# KubeRift

Fuzzy-First Kubernetes Operations Platform

A complete reference for installing, operating, and extending KubeRift — the terminal-native Kubernetes navigator written in Rust. Covers architecture, all keyboard actions, multi-cluster workflows, security model, and the open-source story of how a fuzzy-finder library became the foundation of a Kubernetes operations platform.

| | |
|---|---|
| **Version** | 0.1.2 |
| **Binary** | `kf` |
| **Source** | github.com/syedazeez337/kuberift |
| **Crate** | crates.io/crates/kuberift |
| **License** | MIT |
| **Date** | March 2026 |

## Contents

## 1 The Skim Insight: Architecture Born from a Library

### 1.1 The Problem Space

Every Kubernetes practitioner eventually arrives at the same frustration: `kubectl get pods --all-namespaces` produces a wall of text, `kubectl describe` requires knowing the exact resource name, and switching between resource types means retyping commands. Existing tools like k9s take a "modal" approach—navigate menus, select a resource type, then a namespace, then an item—borrowed from graphical file managers.

KubeRift was conceived around a different premise: *what if the search box is the interface?* Type fragments of a pod name, a namespace, a status — the list filters instantly. No modes, no menus, no hierarchy to navigate. Just a fuzzy query and a live list.

That idea pointed immediately to a fuzzy-finder. But which one, and how?

### 1.2 Discovering Skim as a Library

The first prototype used **fzf** — the ubiquitous Go binary — through a subprocess pipe. It worked but felt architecturally wrong: items had to be serialized to plain text strings, sent through a pipe, and the selection had to be parsed back. Colors required ANSI escapes baked into strings. Preview required a shell command string. Multi-select results came back as newline-delimited text. Every richness in the Rust data model was lost at the pipe boundary and had to be laboriously reconstructed.

The pivotal discovery came from reading the README of **skim** (crates.io/crates/skim), a fuzzy finder written in Rust and MIT-licensed. Its README opens with a distinction invisible in most tools:

**Note from the skim README**

*"Skim can be used as a library in your Rust project. The library allows you to embed the fuzzy-finder TUI directly in your binary and receive typed, structured results back — no subprocess, no pipe, no string serialization."*

This changed everything. Not a subprocess to spawn but a *trait to implement.*

### 1.3 The `SkimItem` Trait as Architectural Foundation

The `SkimItem` trait has four methods. Each one became a design decision that propagated through every layer of KubeRift:

```
pub trait SkimItem: Send + Sync + 'static {
    fn text(&self) -> Cow<str>;           // what skim fuzzy-matches against
    fn display(&self, ctx: DisplayContext) -> AnsiString; // colored list row
    fn preview(&self, ctx: PreviewContext) -> ItemPreview; // right-hand pane
    fn output(&self) -> Cow<str>;         // machine-parseable selection result
}
```

**`text()`** determined the search schema. To make a pod searchable by name, namespace, kind, status, and age simultaneously, all those fields had to be concatenated into a single string. This forced the design of `K8sItem` as a self-contained value type carrying all metadata.

**`display()`** established the visual contract. Because skim calls this on every render frame, the color-coding logic (red for critical, yellow for warning, green for healthy) had to be fast and allocation-minimal. The `StatusHealth` enum and its `classify()` function emerged directly from this requirement.

**`preview()`** revealed the threading model. Skim calls `preview()` from a *background thread* whenever the cursor moves. This is a blocking call — perfect for synchronous `kubectl` invocation. But supporting three preview modes (describe / yaml / logs) meant the preview function needed to read external state. The solution: a tiny file in `$XDG_RUNTIME_DIR/<pid>/preview-mode` containing a single digit. The `ctrl-p` binding writes a new digit to that file and signals skim to refresh. Clean, race-free, zero shared memory.

**`output()`** defined the action dispatch protocol. The string `"pod/production/api-server/my-context"` became the canonical identifier threaded through every action function. The double-dash separator (`--`) before resource names in every kubectl invocation is a direct consequence of needing to reconstruct safe command lines from this identifier.

### 1.4 The Channel as the System's Spine

The second architectural insight was skim's `SkimItemSender`/`SkimItemReceiver` channel pair:

```
let (tx, rx): (SkimItemSender, SkimItemReceiver) = unbounded();

tokio::spawn(async move {
    // K8s watcher sends items as they arrive from the API
    while let Some(event) = watcher.next().await {
        tx.send(Arc::new(K8sItem::from(event))).ok();
    }
});

// Skim consumes from rx -- items appear as the cluster streams them
let output = Skim::run_with(&options, Some(rx));
```

This is a **multi-producer, single-consumer** pattern. For multi-cluster mode, it becomes *multi-producer*: one tokio task per kubeconfig context, all sending into the same channel, all items converging in a single skim session. The architecture handles one cluster and twenty clusters identically — the channel is the only integration point.

### 1.5 The Key Dispatch Pattern

Skim's `SkimOutput` returns a `final_key` field — whichever key ended the session. KubeRift registers every action key with the `accept` binding rather than `execute`. This means control always returns to Rust:

```
match output.final_key {
    Key::Enter     => action_describe(&items).await?,
    Key::Ctrl('l') => action_logs(&items).await?,
    Key::Ctrl('e') => action_exec(&items[0]).await?,
    Key::Ctrl('d') => action_delete(&items).await?,
    Key::Ctrl('f') => action_portforward(&items[0]).await?,
    Key::Ctrl('r') => action_rollout_restart(&items).await?,
    Key::Ctrl('y') => action_yaml(&items).await?,
    Key::Ctrl('x') => { /* context switch, relaunch */ }
    _ => {}
}
```

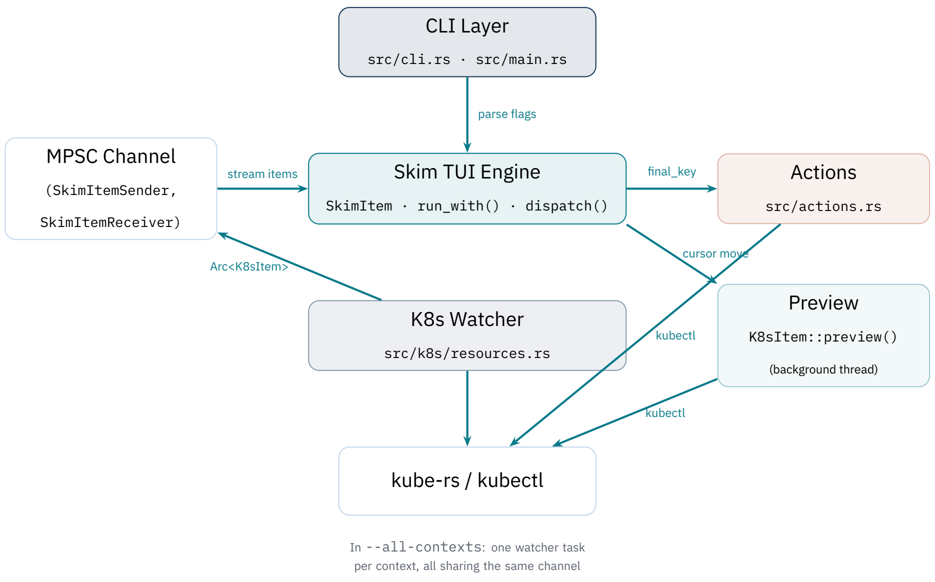
After every action, skim relaunches with a fresh channel but the same configuration. The user sees the live cluster list again, uninterrupted. This loop-after-action design means KubeRift feels more like a shell than an application — you run commands, you return to the prompt.

**The founding insight in one sentence:** Skim's `SkimItem` trait, its MPSC item channel, and its `final_key` dispatch pattern collectively determined KubeRift's data model, threading architecture, preview mechanism, and action system — before a single line of Kubernetes code was written.

## 2 System Architecture

KubeRift is a single Rust binary structured in three layers that communicate through well-defined interfaces.

### 2.1 Layer Map



### 2.2 Component Responsibilities

**CLI Layer** (`src/cli.rs`, `src/main.rs`) Parses arguments via Clap, resolves the kubeconfig, determines single vs. multi-cluster mode, constructs the skim options (header text, keybinding strings, preview window placement), and drives the outer `loop { skim → dispatch → relaunch }`.

**K8s Watcher** (`src/k8s/resources.rs`, `src/k8s/client.rs`) Uses `kube::runtime::watcher` to stream `Applied`/`Deleted` events for all thirteen resource kinds concurrently. A coordinator task collects the initial batch, sorts by health priority (unhealthy first), and flushes to the channel. Subsequent live events are forwarded immediately.

**Skim TUI Engine** The embedded fuzzy-finder. Renders the list, handles keyboard input, calls `display()` on every frame and `preview()` on cursor movement. Returns a `SkimOutput` to Rust.

**Actions** (`src/actions.rs`) Pure functions that receive a slice of `Arc<K8sItem>` and shell out to `kubectl`. All commands use `--` before resource names (argument injection prevention). Dangerous operations include interactive confirmation.

**Preview Thread** Skim's internal background thread calls `K8sItem::preview()`, which reads the mode file and runs `kubectl describe`, `kubectl get -o yaml`, or `kubectl logs`.

### 2.3 Unhealthy-First Ordering

Health classification drives sort priority:

| Status examples | Priority | Display color |
|---|---|---|
| CrashLoopBackOff, OOMKilled, Error, ImagePullBackOff | 0 — Critical | **Red** |
| Pending, NotReady, Degraded (N/M), Terminating | 1 — Warning | **Yellow** |
| Running, Ready, Bound, Active, Scheduled | 2 — Healthy | **Green** |
| `[DELETED]` | 1 — Unknown | **Dark gray** |

Skim renders higher-indexed items at the *top* of the list. The watcher sends critical items *last* in the initial batch so they receive the highest indices and surface automatically without any filter.

## 3 Installation & First Run

### 3.1 Install Methods

Cargo (all platforms):

```
cargo install kuberift
```

Homebrew (macOS and Linux):

```
brew tap syedazeez337/kuberift
brew install kf
```

Arch Linux (AUR):

```
yay -S kuberift          # or paru -S kuberift
```

**Pre-built binaries:** Download from github.com/syedazeez337/kuberift/releases. Tarballs include the binary, shell completions, and a man page for Linux x86_64, Linux aarch64, macOS ARM, and macOS Intel.

### 3.2 Shell Completions

```
# Bash
kf --completions bash >> ~/.bash_completion

# Zsh
kf --completions zsh > "${fpath[1]}/_kf"

# Fish
kf --completions fish > ~/.config/fish/completions/kf.fish
```

### 3.3 First Run

```
kf              # Opens the TUI against your current kubeconfig context
kf pods         # Opens showing only pods
kf --read-only  # Opens in read-only mode (no delete/exec/port-forward)
```

If no cluster is reachable, KubeRift drops into **demo mode** automatically and displays eleven sample resources so you can explore the interface without a live cluster.

### 3.4 Command-Line Reference

| Flag / argument | Behaviour |
|---|---|
| `[RESOURCE]` | Optional. Restrict to a single resource type. Accepts aliases: `po`/`pod`/`pods`, `svc`/`service`, `deploy`/`deployment`, `sts`/`statefulset`, `ds`/`daemonset`, `cm`/`configmap`, `secret`, `ing`/`ingress`, `no`/`node`, `ns`/`namespace`, `pv`, `pvc`, `job`, `cj`/`cronjob`. Unknown strings produce a warning and fall back to all resources. |
| `--context` *NAME* | Use a specific kubeconfig context instead of the current or last-saved one. |
| `--all-contexts` | Stream every kubeconfig context simultaneously in parallel. Each item is prefixed with its cluster name (color-coded). Actions pass `--context` to kubectl automatically. |
| `-n, --namespace` *NS* | Restrict to a specific namespace. Cluster-scoped kinds (Node, Namespace, PersistentVolume) are unaffected and always appear. |
| `-l, --label` *SEL* | Kubernetes label selector, e.g. `app=backend`, `env in (prod,staging)`, `!canary`. |
| `--kubeconfig` *PATH* | Alternate kubeconfig file path. Defaults to `$KUBECONFIG` or `~/.kube/config`. |
| `--read-only` | Disable exec, delete, port-forward, and rollout-restart. Describe, logs, and YAML remain available. Header shows `[READ-ONLY]`. |
| `--completions` *SHELL* | Print shell completions to stdout and exit. Values: `bash`, `zsh`, `fish`. |

| Flag / argument | Behaviour |
|---|---|
| `--mangen` | Print man page source to stdout and exit. |
| `--version` | Print version and exit. |

## 3.5 Configuration Files

KubeRift stores minimal persistent state:

| Path | Purpose |
|---|---|
| `~/.config/kuberift/last_context` | Last-used context name, restored on next launch. Written with `0o600` permissions. |
| `$XDG_RUNTIME_DIR/`*`pid`*`/preview-mode` | Preview mode digit (0/1/2). Per-PID subdirectory prevents symlink attacks. Deleted on exit. |
| `$XDG_RUNTIME_DIR/`*`pid`*`/preview-toggle` | Shell script cycled by `Ctrl-P`. Permissions: `0o700`. |

Falls back to `/tmp/kuberift-<`*`pid`*`>/` when `$XDG_RUNTIME_DIR` is unavailable.

## 4 The TUI Interface

### 4.1 Layout

The TUI occupies 60% of the terminal height. The left pane shows the fuzzy-filtered resource list; the right pane (50% of width) shows the live preview for the highlighted item.

```
KubeRift  ctx:production  res:all              [ctrl-l logs  ctrl-e exec  ctrl-d delete
    ...]
> _                                           | Name:     api-server-abc
  pod  production/api-server-abc  Running 2d  | Namespace: production
  pod  production/worker-xyz      Running 2d  | Status:    Running
  pod  staging/frontend           Pending 5m  | Containers:
  svc  production/api-service     ClusterIP   |   api-server (running)
  deploy  production/backend      2/3     1h  | ...
```

The header line shows the active context, resource filter, and available key bindings. In `--all-contexts` mode it shows `ctx:all`.

### 4.2 Keyboard Reference

| Key | Action | Notes |
|---|---|---|
| Type any text | Fuzzy filter | Matches name, namespace, kind, status, age simultaneously |
| ↑ / ↓ | Move cursor | |
| `Tab` | Toggle selection | Multi-select; selected items highlighted |
| `Esc` | Quit | |
| `Enter` | Describe | `kubectl describe`; multi-select supported |
| `Ctrl-L` | Logs | `kubectl logs --tail=200`; pods only; multi-select |
| `Ctrl-E` | Exec shell | Interactive `kubectl exec -it`; pods only; single item |
| `Ctrl-D` | Delete | Confirmation prompt; `>10` items requires typing `yes`; multi-select |
| `Ctrl-F` | Port-forward | Prompts for local/remote ports; pods and services; single item |
| `Ctrl-R` | Rollout restart | `kubectl rollout restart` + status; deploys/sts/ds; multi-select |
| `Ctrl-Y` | Print YAML | `kubectl get -o yaml`; multi-select |
| `Ctrl-P` | Cycle preview | Describe → YAML → Logs → (repeat) |
| `Ctrl-X` | Switch context | Opens secondary fuzzy picker listing all kubeconfig contexts |

### 4.3 List Item Format

Each row in the list follows a fixed-width layout:

```
[kind:8]  [ctx/][ns/][name:31]  [status:17]  [age]
```

- **Kind** — 8 characters, left-aligned. Color per resource type (blue for services, yellow for deployments, magenta for configmaps/secrets, etc.)
- **Context prefix** — Shown only in `--all-contexts` mode, color-coded deterministically by cluster name hash.
- **Namespace** — Cyan; omitted for cluster-scoped resources (Node, Namespace, PV).
- **Name** — Truncated to 31 characters with an ellipsis (…) if longer.
- **Status** — Right-aligned to 17 characters. Color reflects health (red/yellow/green/gray).
- **Age** — Human-readable: `5m`, `2h`, `1d`. Dark gray.

## 4.4 Preview Modes

| Mode | Key sequence | Command | Output |
|---|---|---|---|
| 0 (default) | First `Ctrl-P` | `kubectl describe <kind> -n <ns> -- <name>` | Full description with events, conditions, labels |
| 1 | Second `Ctrl-P` | `kubectl get <kind> -o yaml ...` | Complete YAML manifest |
| 2 | Third `Ctrl-P` | `kubectl logs --tail=100 ...` | Last 100 log lines (pods only; error message otherwise) |

The mode persists across cursor movements and resets to 0 when skim relaunches after an action.