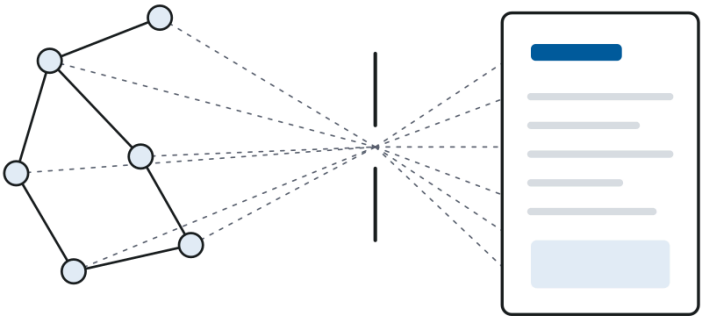
DRAFT 0.1

First Principles of the Web

Graphs are not the thing, they are the thing that gets us to the thing.



Martynas Jusevičius

firstprinciplesoftheweb.org

ATOMGRAPH

First Principles of the Web

Graphs are not the thing, they are the thing that gets us to the thing

Martynas Jusevičius

Contents

1. Preface

DRAFT 0.1 · *atomgraph.com* · *GitHub* · *X* · *LinkedIn*

There is exactly one way to build applications that are *of* the web rather than merely *on* it, and it is data-centric, declarative, and graph-based. "Exactly one" is relative to rules imposed by the web itself. This book derives those rules and examines the consequences of rejecting them. It scores today's JSON APIs, JavaScript frameworks, and compile-to-browser toolchains against the same rules; each turns out to be a partial rediscovery of this way or a detour from it.

The book is structured as a derivation: every statement in it is a definition quoted from the web's own specifications, a proposition that follows from previous statements, or an observation you can verify against the deployed web. If you find a statement that is none of the three, the book has a bug; please report it. There is one deliberate exception: Chapter 5 makes a bridge from the web to the formalism that is argued but not proved. If you want to reject the book's conclusion, that is the step to reject. The book is written for people who build for the web; it assumes no mathematics beyond sets and functions. Chapter 2 explains the method; Appendix A covers notation and reading tracks.

Underneath the method is a choice of genre. The web is mostly treated as software engineering, a craft of frameworks and taste; this book treats it as a science, an object whose structure can be derived, proved, and tested by prediction rather than surveyed and preferred. Chapter 21 returns to it once the scores are in.

One more thing. This book is built to practice what it derives. Its canonical edition is designed as an application of that very kind, in which every proposition is a resource with its own address. That makes the edition an instance of the book's own thesis. As of this writing, that edition is still under construction.

1.1 The Argument in One Page

A web application is two functions. `read` turns a request and the state of the world into a document; `write` turns a request and a state into a new state. That is HTTP restated, and every framework ever shipped is an implementation detail of it.

Strip any page (a newspaper, a dashboard) and the same skeleton emerges: first style, then arrangement, then selection, leaving state. Every `read` can therefore be factored as `present ∘ arrange ∘ select` (one factor per stripped layer). The factorization matters only if the factors are separate, declarative, substitutable, and addressable.

Ask what `State` must be, and the requirements come from the web itself. `State` must host any domain. It must compose across parties who have never met — which forces merging by

union over facts that carry their own meaning. Its names must work globally. The smallest fact meeting all three requirements is a triple: two global names and a value that may itself be such a name. Any minimal model meeting the requirements is isomorphic to this one: sets of triples, merged by union. That is a uniqueness theorem; to reject its conclusion you must fault a step of the proof or reject a requirement.

The resulting structure maps to RDF, SPARQL, XSLT, and CSS, standardized between 1996 and 2014 and later abandoned — abandoned, the book will argue, not refuted. The book then audits current technologies against the derived requirements, examines the compensating industry that grows where a requirement is not met, and ends with one table scoring every stack, the derived one included. The components can then be combined into a complete architecture without introducing a new standard: a generic engine whose behavior is specialized by data rather than application-specific code. This matters again now: software agents need machine-consumable state, and the industry is building new integration layers to provide it.

If the derivation holds, the next web needs no inventing; it needs only to be put to use. The rest of this book is the proof, the scores, and the evidence.

I

Part I — The Object

Before anything can be derived, the object of study must be fixed. This part quotes the web's own definitions (a pair of functions with one deliberate omission) and states the question the rest of the book answers: what must `State` be?

Chapter 1 What the Web Is Chapter 2 Analysis and Synthesis

2. Chapter 1. What the Web Is

Vague but exciting...

— Mike Sendall, on Tim Berners-Lee's 1989 proposal

Nothing in this chapter is mine. That is the point of it.

The web ships with its own definitions, and they are shorter than you probably expect. There are identifiers:

```
I     the set of URIs                                   (RFC 3986)
```

There are requests, which HTTP (RFC 9110) builds from identifiers:

```
Req = I × Method × Headers × Body                       (RFC 9110)
```

The body may be empty; the empty body is a body, the way an empty set is a set. Requests with a *safe* method (RFC 9110's word for the methods that only ask, never change) almost always leave it empty. Definition 1.1 shows what the unsafe methods use the body for. (RFC 9110's own name for it is *content*, a word this book will need for something else, so the older wire name stays.)

Responses come back the same shape, because RFC 9110 defines one message form for both directions. Where the request had a method and an identifier, the response has a status code:

```
Resp = Status × Headers × Body                          (RFC 9110)
```

For the model below, only the response body matters. The body itself is octets. A header names their format (`Content-Type`). How the octets parse is defined by the format's own specification, so the book does not have to define it. On the parsed side of that line lives the document, the thing a user agent displays. Call that domain `Doc`, and leave its internals alone for now. The envelope around it (the status code, the response headers) is how a document travels, ages, and caches. That is transfer machinery, and Definition 1.1 will not mention it.

Definition 1.1. A *web application* is a pair of functions, `read` and `write`:

```
read  : Req × State → Doc
write : Req × State → State
```

In this model, `read` corresponds to HTTP's safe methods: `GET` takes a request and the current state of the world and produces a document. `write` is what the unsafe methods do: `POST`, `PUT`, `PATCH`, `DELETE` take a request and a state and produce a new state. The request's body carries what the change should be. The body belongs to the write side: on a safe request it has no

defined meaning (RFC 9110 §9.3.1), and `read` ignores it. `read`'s output travels in the *response's* body instead. Like `Doc`, `Body` stays opaque for now; Chapter 7 defines its contents.

Every web application you have ever used implements these two functions — from a static homepage to the heaviest single-page application — because HTTP gives it no other way to be an application on the web. The framework it was built in is an implementation detail of Definition 1.1.

Definition 1.1 deliberately leaves `State` unspecified. The central question of the book is therefore: *what properties must State have?* Part II derives those properties from constraints of the web, and Part III shows how they correspond to existing standards.

Prop. 1.2. Every deployed web application implements Definition 1.1. *(Verification: RFC 9110 §9; there is no third kind of method.)*

Prop. 1.3. Definition 1.1 places no constraint on architecture. Both a 1993 CGI script and a 2026 React application satisfy it. *(This is why the definition is safe as an axiom; no one on any side of any framework war can reject it.)*

Persistent connections — why WebSockets and server push are not a counterexample. WebSockets (an open two-way message channel between browser and server) and server push may look like a counterexample because messages on an established connection are not themselves HTTP requests with safe or unsafe methods. At the application level, however, they still carry either data from the server to the client or changes from the client to the server. The first corresponds to `read` output delivered when state changes; the second corresponds to input to `write` delivered over an existing channel. Definition 1.1 therefore still describes the application behavior. What changes is the surrounding HTTP machinery: individual messages no longer necessarily have their own method, cache semantics, or URI. Each dropped piece has a cost; Part IV computes those costs one by one.

The web succeeded against contemporaries such as Gopher, BBSs, desktop applications, and Java applets; Chapter 12 revisits that comparison. It succeeded because its `read` was *transparent*: documents were declarative, addressable, linkable, indexable, and legible to machines that did not produce them. Part IV evaluates later technologies on one variable: how much of that transparency they preserve. Chapter 21 makes the term exact. Before doing that, the undefined `State` in Definition 1.1 needs a model. Chapter 2 explains the method used to derive one rather than selecting it from current practice or personal preference.

3. Chapter 2. Analysis and Synthesis

Chapter 1 defined a web application but left `State` untyped. Looking at existing applications does not determine a model: one uses rows, another trees, another object graphs. That variety is what Proposition 1.3 predicts: any architecture fits Definition 1.1, so every kind got built. Choosing a preferred model would be equally arbitrary. Instead, this book derives requirements for `State` from constraints imposed by the web itself. If the derivation is valid, rejecting the resulting model requires rejecting at least one of those constraints.

`State` itself is not directly observable. Requests and responses are visible on the wire, and the document returned by `read` is public, but the server's internal state is not. The analysis therefore starts from the document and works inward. It removes properties that can vary without changing the underlying information, checking each removal against ordinary web behavior. The layers that can be removed become the factors of `read`; what survives every removal is the first direct evidence of what `State` must contain.

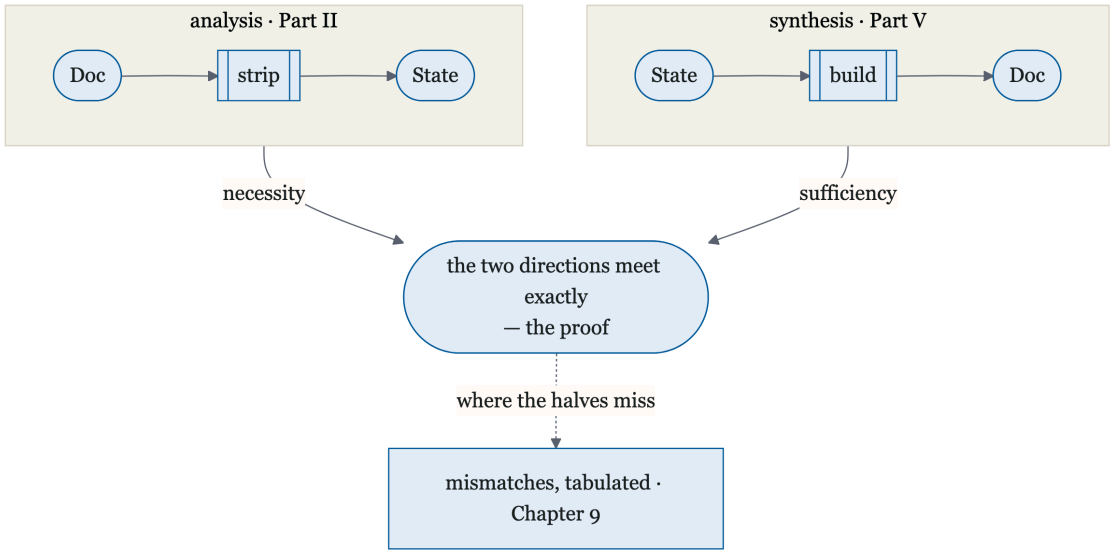
Stripping a document can suggest such a factorization, but it does not by itself prove that the factorization is necessary. The result could still depend on the particular pages chosen or on the order of the removals, rather than apply to every page. The check is independence: set the pages aside, keep only the derived parts, and build with them. A working application space means the full range of applications Definition 1.1 admits, not one rebuilt demo. If that space can be constructed from the derived parts alone, using nothing from the original pages, then the skeleton was in the pages, not in the procedure. And if the construction fails, or quietly needs parts the derivation never produced, the failure is public: either the analysis kept the wrong things or the parts list is incomplete, and each defect is visible on its own.

The Greek geometers called the two directions *analysis* and *synthesis*. Pappus's *Collection* describes the pair: assume the thing sought and work backwards to what is established; then reverse the path, and the reversal is the proof. Newton restates it as a rule of natural philosophy in the *Opticks*: the investigation of difficult things by analysis "ought ever to precede the method of composition." The practice has modern instances wherever a structure must be known rather than guessed. Organic chemists worked out a molecule's structure by breaking it into identifiable fragments, and accepted that structure as proven only once they had built the same compound from known ingredients and it matched. Software has the clean-room: a rebuild is independent exactly when the rebuilding team touched only the derived specification, never the original. Both keep the geometers' point: taking apart suggests a structure; only building back, independently, proves it.

The claim can now be stated precisely. The analysis, if it succeeds, will show necessity: every web application has the derived form, because under Definition 1.1 there is nothing else `read`

and `State` could be. The synthesis, if it succeeds, will show sufficiency: the derived parts are enough to build the full application space, with nothing missing. Neither half proves the claim alone; the proof is that the two match, and the book checks the match twice, in theorems and in running code. Where analysis and synthesis do not coincide, Chapter 9 lists the mismatches explicitly. The opening pages already name the result, but naming it proves nothing. The proof is the derivation, and that is the part a reader can check.

The book follows the same method. Part II performs the analysis first on real pages and then as theorems that quantify over arbitrary pages. Once the properties of `State` are derived, the write side follows. Part V reconstructs an application space from the derived components. Between analysis and synthesis, Parts III and IV compare the result with existing standards and current technologies and record the mismatches. Part VI considers what follows from the resulting architecture.



The method used in the book. Analysis (Part II) works from the observable `Doc` toward `State` and establishes necessity. Synthesis (Part V) builds from `State` back to documents and establishes sufficiency. Chapter 8 compares the two formally, Chapter 19 does so in running code, and Chapter 9 records the mismatches.

One decision remains before the stripping starts, and it is a control on the experiment: which pages to strip. At least two are needed, because one page says nothing about pages in general. The most useful pair is two very different pages: anything that survives the same removals in *both* is less likely to be specific to either one. So the pair is a newspaper front page and a wind-farm dashboard. The front page is written by a handful of editors on an editorial rhythm and read by millions. The dashboard is written continuously by machines and read by one operator on shift. Different domain, different audience, opposite rhythms of `read` and `write`; under Definition 1.1 they have one signature. If these two reduce to the same skeleton, everything between them likely does too. Print both. Now take them apart.

II Part II — The Analysis

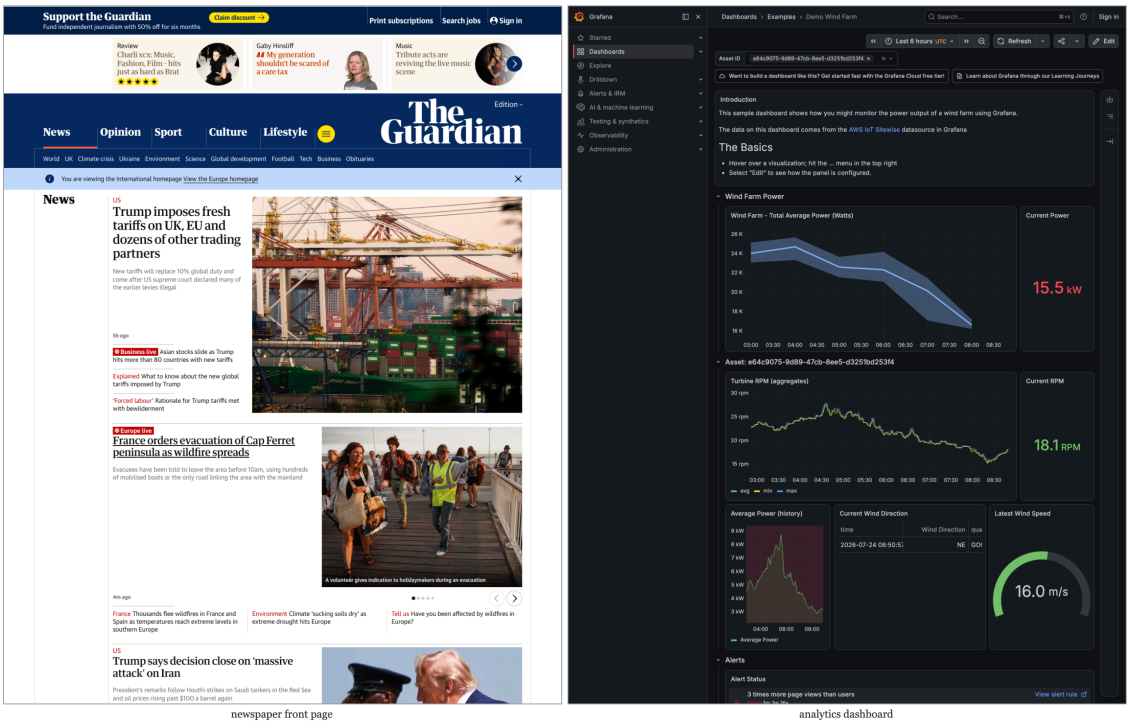
Part I defined a web application as two functions and left `State` unspecified. This part derives a factorization of `read`, the properties required of its factors, the resulting model of `State`, the conversion from graph to tree, and the corresponding write operation.

Chapter 3 Stripping the Page Chapter 4 The Factorization Chapter 5 What State Must Be Chapter 6 From Graph to Tree Chapter 7 The Write Side

4. Chapter 3. Stripping the Page

Chapter 2 chose two pages that are as different as any pair it could find: the newspaper front page and the wind-farm dashboard. Strip them, one layer at a time, and watch the same skeleton emerge from both. The exhibits below do exactly that, to real pages: the Guardian's international front page and a Grafana wind-farm monitoring dashboard, captured on the same morning.

Strip 0: full
the page as shipped



newspaper front page — analytics dashboard

Figure 4.1: Strip 0 — the pages as shipped

The starting material. One page is paper-white and editorial, the other black and numerical; they share, apparently, nothing.

4.1 Strip the style

Turn off CSS, switch to reader view, print in monochrome. The page looks different; nothing it *says* changes. So a document factors:

```
Doc = Style × Content
```

The justification is already deployed on every device you own: dark mode, print stylesheets, reader view, accessibility themes — style varies while content holds fixed.

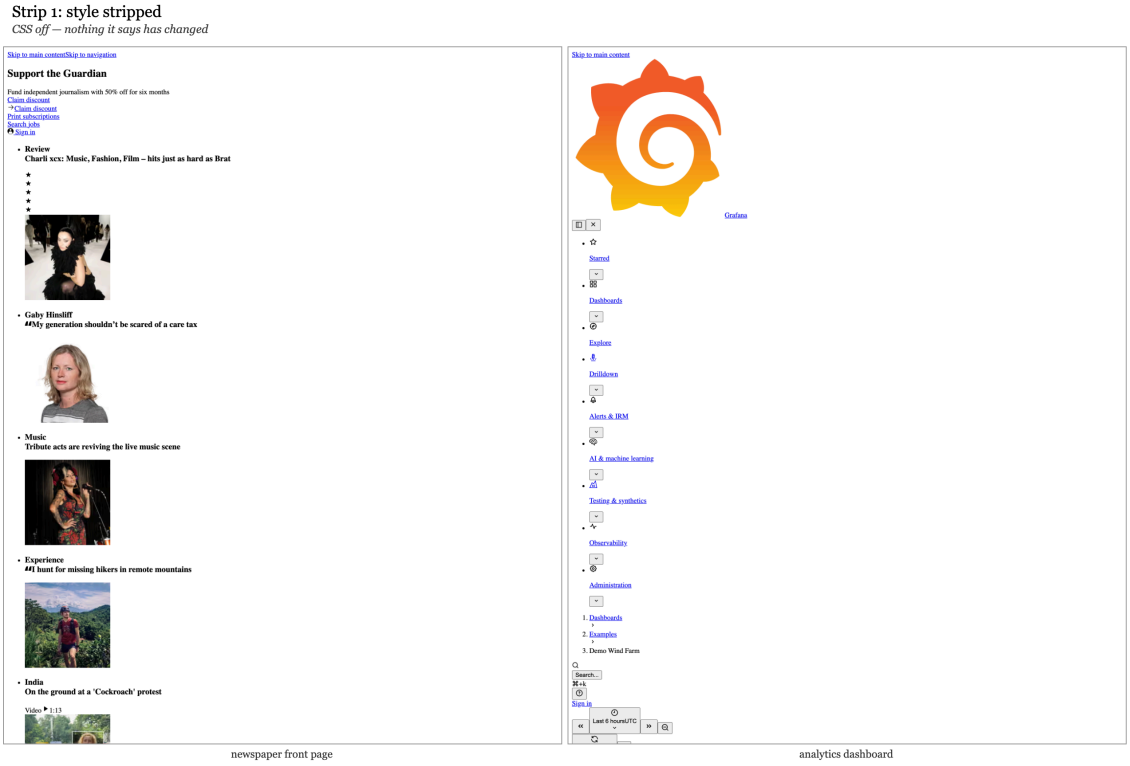


Figure 4.2: Strip 1 — style stripped

CSS off. The newspaper still says everything it said — headlines, bylines, photographs, in browser-default dress. Note the asymmetry on the right, and file it for Part IV: stripping style from the newspaper leaves the news; stripping it from the dashboard leaves mostly chrome. The headline numbers survive as text, but the time-series curves were never in the document at all — they are pixels on a canvas.

4.2 Strip the arrangement

The same content appears as a table on desktop, a card list on mobile, a chart in the summary view. The facts are identical; their shape as a document differs. So content factors:

```
Content = Arrangement × Data
```

Every "view toggle" on the web justifies this factoring, showing the same data as a different tree.

Figure 4.3: Strip 2 — arrangement stripped

Arrangement off. Both pages now use the same format: one block per entity, sorted, with no nesting. A headline is paired with a section, and a panel title with a value. The two sites that shared nothing now differ only in vocabulary.

4.3 Order as data

There is one complication before the next strip. A careful reader has already spotted it: the front page's order *means* something. The lead story leads because an editor judged it should, and a strip that discarded the judgment would be destroying content while claiming to remove arrangement. That judgment can itself be represented as data: *this article, prominence one*. The strip's real effect is that order moves out of the tree and into the data, where it survives every re-arrangement that follows. The deployed web already shows what happens otherwise: the same articles also go out through feeds, and a feed delivers them without the front page's ordering. When prominence lives only in an arrangement, it is lost on every consumer who receives the content without it. Chapter 6 will harden this from a concession into a law: if the order is a message, the order is data.

4.4 Strip the selection

Every page shows a sliver of something much larger. An article's own page and the front page draw from the same pool; the dashboard shows the last six hours, but last week exists. So data factors:

```
Data = Selection × State
```

Pagination, filters, and search are deployed proof that the page is a window, not the world.

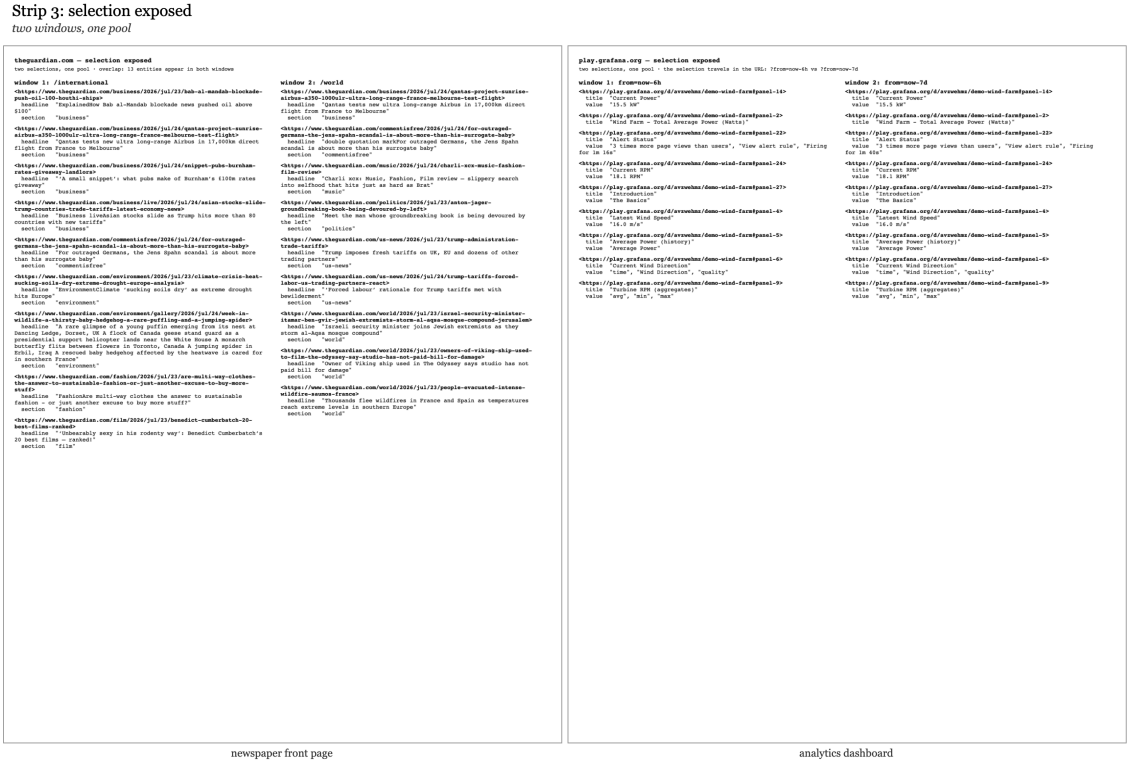
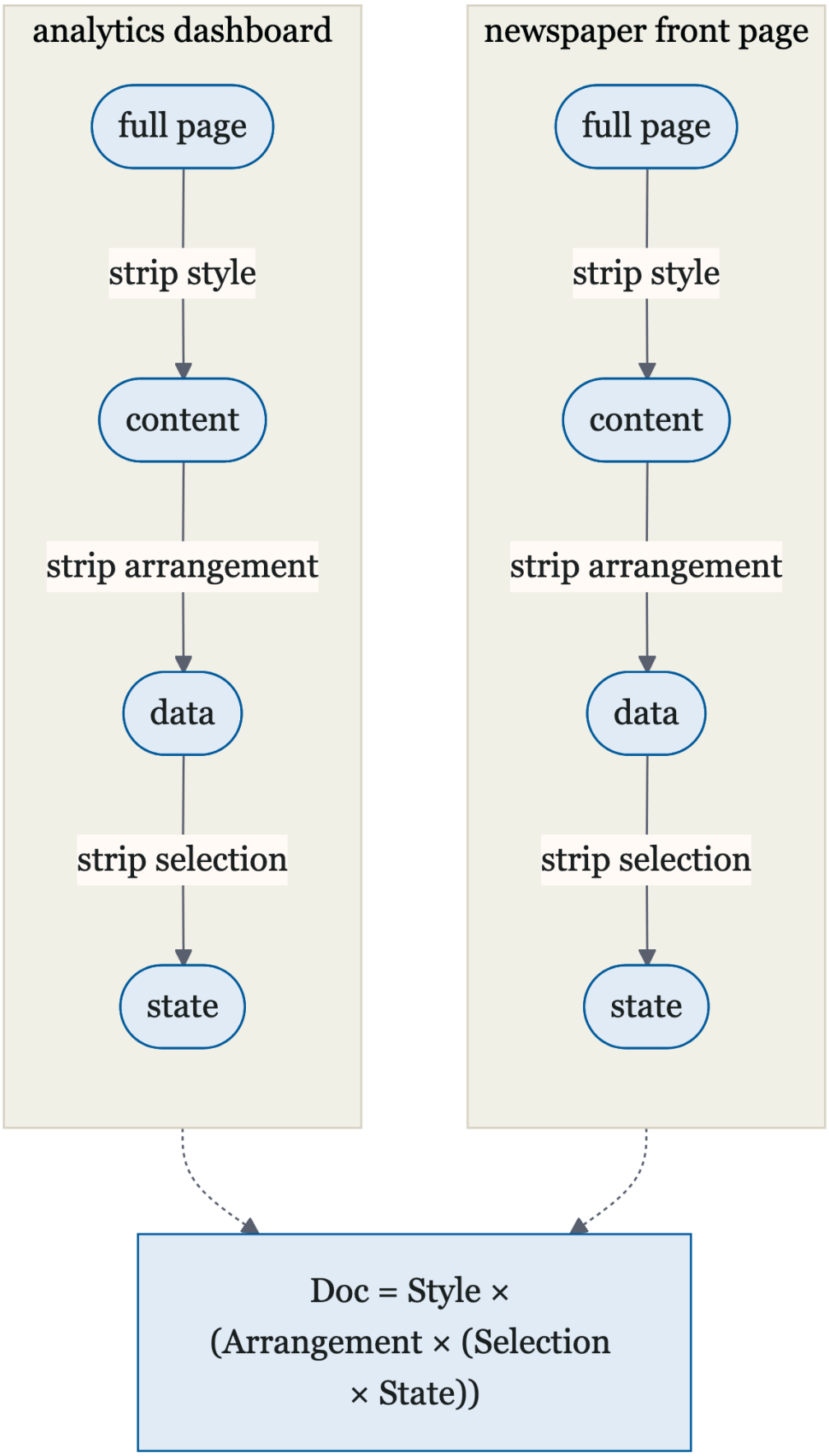


Figure 4.4: Strip 3 — selection exposed

Selection is made visible here as two windows over one pool. On the left, `/international` and `/world` share 13 entities, because the same articles are drawn twice. On the right, the same dashboard appears at `?from=now-6h` and `?from=now-7d`, so the selection travels in the URL, where anyone can change it.

Three strips, and both of our maximally different sites have reduced to the same expression:

```
Doc = Style × (Arrangement × (Selection × State))
```


analytics dashboard
full page
strip style
content
strip arrangement
data
strip selection
state
newspaper front page
full page
strip style
content
strip arrangement
data
strip selection
state
Doc = Style ×
(Arrangement × (Selection
× State))

The three stripping steps applied to both sites: two different pages reduce to the same skeleton.

the page, as shipped

4.5 Current Power

partOf
overview

value
15.5 kW

4.6 Wind Speed

partOf
overview

value
8.2 m/s

– style off

4.7 Current Power

partOf
overview

value
15.5 kW

4.8 Wind Speed

partOf
overview

value
8.2 m/s

– arrangement off

```
⟨…#panel-14⟩
    partOf · ⟨…#overview⟩
    title · "Current Power"
    value · "15.5 kW"

⟨…#panel-7⟩
    partOf · ⟨…#overview⟩
    title · "Wind Speed"
    value · "8.2 m/s"
```

– selection off

```
selection: ?p partOf overview · ?p title ?t · ?p value ?v

(⟨…#panel-14⟩, type, ⟨…#Panel⟩)
(⟨…#panel-14⟩, title, "Current Power")
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-14⟩, partOf, ⟨…#overview⟩)
(⟨…#panel-7⟩, type, ⟨…#Panel⟩)
(⟨…#panel-7⟩, title, "Wind Speed")
(⟨…#panel-7⟩, value, "8.2 m/s")
(⟨…#panel-7⟩, partOf, ⟨…#overview⟩)
(⟨…#panel-3⟩, type, ⟨…#Panel⟩)
(⟨…#panel-3⟩, title, "Grid Frequency")
(⟨…#panel-3⟩, value, "49.98 Hz")
(⟨…#panel-3⟩, partOf, ⟨…#archive⟩)
(⟨…#farm⟩, name, "Anholt Offshore")
```

Interactive exhibit (online edition): here the strip runs live rather than as before-and-after images. The dashboard is rebuilt in place, and each layer can be removed and restored.

Two examples prove nothing about all websites; the strips are illustration. The universal claim is Chapter 4's theorem, quantifying over every `read` at once.

5. Chapter 4. The Factorization

REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

— Roy T. Fielding, dissertation §5.1.5, 2000

Chapter 3 stripped two pages by hand and called the result illustration; this chapter proves the universal claim. The strips become the factors of a typed pipeline. A definition (properness) separates real factorizations from trivial ones. And the analysis theorem guarantees that every `read` factors, and that the factorization can be made proper. The final sections relate the method to Roy Fielding's REST dissertation and derive the four independent timelines produced by a proper factorization.

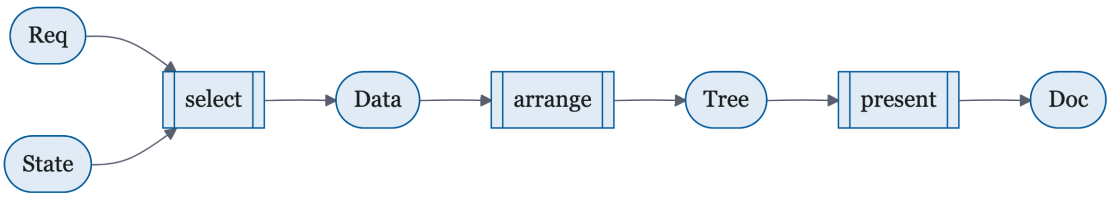
5.1 The pipeline

Chapter 3's strips, read as function types:

```
select  : Req × State → Data          which facts
arrange : Data → Tree                  what structure
present : Tree → Doc                   what appearance

read = present ∘ arrange ∘ select                          (4.1)
```

`select`, `arrange`, and `present` are Chapter 3's Selection, Arrangement, and Style; `Tree` is the arranged data, what Chapter 3 called Content, before style touches it.



The pipeline of (4.1). Rectangles are the factors; rounded nodes are values. Under S4 (the fourth properness condition defined below), each factor's output is a web resource: it has a URI, and a GET on that URI returns it.

Prop. 4.2 (Existence, trivial). Every `read` factors as (4.1). *Proof:* make `select` and `present` do nothing except repackage their input at the required type, and put the whole of `read` inside `arrange`. Call this the *fused* factorization. ■

Proposition 4.2 matters because of what its proof shows. A factorization always exists, so the bare existence of one carries no information. The information is in whether the factors are

genuinely separate. Definition 4.3 states that separation as four checkable properties. Those four properties are what "declarative architecture" means, once the phrase is required to mean anything checkable. It is the book's central definition.

5.2 Properness

Definition 4.3. A factorization (select, arrange, present) is **proper** iff:

S1 — Obliviousness. Each factor communicates with the next only through its output. `arrange` sees data, never the request. `present` sees a tree, never the data. No side channels: `arrange` and `present` are constant in `Req` and `State` except through their arguments.

S2 — Declarativity. Each factor is the *meaning of a term in a language* — there exist three languages, one per factor, with independently defined semantics such that `select = ⟦q⟧`, `arrange = ⟦t⟧`, `present = ⟦s⟧` for terms `q`, `t`, `s`. This is what "declarative" means, made precise: the meaning of the query does not depend on the stylesheet, because each language's semantics is closed.

S3 — Substitutability. Replace any factor with another term of its language and you still have a web application; the change is confined to that factor's concern.

S4 — Addressability. Each factor's output is itself a web resource: the data produced by `select` has a URI and is dereferenceable, independently of the document it is destined to become.

S1–S3 could describe any well-factored program. S4 is the web condition: the factorization itself becomes public, *exposed through the web's own reference mechanism*. An application satisfying S1–S4 is part of the web at every layer, not only at its rendered surface.

The dashboard makes it concrete: `select` pulls the panel's `title` and `value`, `arrange` nests them into a card, `present` themes the card. S4 is the property you can check by hand. Each factor's output (the data, the card, the themed page) has its own URL and *dereferences*: a GET on the URL returns it.

Three of these properties were written down by the web's own architects, as advice. *Architecture of the World Wide Web, Volume One* (W3C Recommendation, 2004; hereafter AWWW) names the separation of content, presentation, and interaction a good practice (§4.3). It names orthogonal and composable specifications a principle (§5.1). And it asks URI owners to provide representations of their resources (§3.5) — S4's demand, minus the intermediates. All of it is stated as SHOULD, because a recommendation can do no more than recommend. Hold that until Proposition 4.4: those norms were theorems all along. AWWW is a witness here, never a premise: assuming §4.3 would be assuming this chapter's conclusion, and the method forbids that.

The benefit of S4 is immediate and measurable. It is Fielding's list: HTTP caching per stage rather than per page; crawlability of *data* rather than of renderings; intermediaries; indepen-

dent evolution of the layers. The last of these becomes a proposition before the chapter ends. Every one will reappear in Part IV, scored against every architecture that forfeits it.

5.3 The analysis theorem

Prop. 4.4 (Analysis theorem). If a `read`'s output depends on `State` only through some finite part, it factors into the three stages, satisfying S1. The factorization then lifts to proper: realize each factor as a term of the languages Part III fixes in advance, give each stage's output a URI, and S2–S4 hold. Finite dependence forces the shape; the lift is a construction the web always permits, never a consequence of finiteness.

The hypothesis is mild: a document renders finitely many facts, so every site ever deployed qualifies. Finiteness is there to guarantee a minimal fragment exists; it excludes nothing real.

Proof sketch — take a minimal fragment the output depends on. Define `select(r, S)` as a minimal fragment of S on which `read(r, ·)` actually depends, guaranteed by finiteness; `arrange` and `present` are the induced quotients. That much is the analysis half: it gives S1 and the three-stage shape. S2–S4 are the lift. S2 is realization in languages whose semantics are fixed in advance and shared across applications, never invented around the `read` (which would make S2 vacuous). S3 is substitution within them, and S4 is publication, an act. The synthesis theorem (8.2) supplies all three. Full proof in Appendix B. ■

Proposition 4.4 is what Chapter 3 was illustrating: stripping a real page is *computing its proper factorization by hand*. The theorem guarantees the exercise terminates for every site — including every site not yet built. And the point held over from AWWW can now be stated: every `read` provably affords the separation that §4.3 could only recommend.

5.4 The debt to Fielding

The debt to Fielding runs deeper than the property list. REST was the last serious attempt to *derive* web architecture rather than fashion it: constraints are applied stepwise, and each constraint induces its own properties. That is the method this book inherits and pushes to theorems. But REST constrains the conversation and leaves the vocabulary open. It says how representations must transfer: statelessly, cacheably, through a uniform interface. It declines, deliberately, to say what a representation or the state behind it must *be*. That open question is this book's subject. Chapter 5 closes it. That closure was unavailable to REST's own method in 2000: the answer had been standardized only the year before. And the pressure that makes it visible (machines reading the web) was two decades out. Read this book as the second half of a derivation whose first half Fielding wrote.

His deepest constraint is also his least defined: the **uniform interface**. In Fielding's own estimation it is the central feature that distinguishes the web from every prior architecture, yet he delivered it as four clauses of prose (§5.1.5) and never formalized it. "Uniform" is the quantifier: one signature for every application. That is why Definition 1.1 could open this book by fitting

every web application ever built, and why Part V can close it with one application for every domain. The interface was always uniform; the state beneath it was not.

The four clauses, typed — one per component. (Borrows names from Chapters 5 and 7 and Appendix B; return here after the pipeline closes.)

Fielding's clause (§5.1.5)	typed here as
identification of resources	`I` — one name space; R3 (Chapter 5), S4
manipulation of resources through representations	Definition 1.1 — `read` and `write` exchange representations; the delta normal form (Chapter 7) is the write's
self-descriptive messages	self-containedness — B-2d, at the message's scale
hypermedia as the engine of application state	the five moves (Chapter 7) — every transition a link in the document

5.5 The four timelines

Definition 4.3 has one more consequence, and it can be stated now. Nothing in this chapter has mentioned time. HTTP has. A representation, per RFC 9110 §3.2, reflects "a past, current, or desired state of a given resource": state *at a time*. The protocol also provides mechanisms for distinguishing a resource's representation at one moment from its representation at another: `Last-Modified` and `ETag` (RFC 9110 §8.8), and the caching calculus built on them (RFC 9111). The web's own specifications already treat the document as a sequence of states; the time index comes ready-made. Index the moving parts, writing `τ` for time, since `t` is taken:

```
S : Time → State                              the world, over time
read_τ = present_τ ∘ arrange_τ ∘ select_τ     the application, over time
doc(r, τ) = read_τ(r, S(τ))                   what the user agent renders
```

Four components can move: the state `S(τ)` and the three factors. Each movement has a name you already know. The state advances when someone writes, which is Chapter 7's whole subject. A factor changes in only one way, by substituting its term, and that is S3 read over time. The selection changes when a query is revised and redeployed, the arrangement when a layout switches or a template ships, the presentation when the theme changes.

And one thing that looks like movement is not: a user paging forward or tightening a filter changes nothing in the application. The filter travels in `r`, and `select` is the same term evaluated at a new argument. If a model makes the selection depend on time just to handle a mouse click, it has confused the function with its argument, a natural mistake, since `select_τ` offers a time subscript to anything that happens over time. The four timelines belong to the application; navigation belongs to the request.

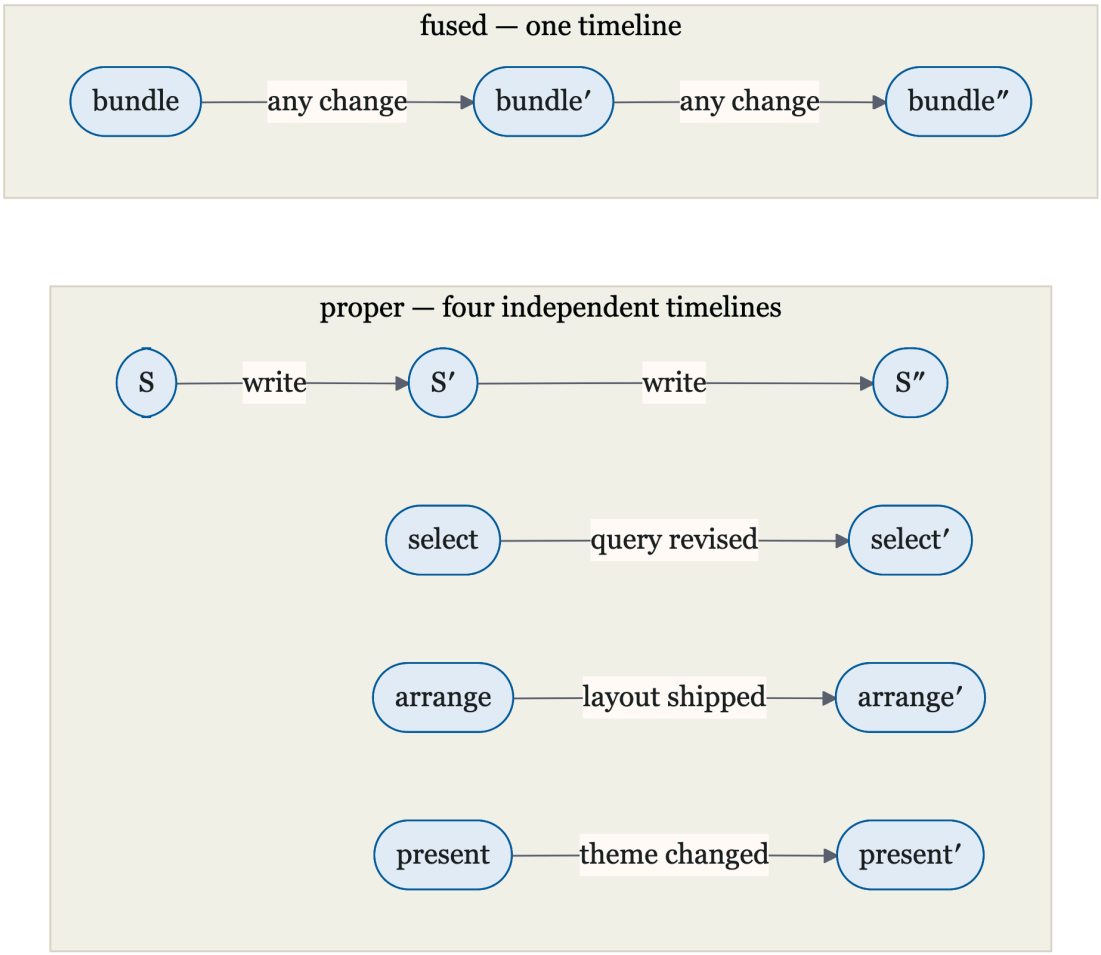
The dashboard makes it concrete. A panel lists alerts, twenty per page, and the user clicks *next*. In the proper factorization the click mints a new request: the page number travels in `r`, visible as a query parameter, and `select` is the same query that produced page one, now evaluated

at `page=2`. The new page is another value of the same function: it has its own URI, so it can be bookmarked, shared, cached, and fetched tomorrow with the same result. In the confused model the click increments a counter kept beside the query, a session variable or a widget's internal state. The selection now depends on when and by whom it is asked: the same URI shows you page two and shows the colleague you sent it to page one. The familiar symptoms follow: the back button misbehaves, the link does not travel, a refresh loses the page. All of them are one defect: the page number was stored in the application instead of carried by the request.

Prop. 4.5 (Independent evolution). In a proper factorization, the document's evolution decomposes into four independent timelines, one per component `(S, select, arrange, present)`: a change to any one component changes the document without requiring a change to, or the participation of, any other. In the fused factorization of Prop. 4.2 there is one component and therefore one timeline: every change, of whatever kind, is a change to the whole.

Ship a new theme on the dashboard and the panel data need not be fetched again. `present` moved on its own timeline while `S`, `select`, and `arrange` stayed put on theirs.

Dependencies — S1 confines, S2 closes, S3 substitutes; proof in Appendix B. Depends on 4.3: S1 confines a change's effect to its factor's output. S2 closes each term's semantics, so substituting a term of one language cannot alter the meaning of a term in another. S3 guarantees the substituted term still yields a web application. Proof: Appendix B.



Prop. 4.5, drawn. In the proper factorization each component advances alone, and caches invalidate per component; in the fused one every change, of whatever kind, is a change to the whole, and invalidates the whole.

Prop. 4.5's corollary is the first item on Fielding's list, caching per stage rather than per page, and S4 now supplies the mechanism for it. Under S4 each factor's output is a resource; each resource has a URI; each URI carries its own validator, an `ETag`, and its own timeline, and every cache on the path can read them. A theme change invalidates one stylesheet resource and leaves the data it styles cached at its own age, untouched. Collapse the factorization and there is one resource (the bundle) with one validator, and the corollary inverts: any change, of any kind, invalidates everything.

And the industry already operates all four timelines at the delivery layer. Fingerprinted stylesheets ship with `Cache-Control: immutable`, data responses are marked `no-store`, and templates are deployed on their own cadence. Every serious deployment on the web runs per-component timelines there, even when the application above is fused. Independent evolution already runs in production, one layer below the framework that obscures it.

One thread left dangling, on purpose: `select` selects *from State*, and State is still abstract. The factorization cannot be completed until we know what it is a factorization *over*. That is Chapter 5.

6. Chapter 5. What State Must Be

To make `State` concrete without choosing a preferred data model, this chapter derives three requirements from properties of the web itself.

6.1 The three requirements

R1 — Universality. The web hosts every application domain there is or will be. Therefore `State` must encode arbitrary application state, with no domain structure baked in. *(Source: the observable web. Try to name the domain the web is "for.")*

R2 — Coordination-free composition. The web has no central schema authority, by design; decentralization is what "world wide" means. Therefore state must be composable across independent parties who have never communicated. Composition without coordination has laws: it must accept any two states (checking compatibility is coordinating), in any order (agreeing an order is coordinating), with duplicates costing nothing (tracking copies is coordinating). And it preserves meaning only if the things composed are *self-contained*. A fact must carry its full meaning with it, because no surrounding structure survives a merge. Those four laws leave exactly one composition. That composition is set union, and Appendix B proves it:

```
State = 𝒫(Fact)        merge = ∪                              (5.1)
```

State is a *set of atomic facts*, and two states, from any two parties, anywhere, compose by union. Union is order-free, idempotent, associative, and commutative, so it has every property that federation needs.

R3 — Global reference. A fact on one site can be about an entity described on another; the web's entire point is that things link. Therefore names *inside* facts need global scope. The web possesses exactly one global naming system (`I`, the URIs from Chapter 1), and inventing a second one would itself violate R2 (two parties' private naming schemes collide on merge). So references in facts are drawn from `I`. R3 asks only for global names; nothing requires that they dereference. They *can*, though, and that comes free with the construction: the naming system and the web's address system are one. Chapter 19 confronts what that identification costs.

6.2 The one bridge

The scope of R2's result, before anything is built on it. The merge laws govern composition's *mechanics*: that any two states merge, without permission, in any order, at no cost. They do not promise that merged facts *join* — that two parties' names for one turbine ever meet in a query. No data model can promise that: parties who never communicated have agreed on nothing, in every model ever proposed, and a requirement pretending otherwise would be

a coordinator in disguise. A model does control two things: what an unjoined union already holds, and what a join requires once someone demands one. Chapter 18 takes that up, where federation stops being algebra and becomes deployment. Here it is enough to be exact about what R2 secures. It secures merge, completely, and it does not secure meaning across sources.

The passage from "no coordinator" to the merge laws is the one bridge in the derivation — the deliberate exception the Preface mentioned. I name it the **Transposition Thesis**. The web already enforces four rules on documents: anyone links to anything without asking, content arrives by any path and in any order, copies cost nothing and change nothing, and aggregators consume content outside its original arrangement. The thesis is that, applied to data instead of documents, the same four rules become the four merge laws, one for one: totality, order-freedom, idempotence, atomicity (B-2a–d, defined in Appendix B). The table below lays out the correspondence so that each of the four pairs can be checked on its own. The same laws were also derived from an unrelated direction: distributed-systems research, needing replicas to converge without coordination, derived them as theorems — the CRDT (conflict-free replicated data type) literature. Two fields with different motives reached the same algebra, so the merge laws are not a matter of taste.

the document layer, deployed	the state layer, transposed
anyone links to anything; no one is asked	composition is total — no compatibility check (B-2a)
content arrives by any path; intermediaries reorder it freely	composition is order-free (B-2b)
copies are free and unmarked; the cache hit *is* the resource	composition is idempotent (B-2c)
aggregators consume content outside its original arrangement, without its publisher's consent	no meaning survives in arrangement (B-2d)

Each left cell is deployed and citable — RFC 9111 carries the middle two, AWWW's global-identifiers principle the first, and the last is every search engine and feed reader in operation. Each right cell is a numbered condition in Appendix B; B.4 proves none is redundant. Appendix D carries the CRDT citation. State-based replication requires a join-semilattice, which means totality, order-freedom, and idempotence. That literature derives those three properties from replication pressure alone. The theorem that follows applies to the web only to the extent that this table holds.

6.3 The smallest fact

The question now is the smallest self-contained fact, and "smallest" is not an aesthetic preference. Every extra position in a fact is one more thing independent parties must agree on, and agreement is what R2 forbids. So minimality is R2 again, applied to the shape of the fact itself. When a genuine requirement justifies an extra position, the derivation will grant it; Chapter 9 does exactly that.

Prop. 5.2 (Arity). The minimal self-contained fact is a triple.

Argument — a pair cannot name its own relation: (employee42, “2026-07-08”) is hired-on, or fired-on, or born-on. A 1-tuple `(x)` asserts nothing — it names without claiming. A pair `(entity, value)` asserts a relation but cannot say *which* relation; the meaning lives outside the fact, which R2 forbids. Three positions, `(entity, attribute, value)`, is the first arity at which a fact names its own relation. And it is the last arity we need: any n-ary fact decomposes into triples by minting a fresh entity for the fact and attaching its n components as attributes. Minimality and universality pin the arity at exactly three. ■

R3 forces the entity position into `I`. It forces the *attribute* position into `I` too, because attributes need global names just as much. Without global names, two sources cannot know they mean the same property, and R2 fails at the first merge. The value position is either a reference or an atomic literal. Write `V` for the literals:

```
Fact  = I × I × (I ∪ V)                                  (5.3)
State = 𝒫(I × I × (I ∪ V))
```

Chapter 3's exhibit already wrote facts in this shape without saying so. The dashboard's strip-2 block held one entity and two attributes: two facts, exactly. Written with `⟨·⟩` for a URI abbreviated to its fragment, they are `(⟨…#panel-14⟩, title, "Current Power")` and `(⟨…#panel-14⟩, value, "15.5 kW")`. Entity and attribute are in `I`, and the value is in `V`. That exhibit already showed the theorem. A state in this shape is also a directed labeled graph by construction: entity and value terms are nodes, and each fact is an edge from its entity to its value, labeled with its attribute. When a value is itself in `I`, the edge lands on a node that other facts describe; that is how two parties' facts connect.

party A — the operator

```
panel-14 type Panel
panel-14 title "Current Power"
panel-14 value "15.5 kW"
panel-7 type Panel
panel-7 title "Wind Speed"
panel-7 value "8.2 m/s"
farm name "Anholt Offshore"
panel-14 partOf farm
panel-7 partOf farm
```

party B — the contractor

```
panel-14 value "15.5 kW"
panel-14 unit "kW"
turbine-3 type Turbine
turbine-3 label "A-03"
```

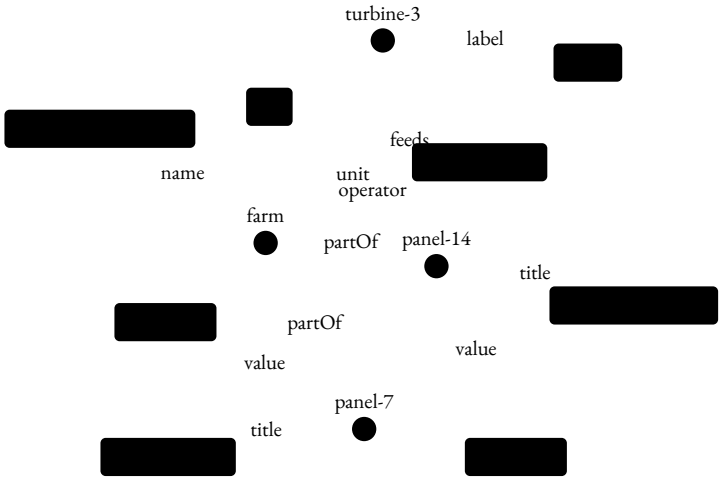
```
turbine-3 feeds panel-14
farm operator "Ørsted A/S"
```

15 facts asserted → |A ∪ B| = 14 — 1 duplicate collapsed; ∪ is idempotent, order never mattered

A ∪ B — one `State`; no coordinator was consulted

```
(⟨…#panel-14⟩, type, ⟨…#Panel⟩)
(⟨…#panel-14⟩, title, "Current Power")
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-7⟩, type, ⟨…#Panel⟩)
(⟨…#panel-7⟩, title, "Wind Speed")
(⟨…#panel-7⟩, value, "8.2 m/s")
(⟨…#farm⟩, name, "Anholt Offshore")
(⟨…#panel-14⟩, partOf, ⟨…#farm⟩)
(⟨…#panel-7⟩, partOf, ⟨…#farm⟩)
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-14⟩, unit, "kW")
(⟨…#turbine-3⟩, type, ⟨…#Turbine⟩)
(⟨…#turbine-3⟩, label, "A-03")
(⟨…#turbine-3⟩, feeds, ⟨…#panel-14⟩)
(⟨…#farm⟩, operator, "Ørsted A/S")
```

the same `State`, as the graph it is



Interactive exhibit (online edition): two parties who have never met. Edit either side, shuffle, duplicate. The union changes only when you add a genuinely new fact, and (5.1) is something you fail to break rather than something you believe. The graph panel draws the same state; delete the contractor's `feeds` fact and it falls apart into one star per entity, then restore it and the stars join.

6.4 The uniqueness theorem

Theorem 5.4 (Uniqueness). Any arity-minimal state model satisfying R1–R3 is isomorphic to (5.3). *(Proof: Appendix B. The proof is an assembly of 5.1–5.3: R2 forces the set-of-atomic-facts shape and union-merge; R1 with minimality forces arity three; R3 forces positions one and two into `I`.)*

It sounds like rhetoric to claim that (5.3) is the only shape the web itself permits. Theorem 5.4 makes it exact: rejecting the conclusion means faulting a step of the proof (Appendix B lays the steps out for that attack) or rejecting a requirement, and each rejection has a name. Reject R1 and your model can't host the web's content. Reject R2 and your data needs a coordinator, which is a central schema authority, so you have built a silo. Reject R3 and your data cannot refer beyond itself, which is a silo again. Reject minimality and you widen the tuple, the one rejection left deliberately open; Chapter 9 takes it, adding a fourth requirement, attribution. Part IV will locate every alternative data model the industry runs on at one of the first three rejections.

Machine-checked. The spine of this chapter's proofs — the union law, the arity bound, and this theorem's assembly — is verified in Lean 4, a proof assistant, and the verification lives in the book's repository. Appendix C reports what is checked and what must be assumed.

6.5 The selection algebra

We are not done deriving. S2 committed `select` to being the meaning of a term in a fixed language, and a language's semantics is a set of operations over a carrier. The carrier is now fixed: `𝒫(Fact)`. What remains is the operations, and four are enough for everything the derivation builds on: match a fact pattern with variables; join matches; union alternatives; project variables out. Each is forced by a page you can point at (any master–detail page is a join; any search page is a pattern; any page merging two lists, events from either calendar, is a union; any list that shows a title per entity, and nothing else, is a projection). None of the four is expressible from the other three, so none can be dropped. Pages force more — comparisons, counts, order — and Part III's deployed language has them; the derivation stops at four because nothing that follows depends on the rest. The algebra reappears in Part III under its deployed name; what a `write` can do to a fact-set is Chapter 7's subject.

Run one join by hand. The dashboard's operator holds `panel-14 title "Current Power"`; the turbine contractor (another party entirely) holds `turbine-3 feeds panel-14`. Merged, the pattern (triples written bare now, URIs still abbreviated, `?` marking a variable)

```
?turbine  feeds  ?panel
?panel    title  ?name
```

returns one row (`?turbine = turbine-3`, `?panel = panel-14`, `?name = "Current Power"`) and the row exists only because the two parties' facts were merged first. One page exercises the whole algebra: match, join, and project.

?p	?t	?v
⟨…#panel-14⟩	"Current Power"	"15.5 kW"
⟨…#panel-7⟩	"Wind Speed"	"8.2 m/s"

pattern — `?x` binds

```
?p type Panel
?p title ?t
?p value ?v
```

solutions — 2 rows · join of 3 patterns

?p	?t	?v
⟨…#panel-14⟩	"Current Power"	"15.5 kW"
⟨…#panel-7⟩	"Wind Speed"	"8.2 m/s"

`Data` — the sub-state the solutions touched

```
(⟨…#panel-14⟩, type, ⟨…#Panel⟩)
(⟨…#panel-14⟩, title, "Current Power")
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-7⟩, type, ⟨…#Panel⟩)
(⟨…#panel-7⟩, title, "Wind Speed")
(⟨…#panel-7⟩, value, "8.2 m/s")
```

Interactive exhibit (online edition): the algebra exercised — patterns with variables, joined and projected over the state merged above. One preset only answers because the merge happened: it joins the operator's facts to the contractor's.

The join also exposes a problem in the pipeline's types. `State` is the graph that (5.3) built, and the pattern just walked it, crossing from the contractor's facts to the operator's along a reference, from `feeds` into `panel-14` and on to its `title`. References point anywhere. But in Chapter 3 every stripped document was a tree. Somewhere between them, the shape must change. That is Chapter 6.

7. Chapter 6. From Graph to Tree

Chapter 5 closed on a mismatch of shapes; the pipeline's types locate it. `State` and `Data` are *graphs*, facts whose references form arbitrary many-to-many structures. `Tree` and `Doc` are *trees*. Documents are hierarchical, and so is human reading. The pipeline crosses from graph to tree exactly once, inside `arrange`.

In the world. The tension this chapter resolves is older than the web. In 1945 Vannevar Bush blamed our trouble finding anything on "the artificiality of systems of indexing": records "filed alphabetically or numerically," found "by tracing it down from subclass to subclass." The mind, he wrote, instead "operates by association." Ted Nelson put the same objection in capitals in 1974: "EVERYTHING IS DEEPLY INTERTWINGLED. In an important sense there are no 'subjects' at all." Both were describing a graph and refusing the tree. This chapter keeps both. The association is what `State` is. The tree is only what a document must become to cross the wire and be read.

Every web framework in history is a strategy for this one crossing. That is an observation you can verify against the deployed web, and Part IV verifies it, framework by framework. But the crossing is not yet well-defined. Graph-to-tree serialization is a *relation*, not a function: one graph, many trees (orderings, nestings, groupings). Yet Chapter 4 typed `arrange` as a function without saying where the choice among the trees lives. The fix is canonicalization:

```
arrange = ⟦t⟧ ∘ canon
canon : Data → Tree      deterministic, lossless, structure-free
⟦t⟧   : Tree → Tree     t — the sole locus of graph→tree structural choice
```

`canon`'s output is the graph in bare tree form — one block per entity, sorted, no nesting, no sugar. *All* structural decisions (what nests under what, what becomes a section versus a sidebar) move into the declarative term `t`, where S2 can hold.

Display order is one of those structural decisions, and it gets the strictest handling: `canon` sorts blocks by a fixed key that means nothing. The order of the blocks is therefore determined by the facts alone, and an order determined by the facts cannot encode a choice about them: two editors holding the same facts get the same block order, whichever story each meant to lead. If the order is a choice, like the front page's lead story in Chapter 3, it must be stated as one more fact (*this article, prominence one*); the two editors now hold different facts, and `t`, reading them, gives each front page its own lead. Chapter 3 made exactly this concession and promised the law; here it is: if the order is a message, the order is data. The historically hard case is facts about *unnamed* entities: `canon`'s sort key is built from names, and a nameless entity offers none. As of 2024 the case has a standardized deterministic answer, a canonical labeling, which

manufactures the missing names. Chapter 9 explains why the model admits unnamed entities at all, what admitting them costs, and names the spec.

One graph, many trees: `d` can nest under only one parent (the other edge survives as a reference) and siblings take an order the graph never fixed. These are the choices the crossing must make somewhere: `canon` makes none of them, `t` makes all of them.

And you have already seen `canon`'s output. Strip 2 *is* it: the dashboard reduced to sorted blocks, one per panel, `title` and `value` beneath. The exhibit's format was the canonical serialization, arrived at by stripping, not a design choice.

`canon` needs three properties (deterministic, lossless, structure-free), and all three are satisfiable, cheaply:

Prop. 6.1. A `canon` with all three properties exists.

Proof — sort lexicographically; unnamed entities need Chapter 9's labeling. For ground states (states whose entities all carry names) order the facts lexicographically by their three positions and emit one block per entity. The map is a function because a total order on tuples exists. It is lossless because the fact set is recoverable by reading the blocks back. It is structure-free because the order is defined by the facts alone, never by their provenance or grouping. States with unnamed entities need a canonical labeling first; that labeling exists, is standardized, and Chapter 9 states its cost. ■

facts, in arrival order

```
farm name "Anholt Offshore"
farm operator "Ørsted A/S"
panel-14 partOf farm
panel-14 title "Current Power"
panel-14 type Panel
panel-14 unit "kW"
panel-14 value "15.5 kW"
panel-7 partOf farm
panel-7 title "Wind Speed"
panel-7 type Panel
panel-7 value "8.2 m/s"
turbine-3 feeds panel-14
turbine-3 label "A-03"
turbine-3 type Turbine
```

`canon` — one block per entity, sorted, no nesting

```
(…#farm)
    name · "Anholt Offshore"
    operator · "Ørsted A/S"
```

```
⟨…#panel-14⟩
    partOf · ⟨…#farm⟩
    title · "Current Power"
    type · ⟨…#Panel⟩
    unit · "kW"
    value · "15.5 kW"

⟨…#panel-7⟩
    partOf · ⟨…#farm⟩
    title · "Wind Speed"
    type · ⟨…#Panel⟩
    value · "8.2 m/s"

⟨…#turbine-3⟩
    feeds · ⟨…#panel-14⟩
    label · "A-03"
    type · ⟨…#Turbine⟩
```

`canon` is a function of the atoms alone — not of their order, grouping, or provenance.

Interactive exhibit (online edition): shuffle the input as many times as patience allows. `canon` does not move. Change a fact and it moves exactly as far as the fact requires.

The graph→tree crossing. A relation (one graph, many trees) becomes a function followed by a term: `canon` chooses nothing, `t` chooses everything.

The seam is not an unsolved research problem, and Chapter 9 presents the evidence for that. And with the crossing fixed, the read side is derived end to end — Definition 1.1 has one component left.

8. Chapter 7. The Write Side

Definition 1.1 has a second component, and everything derived so far concerned the first. `read` was the easy half: documents can be declarative, because a page that only displays needs no program. Applications also change things, and change is widely held to be where declarative architectures fail. This chapter derives the write side in four propositions. Nothing from the read pipeline needs to be taken back; the write side is the factorization's mirror image, and the smaller of the two.

8.1 The delta normal form

Start where Chapter 5 left the state: `State = 𝒫(Fact)`, merge is union. What can change about a set? Elements are removed and elements are added. There is no third possibility.

Prop. 7.1 (Delta normal form). Every state change factors as a pair of fact-sets, a *delta*:

```
write(r, S) = (S \ D⁻) ∪ D⁺                                  (7.1)
D⁻ = S \ write(r, S)          the facts removed
D⁺ = write(r, S) \ S          the facts added
```

Proof — extensionality, then minimality. With `D⁻`, `D⁺` as defined, `(S \ D⁻) ∪ D⁺ = write(r, S)` by set extensionality. Minimality: any pair `(A, B)` with `(S \ A) ∪ B = write(r, S)` satisfies `A ⊇ D⁻` (a fact of `S` absent from the result leaves only by removal) and `B ⊇ D⁺` (a fact new in the result enters only by addition). So `(D⁻, D⁺)` is contained in every such pair: it is the least pair, and a least element is unique. ■

A state change is two sets. That is the entire theory of mutation over a fact-set model. Measure that against what the industry maintains for mutation: object-relational mappers, undo stacks, reconciliation engines. Every one of these is machinery for computing or applying change over a model in which change has no normal form. Trees are the instructive case: two trees have no canonical difference, so deciding what "changed" is a heuristic. The industry's clearest specimen is the virtual DOM's diffing engine (Chapter 11): client-side frameworks re-compare an in-memory copy of the page on every render. That is a whole runtime recovering, approximately, what (7.1) gives exactly by subtraction. Mutation's normal form is therefore a by-product of the model R2 forced for merging, because union and difference belong to one algebra.

On the running example: the wind gusts, and the panel's `value` moves. The delta is `D⁻ = {(⟨…#panel-14⟩, value, "15.5 kW")}` and `D⁺ = {(⟨…#panel-14⟩, value, "16.1 kW")}`, two one-element sets. That is the entire update, transport included.

`S` — the `State`

```
(⟨…#panel-14⟩, title, "Current Power")
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-7⟩, title, "Wind Speed")
(⟨…#panel-7⟩, value, "8.2 m/s")
```

`D⁻` — the facts removed

```
panel-14 value "15.5 kW"
```

`D⁺` — the facts added

```
panel-14 value "16.1 kW"
```

the wind gusts; the delta is two one-element sets.

Interactive exhibit (online edition): the gust, applied. Edit the two sets and apply them against the live state. Apply the same delta twice and watch nothing happen: sets subtract, and re-application is a no-op.

A delta is made of fact-sets. Change is data in the same model as the state it changes. There is no second model, and no change-description language whose semantics would have to be invented. The delta is what fills the request's `Body` from Chapter 1, and Chapter 8 will give the industry's name for it.

8.2 Forms, run backwards

Prop. 7.2 (Forms are inverse transforms). The read pipeline ends at a human; the write side begins at one. The instrument is a form: a tree, part of a document, rendered by `present` like everything else. Its fields are a fact pattern's variables. A form carries one pattern or several, each marked "remove" or "add"; submission binds their fields, and a bound pattern is a set of facts. The marked sets are (7.1)'s delta:

```
form   : Tree                fields ↔ variables of a fact pattern
submit : Bindings → (D⁻, D⁺)                                (7.2)
```

On the running example: the edit form arrives holding "15.5 kW"; the user types "16.1"; the stale binding, marked remove, is `D⁻`, and the fresh binding, marked add, is `D⁺`. One submission produces both sets.

A form is `arrange` run backwards. The one factor that crossed graph→tree (Chapter 6) is also the one that must cross back, and it crosses back using patterns, exactly as before.

Transforms are known to be hard to run backwards. Databases call it view update: given a change to a view, find the change to the data underneath that produces it. That change need not exist, and need not be unique. The programming-language answer is lenses, a forward transform paired with a backward one that must satisfy round-trip laws. The ambiguity does not arise here, because the inverse is declared rather than computed. A form's fields are a pattern's

variables, so the backward direction is fixed when the pattern is written. The restriction comes from S2: the read side is terms of a fixed algebra, never arbitrary functions. Appendix D carries the citations.

A form does two jobs, and they are different in kind. *Construction* decides which fields an edit form offers for an entity of this kind; it is a projection of structure: read the patterns, render inputs. *Validation* decides which deltas are admissible; it is a predicate on `(D⁻, D⁺)`. A schema drafted to do both jobs at once will do both badly. Chapter 20 builds on that point.

For the justification, view source on any HTML form since 1993. Field names are attribute names; `method` names the unsafe verb; the form is a fact pattern whose variables are input boxes. The web has shipped the inverse transform beside the forward one from the beginning.

8.3 One algebra, both directions

Prop. 7.3 (One algebra, both directions). Chapter 5's selection algebra (match, join, union, project) is the write side's algebra too. A pattern with free variables *selects* the facts that match; the same pattern with its variables bound *denotes* the facts of a delta.

```
pattern + state      →  bindings                    (find)
pattern + bindings   →  (D⁻, D⁺)                    (change)
```

The consequence is practical as much as formal: the write side adds no expressive machinery. Whoever can query can update: an implementation gets its update language by handing its pattern matcher the bindings it would otherwise return. When Part III proves the read side complete, the write side inherits the result through this symmetry. Compare, once more, the industry's arrangement. It uses a query language, a separate mutation API, a migration DSL, and a client-side state manager. That is four vocabularies for one algebra.

8.4 The five moves

Prop. 7.4 (Interactivity, decomposed). The objection's strongest form: "real applications are interactive." By independent evolution (Prop. 4.5) the document is `doc(r, τ) = read_τ(r, S(τ))`. That is a value with exactly five inputs: the request `r`, the state `S(τ)`, and the three factor terms. So every interaction ever shipped on the web is one of exactly five moves:

1. **navigate** — a new `r`: link, filter, page, search. The term stays the same; the argument changes.
2. **write** — `S` advances by a delta (7.1): submit, edit, delete.
3. **restyle** — substitute the `present` term: the theme toggle.
4. **rearrange** — substitute the `arrange` term: list to grid, sort, collapse.
5. **reselect** — substitute the `select` term: a saved query edited, a dashboard reconfigured.

Moves 2–5 are independent evolution's four timelines; move 1 is the request, an argument to the application, not a component of it (Chapter 4). There is no sixth move because there is no sixth input. Interactivity *is* the factorization being exercised. Fusion adds no move to this list;

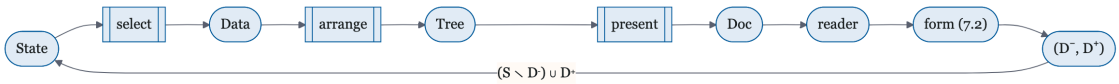
what it gains is the freedom to make the moves without saying which component they touch. Part IV will show what that freedom costs.

8.5 The latency concession

One concession remains, latency, and it too should be met at full strength. The operational complaint is the round trip: a keystroke should not cross an ocean to move a cursor. Granted. But look at what the complaint actually asks for: that the factors be *evaluated near the user*, not that they be fused. And mobility of evaluation is precisely what S2 already secured. A term whose semantics is closed evaluates the same everywhere. So ship `q`, `t`, `s` to the client and run them there, against a local replica of the selected data. No proposition of the architecture has changed. The terms and the factors are the same; only the machine is different. What cannot travel this way is a fused `read`: an opaque program can only be shipped whole and trusted blind. Shipping opaque programs to browsers is an experiment the web has already run; Chapter 12 reports the result. Declarative terms are portable because they mean the same thing everywhere. That is S2, applied as a deployment strategy.

8.6 The closed pipeline

The pipeline is now complete in both directions, and it closes:



The closed pipeline. Three factors turn state into a document, a human reads that document, the human's answer is a delta, and that delta becomes the next state as `τ` ticks (4.5). Every arrow but the human's is a numbered proposition. Everything before this figure derives it; everything after measures the world against it.

Change, the objection said, is where declarative architectures fail. In the diagram, change is the arrow that closes the pipeline.

III Part III — The Reveal

The derivation is complete (a data model, an algebra, a crossing, a write side), and none of it has been named. This part names it, then sets out the mismatches. The names are decades old.

Chapter 8 It Already Exists Chapter 9 The Mismatches

9. Chapter 8. It Already Exists

I would definitely point to RDF as prior art in terms of thinking about properties, independent of aggregates.

— Rich Hickey, creator of Clojure (*Cognicast*, 2016)

Everything in Part II was derived from three RFC-level definitions (identifiers, requests, responses) and three requirements. No W3C recommendation has entered as a premise. Where AWWW is cited (Chapter 4's good practices, Chapter 5's transposition table), it is cited as corroborating evidence, never as a premise. No vocabulary from any data-model community has appeared. The standard names follow:

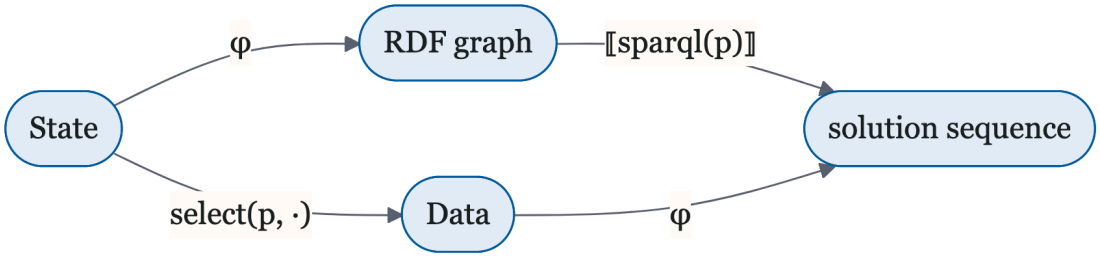
Derived in Part II	Standardized as	Since
`Fact = I × I × (I ∪ V)`	RDF (Resource Description Framework) triple (subject, predicate, object)	1999 / RDF 1.1 2014
`State = 𝒫(Fact)`, merge = ∪	RDF graph; graph merge	1999 / RDF 1.1 2014
selection algebra (pattern, join, union, project)	SPARQL algebra, §18 (Basic Graph Pattern, Join, Union, Project)	2013
delta `(D⁻, D⁺)`	SPARQL Update (`DELETE`/`INSERT`)	2013
dereferencing `select` results (S4)	Linked Data; Graph Store Protocol (HTTP methods addressed to whole graphs)	2006 / 2013
`canon`	canonical RDF/XML; RDFC-1.0 (RDF Dataset Canonicalization) for the unnamed entities (*blank nodes*)	2004 / 2024
`⟦t⟧ : Tree → Tree` after canon	XSLT	1999 / 3.0 2017
`present`	CSS	1996

Table 8.1. The correspondence.

Read the dates first. Every row predates this book, most by decades, and none appears anywhere in Parts I–II. There is one conceded exception: Chapter 3 turns CSS off by name, and turning it off is the act every reader knows it by. The derivation's premises are the RFC layer only, so the match in this table is a check the reader performs, not a construction the author arranged. The columns are independent: the left side is forced by three requirements, and the right side was shipped by working groups. The table asserts they are the same objects.

Prop. 8.1 (Homomorphism). There is a translation `φ`, taking facts to RDF triples, states to graphs, and selection terms to SPARQL terms (write `sparql(p)` for that last translation), such that `φ(select(p, S)) = ⟦sparql(p)⟧(φ(S))`; on ground states `φ` is a bijection. The mapping is a *homomorphism*, not a coincidence of shapes: the operations commute with the translation.

How the check runs — clause by clause against a denotational spec. The equation `φ(select(p, S)) = ⟦sparql(p)⟧(φ(S))` is checkable clause by clause against the SPARQL algebra. The algebra is written denotationally (a rarity among web specs, shared mainly with XQuery's Formal Semantics) and that denotational form is what makes the check possible. Full proof: Appendix B.7.



Prop. 8.1. Two paths give one result: translate then query, or query then translate (the standard calls the result a solution sequence). The square commutes.

You have already accepted RDF. You did it in Chapter 5, before I told you its name.

Whatever you believed about the semantic web when you opened this book (too academic, too complicated, died in the nineties), you derived it yourself. You derived it from three requirements, and you can reject each one only at the cost that requirement itself states. The technologies were not a committee's enthusiasm in search of a problem. They occupy a position that was forced, and the people who standardized them in 1999 had already arrived there. What failed in the nineties was not the position. The tooling failed, and the timing was wrong, because the substrate is built for machine consumption and the machines that could consume it were twenty years away. Chapter 18 makes the demand-side case.

the objects, as the derivation wrote them

`State` = $\mathcal{P}$(Fact) — (5.3)

```
(⟨…#panel-14⟩, title, "Current Power")
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-7⟩, title, "Wind Speed")
(⟨…#panel-7⟩, value, "8.2 m/s")
```

the selection algebra — pattern, join, project

```
(?p, type, ⟨…#Panel⟩)
(?p, title, ?t)
(?p, value, ?v)
```

the delta — (7.1)

```
D⁻ = { (⟨…#panel-14⟩, value, "15.5 kW") }
D⁺ = { (⟨…#panel-14⟩, value, "16.1 kW") }
```

Interactive exhibit (online edition): Chapter 5's state, Chapter 5's algebra, and Chapter 7's delta sit under one switch. One side shows the derivation's notation; the other shows Turtle (the standard's text notation for triples), SPARQL, and SPARQL Update. Nothing is recomputed. Everything is renamed.

Theorem 8.2 (Synthesis). The stack realizes the proper factorizations whose `select` is a term of Chapter 5's algebra and whose `arrange` is generic, invariant under URI renaming, singling out no particular URI. SPARQL carries those selections exactly (Prop. 8.1), XSLT-over-canon the arrangements, CSS the presentation, and S4 holds by construction because query results and graphs are dereferenceable resources. By Prop. 7.3's symmetry, the write side inherits the result: SPARQL Update carries the deltas in the same pattern language, arguments swapped. Full proof: Appendix B.8, which states the genericity condition exactly.

The select-side condition restricts nothing new: it admits exactly the selections Chapter 5's four operations build, and no more. The genericity condition is the claim's deeper caveat, carried inside it: the completeness class for `arrange` excludes transforms that smuggle in knowledge of particular URIs. The exclusion has a reason: such a transform makes the document depend on how an entity is named rather than on what the facts say. A renaming, which asserts nothing, then changes the page, and the page carries meaning with no source in the state. Practitioners call a transform without such knowledge data-driven, and AWWW §2.5 calls the norm URI opacity; invariance under renaming is both, made exact. Analysis said every application has the form; Synthesis says the stack fills the form. The two halves of the argument meet, and that meeting is the book's proof.

10. Chapter 9. The Mismatches

RDF is painfully simplistic, but it allows you to work with real-world data and problems that are horribly complicated.

— Dan Brickley and Libby Miller, foreword to *Validating RDF Data*

Chapter 8 put both halves of the argument into theorem form: every application has the derived form, and the deployed standards fill it. That invites suspicion. An exact match to a deployed stack looks retrofitted until the mismatches are listed openly, so this chapter lists them. The fit is not exact. There are two mismatches between the model Part II forced and the standard Part III named, and two more between the standard and the platform that ships it, the browser. Each is located; the first two are measured (Props. 9.1–9.2), and one of them is turned into a prediction the standard later honored. Part IV holds everyone else's models to the same test.

10.1 Mismatch one: the unnamed entities

RDF permits facts about entities with no name — blank nodes. Nothing in Chapter 5 forced them: the derivation minted a fresh URI wherever it needed an entity, because minting a fresh URI costs nothing. So blank nodes are surplus, and the surplus has a precise reading. A graph containing `_:b` asserts *that something exists* with these properties; RDF's own semantics says exactly this: simple entailment treats blank nodes as existential variables. The extension is well-motivated. Entities routinely exist before anyone names them. A form not yet submitted and an observation not yet reconciled each describe an entity that has no name. So a model that forbade the unnamed would fail R1 at the margins of every domain.

The cost can be stated exactly. Write $\oplus$ for the merge of two states.

Prop. 9.1. Over ground facts, merge is plain set union ($\oplus = \cup$) and Chapter 5's four merge laws hold on the nose: totality, order-freedom, idempotence, atomicity (B-2a–d). With blank nodes, idempotence and atomicity hold up to logical equivalence, and only up to logical equivalence.

Proof — merging a graph with itself doubles the existentials: equivalence survives, identity does not. Blank nodes are scoped to their graph, so composition must standardize them apart: $s \oplus s$ carries two copies of each existential. The result asserts nothing new (it entails s and is entailed by it), so $s \oplus s \equiv s$. But as a set of atoms it is strictly larger, so $s \oplus s \neq s$. B-2c survives semantically and fails syntactically. Atomicity bends the same way: an atom containing a blank node means something only together with the atoms sharing its variable, so self-containedness holds per connected component, no longer per atom. Restoring identity from equivalence costs exactly two computations: canonical labeling (standardized in 2024 as RDFC-1.0, deterministic, with adversarial worst cases the spec itself documents) and redundancy elimination, which is coNP-complete in general. ■

That is the bill for anonymity, and it falls exactly on the party that chose anonymity. If you name your entities, state composes by set arithmetic. If you leave them unnamed, composition becomes theorem-proving in miniature, and it lands precisely at the seam where Chapter 6 put `canon`. The deployed stack's own list idiom shows the cost: encode an ordered collection as a chain of unnamed cells and every link carries the cost of anonymity. Order is the recurring case. If you state order as facts, using one rank fact per member as Chapter 3′s opening example did, then order merges like any other facts. If you fold order into shape instead, you have the chain just described, and the union of two chains is not a chain, so the order does not merge at all.

10.2 Mismatch two: the fourth position

The web adds one requirement that Chapter 5 never imposed. R1–R3 govern facts about the world, and the web also carries *claims* about those facts: one source asserts a fact and another disputes it, and the web records provenance, retraction, and trust. Call it R4:

R4 — Attribution. Facts about who asserts facts.

Prop. 9.2. The arity-minimal state model satisfying R1–R4 is $\mathcal{P}$`(I × Fact)` — quads.

Proof — set union forgets who contributed; reification attributes only descriptions; one position repairs it. Union erases contribution: B-2d says `atoms(s ⊕ s′) = atoms(s) ∪ atoms(s′)`, and a set union keeps no record of which side an element came from. So within $\mathcal{P}$`(Fact)`, "who asserted this atom" is unrecoverable by construction — attribution lives in the history of the state, and states-not-histories is what B-2c chose. Reification (the standard's device of describing a fact in triples of its own) does not escape either: it attributes only a *description* of the fact. The described fact is then either also present as a plain atom or absent. If present, it is asserted outright and the attribution is defeated. If absent, it is attributed but never stated, quoted rather than asserted. The minimal repair types the atom as a pair (source, fact). The source position must refer across parties, hence lies in `I`; this is R3′s argument (condition B-3 in Appendix B), verbatim. One extra position suffices, because attribution of attributions is more quads, not more positions. Rerun B.1–B.3 over the retyped atom: $\mathcal{P}$`(I × I × I × (I ∪ V))`, merge still union. ■

Here the mismatch becomes a prediction. The 1999 core standardized triples. The deployed stack then grew exactly the fourth position: named graphs, RDF datasets, TriG (their text notation), standardized in 2014. The graph name is a URI, so attribution itself dereferences. A derivation that merely matched the 1999 core could be coincidence. But the derivation requires attribution, which the 1999 core did not satisfy, and the standard's own later extension added exactly the position attribution needs. A derivation that anticipates the artifact's next move is tracking the requirements, not copying the artifact. The fourth position exists, but the standard did not fix what the graph name *means*, and its semantics is still argued about. The prediction is structural, and claimed as nothing more. (Annotation syntaxes such as RDF 1.2′s triple term, by contrast, only re-serialize what reification already expressed. They are a convenience, because syntax is not a property and Part IV's audit table scores properties.)

10.3 Mismatch three: the abandoned seam

Chapter 6's crossing needs two pieces: a canonical serialization and a declarative tree-transformation language. The deployed stack had both, and the transformation language (XSLT) was standardized in 1999 and shipped in every browser. The platform then froze it at that 1999 revision for a quarter of a century and, as of this writing, is scheduled to remove it outright. So this mismatch is not a gap in the standards: the technology existed, and the platform stopped maintaining it. This is the evidence Chapter 6 promised: the graph-to-tree crossing needs no invention, only upkeep of software that already existed. The platform declined that upkeep, and the industry built, many times over, the compensating machinery Part IV measures.

Stated generally, that is the book's practical thesis: what separates the modern web from the derived one is abandoned technology, a maintenance failure rather than a research problem. And the failure is the platform's, not the language's: XSLT 3.0 (2017) runs in every current browser through SaxonJS. Its IXSL extension binds browser events to template rules, so Chapter 7's mobility of evaluation is deployed today and interactivity stays declarative. From userland, a vendor performs the maintenance the platform dropped.

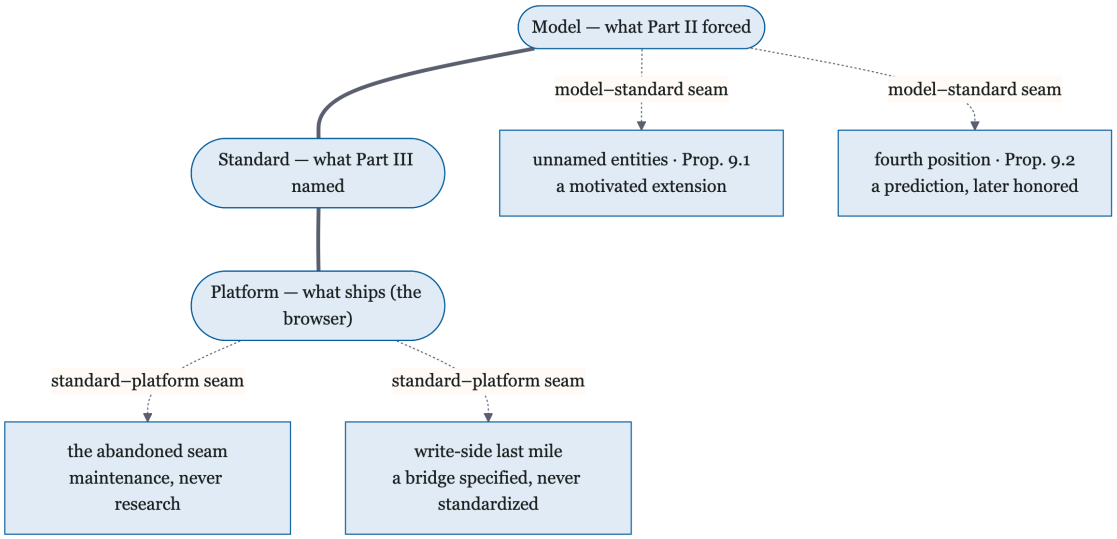
10.4 Mismatch four: the write-side last mile

Forms, run backwards, want submissions that denote deltas (Prop. 7.2). The W3C Recommendation stack stops one step short: SPARQL Update carries the delta, but HTML forms speak `application/x-www-form-urlencoded`, and no recommendation bridges the two. The bridge exists as a community spec, RDF/POST. It flattens the triple positions into form keys (`su`, `pu`, `ou`, `ol`, ...: subject, predicate, object, literal), so that a plain HTML form, with no script, submits a graph. It adds no new data model; it is an encoding of the derived model into the form media type the web already ships. Its non-standardization is a remaining gap on the write side. (Disclosure: the spec is maintained by the author's company, building on Sergei Egorov's original draft. Chapter 19 shows it at work.)

10.5 The inventory

Four mismatches, then:

mismatch	seam	resolves as
the unnamed entities (Prop. 9.1)	model ↔ standard	a motivated extension, its cost computable and falling on whoever chose anonymity
the fourth position (Prop. 9.2)	model ↔ standard	a prediction the standard later honored
the abandoned seam	standard ↔ platform	abandonment — maintenance, never research
the write-side last mile	standard ↔ platform	a bridge specified, never standardized



*The four mismatches. The derivation meets the world at two seams: the **model** (Part II) against the **standard** (Part III), and the standard against the **platform** that ships it. At the model–standard seam (Props. 9.1–9.2): the standard permits unnamed entities the model never required, and the standard's fourth position honors a requirement the model missed. At the standard–platform seam: the platform abandoned a technology it had shipped, and a bridge that was specified but never standardized. Fixing either platform mismatch requires no new invention, only maintenance and standardization.*

None of these mismatches invalidates a proposition from Part II. And listing them matters: an audit is credible only if the auditor's own defects are on record first. With the inventory complete, Part IV applies the same requirements to other architectures.

IV Part IV — The Audit

Part II derived seven properties; Part III showed that one stack satisfies all of them. This part scores everything else the industry runs against the same seven properties, and finally the derived stack itself.

Every audit in this part fills one column of the same table; the final chapter assembles the complete table. The rows are R1 R2 R3 · S1 S2 S3 S4. Each technology is evaluated against those seven properties, and the scores carry the argument. The rows themselves are backed by proofs: R1–R3 by Theorem 5.4, S1–S4 by Proposition 4.4. So rejecting a score means rejecting a requirement, not defending a technology.

11. Chapter 10. Brackets

Since the early 2000s, the web community has executed the largest format migration in its history: XML to JSON. That migration rewrote APIs, retired toolchains, and retrained developers over two decades. And what changed?

1999	2019
`<person><name>Ada</name></person>`	`{"person":{"name":"Ada"}}`

Twenty years of progress.

Activity of this shape is **lateral churn**, motion that looks like innovation but isn't. Real innovation is vertical: new layers, new semantics, new abstractions on top of what already works. That is how the web was designed to grow. Parts II and III have shown, formally, that the vertical direction was open the entire time.

That historical claim can now be tested: if XML→JSON was lateral churn — a change of syntax presented as a change of substance — the seven properties should score both formats the same.

Both formats share one data model. An XML document and a JSON document are ordered labeled trees; their differences (attributes versus members, elements versus arrays) decorate the same structure. Chapter 5's requirements apply to the structure, so the scores transfer wherever a requirement sees only the tree. R3 is the exception, and the formats part ways there.

R2. Trees have no coordination-free merge. Two JSON documents have no defined composition at all. Concatenation is not valid JSON, and "deep merge" is a per-application policy (which key wins, whether arrays append or replace). And a policy shared between parties is coordination. In the terms of Appendix B, meaning lives in arrangement (position, nesting, order). That rejects B-2d (atomicity: a state says exactly what its atoms say). So the representation lemma (B.1), the proof behind the union law, never gets started. ✗

R3. JSON has no reference type. A URL in a JSON string is a string; the format's specifications define grammar and leave interpretation to applications, so nothing distinguishes a link from a postcode. XML held fragments of the property. Namespaces gave vocabulary terms global names; `xml:id` and XLink offered standardized reference, largely unused. The migration shed the fragments too. AWWW §4.4 named three good practices in 2004 (link identification, Web-wide linking, hypertext links), and all three fail in the format the industry migrated *to*. The destination format, JSON, scores ✗; the origin, XML, keeps a ~ for its surviving fragments.

11.1 The tooling gap

capability	XML stack	JSON stack
schema	XSD (XML Schema), 2001	JSON Schema — drafts since 2010, still a draft
query	XPath, 1999	JSONPath — RFC 9535, 2024
transformation	XSLT, 1999	—
intra-document addressing	fragments + XPointer	JSON Pointer — RFC 6901, 2013
vocabulary scoping	Namespaces, 1999	—

The JSON stack's query language was standardized in 2024, twenty-five years after XPath, and it has no transformation or vocabulary-scoping language at all. The transformation role is filled by whatever framework is current (the table's empty cell), and most JavaScript frameworks of 2010 have already been retired. The browsers froze XSLT at its 1999 revision; it still runs in every one of them as of this writing.

11.2 XML stack

The R-rows carry over from the argument above: the tree encodes any domain, the merge is missing, and the reference fragments go unused. The S-rows come from the tooling table. XPath and XSLT give query and transformation specified semantics, so S2 holds. Each capability has one standardized language, so components substitute and S3 holds. S1 is partial: a schema separates the vocabulary, but the document still fuses arrangement and data. S4 is partial: fragments and XPointer address into a document, but they are largely unused.

	XML stack
R1	✓ — trees encode any domain
R2	✗ — no coordination-free merge; meaning lives in arrangement
R3	~ — namespaces, `xml:id`, XLink: standardized fragments, largely unused
S1	~ — vocabulary separated by schema; arrangement and data still fuse
S2	✓ — XPath and XSLT: query and transformation with specified semantics
S3	✓ — one standardized language per capability; components substitute
S4	~ — fragments and XPointer address into documents, largely unused

11.3 JSON/REST

The S-properties of the deployment style the industry calls REST. The style's genuine inheritance from HTTP survives in the scores: resources carry URIs, so S4 holds for whole JSON documents. Inside the JSON it fails: a URL is a string like any other, so a generic client cannot follow it from one document's data to the next. The style's own definition requires hypermedia links (Fielding 2000, §5.1.5); the deployments that adopted the style's name discarded the requirement. Query and transformation semantics are implementation-defined (S2 ✗). Substituting a component means renegotiating a bespoke contract per pair of parties (S3 ~). Every payload ships arrangement and data fused (S1 ~; the endpoint separates, the representation does not).

	JSON/REST
R1	✓ — trees encode any domain
R2	✗ — no defined composition; merge is per-application policy
R3	✗ — no reference type; links are strings
S1	~ — endpoints separate; representations fuse arrangement and data
S2	✗ — interpretation left to applications; query standardized 2024, transformation never
S3	~ — substitution behind bespoke contracts
S4	~ — resources have URIs; values do not link onward

The opening question (*and what changed?*) now has a measured answer: the migration changed brackets, dropped the tooling, and gained no property. Three cells changed (R3, S2, S3), and every one moved down. The numbers show lateral churn.

12. Chapter 11. The Single-Page Application

This chapter audits the single-page application (SPA). In the book's terms its architecture is the trivial factorization of Chapter 4 (Prop. 4.2), deployed at industry scale. `select` and `present` shrink to near-identities; the application is stuffed into one term that computes all of `read`. Fetching, state management, templating, and styling decisions interleave in one program, delivered as one bundle.

S1. State lives at every level of the program (component state, stores, caches, props), with no factor boundary anywhere. The paradigm's own architecture diagrams present that flow of state as a feature. ✗

S2. The term is imperative, so its meaning is defined by execution order. The failure is in principle rather than in implementation. Def. 4.3 requires each factor to denote a term in a language with closed semantics; an imperative program's meaning is the trace of its execution. No discipline within the paradigm can repair this, because the paradigm *is* the choice of trace over denotation. ✗

S3. No factor can be replaced without rewriting the term as a whole. Substitutability needs a factor boundary to swap across, and S1 showed there is none. State, selection, arrangement, and presentation are one program, so changing any concern means editing that program, not substituting a term of a language. ✗

S4. No intermediate value has a URI. The data behind a rendered view cannot be addressed, cached by intermediaries, indexed, or linked. Chapter 3 showed this on the dashboard: stripping style left only the chrome, because the curves were never in the document — a script drew them as pixels on a canvas. The state never reached the application's own document. ✗

12.1 The one-timeline consequence

Independent evolution (Prop. 4.5) states what the fusion costs. The application is one component with one timeline, so any change (data, layout, theme, query) is a change to the whole term. The term is the unit of delivery, so it is also the unit of invalidation. Four consequences follow, and each one forfeits a property that Fielding's constraints were chosen to induce:

- caching degrades to bundle-level, which is the corollary to Prop. 4.5 and now an operating cost;
- crawling requires headless browsers, machines simulating humans in order to read what machines produced;
- reuse requires reverse-engineering a private API, the S4 cost, falling on every integrator separately;

- hydration (shipping the document *and* the program that regenerates it) is the S1 cost: the architecture cannot tell its document from its program, so it ships both.

12.2 The R-properties

R1 holds: a program can keep any state in memory. R2 and R3 fail together. Component state has no merge law. State-synchronization libraries are the compensating industry. And references are pointers: machine-local by definition.

12.3 The platform's component model

One objection to the audit is that these failures belong to frameworks and that the platform is the cure, because the platform has since shipped its own component model. Web Components are the test case. A custom element gives the fused term a tag name; shadow DOM gives it a boundary the document's own selectors cannot cross.

The tag denotes nothing until its class executes, so meaning is still the trace (S2). The element's state lives in the class, threaded as before (S1). The shadow tree is an interior hidden *by design*: no URI reaches it, and now no selector either (S4). And no factor boundary appeared, so substitution still means rewriting the class (S3). The standard draws its boundary around the fused program, not through it: shadow DOM encloses the whole term, while properness needs it cut into select, arrange, and present. A component boundary is not a factor boundary. Moving the component model from framework to platform changes no score. Chapter 1 filed the framework as an implementation detail, and this is the confirming experiment.

What shadow DOM does standardize is encapsulation: state hidden behind a boundary on purpose. Chapter 13 audits the paradigm that encapsulation belongs to, and that paradigm is not the web's.

The corollary: **the SPA is the un-web** — HTTP reduced to a pipe delivering a program whose interior satisfies none of the properties that define the web. The corollary claims nothing beyond the column. Rejecting it means rejecting one of the scores, and each score names the property an objection would have to argue against. A prediction follows: the paradigm caps structurally at Web 2.0, because Web 3.0 *means* machine-consumable state (the definition Chapter 21 makes exact), and the paradigm is defined by hiding state behind `read`. Chapter 14 measures the industry's own retreat from the paradigm; Chapter 23 records the machine consumers that appeared in the meantime.

12.4 SPA/JS

	SPA/JS
R1	✓ — any state, in memory
R2	✗ — no merge; synchronization is bespoke
R3	✗ — references are machine-local pointers

	SPA/JS
S1	✗ — state threaded through one term
S2	✗ — meaning is execution order, in principle
S3	✗ — substituting a factor means rewriting the term
S4	✗ — no intermediate value has a URI

The column restates the corollary, cell by cell.

13. Chapter 12. The Applet Returns

WebAssembly (Wasm) as a paradigm treats the browser as a virtual machine and the application as one compiled binary. It is the extreme case of fusion, because everything collapses into one term, and it therefore scores zero on all of S1–S4 *by construction*. The format's virtues (any language, compiled, opaque) remove every seam the properties require. There is nothing inside the term for the web's semantics to address, and that is the design.

One step separates this column from the last. Chapter 11's fused term still emitted a DOM, a tree the platform could at least inspect. The paradigm audited here renders into a canvas or a buffer. Chapter 6 classified every framework as a strategy for the graph→tree crossing. This one declines the crossing altogether and produces no graph or tree, only pixels. That is the terminal state of fusion.

History has already run this experiment. In 1996, Java applets were compiled programs delivered through the page, executing in a VM, rendering into a rectangle the web could not see into. The same description fits today's paradigm. The applets are gone; the web's declarative documents are still here. The reason, then and now, is the principle of least power: prefer the least powerful language that suffices for the job. The W3C Technical Architecture Group made it a formal finding in 2006: *The Rule of Least Power*. The finding is its own document, not, as often assumed, part of AWWW.

The concession is Wasm as a *leaf*: a codec, a physics kernel, or a solver running inside one factor of a proper factorization. A leaf is useful and harmless, because computation inside a factor leaves every property intact. The factor's boundary is still a declarative term. The objection is to a program *replacing* the three factors, not to a program running inside one of them.

Black-box binaries served by corporations invert the property that let every reader of the early web become an author by viewing source. That inversion is the business model, not an accident.

13.1 Wasm-as-paradigm

	Wasm-as-paradigm
R1	✓ — any state, in linear memory
R2	✗ — linear memory has no merge
R3	✗ — references are addresses in a private address space
S1	✗ — by construction
S2	✗ — by construction
S3	✗ — by construction
S4	✗ — by construction

14. Chapter 13. Pre-Web Paradigms

Technology changes quickly; people's minds change slowly. [...] The next generation of programmers grows up only being shown one way of thinking about programming. [...] They grow up with dogma. And once you grow up with dogma, it's really hard to break out of it.

— Bret Victor, *The Future of Programming*, 2013

Four columns of failures have accumulated: two bracket stacks (Chapter 10), the single-page application (Chapter 11), and the applet (Chapter 12). The pattern repeats. The scores cannot say where architectures that fail this way keep coming from. The answer is that they do not come from the web. Relational databases, object orientation, object-relational mappers (ORMs), imperative languages, and MVC (model–view–controller) all predate the web. Each fails the derived requirements at one identifiable seam, and that is *why* each of them has a compensating industry at the web boundary. The industries are the evidence that the gaps are real: nobody builds a bridge across a gap that isn't there. And the claim is falsifiable: it would be refuted by a paradigm that fails no derived requirement but has a compensating industry.

14.1 Relational

The relational model is the strongest of the pre-web paradigms and the most instructive. Inside one database it scores where nothing else pre-web does. Relational algebra is denotational, which means it had S2-grade semantics decades before the web. Its separation of query from storage is genuine S1 discipline. The failure is R3, and it is total: keys are database-scoped, so a reference is only meaningful to clients holding the same connection string, and two databases share no name for anything if their owners never coordinated. R2 fails as a consequence, because composition now requires a schema authority. The compensating industry is data integration: each pair of silos is bridged by hand, and every new pair needs a new bridge.

The relational world even rediscovered the triple shape (5.3) from the inside. Entity–attribute–value (EAV) is a schema of one three-column table. It appears wherever attributes cannot be fixed in advance (clinical records are the classic case). The practitioner literature files it under anti-pattern: the schema no longer names the attributes, and every query becomes self-joins. The audit agrees with both parties. EAV is the triple shape with both `I` positions stripped of global scope: entities are row keys, attributes are strings, both meaningful only inside one database. R3 still fails, so no second party can merge or join. The shape's benefits are coordination-free merge and cross-party reference, and a single database has no other parties, so EAV carries the shape's costs and none of its benefits. That is why the anti-pattern verdict is correct inside the silo.

14.2 Object orientation

Encapsulation is the deliberate fusion of state and behavior. That is R1 half-inverted: state exists, but in order to be hidden. Objects are designed neither to merge nor to be referenced from outside their runtime; identity is a pointer. The compensating industries are serialization frameworks and data-transfer-object (DTO) layers: machinery that re-extracts the hidden state every time it must leave the runtime.

14.3 ORM

An ORM is a type error between two wrong models: object graphs mapped onto relations, machine-local identity onto database-scoped keys. Each side fails a different requirement (R1 on the object side, R3 on the relational), and an ORM fails both at once. The size of the object-relational impedance-mismatch literature measures the scale of that failure.

14.4 Imperative languages

S2 is unreachable in principle here: an imperative program's meaning is the trace of its execution. Chapter 11 made that argument about the fused term; here it applies to the language itself. The compensating industry is testing: unit, integration, and end-to-end suites. When meaning is execution, every claim about meaning must be executed before it can be checked, so semantics is recovered empirically per program, never once for the language.

14.5 MVC

MVC assembles the paradigms above. Its Model has neither R2 nor R3, its Views lack S2, and its Controllers fuse what S1 separates. Take it apart and each part has a derived generic replacement. The model gives way to the shape of a fact (5.3), the view to $[\![\texttt{t}]\!]$ and $[\![\texttt{s}]\!]$, and the controller to `read` and `write` themselves, which HTTP had already provided.

Paradigm	Fails	The compensating industry
Relational	R3 (keys are database-scoped)	integration: hand-built bridges between silos
OOP	R1 half-inverted (state present but hidden)	serialization frameworks, DTO layers
ORM	a type error between two wrong models	the impedance-mismatch literature
Imperative	S2 unreachable in principle	test suites doing the work semantics should
MVC	all of the above, assembled	all of the above, assembled

These are not outdated because they are old. HTTP is old. They are *pre-web* in the technical sense: their reference, composition, and semantics mechanisms are machine-local, and the web is definitionally the machine-spanning case. The 1960s–70s stack answers "how do I compute inside one machine". The web asks "how do independent parties share state with no coordi-

nator". Theorem 5.4 shows that the second question forces one model, and none of these paradigms matches it.

Two columns cover the five paradigms. OOP and its ORM share one column. The audit of imperative languages comes down to a single cell, S2, which is unreachable in principle. A column for MVC would repeat the others.

	Relational	OOP/ORM
R1	✓ — any domain, one schema at a time	~ — state present but hidden
R2	✗ — no shared names, so composition needs a schema authority	✗ — objects do not merge
R3	✗ — keys are database-scoped	✗ — identity is a pointer
S1	✓ — query separated from storage	✗ — encapsulation fuses state and behavior
S2	✓ — relational algebra is denotational	✗
S3	~ — within one vendor's dialect	~ — behind interfaces, within one runtime
S4	✗ — no value addressable from outside	✗

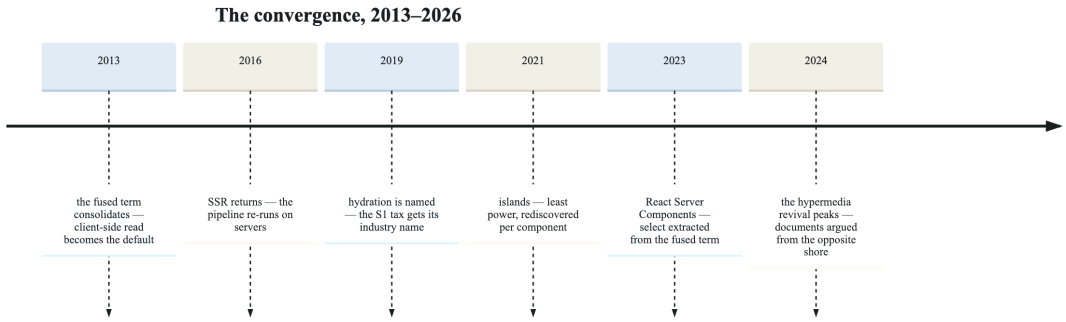
The relational column is worth reading twice: it holds the highest pre-web score in the book, and it fails on exactly the machine-spanning properties. The diagnosis follows from the scores. The relational model is a correct answer to the single-machine question, applied to the machine-spanning question instead. The scores are final, but the compensating work goes on: the bridges, the serializers, the test suites. What the industry changed after a decade of that work is the next chapter.

15. Chapter 14. The Convergence

This chapter is the evidence. The JavaScript ecosystem was not reading a derivation; it had its own problems: pages that crawlers could not read and first paints that took too long. Driven by those problems, it has spent a decade converging back toward the proper factorization, one rediscovery at a time:

- **SSR — server-side rendering.** Documents should arrive as documents. The S4 cost landed as Chapter 11 predicted: every integrator had to reverse-engineer a private API, and search crawlers could not read the page at all. The industry moved for its own reasons (SEO and first paint) and the fix is `present ∘ arrange ∘ select` running on a server, where it had been running all along.
- **Hydration.** The industry gave Chapter 11's S1 cost a name of its own. The name records a category error: it classes the document as an incomplete form of the program, dry until the program re-runs in the browser. Nothing needs hydrating in an architecture that can tell its document from its program.
- **Islands.** The industry rederived the principle of least power from the costs: most of the page needs no program, so most of the page is no longer written as one.
- **React Server Components.** RSC pulls `select` back out of the fused term, ten years after fusion. But its wire format is a bespoke, non-addressable serialization of exactly the intermediate value S4 says should have a URI. So RSC restores the `select` factor, but not the addressable resource that S4 requires.
- **GraphQL.** GraphQL provides declarative queries over a graph model, and those queries return projections. It rebuilds `select` and R1, but without global identifiers. Because R3 is missing, GraphQL stops at the boundary of the organization that defines the schema: federation works within one organization and no further. Its column is below.
- **HTMX and the hypermedia revival.** This is the same convergence, reached from the opposite direction: practitioners argued documents, links, and forms back into fashion on their original merits. Forms are Chapter 7's instrument.

The framing is convergent evolution: independent lineages, under the same pressure, arrive at the same design, and the match is explained by the pressure, not by chance. The lineages had contact with nothing but the costs, and the costs are the derivation's predictions.



Years mark mainstream adoption, not invention. The figure shows when each piece returned, not how much: S1 and S2 have recovered to partial scores, while R3 and S4 have not been recovered at all.

15.1 GraphQL

	GraphQL
R1	✓ — a graph model; encodes any domain
R2	✗ — schemas compose only by coordination
R3	✗ — no global identifiers; names stop at the schema boundary
S1	~ — `select` genuinely separated, but behind one endpoint
S2	~ — a specified but prose-defined semantics
S3	~ — substitution within one schema's contract
S4	✗ — one endpoint; results are not addressable

15.2 SPA/JS, revised

	SPA/JS (Ch 11)	SPA/JS (2026)
S1	✗	~ — `select` re-extracted
S2	✗	~ — declarative islands in an imperative shell
R3	✗	✗
S4	✗	✗ — the wire format has no URI

R1, R2, S3: unchanged from Chapter 11's column.

The convergence recovers S1 and S2, to a tilde, not a check, but neither R3 nor S4, the two properties that make an architecture *the web* rather than an app platform that happens to use browsers. The reason is that the remaining work benefits everyone except the vendor doing it: vendors recover the properties that benefit them privately and stop there.

16. Chapter 15. Property Graphs

Use URIs as names for things.

— Tim Berners-Lee, the first rule of *Linked Data*, 2006

A property graph stores nodes and edges directly, with properties on both. That makes it the closest any deployed model comes to RDF, the state model of the derived stack.

16.1 The model

Nodes and edges carry key–value properties. Edges are typed, directed, and have identities of their own. The model encodes any domain, so R1 holds. Each property is three things: the element it belongs to, a key, and a value. That is the arity Proposition 5.2 derived. Nothing types the positions as global names, and R3 is where that shows.

16.2 The names

A node's identity is local to the store, so applications put a second identifier in a property: a UUID, an asset code, a customer number. That is where R3 fails, and it fails the way Chapter 10 found for JSON. That identifier is a string, and the model does not distinguish it from any other string. Two parties must therefore agree on which property holds the identifier, on where its values come from, and on what they refer to. Those agreements are the coordination R2 forbids. Reference here is global by discipline, never by type.

A URI needs no such agreement. Its authority component delegates minting, so any party can issue a global name without asking anyone, and (5.3) makes reference a matter of type rather than discipline. The obvious repair is to invent a global scheme of your own, and Chapter 5 ruled it out: a second naming system violates R2 by itself, because two parties' private schemes collide on merge.

A merge must decide which nodes are the same node. The model decides nothing.

The same panel, described by two parties. The property graph cannot say whether the two nodes on the left are one panel. The shared name settles it on the right.

16.3 The standards

A query is a term evaluated against the store, not code inside it, so selection is separated from storage and S1 holds. The query side is standardized too, and recently. Cypher shipped with Neo4j in 2011 and was opened as openCypher in 2015. The SQL committee added property graph queries to SQL itself as SQL/PGQ (ISO/IEC 9075-16) in 2023. GQL followed in 2024 as a language of its own (Graph Query Language, ISO/IEC 39075). A query means the same thing across implementations, which is what S2 asks of `select`.

S2 asks the same of `arrange` and `present`. No transformation language was standardized for the property graph, and no presentation language. Chapter 10 scored JSON ✗ for two reasons: no transformation language, and interpretation left to the application. The property graph lacks a transformation language too, but GQL specifies what a query means, so the second reason does not apply. One reason scores a tilde where two scored a cross. S3 takes a tilde for the dialects: products implement GQL to different degrees and keep their own extensions, so one product's term does not always evaluate in another. S4 fails on the protocol. Endpoints exist. The model has two query standards and no wire protocol, so each product ships its own, and the query goes in a request body rather than a URL. The query has no address, and neither does its result, so nothing can link to it or cache it. SPARQL's protocol defines query via `GET` instead, which puts the query in the URL and makes the result a resource. Every integrator reverse-engineers a private API when resources are not addressable. No standard defines this one, so it is private by definition. That is GraphQL's cell again, and for the same reason.

16.4 The edge-property argument

The two models have argued for fifteen years, almost always about edge properties. A property graph hangs data on a relationship directly. Plain RDF cannot. Its standard device is reification, which describes the fact in triples of its own, and Chapter 9 audited it as a mismatch.

Saying when the relationship began. The property graph puts it on the edge; RDF names the fact first.

The rows do not settle that argument, because it is not about any of them. It is about arity. Chapter 9 answered that by adding a requirement, attribution, and deriving a fourth position from it. Both models can carry data about an edge. The rows measure something else: whether the names in the positions refer beyond the store.

RDF 1.2 answers that complaint. It adds the triple term, a triple used as the object of another triple, so a statement can be described without being asserted. It reached Candidate Recommendation in April 2026, and SPARQL 1.2 is still a Working Draft. The addition is narrower than it looks. A triple term is a fourth kind of term beside IRIs, literals and blank nodes, and it is not a name, so nothing can address it. The name belongs to the reifier, the subject that `rdf:reifies` it. The specification makes the feature optional too, splitting conformance into a basic level without triple terms and a full level with them. No score changes, in this column or the derived stack's. The triple term widens (5.3)'s object position to carry annotations. Chapter 9 widened the triple to a quad because attribution required it.

16.5 The compensating industry

Entity resolution is the work of deciding that a node here and a node there are the same thing. It exists because identity is a property rather than a name, which is R3 failing. Two stores can hold the same panel under different keys in differently named fields, and nothing in the model objects, so the matching is done afterwards by hand or by inference. The graph vendors sell the

remedy beside the model, as a solution category of its own. The bridge is per pair of stores, so a third store adds two more pairs and not one more name. Chapter 13 found the same industry at the relational boundary and called it integration.

16.6 Property graph

	Property graph
R1	✓ — any domain, in a graph shape
R2	✗ — merging needs agreement the model does not supply
R3	✗ — reference is by discipline, not by type
S1	✓ — query separated from storage
S2	~ — query standardized, transformation never
S3	~ — GQL conformance varies by product
S4	✗ — the query goes in a body, so the result has no URL

Chapter 13 found the relational model failing R2, R3 and S4, the three machine-spanning properties, because it answers a single-machine question. This column fails the same three. The property graph got the shape right. What fails is the names.

17. Chapter 16. The Derived Stack

Six chapters have scored eight columns between them, and every column holds a failure. One column remains, scored on the same seven rows: the stack Part III revealed as RDF, SPARQL, XSLT and CSS. An audit that stopped here would have skipped exactly the technology the book argues for. This chapter scores it against the same seven properties. It is the audit with the least new work in it, and that is the finding: every cell below carries a citation to a result already proved.

When Chapter 11 scored the SPA, each score was an argument made in prose: the cell named its property, and disputing the score meant attacking that property. The cells below assert nothing new. R2′s ✓ is the union law (5.1), proved in Part II before the stack was named; S2′s ✓ is Theorem 8.2, proved in Part III. The part's opening rule is that rejecting a score means rejecting a property. That rule reaches its hardest case here: each cell below names the proved result behind its score.

The three R-cells are the derivation itself, taken in the order Chapter 5 proved them: R2 first, then R3, then R1. R2 is the union law (5.1): merge is set union, total, order-free, idempotent, the only composition the coordination-free laws leave. R3 is (5.3): the first two positions of every fact lie in `I`, so reference is global by type rather than by discipline. R1 is Proposition 5.2 with Theorem 5.4: triples encode any domain, and any arity-minimal model satisfying all three requirements is isomorphic to this one. Chapter 5 defended the requirements one rejection at a time (reject R2 or R3 and you have built a silo), before any stack had been named, so the rows were not drawn to favor this column. The ✓s carry over because Chapter 8 proved the stack is the derived model, renamed.

The four S-cells draw on the two halves that met in Chapter 8: analysis and synthesis. From the analysis half, S1: it holds by construction of the factorization (4.3). From the synthesis half, S2 and S3: each factor denotes a term in a language with closed, cited semantics (SPARQL's algebra, XSLT-over-canon, CSS) and terms with closed semantics substitute per factor (8.2). S4 draws on both halves. The factorization requires every stage value to be addressable (4.3), and the stack delivers that by construction: the graph, the query result, and the document each have a URI that dereferences (8.2). One qualification carries over with the theorem. Theorem 8.2 covers only transforms that are generic, meaning invariant under URI renaming (B.8): an `arrange` term must not treat particular URIs specially. The S2 and S3 ✓s therefore hold for that class of transforms, not for arbitrary code.

A self-scored column is suspect only if it leaves defects unstated, and Chapter 9 already stated them: four mismatches. The two model mismatches affect the rows most directly. Blank nodes weaken R2 slightly: idempotence holds up to logical equivalence rather than syntactic identity,

and that cost is computed exactly (Prop. 9.1). Over ground facts the row holds exactly, and the composition laws still hold. The fourth position adds a requirement rather than removing a score: the web requires R4 (attribution), and the column meets R4 by moving to quads, where merge is still set union (Prop. 9.2). R4 is not a row below because no column in this part was scored on it; it is Chapter 9's addition, and the quad model satisfies it.

The two platform mismatches leave every row's score unchanged, because no row measures maintenance. The abandoned seam and the write-side last mile are deployment failures, and deployment failure is what this part has measured in every other column's compensating industry. The stack's own deployment failures are itemized in Chapter 9 as mismatches three and four. The mismatches do not soften the column. They are why it can be trusted: Chapter 9 stated all four openly before this audit began.

17.1 Derived stack

	Derived stack
R1	✓ — encodes any domain; the model is forced (5.2, 5.4)
R2	✓ — merge is set union: total, order-free, idempotent (5.1)
R3	✓ — reference is global by type: the first two positions lie in `I` (5.3)
S1	✓ — factor boundaries by construction (4.3)
S2	✓ — each factor a term with closed, cited semantics (8.2)
S3	✓ — terms substitute per factor (8.2)
S4	✓ — every stage value dereferences (4.3, 8.2)

18. Chapter 17. The Properness Table

Every audit in this part ended with a column. This chapter assembles those columns into one table. No cell below is new; placing them side by side is. Each column header cites the chapter that scores it. The derived column cites two: Chapter 8, which proved it, and Chapter 16, which audited it. Its cells add a parenthetical citing the scoring result. The rows are the seven derived properties: R1–R3 on state, S1–S4 on architecture. ✓ is satisfied, ~ partial, ✗ failed. In the online edition every cell of the derived column links to its proposition.

Property	Relational (13)	Property graph (15)	OOP/ORM (13)	XML stack (10)	JSON/REST (10)	SPA/JS (11)	Wasm (12)	GraphQL (14)	Derived stack (8, 16)
R1	✓	✓	~	✓	✓	✓	✓	✓	✓ (5.2, 5.4)
R2	✗	✗	✗	✗	✗	✗	✗	✗	✓ (5.1)
R3	✗	✗	✗	~	✗	✗	✗	✗	✓ (5.3)
S1	✓	✓	✗	~	~	✗	✗	~	✓ (4.3)
S2	✓	~	✗	✓	✗	✗	✗	~	✓ (8.2)
S3	~	~	~	✓	~	✗	✗	~	✓ (8.2)
S4	✗	✗	✗	~	~	✗	✗	✗	✓ (4.3, 8.2)

Notes, one per line where a cell needs it.

- XML's R3 and S4 are ~ for namespaces, `xml:id`, and fragment addressing, standardized slivers of the properties, largely unused (Ch 10).
- GraphQL's S2 is ~ for a specified but prose-defined semantics. Its S1 is ~: `select` is genuinely separated, but every query is served from a single URL. S4 fails at that same URL: a query result has no URI of its own, so it cannot be linked or cached.
- The property graph's S2 is ~ for GQL (2024): query standardized, transformation never (Ch 15).
- SPA/JS is scored as Chapter 11 audited it, at its 2013 consolidated form. Chapter 14 revises S1 and S2 to ~ as of 2026 and leaves the rest unchanged. That revision is Chapter 14's finding, so the column keeps its 2013 values here. The 2026 deltas live in Chapter 14's table.

Read the table by row rather than by column and the finding appears. R2 fails in every column but one: composition without coordination is the property nobody else has, and it is what the "world wide" in the name promises. Count the tildes and R3 and S4 join it: all three machine-

spanning properties fail or fall partial everywhere but the derived column. R3 and S4 are the two that make an architecture the web rather than an app platform that happens to use browsers. Chapter 13 found the relational model, the best pre-web scorer, failing exactly the three machine-spanning properties (R2, R3, S4); Chapter 14 found the JavaScript convergence has recovered neither R3 nor S4. And each S-row failure appeared in this part twice: once as a score, once as a compensating industry.

One column has no failures, and Part II proved its model was forced. Those two facts are the book's argument in a single table. Part V builds with that column.

V

Part V — The Synthesis

This part builds from the one column of the audit that had no failures. It constructs the application space, shows that no new standard is needed, states the economics of generic software, and collects the result.

Chapter 18 Building Up Chapter 19 No New Standard Chapter 20 Generic Software Chapter 21 The Result

19. Chapter 18. Building Up

This chapter runs the synthesis direction constructively, presenting the synthesis theorem as a build log. Start with the derived atoms and compose a working application space, defining each layer by what Part II forced and each concrete technology by the factor it implements.

19.1 The dataspace

What the synthesis yields needs a name, and the name should do for state what "website" did for documents. Call it a **dataspace**: one party's stake in the data web, the unit of publication, ownership, and federation. A website serves documents under an origin; a dataspace serves *state* under an origin. Documents are included, since they are projections of that state. Machines can consume it too, since the names in the state can be dereferenced. The definition needs one primitive Part II never used, and the web already provides it. Chapter 1's pattern holds one last time:

```
O        the set of origins                                (RFC 6454)
I|o      the URIs under origin o
```

An origin is not a new kind of name. RFC 6454 computes it *from* the URI (scheme, host, port), so `O` is a quotient of `I`: many URIs, one origin. The namespace falls into regions, one per party, and "one party's stake" acquires a type. The definition has four components and no more: one origin and three names in that origin's region. The three can be names because on the web every published thing is a name:

```
Dataspace = (o, ont, e, x)      o ∈ O;   ont, e, x ∈ I|o   (18.1)
```

name	component	gloss
`o`	the origin	read-write linked data at every document under it
`ont`	the ontology	what the domain *is*, stated as one namespace
`e`	the SPARQL endpoint	the same state, projected by query
`x`	the stylesheet	declarative rendering, extended by override

Internal storage (file, memory, triplestore) has no row. It is invisible to consumers, as S1 demands.

Behind the four names stands one state `S`, in the shape of Prop. 9.2: quads, grouped by their fourth position into a family of named graphs. `S(u)` is the graph named `u`, and every graph name is a document URI under `o`. The gloss column is then four laws: each restates an earlier result as a rule of deployment.

19.1.1 Documents

Dereference is graph lookup — `select(u, S) = S(u)`, the fourth position as the address (Prop. 9.2). There `read` is defined (S4) and `write` accepts a delta (Prop. 7.1). And the obligation that makes the data *linked*: every name under `o` in a fact position of `S` has `read(name, S)` defined — mint a name only if you serve its description. AWWW §3.5 asked for this as a SHOULD; (18.1) holds it as a condition of being a dataspace at all. So the state is not only composable but recursively discoverable: each reference in a fact is an address, and dereferencing it returns more state, whose references point onward in turn. Practitioners know this as "follow your nose". Documents may also nest. Parent and child are ordinary facts, so a document's children are one more selection. Addressing stays flat (one graph per document) and the hierarchy is a convention over it, not a new kind of resource.

19.1.2 One state

The endpoint `e` answers $[\![q]\!]$ posed to `S` itself, the same `S` the documents project. "Projecting the same state" is an equation: a document serves facts and the endpoint returns facts, both come from one `S`, so they must agree. If a second store drifts from `S`, the agreement breaks observably.

19.1.3 Domain as data

`S(ont)` is schema in the shape of (5.3): the domain's classes and properties are stated as facts, so the ontology is ordinary state and merges by union. The build log below shows what reads it.

19.1.4 Total rendering

`x` dereferences to the arrange term, generic in B.8's sense. The build log below looks at it more closely.

One entity makes the four concrete. `GET …/panel-14` returns the graph of facts about that panel. `PATCH …/panel-14` sends a delta, (D^-, D^+). And the endpoint answers any query that ranges over it. One state is reachable three ways, and each is an HTTP request you can make by hand.

The write side has four methods, and each is Prop. 7.1's delta with part of it fixed; `PATCH` is the general case, and `POST`, `PUT`, and `DELETE` are special cases:

method	fixes	result	
`POST`	$D^- = \varnothing$	$S' = S(u) \cup D^+$	append (a merge)
`PUT`	$D^- = S(u)$	$S' = D^+$	replace, creating if absent
`DELETE`	$D^- = S(u), D^+ = \varnothing$	$S' = \varnothing$	remove
`PATCH`	any D^-, D^+	$S' = (S(u) \setminus D^-) \cup D^+$	general

The Graph Store Protocol leaves `PATCH` informative; realized, it is a graph-scoped SPARQL Update. HTML forms speak only `POST`, so a form's delta arrives through the RDF/POST bridge (Chapter 9). Zoom out from one graph to the whole dataset and the same four return

on quads: `GET` a dataset, `POST` appends quads, `PUT` replaces it, `DELETE` removes it, the extended form some triplestores implement.

PATCH — any D^-, D^+. The general `write`; the other three are it at a fixed delta.

`S` — the graph …/panel-14

```
(⟨…#panel-14⟩, type, ⟨…#Panel⟩)
(⟨…#panel-14⟩, title, "Current Power")
(⟨…#panel-14⟩, value, "15.5 kW")
(⟨…#panel-14⟩, partOf, ⟨…#farm⟩)
```

D^- — removed

```
panel-14 value "15.5 kW"
```

D^+ — added

```
panel-14 value "16.1 kW"
```

Interactive exhibit (online edition): the write methods on …/panel-14. Pick GET, POST, PUT, DELETE, or PATCH. The delta (D^-, D^+) snaps to that method's row, and the graph updates by $S'(u) = (S(u) \setminus D^-) \cup D^+$. The same panel graph the chapter reads, now writable by hand.

(18.1) omits `S` itself: neither the state nor the store holding it is a component. A consumer sees only the four projections, so the store is invisible by definition rather than by an implementer's discipline. That lifts S1 from one factor to the whole system. Two deployments with the same four projections are the same dataspace.

And the union law returns. Federation adds one thing to prove, and B.9 proves it from the types alone. Distinct origins are disjoint regions of `I`, so two dataspaces' graph names never collide, and the union of their states is again well-formed. Every document is still under exactly one origin, and attribution survives the merge because the fourth position carries it. Federation is the union law: merge, and be done.

19.2 The cost of alignment

Merge, and be done. Chapter 5's scope note deferred an objection, and here it returns at full strength: *union is cheap; alignment is not.* Two dataspaces describe the same turbine. Each minted its own name, because minting is free. The union holds two disconnected descriptions and joins nothing. Two ontologies cover the same domain and share no term. The merge laws guaranteed mechanics, never convergence: nothing makes independent parties end up sharing names and terms. *Alignment* is the missing step: stating that the two names denote one turbine and that the two vocabularies' terms correspond. So, says the objection, the model has merely moved the integration cost it claimed to remove. Granted: it moved from the merge to the alignment, and that position makes the difference. The cost is universal, because no model

makes strangers agree on names. So the question is never whether alignment costs, but what you hold before aligning, and what aligning yields.

Before aligning: the unaligned union is well-formed state. Both descriptions are present, queryable, and published, and both are rendered by the vocabulary-blind base term the build log below introduces (B.8). The worst case here is *not yet joined*; in every other column of the audit the worst case is *cannot merge* (two JSON documents do not compose at all, and two schemas do no better).

After aligning: an alignment is one more fact, an equivalence, a subclass, a subproperty, in the shape of (5.3). The fourth position attributes it to its asserter (Prop. 9.2), it can be retracted as a delta (Prop. 7.1), and it composes by union like everything else. A mapping published this way is stated once and serves the whole web. The integration industry holds the same knowledge as a join buried in pipeline code, once per pair of systems. It has no asserter, cannot be shared, and costs N × M forever.

Even the failure mode improves. The known hazard of alignment is the careless identity link: assert that two names denote the same entity when they do not, and the error spreads through every join that uses the link; the literature rightly distrusts it. In this model a wrong link is at least a published fact: anyone can see it, its asserter is recorded, and a delta retracts it. The same mistake in an integration pipeline is a wrong join key buried in code: no one outside can see it, and nothing can retract it.

And convergence has a deployed existence proof at full web scale. Vocabularies converge the way the document web converged: by adoption, not negotiation. Publish, dereference, reuse, the same unilateral move as linking to a page whose owner was never asked. Schema.org spread across tens of millions of sites in exactly this shape, because consumers with reach (the search engines) made the alignment worth making. Vocabularies get aligned when consumers need them aligned; when the semantic web shipped in the nineties, its consumers did not exist yet (Chapter 8). The substrate was built for machine consumption and standardized twenty years before machines consumed it, so every cost of convergence went unmet. The idea was not refuted; nobody yet needed convergence enough to meet those costs.

19.3 The build log

The build log takes the least familiar factor first. The ontology is the component the derivation predicts and the industry outsources to code: the domain, stated as facts. A dataspace's ontology *imports* the vocabularies it builds on (union applied to schema) and everything downstream reads it as data. Forms are constructed from it (Chapter 7's construction half: read the patterns, render inputs); selections range over it; layouts match on it. This is what makes the generic engine generic: the domain travels in the state, so nothing domain-shaped remains to be hardcoded.

The build log, factor by factor:

factor	implemented by	the derived result, deployed
state	a triplestore behind the Graph Store Protocol — the SPARQL suite's HTTP companion, whose *direct graph identification* makes the request URI the graph name — one named graph per document	the fourth position (Prop. 9.2) as an address — attribution and location coincide
domain	a namespace ontology per dataspace, importing the vocabularies it builds on	imports resolve by union — vocabulary is data and composes like it
select	a SPARQL endpoint per dataspace	S4: query results and graphs are resources with URIs of their own
arrange	XSLT over the canonical serialization — a base stylesheet naming no vocabulary, per-vocabulary overrides layered by the language's import mechanism	Chapter 6's seam filled; S3's substitution, performed in daily practice
present	CSS	in continuous service since 1996
write	HTML forms encoding graphs (Chapter 9's RDF/POST bridge, deployed in Chapter 19), written through the Graph Store Protocol's unsafe methods	Chapter 1's unsafe methods applied per graph — POST appends, PUT replaces, DELETE removes; the delta itself a PATCH, a graph-scoped SPARQL Update carrying its two sets

The build log as a picture: the pipeline (4.1) realized in deployed technologies, closed as in Chapter 7. Along the read path the endpoint `e` runs `select` (SPARQL), the stylesheet `x` runs `arrange` (XSLT, ⟦t⟧ ∘ `canon`), CSS runs `present`, (18.1)'s components bound to deployed standards. The return arrow is the write side: a form (Chapter 9's bridge) yields a delta (`D⁻, D⁺`) (Prop. 7.1), carried as a PATCH, a graph-scoped SPARQL Update. Under S4 every rounded node is a web resource with a URI of its own.

19.3.1 Arrangement

Arrangement carries the most machinery in the build log. The arrange term may treat a name specially only if the dataspace's ontology, with its imports, declares it — a custom UI gets its special cases by declaring the vocabulary it renders. That is B.8's relative genericity, running in deployment. Unmatched state falls back to the base rendering rather than to nothing: every graph renders; declared vocabulary renders better. The stylesheets share their templates across the wire: one library, imported by a server-side stylesheet that emits documents and a browser-

side one that binds events. Saxon runs the first and SaxonJS with IXSL runs the second, so two XSLT 3.0 processors share one set of terms. That is Chapter 7's mobility of evaluation in production: the same terms evaluate on the server and in the browser. The convergence shares rendering code too, by running the same framework on both sides (Chapter 14's hydration); here the sides share templates without sharing an engine, because the language's semantics is closed. And independent evolution shows up as operations rather than theory: data, selection, layout, and style invalidate independently, one factor at a time and one cache entry at a time. The four timelines run as infrastructure.

20. Chapter 19. No New Standard

Chapter 18's build log ran on deployed standards end to end, and this chapter shows that nothing more is needed. Three questions remain: how names relate to addresses, how the seams that lack Recommendations get filled, and what shape federation forces on a reference implementation. Each resolves by composing pieces that already ship. No new standard is proposed. The chapter closes with the existence proof: an implementation that runs Chapter 18's assembly.

20.1 Names and addresses

The first `GET` raises the first question. It is the web's oldest identity crisis, filed at the TAG (the W3C Technical Architecture Group) as httpRange-14: what does dereferencing the name of a *thing* return, when the thing is a turbine rather than a page? A decade of W3C argument produced a `303`-or-fragment resolution, a note (*Cool URIs for the Semantic Web*), a reopening, and a deployed practice that largely ignores all of it.

The model here has a shorter account. Names and addresses are different roles, typed apart since Chapters 4 and 5: a URI in a fact position *names* (R3); a URI addressing a projection *locates* (S4). So the question is settled by computation, not argument. Dereferencing a name returns `read(name, S)`, a description of the named entity. Its address may coincide with the name, differ by a fragment, or differ by a redirect; the choice among the three is a wire-level detail, and nothing in the architecture depends on it. Only the collision is real. Use one URI as both name and address, and the entity is conflated with the document that describes it: "the turbine weighs 200 tons" and "the description was modified on Tuesday" now share a subject. That conflation is a data-discipline cost (measurable, like Chapter 9's mismatches), and keeping name and address apart, by fragment or by redirect, avoids it.

The exhibits already resolved the question both ways. The Guardian's articles collapse the two harmlessly (an article *is* its own description) while the wind farm's panels sit one hash away as fragments (`#panel-14`). That fragment is the convention the reference implementation adopts: one `GET` serves entity and description alike. So the crisis reduces to a typing discipline the model already draws, plus an encoding choice the deployment already made.

20.2 Composition, not creation

Three seams lack Recommendations: identity, access control, and the form-native write. The first two have candidates with running code, and both fill their seam with the model itself. WebID has been incubated at the W3C since 2005 and never advanced to Recommendation. It makes an identity a URI whose dereference is a profile: an agent is an entity, its identity a

graph, authentication a proof that the keyholder and the profile agree. WebAccessControl is an ontology grown on the W3C wiki, since adopted by Solid, Berners-Lee's re-decentralization project. It states permissions as facts (who, which mode, over what) so an ACL is data in the same state model it guards. Identity and authorization collapse into the substrate they protect (previewing Chapter 20's thesis) and the reference implementation below runs both. For the third seam, Chapter 9's bridge (RDF/POST) slots a plain HTML form into the write side. RDF/POST is specified, not standardized. And as Chapter 9 showed, it is an encoding rather than an invention: it adds neither a model nor a protocol.

This part has contained no proposal for a new standard, and that absence is the finding. Part III showed the read side complete by 2014. The write side's last mile is an encoding of what already ships. The remaining seams have candidates that compose deployed pieces. Nothing here waits on a working group. The community's long reflex (meeting every gap with a new specification) aims at the wrong layer. After the reveal, the remaining work was never specification. It was combination: an implementation that assembles the standards in the derived shape.

The reflex has a Recommendation-grade instance: the Linked Data Platform (LDP, 2015), which claimed this book's exact slot (a read-write Linked Data architecture) at the interaction layer. The Graph Store Protocol (HTTP's methods addressed to whole graphs) was already standardized. LDP's one addition to it is the *container*, a server-side collection with protocol-managed membership. But a container is a canned selection, a query frozen into the interface. It came after SPARQL had already made every collection open-ended: any members, by any pattern, composed at request time. Subtract the containers and nothing remains that the Graph Store Protocol does not already do: LDP added interface where query semantics sufficed. The gap was in the implementations, which never combined what the protocols already offered.

The chapter's exhibit mirrors Chapter 3's, deliberately: the two stripped sites are rebuilt as dataspaces. The strip-2 fact lists are loaded as state, with a small ontology per domain: articles and sections for one, panels and readings for the other. Each dataspace gets one `select` term per window, an `arrange` term per layout, and a stylesheet per look. Front page and dashboard become two declarative packages over the same generic machine; both domains live entirely in data. Chapter 3 computed the factorization by hand; this chapter runs it forward, on the same material. Analysis and synthesis meet on worked examples. *(The full-scale reconstruction is being built in public, as Chapter 3's exhibit once was; the miniature below runs today.)*

two datasets — the domain travels in the `State`

`State` — the domain lives here

```
panel-14 type Panel
panel-14 title "Current Power"
panel-14 value "15.5 kW"
panel-14 partOf overview
panel-7 type Panel
```

```
panel-7 title "Wind Speed"
panel-7 value "8.2 m/s"
panel-7 partOf overview
panel-3 type Panel
panel-3 title "Grid Frequency"
panel-3 value "49.98 Hz"
panel-3 partOf archive
farm name "Anholt Offshore"
```

`select` — the window

```
?p partOf overview
?p title ?t
?p value ?v
```

`canon(Data)` — the canonical serialization

```
@prefix d: <https://wind.example/farm#> .
@prefix w: <https://wind.example/vocab#> .

d:panel-14
    w:partOf d:overview ;
    w:title "Current Power" ;
    w:value "15.5 kW" .

d:panel-7
    w:partOf d:overview ;
    w:title "Wind Speed" ;
    w:value "8.2 m/s" .
```

`arrange` — the term `t`: $t_1 \cdot$ cards`present` — the stylesheet: console

`read(r, S)` — the document

20.3 Current Power

partOf
overview

value
15.5 kW

20.4 Wind Speed

partOf
overview

value
8.2 m/s

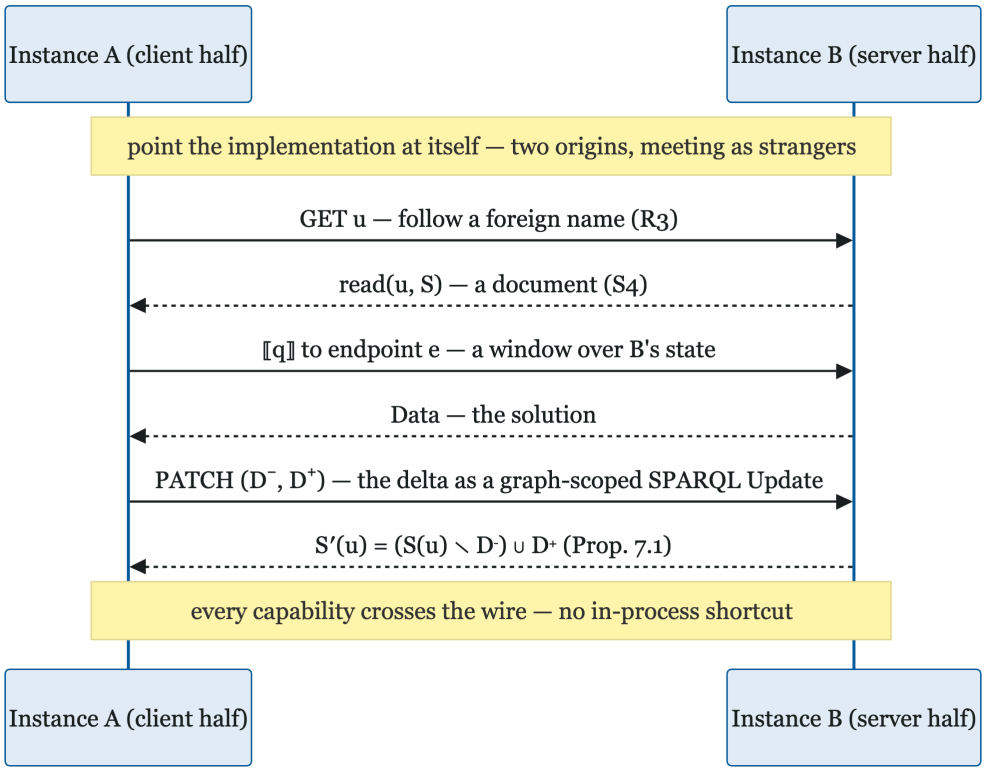
swap any factor — data, selection, term, stylesheet — and the others hold still.

Interactive exhibit (online edition): the reconstruction in miniature. The two datasets from Chapter 3 under one generic engine — swap the data, the selection, the arrangement, or the stylesheet, and the factors you did not touch hold still. The full-scale reconstruction runs the deployed standards (SPARQL, XSLT, CSS); this miniature runs the book's own algebra.

20.5 The federation test

Federation needs a client, and the derivation says so before any implementation does. R3 put foreign names inside local facts. In deployment, following one means calling another party's `read`. A window over another party's state is ⟦q⟧ posed to another party's endpoint. Consuming dataspaces is therefore the other half of the architecture, not a feature an application adds. A dataspace that only serves is a leaf, because it never follows anyone else's names. So a reference implementation has a forced shape, both halves at once: a server publishing (18.1)'s four components (origin, ontology, endpoint, stylesheet) and a client consuming the same four from any other dataspace.

Both halves in one implementation enable a test no bespoke system can run: point the implementation at itself. Two instances run at two origins, and one of them browses, queries, and writes against the other. Every capability crosses the wire or fails visibly, because no in-process shortcut exists for a demo to lean on. Self-federation is the architecture's own strategy put under test: two instances meet as strangers, and the first federation is its own. And the test is not circular. S2 leaves the two instances nothing private to share. Everything that crosses the wire is a term of a closed language (data, query, delta, arrangement), so they meet only on the specifications' surface, with no side channel to agree over. The standards process proves interoperability with two independent implementations. A reference implementation proves it with two instances of itself. That evidence is weaker, but it is available years earlier, and it holds only as long as the wire carries nothing but spec-terms. A second implementation joins by implementing the same denotations, over the same interface the self-federation test already exercised.



The federation test, drawn. Each instance runs both halves. Here Instance A's client consumes Instance B's server across three exchanges. It dereferences a foreign name (R3) for a document (S4). It poses ⟦q⟧ to the endpoint e for a window. It submits a delta (Prop. 7.1) as a PATCH, a graph-scoped SPARQL Update. Because S2 leaves nothing private, the meeting surface is the specifications' surface alone.

The document web bootstrapped exactly this way. The pattern, stated generally: a server and client pair whose self-interoperability is the first running instance of a protocol anyone may join.

The first instance, dated. This is not an analogy; it happened. In 1990 the first web server (`info.cern.ch`) and the first browser ran on two NeXT machines at CERN and interoperated with each other before there was a third program in the world to interoperate with. That browser (WorldWideWeb, soon renamed Nexus so the web could keep the name) was also an editor: reading and writing went through one program. The write side was there on day one, then lost for a generation as the read-only browser became the thing everyone shipped. The federation test above makes that first day a permanent requirement.

20.6 The existence proof

This chapter is where the book's existence proof enters as evidence. The architecture has a reference implementation: **LinkedDataHub**, open source, in production for years. It federates the way the section above requires: instance to instance, its client half consuming what its server half serves. It also delivers the chapter's other promises: WebID and WebAccessControl are running, RDF/POST is accepted on the write side, and an entity's name differs from its

description's address by only a fragment. And the online edition of this book is being built on it, keeping the promise the preface made. The point of an installable existence proof is that no one has to take the book's word for it.

21. Chapter 20. Generic Software

This chapter converts the derivation into economics. A one-line corollary collapses every domain application into one generic engine specialized by data; the browser and the spreadsheet are the existence proofs. Two consequences follow: domain functionality ships as data, and computation stays outside the engine, submitting its results through `write` like any other caller. The chapter closes on the incentives that keep bespoke code in place despite them.

21.1 Specialized by data

Start from a corollary the apparatus yields at once:

Prop. 20.1. Two proper applications over (5.3) differ only in their terms and their state.

Proof — S2 plus Theorem 5.4 leave nothing else to vary. By S2, each factor is the denotation of a term; by Theorem 5.4, the state model is shared. What remains to vary is `(q, t, s)` and the facts. ■

It follows that the difference between a CMS, a CRM, and an ERP is data. Each is a UI layer around CRUD over a domain model. The domain model is facts (5.3), and the UI is ⟦`t`⟧ and ⟦`s`⟧. CRUD is `read` and `write`, Definition 1.1 and Prop. 7.1, which HTTP already implements. One application can serve every domain, specialized by data rather than by code.

21.2 Two proofs and a failure

The web has already run this experiment once, and the result is so familiar it goes unnoticed as an example: the browser. One client for every website. Nobody writes a per-site browser, and nobody marvels at that, which measures how completely the uniform interface won at the document layer. Chapter 4 typed the four clauses that won it, and generic software is their result, built layer by layer: generic caches, generic crawlers, one generic renderer. That generic software reached only as far as the interface was uniform. Behind every `GET` the verbs are shared and the state is bespoke, so the client that is generic in transfer stays bespoke in understanding. That means one adapter per API, and Chapter 23 totals the arithmetic. This chapter asks why the browser, generic over documents, never got a counterpart generic over state, and the answer is that nothing was missing except the state model Chapter 5 derived.

The claim has a second existence proof, older than the web. The spreadsheet is the most successful generic application in history: one engine serves every domain, specialized by nothing but its data. No vendor ships an accounting spreadsheet and a separate logistics spreadsheet; users pour the domain in as rows and formulas. The spreadsheet's own limits explain why it could prove no more: cell references are sheet-local (R3), two workbooks have no merge (R2), and the world's operational data lives in a million silos named `final_v2.xlsx`. The derived

stack is the same generic engine, but with names that cross files and states that compose. Each proof carries half the claim: the browser is generic with the web's properties, at the document level; the spreadsheet is generic over domains, with none of the web's properties. This chapter describes an application that holds both halves at once, and it was sitting in the standards all along.

The idea has also failed before, and the failure shows exactly what to avoid. Model-driven architecture (MDA) promised applications generated from models, and broke on its own compiler. The model was translated into code, the code drifted from the model, and the model ended up as documentation only, because that first generation step severed S2. The generic engine makes no such translation. The ontology is never compiled into the application; it *is* the application's data, interpreted at runtime like everything else, so nothing drifts because nothing is copied.

21.3 A codebase is a liability

The economics follow. A codebase is a liability, not an asset: behavior held equal, the number of bugs is roughly proportional to the number of lines. So a system is better when it achieves equal behavior with less code, and the generic system achieves it with *no domain code at all*. Domain functionality becomes a declarative package: an ontology and a stylesheet pair, imported into a running application. Installation is not a deployment but a merge: the package is data, so adding it is a union, and removing it is a delta — it uninstalls the way it installed. And the pair carries its own correctness check. By B.8′s relativization result, the stylesheet may treat specially only the names the ontology and its imports declare: what it touches stays inside the declared vocabulary, checkable from the term alone. So a package either declares the vocabulary it renders, or the check finds the undeclared names it renders. Chapter 12 audited the binary-delivery web; this is its constructive alternative: behavior defined by data, shipped as data, revocable as data.

The reference implementation ships exactly this: applications as importable datasets, administered by an application defined in the same terms it administers. Chapter 19′s exhibit rebuilds a newspaper and a dashboard on one machine. This chapter's claim is that the rebuild generalizes: the two reconstructions are datasets for the same generic engine, and the book's online edition is a third.

In the world. Enterprise architecture reached this chapter's conclusion from the cost side, without deriving it. Dave McComb's *Software Wasteland* (2018) is a book-length audit of the application-centric mindset, every enterprise rebuilding the same CRUD over its own bespoke model. Its sequel *The Data-Centric Revolution* (2019) prescribes the data-centric cure this chapter derives: make the data the fixed point and let one generic substrate be specialized by an evolving model, not by code. Those books argue it from decades of enterprise waste; Proposition 20.1 states the same result as a corollary.

In the world, dated 2017. The liability has a mainstream witness. James Somers's *Atlantic* essay "The Coming Software Apocalypse" surveys the damage done by software that has grown too large to understand. A statewide 911 outage was traced to one counter's threshold. Eighteen

months of expert review barely untangled the throttle code of a runaway Toyota. A car now carries a hundred million lines. Unmoored from anything physical, software "tends to grow without bound", into systems, as Nancy Leveson writes, "beyond our ability to intellectually manage." One interviewee states this chapter's remedy: "Nobody would build a car by hand." The essay also records the wall those remedies hit: engineers recoil from the word "formal," and an Amazon engineer won colleagues over only by retitling formal specification "debugging designs." The tools were sound and the resistance was to their image — the same failure this book calls abandonment, not refutation.

21.4 Computation on the write side

One objection lands here with real force, and it deserves the treatment latency got in Chapter 7: *real domains compute.* A payroll run turns timesheets into pay; an allocation turns orders into reservations; an invoice's total is computed, not typed in. If the engine houses no domain code, who computes? Definition 1.1 answered before the question arose: it types what the application *is* (`read` and `write`) and says nothing about who calls it. Chapter 7′s caller was a human holding a form. A computation is another caller: an agent that reads, computes, and submits its conclusion through the same `write`, in the same normal form, reviewable and invertible like every delta. Chapter 23 shows what that reviewability does for governance.

Much of what domains call computation is derivation: producing facts that already follow from the facts held — the invoice's total from its lines. The derivation step needs no new language either. Prop. 7.3 turned a pattern plus bindings into a delta, with the bindings supplied by a form; draw the bindings from the state instead and you have a rule: match what holds, assert what follows. SPARQL Update ships exactly this construction, an `INSERT` whose delta is computed by its own `WHERE` clause — the same algebra as the form, with the state supplying the bindings a human would have typed.

What lacks a recommendation is *when* such a term runs — schedule, trigger, threshold. That is the orchestration seam, open like identity and access in Chapter 19, and like them awaiting convention rather than invention.

So the domain's logic divides cleanly. Validation is a predicate on deltas (Chapter 7 drew that line). Derivation is an update term over the ontology. That places inference precisely: an entailed fact enters by that write, not by a closure computed at read time, so the state remains the asserted set. And whatever imperative computation remains (the solver, the optimizer, Chapter 12′s leaf) runs behind a caller, submitting deltas like everyone else: outside the engine, never inside it.

21.5 The incentives

Why, then, does every domain still get its own codebase? Chapter 13 supplied the mental models; the incentives supply the motive. Generic software commoditizes its vendor: a domain application's defense is precisely its bespoke code; the industry charges rent on that code and

will not embrace generic software, which dissolves the asset. So the push has to come from consumers, and Chapter 23 names them: human users never noticed the cost of bespoke code, but an agent needs a bespoke adapter for every application it touches.

22. Chapter 21. The Result

The audit table, completed: Chapter 17 fills one page, every cell carries a chapter's score, and one column records no failures. That table contains the whole of the book's argument. This chapter states what follows from it.

Web 3.0, defined: on this web, `read` is transparent all the way down (S1–S4 at every layer, R1–R3 at the substrate), for humans *and* machines alike. An agent is only another reader, as Chapter 23 will show. Every earlier use of the term outside this book named no particular property, and this one names the properties the audit table scores.

In Web 1.0, `read` was transparent over documents, and those documents were declarative, addressable, and indexable, the properties that beat every contemporary in Chapter 1′s history. In Web 2.0, `write` was added, and the fused term came with it: selection, arrangement, and presentation collapsed into one component, the application stayed on the web only at its rendered surface, and `read` no longer reached the state behind it. Web 3.0, on this definition, adds no third invention. It extends the transparency Web 1.0 gave documents to the state that lies behind the fused term. Chapter 10′s lateral churn was two decades spent inside Web 2.0. The vertical direction, building new layers on top, was open the whole time, and Part II proved that under the three requirements the next layer had exactly one shape: the state model (5.3). The table now shows what the preface could only claim: the JSON APIs, the JavaScript frameworks, and the compile-to-browser toolchains each have a column, and each one reads as a partial rediscovery of the derived architecture or a detour from it.

One cost remains: the advanced web forfeits the lowest common denominator. Part II derived what advancing requires, and Part IV computed what refusing it costs; Chapter 17 tabulates both. The choice is the reader's.

Beneath the particular scores is the claim about method the preface promised to return to. The web has been treated as software engineering (frameworks, taste, iteration) and Chapter 10′s lateral churn is what that treatment costs: motion with no fixed point to measure it against. An object with forced structure should be treated as a science instead: derived, proved, and pinned to claims that an observation could refute. The stack is the finding, and the genre of science is the method that found it.

The audit applies to this book as strictly as to the paradigms it scores, so the register below lists the book's own open claims, each paired with the observation that would refute it:

claim	where	falsified by
the SPA paradigm caps at Web 2.0	Ch 11	a fused-architecture deployment whose state is machine-consum-

claim	where	falsified by
		able at web scale without a compensating adapter layer
the convergence recovers neither R3 nor S4 without new incentives	Ch 14	a mainstream framework shipping addressable intermediates and global references as defaults
the agent economy converges on generic systems with domains as data	Ch 23	agent infrastructure stabilizing permanently on per-application protocol servers, adapter counts growing linearly
attribution pressure keeps selecting the fourth position	Prop. 9.2	a successor standard that discards named graphs
a requirement-failure and a compensating industry always coincide	Ch 13	a paradigm that fails a derived requirement with no compensating market at its web boundary, or such a market around a paradigm that fails none

Retrodictions (claims history had already graded, like quads and the JS convergence) are marked as such where they occur. This table lists only what is still open. Registered July 2026.

The canonical edition of this book (under construction) is a dataspace on Chapter 19's machine: its propositions are resources, its dependencies are typed links, and its figures are live queries. Reading it there is itself a demonstration of the architecture it argues for.

The book rests on one claim. Strip any page and the same skeleton appears. Three requirements determine what the state behind the pages must be, and the web already meets all three. That model was standardized by 2014. The next web needs no inventing; it needs only to be put to use.

VI Part VI — The Future

This part carries the synthesis into the future: the adoption already underway, the era AI agents bring on, and the web the derived stack makes possible.

Chapter 22 Knowledge Graphs Chapter 23 The Agent Era Chapter 24 The Next Web

23. Chapter 22. Knowledge Graphs

The future is already here — it's just not evenly distributed.
— William Gibson, on NPR's *Talk of the Nation*, 1999

23.1 The name

The state Part II derived is in production at scale, under an industry name. A knowledge graph is instance data and the ontologies that describe it, stored and queried as one graph. The name went mainstream in May 2012, when Google introduced its Knowledge Graph, "things, not strings." Google had bought Freebase, a collaboratively edited database of entities, in 2010, and Freebase's data became the Knowledge Graph's starting content. Chapter 24 shows Freebase browsed live in 2008. Google has never published what its graph runs on, but the contents are expressible as triples, and its public API serves them that way. Knowledge graphs are also built on other graph models; this chapter's examples are RDF.

23.2 The first movers

Organizations were building knowledge graphs before the term went mainstream, and the first movers documented why: a private thesaurus turned into public identifiers, faster publishing at scale, an order of magnitude less code, integration without bespoke pipelines.

In the world, dated 2009. The New York Times had tagged its archive for nearly a century against a thesaurus of "more than a million terms organized into five controlled vocabularies". "Unfortunately, our list of subject headings is an island," its architects wrote. So they gave the first five thousand HTTP URIs and RDF representations under a Creative Commons license, each one mapped by hand to its match in DBpedia (Wikipedia's data as a graph) and Freebase. Today data.nytimes.com no longer resolves.

In the world, dated 2010. For the World Cup the BBC generated 700-plus pages from an RDF triple store, more index pages than the rest of BBC Sport combined. By London 2012 the same architecture kept a page for every athlete, team and discipline, ten thousand of them, a scale its architect called "simply impossible to manage using a static CMS driven publishing stack."

In the world, dated 2011. The Danish comics site Helt Normalt rebuilt its publishing on RDF, SPARQL and XSLT. Its builders told a W3C workshop that the codebase shrank by an order of magnitude against the relational system it replaced. Its ontologies were reused rather than written: even the daily horoscope strip ran on a zodiac vocabulary found on the open web. The platform was Graphity, LinkedDataHub's predecessor (disclosure: the author's, per Chapter 19).

In the world, dated 2013. NXP Semiconductors described its product data as "scattered and duplicated across numerous applications and databases," and published the fix: every product got an HTTP URI, every source was converted to RDF, and SPARQL ran underneath. "The Linked Data is the API." NXP's pages were served by the same Graphity.

23.3 The wave

The wave behind the first movers came a decade later, at the top of the market. NASA runs the systems engineering of its Moon program on an RDF graph. Siemens holds 1.2 million products in one. The banks maintain FIBO, a shared financial ontology, in OWL, the W3C's ontology language. Gartner dated the wave in 2021: "by 2025, graph technologies will be used in 80% of data and analytics innovations, up from 10% in 2021", a figure spanning every graph model, not RDF alone. And AI accelerated the wave. Answering business questions over an enterprise database, an LLM got 17 of 100 right when queried over raw SQL, and 54 over the same data as an RDF graph. Chapter 23 derives what the graph does underneath the LLM.

The words went mainstream too. Palantir has sold its platform's core abstraction as the Ontology since 2018 and put the word in its SEC filing in 2020. Microsoft followed: Power BI's datasets became semantic models in 2023, and Fabric now ships an ontology of its own, "a shared, machine-understandable vocabulary of your business." Neither ontology runs on RDF. The point is small but real: *ontology* was an academic word the industry avoided, and now it is a product name.

23.4 The annotated web

Schema.org is a shared vocabulary that any site can use to describe its own pages, and the ordinary web uses it (Chapter 18). The markup rides in the page. RDFa puts the triples in HTML attributes, so the document a person reads carries the statements a machine reads, with no second copy and no second endpoint. Microdata does the same in attributes of its own. JSON-LD, now the common choice, sets a block of data beside the markup instead of inside it. The October 2024 Common Crawl (the open archive of web crawls) found structured data on 16.5 million of the 37.4 million domains it covered, 74 billion statements in all. Schema.org's own count is higher, over 45 million domains. Nobody assembles any of it into one store. Each site describes itself for one consumer, the search engines, and stops there.

23.5 The open giants

The largest knowledge graphs are not corporate. UniProt, the protein knowledge base, has published its data as RDF since 2008 and holds 232 billion triples behind a public SPARQL endpoint, the largest knowledge graph anyone can query. Wikidata serves about eighteen billion more; it inherited part of Freebase's data when Google closed it. Google last counted its own graph in 2020: 500 billion facts that nobody outside Google can query. And the open graphs link to one another: the Linked Open Data cloud maps 1,360 interlinked datasets as of June 2026.

23.6 The head and the tail

The wave has reached the largest enterprises but few smaller ones, and the reason is arithmetic. Integration pain scales with organization size: General Electric ran about seventy-five procurement systems, Merck about four thousand Oracle databases. Chapter 18 counted the bridges between silos at N × M, and every acquisition raises the count. Adoption follows organization size: smaller organizations run only a handful of data silos, so integration stays manageable by hand and a knowledge graph never becomes necessary.

23.7 The unclaimed half

Many have realized RDF's potential for data integration. Very few have realized its potential for web application architecture. The audit says the same in its own columns: the industry adopted the R-rows and left the S-rows unclaimed. A knowledge graph is Chapter 5's state without Chapter 4's architecture. The two chapters that follow are about the other half.

24. Chapter 23. The Agent Era

Improper architecture is locally cheap and globally expensive: fusing the factors into one program is always less work *today*, and the costs land on caches, crawlers, integrators, and the future. For thirty years the deferral had no consequences. Now it does. Software agents are trying to read the web, and they find what Part IV measured: rendered pixels and private APIs. The response is a compensating industry: scraping harnesses, headless browsers, and a per-application protocol server bolted onto every system whose owners want it read by machines. Read that list against Chapter 11's scores: it is the S4 cost at industry scale, one adapter per application, exactly as the model predicts. The machine-readable web is being retrofitted from outside because the platform abandoned it — Chapter 9's maintenance failure, its cost still compounding.

The arithmetic of that compensating industry is the integration industry's arithmetic (Chapter 18) at a new scale. `N` agents meeting `M` applications through bespoke adapters need on the order of `N × M` integrations. The moment state shares one model and one query semantics, the count collapses to `N + M` — each side implements the common substrate once. Engineers have re-learned this sum in every generation of middleware, and they are re-learning it now, in the agent era, with `N` growing by the month. The per-application protocol server (MCP, the Model Context Protocol, the emerging convention as of this writing) is the `N × M` answer. The protocol is shared, but the model and query semantics are not: each server exposes its own vocabulary, so every agent still learns every application one at a time. The derived stack is the `N + M` answer, shipped since 1999.

The arithmetic also has a way out. Instead of one adapter per consumer, write one adapter per application, translating it into the common model: written once, it serves every consumer. Such an adapter is temporary by design. Once the application serves its own state natively, its answers are the ones the adapter was already giving, so nothing downstream changes and the adapter can be switched off. Until then it can answer in either of two ways: translate from the silo at query time and store nothing, or import the translated facts into the consumer's own dataspace, where they survive after the silo account that supplied them is closed. And the adapters need not be built from scratch: the industry already maintains scrapers and protocol servers around every silo, and aiming them at the common model turns them into the bridge.

The vision, dated 2001. This chapter derives a scenario that was written as fiction twenty-five years ago. The May 2001 *Scientific American* article "The Semantic Web" is by Tim Berners-Lee, James Hendler, and Ora Lassila. It opens with Lucy's agent negotiating a course of medical appointments over machine-readable data on her behalf, an agent reading the web, not scraping its pixels. It read as science fiction because the agents did not exist. They exist now. Chapter 8

said the substrate was built for machine consumption and the machines were twenty years out; those twenty years have now elapsed.

Underneath runs the grounding problem. Statistical models interpolate, and interpolation hallucinates. What agents need beneath them is a substrate whose answers are computed rather than guessed. Fact-sets with a formal query semantics are that substrate, and Part III named the deployed one. The two divide the work, statistical model above, fact-set substrate below. Where the substrate covers a question, the answer is computed from it; the model interprets the rest, the rare and one-off questions it does not yet cover. What proves valuable there is promoted into the substrate, and promotion is simply a write: the fact is asserted into some party's dataspace and attributed to that party, like any other fact. The substrate is governed as any origin is, by whoever holds the right to write it. And integration accumulates instead of repeating: every promoted fact composes by union, and stays.

"Machine-consumable" requires nothing the book has not already derived: state with a universal model (R1), a coordination-free merge (R2), global reference (R3), and addressable intermediates (S4). An agent is a user agent. The requirement was in Definition 1.1′s type signature from the start.

And reading is only half of Definition 1.1. The write side serves agents even better. An agent's change arrives as Chapter 7′s delta: two fact-sets, (D^-, D^+). The delta is a *reviewable object*: a human can inspect it before it applies, an audit log can store it verbatim, an operator can invert it by swapping the sets. The alternative is an opaque API call: its effect is whatever the endpoint's code does, and nothing can reverse it. Agent autonomy is a governance problem only where agent actions are opaque, and the delta normal form makes the action a document. And what holds for one change holds for a whole plan of them. An agent's entire intended course can be stated as one document and read before any of it runs: what it will read, what it will change, what it will do if an answer comes back empty.

The chapter ends with a question, put to any agent directly: *is it more efficient for you to write a custom system for every domain, or to reuse one generic system and define the domain as data?* Reusing one generic system is plainly the more efficient of the two. What stands between agents and that option is the set of human mental models Part IV audited: pre-web paradigms, defended now by habit rather than argument. The web that agents need is the web this book derived.

25. Chapter 24. The Next Web

Chapter 21 stated the result; Chapter 23 described the agents that read it. What remains is the web the agents and the dataspaces compose together: Web 3.0, as Chapter 21 defined it. Everything this chapter describes is already permitted by the derived properties; none of it is a forecast. No one builds and ships it: parties take it up one origin at a time. It does a handful of things no fused web can do.

This web gives the end user three things. The end user can navigate and drill into any data without knowing a query language, because Chapter 7's five moves (navigate, write, restyle, rearrange, reselect) are interface primitives and none of them requires a programmer. The end user can fork and augment a running application declaratively — swap the stylesheet, change a layout, add a saved query — with no vendor involved. S3 becomes a right the user exercises rather than a courtesy the vendor grants. And the end user can federate without asking permission: federation is nothing more than merging two states, and merge is union (5.1).

None of this asks the existing web to stop, or even to notice. A silo doesn't have to migrate to be included: wrapped (Chapter 23), it enters a federation as a view of itself, before its vendor has agreed to anything. So the transition to this next web has no launch and no platform to join. What forms instead is an accumulation: small, private dataspaces gathering around the silos until the silos hold the copy and the dataspaces hold the original. Each dataspace is detachable from the services it summarizes, from the software that serves it, and from the machine it happens to sit on. Union is additive: a dataspace runs alongside the systems already in place, and whatever stays fused keeps the failures Part IV measured. Only one thing is required: that new systems be built as dataspaces, publishing their state, instead of as applications around private databases.

The rest of the chapter walks through these capabilities, from a single application in the user's hands out to the network that forms when many parties build this way.

25.1 The application bends

When the application's state is a graph, the reader can reshape the application — navigate it, re-render it, reselect from it — in ways no fused page allows.

25.1.1 Navigating the graph

Navigation is the first of the five moves, and on a graph it does what the document web never could. Every fact links two entities, so every relationship in the data can be browsed the moment it is stated. On the document web a browsable path existed only where someone had built the page and its links; here it exists wherever a fact does. Narrow a set by its own facts (products

in stock, turbines below tolerance) and the filter is a selection, not a screen someone wrote. Follow a relation out of that set and arrive at the set it relates to: from a filtered list of products to the list of their vendors. That is a set reached from a set rather than a page linked to a page. Chain the two moves and the reader can explore the whole graph freely. Each intermediate set is a resource with its own address (S4). The document web's navigation ends at an item's page and its authored links; on a graph it continues through every relation the item has.

In the world, dated 2008. This is not speculative. David Huynh's *Freebase Parallax* demonstrated exactly this move over graph data in 2008 (a set carried in one step to the set it relates to) on real data, years before this book described it. Metaweb, its maker, was acquired by Google in 2010, and Freebase went into the Google Knowledge Graph. The demonstration is eighteen years old. The missing piece was the state model derived here, not a new invention.

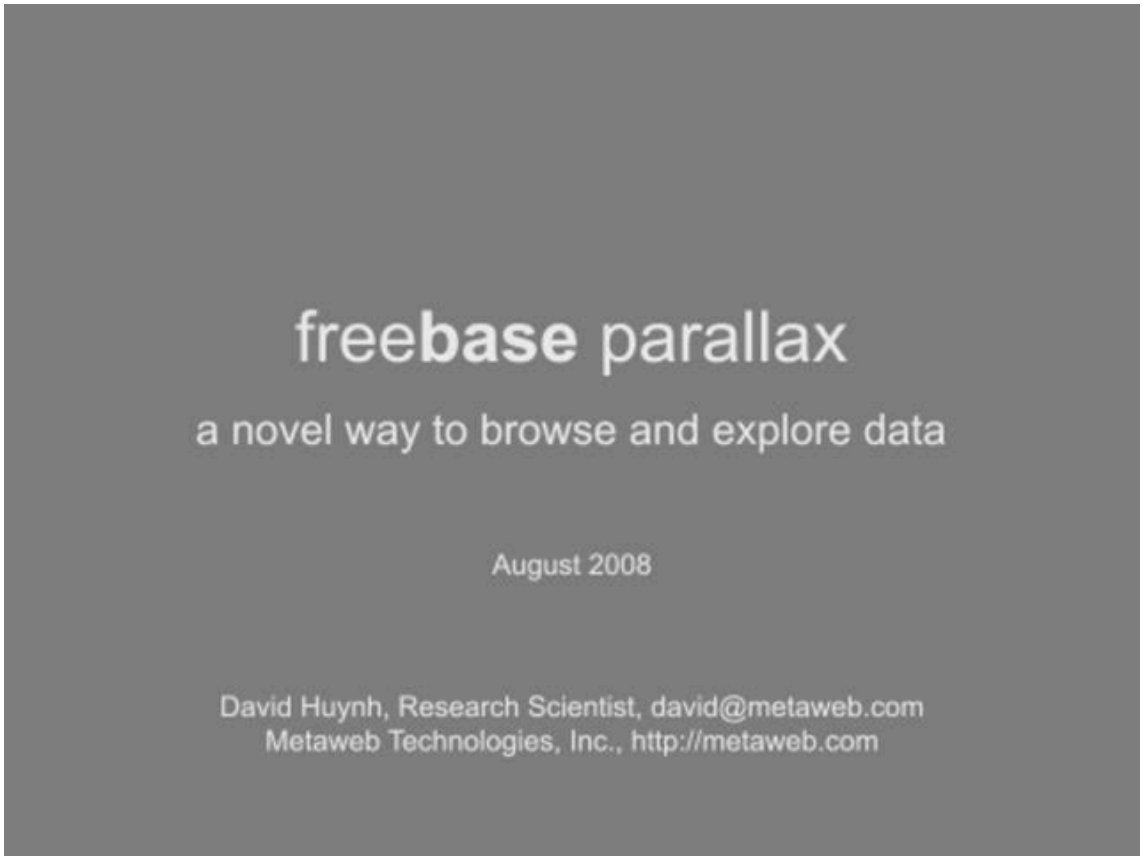


Figure 25.1: Freebase Parallax: A new way to browse and explore data

► Watch online: Freebase Parallax: A new way to browse and explore data — https://vimeo.com/1513562

25.1.2 One state, many faces

Presentation is its own factor (S1), independent of the facts beneath it, so anyone can render a publisher's state in their own way. The reader who needs large type, a screen reader's linear order, another language, a watch-sized screen — each takes the same facts through a different

`present` term and gets a document fit for them. Accessibility stops being a retrofit and personalization stops being surveillance: the state is shared and the rendering is the reader's own. The document web bolted this on (parallel mobile sites, accessibility overlays) because content and presentation were fused. CSS Zen Garden (2003) showed the other way at the presentation layer alone, dressing one unchanged HTML document in hundreds of unrecognizably different designs. It reached this independence at one factor only, because the rest of the skeleton stayed fused.

25.1.3 A feature is a package

An application on the derived web is data, a vocabulary and a stylesheet over state (Chapter 20), so to change what it does is to change data, not code. A feature ships as a *package*: the terms it introduces and the templates that render them. The package merges into a running application and is withdrawn by the reverse delta, asserted and retracted the way any fact is. Nothing recompiles and nothing redeploys, because there is no application-specific code to rebuild — the catalogue becomes a storefront once a checkout package merges, and a catalogue again once it is retracted. And since the change is a delta, applying it needs neither a programmer nor even a person. A human installs it from the interface, or an agent submits the same delta (Chapter 7); one request later, the application has the new behavior. S3 stops being a vendor's release cycle and becomes an ordinary write, open to anyone holding the right to make it.

The wish is old: HyperCard let people reshape a running application in place in 1987. What it lacked was a substrate where the reshaping composes across parties instead of trapping the stack on one machine.

In the world, dated 1987. HyperCard shipped with the Macintosh: stacks of cards a user could edit while using them, buttons and behavior rearranged in place.

Figure 25.2: The Computer Chronicles — HyperCard (1987)

► Watch online: The Computer Chronicles — HyperCard (1987) — https://youtu.be/FquNpWdf9vg

In the world, prototyped. This is running code. LinkedDataHub (Chapter 19's reference implementation) ships a package as exactly an ontology and a stylesheet: installing one merges its vocabulary into the running application and adds the XSLT templates that render it; uninstalling one is the reverse delta. The operation is an ordinary authenticated write, so an owner runs it from the interface and an agent runs it over the same endpoint, with nothing rebuilt or redeployed. It is young: it shipped in 2026, it carries one feature so far, and a new package's templates take effect only after the stylesheet recompiles. But the mechanism is the derived one, and it runs in production.

25.2 The machine reads

The agent reads rather than scrapes (Chapter 23's diagnosis, flipped to a capability), and each fact arrives with its source attached: the asserting party is recorded in the fourth position (Prop. 9.2), which is the graph's own name. A claim and its provenance are one object, so a fact cannot be separated from who said it. On a web of rendered pixels a fabrication looks the same as a record; on a web of attributed facts, *who says so?* is answered in the data, not reconstructed after it. This does not make claims true (attribution is not verification) but it makes them

accountable. Every assertion names a source to query, corroborate, or impeach, and an agent merging two graphs sees exactly which origin contributed which fact. The Semantic Web stack always sketched *proof* and *trust* as its top layers; the deployed web built the fact-sets and deferred proof and trust. Machines reading at scale now make those layers urgent.

25.3 The data is yours

The deepest change is in the state itself: it belongs to the party it describes.

25.3.1 State stops being scattered

Under one model and one law of merge, what lives today across a dozen unspeaking silos composes into a personal dataspace: a body of facts a party keeps under its own origin and federates with the origins it trusts. Nothing is copied around to go stale: dereferencing a name returns the owner's current answer, because each origin is the authority on its own data.

The vision, dated 2009. This section's personal dataspace has an earlier name. In *Pull* (2009), David Siegel called it the *personal data locker*: a person's world (home, possessions, finances, media, health) held as one graph under its owner's control. The services that touch it become sources that read and write, instead of keepers of their own copies. Siegel argued from market economics rather than from requirements and reached the same design this section derives: state gathered under its owner's origin instead of scattered across silos. He described it years before any substrate could support it.

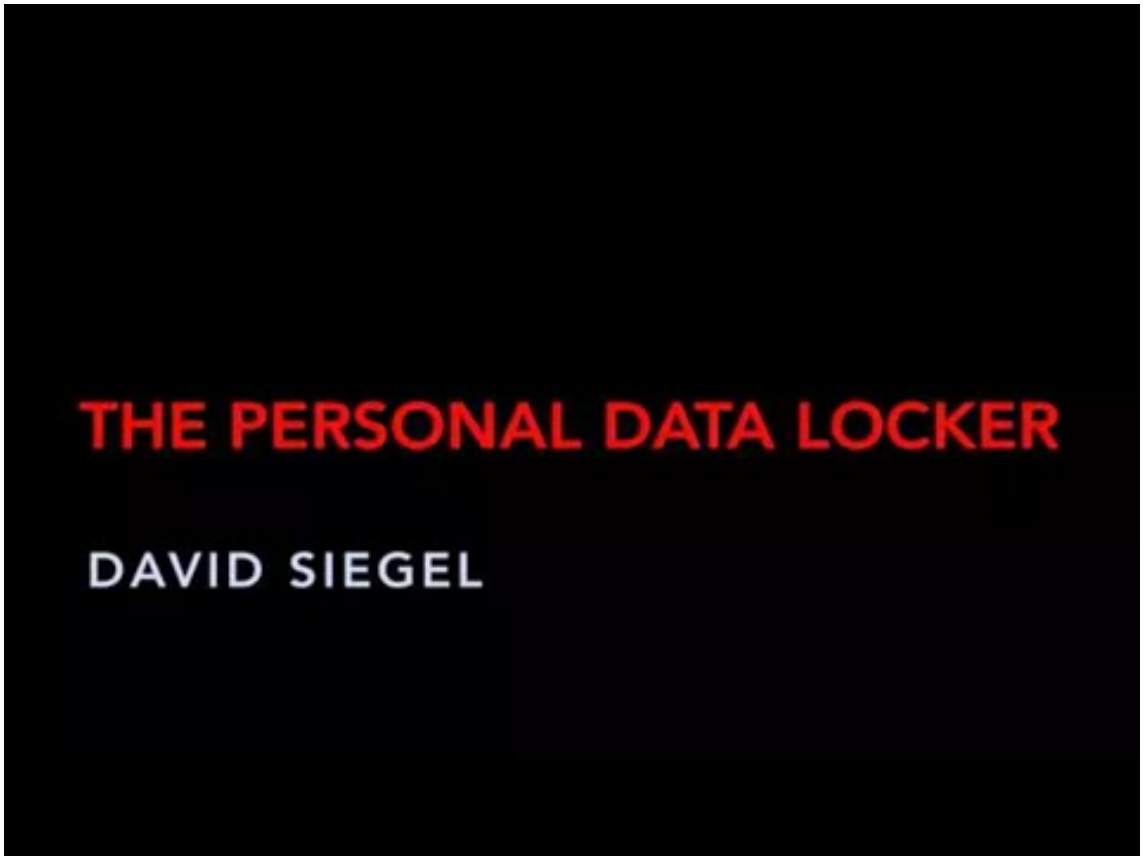


Figure 25.3: David Siegel — Personal Data Locker Vision

► Watch online: David Siegel — Personal Data Locker Vision — https://youtu.be/xOch5o3MhUg

25.3.2 What outlives the software

State lives under its owner's origin, not inside the software that happens to touch it, so it outlives that software, the application and the agent alike. The service shuts down, the vendor is acquired, the framework is rewritten, the assistant is replaced by a better one; the facts remain where they were, under a name that still resolves, readable by whatever reads next.

On the fused web an application takes its contents down with it: when the account closes, its history is gone, and when the link rots, the record is gone. What an agent has learned about its user is locked in the vendor's assistant and is lost the same way. Here the code is the part that gets replaced and the data is the part that lasts: an agent gives way to a better one with nothing forgotten, because its memory was state under the served party's origin, never the agent's to keep.

Even the conversation that produced a change can be kept as state, each turn a document, so a delta's provenance can point at the very utterance that caused it. Why something was done is then answered by reading the record, not by hoping a model remembers.

The principle is old and mostly ignored. Berners-Lee's *Cool URIs don't change* (1998) asked publishers to keep their URIs stable, and most publishers did not. Bush's 1945 proposal (Chapter 6) wanted a personal store that no session's end would erase. Both wanted what owner-held state supplies: a name that still answers, durable past the software that reads it.

25.3.3 Permission is a fact

What may be read, and by whom, is stated in the same state it governs, narrowed or withdrawn as an ordinary change, with no platform to grant access and none that can revoke it. One party opens a region of its world to another's agent with a single assertion, and the boundary holds at every hop of a query that crosses between them. Composition and disclosure are not the same boundary. Any two states still compose without permission (R2), but what a given agent may read is a projection the permissions cut, so universal composition never meant universal visibility.

The vision, dated 2016. The web's own inventor built toward this. Solid (Tim Berners-Lee's re-decentralization project, begun in 2016) gives each person a *pod* whose access is itself data. Web Access Control rules are held beside the resources they govern; they state who may read what and are edited and revoked like any other fact. Permission as a statement rather than a platform setting is Solid's design and this section's, for the same reason: put access in the state, and no intermediary owns the gate.

25.4 What gets done

Reading is half of Definition 1.1; the write side turns capability into action.

Search runs one way: a person forms a query and the market's pages answer, each provider having guessed in advance what to publish. Structured state runs it the other way. A need is itself facts (constraints, preferences, the criteria that qualify an answer) so it can be published as a document and read by providers, instead of typed into a search box and retained by the platform. A person states once, in the open, what they want; whoever can meet it answers, matched against the very facts that generated the request, nothing re-entered and no form filled. Stating a need is Chapter 7's write, addressed to the market instead of to one application.

The vision, dated 2006. Publishing a need instead of searching for one has a name. Doc Searls called it the *intention economy*. The term dates to 2006 and the book to 2012. In that market a buyer broadcasts a qualified intent and sellers compete to satisfy it, which is search run backwards. The practice took a verb, *intentcasting*, and Siegel's *Pull* (2009) tied it to the data locker above: the locker stores the facts, the intentcast publishes them as demand. What it always lacked was a substrate of owner-held state for the market to read.

25.5 The network forms

And once many build this way, the whole becomes more than its origins. Because states merge by union without a coordinator (R2) and any stylesheet can be pointed at the result (S3), value arises from unplanned combinations. Two dataspaces compose the moment their names meet,

though their owners never coordinated, and a third party who owns neither can render the join as something new. The document web tried this once and lost it: the mashups of the mid-2000s stitched maps and listings into applications nobody's vendor had shipped, until the platforms metered their APIs and re-siloed the data, ending the combinations at their own discretion. Union grants no such power. The joins hold because no platform owns the point where the data meets, so combining other parties' data becomes routine rather than a fad.

And the loop compounds: publish a dataspace, and every dataspace federated with it is worth more. This is the network effect that once turned a single physicist's filing system into the world's front page. It now reaches the layer Web 2.0 hid, this time with no one in the middle owning the graph or charging rent on the joins.

The same compounding reaches the agent. When the agent meets a new domain, it needs no new system: it states the domain as facts over the one generic engine, and the work is finished. Chapter 23 put a question to the agent (a bespoke system per domain, or one engine with the domain as data); the answer is obvious.

So the next web is not built by a consortium or shipped in a release. It begins wherever someone stops the lateral churn and the taste-based technology trends, and treats the web as this book has treated it: from first principles, as a science. Those who do, the agents included, will have an edge over those who don't, and the gap will only grow, because churn starts over and derivation compounds. Which curve do you want to be on?

26. Draft status

All twenty-four chapters and Appendices A–D are drafted in prose, with exhibits, scored audit columns, and full proofs (B.1–B.9). Under construction: the Chapter 19 reconstruction exhibit, the mechanization of the proofs, and the online edition. Feedback is most valuable on R1–R3, the arity argument, and the Transposition Thesis (Chapter 5, Appendix B) — if something is smuggled, it is there.

Part	Status
Preface, The Argument in One Page	drafted
Ch 1, 2	drafted
Ch 3	drafted; screenshot exhibits captured
Ch 4, 5	drafted — the core of the analysis
Ch 6, 7	drafted, with figures
Ch 8, 9	drafted — the reveal and the mismatches
Ch 10–17	drafted — every audit column scored; the table assembled
Ch 18–21	drafted — LinkedDataHub as reference implementation; reconstruction exhibit pending
Ch 22–24 (Part VI)	drafted — knowledge graphs, the agent era, and the next web; `part6` frontispiece pending
Appendices A, C, D	drafted
Appendix B	complete — B.1–B.9, machine-checked in `proofs/` to the scope Appendix C states
Rigor & prior art	uniqueness, arity, and genericity checked against prior work; corroborations and the full prior-art ledger are in Appendix D
Figures	mermaid diagrams and screenshot strips complete; Chapter 19 reconstruction exhibit pending

27. Colophon

First Principles of the Web — Draft 0.1.

Built from a single Markdown source with Quarto; diagrams are pre-rendered to SVG. Available as HTML, EPUB, and PDF. Source at github.com/AtomGraph/First-Principles-of-the-Web.

Appendices

A. Method, notation, and reading order

Persuasion is what you need when you don't have a proof, so this book runs on apparatus, and the apparatus has rules. A statement's sources come in the three kinds the preface names: spec definitions, earlier propositions, checkable observations. A fourth kind, called *witnesses*, corroborates but never serves as a premise: these are documents that stated as norms what this book derives as theorems. Strike every witness and no proof changes. One claim is deliberately unprovable, flagged where it stands: the Transposition Thesis of Chapter 5, the bridge between the formalism and the web itself. It is secured there and in the appendices, never proved. Disagreement belongs at the bridge; given the requirements, the theorems add no premise of their own.

The shape of the whole is Chapter 2's method, run at book length: analysis and synthesis, in the geometers' sense. Parts II–III strip the web application as found and derive what its parts must be. Part V builds the application space back from the derived parts. The two directions meeting exactly is the book's proof. Part IV, between them, audits what the industry runs instead, ending with the derived stack itself. There is no editorializing, only scores against Part II's seven properties: three requirements on state (R1–R3), four separations on architecture (S1–S4). Its closing chapter is the completed table.

If you have ever read a type signature, you have read every formula in this book: `×` is a tuple, `→` a function; the crib below translates the rest. Proofs are collapsed: the claim stays in the text, and the argument opens on demand. Results carry names, because prose argues by name; the numbers let the appendices argue by label.

This book uses sets, tuples, total functions, and composition ∘ — first-year material, as promised. Two conventions: ⟦ · ⟧ is a denotation function and always someone else's, cited from the governing specification, never defined here; `t` names the arrange term (S2), so time is written `τ`. The numbered apparatus is as follows. R1–R3 are the requirements on state, with R4 (attribution) added in Chapter 9. S1–S4 are the separation properties of a factorization. The B-conditions are the formalizations in Appendix B. Propositions are chapter-numbered.

The symbol crib, for readers who live in code:

symbol	reading	in code
`A × B`	a pair: an A and a B	a tuple; a two-field record
`A → B`	function from A to B	`(a: A) => B`
`𝒫(A)`	all sets of As	`Set<A>`
`∪`, `∩`, `∖`	union, intersection, difference	`union()`, `intersection()`, `difference()`

symbol	reading	in code
`⊔`	disjoint combination of independently asserted structures	two sources' data, side by side
`∘`	composition, right to left	`compose(f, g)`
`⟦q⟧`	what `q` means, per its spec	the standard defines what your query returns, not your driver
`s ⊕ s′`	merge two states	set union of their facts — the union law (5.1)
`s ⊕ s = s`	idempotence	re-merging is a no-op; safe to retry
`(D⁻, D⁺)`	a delta	a diff: deletions, additions
`≅`	isomorphic	same shape; lossless conversion both ways
`≡`	logically equivalent	equal after normalizing — same meaning, maybe different syntax
`τ`	time	a version, a timestamp
`I`	the set of URIs	globally unique identifiers (RFC 3986)
`V`	atomic literal values, disjoint from `I`	strings, numbers, dates — the leaves
`O`	the set of origins	a scheme–host–port triple (RFC 6454)
`I|o`	the URIs under origin `o`	one party's region of the namespace

Reading order: Parts I–III go linearly. Chapters 10–15 need only Chapter 8 and go in any order, then Chapters 16 and 17. Part V needs nothing past Part III. Three tracks, if you are choosing a path:

- In a hurry: the Preface and The Argument in One Page, then Chapters 3, 8, 16, and 21.
- Building things: the hurry track, plus Chapters 7, 14, and 19.
- Refereeing: Chapter 5 and Appendix B, where the load-bearing walls are.

The formulas are skippable and the prose carries every argument; the formulas make the prose auditable.

The named results follow, so that a reader can move between the prose and the apparatus:

name	label	where
the trivial factorization	Prop. 4.2	Ch 4
properness	Def. 4.3 (S1–S4)	Ch 4
the analysis theorem	Prop. 4.4	Ch 4; B.5
independent evolution	Prop. 4.5	Ch 4; B.6
the union law	(5.1)	Ch 5; B.1

name	label	where
the arity argument	Prop. 5.2	Ch 5; B.2
the shape of a fact	(5.3)	Ch 5; B.2
the uniqueness theorem	Thm. 5.4	Ch 5; B.3
the classification	a thesis, deliberately unnumbered	Ch 6
the canonical serialization (`canon`)	Prop. 6.1	Ch 6; B.8
the delta normal form	Prop. 7.1	Ch 7
forms as inverse transforms	Prop. 7.2	Ch 7
one algebra, both directions	Prop. 7.3	Ch 7
the five moves	Prop. 7.4	Ch 7
the homomorphism	Prop. 8.1	Ch 8; B.7
the synthesis theorem	Thm. 8.2	Ch 8; B.8
the bill for anonymity	Prop. 9.1	Ch 9
the erasure argument	Prop. 9.2	Ch 9
the dataspace	(18.1)	Ch 18; B.9
nothing else to vary	Prop. 20.1	Ch 20
the Transposition Thesis	a thesis, deliberately unnumbered	Ch 5; B.2

B. Proofs

Prop. 5.2 and Thm. 5.4 come first, reached through B.1's formalization of R2. They are the two results everything downstream rests on, so they get the most care. Then come independence (B.4), analysis (B.5), timelines (B.6), the homomorphism (B.7), synthesis with genericity made exact (B.8), and federation closure (B.9).

Machine-checked. A Lean 4 companion to this appendix lives in the book's repository under `proofs/`. Appendix C reports what it checks and what must be assumed.

B.1 B.1 R2, formalized — and the representation lemma

Chapter 5 argued in prose; a proof needs the requirements as mathematics. The translation is itself the honest step: every choice below is a numbered condition with its one-line justification from the web, so that rejecting one is a precise act rather than a suspicion. Chapter 17's Properness Table already tells you what each rejection costs. Conditions carry the number of the requirement they formalize (B-2a–e for R2, B-1 for R1, B-3 for R3, B-0 for the form-level ground), hyphenated, to keep condition B-1 apart from section B.1 and its lemma.

Fix `I`, the URIs (RFC 3986), and `V`, a set of atomic literal values disjoint from `I`. A **state model** is a pair `(M, ⊕)`: a set of states and a composition. R2 (composition among parties who have never communicated) formalizes as four laws and one closure condition:

- **(B-2a) Totality.** `⊕` is defined on every pair of states. Composing may never require a compatibility check, because checking is coordinating.
- **(B-2b) Order-freedom.** `⊕` is associative and commutative. States arrive from independent parties in no agreed order; if order mattered, the order would have to be agreed.
- **(B-2c) Idempotence.** `s ⊕ s = s`. On the web copies are free and copies of copies are unmarked; a state received twice is the state received once. A model that counts arrivals must know which arrivals are "the same sending", and that knowledge is coordination.
- **(B-2d) Atomicity, no emergence.** Write `s ≤ s′` for `s ⊕ s′ = s′` ("s says at most what s′ says"). `M` has a least element `∅`, the party with nothing to say. An **atom** is a minimal state above `∅`. Require: every state is the join of the atoms below it, and `atoms(s ⊕ s′) = atoms(s) ∪ atoms(s′)`. In words: a state says exactly what its atoms say, and composition neither creates nor destroys atoms. This is Chapter 5's "a fact must carry its full meaning with it," as algebra. If a combination of states could mean more (or less) than its facts together, the surplus would live in an arrangement. Some union would fail to preserve it, and the parties would have to agree on its interpretation, coordination again.
- **(B-2e) Accumulation.** Every directed family of states has a join, with `atoms` of the join the union of the family's atom sets. This one is bookkeeping, not a transposed law, which

is why Chapter 5's table has four rows, not five. The four laws govern the merge of finite messages; a store keeps receiving, and B-2e says the model has a state for the limit of any such accumulation.

Lemma B.1 (Representation). A state model satisfying B-2a–e is isomorphic to `(𝒫(A), ∪)`, where `A` is its set of atoms. *Proof.* Map `s ↦ atoms(s)`. Injective: by B-2d every state is the join of its atoms, so two states with the same atoms are the same join. Surjective: a finite set of atoms is realized by its join (B-2a supplies the join; B-2d's second clause guarantees the join's atoms are exactly the atoms joined), and an arbitrary set is the directed join of its finite subsets' realizations (B-2e supplies that join and its atoms). Homomorphism: `atoms(s ⊕ s′) = atoms(s) ∪ atoms(s′)` is B-2d verbatim. ■

(Without B-2e the lemma holds in the finite: the atom map embeds `M` into `𝒫(A)`, and its image contains every finite set of atoms, the version that carries the operational content, since messages are finite. B-2e is exactly what the full powerset costs, and it is now on the bill rather than in a remark.)

Rename `Fact := A`, and (5.1) is proved from the named conditions. (A scope note for readers arriving from deployed RDF: over ground atoms, composition is set union exactly. With blank nodes, RDF itself distinguishes *union* from *merge*, standardize-apart, the Merging Lemma of RDF Semantics (2004), and the laws hold up to logical equivalence, as Prop. 9.1 states and measures. The characterization is over ground atoms; 9.1 is its honest extension.) The rejections are visible already — each law rejected is a deployed model, catalogued after Theorem 5.4 in B.3.

B.2 B.2 Proposition 5.2 — the arity of a fact

Prop. 5.2 (restated). The minimal self-contained fact is a triple.

What remains free is the structure of an atom. Three more conditions name what Chapter 5's prose used:

- **(B-0) Finite, self-interpreting atoms.** An atom is a finite tuple over `I ∪ V`, one length per model (a reading that dispatched on length would be two readings), and its meaning is a *fixed, universal* function of the tuple alone. This is B-2d again, at the atom's own scale: no atom means by way of its neighbors, and no atom's reading varies by domain or by party. One reading, agreed once, for everything. (Agreeing on that single reading is itself an act of coordination, performed once, about form, never about content. That is what a specification is. R2 forbids per-domain agreements, and licenses exactly this one.)
- **(B-1) Faithful universal encoding** (R1). For every finite relational structure `D` (entities and relations, both named; the shape every deployed data model reduces to) there is an encoding `enc(D) ⊆ Fact`, injective up to isomorphism, and compositional: independently asserted structures encode independently, `enc(D ⊔ D′) = enc(D) ∪ enc(D′)`.
- **(B-3) Global names** (R3). Names occurring in `D` are drawn from `I` and occur verbatim in `enc(D)`: references must survive encoding, or cross-source references stop matching at exactly the moment sources encode independently.

Arity 1 fails. A 1-tuple `(x)` occurs or does not occur; by B-0 its meaning is a function of one name. For `enc` to distinguish `R(a,b)` from `R(b,a)` (same names, different structure) some atom must mean a whole proposition. A bare name can only mean a proposition by *assignment*: its owner publishes, somewhere, what the name stands for. But the publication must itself be stated in some model, and if that model is again bare names, the regress never grounds. A model of pure names is parasitic on a model of higher arity. Naming is not asserting.

Arity 2 fails. By B-0 a pair carries one fixed universal reading, some single relation between its components; a reading that varied per pair would be per-pair agreement, which is coordination. One fixed binary relation is expressively a single unlabeled directed graph, and B-1 demands arbitrarily many distinguishable relations over the same entities: `R(a,b)` and `S(a,b)` must encode differently, yet the pair `(a,b)` is all a pair can say. The remaining escape is the gadget: encode the relation's identity as a *shape* built from fresh nodes around the edge. Two conditions kill it. First, shapes must be assigned to relation names by a global scheme over the unbounded namespace `I`, a label registry, which is a central schema authority, the exact thing R2 excludes. Assigning shapes by dereference instead reruns the arity-1 regress. Second, inside a gadget the individual pair means nothing until the whole shape completes. Atoms have stopped carrying their meaning, and B-2d is violated at atom scale. Two parties independently asserting gadget-encoded facts about shared entities can union into a state whose decoding is ambiguous. The pair's failure is Chapter 5's line made precise: `(employee42, "2026-07-08")` is hired-on, or fired-on, or born-on, the meaning lives outside the fact.

Arity 3 suffices. The one universal reading: `(s, p, o)` asserts *the relation named `p` holds of `s` and `o`*. The atom names its own relation — position two supplies what arity 2 lacked, and B-3 draws it from `I`. The encoding, case by case. A binary fact `R(a,b)` becomes `(a, R, b)`. A unary classification `P(a)` becomes `(a, kind, P)` for one distinguished attribute `kind`, licensed like the reading itself: one form-level convention, agreed once. An n-ary fact $R(a_1, \ldots, a_n)$ becomes a fresh entity `e` with atoms `(e, rel, R)` and (e, role_i, a_i), the role names published by `R`'s owner alongside `R`. Minting fresh names is free, and every party owns a namespace to mint in (RFC 3986's authority component). Note the contrast with the gadget: here every atom still means alone ((e, role_2, a_2) says, completely and context-freely, that `e`'s role_2 is a_2) and the n-ary fact is the *conjunction* of its atoms. Accumulation, never emergence. Faithfulness and compositionality are routine to check; injectivity is up to the choice of fresh names, a degree of freedom Chapter 9 meets again under its deployed name.

Minimality pins three. Arities 1 and 2 fail; arity 3 succeeds; and every arity above 3 also succeeds: quads meet every condition. The conditions bound arity from below only. What selects three is parsimony: any k-model with `k > 3` encodes into the 3-model by the same decomposition just given, machine-checked in Appendix C, so the 3-model is the minimal universal one, and minimality is stated in Theorem 5.4's hypotheses rather than smuggled. The wider tuples become minimal only under an added requirement: make facts themselves attributable (R4) and the minimum becomes four (Chapter 9).

Positions, typed. Position 2 lies in `I`: the reading makes it a relation *name* whose meaning must hold across independent sources. Shared meaning without coordination is ownership plus documentation, which only `I` provides; a literal owns nothing and dereferences to nothing. Position 1 lies in `I`: subjects are where facts accumulate across sources, accumulation is join-on-subject, and joining beyond a single source requires reference. A literal denotes itself — you do not add facts *to* the number five. Position 3 lies in `I ∪ V`: descriptions must terminate in values, or no fact ever touches data. Literal subjects add no expressive power (any structure "about" a value factors through the entities that carry it) so minimality removes them. This yields (5.3): `Fact = I × I × (I ∪ V).` ■

Scope, and the history that shapes it. The mathematics here has a pedigree and a trap, and both belong on the table. As pure relation theory, the shape of this result is **Peirce's Reduction Thesis**: dyads insufficient, triads sufficient, higher arities decompose. The thesis was conjectured in the 1880s and proven in modern form by Burch (1991), Dau & Hereth Correia (2006), Hereth Correia & Pöschel (2011), and Koshkin (2022–2025). The arity theorem is not new mathematics, and this appendix does not claim it. What is claimed is the derivation of its *hypotheses* from web-state axioms, and the typing of its positions into `I` — no work in the Peirce line mentions the web or its data models.

The trap: the insufficiency of pairs is *operation-relative*, and stated without qualification it is false. Löwenheim (1915) and Quine (1954) proved that under unrestricted set-theoretic pairing, every relation of every arity reduces to dyads, the same fresh-entity move this proof uses to decompose n-ary facts, pushed one step further. What blocks the push here is B-0 and B-2d: the pairing reduction manufactures atoms that no fixed universal reading interprets alone, pairs that mean only via their neighbors. That is the gadget escape, closed above and machine-checked in Appendix C. In Peirce's setting, the analogous restriction has been accused of gerrymandering (Skidmore 1971; Koshkin 2022): drawn where it must be for triads to win. In this setting the accusation has an answer the Peircean one lacks. The restriction is the Transposition Thesis's fourth row: no meaning survives in arrangement. That row is a deployed invariant of the web, adopted for reasons that predate any question about arity. The web drew the line, not the theorem.

One prior assertion completes the record. Robertson (2005) wrote of RDF that ternary relations are "the minimal ... way to encode semantics wherein metadata may be treated uniformly with regular data", asserted as motivation for a triadic query algebra, underived. This appendix is, among other things, the derivation that assertion was owed.

B.3 B.3 Theorem 5.4 — uniqueness, assembled

Theorem 5.4 (restated). Isomorphism of state models here is *reading-preserving*: the bijection carries ⊕ to ⊕ and atoms to atoms, and on atoms it acts position by position, holding names and values fixed. It may permute tuple positions and relabel nothing. Bare ⊕-preserving bijections are too coarse to carry the claim: any two powersets of equinumerous atom sets admit one, and

the widened tuples would pass. The content lives in the structured notion. Let `(M, ⊕)` satisfy B-2a–e, B-0, B-1, B-3, and among such models be arity-minimal. Then `(M, ⊕) ≅ (𝒫(I × I × (I ∪ V)), ∪)`, reading-preservingly.

Proof. Lemma B.1 gives `M ≅ 𝒫(A)` with `⊕` carried to `∪`. B.2 gives `A = I × I × (I ∪ V)` up to permuting positions: arities below three cannot satisfy B-0/B-1/B-3, arity three can, and minimality excludes the rest. The product is full because B-1 refuses no structure — every triple over the typed positions is `enc` of some `D`, so every triple is an atom. Compose the isomorphisms; each fixes entries and at most permutes positions, so the composite is reading-preserving. Uniqueness is up to the permutation of tuple positions, a renaming of the reading, and not a distinction worth disputing. ■

The rejections, restated with their numbers. Reject **B-2d** and meaning moves into arrangement: the document family (XML, JSON) audited in Chapter 10. Reject **B-3** and names stop at the database boundary: the relational world and its integration industry, Chapter 13. Reject **B-1** and the format serves one domain: the per-API bridge industry, Chapter 13 again. Reject **B-2c** and you are modeling events; their replay into state must land in a model satisfying the rest, and the rejection returns you here. Reject **minimality** upward and you have quads, the one rejection that leads deeper in rather than out, Chapter 9. Each rejection has a name, a chapter, and a cost.

B.4 B.4 Independence of the conditions

None of B-2a–e is redundant, and the proof is the standard one: for each condition, a model satisfying the other three (suitably restated where the dropped law is presupposed by another's phrasing) in which the characterization fails.

- **Drop B-2a (totality).** Relational union under schema compatibility: composition defined only where schemas agree. Order-free, idempotent, and atomic where defined, and composing across independent parties now requires the compatibility check, which is the coordinator returning. States are schema-indexed families, and Lemma B.1′s target is gone.
- **Drop B-2b (order-freedom).** Event logs under append, deduplicated by entry identity: total and idempotent, and the composite depends on interleaving, so two parties' logs have no canonical join. The structure is a monoid, and the representation fails.
- **Drop B-2c (idempotence).** Multisets under multiset union: total, order-free, atom-determined, and `s ⊕ s ≠ s` semantically, because arrival counts. The representation lands on `ℕ^A`, and federation now needs to know which arrivals are "the same sending", provenance machinery, which is coordination.
- **Drop B-2d (atomicity).** `(ℕ, max)`, Bloom^L's own `lmax` lattice: total, order-free, idempotent, with least element 0 and sole atom 1. The state 2 is not the join of the atoms below it, so states are no longer determined by their atoms and the representation fails. Trees under meaning-bearing grafting fail the same way at scale, and Chapter 10′s audit is the deployed consequence.

- **Drop B-2e (accumulation).** The finite subsets of an infinite `A` under union: all four laws hold, and the store has no state for the limit of an unbounded accumulation. The representation lands on the finite-subsets lattice, short of the powerset.

Each law's countermodel is one of Chapter 5's rejections, its non-redundancy now proved: remove any law and the uniqueness theorem loses its target. All four are load-bearing, so the work of the characterization is distributed — no single condition smuggles the conclusion. For B-2d the literature supplies a deployed witness. Bloom^L (Conway et al., 2012) generalizes coordination-free programming from relations-under-union to *arbitrary* lattices with associative, commutative, idempotent merge, counters, maps, booleans. It demonstrates in running code that the merge laws alone leave the data model open, exactly as this section claims. ■

B.5 B.5 The analysis theorem (Prop. 4.4)

Formalize the hypothesis first. *Finite dependence* says: for every request `r` there is a finite set of facts `K(r) ⊆ Fact`, the request's *window*, such that `read(r, S) = read(r, S ∩ K(r))` for all states `S`. In words: the response depends only on the facts inside the window, and the rest of the state can change without changing the response. This is what "depends on `State` only through some finite part" means, and it sets the theorem's scope: a `read` that inspects the whole infinite state at once has no window, and the theorem does not apply to it. Deployed reads have windows, because a response is finite and is computed in finite time from finitely many facts.

For each `r`, fix one window that is as small as possible: a window no proper subset of which is still a window. Such a minimal window exists inside any window, because windows are finite. Several minimal windows may exist; pick any one, because the construction below works for every choice.

Now the construction. One wrinkle must be handled in the open: the output may depend on `r` beyond the selection (the same data renders differently for a different `Accept-Language`) and `arrange` is forbidden by S1 from seeing `r`. The repair uses the derivation's own move: a request is a finite named structure, so by B-1 it encodes as facts. Define:

```
select(r, S)  =  (S ∩ K(r)) ∪ enc(r)
arrange(D)    =  canon(read(dec(D)))     dec: recover (r, S∩K(r)) from D
present       =  the rendering of a canonical tree as Doc
```

`select` is a function of `(r, S)`, as typed. `dec` is well-defined because `enc` is injective and its facts are disjoint from `K(r)` (mint them under a reserved authority, which costs nothing). `arrange` is a function of `D` alone: everything it needs (the window's facts and the request's) arrived in its argument, so S1 holds by construction rather than by discipline. `canon` is overloaded here, deliberately and in the open: Chapter 6 typed it on states, but its argument above is `read`'s output, a `Doc`. At this type `canon` means the document's canonical tree, and `present` is its inverse, rendering the tree back unchanged. `present` sees a tree, never the data. Composing:

`present(arrange(select(r, S))) = read(r, S ∩ K(r)) = read(r, S)` by finite dependence. ■

What this proof does and does not give: it gives S1 and the shape: every windowed `read` has the three-stage form with no side channels. The shape alone is cheap: here `arrange` carries the whole computation and `present` only inverts `canon`. Three functions become three concerns under S2–S4, and not before. S2 through S4 are claims about *languages and addresses*, and no analysis argument can conjure those. They are exactly what the synthesis theorem supplies (B.8). Analysis and synthesis are halves of one proof, and the halves meet in the middle, as promised.

B.6 B.6 Independent evolution (Prop. 4.5)

Prop. 4.5 (restated). In a proper factorization, the document's evolution decomposes into four independent timelines, one per component; in the trivial factorization, one timeline carries every change.

An application over time is a quadruple of trajectories `(S(τ), q_τ, x_τ, s_τ)` (`x` is Chapter 4's arrange term `t`, renamed to stay clear of the time index `τ`) with `doc(r, τ) = ⟦s_τ⟧(⟦x_τ⟧(⟦q_τ⟧(r, S(τ))))`. Write the three stage values as `v₁(r, τ) = ⟦q_τ⟧(r, S(τ))`, `v₂ = ⟦x_τ⟧(v₁)`, `v₃ = ⟦s_τ⟧(v₂)`.

Dependency triangle. By S1 each factor is a function of its displayed arguments only, so the dependency matrix of the stage values on the four components is triangular: `v₁` depends on `{S, q}`; `v₂` on `{S, q, x}`; `v₃` on `{S, q, x, s}`. Substituting `s_τ → s′` leaves `v₁` and `v₂` identical: S2 guarantees the substitution cannot reach into another language's meaning, S3 that the result is still an application. The same argument, one row up, for `x` and for `q`. So a change to any one component changes the document while every stage value upstream of that component is untouched: four timelines, advancing independently. *Effectiveness* means that each timeline can actually move the document; it is witnessed rather than proved. The witnesses are a theme that inverts colors, a layout that reverses order, a query that widens a window, and a write (delta normal form) that adds a fact inside the window. One witness each is all "independent" needs.

The fused half. In the trivial factorization there is one component; its dependency matrix is one full row; any change is a change to it. One timeline, by counting.

Corollary, cache granularity. Under S4 each `vᵢ` is a resource with a URI, hence with its own validator (RFC 9110 §8.8). By the triangle, `vᵢ`'s validator changes exactly when a component in its row changes: invalidation sets are the rows. Fused: the only resource is `v₃`, its row is everything, and every change invalidates the one entry there is. ■

B.7 B.7 The homomorphism (Prop. 8.1)

Prop. 8.1 (restated). A translation `φ` (facts to triples, states to graphs, bindings to solution mappings) carries match, join, union, and project to SPARQL's evaluation, clause for clause; on ground states it is a bijection.

Both sides first, then the map, then the commuting, clause by clause.

The derived side. Chapter 5's algebra over $\mathcal{P}(\texttt{Fact})$: a *pattern* $\texttt{P}$ is a finite set of triples over $\texttt{I} \cup \texttt{V} \cup \texttt{Var}$; its evaluation is $\texttt{match(P)(S)} = \{ \beta : \texttt{Var(P)} \to \texttt{I} \cup \texttt{V} \mid \beta(\texttt{P}) \subseteq \texttt{S} \}$; $\texttt{join}(\Omega_1, \Omega_2) = \{ \beta_1 \cup \beta_2 \mid \beta_1 \in \Omega_1, \beta_2 \in \Omega_2, \beta_1, \beta_2 \text{ agree where both defined} \}$; $\texttt{union}(\Omega_1, \Omega_2) = \Omega_1 \cup \Omega_2$; $\texttt{project}(\Omega, \texttt{W}) = \{ \beta|\_\texttt{W} \mid \beta \in \Omega \}$.

The deployed side. SPARQL 1.1 Query §18 defines, denotationally: basic graph pattern evaluation over a graph $\texttt{G}$ as the solution mappings μ with $\mu(\texttt{BGP}) \subseteq \texttt{G}$ (§18.3, §18.5); `Join` as the compatible merge of solution mappings; `Union` as their set union; `Project` as restriction to the projection variables. The four defining clauses are, symbol for symbol, the four clauses above.

The map. φ sends a fact $\texttt{(s, p, o)}$ to the RDF triple with $\texttt{s, p}$ as IRIs and $\texttt{o}$ as IRI or literal. It sends a state to the graph of its facts' images, and a binding to the solution mapping composed with φ. On ground states φ is a bijection onto ground graphs.

The commuting. By induction on the structure of the selection term. Base: $\varphi(\texttt{match(P)(S)}) = \texttt{eval(BGP\_}\{\varphi(\texttt{P})\}, \varphi(\texttt{S}))$ because $\beta(\texttt{P}) \subseteq \texttt{S} \iff (\varphi \circ \beta)(\varphi(\texttt{P})) \subseteq \varphi(\texttt{S})$, φ bijective on ground material, applied pointwise. Inductive cases: compatibility of bindings is preserved and reflected by φ (it is injective on values), so the `join` clauses coincide; `union` and `project` are set union and restriction on both sides, and φ commutes with both by construction. Four clauses, four checks, no remainder. ■

Three boundaries, stated rather than buried. First, the correspondence is proved on the ground fragment; blank nodes in data re-enter through the bill for anonymity (Prop. 9.1). SPARQL's default regime (matching blank nodes in the queried graph as constants) is skolemization, and the bill already covers it. Second, the derived algebra is the monotone core, and the correspondence covers exactly its image: the `AND`/`UNION`/`SELECT` fragment under set semantics (`DISTINCT`). SPARQL's non-monotone extensions (`OPTIONAL`, `MINUS`) and its default multiset semantics are conveniences beyond the derived minimum, and the theorem claims nothing about them. Third, $\texttt{V}$ is instantiated here as RDF's literal *terms*, lexical form with datatype, identity at the character level (RDF 1.1 Concepts §3.3), not as the values they denote: `"1"^^xsd:integer` and `"01"^^xsd:integer` are distinct terms denoting one value, and the bijection is term-level. The distinction is RDF's own, and renamings never enter $\texttt{V}$'s elements, so the datatype IRIs riding inside literals stay opaque. The fragment is not a retreat: it is what Chapter 5 forced, found verbatim in the standard.

B.8 B.8 The synthesis theorem, with genericity exact (Thm. 8.2)

Thm. 8.2 (restated). The stack realizes the proper factorizations whose `select` is a term of the derived algebra and whose `arrange` is generic (invariant under URI renaming) and only those. The select-side condition states the theorem's scope. S2 requires that `select` be written in some language with closed semantics; it does not require that language to be the derived algebra. A `select` beyond the derived algebra (a recursive one, say) is therefore outside the theorem's scope: Chapter 5 never derived a requirement that the stack express it.

Genericity, defined. Fix the reserved vocabulary $V_0 \subset I$, the form-level names the once-for-all conventions license (`kind`, `rel`, the role scheme). A *renaming* is a bijection $\rho : I \to I$ fixing V_0 pointwise. Renamings act on states, trees, and documents by rewriting embedded names, with one clause Chapter 6′s law forces: on a canonical tree the action is rewrite-then-recanonicalize, $\rho \cdot \mathrm{canon}(S) = \mathrm{canon}(\rho S)$, because `canon`'s sort is an accident of spelling and carries no meaning to preserve. Without the clause even the identity transform would fail the equation below, tripped by block order alone. A transform $T : \mathrm{Tree} \to \mathrm{Tree}$ is **generic** iff for every renaming ρ and state S: $T(\mathrm{canon}(\rho S)) = \rho \cdot T(\mathrm{canon}(S))$, the transform commutes with renaming. (The notion has a family history worth citing exactly. Genericity as invariance under domain permutations is Chandra–Harel (1980). Abiteboul and Vianu transposed it to a formal web model in 1997, and Fletcher et al. (2009) restated it for search queries. Deployed RDF practice, per Hogan's canonicalization work (2017), renames only blank nodes and holds IRIs rigid. The definition here is the family's missing member, IRI-renaming invariance, imposed on *transforms*.)

The free-theorem consequence, in Wadler's sense, "cannot hardcode identifiers," made exact. Say T *treats* u *specially* ($u \notin V_0$) if there is a state S and a fresh u' such that replacing u by u' in S does not change the output by exactly the action of $(u\ u')$. If T is generic, T treats no name specially: apply the transposition $\rho = (u\ u')$, which fixes V_0, and commuting forces the output to change by exactly that action. Contrapositive: a transform that singles out a particular name outside the reserved vocabulary — one whose behavior depends on the text of that URI rather than treating it as an opaque token — is not generic. This gives data-drivenness a precise test.

Relative genericity, the override, bounded. Deployed arrangements are rarely generic in the strict sense, and should not be: a term that renders persons as cards must name the person class, and naming is special treatment by the definition just given. The repair is not to relax the definition but to index it. For $W \subseteq I$, say T is **generic relative to** W iff T commutes with every renaming that fixes W pointwise as well; strict genericity is the case $W = \varnothing$. The relativization is the family's own standard allowance (Chandra–Harel's queries are generic up to a finite set of constants) transposed like the rest. The free theorem relativizes verbatim: a transform generic relative to W treats no name outside W specially. Its special treatment is confined to a declared set, which can be checked from the term's text alone. The names it spells must lie in $V_0 \cup W$. The names it does not spell it may use only opaquely, matched by equality and copied, never inspected as strings (AWWW §2.5′s opacity, now a syntactic discipline rather than a norm). Vocabulary-specific behavior is explicit when the vocabulary it depends on is declared.

The deployed shape of relative genericity is layering. A *base* term renders the canonical serialization and names no vocabulary, generic relative to $\varnothing$, and total, because `canon` refuses no state. Vocabulary-specific terms are *overrides*, layered onto the base by the transformation language's own import mechanism. Precedence is part of XSLT's closed semantics, so S2 is undisturbed, and adding or removing an override is S3′s substitution, exercised in place. One further condition, and it does real work: an override must *refine* coverage, never restrict it. For every state,

the layered term renders every entity the base renders, differing only where descriptions meet `W`. Under refinement, the failure mode of an unknown name is the base rendering rather than no rendering: every state renders; declared vocabulary renders better. (An override that seizes the root and renders only what it recognizes passes the footprint check and fails this one; both clauses are load-bearing.)

The relativization does not readmit hidden domain knowledge; it settles where such knowledge can live. There are exactly two places, and both are visible. The first is the transform itself. The names it treats specially must be spelled in its declared set `W`, and relative genericity guarantees that it treats every name outside `W` the same. So reading the transform's text shows exactly which vocabulary it depends on. The second is the data. An ontology is facts like any other facts, so the vocabulary can travel in the state, and Chapter 18 puts it there. The transform then reads the vocabulary from its input, and a renaming moves the ontology facts and the data facts together, so such a transform stays fully generic (`W = ∅`). A transform that depends on a name found in neither place — not spelled in its text, not present in its input — is not generic, and the free theorem detects it.

Synthesis. Let `(select, arrange, present)` be any proper factorization whose `select` is a term of the derived algebra and whose `arrange = T ∘ canon` with `T` computable and generic, strictly, or relative to a declared `W`; the construction is indifferent. Realize the three factors in the deployed stack. The selection is a term of the derived algebra, hence by the homomorphism (B.7) a SPARQL term evaluating identically. `canon` exists and is deterministic (Prop. 6.1, RDFC-1.0 for the unnamed). `T` is a computable tree-to-tree function and XSLT is computationally complete on trees, so a term `t` with `⟦t⟧ = T` exists. Genericity is preserved by writing `t` with no URI literals outside V_0, names otherwise held opaque. It relativizes intact: base plus declared overrides realizes the class generic relative to `W`, with the relativized free-theorem clause as the check that nothing was smuggled. `present` is a stylesheet by S2′s own requirement. S4 holds because in the deployed stack every stage value is a resource: the graph, the query result, the document each dereference (Graph Store Protocol; SPARQL protocol; HTTP). So the stack realizes the factorization, and realizes only proper ones: a non-generic `arrange` fails the definition just given, which is the "excluding smuggling" caveat of Chapter 8, now a clause rather than a caution.

Together with the analysis theorem (B.5): every windowed `read` has the form, its select window-shaped (a union of ground matches, inside the derived algebra) and the stack fills the form. This section is where the halves meet. ■

B.9 B.9 Federation closure (18.1)

The claim. The union of two dataspace states is again dataspace-shaped: one graph per document, every document under exactly one origin, attribution intact, so federation needs no machinery beyond (5.1).

The proof. RFC 6454 computes an origin from every URI; distinct origins are therefore disjoint regions of $\mathtt{I}$: $\mathtt{o} \neq \mathtt{o'} \implies \mathtt{I|o} \cap \mathtt{I|o'} = \varnothing$. A dataspace's graph names are document URIs under its own origin (18.1), so two dataspaces' graph families have disjoint name sets and union as families. No graph name is claimed twice, every document is still served by exactly one party, and the fourth position still carries who. On the facts, the union is (5.1) over the retyped atoms of Prop. 9.2 (merge is still set union) so federation inherits totality, order-freedom, and idempotence unchanged. Facts join where names are shared (R3); the proposition makes no stronger claim. ■

One boundary, stated rather than buried: the closure is of *states*. A federation is not itself a dataspace (it has many origins, no single ontology) and (18.1) claims no such thing. What the parties hold before aligning and what alignment yields is Chapter 18′s cost-of-alignment section, not this lemma.

C. The mechanization

Appendix B is prose, and prose proofs can hide a step. This appendix reports the machine check: a self-contained Lean 4 development in the book's repository under `proofs/`, using no mathematics library, so every ingredient of the argument appears explicitly. One command rechecks everything, on a pinned toolchain. No proof contains a placeholder, and the checker reports each theorem's axioms.

Checked — every result on the formalizable list:

result	Lean theorem
B.1 representation, finite and full (the union law)	`representation_finite`, `representation_full`
B.2 arity core (Prop. 5.2)	`arity_minimal_is_three`
B.2's trap: the pairing reduction, located	`gadget_escape_closed`
B.3 uniqueness, assembled and exact (Thm. 5.4)	`uniqueness`, `uniqueness_iso`
B.4 independence, four of the five conditions	`atomistic_independent`, `order_freedom_independent`, `idempotence_independent`, `accumulation_independent`
B.5 analysis theorem, the shape half (Prop. 4.4)	`analysis_shape`, `minimal_window_exists`
B.6 independent evolution (Prop. 4.5)	`present_substitution`, `timelines_independent`
B.7 homomorphism, against a model of §18 (Prop. 8.1)	`homomorphism`, `seval_monotone`
B.8 genericity core: the free theorem (Thm. 8.2)	`transposition`, `treats_no_name_specially`, `not_generic_of_treatsSpecially`, `canon_renaming_commutes`
Prop. 7.1 delta normal form	`delta_normal_form`
B.9 federation closure	`federation_closure`
B.5 ↔ B.7, the halves meeting	`halves_meet`
Prop. 6.1 `canon` exists, ground states	`canon_exists`
Props. 7.2–7.4 forms, one algebra, five moves	`submit`, `find_then_denote`, `five_moves`
Prop. 9.1 the bill for anonymity, algebraic core	`bill_for_anonymity`
Prop. 9.2 erasure, and the quad repair	`attribution_erased`, `attrOf_merge`, `attribution_closes`
Prop. 20.1 nothing else to vary	`nothing_else_to_vary`

The embedding half of B.1 and the arity core depend on no axioms at all; the rest use only Lean's three standard ones.

One result in the development checks nothing above: coordination-free convergence — parties that received the same updates agree, in any order, with any duplication — follows from the merge laws alone, which turns Chapter 5′s corroboration from the replication literature into a theorem about this model.

The check sharpened two things the prose states only in passing. B-2d splits into two named hypotheses: one half of the atom-map homomorphism is derivable from the merge laws alone, and only the "no emergence" half is genuinely axiomatic. And that axiom is necessary: the model `(ℕ, max)` satisfies every merge law yet violates atomicity, so B.1 must assume what it assumes. The formalization answers "did you assume what you needed?" with: yes, necessarily, and here is the witness.

The ceiling is a category boundary, not a shortfall. Lean checks the conditional skeleton: given the requirements as axioms, everything downstream follows. What it can never do is discharge the axioms, because their justification is the Transposition Thesis, and that is unprovable the way the Church–Turing thesis is unprovable: it equates an informal subject (what the web enforces) with a formal object (the merge laws). No formalism can certify its own adequacy to an informal subject. The thesis is corroborated and consequence-tested, never proved.

Nothing formalizable remains. Four results are checked to a stated scope: B.7′s correspondence is to a model of the query algebra's specification, never to an engine; B.8′s is the free-theorem core; `canon` is checked on named entities; and the bill for anonymity is checked as algebra. What those four defer — the transformation language's completeness, the canonical labeling of unnamed entities, the cost of redundancy elimination — stays cited, because each is someone else's theorem. The fifth independence condition, totality, is enforced by the formalization's own typing, so dropping it leaves nothing to state. Parts I, IV, and VI are not on the list, because scores and history are not mathematics.

D. References

This list is the spec concordance. The axioms below are the book's external dependencies, and deliberately its only ones. Four lists follow, kept separate per the discipline of Appendix A: the witnesses, the candidates, the prior art, and the works the audit examines.

Axioms — definitions used as premises:

definition	source	first used
`I` — URI syntax; decentralized minting via the authority component	RFC 3986	Ch 1; B.2
`Req`, `Resp` — the message form; the safe/unsafe method split	RFC 9110 §6, §9	Def. 1.1
representations reflect resource state over time	RFC 9110 §3.2	Ch 4 (τ)
validators `Last-Modified`, `ETag`	RFC 9110 §8.8	Prop. 4.5
the caching calculus	RFC 9111	Prop. 4.5, corollary
origins — `I` partitioned into parties' regions	RFC 6454	(18.1); B.9
triple, graph, merge	RDF 1.1 Concepts (2014)	Ch 8
blank nodes as existentials	RDF 1.1 Semantics	Prop. 9.1
datasets and named graphs	RDF 1.1; TriG (2014)	Prop. 9.2
the selection algebra, denotationally	SPARQL 1.1 Query §18	Ch 8; Prop. 8.1
query via `GET`, so a result is a resource	SPARQL 1.1 Protocol	Ch 15; Ch 16
the delta on the wire	SPARQL 1.1 Update	Ch 8 (Prop. 7.1′s reveal)
documents as named graphs, read-write	SPARQL 1.1 Graph Store HTTP Protocol	Ch 8; Ch 18
canonical labeling of unnamed entities	RDFC-1.0 (2024)	Prop. 6.1; Prop. 9.1
tree transformation	XSLT (1999; 3.0, 2017)	Ch 8
presentation	CSS (1996)	Ch 8
forms as the write instrument	HTML: forms	Prop. 7.2

Currency — checked July 2026: RFC 3986 remains Internet Standard 66 — updated, never obsoleted. The update is BCP 190 (RFC 8820): guidance on URI *ownership*, with no change to syntax. That is B.2′s minting doctrine, in BCP form. RFC 9110 and 9111 are the current

HTTP standards. RFC 6454 stands unrevised since 2011; the HTML Standard restates the same scheme–host–port tuple for browsers. RDF 1.2 (Candidate Recommendation, April 2026) preserves every definition cited above: data conforming to 1.1 remains conforming. Its headline addition, the triple term, is the annotation syntax Chapter 9 contrasts and scores. And RDF 1.2 keeps named graphs — the fourth position, Chapter 9′s structural prediction, survives another revision. SPARQL is cited at 1.1 throughout, the current Recommendation; 1.2 remains a Working Draft.

Witnesses — norms and independent results corroborating, never premises:

- *Architecture of the World Wide Web, Volume One*, W3C Recommendation, 2004 (TAG):
  - §2.5 URI opacity — Thm. 8.2′s genericity.
  - §3.5 available representations — S4, minus the intermediates, and (18.1)‘s minting obligation.
  - §4.3 separation of content, presentation, interaction — Def. 4.3.
  - §4.4 link identification, Web-wide linking, hypertext links — Ch 10′s scoring.
  - §5.1 orthogonality — S2/S3.
- *The Rule of Least Power*, TAG finding, 2006 — prefer the least powerful language that suffices: the norm behind Ch 12′s scoring.
- R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, dissertation, 2000 — Ch 4′s payoff: the positioning of this book as the second half of a derivation whose first half Fielding wrote, and the uniform interface’s four clauses (§5.1.5) typed there. Also the discarded hypermedia constraint in Ch 10, and the property list of Ch 11.
- M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, *Conflict-free Replicated Data Types*, 2011 — the independent derivation of the merge laws from replication pressure (Ch 5′s corroboration; B.4).
- httpRange-14, W3C TAG issue, resolved 2005 — the name/address distinction the model types apart (R3 vs. S4); Ch 19′s encoding choices.
- *Cool URIs for the Semantic Web*, W3C Interest Group Note, 2008 — the deployed encodings (fragment, `303`) of that distinction.
- Pappus of Alexandria, *Collection*, Book VII — the classical statement of the twin method of analysis and synthesis; Chapter 2′s name for the book’s shape.
- I. Newton, *Opticks*, Query 31 — “the Investigation of difficult Things by the Method of Analysis ought ever to precede the Method of Composition”; Chapter 2.
- T. Berners-Lee, *Information Management: A Proposal* (CERN, 1989) — the origin memo, and Mike Sendall’s cover note “vague but exciting”; Chapter 1′s epigraph, and the browser-editor bootstrap of Chapter 19.
- V. Bush, *As We May Think* (The Atlantic, 1945); T. Nelson, *Computer Lib / Dream Machines* (1974) — association over hierarchy, the graph refusing the tree, stated before the web; Chapter 6.

- T. Berners-Lee, J. Hendler, O. Lassila, *The Semantic Web* (Scientific American, May 2001) — the agent-over-machine-readable-data scenario Chapter 23 derives; written as fiction, now falsifiable.
- R. Hickey, *Cognicast* Episode 103: "Clojure spec with Rich Hickey" (June 2016) — the creator of Clojure and Datomic naming RDF as prior art for properties defined independent of aggregates; Chapter 8's epigraph.
- J. E. Labra Gayo, E. Prud'hommeaux, I. Boneva, D. Kontokostas, *Validating RDF Data* (2017), foreword by D. Brickley and L. Miller — Chapter 9's epigraph.
- D. McComb, *Software Wasteland* (Technics, 2018) and *The Data-Centric Revolution* (Technics, 2019) — the data-centric case (Semantic Arts) for data over application code; Chapter 20's corollary reached from enterprise waste rather than derivation.
- P. Nadkarni, L. Marenco, R. Chen, E. Skoufos, G. Shepherd, P. Miller, *Organization of Heterogeneous Scientific Data Using the EAV/CR Representation* (JAMIA, 1999) — the triple shape rediscovered inside one database, where attributes cannot be fixed in advance; Chapter 13's EAV witness.
- B. Karwin, *SQL Antipatterns* (Pragmatic Bookshelf, 2010) — the practitioner verdict against EAV inside a silo; Chapter 13's counter-witness.
- J. Somers, *The Coming Software Apocalypse* (The Atlantic, September 2017) — code grown past comprehension in safety-critical systems, and the remedies that move engineers above it (model-based design, Lamport's TLA+); Chapter 20's liability, reported from the field.
- R. Larson, E. Sandhaus, *NYT to Release Thesaurus and Enter Linked Data Cloud* (NYT Open, June 2009) and *First 5,000 Tags Released to the Linked Data Cloud* (October 2009) — a century-old newspaper thesaurus given HTTP URIs and RDF under a Creative Commons license, mapped by hand to DBpedia and Freebase; the host stopped resolving (checked September 2026); Chapter 22's first mover on the vocabulary side.
- J. Rayfield, *BBC World Cup 2010 dynamic semantic publishing* (BBC Internet Blog, July 2010) and *Sports Refresh: Dynamic Semantic Publishing* (April 2012) — 700-plus World Cup pages, then ten thousand Olympic pages, generated from an RDF triple store; Chapter 22's first mover on the publishing side.
- M. Jusevičius, A. Smirnovas, J. Šėporaitis, *Graphity – A Generic Linked Data Platform* (position paper, W3C Workshop on Linked Enterprise Data Patterns, December 2011) — the comics site Helt Normalt on RDF, SPARQL and XSLT, the codebase an order of magnitude smaller than the relational system it replaced, ontologies reused down to a zodiac vocabulary; Chapter 22's first mover on the application side (the author's, per Chapter 19).
- A. Singhal, *Introducing the Knowledge Graph: things, not strings* (Google, May 2012) — the announcement that made *knowledge graph* the industry's name for graph-shaped state; Chapter 22's dating of the term.
- J. Walker, *Is Linked Data the Future of Data Integration in the Enterprise?* (NXP Smarter World Blog, January 2013) — product data scattered across systems, converted to RDF,

dereferenceable per product; "the Linked Data is the API"; Chapter 22's first mover on the integration side.

- M. Stonebraker, I. F. Ilyas, *Data Integration: The Current Status and the Way Forward* (IEEE Data Eng. Bulletin, 2018) — silo counts at enterprise scale: GE's ~75 procurement systems, Merck's ~4,000 databases; Chapter 22's head of the curve, measured.
- Palantir, *Ontology* (platform page; on Foundry's marketing since 2018) and the S-1 registration statement (SEC, August 2020) — "a data model that reflects the real world," the word filed with the regulator; not built on RDF; Chapter 22's vocabulary crossover.
- Google, *A reintroduction to our Knowledge Graph and knowledge panels* (May 2020) — "over 500 billion facts about five billion entities"; proprietary, no public endpoint or dump; Chapter 22's closed giant.
- Gartner, *Top 10 Data and Analytics Technology Trends for 2021* (press release, March 2021) — "by 2025, graph technologies will be used in 80% of data and analytics innovations, up from 10% in 2021"; Chapter 22's wave, dated by its analysts; the figure spans all graph models, not RDF alone.
- J. F. Sequeda, D. Allemang, B. Jacob, *A Benchmark to Understand the Role of Knowledge Graphs on Large Language Model's Accuracy for Question Answering on Enterprise SQL Databases* (2023) — GPT-4 at 16.7% over enterprise SQL, 54.2% over the same data as an RDF graph; Chapter 22's grounding motive, measured.
- Microsoft, Fabric IQ ontology (preview, announced November 2025) — "an ontology is a shared, machine-understandable vocabulary of your business"; Power BI's datasets became *semantic models* in November 2023; neither on RDF; Chapter 22's vocabulary crossover.
- The UniProt Consortium, sparql.uniprot.org (release 2026_02, June 2026) — 232.5 billion triples behind a free public SPARQL endpoint, RDF distribution since 2008; the largest publicly queryable knowledge graph; Chapter 22's open giant.
- Wikidata, statistics (2026) — ~123 million items, ~18 billion triples across the query service and its scholarly split (May 2025); where part of Freebase settled; Chapter 22's open giant.
- The Linked Open Data cloud (version of 15 June 2026; diagram CC BY) — 1,360 interlinked open datasets; Chapter 22's map of the open graphs.
- Web Data Commons, Structured Data Extraction Report (October 2024 Common Crawl corpus, released January 2025) — structured data on 16.5 million of 37.4 million pay-level domains, 74 billion quads; JSON-LD on 11.6 million domains against RDFa's 0.5 million; schema.org claims over 45 million domains and 450 billion objects; Chapter 22's annotated web, measured independently.
- D. Huynh, *Freebase Parallax: A new way to browse and explore data* (Metaweb, 2008) — set-based navigation over graph data: a set carried to the set it relates to. Google acquired Metaweb in 2010, and Freebase was a seed of the Google Knowledge Graph. Chapter 24's set-to-set capability, demonstrated.

- D. Siegel, *Pull: The Power of the Semantic Web to Transform Your Business* (Portfolio, 2009) — the *personal data locker* (a life held as one owner-controlled graph) and *intentcasting* (Searls's coinage, below); Chapter 24's personal dataspace, reached from the demand side. Video: *Personal Data Locker Vision*.
- D. Searls, *The Intention Economy: When Customers Take Charge* (Harvard Business Review Press, 2012; the term *intention economy* coined 2006) — customers broadcasting qualified intent for vendors to answer (*intentcasting*); Chapter 24's stating-a-need capability, from the demand side. Talk: *The Intention Economy* book talk, 2012.
- Apple, *HyperCard* (1987) — a running application reshaped in place, its stacks editable documents, with no rebuild and no redeploy; Chapter 24's live-morphability lineage, lacking only a cross-party substrate.
- D. Shea, *CSS Zen Garden* (2003) — one unchanged HTML document redressed by hundreds of stylesheets; Chapter 24's presentation independence (S1), demonstrated at the presentation layer alone.
- The Semantic Web stack (the "layer cake"), W3C, early 2000s — *proof* and *trust* sketched as its top layers and left unbuilt; Chapter 24's provenance capability (R4), the layer the deployed web skipped.
- T. Berners-Lee, *Linked Data* (W3C Design Issues, 2006) — four rules, the first of which is "Use URIs as names for things"; asked of publishers, and Chapter 15's epigraph because the property graph does not.
- T. Berners-Lee, *Cool URIs don't change* (W3C, 1998) — persistence of names asked of the web and mostly declined; Chapter 24's data-outlives-the-application (R3).
- The mashup era — P. Rademacher's *HousingMaps* (Google Maps × Craigslist, 2005) and *ProgrammableWeb* (2005) — combination without permission, until the APIs metered and re-siloed; Chapter 24's applications-nobody-planned (R2/S3).

Candidates — specified, not standardized; Part V's seams:

- WebID — W3C Incubator, 2005–; identity as a dereferenceable URI. The identity seam (Ch 19).
- WebAccessControl — the `acl` ontology, grown on the W3C wiki; adopted by Solid. The access-control seam (Ch 19).
- RDF/POST — community spec, AtomGraph, building on Sergei Egorov's original draft. The form-native write seam (Ch 9; Ch 19).
- SaxonJS 3 with IXSL — XSLT 3.0 evaluated in the browser; the interactive extension binds events to template rules. The arrange seam, filled client-side (Ch 9; Ch 18).

Prior art — the formal neighbors of Appendix B, cited so the boundaries can be checked:

- The RDF foundations school — closest formal treatment; stipulates triples, derives nothing about their necessity: C. Gutierrez, C. Hurtado, A. O. Mendelzon, J. Pérez, *Foundations of Semantic Web Databases* (PODS 2004; JCSS 77(3), 2011); M. Arenas, C. Gutierrez, J. Pérez,

Foundations of RDF Databases (Reasoning Web 2009); S. Muñoz, J. Pérez, C. Gutierrez, *Minimal Deductive Systems for RDF* (ESWC 2007).

- E. L. Robertson, *Triadic Relations: An Algebra for the Semantic Web* (SWDB 2004, LNCS 3372) — asserts triples-minimal for RDF as motivation, underived; B.2 supplies the derivation.
- The Peirce line — the arity theorem's owners: R. W. Burch, *A Peircean Reduction Thesis* (Texas Tech UP, 1991); H. Herzberger, *Peirce's Remarkable Theorem* (1981); F. Dau, J. Hereth Correia, *Two Instances of Peirce's Reduction Thesis* (ICFCA 2006); J. Hereth Correia, R. Pöschel, *The Teridentity and Peircean Algebraic Logic* (ICCS 2006; Semiotica 186, 2011); S. Koshkin, *Is Peirce's reduction thesis gerrymandered?* (TCSPS 58(4), 2022) and the relational-database formalization (Logic J. IGPL, 2024/25 — `ternarity(R) = n − 2`).
- The dyadic-reduction results that B.2's scope note answers: L. Löwenheim, *Über Möglichkeiten im Relativkalkül* (Math. Annalen 76, 1915); W. V. O. Quine, *Reduction to a Dyadic Predicate* (JSL 19(3), 1954).
- The coordination-free line — programs characterized, data model left open: J. M. Hellerstein, P. Alvaro, *Keeping CALM: When Distributed Consistency Is Easy* (CACM 63(9), 2020); T. J. Ameloot, F. Neven, J. Van den Bussche, *Relational Transducers for Declarative Networking* (JACM 60(2), 2013 — the CALM proof); N. Conway et al., *Logic and Lattices for Distributed Programming* (Bloom^L, SoCC 2012 — B.4's deployed witness); S. Laddad et al., *Keep CALM and CRDT On* (VLDB 16, 2023).
- CRDTs meet RDF, as engineering: L.-D. Ibáñez, H. Skaf-Molli, P. Molli, O. Corby, *Live Linked Data: Synchronising Semantic Stores with Commutative Replicated Data Types* (IJMSO 8(2), 2013 — SU-Set; the `(triple, id)` tags it needs for deletion are the erasure argument surfacing as an engineering symptom).
- The genericity family: A. Chandra, D. Harel, *Computable Queries for Relational Data Bases* (JCSS 21(2), 1980); S. Abiteboul, V. Vianu, *Queries and Computation on the Web* (ICDT 1997; TCS, 2000); G. Fletcher, J. Van den Bussche, D. Van Gucht, S. Vansummeren, *Towards a Theory of Search Queries* (ICDT 2009); A. Hogan, *Canonical Forms for Isomorphic and Equivalent RDF Graphs* (TWEB 11(4), 2017 — IRIs rigid, blank nodes renameable: the practice that B.8's definition departs from).
- M. Franklin, A. Halevy, D. Maier, *From Databases to Dataspaces* (SIGMOD Record 34(4), 2005) — the word's database-literature sense, pay-as-you-go integration; disambiguated from Chapter 18's web-native sense.
- Formal separation, one seam, 2004: T. Parr, *Enforcing Strict Model-View Separation in Template Engines* (WWW 2004) — definitions and theorems for model-view separation ("there was no formal definition of separation"). The paper excludes XSLT from its scope and characterizes template power via the Chomsky hierarchy. It is the priority citation for formalized separation, and the single-seam treatment the factorization generalizes.

- The backward direction — view update in databases, lenses in programming languages: F. Bancilhon, N. Spyratos, *Update Semantics of Relational Views* (TODS 6(4), 1981); J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, *Combinators for Bidirectional Tree Transformations* (POPL 2005; TOPLAS 29(3), 2007). Both compute an inverse for arbitrary views; Prop. 7.2 declares one, because the form carries the pattern.

Audited — works the book examines to score, chiefly in Part IV:

- XML 1.0 (1998); Namespaces in XML (1999); XPath 1.0 (1999); XSD (2001); `xml:id` (2005); XLink; XPointer; XQuery 1.0 and XPath 2.0 Formal Semantics (2007) — Ch 8's rarity remark, Ch 10.
- JSON — RFC 8259 / ECMA-404; JSON Pointer — RFC 6901 (2013); JSONPath — RFC 9535 (2024); JSON Schema (drafts) — Ch 10's tooling table.
- GraphQL — Ch 14; Ch 17's table.
- GQL (ISO/IEC 39075, 2024); SQL/PGQ (ISO/IEC 9075-16, 2023); openCypher (2015); Neo4j's Query API — Ch 15's column; the language is standardized and the protocol is not.
- Linked Data Platform 1.0 (W3C REC, 2015) — Ch 19's wrong-layer instance: containers as canned selections; subtract them and the Graph Store Protocol remains.
- *Should we remove XSLT from the web platform?*, WHATWG HTML issue, August 2025 — Ch 9's third mismatch, with the removal underway. The stated grounds are unmaintained implementations, which is Ch 9's maintenance-failure finding in the platform's own words.