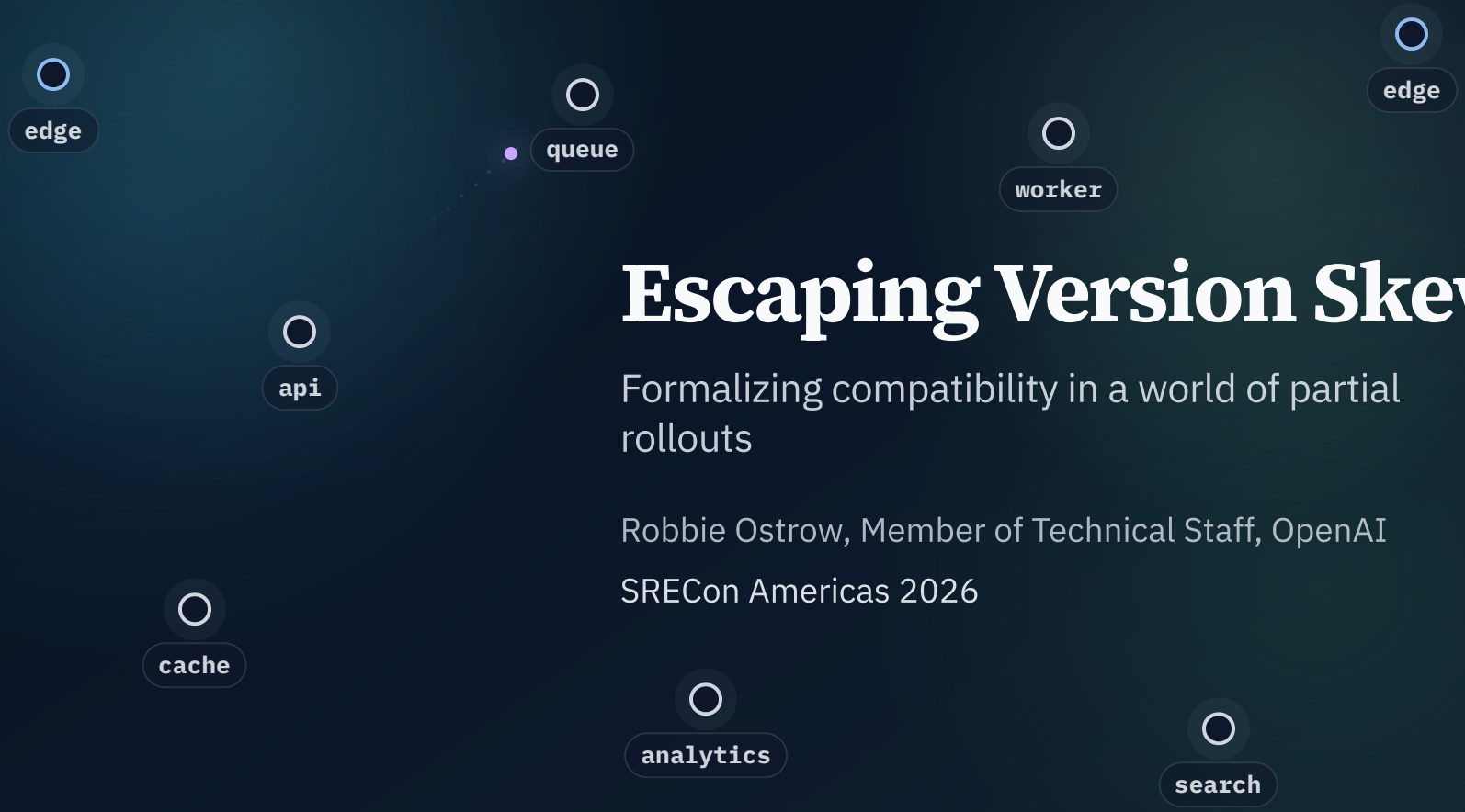


Escaping Version Skew

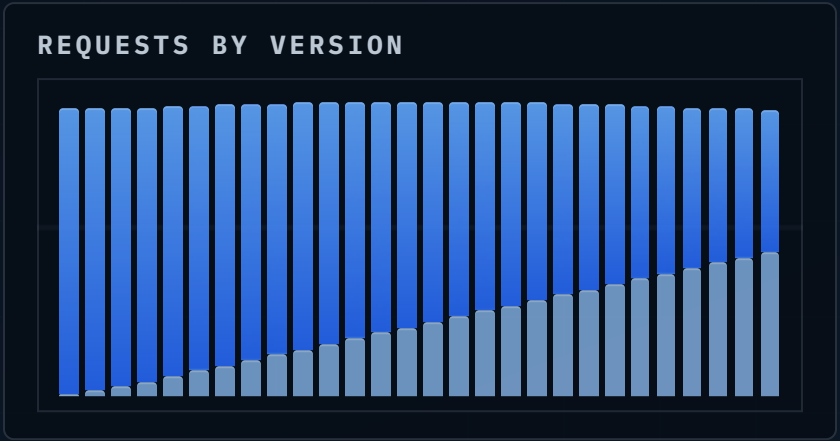
Formalizing compatibility in a world of partial rollouts

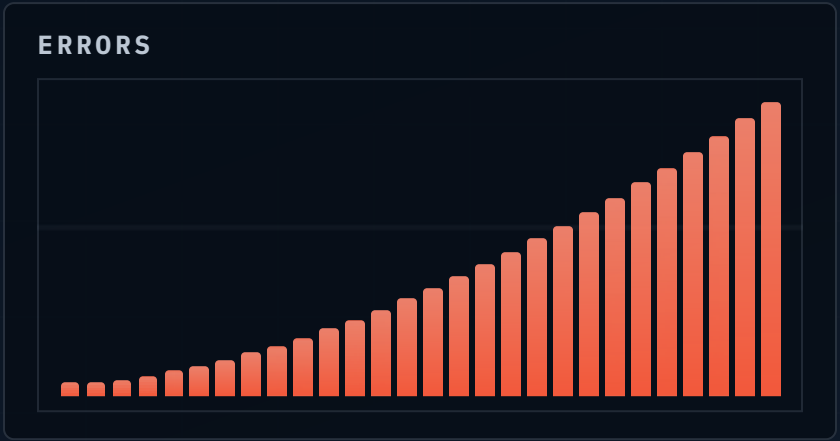
Robbie Ostrow, Member of Technical Staff, OpenAI

SRECon Americas 2026

Errors are climbing during a deploy.

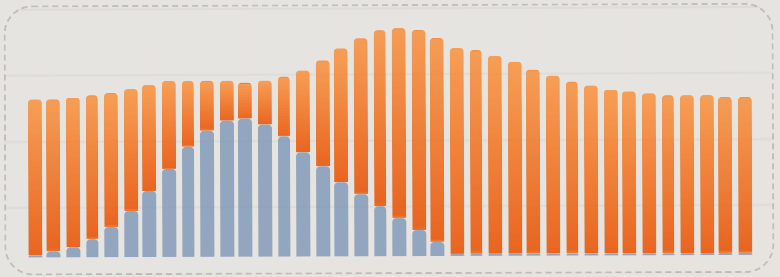
What do you do?





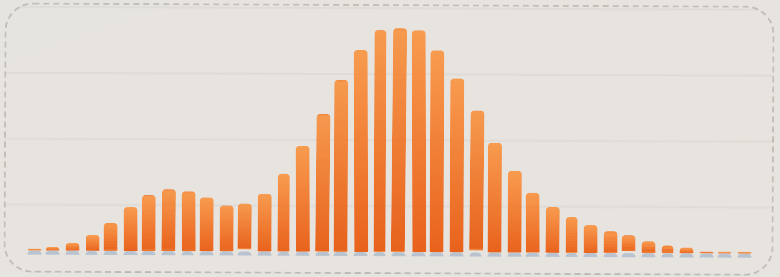
A mixed fleet shared one cache

REQUESTS BY VERSION



old version new version

ERRORS BY VERSION



old version new version (0)

WHY THIS SPREAD

not all pods switched at once

new pod writes new version -> shared cache persists across readers -> old pod reads parse failure

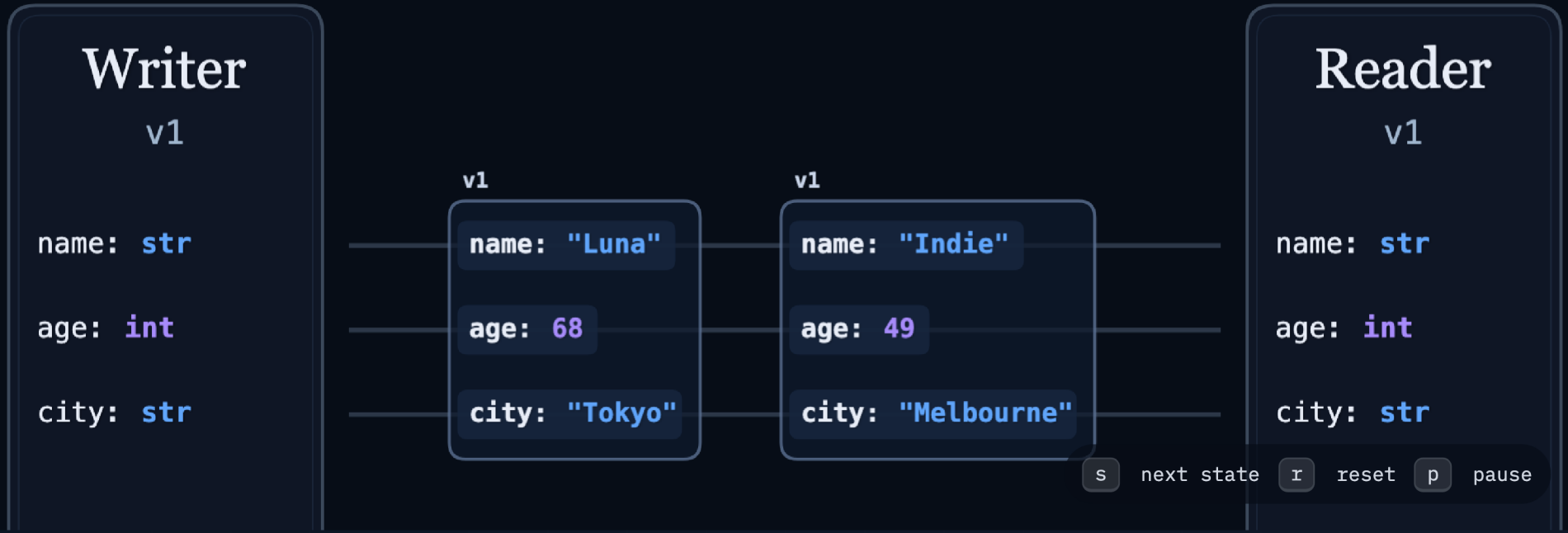
Rollback increased errors

Old readers came back while bad cached data was still alive.

the secret to coordinating ordered rollouts at scale

give up

don't rely on humans


Writer
v1
name: str
age: int
city: str
v1
name: "Luna"
age: 68
city: "Tokyo"
v1
name: "Indie"
age: 49
city: "Melbourne"
Reader
v1
name: str
age: int
city: str
s next state
r reset
p pause

Parseable is not enough

Transport compatibility can still admit states your logic cannot handle.

Grammar

What can be decoded.

Validation

What your system is willing to accept.

> If the logic depends on the rule, the rule belongs at the boundary.

Avoid optionalslop

LEGACY

```
message UserProfile {
  optional string display_name = 1;
  optional string first_name = 2;
  optional string last_name = 3;
  optional string legacy_full_name = 4;
  optional string avatar_url = 5;
  optional string avatar_id = 6;
  optional string locale = 7;
  optional string timezone = 8;
  optional bool email_verified = 9;
  optional bool phone_verified = 10;
  optional string phone_number = 11;
  optional string backup_phone_number = 12;
  optional string city = 13;
  optional string region = 14;
  optional string country = 15;
  optional string legacy_metadata_json = 16;
}
```

COMPATIBILITY RESIDUE

One type gets weaker over time

As old fields accumulate for compatibility, the shared proto stops expressing the real domain model and turns into "maybe this, maybe that".

Impossible states become routine

Now business logic has to remember which subsets belong together, which are stale, and which combinations should never exist.

Strict contracts are better for ~~humans~~ agents

Smaller legal state space

Fewer ambiguous shapes for an agent depend on.

Hidden assumptions become explicit

Put the rule at the boundary so the agent does not have to recover it.

Crisper test oracle

A strict contract allows an agent loop to quickly iterate upon correctness.

Agentic workflows get safer when the boundary is narrow enough to make bad states impossible, not just unlikely.

Stop sharing types.

Writers should be as strict as possible

Emit today's contract, not a mushy superset shaped by every historical rollout.

Readers should accept the union of the last few writers

Carry compatibility in the reader, where skew actually lands.

> Stop sharing types between client and server.

A strict writer, a union reader

```python
class UserProfileWriter(BaseModel):
    name: str = Field(min_length=1)
    age: int = Field(ge=0)
```

```python
type UserProfileReader =
    | UserProfileV1Reader
    | UserProfileV2Reader
    | UserProfileV3Reader

match payload:
    case UserProfileV3Reader(name=name,
        ...
    case UserProfileV2Reader(full_name=
        ...
```

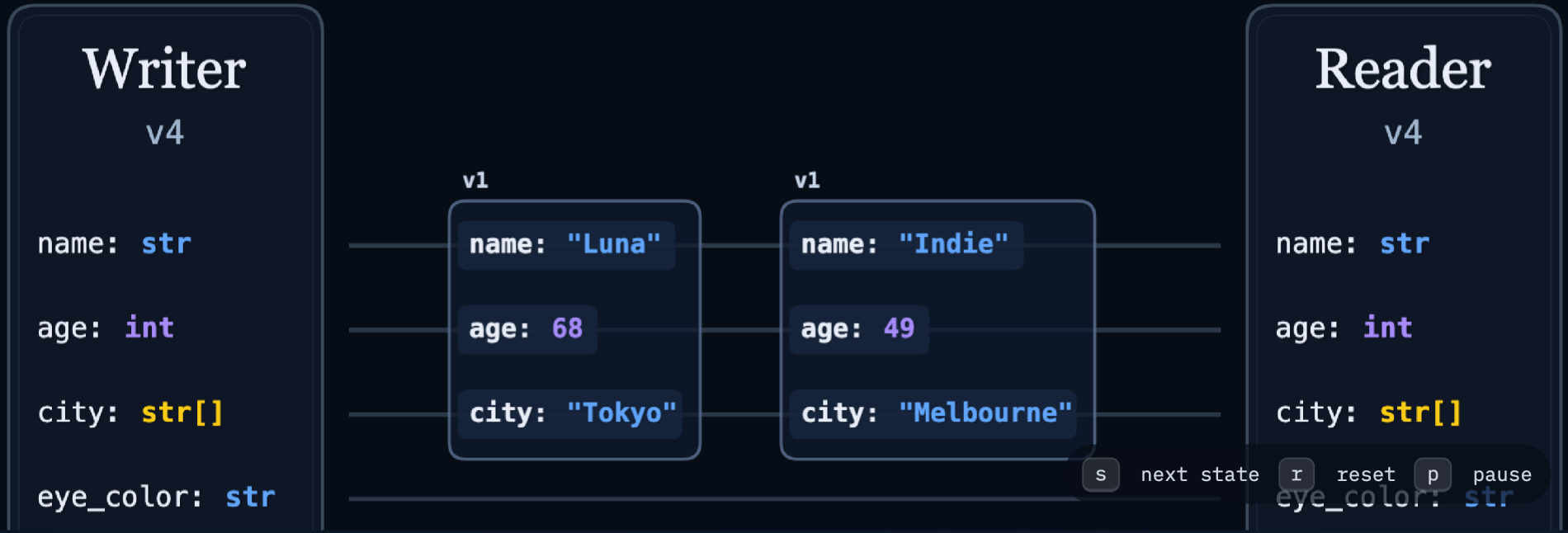
New writes stay clean. Compatibility is quarantined to explicit old-version branches.

Stamp every payload with a writer version.

Writers stamp the shape they emitted

Readers branch on the stamp, not on custom logic

1. Update the schema.
2. Detect breaking changes.
3. Keep the writer as strict as possible.
4. Make readers a tagged union of the last few writers.
5. Measure how often old writer branches still deserialize.
6. Delete old branches once those metrics hit zero.


Writer
v4
name: str
age: int
city: str[]
eye_color: str
v1
name: "Luna"
age: 68
city: "Tokyo"
v1
name: "Indie"
age: 49
city: "Melbourne"
Reader
v4
name: str
age: int
city: str[]
eye_color: str
s next state
r reset
p pause

Tooling!

Prove when possible. Fuzz when not.

https://jsoncompat.com/checker

jsoncompat Checker Fuzzer Install Usage

Schema compatibility checker

Old schema

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" }
  }
}
```

New schema

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string", "minLength": 5
},
    "age": { "type": "integer", "minimum": 18 }
  }
}
```

A subsumption checker asks set containment

New writer safe for old reader

$L(new) \subseteq L(old)$

Old writer safe for new reader

$L(old) \subseteq L(new)$

A schema change is compatible in a direction exactly when every value accepted before is still accepted after, or vice versa.

Serializer assumption: no extra emitted fields beyond the declared schema.

Two passes: prove, then search

Static checker

Fast, deterministic proofs for the common cases.

Fuzzer

Concrete counterexamples when the schema is too expressive for a complete proof.

1. Try to prove set containment from the schemas alone.
2. If the proof is incomplete, search for a witness value.
3. Use the witness to make the breakage obvious to humans and agents.

A concrete witness makes breakage obvious

OLD SCHEMA

```
"if": { "properties": { "mode": { "c
"then": {
  "properties": {
    "value": { "maximum": 100 }
  }
}
```

one keyword tightens

NEW SCHEMA

```
"if": { "properties": { "mode": { "c
"then": {
  "properties": {
    "value": { "exclusiveMaximum": 1
  }
}
```

WITNESS `{"mode":"percent","value":100}` Valid before. Rejected after.

https://jsoncompat.com/checker

jsoncompat Checker Fuzzer Install Usage

Schema compatibility checker

Old schema

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" }
  }
}
```

New schema

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string", "minLength": 5
},
    "age": { "type": "integer", "minimum": 18 }
  }
}
```

Make compatibility checks live next to the type

```python
from pydantic import BaseModel, Field

@jsoncompat_check(direction="both", stable_id="user-profile")
class UserProfile(BaseModel):
    name: str = Field(min_length=1)
    age: int = Field(ge=0)
```

> The stable ID ties this model to its historical schema snapshots, and CI checks both rollout directions on every change.

Adopt it in phases

1. Start by annotating storage-boundary types and checking both rollout directions in CI.
2. Add writer-version stamps and measure which old branches are still being read.
3. Split strict writer types from union reader types on the boundaries that matter most.

> You do not need the whole end-state on day one to start catching real breakages.

When not to do this

Probably not worth it

Ephemeral internal RPCs with no durable state, no queues, and no meaningful rollback tail.

Absolutely worth it

Caches, queues, databases, durable workflows, mobile or external clients, and any boundary where state outlives binary.

> Use the heavy machinery where old code and new state can meet. That is where version skew turns into incidents.

Constrain. Split. Gate. Observe.

Constrain

Make strict schemas a cultural default: hidden assumptions should become contract rules, not tribal knowledge.

Split

Generate reader and writer types in your language of choice from the schema, and make historical unions cheap to maintain.

Gate

Run CI against the schema itself and against previous versions, detect breakages mechanically, and fail unsafe changes before merge.

Observe

Measure deserializations by payload version so you can see old tails, rollback risk, and when a branch is really gone.

Questions?

slides and tooling at jsoncompat.com