kaiv for Avro Users

A Cookbook

The kaiv Project

for kaiv 0.4.0

1 Introduction

You use Avro — records with schemas, logical types, compact binary files flowing through Kafka topics and data lakes. This cookbook maps Avro onto **kaiv**, which shares Avro's core belief — data travels with its contract — and inverts the economics: where an Avro datum is unreadable without its schema, every kaiv canonical line is self-describing text, and where Avro Schema stops at structure, kaiv constrains *values* — ranges, patterns, enums, lengths.

Conversion runs both ways: your `.avsc` converts to a kaiv schema, your data moves between Avro object container files and canonical text losslessly, and the pipeline keeps its binary files where binary earns its keep.

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 Your Schema Converts

A record with the shapes Avro pipelines lean on — a logical type, the `["null", ...]` optionality idiom, a decimal:

```
{
  "type": "record",
  "name": "Reading",
  "fields": [
    {"name": "sensor", "type": "string"},
    {"name": "celsius", "type": "double"},
```

```
    {"name": "taken", "type": {"type": "long", "logicalType":
       ↪  "timestamp-millis"}},
    {"name": "note", "type": ["null", "string"], "default":
       ↪  null},
    {"name": "price", "type": {"type": "bytes", "logicalType":
       ↪  "decimal", "precision": 9, "scale": 2}}
  ]
}
```

```
$ kaiv import-schema --avro --name acme/reading
   ↪  reading.avsc
.!kaivschema 1 acme/reading
.!types std/num
.!types std/time

sensor=
!float|std/num/inf|std/num/nan
celsius=
&datetime
taken=
!null|str
note=
!float
price=
```

Reading it back: `timestamp-millis` became `&datetime` from `std/time` — a named type, not a bare long, so the value reads as a timestamp *and* validates as one. The `["null","string"]` idiom became the tagged union `!null|str` — the same idea, one token. And `double` became the extended-reals union: the Avro spec admits `NaN` and the infinities in a `double`, so a faithful conversion must too — `std/num` names them instead of pretending they cannot arrive.

3 Records In, Records Out

kaiv writes and reads Avro *object container files* — the self-contained kind, schema embedded in the header:

```
$ printf '{"sensor":"t-01","celsius":21.5,"taken":"2026-0
   ↪  7-17T09:30:00Z","note":null,"price":"19.99"}' |
   ↪  kaiv import --json | kaiv build > reading.daiv
$ kaiv export --avro reading.daiv | wc -c
```

```
317
$ kaiv export --avro reading.daiv | kaiv import --avro |
   ↪  kaiv build
.!kaiv 1
!str'::sensor=t-01
!float'::celsius=21.5
!str'::taken=2026-07-17T09:30:00Z
!null'::note=
!str'::price=19.99
```

Three hundred seventeen bytes of container — header, embedded schema, datum — and every value returns intact, the null included as an explicit `!null` line. The active-variant rule governs unions in canonical form: the data line's type annotation says which alternative this value took, so `note` is `!null` when absent and `!str` when present — Avro's union branch index, readable.

4 Validation with Teeth

The converted schema validates canonical documents directly — and because `taken` is `&datetime`, not a bare long, malformed timestamps have nowhere to hide:

```
$ kaiv import-schema --avro --name acme/reading
   ↪  reading.avsc > reading.saiv
$ kaiv validate reading.daiv reading.saiv
pass
$ printf '{"sensor":"t-01","celsius":21.5,"taken":"yester」
   ↪  day","note":null,"price":"19.99"}' | kaiv import
   ↪  --json | kaiv build > bad-taken.daiv
$ kaiv validate bad-taken.daiv reading.saiv
kaiv: ConstraintViolationError: ::taken=yesterday (type
   ↪  !str) violates /^\d{4}-\d{2}-\d{2}[Tt
   ↪  ]\d{2}:\d{2}:\d{2}(\.\d+)?([Zz]|[+-]\d{2}:\d{2})$/
   ↪  ..time (line 4)
```

The error carries `std/time/datetime`'s full compiled pattern and temporal span — field, value, constraint, line. Exit code 1; `pass` and exit 0 otherwise. From here the schema is a kaiv schema: tighten it with ranges and patterns Avro Schema has no syntax for.

5 The Mapping

Avro	kaiv	
`record`	namespace / document	ordered in both
`["null", T]`	`!null|T` union	the active variant annotates
`enum`	enum constraint	`{a,b,c}`
`map`	`!map<T>`	
`array`	`@`-array	
`fixed` / `bytes`	`!b64` / `&bin`	base64url
logical types	`std/time` named types	`&datetime`, `&date`, `&uuid`
`decimal`	`!float`	exact decimals on export
`double`	extended-reals union	NaN/Inf admitted, as in Avro
`aliases`	mapping documents	a graph edge, not an attribute
`doc`	`//` comments	

6 When to Stay with Avro

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

- Kafka, Hadoop, or Spark is the backbone — schema registries, splittable container files, and per-block compression are ecosystem machinery kaiv does not reproduce;
- reader/writer schema resolution is load-bearing — Avro negotiates between schema versions at read time; kaiv handles evolution through mapping documents instead;
- per-field sort order in the binary encoding matters to your jobs.

kaiv's pull is at the edges of the pipeline: the sample a human inspects, the fixture a test pins, the record an audit must keep readable for a decade, the contract that needs a range and not just a type — with the container round trip above as the bridge, both directions, byte-faithful.

7 Where Next

From here:

- the kaiv specification — the formal grammar and semantics;
- `kaiv help` — the full toolchain surface;

- the sibling cookbooks — kaiv for JSON, YAML, TOML, XML, Protocol Buffers, CBOR, ASN.1, and GraphQL users.