Wiki contributors, "Comparison of secret managing schemes."

[2] R. Munroe, "Protocol." Mar. 10, 2014.

[3] NASA, "Pluto and its satellites (2005)." 2005.

[4] purpleidea/mgmt contributors, "mgmt_logo_default_tall.svg." 2026.

[5] purpleidea and mgmt contributors, "mgmt: Next generation distributed, event-driven, parallel config management." Accessed: Jan. 31, 2026. [Online]. Available: https://github.com/purpleidea/mgmt