

The kaiv User Manual

from .env to typed, validated data

The kaiv Project
for kaiv 0.5.0

Contents

I	First Session	2
1	You Already Write kaiv	3
2	Types	3
3	Structure	4
4	The Pipeline	6
5	Variables	7
6	Schemas	7
7	Units, Libraries, and the Registry	8
8	Provenance	9
9	Where Next	9
II	The Language	9
10	Values, Names, and Comments	10
11	Types in Depth	11
12	Structure in Depth	12

13 Variables in Full	12
III Contracts	13
14 Requiredness, Defaults, and Materialization	13
15 Strict Schemas	14
16 The Constraint Vocabulary	14
17 Tables: Uniqueness, References, Cardinality	15
18 Your Own Type Library	16
19 The Standard Libraries	17
20 Units	17
21 Provenance in Full	18
IV The Toolchain	18
22 The Command Surface	18
23 Interop	19
24 Errors and the Conformance Ladder	19
25 A Worked Finish	19

Part I

First Session

Part I is a complete first session: by its end you will have typed data, structure, schemas, and provenance under your fingers. The rest of the manual deepens each layer — stop at the end of this part and you already write useful kaiv.

1 You Already Write kaiv

Here is a fact that makes kaiv the easiest format you will ever adopt: **a strict .env file is already valid kaiv**. Take one — keys, values, a comment, an empty value:

```
HOST=api.example.com
PORT=8443
# comment
EMPTY=
```

Build it with the kaiv toolchain (`cargo install kaiv-cli`), unmodified:

```
$ kaiv build app.env
.!daiv
!str':HOST=api.example.com
!str':PORT=8443
!str':EMPTY=
```

That output is *canonical* kaiv — *denormalized* kaiv, by its full name, saved in `.daiv` files (authored sources are `.kaiv`; the intermediate `.raiv` form and the pipeline that produces all three arrive in §4): one line per value, each carrying a type (`!str`), an address (`:HOST`), and the value, separated by `'` and `=`. No quotes, no escaping — a value runs verbatim to the end of the line. The line above them, `.!daiv`, is the *format declaration*: pipeline output always opens by naming its own kind. Authored input needs no such header — a file that carries no declaration *is* authored kaiv, which is exactly why the plain `.env` file passed unmodified (write `.!kaiv` up top when you prefer the kind explicit). “Strict” `.env` means the plain dialect: `KEY=value`, no export, no quoting, no spaces around `=`. That dialect is kaiv’s ground floor, and everything in this manual is built on top of it, one optional layer at a time.

2 Types

Every value so far was a string — `str` is kaiv’s default and only primitive. To say more, put an annotation on the line *above* a field:

```
HOST=api.example.com
!int
```

```
PORT=8443
!bool
TLS=true
!null
MOTD=
```

```
$ kaiv build typed.kaiv
.!daiv
!str': :HOST=api.example.com
!int': :PORT=8443
!bool': :TLS=true
!null': :MOTD=
```

Note `!null` versus the earlier `EMPTY=`: both have nothing after `=`, but one is a null and the other an empty string — the type is the discriminant, not the payload. And because every canonical line names its type, `grep` becomes a typed query:

```
$ kaiv build typed.kaiv > typed.daiv
$ grep "^!int" typed.daiv
!int': :PORT=8443
```

3 Structure

`.env` is flat. `kaiv` adds structure with *one* concept: the **namespace** — a generalized struct, a container of named things. What other formats treat as two different constructs are both namespaces here:

- an *object* (mapping, table, record) is a namespace whose keys *you* choose — `/server` holds host and port;
- an *array* is a namespace whose keys are *managed for you* — consecutive integers 0, 1, 2, assigned in order. The `@` prefix is what marks a namespace as array-shaped: `@tags` is a namespace with managed integer keys.

Because both are namespaces, one pair of operators covers all navigation — `/name` descends into any namespace, `::` projects a field (a leaf value) out of one — and there is no nesting syntax at all: structure lives in the path, not in braces. Four authored shapes cover everything:

```

# A namespace (struct): keys you choose, one field per line.
(/server)
host=api.example.com
!int
port=8443
()

# The same kind of thing, inline: the := struct assignment.
/limits:=alice=10|bob=20

# A scalar array: managed integer keys, the ;= vector form.
/@tags;=prod;eu

# An array of namespaces: one [/@mirrors] block per element.
[/@mirrors]
host=eu.example.com
[/@mirrors]
host=us.example.com
[]

```

```

$ kaiv build shape.kaiv
.!daiv
!str'/server::host=api.example.com
!int'/server::port=8443
!str'/limits::alice=10
!str'/limits::bob=20
!str'/@tags::0=prod
!str'/@tags::1=eu
!str'/@mirrors/0::host=eu.example.com
!str'/@mirrors/1::host=us.example.com

```

Structs first: the block form `(/server)...` opens a namespace and lists its fields, and the inline form `:=` writes a small one in a single line — two spellings of the same chosen-key namespace. Then the managed-key kind: the `;` vector for scalars, and one `[/@mirrors]` block per element (closed by `[]`) for an array of namespaces. Read the canonical lines through that lens: `/server::host` and `/limits::alice` both project a field from a chosen-key namespace — block or inline, the result is identical; `/@tags::0` projects from a managed-key one, where the index *is* the key, which is why an array element’s address looks exactly like any other field’s; and `/@mirrors/0::host` chains the two: descend into element 0 (itself a namespace), project `host`. All of it is authoring sugar: canonical form is always the flat, fully addressed lines you see — self-contained, movable, diffable.

4 The Pipeline

What `kaiv build` runs is a two-stage compile: authored `.kaiv` → compile → relational `.raiv` → denorm → canonical `.daiv`. The middle form exists for one reason: *field references*. Reference another field's value with `$path::field`:

```
| /db::host=db.internal  
| /app::db_url=$/db::host
```

```
$ kaiv compile refs.kaiv  
.!raiv  
!str'/db::host=db.internal  
!str'/app::db_url=$/db::host  
$ kaiv compile refs.kaiv | kaiv denorm  
.!daiv  
!str'/db::host=db.internal  
!str'/app::db_url=db.internal
```

The `.raiv` keeps the reference (for diffing and design tools); the `.daiv` resolves it — and each stage stamps its output with its own kind's declaration, `.!raiv` and `.!daiv`, so a canonical file always says what it is. Deployment artifacts are always `.daiv`: nothing left to resolve, ever.

The canonical forms are for machines; you never have to read one. `kaiv fmt` is the standard formatter: on an authored file it settles style questions the way `gofmt` does — per group of fields it picks the most readable of the equivalent syntaxes (a flat namepath line, an inline `:=/+:=` assignment within 72 columns, a `(...)/[...]` block), treating your blank lines as grouping hints and never touching values, order, or comments (`-w` rewrites in place, `--check` guards CI). And pointed at a canonical file, it renders the authored view back:

```
$ kaiv build shape.kaiv | kaiv fmt  
.!kaiv  
  
(/server)  
host=api.example.com  
!int  
port=8443  
(  
  
/limits:=alice=10|bob=20
```

```
/@tags;=prod;eu

/@mirrors+:host=eu.example.com
/@mirrors+:host=us.example.com
```

Structure, types, everything intact — only the authoring sugar the compiler resolved away (variables, references) does not come back. Whenever a `.daiv` lands, the readable form is one command away.

5 Variables

For repetition *within* a document, define a hidden variable with a leading dot and reference it with `$` — including mid-value:

```
.host=api.example.com

primary=$.host
backup=$.host
url=https://$.host/v1
```

```
$ kaiv build vars.kaiv
.!daiv
!str':primary=api.example.com
!str':backup=api.example.com
!str':url=https://api.example.com/v1
```

Variables resolve away at compile time — like the block sugar, they exist for the author, not the consumer. (A literal dollar is `$$`.)

6 Schemas

A schema is written in the same line grammar as data — fields with their annotations, values left empty — under a `!.saiv VERSION ID` header: unlike a data file, a schema carries an identity, so its declaration is required and names it. This one uses two named types from the standard `std/net` library and an optional field with a default:

```
!.saiv 1 manual/server
.!types std/net
```

```
&hostname
host=
&port
port=
!bool
tls?=true
```

Validate a conforming document, then a broken one:

```
$ kaiv build ok.kaiv > ok.daiv
$ kaiv validate ok.daiv server.saiv
pass
$ printf 'host=api.example.com\n!int\nport=70000\n' | kaiv
  ↪ build > bad.daiv
$ kaiv validate bad.daiv server.saiv
kaiv: ConstraintViolationError: ::port=70000 (type !int)
  ↪ violates /^-?[0-9]+$/ ..num [0,65535] (line 3)
```

The error names the field, the value, the compiled constraint (&port is an integer refined to `[0,65535]`), and the line. `?=` marks a field optional — when a document declares its schema, the build *materializes* absent optional fields from their defaults, so validated documents are always complete. You rarely write a first schema by hand: `kaiv infer` derives one from a document you already have, and you tighten from there. Constraints compose from three pieces — pattern `/re/`, range `[min,max]`, length `#[min,max]` — plus enums `{a,b,c}` and tagged unions `!null|str`.

7 Units, Libraries, and the Registry

Numeric types can carry a *unit* — part of the type, byte-compared, never silently converted — and the type libraries live at permanent addresses the toolchain resolves by itself (`t.kaiv.io` during alpha). A wrong-shaped value fails against the whole typed contract:

```
$ printf '!float:km/h\nspeed=fast\n' | kaiv build >
  ↪ sp.daiv
$ kaiv validate sp.daiv sp.saiv
kaiv: ConstraintViolationError: ::speed=fast (type
  ↪ !float:km/h) violates !float:km/h
  ↪ /^-?[0-9]*\.[0-9]+([eE][+-]?[0-9]+)?$/ ..num (line
  ↪ 2)
```


SI and common units are built in; custom units and currencies (!float:~EUR) come from .faiv libraries. The standard type libraries — std/core, std/enc, std/time, std/num, std/net, std/math — ship embedded in every implementation and need no lookup at all.

8 Provenance

Any value can say *who says so, and when* — a source declared once, referenced per line, with a timestamp:

```
.?lab https://lab.example.com/sensors
```

```
!float:mm?lab@20260717T090000Z  
rainfall=2.4
```

```
$ kaiv build rain.kaiv  
.!daiv  
.?lab https://lab.example.com/sensors  
!float:mm?lab@20260717T090000Z':::rainfall=2.4
```

One line, five attributions: type, unit, source, time, and address. A schema can *require* provenance (.!provenance:required), making lineage part of the contract rather than a convention.

9 Where Next

You now have the whole ladder: .env lines, types, structure, the pipeline, variables, schemas, units, and provenance. Parts II–IV deepen each rung; the [guides](#) map your other formats onto it; the [specification](#) is the law behind every rule used here.

Part II

The Language

10 Values, Names, and Comments

Three lexical rules carry most of kaiv’s ergonomics. First: *a value runs verbatim from = to the end of the line* — no quoting, no escaping, with a single exception: a literal dollar sign is written \$\$ (because \$ introduces references). Second: names are bare by default and *quoted only when they need characters a bare name cannot carry* (spaces, slashes, an =). Third: # starts a comment for humans, while // starts *doc metadata* — commentary addressed to tools and future readers of the authored file:

```
// The primary endpoint.  
host=api.example.com  
price=it costs $$5  
"listen on"=0.0.0.0
```

```
$ kaiv build lines.kaiv  
.!daiv  
!str':host=api.example.com  
!str':price=it costs $5  
!str':"listen on"=0.0.0.0
```

Both comment kinds serve the authored file and are gone from canonical output. Quoting, meanwhile, is not a stylistic choice — canonical form has *one spelling* per name, and a quoted name that could be bare normalizes:

```
$ printf '"port"=8443\n' | kaiv build  
.!daiv  
!str':port=8443
```

Formally speaking. One spelling per construct is a design invariant of canonical form: quoted-but-bare names, leading-zero indices, and alternative operator spellings all normalize or reject, so byte comparison is semantic comparison (specification, *Canonical Form*).

11 Types in Depth

Annotations do more than name a scalar. A *tagged union* offers alternatives, and the data line’s own type records which alternative it took — the *active variant*:

```
$ printf '!int|str\|na=42\n!int|str\|nb=hello\n' | kaiv
    ↪ build
.!daiv
!int':a=42
!str':b=hello
```

The same authored annotation, two canonical types — each line answers “which arm?” by itself, no schema needed at read time. Two more shapes complete the everyday toolkit. A *map* is a namespace whose keys are open — the schema constrains the value type, not the key set:

```
!.saiv 1 g/m

!map<int>
limits=
```

```
$ printf '!int\n/limits:=alice=10|bob=20\n' | kaiv build >
    ↪ m.daiv
$ kaiv validate m.daiv map.saiv
pass
```

And the *extended reals*: kaiv floats are deliberately finite (so numeric ordering is total and decidable), with the IEEE non-finite markers available as named types — a union admits them exactly where you mean to:

```
$ printf '!.types std/num\n&inf\nratio=inf\n' | kaiv build
    ↪ > inf.daiv
$ kaiv validate inf.daiv inf.saiv
pass
```

12 Structure in Depth

Part I showed the block forms; two more authored shapes round out structure. The *inline element* operator `+=` appends one namespace-array element per line — the most compact spelling for tabular data:

```
$ printf '/@servers+=host=a|port=1\n/@servers+=host=b|p_
    ↪   ort=2\n' | kaiv build
.!daiv
!str'/@servers/0::host=a
!str'/@servers/0::port=1
!str'/@servers/1::host=b
!str'/@servers/1::port=2
```

Arrays may also be *mixed* — element *i* can be a scalar (addressed `@name::i`) or a namespace (addressed `@name/i`); the operator before the index says which, per element. Indices are always managed: consecutive, zero-based, and spelled without leading zeros — `@a/01` is not an address but an error, one more one-spelling rule.

13 Variables in Full

Beyond the scalar variables of Part I, whole containers can be named. An *array variable* accumulates with `@.name+=` and splices with `$@.name`:

```
$ printf '@.regions+=eu\n@.regions+=us\n\n/@primary;=$@.r_
    ↪   egions\n/@backup;=$@.regions\n' | kaiv build
.!daiv
!str'/@primary::0=eu
!str'/@primary::1=us
!str'/@backup::0=eu
!str'/@backup::1=us
```

A *namespace variable* is defined with the map operator under a dotted name and splatted with `$/.name` — its pairs expand as if written at the reference point:

```
$ printf '/.base:=retries=3|timeout=30\n\n/api:=$/.base\n_
    ↪   /worker:=$/.base\n' | kaiv build
```

```

.!daiv
!str'/api::retries=3
!str'/api::timeout=30
!str'/worker::retries=3
!str'/worker::timeout=30

```

Splat means *expansion*, not merge: if you splat a base and then restate a field, the document simply contains both lines, and a schema will call the duplicate what it is. Like every variable, these are authoring machinery — canonical output carries no trace of them.

Part III

Contracts

14 Requiredness, Defaults, and Materialization

kaiv’s optionality model has one rule that surprises newcomers and then becomes their favorite guarantee: *validated documents are complete*. An optional field (?:) must say what absence means — a default on the right side, or a !null alternative — and when a document declares its schema, the build *materializes* the answer:

```

.!saiv 1 acme/app

host=
!int
retries?=3
!null|str
note?=

```

```

$ mkdir -p acme
$ kaiv schema app2.saiv > acme/app.csaiv
$ printf '!.!registry
    ↪ acme=.\n!.!schema:acme/app\n\nhost=h1\n' | kaiv
    ↪ build
.!daiv
!.!registry acme=.
!.!schema:acme/app
!str'::host=h1
!str'::retries=3

```

```
!null'::note=
```

The author wrote one line; the artifact carries three — the default materialized for `retries`, the explicit null for `note`. Consumers never re-derive defaults, and the validator’s scan is strict lockstep: every declared field, present, in order. (`.!registry acme=.` points schema resolution at the local directory — the same mechanism that reaches the real registries, aimed at `..`)

15 Strict Schemas

By default a schema is *relaxed*: fields it does not declare may ride along. Add `strict` to the header and the document may contain declared fields only:

```
..!saiv 1 g/s strict
```

```
name=
```

```
$ printf 'name=x\nextra=y\n' | kaiv build > ex.daiv
$ kaiv validate ex.daiv strict.saiv
kaiv: UndefinedFieldStrictSchemaError: ::extra is not
    ↪ defined in the strict schema (line 3)
```

Formally speaking. Strictness is what GraphQL types and ASN.1 SEQUENCEs enforce implicitly; a `kaiv strict` schema is the same assertion made explicit (specification, *Strict vs. Relaxed*).

16 The Constraint Vocabulary

Every constraint is one of three components — pattern `/re/`, ordering (*span*), range `[min,max]` — plus enums, lengths `#[min,max]`, and unions. The span is what makes ranges mean the right thing. Compare version numbers under `..ver`, which orders *segment-wise numerically*:

```
..!saiv 1 g/v
/^[0-9.]+$ / ..ver [1.2,]
version=
```

```

$ printf '!str\nversion=1.10\n' | kaiv build > v.daiv
$ kaiv validate v.daiv ver.saiv
pass
$ printf '!str\nversion=1.1\n' | kaiv build > v3.daiv
$ kaiv validate v3.daiv ver.saiv
kaiv: ConstraintViolationError: ::version=1.1 (type !str)
    ↪ violates /^[0-9.]+$ / ..ver [1.2,] (line 2)

```

1.10 passes a [1.2,] floor and 1.1 fails it — lexical order would get both wrong. The other spans are ..num (numeric), ..lex (byte order, the default), and ..time (temporal); ranges are inclusive and either end may be omitted. Patterns use a deliberately finite dialect — no backtracking, no lookahead, and \xHH for the one character a pattern cannot carry literally (the apostrophe, which e-mail and URI grammars need).

17 Tables: Uniqueness, References, Cardinality

Arrays of namespaces can graduate to *tables*: the block header takes clauses for unique keys, foreign keys, and element counts — and validation gains a linear post-scan pass that enforces them:

```

.!saiv 1 g/fleet

[/@servers host=! min=1 max=10]
host=
!int
port=
[]
[/@disks server=/@servers/*::host]
server=
!int
size=
[]

```

Three contracts, three named refusals:

```

$ printf '/@servers+=:host=a|port=1\n/@servers+=:host=a|p
    ↪ ort=2\n' | kaiv build > dup.daiv
$ kaiv validate dup.daiv fleet.saiv
kaiv: UniquenessViolationError: /@servers: duplicate value
    ↪ (a) for unique key (host)

```

```
$ printf '/@servers+=:host=a|port=1\n/@disks+=:server=gho
↪ st|size=100\n' | kaiv build > fk.daiv
$ kaiv validate fk.daiv fleet.saiv
kaiv: ReferentialIntegrityError: /@disks: server=ghost has
↪ no match in /@servers/*::host
$ printf '/@disks+=:server=a|size=1\n' | kaiv build >
↪ empty.daiv
$ kaiv validate empty.daiv fleet.saiv
kaiv: CardinalityViolationError: /@servers has 0 elements,
↪ schema requires min=1 max=10
```

`host=!` declares a unique key (compound keys separate fields with commas); `server=/@servers/*::host` declares that every disk's server must match some server's host. This is Level 2 of kaiv's conformance ladder — the relational essentials, still one forward scan plus a bounded pass.

18 Your Own Type Library

Named types live in `.taiv` libraries — the same definition shape as `std/core`, under your namespace:

```
!.taiv 1 acme/types

// Percentage
/^[0-9]+$ / ..num [0,100]
&percent=
```

```
$ printf '!.registry acme=.\n!.types
↪ acme/types\n\n&percent\nload=87\n' | kaiv build
!.daiv
!.registry acme=.
!acme/types/percent'::load=87
```

The reference `&percent` is authoring shorthand; canonical form carries the full name `!acme/types/percent`, so the artifact is unambiguous forever. During alpha, publishing a library means depositing it at the `kaiv.io` registries (your namespace, one claim); the name-is-the-version rule means evolution is a new name — `acme/types2` — never an edit.

19 The Standard Libraries

Six libraries ship embedded in every implementation — no lookup, ever: `std/core` (the scalar types), `std/enc` (typed embed channels for JSON, XML, and binary payloads), `std/time` (the four datetime shapes with temporal ordering), `std/num` (the non-finite markers), `std/net` (uri, url, email, hostname, port), and `std/math`:

```
$ printf '!.types std/math\n&complex\nz=3+2i\n' | kaiv
↪ build
!.daiv
!std/math/complex'::z=3+2i
```

Part I already put `std/net` to work; the guides show `std/enc` and `std/time` arriving automatically when foreign formats need them.

20 Units

Units are part of a numeric type — compared byte-for-byte, never converted. The toolchain canonicalizes compound spellings so equal dimensions compare equal:

```
$ kaiv unit m*N
N*m
$ kaiv unit s^-2*m
m/s^2
```

Factors sort, negative exponents move below the bar — one spelling again. Currencies are tilde-prefixed and deliberately carry no conversion factors (rates are external and time-varying), so a mismatch is a type error, loudly:

```
$ printf '!.float:~EUR\ntotal=99.50\n' | kaiv build >
↪ eur.daiv
$ kaiv validate eur.daiv usd.saiv
kaiv: TypeMismatchError: ::total=99.50 (type !float:~EUR)
↪ does not satisfy !float:~USD
↪ /^-?[0-9]*\.[0-9]+([eE][+-]?[0-9]+)?$/ ..num (line
↪ 2)
```

Custom units and currencies live in `.faiv` libraries (`!.faiv` header, imported with `!.units`) — the same registry model as types.

21 Provenance in Full

The full provenance triple is `?source@timestamp#dpid` — who says so, when, and which upstream data point — with sources declared once as `.?id uri` and multiple sources comma-separated. A schema can make lineage mandatory:

```
!.saiv 1 g/audit
!.provenance:required

!float
reading=
```

```
$ printf '!float\nreading=2.4\n' | kaiv build > r.daiv
$ kaiv validate r.daiv audit.saiv
kaiv: ProvenanceSchemaError: ::reading: schema requires
  ↳ source and timestamp on every line (line 2)
$ printf '!.?lab https://lab.example.com\n\n!float?lab@202」
  ↳ 60717T090000Z\nreading=2.4\n' | kaiv build >
  ↳ r2.daiv
$ kaiv validate r2.daiv audit.saiv
pass
```

Three levels exist: required (source and timestamp), source (source at least), none (provenance prohibited — for schemas where it would be noise). Because provenance rides the line, `grep '?lab'` is an audit query.

Part IV

The Toolchain

22 The Command Surface

Everything in this manual used six commands; the full surface is small enough to hold in your head. `compile`, `denorm`, and `build` are the

pipeline; schema compiles contracts; `validate` enforces them; `unit` canonicalizes; `infer` derives a schema from data; `import/export` cross formats; `import-schema` converts foreign schemas. All read `stdin` when no file is given. Registry resolution is configured by the nearest `kaiv.kaiv` up from the working directory, overridden by `KAIV_REGISTRY_*` variables; immutable artifacts cache forever, and `--offline` (or `KAIV_OFFLINE=1`) resolves from cache alone — a warm cache is a complete resolution surface.

23 Interop

Eight data formats (JSON, YAML, TOML, XML, CBOR, Avro, Protocol Buffers, ASN.1 DER) and five schema languages (JSON Schema, XSD, `.proto`, Avro Schema, GraphQL SDL) convert in and out of the canonical form you now read fluently. That story has its own shelf: each [guide](#) is a working session for one format, recipe by verified recipe, with the honest fidelity notes where a format cannot carry something back.

24 Errors and the Conformance Ladder

Every `kaiv` error is `Name: context (line)` — the pinned catalog name first (stable across implementations, what conformance tests compare), then the site: field, value, and the violated constraint exactly as compiled. You have seen a dozen; the shape is always the same, and exit codes are always 0 for pass, 1 for any refusal. Implementations declare a *conformance level*: L0 scalars and L1 trees validate in constant memory (the certifiable core), L2 adds the table pass, L3 adds locale-aware collation. Everything in this manual through Part II is L0–L1; the tables chapter is L2 — know which level your target runtime claims.

25 A Worked Finish

The whole ladder, end to end, in four moves. A fleet starts as `.env-`grade lines and ends schema-validated with a unique key — then leaves for a legacy consumer as JSON:

```

$ printf '/@fleet+:=host=eu.example.com|port=443\n/@fleet_
↳ +:=host=us.example.com|port=443\n' | kaiv build >
↳ fleet.daiv
$ printf '!.saiv 1 g/f\n.!.types std/net\n\n\n/@fleet host=!
↳ min=1]\n&hostname\nhost=\n&port\nport=\n[]\n' >
↳ f.saiv
$ kaiv validate fleet.daiv f.saiv
pass
$ kaiv export --json fleet.daiv
{"fleet":[{"host":"eu.example.com","port":"443"}, {"host":
↳ "us.example.com","port":"443"}]}

```

One parting detail, if you caught it: the ports exported as *strings* — the compact `+=` lines carried no annotation, so `443` is a `str` that happens to satisfy `&port`'s integer refinement. The constraint is the type; the annotation is the record of intent. Annotate when downstream consumers care — and you are ready for the [specification](#) whenever a rule needs its exact wording; until then, the toolchain and this manual are the working surface.