kaiv for YAML Users

A Guide

The kaiv Project

for kaiv 0.5.0

1 Introduction

You write YAML — for services, pipelines, deployments, playbooks. This guide maps YAML onto **kaiv**, a text format with none of YAML's ambient hazards: no implicit typing, no indentation sensitivity, no anchors that only some parsers resolve. Every kaiv line carries its own type, its own full address, and its own value, and a schema layer travels with the format instead of living in a separate linter.

kaiv covers YAML's data model completely — YAML is a superset of JSON, kaiv covers JSON at 100%, and the YAML-specific surface (anchors, merge keys, block scalars, tags) maps or rides a typed escape hatch. Conversion is a hub, not a migration: what comes in from YAML can leave as JSON, XML, CBOR, TOML, or the binary schema formats, and YAML remains your authoring surface for as long as you want it.

Every example is a verified transcript of the reference `kaiv` tool (`cargo install kaiv-cli`), run against the fixture files shipped beside this document.

2 Hello, kaiv

A service list, the shape you deploy every week:

```
region: eu-west
replicas: 3
services:
  - name: web
    port: 8080
    public: true
  - name: db
```

```
    port: 5432
    public: false
```

Import, then build to canonical form:

```
$ kaiv import app.yaml | kaiv build
.!daiv
!str'::region=eu-west
!int'::replicas=3
!str'/@services/0::name=web
!int'/@services/0::port=8080
!bool'/@services/0::public=true
!str'/@services/1::name=db
!int'/@services/1::port=5432
!bool'/@services/1::public=false
```

One line per field: `!type ' address = value`. The sequence became an indexed `@`-array; nesting became path segments. There is no indentation to preserve — each line is self-contained, movable, and diffable on its own. The only structural characters are `'` and `=`.

3 The Mapping

YAML	kaiv	
mapping	namespace	`/services...`
sequence	`@`-array	`/@services/0::name=web`
scalar types	explicit `!type`	`!int`, `!bool`, `!null`
~ / empty	`!null`	`!null` then `tilde=`
anchors & aliases	resolved at import	(values inlined)
merge key `<<:`	resolved at import	overrides honored
block scalars	`!text`	`|:|`-joined lines
tags (`!!str`)	type annotations	`!str`
multi-document stream	not covered	one document per file

4 The End of Implicit Typing

YAML's most famous hazard: under YAML 1.1 rules, `NO` means *false* — ask Norway. `yes`, `on`, `0o77`, `1e3`, and bare dates each silently become something other than a string in some parser somewhere, and two YAML consumers routinely disagree.

kaiv's importer reads YAML 1.2 (the strict rules), and what comes out the other side is *pinned*:

```
countries:
  - SE
  - NO
  - DK
answers:
  yes_11: yes
  on_11: on
  norway: NO
version: 1.0
octal: 0o77
big: 1e3
date: 2001-01-23
tilde: ~
```

```
$ kaiv import norway.yaml
.!kaiv

/@countries;=SE;NO;DK
/answers:=yes_11=yes|on_11=on|norway=NO
!float
version=1.0
!int
octal=63
!float
big=1e3
date=2001-01-23
!null
tilde=
```

`NO` is the string `NO`, in the country list and in the mapping. `yes` and `on` are strings. The octal literal resolved to `63` *once, at import* — and is an explicit `!int` forever after. The date stayed a string; nothing invents a datetime behind your back. After conversion there is no second parse and no second opinion: the type is on the line.

5 Anchors Resolve, Results Stay Flat

Anchors, aliases, and merge keys are YAML's reuse mechanism — and a classic source of parser disagreement (merge keys are YAML 1.1 only). kaiv resolves them at import, overrides included:

```
defaults: &base
  retries: 3
  timeout: 30
prod:
  <<: *base
  timeout: 60
staging:
  <<: *base
```

```
$ kaiv import anchors.yaml
.!kaiv

!int
/defaults:=retries=3|timeout=30
!int
/prod:=retries=3|timeout=60
!int
/staging:=retries=3|timeout=30
```

`prod` kept its override (`timeout=60`); `staging` inherited the base. The canonical result carries no references at all — nothing left to resolve, ever again. (For authoring reuse, kaiv has its own variables — `.name` definitions, `$.name` references, whole-namespace templates — which likewise resolve away at build time.)

6 Multiline Strings Stay Readable

kaiv values are single-line by design — that is what keeps the grammar regular and the validator certifiable. Multi-line text still travels readably: the `!text` type carries it as one line, the text's lines joined by the fixed separator `|:|`. A YAML block scalar imports as exactly that:

```
$ printf 'motd: |\n  line one\n  line two\n' | kaiv import
   ↪  --yaml
.!kaiv

!text
motd=line one|:|line two|:|
```

Each separator is one newline — the trailing `|:|` is the block scalar's trailing newline — and exporters resolve them back to real line breaks.

Content the separator cannot carry (a CR, a literal `|:|`) still embeds via the `std/enc` channel, base64url-armored — lossless, though no longer readable in place. Where multiline text dominates as laid-out paragraphs, YAML remains the better authoring surface.

7 One Document per File

YAML streams several documents through one file with `---` separators. kaiv deliberately does not — a document is a file, and boundaries belong to the filesystem or the archive, not the byte stream:

```
$ printf -- '---\na: 1\n---\nb: 2\n' | kaiv import --yaml
kaiv: expected one YAML document, found 2
```

Loud refusal, not silent truncation. Split the stream, import each document.

8 The Round Trip

Back out to YAML, structure intact:

```
$ kaiv import app.yaml | kaiv build > app.daiv
$ kaiv export --yaml app.daiv
---
region: eu-west
replicas: 3
services:
  - name: web
    port: 8080
    public: true
  - name: db
    port: 5432
    public: false
```

And because the canonical form is format-neutral, the same `app.daiv` exports to JSON, XML, CBOR, TOML, and the rest — see the sibling guides.

9 A Schema from Your Example

YAML has no schema language of its own — validation lives in external linters, if anywhere. kaiv infers a schema from the document you already have:

```
$ kaiv infer --name acme/app app.yaml
.!saiv 1 acme/app

region=
!int
replicas=
[/@services]
name=
!int
port=
!bool
public=
[]
```

The `[/@services] ... []` block declares the array's element shape: every element must carry `name`, `port`, `public`, in order, with those types. Break the contract and the error names everything:

```
$ kaiv infer --name acme/app app.yaml > app.saiv
$ sed 's/public: false/public: maybe/' app.yaml >
   ↪  broken.yaml
$ kaiv import broken.yaml | kaiv build > broken.daiv
$ kaiv validate broken.daiv app.saiv
kaiv: ConstraintViolationError: /@services/1::public=maybe
   ↪  (type !str) violates {true,false} (line 9)
```

Field, value, violated constraint, line. Exit code 1; `pass` and exit 0 otherwise.

10 The Constraint Is the Type

One more YAML classic: quote a number, and it is a different value in every typed consumer. Watch what kaiv does with a quoted port:

```
$ sed 's/port: 5432/port: "5432"/' app.yaml > quoted.yaml
$ kaiv import quoted.yaml | kaiv build > quoted.daiv
$ grep 'port' quoted.daiv
!int'/@services/0::port=8080
!str'/@services/1::port=5432
$ kaiv validate quoted.daiv app.saiv
pass
```

The quoted port imported as a string — and still *validates* against the schema's `!int` field. That is kaiv's structural typing at work: `str` is the only primitive, and `!int` is a *refinement* of it — a pattern and a numeric ordering. The serialization 5432 satisfies the refinement, so it *is* an int, whatever annotation the importer stamped. The type system judges the value, not the label — one more ambiguity that stops mattering after the hub.

11 When to Stay with YAML

An honest map marks the roads not taken. YAML remains the right choice when:

- your tools *are* YAML — Kubernetes, Compose, CI pipelines, Helm, Ansible read it natively and their ecosystems assume it;
- multiline prose dominates the document (literal blocks stay laid out as paragraphs; kaiv's `!text` folds them onto one line);
- you rely on multi-document streams as a wire format.

kaiv's pull grows with what surrounds the data: explicit types, schemas inferred and enforced, validation errors that name the field, units, provenance, a canonical form that diffs line-by-line — and conversion back out at any moment, so the door never locks behind you.

12 Where Next

From here:

- the kaiv specification — the formal grammar and semantics;

- `kaiv help` — the full toolchain surface;
- the sibling guides — kaiv for JSON, TOML, XML, Protocol Buffers, Avro, CBOR, ASN.1, and GraphQL users; the JSON guide covers JSON Schema conversion, which applies to YAML documents unchanged.