CMS-0057-F · FHIR R4 · DA VINCI

From Regulation to FHIR API

CMS-0057-F Compliance for Legacy Payer Systems

Architecture decisions and engineering delivery for Prior Authorization, Patient Access, Provider Access, and Payer-to-Payer



v1.0 · 2026

Rong (Rex) Fan

From Regulation to FHIR API

CMS-0057-F Compliance for Legacy Payer Systems: Architecture Decisions and Engineering Delivery

Edition date: November 15, 2026 · Version 1.0.0

What this book covers: how a payer that runs on legacy core systems can meet the CMS-0057-F requirements through FHIR APIs. The book takes two viewpoints. On the architecture side, it covers regulatory scope and compliance measurement, the precedence of overlapping standards, FHIR version selection, the facade-versus-hybrid decision, exchange mechanisms, and trust and governance. On the engineering side, it uses the Prior Authorization (PA) API as a worked example to cover interface layering, the status-code contract, validation and mapping, safe writes into legacy systems, and conformance evidence.

Copyright and License

© 2026 Rong (Rex) Fan. Version 1.0.0, November 15, 2026. Official source: northpine.dev.

This book is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0). You are welcome to share the complete, unmodified book free of charge, provided you credit the author and the source (northpine.dev). You may not use it for commercial purposes or distribute modified versions. The sample code in this book is separately available under the MIT License and may be used freely.

Trademarks: HL7®, FHIR®, and the FHIR flame design are registered trademarks of Health Level Seven International, and their use does not constitute endorsement by HL7. Da Vinci, CARIN, Inferno, and other product names mentioned in this book belong to their respective owners and are used here for descriptive purposes only.

Disclaimer: This book is for educational and reference purposes only and does not constitute legal, compliance, or medical advice. Regulations and standards are authoritative only in their latest official publications. The author accepts no liability for any action taken on the basis of this book.

About the Author



Rong (Rex) Fan holds two master's degrees, in Lightning Science and Technology and in Computer Science and Technology. He is a healthcare interoperability engineer and architect, a .NET/C# developer, an author, and a certified Project Management Professional (PMP, certification number 3453332).

- **Engineering background:** .NET/C# since 2009, and software engineering in the United States since 2016. Legacy systems, data pipelines, and enterprise integration are his home ground.
- **Current focus:** since 2026, US healthcare interoperability — FHIR R4, US Core, SMART, Bulk Data, Da Vinci (PAS, CRD, DTR), X12, and payer API compliance under CMS-0057-F, the Interoperability and Prior Authorization Final Rule. The book's through-line (the Prior Authorization API, `$submit` / `$inquire`, the status-code contract, and the draft-then-release write pattern) distills public specifications and patterns common in hands-on work in this field.

- **Why this book:** there is no shortage of material on FHIR, the CMS rules, or architecture in isolation. There is very little practical, public material on how a regulation becomes measurable compliance, or on how to connect a legacy system to FHIR safely. This book offers a practitioner's view of both.

Find the author online: - GitHub: memoryfraction/HealthData-Interoperability-Csharp — an open-source FHIR/C# interoperability project - LinkedIn: https://www.linkedin.com/in/rexfan18/ - Google Scholar: https://scholar.google.com/citations?user=Zxn84ckAAAAJ&hl=en - Medium: https://medium.com/@rex.fan18

Disclaimer This book summarizes the author's own study and practice of CMS-0057-F. All content is drawn from general knowledge, public specifications (such as the CMS rule text and official FAQs, FHIR R4, and US Core), publicly available industry material, and the author's general learning. It contains no proprietary information or intellectual property of any organization.

A note on dates and versions Regulations, standard versions, and implementation guides change over time. Wherever a conclusion carries a version number or a date, check it against the current published version.

Contents

About the Author 4

Glossary 7

Executive Summary 9

Part I — Regulation and Architecture 13

1. The Big Picture: What Interoperability Means and Who You Conne... 14

2. The Regulatory Layer: CMS-0057-F 17

3. Precedence Among Overlapping Standards, and How Each Shows ... 25

4. Choosing a FHIR Version: R4 or R5 34

5. Choosing an Architecture Pattern: Facade, Hybrid, or FHIR-Native 41

6. An Architecture Decision Method: Size the Work First, Then Choo... 50

7. Trust and Governance: Who Can See What, and How the Network ... 59

Part II — Engineering Delivery 67

8. The Interface Layer: How to Layer $submit 68

9. The Status-Code Standard (An External Contract) 73

10. Validation and Fail-Fast 77

11. Mapping, Defaults, and Business Rules 79

12. Writing to Legacy Systems: Draft, Release, Retry, Quarantine 85

13. Triggering the Rules Engine 91

14. Conformance and Evidence 92

15. Delivery Governance: Scope, Definition of Done, and How to Rep... 97

Part III — Synthesis 102

16. Anti-Pattern Catalog 103

17. Architecture Review Questions 105

18. The End-to-End Path 107

19. Conclusions and Outlook 109

Stay in Touch 115

References 116

Glossary

CMS (Centers for Medicare & Medicaid Services) is the agency within the U.S. Department of Health and Human Services (HHS) that sets the payment and regulatory rules for Medicare, Medicaid, and the Federally-facilitated Exchanges (FFEs).[3]

CMS-0057-F (the Interoperability and Prior Authorization Final Rule, RIN 0938-AU87) was released by CMS on January 17, 2024, published in the *Federal Register* on February 8, 2024 (89 FR 8758–8988, FR Doc 2024-00895), and took effect on April 8, 2024.[1][2]

Scope. The rule applies to five types of "impacted payers": Medicare Advantage (MA) organizations, state Medicaid and CHIP fee-for-service (FFS) programs, Medicaid managed care plans, CHIP managed care entities, and Qualified Health Plan (QHP) issuers on the FFEs. It also adds an electronic prior authorization (ePA) measure for MIPS eligible clinicians and for hospitals and critical access hospitals (CAHs) in the Medicare Promoting Interoperability Program. **Commercial health plans are outside the rule's scope** (the only exception is QHPs on the FFEs); CMS encourages voluntary adoption across all lines of business, but it is not required.[3][4]

A common misconception. The industry shorthand "stand up all five FHIR APIs" overstates what this rule adds. The Patient Access API and the Provider Directory API were already required by the 2020 rule (CMS-9115-F).[7] What CMS-0057-F actually does is **add three APIs — Provider Access, Payer-to-Payer, and Prior Authorization — and require prior authorization information to be added to the Patient Access API**. Compliance dates come in two waves. API development and enhancement is generally due by **January 1, 2027** (for Medicaid and CHIP managed care, by the rating period beginning on or after that date; for QHP issuers, by the plan year beginning on or after that date). The

operational provisions — prior authorization decision timeframes of 72 hours for expedited and 7 calendar days for standard requests, and annual reporting of Patient Access API usage metrics to CMS — apply from **January 1, 2026**.[2][6]

Executive Summary

In one sentence: an interoperability and CMS-0057-F compliance program succeeds when five judgments are made correctly — **measure compliance against the rule text, choose the architecture by where the system of record lives, pick the exchange mechanism by time horizon, write to legacy systems with draft-then-release, and count work as done only when there is evidence.**

> Sources for the summary (all official, retrieved November 15, 2026): the full rule text in the *Federal Register*, February 8, 2024 (federalregister.gov, FR Doc 2024-00895); the CMS fact sheet "CMS Interoperability and Prior Authorization Final Rule CMS-0057-F"; the CMS interoperability FAQs (which confirm commercial plans are out of scope); and the CMS Electronic Prior Authorization and API/IG standards pages (compliance dates and standard versions).[1][2][4][5][6]

Questions this book answers - How do you produce a CMS-0057-F Gap Document? - What does the rule require, and how do you calculate a compliance percentage honestly? - When HIPAA, the rule, FHIR, implementation guides, and clinical guidelines all apply, what takes precedence, and how does each show up in the build? - FHIR R4 or R5 — which one, why, and when should you revisit the choice? - For a payer with legacy systems, the realistic options are facade and hybrid. What does each buy, and what does each cost? - How do you break the rule's requirements into "actions" and pick the right exchange mechanism for each? - How should authorization, consent, audit, identifiers, and versions be governed so a multi-party network keeps working? - How should a `$submit` endpoint be layered, and how do you define its status-code contract? - Where do validation, defaults, and business rules belong, and what happens when they fail? - How do you write safely into a legacy system and trigger

downstream rules? - How do you prove conformance, keep evidence, and define “done”? - How do you report progress, spot anti-patterns, and run an architecture review? - Conclusions and outlook - References

The book’s through-line

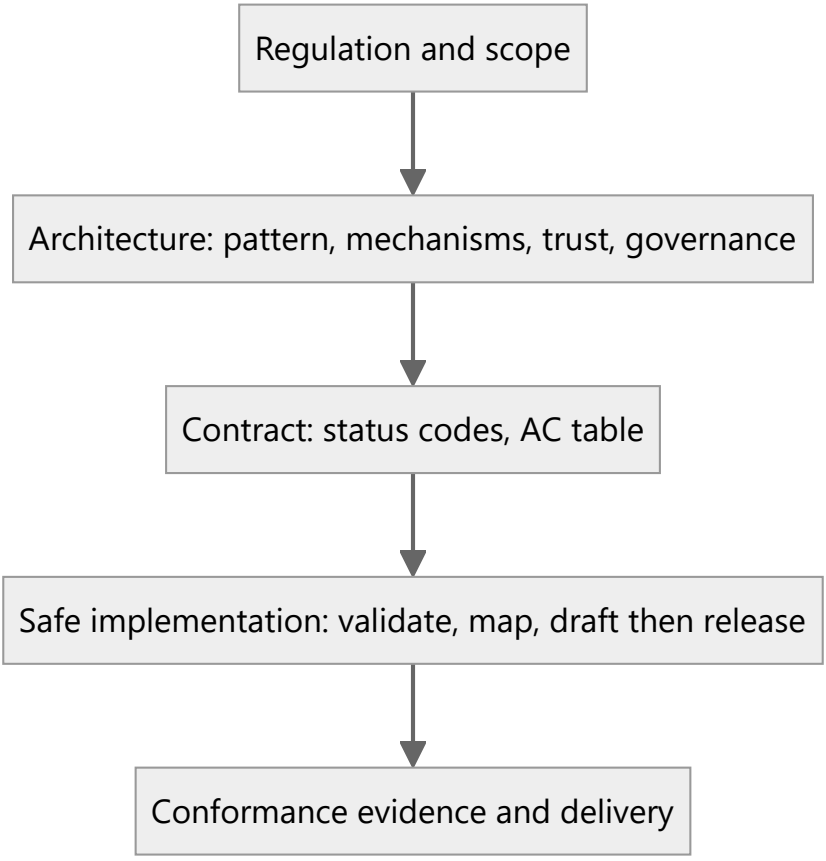

Figure 0.1: The book’s through-line (regulation → architecture → contract → implementation → evidence)

How to read this book. The book has three parts. Part I (Chapters 1–7) covers regulation and architecture; read it first. Part II (Chapters 8–15) covers engineering delivery; read it as needed. Part III (Chapters 16–19)

provides review tools and conclusions. To build a CMS-0057-F Gap Document, see Chapter 2 (especially Section 2.4) and Section 3.4. For an architecture proposal, see Chapters 5–7. For a design review, use Chapter 16 (anti-patterns) and Chapter 17 (review questions). Conclusions and outlook are in Chapter 19. Every chapter opens with a “Bottom line” box; read those boxes in sequence and you have the skeleton of the whole book.

How the book is organized. The book moves from the big picture to the details and back. The executive summary states the conclusions up front. Part I (Chapters 1–7) takes the architect’s view — regulation, standards, versions, architecture pattern, mechanisms, and trust and governance. Part II (Chapters 8–15) takes the engineer’s view — interface, status codes, validation, mapping, writes, rule triggering, verification, and delivery. Chapters 16–18 in Part III bring the two together as anti-patterns, review questions, and an end-to-end path, and Chapter 19 returns to the conclusions and looks ahead. Each chapter ends with a summary. Architecture and engineering content are weighted roughly 60:40. Factual statements carry superscript citations (for example, [1]) that point to the References at the end of the book.

Questions mapped to chapters

Table 0.1: Questions this book answers and where to find them

Question	Where
How do you produce a CMS-0057-F Gap Document?	Sections 2.4 and 3.4
What does the rule require, and how do you measure compliance honestly?	Chapters 1 and 2
How do overlapping standards rank, and how do they show up in the build?	Chapter 3
FHIR R4 or R5?	Chapter 4
Facade or hybrid for an existing system?	Chapter 5
How do you choose mechanisms by action and exchange style by time horizon?	Chapter 6
How should authorization, consent, audit, and governance be designed?	Chapter 7
How should `$submit` be layered, and what is the status-code contract?	Chapters 8 and 9
Where do validation, defaults, and business rules belong?	Chapters 10 and 11
How do you write safely to a legacy system and trigger downstream rules?	Chapters 12 and 13
How do you prove conformance, keep evidence, and define “done”?	Chapters 14 and 15
How do you report progress, spot anti-patterns, and review designs?	Chapters 15–18
Conclusions and outlook	Chapter 19
Sources	References

Part I — Regulation and Architecture

This part answers “what must we do, and what should we choose”: regulatory scope and compliance measurement (Chapters 1–2), precedence of standards and FHIR version (Chapters 3–4), and architecture pattern, mechanism selection, and trust and governance (Chapters 5–7).

1. The Big Picture: What Interoperability Means and Who You Connect To

> **Bottom line:** The regulation defines *what* you must do and *by when*; FHIR and the implementation guides define *which formats and interfaces* you use. Track and report those two layers separately. CMS-0057-F spans several workstreams — Prior Authorization, the Patient Access enhancements, Provider Access, Payer-to-Payer, and the operational provisions — and each one is tracked and reported on its own. This book uses the PA API to illustrate engineering details, but compliance scope always covers every workstream.

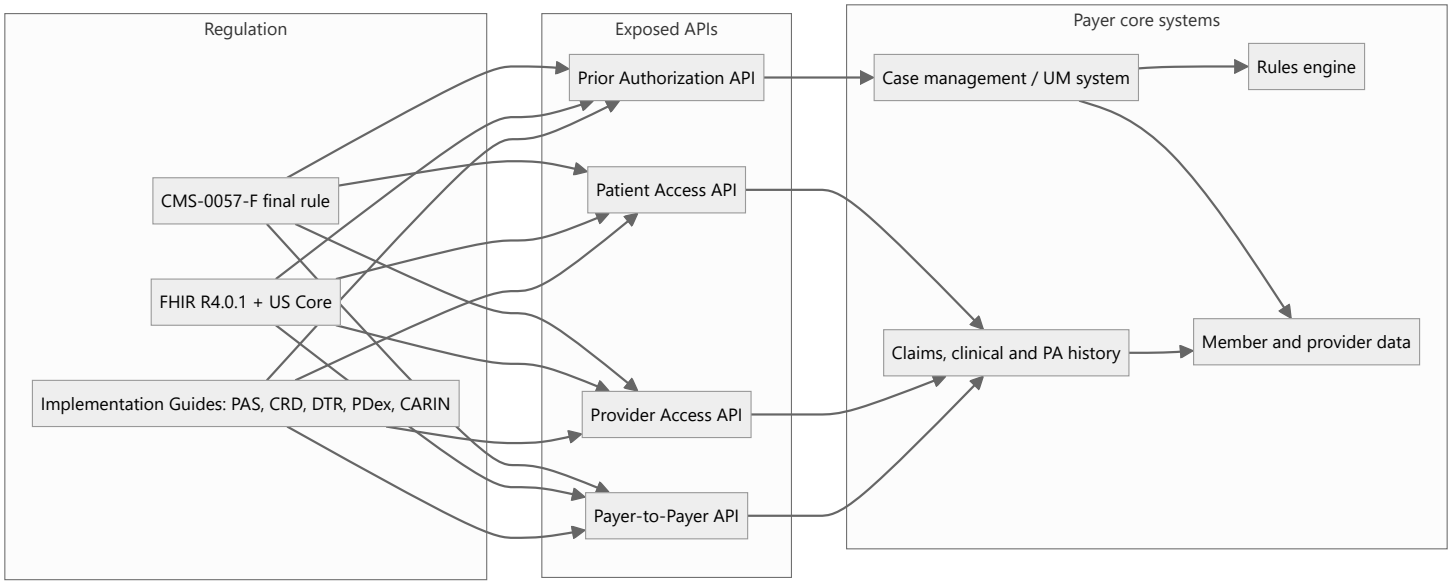

Figure 1.1: Three layers from regulation to legacy systems (Regulation → Exposed APIs → Payer core)

Key points:

- The regulation states what must be done and when; FHIR and the IGs state which formats and interfaces to use. They are not the same thing.
- Each of the four APIs has its own job.[2][3] The **Prior Authorization API** lets providers discover whether authorization is required (CRD), retrieve documentation requirements (DTR), and submit requests and track their status (PAS `$submit`, `$inquire`, and status notifications).[13][14][15] The **Patient Access API** adds prior authorization information to what it already exposes.[7] The **Provider Access API** gives in-network providers with a treatment relationship access to member data. The **Payer-to-Payer API** exchanges data between payers when a member changes plans or has concurrent coverage.[16]
- Separately, there are operational provisions that do not depend on any API: decision timeframes, specific denial reasons, and public reporting of prior authorization metrics (Section 2.1).
- Track and report each workstream on its own (Section 15.3); never fold them into a single percentage. Chapters 8–13 use the PA API's submission path to illustrate engineering details, and the same methods — layering, the status-code contract, fail-fast, safe writes, and evidence — apply to the other APIs.

Chapter summary: The regulation answers "what and when"; FHIR and the IGs answer "in what format." Track and report them separately. CMS-0057-F covers four API workstreams plus a set of operational provisions, and compliance is measured across all of them; the PA API is simply the example this book uses for engineering detail.

2. The Regulatory Layer: CMS-0057-F

> **Bottom line:** There is only one legitimate way to calculate a compliance percentage. The denominator comes from the requirements in the rule text, item by item; the numerator counts only requirements backed by test evidence. Progress on your own business rules must be reported under a separate heading and never labeled "compliance."

2.1 Fact Sheet (Public Information Used for Measurement)

- The six Prior Authorization API requirements (as organized in the CMS fact sheet):[2]
  1. A queryable list of covered items and services that require prior authorization
  2. Documentation requirements
  3. Exchange of requests and responses
  4. Approval, including the date or period the approval covers
  5. Denial, including a specific reason
  6. Requests for additional information
- Other APIs: Provider Access and Payer-to-Payer are **new** under this rule. Patient Access is an existing API from CMS-9115-F, and this rule requires it to **add prior authorization information** (see "A common misconception" in the Glossary).[1][7]
- Who is covered: the five types of impacted payers; commercial plans are out of scope (except QHPs on the FFEs).[3][4]
- API compliance dates: generally January 1, 2027; for Medicaid and CHIP managed care, the rating period beginning on or after that date; for QHP issuers, the plan year beginning on or after that date.[2][3]

- Operational provisions effective January 1, 2026: decision timeframes (72 hours expedited, 7 calendar days standard), a specific reason for every denial, public reporting of prior authorization metrics, and annual reporting of Patient Access API usage metrics to CMS.[1][2]
- Required standards: FHIR R4.0.1, US Core, SMART, and Bulk Data. Recommended IGs: CARIN, PDex, CRD, DTR, and PAS.[2][6][16][17]

> A note on versions: the CMS fact sheet lists the versions current when the rule was published (for example, US Core STU 3.1.1, SMART App Launch 1.0.0, Bulk Data 1.0.0, and STU 2.0.1 for PAS and CRD).[2] As of November 15, 2026, the HL7 publication pages show newer versions (for example, US Core 9.0.0, SMART 2.2.0, Bulk Data 3.0.0, and 2.2.1 for PAS and CRD).[10][11][12][13][14] Which versions you may use is governed by 45 CFR 170.215 and the latest CMS guidance;[6][8] record the versions you rely on in the Gap Document.

2.2 How to Measure Compliance

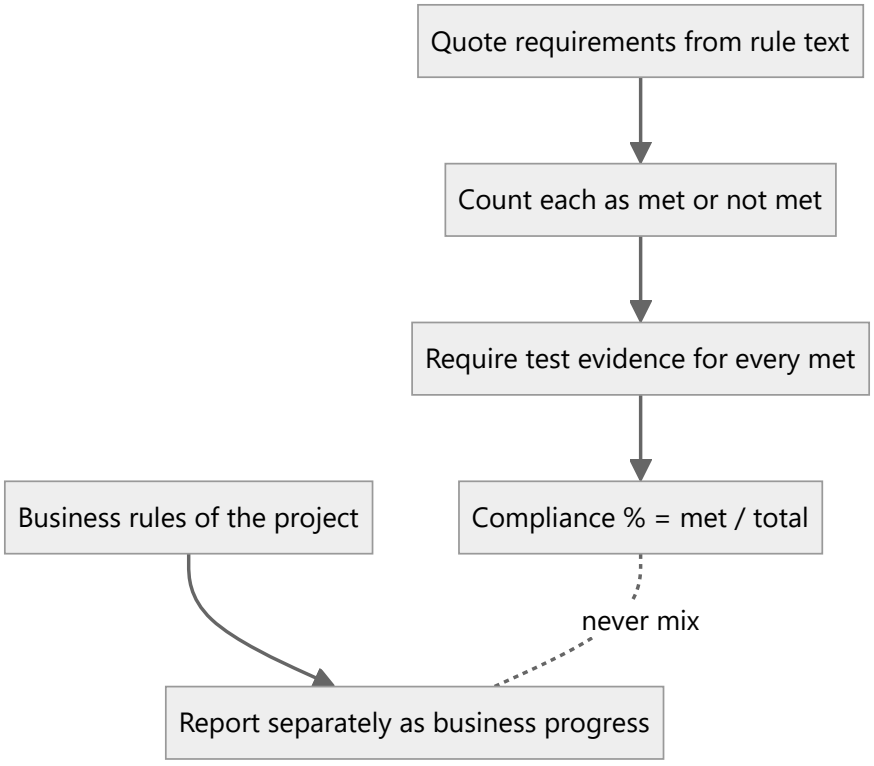

Figure 2.1: Measuring compliance — rule requirements and business progress are counted separately

- Denominator: the requirements extracted from the rule text, one by one.
- Numerator: a requirement counts as met only when there is test evidence. No partial credit.
- Report progress on the project's own business rules separately, labeled "business progress," never under a "compliance" heading.
- A common mistake: reporting the completion ratio of self-defined business rules as a compliance percentage. A self-defined yardstick cannot be called compliance.

2.3 The Leadership View (An Inference, Useful for Setting Expectations)

- Targets such as “percent usable before the regulatory date” are usually management goals, not regulatory measures.
- Report two numbers side by side: requirements met under the rule (a hard number) and business capability coverage (an estimate, with its basis stated). Being roughly right beats being precisely wrong.

2.4 The Deliverable: The CMS-0057-F Gap Document

The Gap Document turns the fact sheet in 2.1 and the measurement method in 2.2 into a deliverable: **what the rule requires, what we have today, how big the gap is, who owns it, when it closes, and where the evidence is.** It is the single source of the compliance percentage and the source of the "requirements met" figure used in progress reporting (Section 15.3).

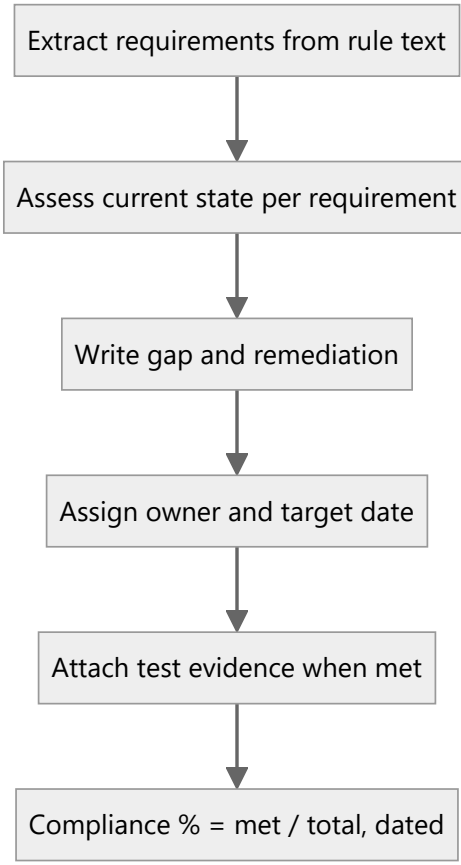

Figure 2.2: Producing the Gap Document (extract → assess → describe the gap → assign → attach evidence → calculate)

Steps

1. **Extract requirements (the denominator).** From the rule text (*Federal Register*, FR Doc 2024-00895),[1] extract every requirement that applies to your organization, each with its citation (CFR section or preamble passage). Group them by API: Prior Authorization (six requirements), Provider Access, Payer-to-Payer, Patient Access (added PA information), and operational provisions (timeframes, denial reasons, metrics reporting).
2. **Determine applicability.** Confirm which type of impacted payer you are and which compliance date applies. Mark requirements that do not apply as "Not applicable" with a reason — do not quietly remove them from the denominator.
3. **Assess the current state.** For each requirement, describe what the system does today: full capability, partial capability, or none. State facts, not intentions.
4. **Describe the gap and the remediation.** The gap is the requirement minus the current state. Write the remediation at an actionable level of detail (which interface, which work item).
5. **Assign an owner and a date.** One owner and one target date per requirement; no target date may fall after the regulatory date.
6. **Attach evidence (the numerator).** A requirement is "Met" only when test evidence is attached (Chapter 14). No partial credit.
7. **Calculate and date the percentage.** Requirements met divided by applicable requirements, with the calculation date and the rule version it is based on.

Table 2.1: Gap Document template (sample rows)

ID	Citation	Requirement (summary)	Applies	Current state	Gap	Remediation / work item	Owner	Target date	Evidence	Status
G-PA-03	PA API: request and response exchange	Accept submissions in FHIR and return a decision	Yes	`$submit` live in test	`$inquire` not built	Build `$inquire`	Developer A	2026-12-15	—	Not met
G-PA-05	PA API: denial reason	Every denial gives a specific reason	Yes	Rules engine returns only a generic denial code	No specific reason	Rule output adds reason code and cited basis	Rules team	2026-12-31	—	Not met
G-OP-01	Operational: decision timeframes	72 hours expedited / 7 calendar days standard	Yes	Process runs to the timeframes	None	—	Operations	2026-01-01	Timeliness report	Met

Ground rules

- One requirement per row, and one status per row (Met / Not met / Not applicable).
- The project's own business rules do not go in this table; they get a separate "business progress" table (Section 2.2).
- Keep it consistent with the requirements traceability matrix in Section 3.4: the Gap Document covers only the regulatory layer, while the traceability matrix covers every layer (HIPAA, the rule, IGs, contracts, and clinical guidelines).
- Date and version the document. When the rule or an IG changes, re-review it using the triggers in Section 4.5.

Chapter summary: The denominator of a compliance percentage comes only from the rule text, and the numerator counts only requirements with test evidence. Business progress gets its own heading and never mixes with "compliance." Report the management target and the regulatory figure side by side, each with its basis. Make this concrete in a CMS-0057-F Gap Document: extract each requirement, assess each one, attach evidence to each one, and date the percentage.

3. Precedence Among Overlapping Standards, and How Each Shows Up in Practice

> **Bottom line:** Precedence runs in one chain, from highest to lowest — **HIPAA (law) > the regulation (CMS-0057-F) > technical standards named by the regulation (FHIR R4, US Core, SMART, Bulk Data) > implementation guides (PAS, CRD, DTR, PDex, CARIN) > clinical guidelines and coverage criteria (CG).** Contracts and internal conventions each have a fixed place in the chain: contracts sit below the regulation and above the IGs, and internal conventions are always last. When requirements conflict: take the strictest rule for privacy, follow the regulation for obligations, the named standard for format, the IG's SHALL statements for detail, and the cited version of the guideline for content.

A project is bound by HIPAA, CMS-0057-F, FHIR, the IGs, and clinical guidelines at the same time. They do not sit at the same level: some are law, some are technical formats, and some are business content. Sort them into layers before you rank them.

> Note: this chapter reflects an architecture and engineering perspective and is not legal advice; your compliance and legal teams have the final word on interpretation. In this book, "CG" means clinical guidelines or coverage criteria (for example, medical necessity criteria or a plan's own coverage policies).

3.1 What Layer Each Standard Belongs To

Table 3.1: Layer, nature, and binding force of each standard

Standard	Nature	Question it answers	Binding force
HIPAA (Privacy, Security, Transactions and Code Sets)[18][19]	Law	Whether data may be used, how it is protected, which transaction standards already apply	Mandatory, with penalties
State law, special protections (e.g., substance use disorder records[21]), contract terms	Law or contract	Requirements stricter than HIPAA; what customers require of you	Mandatory (the stricter rule wins)
CMS-0057-F	Regulation	Which APIs must be offered, with what timeframes and content	Mandatory, with compliance dates
FHIR R4.0.1, US Core, SMART, Bulk Data	Standards named by the regulation	The foundation for data format, authorization, and bulk exchange	Mandatory (because the regulation names them)
IGs: PAS, CRD, DTR, PDex, CARIN	Recommended implementation guides	Narrow FHIR options and define interactions for specific use cases	Recommended, not required, but they reduce interoperability risk
CG: clinical guidelines or coverage criteria	Business content	The basis for decisions: is a service medically necessary, is it approved	Set by the customer or plan; usually licensed
Internal project conventions (status-code table, defaults)	Internal standard	Specific choices within the space above	Your call, but they may not violate any layer above

3.2 How Precedence Works (Highest to Lowest)

In one line: `HIPAA (law) > REGULATION (CMS-0057-F) > standards named by the regulation, such as FHIR > IGs > CG`. Where contracts and internal conventions fit: contracts sit under the regulation (they may be stricter, never looser), and internal conventions are always at the bottom.

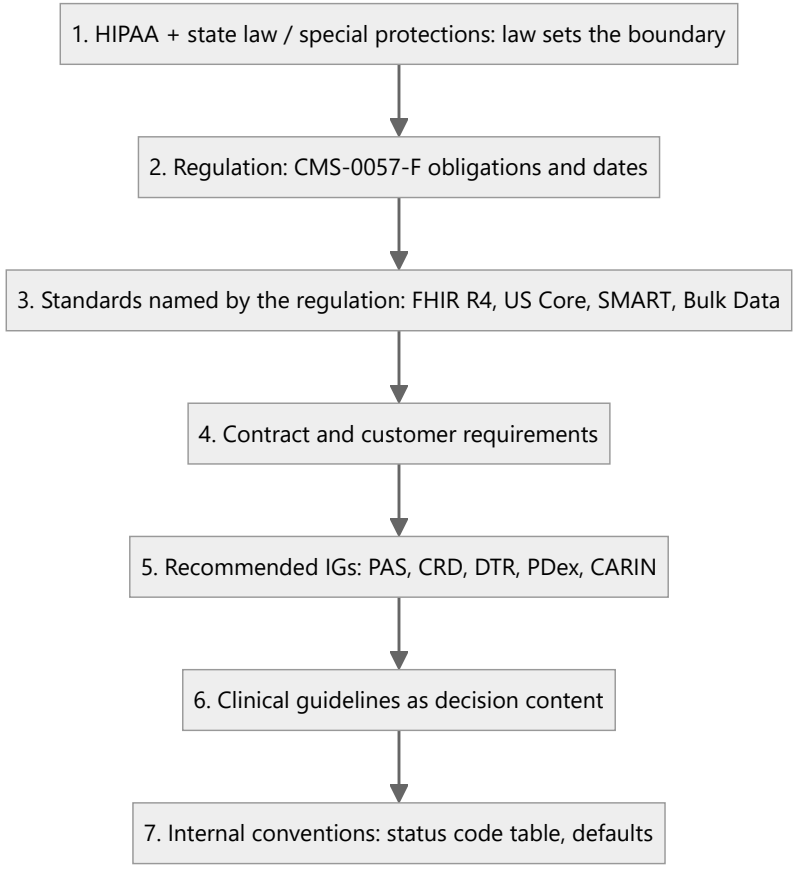

Figure 3.1: The precedence chain (highest to lowest)

In a sentence: **law sets the boundary, the regulation sets the obligation, standards set the format, IGs set the detail, guidelines set the content, and internal conventions settle the remaining choices.**

Rules for resolving conflicts

1. Privacy and security: apply the **strictest** rule (when HIPAA and a stricter state law both apply, the stricter one wins).

2. Obligations and timeframes: follow the regulation. A contract may be stricter but never weaker.
3. Data format: the standards named by the regulation come first. An IG may narrow what a standard allows but may not contradict it.
4. Where an IG is silent: fall back to base FHIR or US Core. Use an internal convention only when the base standards are silent too.
5. When an IG conflicts with a customer contract: follow the contract, but document the deviation and its impact on interoperability.
6. Internal conventions always rank last. When one conflicts with any layer above, change the convention, not the layer above.

3.3 How Each Layer Shows Up in Practice

HIPAA (applies everywhere; it is not a feature)

- Minimum necessary: return only the fields the use case needs; keep PHI out of logs; never echo sensitive values in error messages.[18][20]
- Encrypt in transit and at rest; enforce role-based access; keep an audit trail that can answer "who accessed whose data, and when."[19]
- Tokens and credentials never go into source control, documents, or chat (consistent with Section 14.3).
- Existing transaction standards such as X12 278 remain in force.[22] The FHIR PA API is an **additional** channel required by the regulation, not a replacement. Both channels must produce the same business outcome for the same request.
- Incident and breach response belongs to the compliance team; engineering's job is to make events searchable, containable, and reportable.

CMS-0057-F (decides what to build and when)

- List the six PA API requirements as compliance denominators and record them in the Gap Document (Sections 2.2 and 2.4). Put the dates

in the plan: APIs generally by January 1, 2027; decision timeframes, denial reasons, and metrics reporting from January 1, 2026.

- Decision timeframes and specific denial reasons are **business process** requirements, not just interface requirements. The API must be able to carry them, but meeting the timeframes depends on process and rules.
- Measurement: a requirement is met only when there is test evidence.

FHIR R4 and US Core (decide what the data looks like)

- Validate resources and profiles at the front door (Chapter 10); report every error as an OperationOutcome.[10][29]
- SMART handles authorization; Bulk Data handles bulk export.[11][12] A bearer token pasted into a testing tool such as Swagger is a test convenience, not a production authorization design.

Implementation guides (decide how systems interact)

- PAS defines the submission and inquiry Bundles, `$submit`, and `$inquire`; CRD signals when authorization is required; DTR collects supporting documentation.[13][14][15]
- Approach: implement the IG's **SHALL** statements first; decide SHOULD and MAY statements on business value, and record the rationale.
- Never skip something the regulation requires because "the IG doesn't require it," and never expand scope because "the IG recommends it" (scope follows the acceptance criteria; see Section 15.1).

Clinical guidelines (decide how requests are adjudicated)

- Guidelines are inputs to the rules engine and to clinical review, not an API format. The API's job is to bring in the clinical information the decision needs and to send out the decision with a **specific reason**.
- Licensing: commercial guideline content is usually licensed. Do not copy it into code, documents, or test data; store only the reference identifier and version.

- Versioning: guidelines change, so record which version each rule relies on, for retrospectives and appeals.
- Decisions and audit: every denial must trace back to a specific basis, not just “does not meet criteria.”

3.4 A Practical Tool: The Requirements Traceability Matrix

Record every requirement in one row, with its source layer, citation, where it is implemented, and its test evidence. With this matrix, “are we compliant, and why did we build it this way?” always has a documented answer.

Table 3.2: Requirements traceability matrix (sample rows)

ID	Source layer	Citation	Implemented in	Test evidence	Status
R-01	Regulation	Request and response exchange	`$submit` entry point	Positive and negative cases pass	Met
R-02	HIPAA	Minimum necessary; no PHI in logs	Logging and response layers	Log sampling	Pending verification
R-03	IG (SHALL)	Submission Bundle structure	Validation layer	Structure test cases	Met
R-04	Contract	Respond per the customer’s status-code table	Status-code mapping	Status-code test cases	Met
R-05	CG	Denials cite a specific basis	Rules output	Sample comparison	Not started

3.5 Decision Flow: When Requirements Conflict or Are Unclear

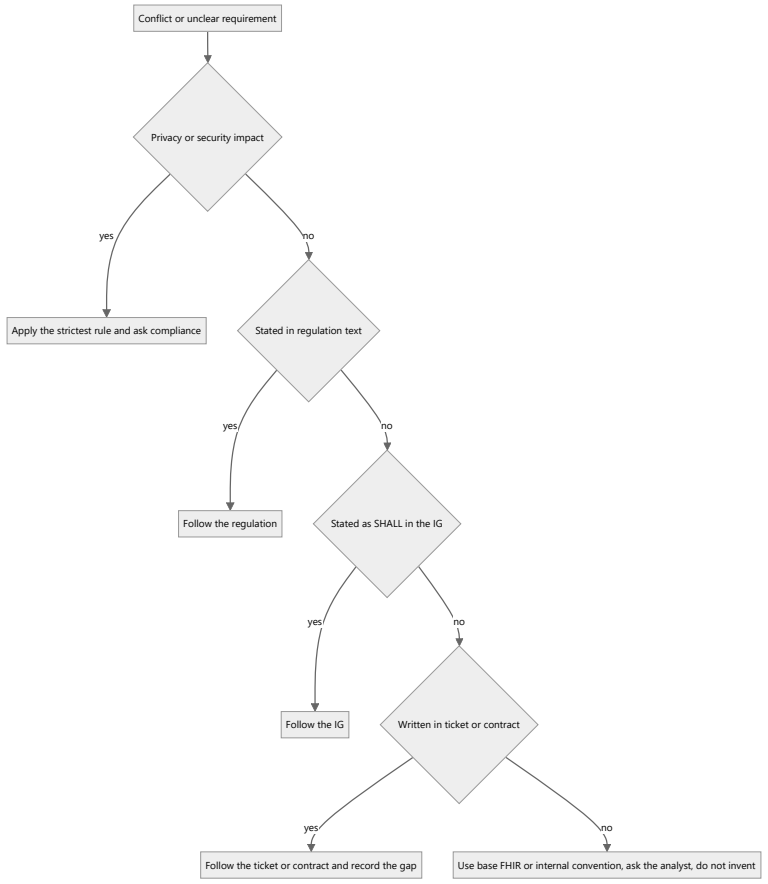

Figure 3.2: Decision flow for conflicting or unclear requirements

3.6 Common Mistakes

1. Treating a recommended IG as a mandatory regulation — or, conversely, ignoring the standards the regulation names.

2. Counting progress on the project's own business rules as a compliance percentage (Section 2.2).
3. Building only the FHIR channel and forgetting that its results must match the existing transaction channel.
4. Copying guideline text into code or test data, in violation of the license.
5. Assuming that passing FHIR validation means compliance. Validation covers format only, not privacy, timeframes, or denial reasons.
6. Carrying a relaxed test authentication setup into the production design.

Chapter summary: Sort into layers first, then rank: law sets the boundary, the regulation sets the obligation, standards set the format, IGs set the detail, guidelines set the content, and internal conventions settle the rest. In a conflict, privacy takes the strictest rule, obligations follow the regulation, format follows the standard, detail follows the IG's SHALL statements, and everything else follows the acceptance criteria. Every requirement goes into the traceability matrix with a source, an implementation, and evidence.

4. Choosing a FHIR Version: R4 or R5

> **Bottom line (as of November 10, 2026):** Use **R4 (4.0.1)** for every API governed by the regulation; watch R5, but do not build on it. The deciding factors are not technical taste but three facts about the ecosystem: the regulation names R4, the IGs are built on R4, and your trading partners can consume R4.

4.1 The Conclusion

For every API governed by the regulation (Prior Authorization, Patient Access, Provider Access, and Payer-to-Payer), use R4 (4.0.1). Treat R5 as something to watch, not as the implementation baseline.

The reasoning: **a FHIR version is an ecosystem constraint, not a technical preference.** The regulation names it, the IGs are built on it, and your partners can consume it — and all three point to R4.

4.2 Why R4

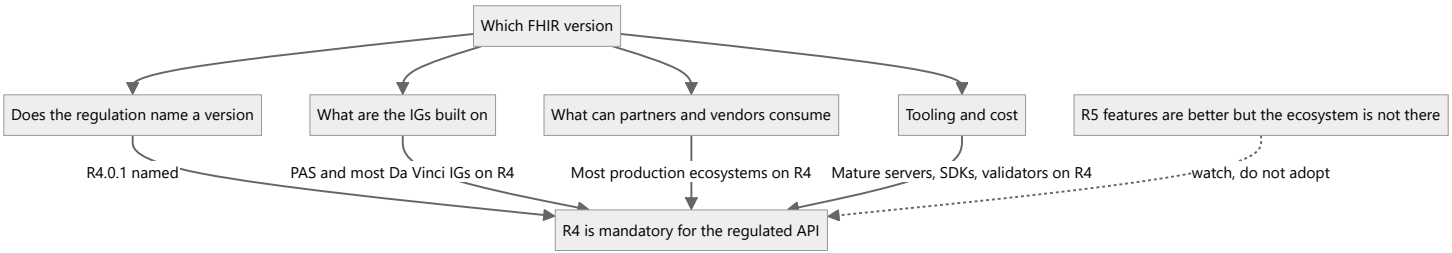

Figure 4.1: All four lines of evidence point to R4

1. **Regulation.** The CMS API requirements page lists HL7 FHIR R4.0.1 as the base standard. The accompanying US Core and SMART versions are governed by 45 CFR 170.215[6][8] and are updated on a rolling basis (for example, US Core 6.1.0 as an advancing version with 3.1.1 expiring, and SMART 2.0.0 as a newer option).[2][10][11] Choosing R5 means leaving the baseline the regulation names, and compliance would need additional justification.
2. **Implementation guides.** PAS is built on R4 (the current page shows PAS 2.2.x on R4; check the publication directory for the latest). Its subscription support relies on the Subscriptions R5 Backport, which **brings R5 topic-based subscriptions to R4 through extensions**.[13][23] That is exactly how the ecosystem works: borrow R5's benefits on top of R4 rather than migrating wholesale.
3. **Trading partners.** The vast majority of counterparties — providers, other payers, intermediaries — speak only R4. If you expose R5, they cannot connect.
4. **Tooling cost.** Servers, SDKs, validators, and test tools are most mature on R4, and so is your team's experience.
5. **Migration cost versus benefit.** R5's improvements are real, but they do nothing to help you meet the 2027 compliance date, and they add new risk.

4.3 How the Versions Relate

Table 4.1: Where each FHIR version stands (as of November 10, 2026)

Version	Number	Position	Notes
R4	4.0.1	The working baseline for current regulations and IGs	Named by the regulation[2][9]
R4B	4.3.0[24]	A minor revision of R4, mainly strengthening subscriptions and some domains	Not fully interchangeable with R4; check dependencies before mixing
R5	5.0.0	The newer normative release	Lower ecosystem adoption than R4
R6	In development	The next generation	Status not verified here; do not depend on it

Note that version differences are more than “a few extra fields.” Resource structures, terminology bindings, and the subscription model all change. Moving from R4 to R5 is not an in-place upgrade; it requires a translation layer.

4.4 In Practice: Make the Version Replaceable

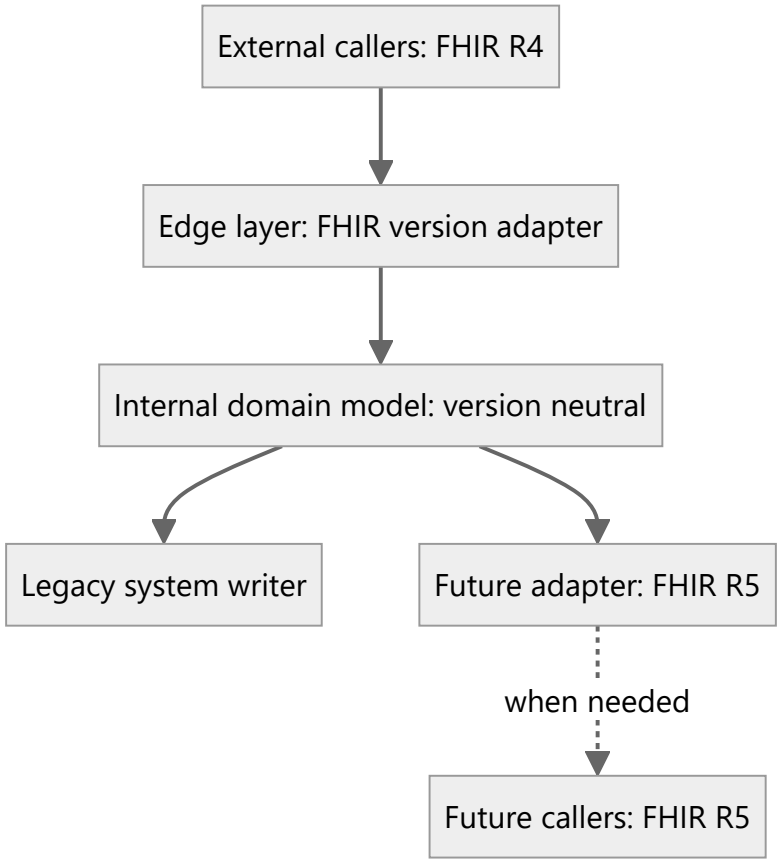

Figure 4.2: Isolating the FHIR version in an edge adapter

1. **Isolate the FHIR version at the edge.** Validation, parsing, and serialization happen at the edge; inside, use a version-neutral domain model. Supporting R5 later then means adding an adapter, not reworking the whole system.
2. **Declare the version explicitly.** State the supported FHIR version in the CapabilityStatement, and use the `fhirVersion` media-type parameter defined by FHIR (for example, `application/fhir+json;`

`fhirVersion=4.0`) so that version negotiation is observable and testable. [25][26][27]

3. **Never mix R4 and R5 on the same endpoint.** If you must run both, use separate paths or base URLs, each with its own validation and tests.
4. **When you need an R5-only capability,** look for a backport first (such as the R4 subscriptions backport) before considering a version upgrade.
5. **Pin your dependencies too.** Lock US Core, SMART, and IG versions together with the FHIR base version and document them. An upgrade is a planned change, not an incidental dependency bump.
6. **Put version information in the traceability matrix.** Record the version each requirement relies on (Section 3.4); otherwise no one will be able to explain later why it was built that way.

4.5 Review Triggers (Revisit the Decision If Any of These Occurs)

- The regulation or a related rule (including updates to 45 CFR 170.215, or a new CMS final or proposed rule) names a new FHIR version or new US Core or SMART versions.
- A major IG (especially PAS) publishes an R5-based version or deprecates the current baseline.
- An important partner or customer requires R5 in writing.
- Your platform, server, or gateway vendor officially supports R5 and migration cost drops significantly.
- An R4 backport for a key capability, such as subscriptions, hits a limitation you cannot work around.

4.6 Common Mistakes

1. “R5 is newer, so it’s better.” Newer is not the same as suitable; suitability depends on whether the regulation, the IGs, and your partners are on that version.
2. Writing an IG or standard version number into a document as a permanent fact, without a date. Everything changes; documents must be dated.
3. Trying to jump to a new version in one step and ignoring the cost of the translation layer and regression testing.
4. Forgetting that R4B differs from R4, and mixing dependencies until tools fail.
5. Leaving no decision record for the version choice, so no one knows the reasoning once people move on.

Chapter summary: Use R4, because the regulation, the IGs, your partners, and the tooling are all on R4. Isolate the version in an edge adapter so a future switch stays affordable, and use explicit review triggers — not instinct — to decide when to revisit. Date every conclusion (November 10, 2026).

5. Choosing an Architecture Pattern: Facade, Hybrid, or FHIR-Native

> **Bottom line:** There are three FHIR architecture patterns — facade, hybrid, and FHIR-native. **For a compliance program built on existing legacy systems (the focus of this book), the realistic options are facade and hybrid; FHIR-native belongs to new systems and is out of scope here.** A facade is inherently slower: every request queries the legacy system in real time and transforms the result into FHIR resources on the fly, so response time is bounded by the slowest source system. But it is fast to deliver — typically a few months — which makes it a good way to meet a compliance date. A hybrid performs well because requests are answered directly by a FHIR server, at the cost of a synchronization pipeline that is hard to build and harder to run.

5.1 The Three Patterns (Following Devitt's Definitions)

Table 5.1: The three FHIR architecture patterns compared

Pattern	Source of truth	How it works	Performance	Time to deliver	Best fit
Facade	Legacy system	Queries the legacy store when a request arrives and maps the result to FHIR on the fly	Low — every request is a live query against the source; assembling a Bundle from several systems is worse	Fast, typically months	A regulatory deadline, no appetite for copying data, legacy structures stay as they are
Hybrid	Legacy system (the FHIR server holds a synchronized copy)	Change data capture or events keep the FHIR server in sync; consumers read the copy	High — the FHIR server answers requests directly	Slow; synchronization and error handling are complex	The legacy system stays long term, the API will see real use, performance matters
FHIR-native	FHIR server (for some or all resources)	Requests read and write the FHIR server directly, with no legacy translation	High	Slow — governance, MDM, MPI, and similar components must be built	Greenfield systems or new business domains

(Pattern definitions and the performance analysis follow Devitt, *FHIR Architecture Decisions*; see References.[33])

5.2 Decision Tree: Answer Four Questions in Order

Choosing a pattern comes down to answering a few questions in a fixed order: first where the system of record lives, then what is driving the project and how the API will be consumed, and finally whether the team can build and operate synchronization.

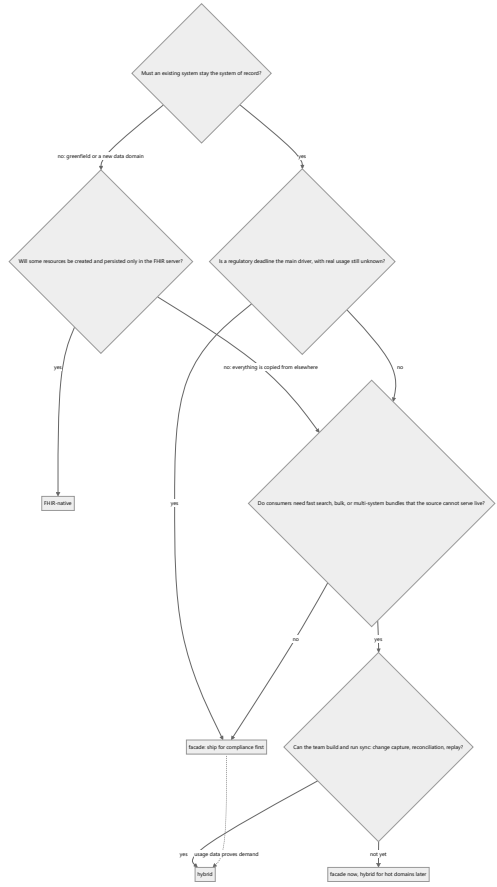

Figure 5.1: Architecture pattern decision tree (system of record → driver → consumption → sync capability)

Table 5.2: What each question in the decision tree is really asking

Question	Why ask it	Typical signals
Must an existing system remain the system of record?	Determines whether FHIR-native is even possible	Core business systems will stay for years; other applications still write to them
Will some resources be created and persisted only in the FHIR server?	The one test that separates native from hybrid[33]	Data in a new business domain is born as FHIR resources from day one
Is a regulatory date the main driver?	Deadline-driven compliance favors speed to delivery	Must be live before January 1, 2027; real call volume unknown
Can the source systems serve the consumption pattern directly?	Determines whether you need an operational copy	Bulk export, complex search, Bundles assembled across systems, or source systems that cannot take extra load
Can the team build and operate synchronization?	Most of a hybrid's cost is in synchronization, not in the FHIR server	Change data capture, reconciliation, failure replay, latency monitoring

"Facade now, hybrid later" is not a compromise; it is often the right answer. Ship a facade to meet the compliance date, then use real usage data to pick the hot domains and add synchronized copies one at a time.

5.3 Why Facade and Hybrid Are the Only Options for an Existing System

FHIR-native assumes the FHIR server becomes the source of truth for some or all data. For a payer with existing core systems — case management, a rules engine, member data — moving the source of truth means rebuilding those core systems. That is impossible before the regulatory deadline and rarely worth it afterward. Devitt offers a practical test: **"Is there any resource that is created and persisted only in the FHIR server?" If yes, it is native; if every piece of data is copied from somewhere else, it is hybrid.**[33]

So the choice narrows to a single trade-off:

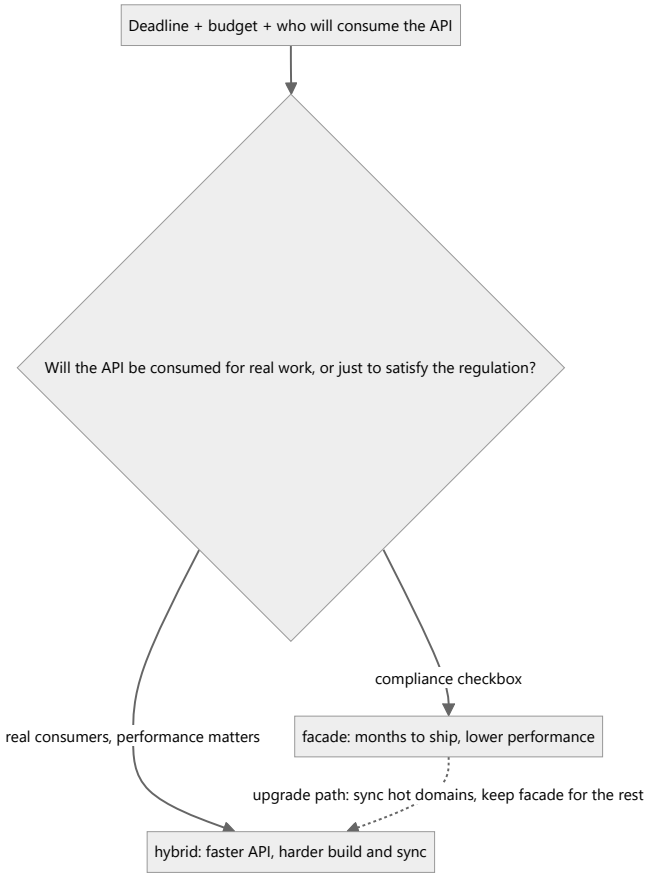

Figure 5.2: The trade-off for existing systems — facade versus hybrid (with the upgrade path)

5.4 The Performance Cost of a Facade

A facade is not slow because it was built carelessly; it is slow by design. **Every request is a live query against the source system plus an on-the-fly mapping.** Response time equals the time of the slowest legacy database or internal API — usually outside the FHIR team's control — and assembling a single Bundle from several systems makes it worse. The regulation says almost nothing about performance (at most a "timely

manner" requirement with no teeth), so a facade can be fully compliant, pass an audit, and still take seconds or even minutes to respond — and then never get fixed.[33]

What this means for engineering (consistent with the rest of the book):

- On the facade route, the validation and mapping layers in Chapters 10 and 11 sit on the hot path and must be treated that way.
- The 503 and 504 responses in the Chapter 9 status-code table (downstream unavailable, timeout) are a **routine risk** in a facade architecture, not a rare exception.
- Load-test against realistic legacy data volumes before go-live, and record the P95 response time as a known limitation in the Gap Document — don't wait for consumers to complain.

5.5 Why a Facade Is Fast to Deliver

- No data copying and no changes to legacy structures — you avoid a hybrid's synchronization pipeline and reconciliation.
- The regulation usually names only a small set of resource types and search parameters; a small team can focus on that subset and go live in a few months.
- You meet the January 1, 2027 compliance date first and save the hybrid investment for the domains that prove they are actually used.

This book's position: **a facade is the default starting point for compliance on existing systems, not the destination.** Used correctly, it separates the compliance obligation from real consumption — comply first, then let usage data decide which domains deserve an upgrade to hybrid.

Direction of migration: facade → hybrid → FHIR-native is a workable, incremental path — each step adds a copy on top of the existing source or moves the source of truth for some resources into the FHIR server, and it can proceed domain by domain. Migration in the other direction is rare and

often impractical: once some data is born only in the FHIR server, going back to hybrid or facade means building a legacy store and a full write path for that data. So you only need to pick the right starting point for today, not the final destination.

Chapter summary: Of the three patterns, FHIR-native does not apply to existing systems. A facade trades performance for speed and is the default for deadline-driven compliance; a hybrid trades implementation complexity for performance and is the upgrade once the API sees real use. To tell which pattern you have, ask whether any resource is persisted only in the FHIR server; to choose, walk the decision tree in Figure 5.1 in order. Migration works only in one direction — facade → hybrid → FHIR-native — and rarely, if ever, in reverse.

6. An Architecture Decision Method: Size the Work First, Then Choose a Mechanism per Action

> **Bottom line:** Architecture decisions follow a fixed order — **scope → actors → constraints → mechanisms → components.** The right design satisfies every constraint and solves nothing beyond them. Break the requirements into concrete business actions and choose the most suitable mechanism for each. For a deadline-driven compliance program, building a do-everything platform rarely pays off.

6.1 The Order of Decisions

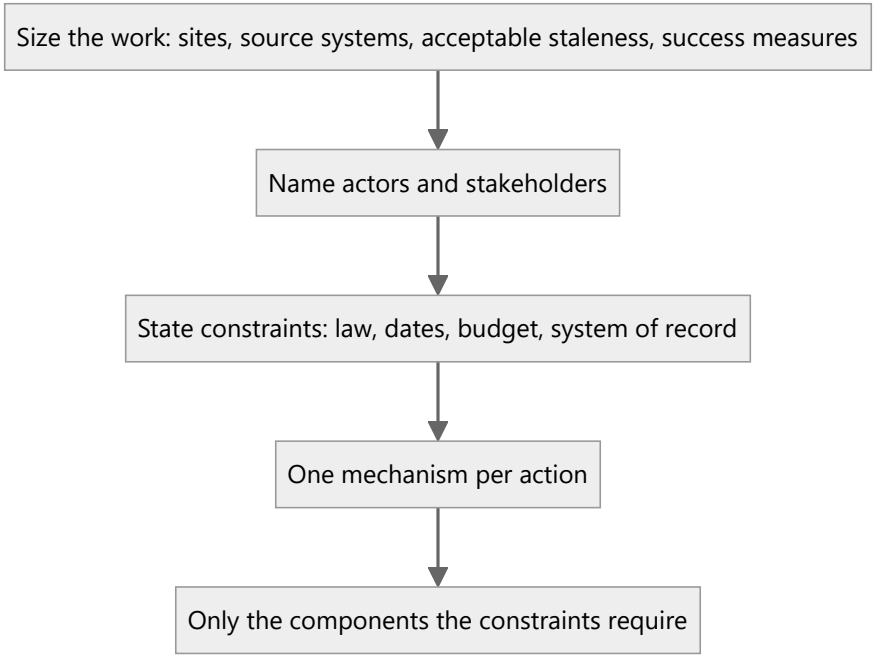

Figure 6.1: Five steps of an architecture decision (classify before you design)

- **Size the work first.** How many source systems, how many trading partners, how stale the data may be, and what success looks like — write these down before you open any IG.
- **Separate actors from stakeholders.** *Actors* are the systems doing work on the information path (EHRs, the PA API, the rules engine, the consent service, the gateway). *Stakeholders* are the people and organizations with interests (providers, members, regulators, vendors). Keep two lists, so that a person's wish does not get mistaken for a system interface.
- **State the constraints out loud.** Regulatory dates, budget, privacy rules — and, most important, **where the system of record lives**,

which directly decides facade versus hybrid (Chapter 5).

- **Deliver a well-scoped API first.** One compliance gap plus a fixed budget calls for a well-scoped API. Choose a server product only after the requirements and constraints are clear; otherwise the product's features end up shaping the architecture.

6.2 The “Actions” of CMS-0057-F and Their Mechanisms

Translate the regulation’s requirements into actions and assign one mechanism to each action, and the architectural skeleton of all four APIs falls out.

Table 6.1: CMS-0057-F workstreams — action → mechanism → standard

Workstream	Action (what must happen)	Mechanism	Standard / IG	Authorization profile
Prior Authorization	Discover whether a service needs authorization	Call the payer’s service from clinical workflow hooks (ordering, scheduling) and return guidance cards	CDS Hooks[34], CRD[14]	CDS Hooks’ own JWT
	Gather documentation requirements and prepopulate them	Questionnaire plus CQL prepopulation, hosted in a SMART app	DTR[15]	SMART App Launch (provider)
	Submit a request and receive a decision	The `$submit` operation with a Claim-centered Bundle	PAS[13]	System-to-system authorization
	Track a pended request	`$inquire`; status changes may be pushed via subscription notifications (check the current PAS version)	PAS, Subscriptions Backport[23]	Same as above
Patient Access (enhanced)	Members read their PA information through an app	REST reads and searches	US Core, CARIN BB, PDex[10][17][16]	SMART App Launch (patient)

Workstream	Action (what must happen)	Mechanism	Standard / IG	Authorization profile
Provider Access	Determine which members are attributed to which provider	An attribution list (expressed as a Group)	PDex[16]	—
	Pull data in bulk for attributed members	Asynchronous bulk export (`$davinci-data-export`, built on Bulk Data)	Bulk Data[12], PDex	SMART Backend Services[35]
Payer-to-Payer	Identify the same member across the old and new payer	Member-match operations (such as `$bulk-member-match` / `$multi-member-match`)	PDex	SMART Backend Services
	Record and check the member's opt-in	Consent records with a centralized decision	FHIR Consent[37]	—
	Exchange historical data	Asynchronous bulk export	Bulk Data, PDex	SMART Backend Services
Operational provisions	Calculate decision timeframes and publish metrics	Not an API: an analytics store plus reporting	Not prescribed	—

> Note: PDex defines more than one Provider Access workflow version. Check the current publication for the version the IG recommends for CMS-0057-F, and confirm operation names against the IG version you adopt.[16]

6.3 Choosing an Exchange Mechanism by Time Horizon

The same data is consumed on different clocks by different consumers. The time horizon decides the mechanism, not the other way around.

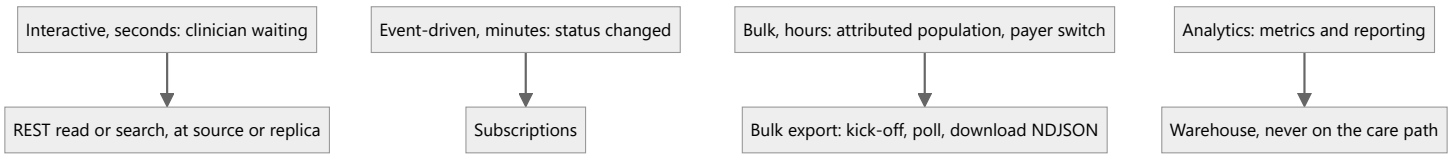

Figure 6.2: Time horizon → exchange mechanism

Table 6.2: Choosing a Bundle type or exchange envelope[28]

Constraint signal	Mechanism	Common misuse
The caller is waiting, and several resources must all succeed or all fail	transaction (POST to the server base)	Using batch — each entry is independent and partial success is allowed
A signed snapshot with narrative that must be stored whole	document (Composition)	Using a transaction as an archive
Filtered change notifications within seconds	Subscription[38]	Using bulk export for real-time notification
Replaying every event in order after hours offline	Message queue	Treating subscriptions as a replayable queue
Population data, hundreds of thousands to millions of resources	Bulk `$export` (asynchronous NDJSON)	High-frequency, one-at-a-time REST GETs

How this connects to the engineering chapters: a PA submission needs all-or-nothing semantics. When the target is a legacy system with no FHIR transaction to lean on, Chapter 12's draft-then-release pattern is how a facade architecture delivers those semantics.

Architecture judgment calls

- **An API gateway cannot fix back-end performance.** A gateway handles routing, authentication, throttling, and caching, but every request still has to reach the back-end systems. If assembling data live from several source systems is too slow, improve the read path (for example, add a copy of the data and move toward hybrid) rather than counting on the gateway.
- **Check network direction for subscriptions.** A REST hook is an inbound call from the server to the consumer. If the consumer's network allows outbound traffic only, you need an outbound channel (such as WebSocket, or an intermediary that receives on the consumer's behalf).
- **Identifiers first.** When you have a trusted identifier, search by `identifier=system|value`; don't default to matching on name and birth

date. `_id` is a server-private key and must not be used across servers.[25]

- **Must Support does not mean always present.** It means "if you have the data, you must be able to send and process it," and it holds only when backed by source data and the server's declared behavior.

Chapter summary: Size the work, separate actors from stakeholders, state the constraints, and then choose a suitable mechanism for each business action. The four CMS-0057-F APIs reduce to the skeleton in Table 6.1. Time horizon decides the exchange mechanism, and semantics decide the Bundle type. Performance bottlenecks have to be fixed on the read path; a gateway will not fix them.

7. Trust and Governance: Who Can See What, and How the Network Stays Resolvable

> **Bottom line:** Security is made up of several independent capabilities — authentication, authorization, consent, audit, API gateway, and signatures — and each one is needed. Every read or write of member data must decide **who the requester is, whether their role and attributes allow it, and whether the member has approved this purpose**, and leave an append-only audit record. TLS only encrypts the transport; those decisions are made by the other capabilities. Governance keeps identifiers, profiles, versions, and terminology resolvable across a multi-party network over time.

7.1 Access Decisions and Audit on Every Request

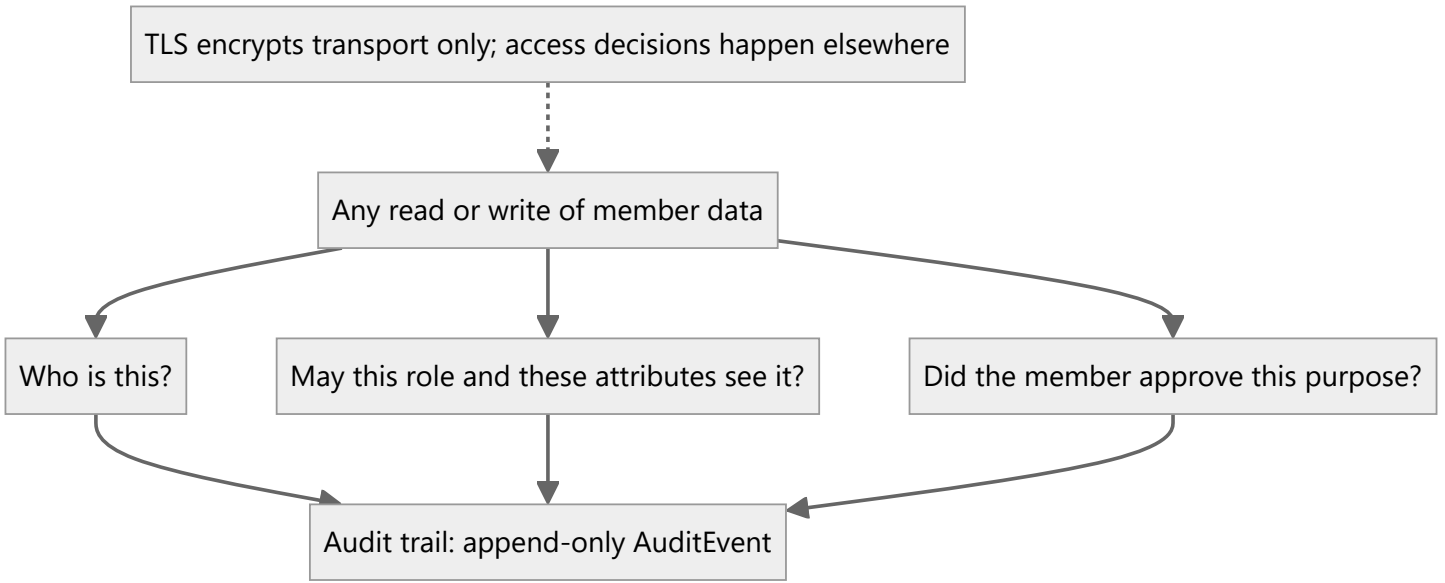

Figure 7.1: Access decisions and the audit trail

- **Audit.** Record who accessed what, when, and for what purpose as AuditEvent resources in append-only storage, retained for the legally required period. Resource version history in the data store is not an audit solution.[36][40]
- **Centralize consent decisions.** Provider Access opt-out and Payer-to-Payer opt-in are consent rules. Put the decision in one consent service that every API calls, rather than scattering it across each interface's code.[37]
- **Attributes matter more than roles.** Provider Access depends on a treatment relationship, or attribution, which changes over time. Role-based access control (RBAC) alone is not enough; evaluate multiple attributes — role, attribution, consent, purpose — at request time, which is attribute-based access control (ABAC). A roster generated nightly is a stale snapshot.
- **Minimum necessary.** Achieve it with narrow scopes and the smallest data set, not by "taking everything and promising not to look."[20] De-identification belongs to secondary-use pipelines and must never sit on a business path that still needs real identities.

7.2 Two SMART Authorization Profiles

Table 7.1: The two SMART profiles and the workstreams they serve

Profile	Scenario	Key mechanisms	CMS-0057-F workstreams
SMART App Launch	One person, one browser, one patient in context	Authorization code with PKCE, short-lived access tokens, scopes limited by resource and interaction	Patient Access (member apps); DTR apps (providers)
SMART Backend Services	Unattended system jobs	Asymmetric JWT client authentication, system-level scopes[35]	Bulk exchange for Provider Access and Payer-to-Payer

There is only one signal to read: **is a person at the keyboard?** If yes, it is App Launch; if not, it is Backend Services. Shared passwords and open URLs are never the answer.[11]

7.3 The Building Blocks of Security

Table 7.2: Security capabilities at a glance

Service	Responsibility	Common basis
Authentication	Proves identity and issues tokens; the FHIR server never sees passwords	OAuth 2.0 / OpenID Connect
Authorization	Scopes determine what a token may do	SMART App Launch / Backend Services
Consent	Defines, stores, and enforces member choices	FHIR Consent
Audit	Records access in append-only form	AuditEvent, IHE BALP[40]
API gateway	Routing, single sign-on at the edge, throttling, caching, logging	Gateway products
Signature	Authenticity, integrity, non-repudiation	Provenance[39] and PKI

7.4 Governance Essentials

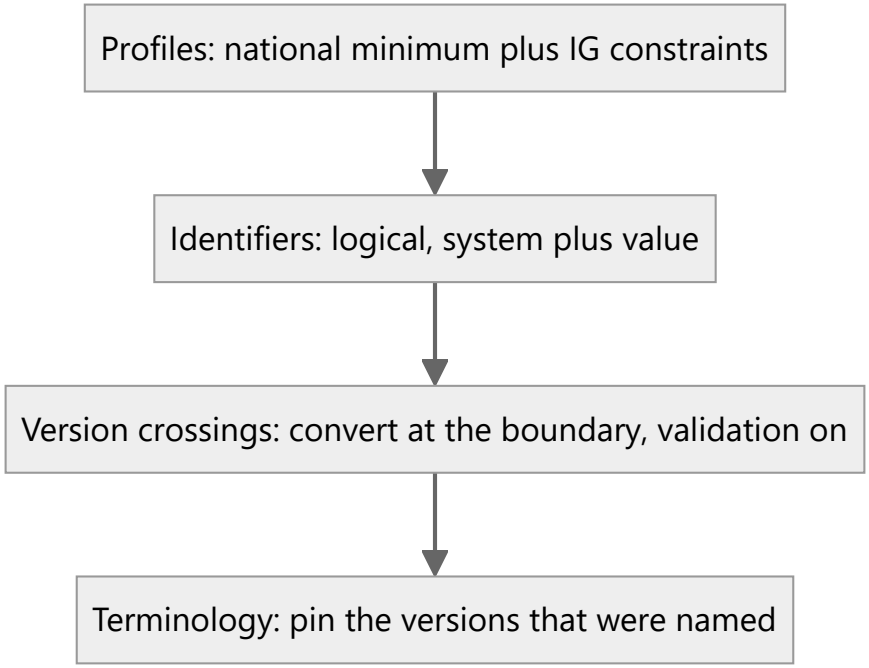

Figure 7.2: Governance essentials

- **Profiles.** Use US Core as the national baseline and constrain further per the IGs; do not invent a private schema.
- **Identifiers.** Reference across organizations with logical identifiers (`system` plus `value`, such as a member ID with its namespace URI), never with a server's internal relative IDs — the moment data migrates, relative references break.
- **Version crossings.** Convert versions at the boundary and keep validation on (consistent with Section 4.4).
- **Terminology.** Pin the code system versions named by the regulation and the IGs. The terminology chain runs code system → value set → element binding, and ConceptMap translates between code systems.[41]

Pragmatic governance trade-offs

- A source system cannot capture a “required” field before go-live: relax the cardinality to optional, mark it Must Support, and **keep validation on** — do not turn validation off to let data through.
- When to return 200: **only after the data is actually committed** — which is exactly why Chapter 12 returns success only after release.
- Non-conformant payloads: validate and reject them at the boundary. An “informative” guide will not reject anything for you.

7.5 The Component Checklist: Add Components by Constraint

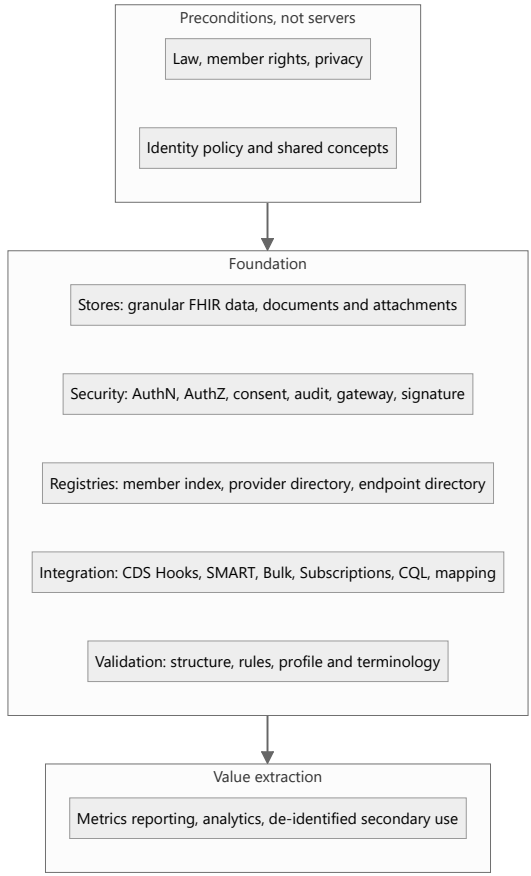

Figure 7.3: A payer-side interoperability component catalog (preconditions → foundation → value extraction)

How to use it: go through the list one category at a time, and **add a component to the design only when a specific constraint actually needs it.** Preconditions are not software — a legal basis and an identity policy cannot be invented after go-live, and consent software can only enforce rules; it cannot decide them for the organization.

Chapter summary: Every access must decide identity, permission, and consent, and leave an audit record. Use App Launch when a person is at the keyboard and Backend Services for unattended jobs. Security rests on several capabilities working together; TLS only encrypts the transport. Governance rests on profiles, logical identifiers, version conversion at the boundary, and pinned terminology versions. Add components one constraint at a time, and avoid do-everything designs.

Part II — Engineering Delivery

This part answers “how do we build it right”: using the PA API’s submission path as the example, it covers interface layering and the status-code contract (Chapters 8–9), validation and mapping (Chapters 10–11), safe writes into legacy systems and rule triggering (Chapters 12–13), and conformance evidence and delivery (Chapters 14–15).

8. The Interface Layer: How to Layer `$submit`

> **Bottom line:** Each layer does exactly one thing (validate → map → apply defaults → write → respond). Validation comes first, and a failure rejects the whole request with zero side effects. The response is always a FHIR resource — the business resource on success, an OperationOutcome on failure, never custom JSON.

8.1 The Layers

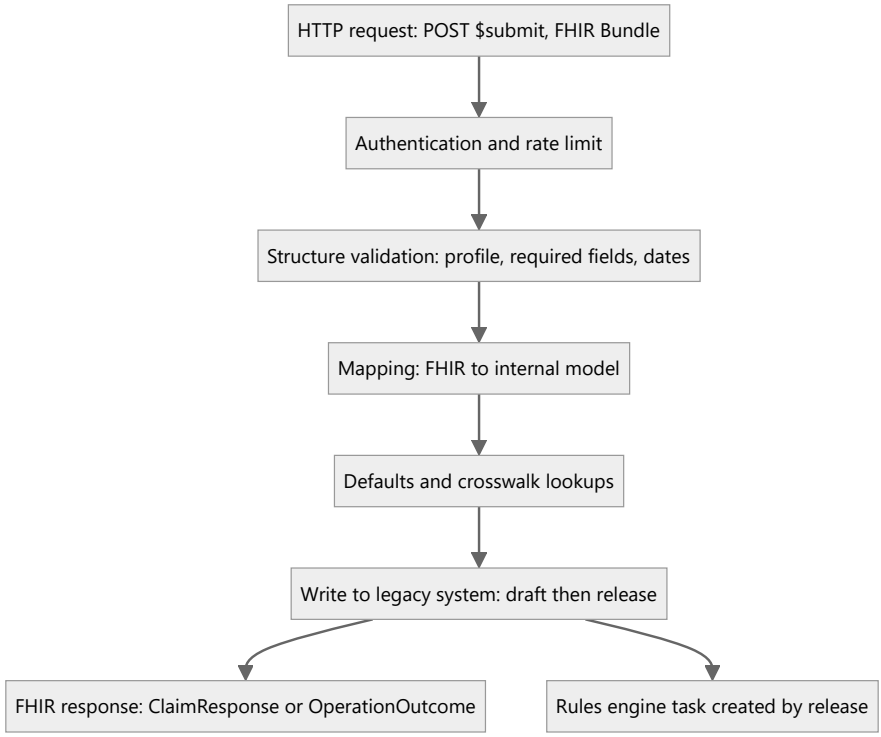

Figure 8.1: How a `$submit` request is processed (validate → map → defaults → write → respond)

Principles:

- Each layer does one thing: the validation layer does not map, and the mapping layer does not write.
- Validation comes first. When it fails, the whole request is rejected with no side effects.
- The response is always a FHIR resource (the business resource on success, OperationOutcome on failure). Never return custom JSON.[25] [29]

8.2 Handling the Bundle

- Validate every resource in the Bundle (Claim, Patient, Practitioner/Organization, Coverage, and so on) and confirm that all references resolve.[13][28]
- Identifiers (on the Bundle and the Claim) must be unique, and resubmission must have well-defined behavior (see idempotency in Chapter 12).
- Never trust a "status" field sent by the submitter; the system's own rules decide the initial status (see Chapter 11).

8.3 Code Skeleton (Illustrative)

In code, the layering becomes a pipeline with exactly one side-effect point. The C# below is illustrative; logging, dependency injection, and exception details are omitted.

```
public sealed class SubmitPipeline
{
    private readonly IPasBundleValidator _validator; // structure and profile
validation
    private readonly IClaimMapper _mapper;           // FHIR -> domain model, a
pure function
    private readonly IDefaultsProvider _defaults;    // configured defaults and
lookups, validated at startup
    private readonly ILegacyWriter _writer;          // the only side effect:
draft write + release
    private readonly IResponseBuilder _responses;    // ClaimResponse or
OperationOutcome

    public async Task<FhirResult> HandleAsync(Bundle request, CancellationToken
ct)
    {
        var issues = _validator.Validate(request);
        if (issues.HasErrors)
            return _responses.BadRequest(issues);       // 400, zero side
effects

        var draft = _mapper.Map(request);                // no database access;
unit-testable
        _defaults.Apply(draft);

        var result = await _writer.WriteDraftThenReleaseAsync(draft, ct);
        return _responses.FromWriteResult(result);       // 200 / 409 / 503 ...
    }
}
```

Engineering notes:

- Keep the mapper a pure function (Bundle in, domain object out) so mapping rules can be unit-tested without a database.
- The writer is the only component that touches the legacy system; transaction boundaries, retries, and quarantine all live there (Chapter 12).
- In a facade architecture, the validator and mapper are on the hot path: avoid per-item database lookups during mapping, and load lookup tables into memory at startup.

Chapter summary: `$submit` is a one-way pipeline: each layer does one thing, validation comes first and fails with zero side effects, and the

response is always a FHIR resource. Validate every resource and reference in the Bundle, keep identifiers unique, and never trust a submitted status.

9. The Status-Code Standard (An External Contract)

> **Bottom line:** Status codes are an external contract — define the table before you write code. A 4xx means “the request is wrong; it was rejected in full; do not retry.” A 5xx means “the problem is ours; it is safe to retry.” One table governs the code, the acceptance criteria, and the responses partners see.

Define first, then code; one table governs everything.

Table 9.1: The status-code contract (meaning / retry / typical cases)[25][30][31]

Status	Meaning	Retry?	Typical cases
200	Success (including a clearly defined “partial load”)	No	Normal processing; partial data loads handled as the table defines
200 (with warning)	Success, but placeholder data was used	No	Provider not found, so a placeholder value agreed in the contract was used; a warning-severity OperationOutcome in the response explains it
400	Invalid request format or field	No — fix and resend	Missing or invalid date; profile violation
401 / 403 / 429	Not authenticated / not authorized / rate limited	Depends	Token missing or expired; insufficient permissions; over the limit
404	Referenced object does not exist	No	Member or authorization not found
409	Conflict	No	Duplicate submission with conflicting content
415	Unsupported media type	No	Content-Type is not FHIR JSON
422	Semantically unprocessable	No	Well formed, but not valid as business
503 / 504	Downstream unavailable / timeout	Yes	Legacy system or network temporarily unavailable
500	Our own error	Yes (with care)	Unexpected exception; validation was bypassed

Rules:

- Every 4xx is all-or-nothing: you may not write half a request and then return a 4xx.

- Do not use 201 to say "succeeded, with reservations." In HTTP, 201 means only that a new resource was created;[30] convey business reservations with a warning in the OperationOutcome.
- Every 5xx must be safe to retry: resending the same request must produce a safe result (see Chapter 12).
- 401 versus 403: 401 means "not authenticated, or the token is invalid"; 403 means "authenticated, but not permitted."[30] When troubleshooting a 401, check the token before suspecting permissions.

9.1 Mapping Status Codes to OperationOutcome

The status code tells the caller whether it can retry; the OperationOutcome tells the caller what went wrong. Take `issue.code` from FHIR's IssueType value set and put it in the contract table alongside the status code:[29]

Table 9.2: Status codes and OperationOutcome

Status	`issue.severity`	`issue.code`
400	error	`required` / `value` / `invalid` / `structure`
404	error	`not-found`
409	error	`conflict` / `duplicate`
422	error	`business-rule`
429	error	`throttled`
500	fatal	`exception`
503 / 504	error	`transient` / `timeout`

A sample 400 response — it identifies the exact field without echoing sensitive values:

```
{
  "resourceType": "OperationOutcome",
  "issue": [
    {
      "severity": "error",
      "code": "required",
      "details": { "text": "Service date is required on every item." },
      "expression": [ "Bundle.entry[0].resource.item[1].serviced" ]
    }
  ]
}
```

Chapter summary: Define the status-code table before writing code. A 4xx is all-or-nothing and must not be retried; a 5xx must be safe to retry. For a 401, check the token; for a 403, check permissions. The same table serves the code, the acceptance criteria, and partner-facing responses.

10. Validation and Fail-Fast

> **Bottom line:** Edge validation is the first line of defense (a failure returns 400 with an OperationOutcome). Internal assertions are the second (if validation is bypassed, throw and return 500). Never silently fill in a default and keep going, and never build a "fallback mode" — with healthcare data, a silent degradation is more dangerous than an error.

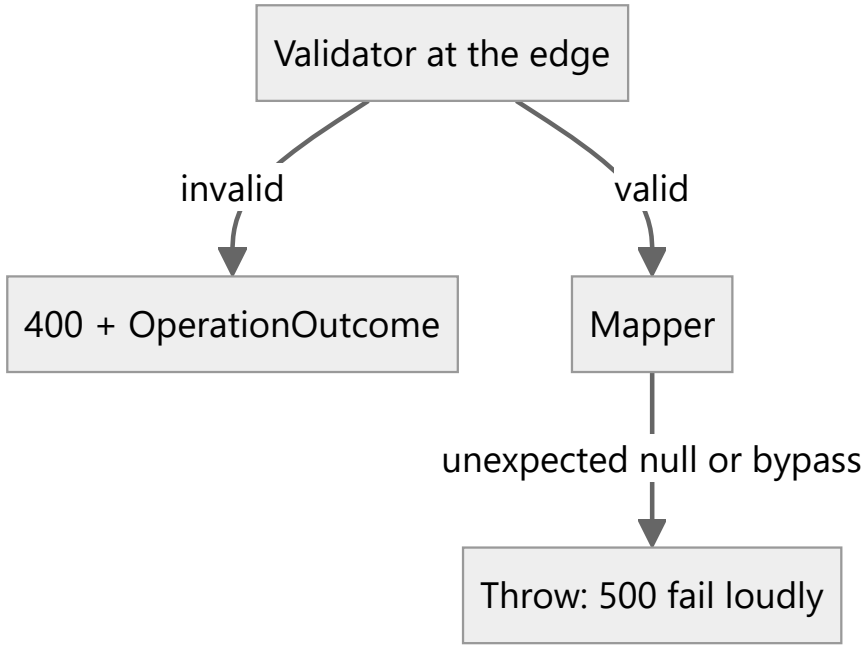

Figure 10.1: Two lines of defense — edge validation (400) and internal assertions (500)

- Edge validation is the first line of defense. On failure, return 400 and say exactly which field is wrong and why in the OperationOutcome.[29]

- Add internal assertions as well. If validation has been bypassed (for example, a caller skipped the validation layer), the code must throw (500) rather than quietly continue with a default.
- Why: with healthcare data, “quietly using a default date” produces an authorization that looks right but is wrong — far more dangerous than an error.
- Validate and load configuration dependencies (lookup tables, rule files) at startup. If something is missing, refuse to start, rather than discovering it at run time.
- Do not add a “fallback mode.” A fallback hides the problem and pushes the responsibility onto downstream testing.

10.1 Two Implementation Techniques (Illustrative)

Internal assertions. If a value is still null after validation has passed, some call path has bypassed the validation layer. Throw immediately rather than filling in a default.

```
public static DateOnly RequireServiceDate(DateOnly? value, string path) =>
    value ?? throw new InvariantViolationException(
        $"{path} passed edge validation but is null"); // maps to 500 and
raises an alert
```

Validate configuration at startup. Refuse to start when a lookup table is missing keys. In .NET, use options validation with `ValidateOnStart`:

```
builder.Services.AddOptions<CrosswalkOptions>()
    .Bind(builder.Configuration.GetSection("Crosswalk"))
    .Validate(o => o.RequiredKeys.All(o.Map.ContainsKey),
              "Crosswalk is missing required keys")
    .ValidateOnStart();
```

Chapter summary: Validate at the edge and assert inside: the two lines of defense map to 400 and 500. Validate configuration dependencies at startup and refuse to start if anything is missing. Never degrade silently, and never build a fallback mode.

11. Mapping, Defaults, and Business Rules

> **Bottom line:** Every internal field has exactly four legitimate sources — the payload, a configured default, a lookup table, or a rule-based derivation. Each rule has a single source and lives in configuration, not in code constants. Compare dates using the calendar date in the business time zone, and reject the whole request with a 400 if any line item has an invalid date.

11.1 The Four Sources

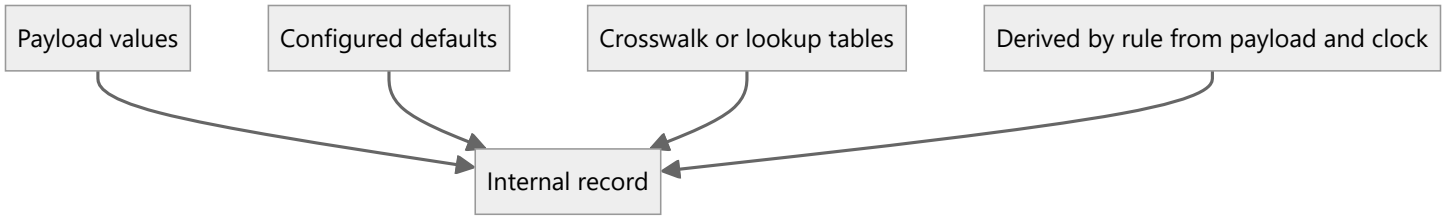

Figure 11.1: The four legitimate sources of an internal field

- From the payload: mapped directly.
- From a default: for example, the initial status. Kept in configuration and decided by the system; submitters cannot override it.
- From a lookup table: codes translated to internal keys. At startup, verify that every required key exists and is active.
- Derived by rule: for example, the request type.

11.2 Example: Deriving the Request Type from the Earliest Service Date (A Reusable Date-Rule Pattern)

The rule: take the earliest service start date across all line items and compare it with the calendar date, in the business time zone, at which the request was received. If the start date is on or after that date, the request is prospective; if it is earlier, the request is retrospective. The header and every line item get the same value.

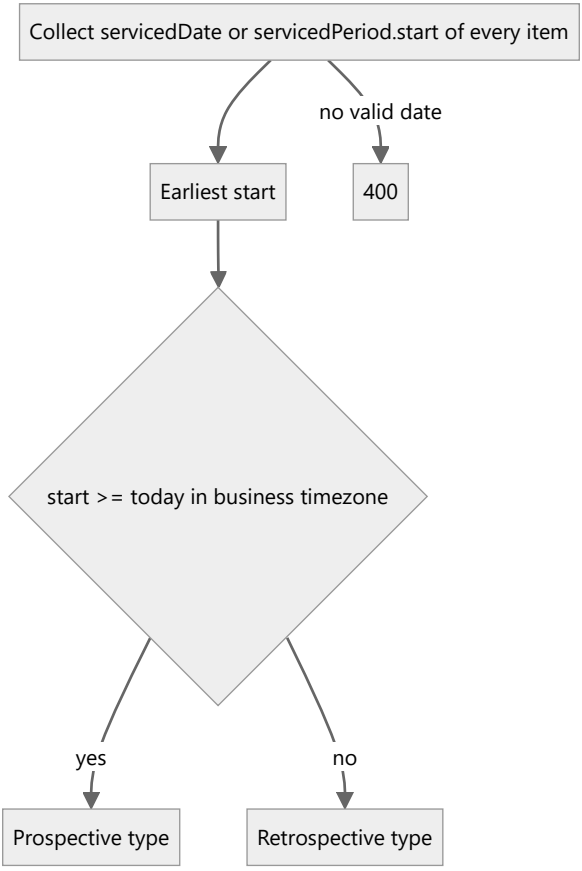

Figure 11.2: The date rule for deriving the request type (prospective or retrospective)

Details that are easy to get wrong:

- Compare against the calendar date in the business time zone — not UTC, and not the server's local date. A fixed-offset time zone (no daylight saving time) is easier to verify.
- For timestamps with a time-zone offset, take the calendar date as written; do not convert it.
- For a period, only the start counts; the end date plays no part.
- If any line item's date is missing or invalid, return 400 for the whole request.
- Write the same request-type value to the header and every line item so they can never disagree.

A C# sketch. Note that the item dates are the calendar dates as written in the payload; only the time of receipt is converted to the business time zone:

```
public static RequestType DeriveRequestType(
    IReadOnlyCollection<DateOnly> itemStartDates, // calendar date of each
item's servicedDate / servicedPeriod.start
    DateTimeOffset receivedAt,
    TimeZoneInfo businessZone)
{
    if (itemStartDates.Count == 0)
        throw new FhirValidationException("No valid service date"); // -> 400

    var earliest = itemStartDates.Min();
    var today = DateOnly.FromDateTime(
        TimeZoneInfo.ConvertTime(receivedAt, businessZone).DateTime);

    return earliest >= today ? RequestType.Prospective :
RequestType.Retrospective;
}
```

Passing the time of receipt in as a parameter (rather than reading the system clock inside the function) makes the boundary cases deterministic to test.

11.3 Initial Status

- The header status and its status detail are system defaults (for example, “Requested”); any status sent by the submitter is ignored.
- Whether line-item statuses must also be set depends on the acceptance criteria. If the criteria mention only the header, do not extend the change to the line items on your own.

Chapter summary: Every internal field must be able to say which of the four sources it came from. Each rule has one source and lives in configuration. Date comparisons are anchored to the calendar date in the business time zone, and a single bad date rejects the whole request with a 400. The system decides the initial status; whatever the submitter sends is ignored.

12. Writing to Legacy Systems: Draft, Release, Retry, Quarantine

> **Bottom line:** This is the most reusable chapter in the book. Legacy systems typically "fire downstream on save," so you must **write with a draft flag → release only after every write succeeds → retry the release a bounded number of times.** When a write fails partway, do not delete the draft; **quarantine it by renaming** so the idempotency key is freed and the client can safely resend with the same identifier.

12.1 The Problem

Legacy systems often trigger downstream work the moment a record is saved: the rules engine, work queues, notifications. If the API writes in several steps (header, line items, attachments) and fails partway, it leaves a half-written record — and the rules have already fired on it.

12.2 The Solution: A Draft Flag, Released Only After Everything Succeeds

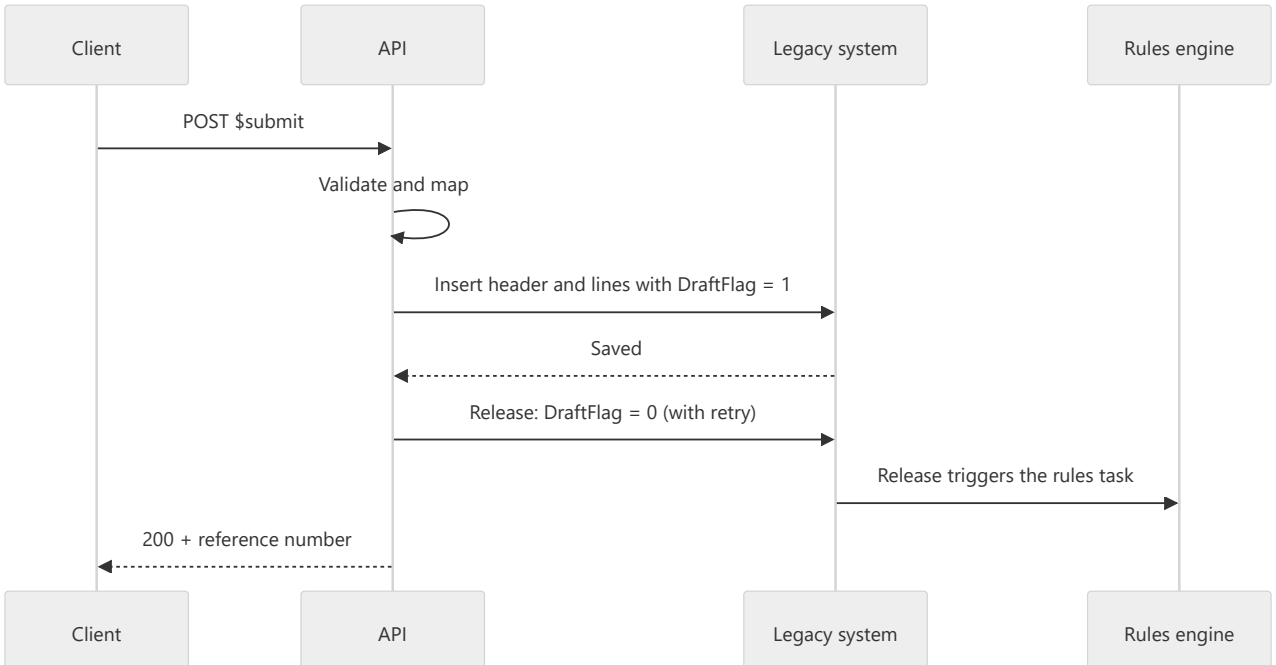

Figure 12.1: Draft write → everything succeeds → release triggers the rules (with bounded retries on release)

- Insert with the draft flag set; downstream rules do not fire.
- After every write succeeds, “release” the record (set the draft flag to 0). Only the release triggers the rules.
- Retry the release a bounded number of times with exponential backoff, because it is often a network call.[32]
- What to verify: downstream tasks triggered by release are usually created asynchronously, so allow time before you check; a failed draft must not produce any downstream task.

12.3 Partial Failure: Quarantine by Renaming

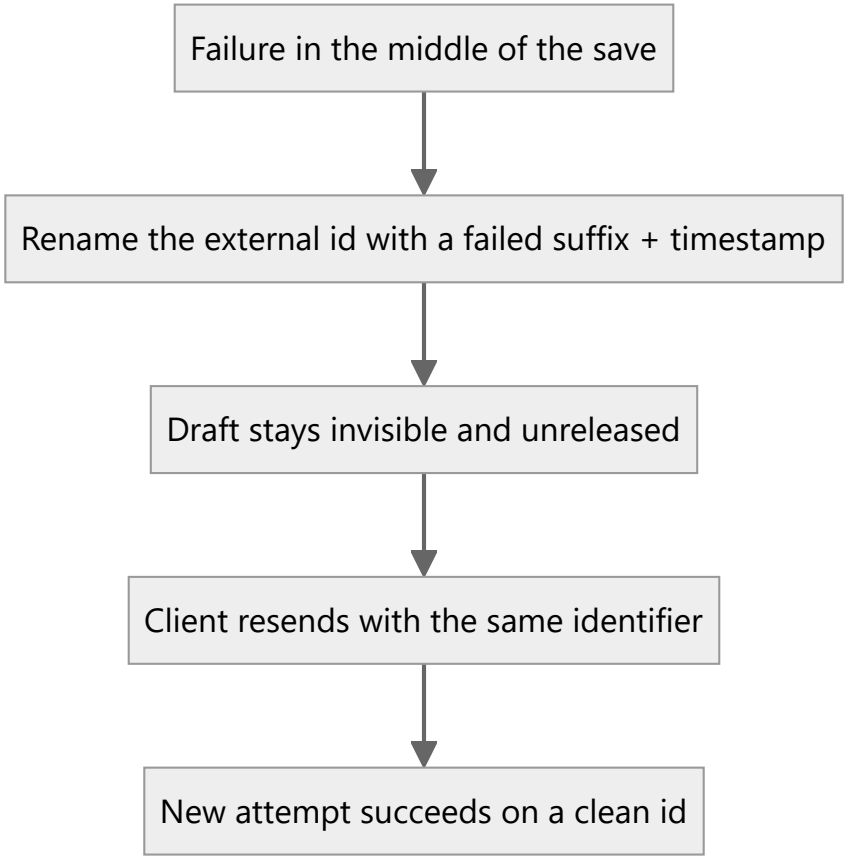

Figure 12.2: Partial failure → quarantine rename frees the idempotency key → client resends with the same identifier

- Do not delete the failed draft (keep it for audit), but rename it to free the original identifier, so that when the client resends with the same identifier it does not collide with an "already exists" record.
- Include a timestamp in the rename suffix, and respect the target field's length limit — truncate while keeping the value unique.

- Verify this in the test environment with fault injection. QA generally cannot reproduce it, so developers must do it themselves and attach the evidence.

12.4 Idempotency and Resubmission

- Use the external identifier as the idempotency key.[25]
- The convention: resending after success gets a well-defined "already exists" response (never a silent duplicate); resending after failure can succeed.
- Write this convention into the status-code table (the 409 case).

Implementation skeleton (illustrative):

```
public async Task<WriteResult> WriteDraftThenReleaseAsync(AuthDraft d,
CancellationToken ct)
{
    if (await _repo.ExistsReleasedAsync(d.ExternalId, ct))
        return WriteResult.Duplicate(d.ExternalId);                 // 409, or
return the existing result

    var id = await _repo.InsertDraftAsync(d, ct);                    //
DraftFlag = 1; nothing downstream fires
    try
    {
        await _repo.InsertLinesAsync(id, d.Lines, ct);
        await _retry.ExecuteAsync(() => _repo.ReleaseAsync(id, ct)); //
DraftFlag = 0; rules fire
        return WriteResult.Created(id);
    }
    catch (Exception ex) when (ex is not OperationCanceledException)
    {
        var quarantineId = QuarantineName(d.ExternalId, DateTime.UtcNow,
maxLength: 50);
        await _repo.QuarantineAsync(id, quarantineId, ct);           // keep
for audit; free the idempotency key
        return WriteResult.Transient(ex);                            // 503,
safe to retry
    }
}
```

The same semantics in hybrid or native architectures: when the write target is a FHIR server, you do not need your own idempotency table.

Use a conditional create with the header `If-None-Exist: identifier=system|value`, and the server guarantees that a given identifier is created only once.[25] For all-or-nothing writes across several resources, use a transaction Bundle (Section 6.3).

12.5 Common Pitfalls

- The legacy system's own rules may modify data after release (for example, filling in or clearing certain fields according to its own logic). That is the legacy system's rule, not an API bug; note it on the work item so QA does not log a false defect.
- If the legacy system provides a change data capture (CDC) or audit table, it shows who changed a field and when, which lets you tell "we wrote it wrong" from "their rules changed it."

Chapter summary: Write as a draft so nothing fires downstream, release only after everything succeeds, and retry the release a bounded number of times. Quarantine a failed draft by renaming it to free the idempotency key, and combine that with an idempotency convention on the external identifier, so that "a failure leaves no half-written record, and a resend never collides." Expect legacy systems to modify data after release, and use CDC to establish who changed what.

13. Triggering the Rules Engine

> **Bottom line:** The acceptance criterion is "a record created through the API must trigger the rules exactly as a record keyed in through the user interface does — and exactly once." The evidence comes from comparing the task table. **If you have not compared the outcomes, write "not verified," not "passed."**

- Goal: a record created through the API must trigger the rules just as a record entered by hand in the user interface does.
- How to gather evidence: query the task table where the object identifier equals the authorization identifier and the type is a rules task; note when the task appears; for failed drafts, confirm there is no task.
- Before and after: run the same samples before and after the change and compare the number of tasks in the task table; the goal is exactly one task per record, triggered once.
- Verification not yet done (state it honestly): whether the rules' *outcome* matches the outcome for the same data entered by hand requires a manual sample with identical data. If you have not done it, write "not verified," not "passed."

Chapter summary: Prove that triggering is correct by comparing task-table evidence (count, timing, and zero tasks for failed drafts). If the triggered *outcome* has not been compared, mark it "not verified." This chapter models the standard way to state unfinished work honestly.

14. Conformance and Evidence

> **Bottom line:** A CapabilityStatement is only a system's self-description; whether it truly conforms takes evidence. Evidence comes from three places: **layered validation** (structure → rules → profile and terminology), **public test kits** (such as the Inferno Da Vinci test kits), and **positive and negative evidence for every acceptance criterion.** Nothing goes into the Gap Document as "Met" without evidence, and anything not yet verified is marked "not verified."

14.1 Three Depths of Validation

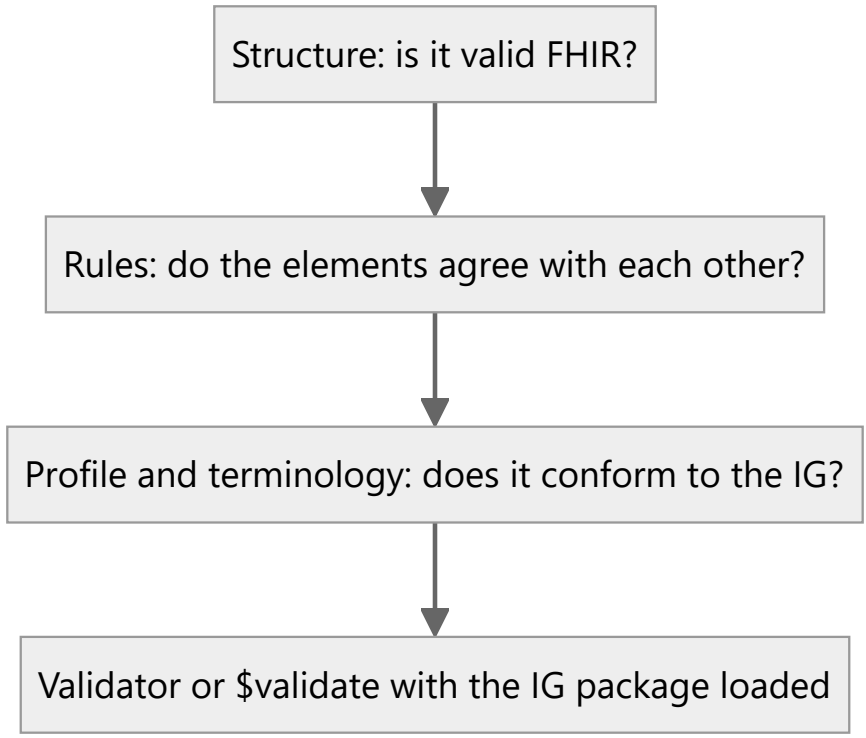

Figure 14.1: The three depths of validation[42]

- Structural validation answers “is this valid FHIR?”; profile validation answers “does this conform to the IG?” Neither substitutes for the other.
- The validator must load the same IG package version your partner uses. A version mismatch is the leading cause of “the data looks fine but fails validation” (Section 4.4).
- Keep validation on at the production boundary, and return an OperationOutcome on failure (Chapter 10).

14.2 Levels of Evidence

Table 14.1: Levels of evidence

Level	Question answered	Typical means
Claim	What do we say we support?	CapabilityStatement[27]
Conformance	Do we meet the IG's SHALL statements?	Public test kits, such as the Inferno PAS / CRD / DTR kits[43]
Contract	Do status codes, errors, and idempotency match the contract?	Contract tests written from the Chapter 9 status-code table
Business	Does each acceptance criterion's input produce the expected result?	Positive cases, negative cases (proving zero side effects), boundary cases
Resilience	What does the system leave behind after a partial failure?	Fault injection (Section 12.3)
Interoperability	Does it work end to end with real partners?	Partner testing and Connectathon-style joint testing

14.3 The Acceptance-Criteria Matrix

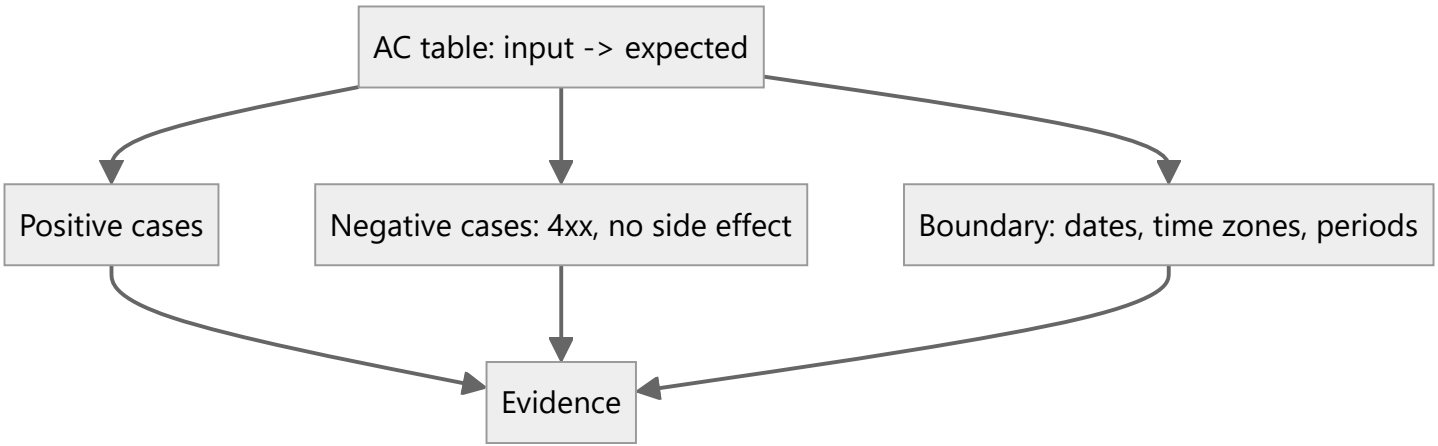

Figure 14.2: Each acceptance criterion expands into positive, negative, and boundary cases, all leading to evidence

- Write each acceptance criterion as one row, “input → expected result,” including negative cases; a negative case must prove that no record was created.
- Compute “today” dynamically in the business time zone in date-based tests, or they will fail tomorrow. Put a timestamp in identifiers to avoid collisions with historical data.
- For every case, keep the request, the response, the returned reference number, and the actual values in the database. Database checks are read-only, and list the fields you checked.
- Tokens and passwords never go into any document or script; pass them in through environment variables only.

A negative test (xUnit, illustrative) — asserting the status code, the response type, and zero side effects together:

```
[Fact]
public async Task Submit_without_service_date_returns_400_and_writes_nothing()
{
    var bundle = TestBundles.WithItemMissingServiceDate(externalId:
Unique.Id());

    var response = await _client.PostFhirAsync("Claim/$submit", bundle);

    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
    Assert.Equal("OperationOutcome", (await
response.ReadFhirAsync()).TypeName);
    Assert.False(await
_db.AnyAuthorizationWithExternalIdAsync(bundle.ExternalId()));
}
```

Chapter summary: A capability statement needs evidence behind it. Evidence comes from layered validation, public test kits, and positive and negative evidence for every acceptance criterion, covering contract, resilience, and interoperability. Evidence decides whether a requirement can be marked “Met” in the Gap Document.

15. Delivery Governance: Scope, Definition of Done, and How to Report Progress

> **Bottom line:** Team structures and processes vary from one organization to the next, so this chapter keeps only three principles that hold everywhere: **scope is what the written acceptance criteria say; "done" is what the evidence shows; progress uses the Gap Document as its denominator and is reported by workstream.**

15.1 Scope

- Scope = the written acceptance criteria + later written answers from the requirements owner, with newer answers superseding older ones. Add nothing that is not written down.
- Implement the IG's SHALL statements first; decide SHOULD and MAY statements by business value and record the rationale (Section 3.3).

15.2 Definition of Done

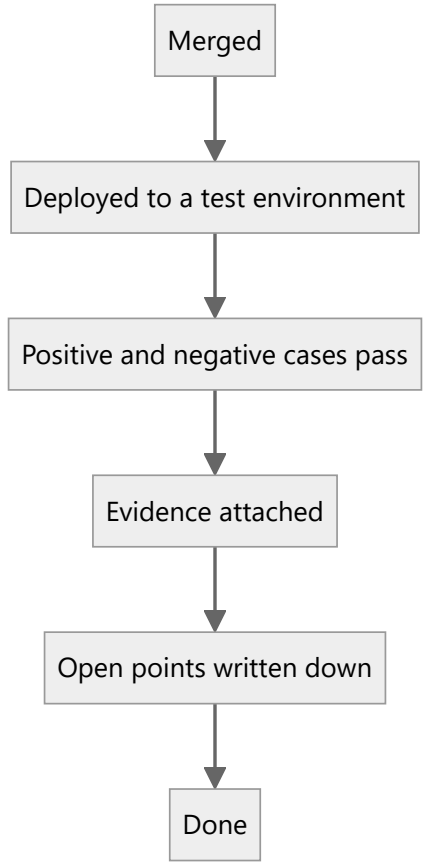

Figure 15.1: Definition of done — based on evidence, not verbal confirmation

15.3 How to Report Progress

- **Always report two numbers together:** requirements met under the rule (from the Gap Document: met / applicable — a hard number) and business capability coverage (an estimate, with its method stated), both dated.

- **Report by workstream:** Prior Authorization (every CRD, DTR, and PAS interaction), the Patient Access enhancements, Provider Access, Payer-to-Payer, and the operational provisions each get their own row (Table 6.1). Never report only the interface your team owns.
- **Break out external dependencies:** testing with partners, EHRs, and other payers is the most common blocker, and it only becomes visible when it has its own column.
- **Use regulatory dates as milestones:** draw a timeline with one swim lane per workstream, and put January 1, 2026 and January 1, 2027 on the same chart.

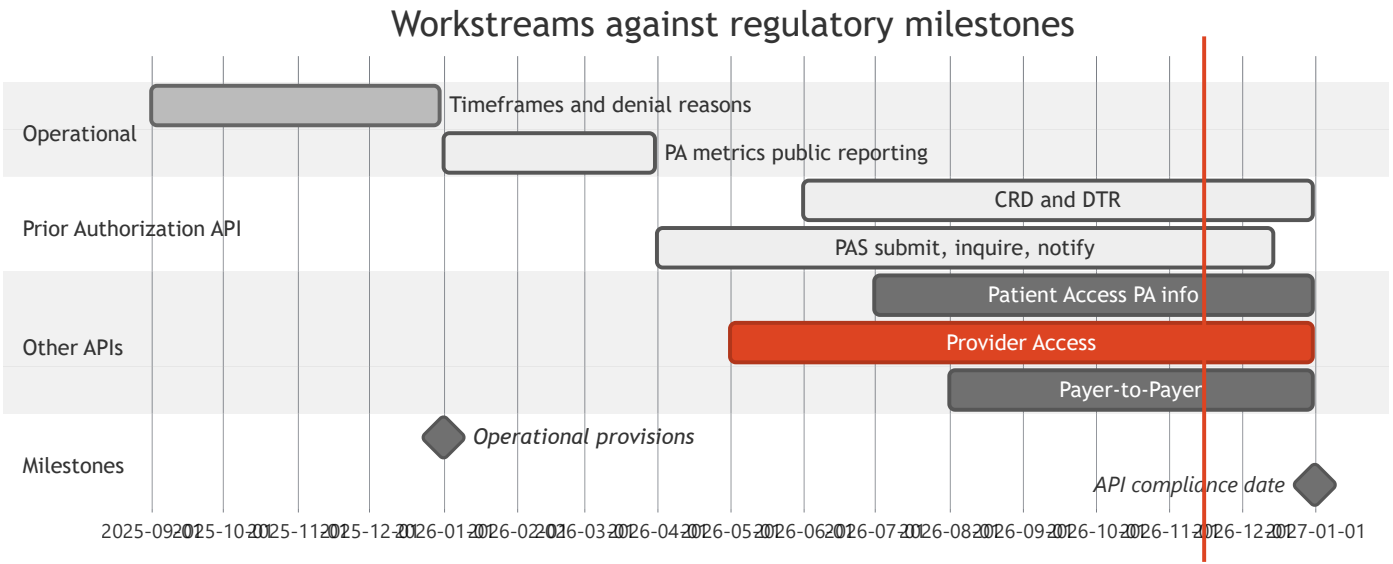

Figure 15.2: A timeline with one swim lane per workstream (illustrative data; regulatory dates as milestones)

Chapter summary: Scope follows the written acceptance criteria, “done” follows the evidence, and progress follows the Gap Document across every workstream. Leave the remaining process details to each organization’s own way of working.

Part III — Synthesis

This part turns the first two parts into tools you can use: a catalog of anti-patterns (Chapter 16), architecture review questions (Chapter 17), and an end-to-end path (Chapter 18), followed by conclusions and outlook (Chapter 19).

16. Anti-Pattern Catalog

> **Bottom line:** The twelve anti-patterns below are the mirror image of the practices in this book, grouped into six categories: measurement, architecture, exchange, trust, writes, and evidence. During a design review, go through them category by category; if one applies, go back to the chapter it points to.

Measurement and scope

1. Reporting the completion ratio of self-defined business rules as a compliance percentage (Section 2.2).
2. Reporting progress on a single interface while leaving out the other workstreams and the operational provisions (Section 15.3).

Architecture decisions

3. The do-everything design: answering one compliance gap with a full platform, a data lake, and an AI assistant (Section 6.1).
4. Choosing a server product before the requirements and constraints are clear, so the product's features shape the architecture (Section 6.1).
5. Expecting an API gateway to solve back-end performance — improve the read path instead (Section 6.3).

Exchange

6. Treating subscriptions as a replayable message queue, or treating a batch as a transaction (Section 6.3).
7. Referencing across systems with a server's internal relative IDs or `_id` (Section 7.4).

Trust and governance

8. Assuming TLS alone makes a system secure; scattering consent rules across every interface (Sections 7.1 and 7.3).

9. Turning validation off to “go live first,” instead of relaxing cardinality and keeping validation on (Section 7.4).

Writes and failures

10. Writing to a legacy system in multiple steps with no draft-and-release and no failure quarantine; silently substituting a default after validation fails (Chapters 10 and 12).

Evidence

11. Accepting a CapabilityStatement as evidence of conformance, or testing only positive cases without testing “zero records after a 4xx” (Chapter 14).
12. Writing “passed” for something that was never verified; leaving tokens or passwords in documents, scripts, or chat.

Chapter summary: The twelve anti-patterns fall into six categories — measurement, architecture, exchange, trust, writes, and evidence — and each one points back to the corresponding practice in the book. Checking them category by category during a review is far cheaper than troubleshooting after the fact.

17. Architecture Review Questions

> **Bottom line:** A review of an interoperability design pursues only three things — **where each requirement comes from, where the evidence is, and what the system leaves behind when it fails.** The questions below expand those three across the architecture domains.

Table 17.1: Architecture review questions

Domain	Questions
Scope and measurement	Where does each requirement come from — the rule text, an IG SHALL, a contract, or our own assumption? What is the denominator of the compliance percentage, and can someone else recompute it?
Architecture pattern	Where is the system of record? Why facade (or hybrid)? Are the performance limits recorded as known limitations?
Mechanism selection	Which single mechanism serves each action? Is any component there without a constraint that requires it?
Exchange	What is this workflow's time horizon? Which Bundle type and which search approach does it use?
Trust	How does every access decide identity, permission, and consent? Where is consent decided? Can the audit trail answer "who accessed what, when, and why"?
Governance	Are cross-organization references logical identifiers? Are the IG, terminology, and FHIR versions pinned and recorded?
Failure and retry	What does the system leave behind after a partial failure? What happens if the same request is sent twice?
Evidence	Which requirements have conformance tests and positive and negative evidence? Which are not yet verified, and where is that written down?

Chapter summary: The review questions are organized by domain, but they all come back to three things: where requirements come from, where the evidence is, and what failure leaves behind.

18. The End-to-End Path

> **Bottom line:** Getting from regulation to production is a ten-step path. The first five steps are architecture (scope, the Gap Document, architecture pattern, mechanisms, and trust and governance); the last five are engineering (contract, validation and mapping, safe writes, conformance evidence, and delivery).

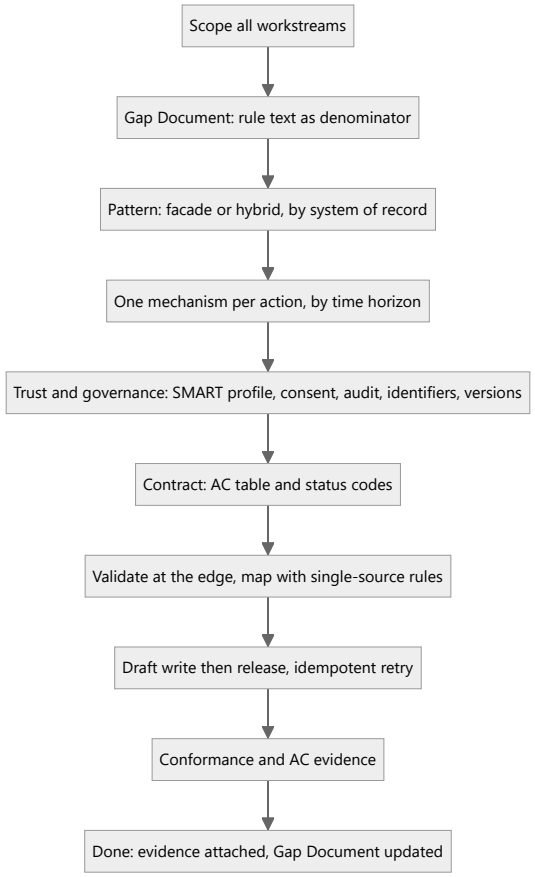

Figure 18.1: The end-to-end path (five architecture steps, then five engineering steps)

Chapter summary: Architecture first, then engineering. Once scope, the denominator, the pattern, the mechanisms, and trust and governance are settled, move on to the contract, validation, writes, and evidence. Every step's output feeds back into the Gap Document.

19. Conclusions and Outlook

> **Bottom line:** The whole book compresses into one sentence — **measure compliance against the rule text, choose the architecture by where the system of record lives, pick the exchange mechanism by time horizon, write with draft-then-release, and count work as done only when there is evidence.** The ten engineering conclusions and six architecture conclusions below unpack that sentence, and each one is grounded in the chapters of this book.

19.1 Key Conclusions

Ten engineering conclusions

1. Eighty percent of the difficulty in an interoperability project lies not in FHIR itself but in "writing to legacy systems" and "the boundaries of the requirements."
2. A compliance percentage can only be counted against the requirements in the rule text, never against rules you set yourself. Rules you set yourself can change at any time, which makes the number meaningless.
3. Status codes are an external contract: a 4xx means "the request has a problem; it was rejected in full; do not retry," and a 5xx means "the problem is ours; you can retry." Once defined, write them into one table, and use that table for the code, the acceptance criteria, and the responses.
4. Writes to a legacy system must have two phases, "draft + release." Otherwise a failure partway through leaves a half-written record, and the downstream rules engine fires on it.
5. When validation fails, fail loudly (fail-fast); do not degrade silently in the name of "availability." In healthcare software, a silent degradation is

more dangerous than an error.

6. Defaults, lookup tables, and date rules live in configuration, not in code constants; every rule has exactly one source.
7. Before development starts, write the acceptance criteria (AC) as a table — one input and one expected output per row, including negative cases.
8. In the test environment, run the tests QA would run yourself first, attach the evidence to the work item, and only then say "done."
9. The definition of "done" for a requirement: merged, deployed, every positive and negative sample passing in the test environment, evidence attached, and development tasks and hours closed out.
10. A retrospective always comes down to three questions: Who defined the requirement? Where is the evidence? What does the system leave behind when it fails?

Six architecture conclusions

11. **CMS-0057-F is more than a `$submit`.** It includes the PA API's discovery of authorization requirements, documentation requirements, submission, inquiry, and status notifications; the PA information added to Patient Access; the new Provider Access and Payer-to-Payer APIs; and the decision timeframes, specific denial reasons, and metrics reporting that took effect on January 1, 2026. Leave any of them out and the compliance number is inflated (Chapters 2 and 6).
12. **One compliance gap is best served by a well-scoped API rather than a platform rebuild.** Start existing systems on a facade and upgrade to hybrid once the API sees real use; migration only works in one direction, facade → hybrid → FHIR-native (Chapters 5 and 6).
13. **The exchange mechanism is decided by how fast the data is needed.** Use REST when a clinician is waiting, subscriptions for status changes, bulk export for attributed populations and plan switches, and an analytics store for metrics reporting. When an interface is slow, an API gateway will not help; build a copy of the data (Chapter 6).

14. **Resolve conflicting standards by layer, and let the ecosystem choose the FHIR version.** Take the strictest rule for privacy, follow the regulation for obligations, the named standard for format, and the IG's SHALL statements for detail. As of November 2026, use R4; put version conversion in an edge adapter, and date the conclusion (Chapters 3 and 4).
15. **Every read or write of member data must answer three questions and leave an audit record.** Who is this? Do their role and treatment relationship allow it? Has the member approved this purpose? Use SMART App Launch when a person is at the keyboard and Backend Services for batch system jobs; TLS only encrypts the transport; these decisions are made by other security capabilities (Chapter 7).
16. **Reference across organizations only with logical identifiers, never turn validation off, and prove conformance with evidence.** Use member IDs in `system|value` form rather than a server's internal IDs; when a source system lacks a field, relax the cardinality and keep validation on; a CapabilityStatement is only a self-description, and conformance is shown by validation, public test kits, and positive and negative evidence (Chapters 7 and 14).

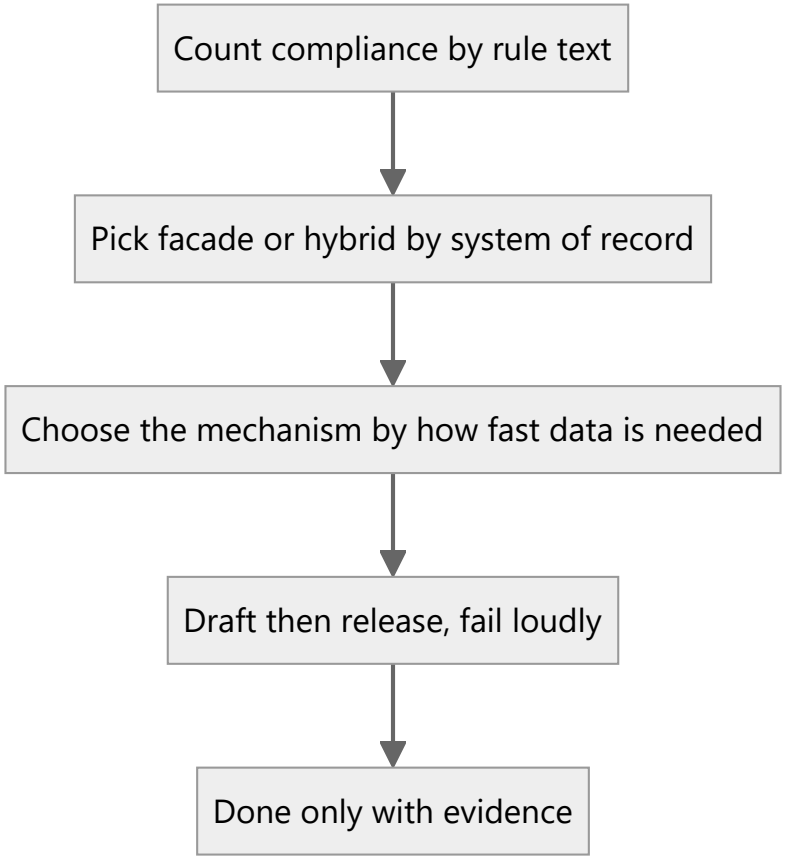

Figure 19.1: The five key decisions that run through the book

19.2 Outlook

The following are directions visible at the time of writing (November 15, 2026). They are personal judgments, not predictions; for specific rules and versions, defer to the latest official publications.

- **From "having an API" to "using it":** after January 1, 2027, regulatory attention will shift from "is the API live?" to "is it actually used, are decisions on time, and are denial reasons specific?" Metrics reporting will make the gaps publicly visible, and the Gap Document

must change from a one-time deliverable into a continuously maintained document.[2][3]

- **From facade toward hybrid:** once the APIs see real use, the performance limits of a facade will show. Domains with usage data to justify it will gradually get operational copies, and the architecture will shift from "meeting compliance" to "supporting the business" (Chapter 5).
- **The provider side catches up:** a payer's PA API is only half of the path. CRD, DTR, and PAS capabilities and certification requirements on the EHR side are advancing, and end-to-end electronic prior authorization will mature only when both sides are in place.[5][14][15] Integration testing will expand from "the gateway" to "real EHRs."
- **Standards keep evolving:** US Core and SMART versions roll forward under 45 CFR 170.215, and PAS and other IGs keep publishing new versions.[8][13] Adoption of FHIR R5 or R6 should be re-evaluated using the triggers in Section 4.5. The value of edge adapters and pinned dependencies will only grow.
- **Automation and AI enter utilization review:** beyond rules engines, automation and AI-assisted decisions will appear more often in prior authorization. The discipline in this book applies just the same — decisions must trace to a specific basis, failures must be loud, and nothing unverified gets marked as passed. With AI output, keeping evidence and preserving the right to appeal matter even more.

19.3 Putting It into Practice

- **In the first week of a project:** produce two documents — a first draft of the Gap Document (covering every workstream, with a citation and compliance date for each requirement) and a "action → mechanism → standard" table in the form of Table 6.1.
- **At the architecture review:** use the decision tree in Figure 5.1 to settle on facade or hybrid, state where the system of record lives, and

record known performance limits (such as P95 response time) in the Gap Document.

- **Before connecting a legacy system:** answer three questions first — does saving trigger anything downstream? Is there a draft or status flag you can use? Which external identifier is the idempotency key?
- **Before any interface goes live:** the status-code table and the OperationOutcome mapping are final, positive and negative evidence and public test-kit results are attached, and unverified items are listed.
- **When the regulation or an IG changes:** revisit the version decision using the triggers in Section 4.5, and update the Gap Document's date and versions accordingly.

Chapter summary: The ten engineering conclusions are about getting one interface right; the six architecture conclusions are about getting the whole design right. Together they come down to five judgments — measure compliance against the rule text, choose the architecture by where the system of record lives, pick the exchange mechanism by time horizon, write with draft-then-release, and count work as done only when there is evidence. Regulations, standards, and IGs will keep changing; these five judgments will not.

Stay in Touch

The official source of this book is **northpine.dev**. Subscribe at **northpine.dev/newsletter** to receive occasional update notifications.

Errata, questions, and feedback are welcome at hello@northpine.dev. You can also follow the author here:

- Website: https://northpine.dev
- LinkedIn: https://www.linkedin.com/in/rexfan18/
- GitHub: https://github.com/memoryfraction/HealthData-Interoperability-Csharp
- Medium: https://medium.com/@rex.fan18

Feel free to share this book with colleagues working on interoperability and compliance — please keep it complete and credit the source.

References

All sources below are public. They were retrieved and checked on **November 15, 2026** (the version conclusions in Chapter 4 were checked on November 10, 2026). Regulations, standards, and IGs are updated over time; when citing them, use the publisher's latest version. Superscript numbers in the text ([n]) refer to the entries below.

Regulations and regulators (CMS / HHS)

1. Centers for Medicare & Medicaid Services. *Medicare and Medicaid Programs; Patient Protection and Affordable Care Act; Advancing Interoperability and Improving Prior Authorization Processes…* (CMS-0057-F). Federal Register, 89 FR 8758, FR Doc 2024-00895, 2024-02-08. https://www.federalregister.gov/documents/2024/02/08/2024-00895/medicare-and-medicaid-programs-patient-protection-and-affordable-care-act-advancing-interoperability
2. CMS. *CMS Interoperability and Prior Authorization Final Rule CMS-0057-F* (Fact Sheet), 2024-01-17. https://www.cms.gov/newsroom/fact-sheets/cms-interoperability-and-prior-authorization-final-rule-cms-0057-f
3. CMS. *CMS Interoperability and Prior Authorization Final Rule (CMS-0057-F)* (policy page). https://www.cms.gov/priorities/key-initiatives/burden-reduction/interoperability/policies-and-regulations/cms-interoperability-and-prior-authorization-final-rule-cms-0057-f
4. CMS. *Interoperability Frequently Asked Questions*. https://www.cms.gov/priorities/key-initiatives/burden-reduction/interoperability/faqs
5. CMS. *Electronic Prior Authorization – Overview*. https://www.cms.gov/priorities/electronic-prior-authorization/overview

6. CMS. *Application Programming Interfaces (APIs) and Relevant Standards and Implementation Guides (IGs)*. https://www.cms.gov/priorities/key-initiatives/burden-reduction/implementation-guides-standards/application-programming-interfaces-apis-and-relevant-standards-and-implementation-guides-igs
7. CMS. *Interoperability and Patient Access Final Rule* (CMS-9115-F). Federal Register, 85 FR 25510, FR Doc 2020-05050, 2020-05-01. https://www.federalregister.gov/d/2020-05050
8. 45 CFR 170.215 – Application Programming Interface Standards. eCFR. https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-D/part-170/subpart-B/section-170.215

HL7 FHIR base standards and implementation guides

9. HL7. *FHIR R4 (v4.0.1)*. https://hl7.org/fhir/R4/
10. HL7. *US Core Implementation Guide*. https://hl7.org/fhir/us/core/
11. HL7. *SMART App Launch*. https://hl7.org/fhir/smart-app-launch/
12. HL7. *FHIR Bulk Data Access (Flat FHIR)*. https://hl7.org/fhir/uv/bulkdata/
13. HL7 Da Vinci. *Prior Authorization Support (PAS) IG*. https://hl7.org/fhir/us/davinci-pas/
14. HL7 Da Vinci. *Coverage Requirements Discovery (CRD) IG*. https://hl7.org/fhir/us/davinci-crd/
15. HL7 Da Vinci. *Documentation Templates and Rules (DTR) IG*. https://hl7.org/fhir/us/davinci-dtr/
16. HL7 Da Vinci. *Payer Data Exchange (PDex) IG*. https://hl7.org/fhir/us/davinci-pdex/
17. HL7 CARIN Alliance. *CARIN Consumer Directed Payer Data Exchange (CARIN IG for Blue Button)*. https://hl7.org/fhir/us/carin-bb/

Privacy, security, and transaction standards

18. HHS. *HIPAA Privacy Rule*. https://www.hhs.gov/hipaa/for-professionals/privacy/index.html
19. HHS. *HIPAA Security Rule*. https://www.hhs.gov/hipaa/for-professionals/security/index.html
20. HHS. *Minimum Necessary Requirement*. https://www.hhs.gov/hipaa/for-professionals/privacy/guidance/minimum-necessary-requirement/index.html
21. 42 CFR Part 2 – Confidentiality of Substance Use Disorder Patient Records. eCFR. https://www.ecfr.gov/current/title-42/chapter-I/subchapter-A/part-2
22. X12. *X12 Standards* (including the 278 Health Care Services Review transaction). https://x12.org/

FHIR specification details

23. HL7. *Subscriptions R5 Backport IG*. https://hl7.org/fhir/uv/subscriptions-backport/
24. HL7. *FHIR Publication (Version) History*. https://hl7.org/fhir/directory.html
25. HL7. *FHIR R4 – RESTful API (http)* (including the fhirVersion media-type parameter and conditional operations). https://hl7.org/fhir/R4/http.html
26. HL7. *FHIR R4 – Versions*. https://hl7.org/fhir/R4/versions.html
27. HL7. *FHIR R4 – CapabilityStatement*. https://hl7.org/fhir/R4/capabilitystatement.html
28. HL7. *FHIR R4 – Bundle*. https://hl7.org/fhir/R4/bundle.html
29. HL7. *FHIR R4 – OperationOutcome*. https://hl7.org/fhir/R4/operationoutcome.html

HTTP and engineering practice

30. Fielding, R., Nottingham, M., Reschke, J. *RFC 9110: HTTP Semantics*. IETF, 2022-06. https://www.rfc-editor.org/rfc/rfc9110
31. Nottingham, M., Fielding, R. *RFC 6585: Additional HTTP Status Codes* (including 429). IETF, 2012-04. https://www.rfc-editor.org/rfc/rfc6585
32. Brooker, M. *Exponential Backoff And Jitter*. AWS Architecture Blog, 2015-03-04 (updated 2023). https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

Architecture and industry literature

33. Devitt, D. *FHIR® Architecture Decisions: Selecting a FHIR server and choosing between facade, hybrid, and FHIR-native approaches* (free PDF / ePub). darrendevitt.com. https://darrendevitt.com/fhir-architecture-decisions-book/ (The facade / hybrid / FHIR-native definitions and performance analysis in Chapter 5 draw on this book.)

Interoperability architecture specifications

34. HL7. *CDS Hooks* (v2.0.1). https://cds-hooks.hl7.org/
35. HL7. *SMART App Launch – Backend Services*. https://hl7.org/fhir/smart-app-launch/backend-services.html
36. HL7. *FHIR R4 – AuditEvent*. https://hl7.org/fhir/R4/auditevent.html
37. HL7. *FHIR R4 – Consent*. https://hl7.org/fhir/R4/consent.html
38. HL7. *FHIR R4 – Subscription*. https://hl7.org/fhir/R4/subscription.html
39. HL7. *FHIR R4 – Provenance*. https://hl7.org/fhir/R4/provenance.html
40. IHE ITI. *Basic Audit Log Patterns (BALP)*. https://profiles.ihe.net/ITI/BALP/
41. HL7. *FHIR R4 – Terminology Module*. https://hl7.org/fhir/R4/terminology-module.html
42. HL7. *FHIR R4 – Validating Resources*. https://hl7.org/fhir/R4/validation.html

13. ASTP/ONC. *Inferno Test Kits* (including the Da Vinci PAS / CRD / DTR test kits). https://inferno.healthit.gov/test-kits