

kaiv for Protocol Buffers Users

A Cookbook

The kaiv Project
for kaiv 0.4.0

1 Introduction

You use Protocol Buffers — schema-first, compact on the wire, generated everywhere. This cookbook maps proto onto **kaiv**, which shares proto’s convictions (the schema is the contract; data without one is a liability) and departs on one axis: kaiv’s canonical form is *self-describing text*. Every line carries its own type and address, so no schema is needed to *read* data — only to validate it. And where proto stops at structure, kaiv adds what `.proto` cannot express: value constraints — ranges, patterns, enums with teeth — plus units and provenance.

Conversion is a hub, not a migration: your `.proto` converts to a kaiv schema, your wire messages convert to canonical text and back byte-for-byte, and gRPC keeps speaking proto on the wire.

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 Comes First

A proto3 message with the usual furniture — scalar fields, a repeated field, a oneof:

```
syntax = "proto3";

message User {
  string name = 1;
  uint32 id = 2;
  bool active = 3;
  repeated string roles = 4;
```

```

    oneof contact {
        string email = 5;
        uint64 phone = 6;
    }
}

```

Convert it:

```

$ kaiv import-schema --proto --name acme/user user.proto
.!kaivschema 1 acme/user

name?=  
!null|int[0,4294967295]  
id?=  
!null|bool  
active?=  
/@roles;=  
email?=  
!null|int[0,18446744073709551615]  
phone?=

```

Reading it back: `uint32` and `uint64` carry their *exact* ranges as value constraints. Every `proto3` field is optional (`?`) — `proto3` semantics, where absence is unobservable — and each optional non-string takes a `!null` alternative so absence has an explicit spelling. The `oneof` members arrive as two optional fields; exclusivity is the one thing this schema does not enforce — a sound weakening, since the wire cannot carry both anyway.

3 The Wire Round Trip

Proto wire data is not self-describing — field numbers, no names, no types. `kaiv` reads and writes it with the schema at hand (`--schema`, and `--message` when the file declares several). Build a document, take it through the binary wire, and back:

```

$ printf '{"name":"Ada","id":7,"active":true,"roles":["ad  
  ↪ min","dev"],"email":"ada@example.com"}' | kaiv  
  ↪ import --json | kaiv build > user.daiv  
$ kaiv export --proto --schema user.proto --message User  
  ↪ user.daiv > user.pb  
$ xxd user.pb

```

```

00000000: 0a03 4164 6110 0718 0122 0561 646d 696e
    ↪ ..Ada....".admin
00000010: 2203 6465 762a 0f61 6461 4065 7861 6d70
    ↪ ".dev*.ada@examp
00000020: 6c65 2e63 6f6d                                le.com
$ kaiv import --proto --schema user.proto --message User
    ↪ user.pb | kaiv build
.!kaiv 1
!str'::name=Ada
!int'::id=7
!bool'::active=true
!str'/@roles::0=admin
!str'/@roles::1=dev
!str'::email=ada@example.com

```

Thirty-eight bytes of wire, reconstructed field for field. Note the contrast in kind: the `.pb` needs `user.proto` to mean anything; the `.daiv` above it is readable in a text editor in a decade with nothing else on disk.

4 Absence, Materialized

proto3 made field absence unobservable — a missing field and a default value are indistinguishable, a periodic source of bugs. `kaiv` takes the opposite stance: when a document declares its schema, the build *materializes* every absent optional field explicitly.

A `kaiv` document authored against the converted schema — `.!registry` points schema resolution at the local directory, `.!schema` declares the contract:

```

.!kaiv 1
.!registry acme=.
.!schema:acme/user

name=Ada
!int
id=7
!bool
active=true
/@roles;=admin;dev
email=ada@example.com

```

```

$ kaiv import-schema --proto --name acme/user user.proto >
    ↪ user.saiv

```

```

$ mkdir -p acme
$ kaiv schema user.saiv > acme/user.csaiv
$ kaiv build ada.kaiv
.!kaiv 1
.!registry acme=.
.!schema:acme/user
!str'::name=Ada
!int'::id=7
!bool'::active=true
!str'/@roles::0=admin
!str'/@roles::1=dev
!str'::email=ada@example.com
!null'::phone=

```

The one of branch that was not taken — phone — appears as an explicit `!null` line. Nothing about this document's shape depends on knowing proto3's default rules: absence is spelled out, on the line, and the strict-lockstep validator accepts it:

```

$ kaiv build ada.kaiv > ada.daiv
$ kaiv validate ada.daiv user.saiv
pass

```

5 Constraints .proto Cannot Express

Proto has no value constraints — no ranges beyond the integer widths, no patterns, no bounded strings. Your converted schema is a kaiv schema now, so it can say more. Require names to be capitalized single words:

```

$ sed -i 's/^[name]?=#!null|str/^[A-Z][a-z]+$/$/\nname?=# '
↪ user.saiv
$ kaiv validate ada.daiv user.saiv
pass
$ sed 's/name=Ada/name=ada lovelace/' ada.kaiv >
↪ lower.kaiv
$ kaiv build lower.kaiv > lower.daiv
$ kaiv validate lower.daiv user.saiv
kaiv: ConstraintViolationError: ::name=ada lovelace (type
↪ !str) violates !null(/^[^$)/|str(/^[A-Z][a-z]+$/)
↪ (line 4)

```

Field, value, the compiled union with each alternative’s constraint group, and the line. One detail is load-bearing: the tightened alternative is `!null|str/.../` — had we put a bare pattern on the optional field, the schema compiler would have rejected it with `SchemaOptionalWithoutDefaultError`, because an optional field whose default no longer satisfies its constraints and whose type admits no null has *no representable absent state*. `kaiv` makes you say what absence means before it lets you require anything else.

6 The Mapping

Protocol Buffers	kaiv	
message	namespace / document	
nested message	path segment	/outer/inner::field=
field number	field order	position, not tag
repeated	@-array	/@roles;=
oneof	optional members	exclusivity not enforced
map<K,V>	!map<T>	
enum	enum constraint	{a,b,c}
uint32 etc.	exact ranges	!int[0,4294967295]
bytes	!b64	base64url
proto3 optionality	?= + null	absence explicit
reserved	—	renames via mappings

7 When to Stay with Proto

An honest map marks the roads not taken. Protocol Buffers remain the right choice when:

- gRPC is your transport — the RPC framework, streaming, and deadline machinery assume proto end to end;
- generated code is the point — per-language classes with IDE completion across a large polyglot codebase;
- schema evolution by field number is load-bearing — proto’s renumber-nothing discipline lets very old readers skip very new fields; `kaiv` orders fields by position and handles renames through mapping documents instead;
- every byte on the wire is billed — proto’s varint packing beats text for integer-heavy payloads.

kaiv’s pull grows at the edges of that world: configuration and documents humans read, data that outlives the services that produced it, contracts that need real value constraints, audits that need provenance — with the wire round trip above as the bridge, in both directions, whenever you need it.

8 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, Avro, CBOR, ASN.1, and GraphQL users.