The Architecture of Continuity

Naming, Memory, Versioning, and the Mathematical Foundations of Scientific Institutions

Flyxion
Independent Researcher

November 2026

Abstract

This monograph develops a general account of how artifacts, and the people and institutions that produce them, remain usable by future agents. The starting point is elementary: an author who saves a file, names a directory, or writes a comment above a function is making a prediction about a later encounter in which the author's present knowledge will be partly unavailable. The book argues that this prediction problem is the common root of operating-system namespaces, filesystem hierarchies, naming conventions, documentation, package management, version control, repository hosting, computational reproducibility, and the public argumentation systems through which scientific claims are corrected. A sequence of formal definitions separates three kinds of persistence (physical, referential, and operational), introduces a hierarchy of continuation classes with minimal sufficient contexts, and treats naming as a retrieval problem under uncertainty. The same vocabulary is then carried through the history of software distribution, the formation of the individual developer, a mathematical theory of repository institutions (typed graphs, lineage, authority, admissibility, federation, and portability), the repository as a scientific instrument, and semantic and epistemic infrastructure for knowledge versioning. The final part examines the architectural requirements for independent scientific infrastructure and the limits of repository-based science. Every major claim carries one of four epistemic labels: proved under stated assumptions, experimentally investigated, a design argument, or conjectural. No architectural aspiration is presented as a mathematical result.

Contents

Preface **xiii**

I The Problem of Future Recall **1**

1 Externalized Memory Under Limited Recall **3**
1.1 The prediction made at the moment of saving . . . 3
1.2 Recall, retrieval, and interpretation . . . 3
1.3 Externalization and the economy of attention . . . 4
1.4 Scope of the claims . . . 4

2 Three Kinds of Persistence **7**
2.1 Physical, referential, and operational persistence . . . 7
2.2 Dependencies among the three . . . 7
2.3 Failure modes as a classification . . . 8
2.4 The role of verification . . . 9

3 Recovery as a Probabilistic Problem **11**
3.1 A simple model of delayed recovery . . . 11
3.2 Non-monotonicity in realistic settings . . . 12
3.3 What the record channel consists of . . . 12

4 The Continuation Problem **15**
4.1 From persistence to capability . . . 15
4.2 Structure of sufficient contexts . . . 16
4.3 The ladder and its limits . . . 17
4.4 The unifying claim . . . 18

II Identity, Filesystems, and Rights **19**

5 Accounts, Principals, and Namespaces **21**
5.1 The first institutional distinction . . . 21
5.2 Identity as a relation . . . 21
5.3 The home directory and the persistent address . . . 22

6 Permissions as an Authorization Relation **25**
6.1 The permission check . . . 25
6.2 Expressivity limits of mode bits . . . 26
6.3 Directories, traversal, and the semantics of write . . . 26

7 Paths, Files, and Classification **29**

7.1 Files, names, and the identity of content . . . 29
7.2 Classification by hierarchy . . . 30
7.3 Context contributed by the path . . . 31

8 Naming as Prediction **33**
8.1 The retrieval loss of a naming policy . . . 33
8.2 Bounds on label-only retrieval . . . 33
8.3 Conventions as constraints on the search space . . . 34
8.4 Names in code, in commits, and in headings . . . 35

III The Discipline of Reuse **37**

9 Scripts, Templates, and Portable Configuration **39**
9.1 The script as externalized procedure . . . 39
9.2 Idempotence and the safe rerun . . . 39
9.3 Dotfiles and the portable environment . . . 40
9.4 Templates and the reuse of structure . . . 41

10 Documentation as Communication with an Absent Agent **43**
10.1 The manual page tradition . . . 43
10.2 Contracts and what the code does not say . . . 43
10.3 History as documentation . . . 44
10.4 Documentation decay . . . 44

11 Dependencies and Environments **47**
11.1 The hidden context of imported code . . . 47
11.2 Version constraints and resolution . . . 47
11.3 The computational difficulty of resolution . . . 48
11.4 Environments as artifacts . . . 48

12 Conventions, Noise, and the Economy of Reading **51**
12.1 More context can mean less usability . . . 51
12.2 Ordering by expected relevance . . . 51
12.3 Standard files as institutional organs . . . 52
12.4 Closing the first three parts . . . 52

IV The History of Software Continuity **53**

13 Software Before Persistent Distribution **55**
13.1 Programs as physical objects . . . 55
13.2 Networks of exchange . . . 55
13.3 The problem of identification . . . 56

14 The Dependency and Version Problem **59**
14.1 Shared libraries and the single-slot namespace . . . 59
14.2 Registries and package managers . . . 59

14.3 Semantic versioning as a convention 60

15 Version Control Before Git **63**
15.1 The first systems 63
15.2 Concurrent versioning and the repository state 63
15.3 Centralized coordination and centralized storage 64
15.4 The circumstances of Git 64

16 Git and the Content-Addressed Object Graph **67**
16.1 Objects and identifiers 67
16.2 The Merkle property 67
16.3 Operations and what they do to identifiers 68
16.4 What the object graph does not contain 69

17 The Emergence of a Recognized Upstream **71**
17.1 Convention built over symmetry 71
17.2 Forks as challenges 71
17.3 Hosting as consolidation 72
17.4 A taxonomy of functions 72

V The Formation of the Developer **75**

18 Account Creation and Persistent Identity **77**
18.1 From principal to public identity 77
18.2 Authorship in the object graph is a claim 77
18.3 Pseudonyms, handles, and recognition across time 78

19 Attention Before Contribution **79**
19.1 Five relations that are often conflated 79
19.2 Counts as signals 79
19.3 Cumulative advantage 80
19.4 Attention as the beginning of a trajectory 80

20 Forking and Independent Continuation **83**
20.1 The discovery of the fork 83
20.2 Technical copying and legal permission 83

21 Contribution, Refusal, and Maintainer Authority **85**
21.1 The pull request as a protocol 85
21.2 Mechanism and decision 85
21.3 Refusal as information 86
21.4 Governance mechanisms available to owners 86

22 From Developer to Institutional Organizer **89**
22.1 Creating a first repository 89
22.2 A developmental model 89
22.3 Responsibility and irreversibility 90

22.4 Governance levels 90

23 Continuity Debt, Orphaning, and Succession **93**
23.1 Continuity debt 93
23.2 Qualified agents and orphaning 94
23.3 Succession 95
23.4 Overload and the maintainability horizon 95
23.5 What the chapter contributes 96
23.6 Result of Part V 96

VI The Mathematics of Repository Institutions **97**

24 Typed Repository Graphs **99**
24.1 Why a typed graph 99
24.2 The native layer and the hosted layer 99
24.3 Admissible transformations 100
24.4 Layering and the direction of dependence 102

25 History, Lineage, and Continuation **103**
25.1 Histories as partial orders 103
25.2 Which properties of lineage survive which operations 103
25.3 Squashing and the compression of history 105
25.4 Continuation of a history 105

26 Authority Without Centralized Storage **107**
26.1 Technical existence versus recognition 107
26.2 History alone does not select an authority 107
26.3 Recognition without unanimity 108
26.4 How a recognized upstream emerges 109

27 Contribution as an Admissibility Problem **111**
27.1 Four predicates 111
27.2 Governance regimes and review 112
27.3 The role of automated checks 113

28 Forks, Federation, and Governance Divergence **115**
28.1 The three-way merge as a partial operation 115
28.2 Divergence of governance 116
28.3 When separation is appropriate 116

29 Portability, Dependence, and Institutional Capture **119**
29.1 History is portable; the institution is not guaranteed to be 119
29.2 Lineage preservation and its limits 119
29.3 The institutional portability criterion 120
29.4 Degrees of dependence and capture 121
29.5 Mitigations 122
29.6 Result of Part VI 122

VII The Repository as Scientific Instrument **123**

30 The Incomplete Scientific Publication **125**
30.1 What a paper can carry . . . 125
30.2 The specification gap . . . 125
30.3 Reproduction, replication, and the vocabulary of agreement . . . 126

31 Reproducibility as a Dependency Structure **129**
31.1 The experiment as a structured object . . . 129
31.2 Declared inputs and the closure criterion . . . 129
31.3 Rerunning, reproducing, replicating as levels of the dependency graph . . . 131
31.4 A catalog of failure modes . . . 131

32 Provenance and the Executable Scientific Record **133**
32.1 What a hash, a tag, and a signature establish . . . 133
32.2 Authorship and timestamps are claims . . . 134
32.3 External anchoring . . . 134
32.4 Persistent identifiers . . . 135

33 Continuous Verification and Scientific Revision **137**
33.1 Three distinct activities . . . 137
33.2 Tests as finite samples . . . 137
33.3 Continuous integration and its evidential role . . . 138
33.4 Proof assistants and formalized correctness . . . 138
33.5 Revision as a normal condition . . . 139
33.6 Symbolic derivations and the audit of formulas . . . 139

34 Scientific Priority and Public Research Lineage **143**
34.1 Priority as a social institution . . . 143
34.2 Evidence of prior publication and conclusive proof . . . 143
34.3 Public lineage as a research practice . . . 144
34.4 Result of Part VII . . . 144

VIII Semantic and Epistemic Infrastructure **145**

35 The Limits of File-Centric Representation **147**
35.1 What files, directories, commits, and diffs represent . . . 147
35.2 Co-change is not dependency . . . 148
35.3 Identity of content versus location of a document . . . 148

36 Typed Semantic Hyperstructures **149**
36.1 Semantic Structure and Preservation . . . 149
36.2 Content atoms, references, and transclusion . . . 149
36.3 Transclusion and expansion . . . 149
36.4 Semantic embeddings and the limits of similarity . . . 150

37 Semantic Transformations and Preservation **153**

37.1 Which transformations, which invariants . . . 153
37.2 Extraction . . . 153
37.3 Hoisting . . . 154
37.4 Operations whose preservation properties must be established from their definitions . . . 155

38 Adversaria and the Public Claim Graph **157**
38.1 Claims, Grounds, and the Persistence of Justification . . . 157
38.2 Preserving a proposition and preserving the history of its justification . . . 158
38.3 An append-only record . . . 158
38.4 An example status function . . . 158
38.5 What the example shows about structure . . . 159

39 Defeat, Revision, and Epistemic Continuity **161**
39.1 What Revision Requires of a Record . . . 161
39.2 Rules are part of the record . . . 162
39.3 Retraction is an event, not a deletion . . . 162
39.4 Defeat of different kinds . . . 163

40 The Semantic-Epistemic Interface **165**
40.1 Two layers with different jobs . . . 165
40.2 A candidate architecture . . . 165
40.3 Soundness of the interface . . . 166
40.4 When a transformation changes what a claim is about . . . 166
40.5 Result of Part VIII . . . 167

IX From Personal Memory to Public Institution **169**

41 Requirements for a General Scientific Repository **171**
41.1 Method: requirements from functions . . . 171
41.2 Indispensable functions . . . 172
41.3 Conveniences inherited from existing interfaces . . . 172

42 A Layered Repository Architecture **173**
42.1 Seven layers . . . 173
42.2 Independence of layers . . . 173
42.3 What can be implemented over Git and what cannot . . . 173

43 Migration, Federation, and Independence **177**
43.1 Institutional history is what must survive . . . 177
43.2 Replication and common-cause failure . . . 177
43.3 Governance continuity and the dead-hand problem . . . 178
43.4 Succession as data . . . 178

44 Plenumhub as a Candidate Implementation **179**
44.1 Status and scope . . . 179
44.2 Independence as a design requirement . . . 179

44.3 An acceptance plan . . . 179
44.4 What the design cannot guarantee . . . 180

45 The Individual as an Institutional Center 181
45.1 Functions and legitimacy . . . 181
45.2 Key-person dependence . . . 181
45.3 Public lineage as evidence . . . 182

46 Network Effects and De Facto Standards 183
46.1 Why technical alternatives do not displace incumbents . . . 183
46.2 Critical mass . . . 183
46.3 Interoperability as threshold reduction . . . 184
46.4 Cumulative value of established histories . . . 184
46.5 Professional signaling . . . 184

47 The Limits of Repository-Based Science 187
47.1 What repositories cannot supply . . . 187
47.2 Independence of evaluation . . . 187
47.3 Infrastructural possibility is not institutional sufficiency . . . 188

48 From Repository to Institution 189
48.1 The Criterion . . . 189
48.2 Summary of the argument . . . 190
48.3 Open problems . . . 190
48.4 Closing . . . 191

A Notation Glossary 193

B A Chronology of Software Distribution and Version Control 195

C Comparative Models of Git and Hosting-Service Objects 197

D A Catalog of Reproducibility Failure Modes 199

E Cross-Reference Mapping to the Polyxan and Adversaria Specifications 201

F Computational Appendix: Experiments 203
F.1 Purpose and limits . . . 203
F.2 Kinds of experiment . . . 203
F.3 E1: lineage under clone, rebase, and cherry-pick . . . 203
F.4 E2: migration of Git data and the absence of platform records . . . 204
F.5 E3: properties of the pathwise merge . . . 204
F.6 E4: dependency resolution over time . . . 204
F.7 E5: Monte Carlo check of the recovery model . . . 204
F.8 E6: closure check by recording reads . . . 205
F.9 E7: append-only claim log, replay, and reinstatement . . . 205
F.10 E8: extraction and hoisting . . . 205
F.11 E9: symbolic derivations and declared assumptions . . . 205

G Protocol for Estimating Recovery Parameters **207**

H Empirical Agenda **209**

I Register of Results by Epistemic Status **211**

Bibliography **217**

Preface

A repository begins as a convenience and ends as an institution only if certain things happen along the way. Files must acquire names that can be searched for. Names must acquire contexts that disambiguate them. Contexts must be recorded in forms that survive the departure of whoever understood them. Changes must be recorded in a manner that allows later readers to tell what was altered and why. Disagreements about which continuation of a body of work is authoritative must have procedures, however informal. Claims drawn from the work must be connected to the evidence and the reasoning that support them, so that a later reader can evaluate them without trusting the author. This book is an attempt to say precisely what these requirements are, in what order they arise, which of them can be mechanized, and which of them cannot.

The subject is deliberately broader than any hosting service. Repository hosting is a late and contingent development in a long history of attempts to keep work usable across time, and a theory that took one provider as its object would inherit the provider's accidents. The book therefore begins with externalized memory, proceeds through accounts, namespaces, permissions, paths, and naming, and arrives at hosted collaboration only after the problems that collaboration solves have been stated in neutral terms. Hosting services appear as historical examples and as sources of counterexamples, never as the definition of the object of study.

Three conventions govern the exposition. First, formal definitions are given for the notions that carry weight in later arguments, and informal discussion is separated from them. Second, results are classified. A statement marked [PROVED UNDER STATED ASSUMPTIONS] is a theorem or proposition proved from assumptions that are stated beside it. A statement marked [EXPERIMENTALLY INVESTIGATED] has been investigated through small computational experiments described in the appendices, with the understanding that experiment does not establish a universal claim. A statement marked [DESIGN ARGUMENT] is an argument about what a system ought to provide, not a claim about what any system does provide. A statement marked [CONJECTURAL] is a conjecture, offered because it is precise enough to be refuted. Second-order claims of this kind are kept apart from the historical narrative, which relies on documented sources listed in the bibliography and is otherwise described cautiously where records are incomplete.

Third, and with a stated limitation, the treatment of the semantic and argumentative systems discussed in Part VIII is interface-first. Those systems exist as independent specifications with their own established definitions, and this book does not replace their vocabulary. Where a chapter requires a notion from them, the notion is stated as an abstract interface, and the correspondence with the specification is recorded in the appendix on cross-reference mapping. Results about them proved here hold for the abstract interface and transfer to a specification exactly when the specification satisfies the stated assumptions.

The nine parts are arranged so that each depends on the earlier ones but can be read for its own sake. Part I states the problem of future recall and the three notions of persistence. Part II treats identities, namespaces, permissions, and naming. Part III treats the disciplines through which artifacts are made reusable: scripts, templates, documentation, dependencies, and

conventions. Part IV gives the history of software continuity up to Git and hosted repositories. Part V describes the progressive acquisition of capability by a developer, distinguishing what a person can do from what the person knows can be done. Part VI is the mathematical core, concerned with typed repository graphs, lineage, authority, admissibility, divergence, and portability. Part VII applies the framework to computational science, reproducibility, and priority. Part VIII treats knowledge versioning beyond files. Part IX turns to architectural requirements, independence from particular providers, and the limits of the whole enterprise.

Relation to Previous Work

This book is a synthesis of research lines developed earlier by the author, and the new contribution is the explicit derivation that runs from individual memory and filesystem organization to repository institutions and the preservation of scientific knowledge. Three lines converge here. The first is the semantic infrastructure line, which begins with the RSVP-Polyxan unified field theory of semantic hyperstructures [12] and its category and sheaf formalization of selective semantics [13], and continues through a coherence-first account of identity and provenance [14] and a study of manifold-aligned semantic dynamics [15], together with the Polyxan specifications and PlenumHub that are treated in Part VIII. The second is the historical and operational continuity line, represented by the work on persistence, event histories, replay, procedural ontology, and repair, including the deterministic event-sourced operating system Spherepop. The third is the epistemic accountability line, represented by the study of correction and answerability [16], by the Adversaria specification of a public claim graph [17], and by a series of shorter essays on evidence, provenance, and verification. Earlier essays on memory, retention, and digital decay supply the motivation for Part I. The titles and dates of these works have been recovered from the author's repositories and have not all been checked against canonical versions; several are specifications or project names and not independently published papers, and some exist in more than one version. They are cited for orientation and not as evidence for any result in this book, and each theorem here is proved or investigated from the definitions given in these pages.

Information Retrieval, Personal Information Management, and the Governance of Shared Repositories

The problem of preserving an artifact for a future interpreter has antecedents in several established research traditions. Information retrieval examines the representation, indexing, ranking, and recovery of documents; personal information management examines how individuals keep, organize, and rediscover information over time; and research on open-source governance examines the rules through which distributed communities coordinate contributions and maintain collective projects. These literatures address different scales of a common difficulty: information does not remain useful merely because it remains stored. Its continued use depends on relations among artifacts, agents, descriptions, practices, and institutions.

Information retrieval and the problem of recoverability

Classical information retrieval distinguishes a collection of documents from the mechanisms by which documents become accessible in response to queries. Manning, Raghavan, and Schütze [41] develop the foundations of document indexing, query processing, ranking, and retrieval evaluation. Their treatment makes clear that the existence of relevant information in a collection does not entail that a retrieval system will return it, or that a returned document will satisfy the user's information need. This anticipates the separation developed in this monograph between physical and referential persistence. A file may be preserved without remaining discoverable to an agent whose vocabulary, knowledge, or retrieval tools have changed, and an improved index may restore accessibility to an artifact whose physical representation has not changed at all.

The present framework extends the retrieval problem by asking what happens after the artifact is found. A retrieved program may lack its dependencies, a recovered dataset may lack its measurement protocol, and a discovered scientific claim may lack the evidentiary record necessary for evaluation. Referential persistence is therefore necessary for many forms of continuation but is not sufficient for operational persistence. There is also a difference between the usual evaluation of retrieval systems and the problem considered here. Retrieval effectiveness is commonly assessed relative to a specified collection, query set, and relevance judgments. Continuation introduces a temporal dimension: the future agent may not know which query to formulate, which collection to search, or which interpretation of the recovered artifact is correct. Naming and metadata must therefore be understood partly as predictions about the information available to an unknown future searcher.

Personal information management and the future self

The personal information management literature approaches this difficulty from the standpoint of an individual who must preserve information for later use. Jones [39] treats personal information management as a collection of activities involving finding, keeping, organizing, maintaining, and making sense of information, and Jones and Teevan [40] situate these activities within the practical limitations of attention, time, and the changing circumstances in which information is needed. An especially relevant empirical contribution is the study by Barreau and Nardi [37], which synthesizes investigations of how people organize and recover computer files. Their findings emphasize the importance of location-based retrieval as a means of reminding, alongside users' reluctance to construct elaborate filing systems. The location of an artifact can therefore serve not merely as an address but as a cue to the activity, project, or purpose with which the artifact was associated.

This observation has consequences for the interpretation of directory hierarchies. A directory path is simultaneously a retrieval mechanism and a partial representation of the context in which a file was produced. Reorganization may improve one mode of retrieval while damaging another, and replacing a familiar hierarchy with a search index can reduce lookup costs without necessarily preserving the contextual cues supplied by the original arrangement. The present monograph differs from personal information management research in treating the future self as one instance of a more general class of successor agents. The later interpreter may be another developer, a scientific reviewer, an automated build system, or an institution that inherits responsibility for an abandoned project. The formal notion of continuation developed here is intended to express that transfer without assuming that the successor possesses

the producer's memories. It also distinguishes the preservation of sufficient information from the practical problem of selecting and interpreting that information under bounded effort, a distinction that remains an important limitation of the monotone sufficiency model.

Open-source development as collective institutional organization

The open-source literature examines the corresponding problem at the level of distributed communities. Raymond's account of the cathedral and bazaar models [43] offers an influential comparison of development practices, particularly the role of distributed participation and iterative correction. It is best read as a historically influential architectural argument and not as a universal empirical law of software production. O'Mahony and Ferraro [42] provide a more directly institutional analysis. Their study of an open-source community describes the emergence of formal authority together with mechanisms that limit and legitimate its exercise. Governance in this account is not reducible to the technical ability to modify shared code. It concerns the procedures through which participants acquire authority, contest decisions, and coordinate their contributions.

Schweik and English [44] extend the institutional perspective by examining open-source software projects as commons, including the conditions associated with project success and abandonment. Their analysis makes the persistence of a collaborative project a question of social organization as well as technical maintenance. A repository can remain publicly accessible after its community has ceased to maintain it; equally, a community can preserve the practical continuity of a project while migrating its infrastructure. The development of hosted repositories adds another dimension. Dabbish and colleagues [38] investigate how visible activity on GitHub supports social inference, coordination, learning, and reputation. The repository is consequently more than a store of source-code revisions, since its public activity records provide information about participants and their relationships that cannot be reconstructed from the source tree alone. This motivates the separation, developed later in this book, between the Git object graph and the institutional graph maintained around it. Commit ancestry records one kind of continuity, while issues, reviews, permissions, organizational membership, and public recognition record others, and the latter participate in the determination of which contributions are accepted, which histories are treated as authoritative, and which agents can legitimately continue a project.

The contribution of a unified continuation framework

These research traditions establish much of the phenomenology from which the present argument begins. Information retrieval already recognizes the gap between storage and discovery, personal information management the difficulty of preserving useful context for later encounters, and open-source governance research the need of distributed production for institutions of authority, coordination, and legitimation. The proposed contribution of this monograph is not the discovery of these difficulties individually. It is an attempt to place them within a common formal vocabulary and to examine their dependencies across scales. The resulting questions are more demanding than whether a document can be found, a program can be executed, or a project can attract contributors. They concern whether a future agent can recover the conditions under which an artifact may be used correctly, modified safely, evaluated independently, and incorporated into an authorized continuation of a shared history.

No claim is made that the three literatures can be reduced to one another. Retrieval effectiveness does not determine institutional legitimacy, and a valid governance procedure does not guarantee scientific correctness. The value of the continuation framework depends instead on its ability to identify the information and relations that each kind of activity requires, to distinguish those requirements from one another, and to specify the transformations under which they are preserved or lost. In this sense the architecture of continuity is neither a theory of search alone nor a theory of version control alone. It is a proposed theory of the conditions under which recorded work remains available for responsible action after the circumstances of its production have changed.

Part I

The Problem of Future Recall

Chapter 1

Externalized Memory Under Limited Recall

1.1 The prediction made at the moment of saving

Every act of saving is addressed to a reader who does not yet exist. The reader may be the same biological individual, but the individual who returns to a file after a year is in a different informational state from the one who wrote it. The writer knew why the file was created, what problem it addressed, which assumptions it embodied, where related files could be found, and what had been tried and abandoned. The returning reader knows some of this, imperfectly, and is typically unaware of how much has been lost. The asymmetry has a precise consequence: the cost of making an artifact usable is borne at the moment when its necessity is least apparent, because the writer's present understanding makes the artifact seem self-explanatory.

This asymmetry is the basic fact from which the rest of the book proceeds. It is not a claim about unusual forgetfulness. It follows from any model in which an agent's accessible knowledge decays or is displaced faster than the artifacts the agent produces. Human memory has this property; so do organizations, in which personnel turn over; so do research communities, in which the practices that make a result reproducible are seldom written down. The problem is general, and so the vocabulary used to treat it should not presuppose any particular medium.

Definition 1.1 (Agents, artifacts, and encounters)**.** An *agent* is a system with an internal state that can change over time, a repertoire of operations, and a capacity to search. Write $\sigma_a(t)$ for the state of agent a at time t. An *artifact* is a finitely describable object x that persists, if it persists, independently of the state of any particular agent. An *encounter* is a triple (a, x, t') in which agent a attempts at time t' to find, interpret, or use artifact x that was produced at some earlier time $t < t'$ by an agent a_0. The *delay* of the encounter is $\Delta = t' - t$. The agents a_0 and a may be identical as individuals while differing as states, so that $\sigma_{a_0}(t) \neq \sigma_a(t')$ in general.

The definition makes no assumption that the later agent is a person. A build system, a search index, a language model that is asked to continue a project, and a reviewer reconstructing a published analysis are all agents in the relevant sense. What distinguishes them is the repertoire of operations they can perform and the background knowledge they bring, both of which enter the analysis through the notion of context introduced below.

1.2 Recall, retrieval, and interpretation

The word remembering conflates at least three distinct capacities. An agent may remember that an artifact exists, may remember where it is, and may remember what it is for. The first

is awareness, the second is location, and the third is understanding. Ordinary usage treats them as a single capacity because in everyday objects they usually fail together or not at all. In stored digital artifacts they come apart. A person may recall writing a script that computes a graph statistic, have no idea which directory contains it, and be unable, on finding it, to say why it uses a particular normalization. Each of these failures is independent of the others, and each calls for a different remedy. Awareness is supported by indexes and catalogs, location by naming and structure, and understanding by documentation and tests.

Principle 1.2 (Remembering is not retrieving)**.** For an artifact x and an encounter (a, x, t'), the following three conditions are logically independent: that x continues to exist in some physical form; that a can identify and obtain x within a bounded effort; and that a can use x correctly for the purposes of the encounter. Practices intended to support future encounters must be evaluated against each condition separately.

This principle is stated here without proof because it is a design principle; the formal version, with independence established by countermodels, is given as Proposition 2.4 in the next chapter. Its practical force is that an investment aimed at one condition does not purchase the others. A reliable backup system secures existence and nothing more. A disciplined naming scheme improves location, whether or not the files it names remain intelligible. A comprehensive manual improves understanding but is useless if nobody can find it.

1.3 Externalization and the economy of attention

A person who writes something down is externalizing a computation. The note holds the result so that the person does not have to hold it. The benefit is the release of attention for present work; the cost is a dependence on the note being found and understood later. Externalization is thus a loan against the future, and like any loan it is cheap when contracted and costly when called in. The relevant economy has two sides. Writers underinvest in future-oriented metadata because the benefit is deferred and uncertain. Readers overinvest in reconstruction because the alternative is to abandon the artifact. The equilibrium is a body of work in which the most important artifacts are the ones that are hardest to use, since the artifacts that matter most are the ones that depend on the most accumulated context.

Institutions can be understood as arrangements that correct this misallocation by making the externalization of context a shared obligation. A laboratory notebook convention, a code review that requires a description, a journal's data availability policy, and a distribution's packaging guidelines are all mechanisms for obliging writers to leave information for readers. They differ in what they require, whom they bind, and how they are enforced. They are comparable because each responds to the same asymmetry.

1.4 Scope of the claims

Some caution about what follows is needed from the outset. The book does not claim that all information should be preserved, that more documentation is always better, or that every artifact deserves long-term care. Preservation has costs, and the formal development in later chapters treats those costs explicitly. Most artifacts should be allowed to lapse, and the interesting question is how to decide which. The claims are conditional: if an artifact is to

support a given class of future operations by a given class of agents, then a certain minimum of context must accompany it, and the minimum has a structure that can be studied.

Chapter 2

Three Kinds of Persistence

2.1 Physical, referential, and operational persistence

The distinctions drawn informally in the previous chapter can be sharpened by relating each to an effort bound. An agent never has unlimited effort available, and an artifact that can be found only by an exhaustive search of every storage medium accessible to the agent is, for practical purposes, not findable. The definitions below therefore carry an explicit budget.

Definition 2.1 (Physical persistence)**.** Let W_t denote the set of storage states of the world at time t. An artifact x is *physically persistent* at t, written $\mathrm{Phys}(x, t)$, if there is some state $w \in W_t$ and some decoding procedure d such that $d(w) = x$. The definition ignores whether any agent knows of w or d.

Definition 2.2 (Referential persistence)**.** Let β be a positive effort budget, measured in whatever units are appropriate (time, queries, cost). An artifact x is *referentially persistent for agent a at time t' within budget β*, written $\mathrm{Ref}(x, a, t', \beta)$, if there is a retrieval procedure ρ available to a with cost at most β such that ρ returns an object $\hat{x}$ and a can verify that $\hat{x} = x$ by a check available to a at cost at most β. Verification may rely on a stored identifier such as a content hash, on comparison with a remembered feature, or on any other criterion that, given the agent's knowledge, distinguishes x from alternatives.

Definition 2.3 (Operational persistence)**.** Let K be a class of operations. An artifact x is *operationally persistent for agent a at t' with respect to K within budget β*, written $\mathrm{Op}(x, a, t', K, \beta)$, if $\mathrm{Ref}(x, a, t', \beta)$ holds and a can perform every operation in K on x correctly, where correctness is judged against the specification of the operation, within budget β.

The verification clause in Definition 2.2 excludes the case in which the agent retrieves something and takes it for the artifact. Retrieval without verification is a weaker property which is worth naming but does not support continuation, since the agent acting on a wrongly identified object proceeds with false confidence. The operation class K in Definition 2.3 is left abstract here and given a structure in the next chapter.

2.2 Dependencies among the three

From the definitions, two implications hold under a natural assumption, and their converses fail.

Proposition 2.4 (Ordering of the persistence notions)**.** *Assume that retrieval procedures return only objects obtained from existing storage states. Then:* (i) $\mathrm{Ref}(x, a, t', \beta)$ *implies* $\mathrm{Phys}(x, t')$*;* (ii) $\mathrm{Op}(x, a, t', K, \beta)$ *implies* $\mathrm{Ref}(x, a, t', \beta)$*;* (iii) *Neither converse holds in general. There exist models in which* Phys *holds and* Ref *fails for every agent and budget, and models in which* Ref *holds and* Op *fails for a nonempty K.* [PROVED UNDER STATED ASSUMPTIONS]

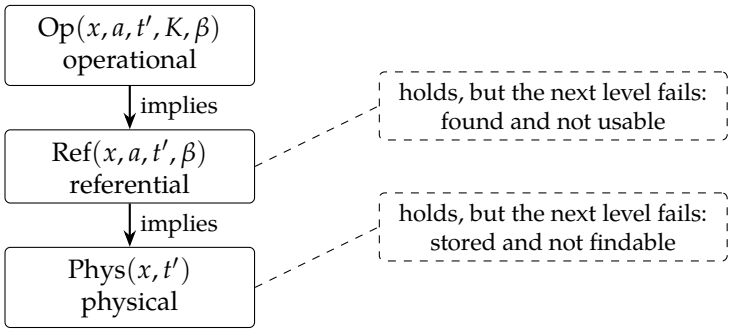


Figure 2.1: The three kinds of persistence of Proposition 2.4. Solid arrows are implications that hold under the stated assumption. The dashed boxes mark the converses, which fail: an artifact can satisfy a lower level without satisfying the one above it. The figure shows logical dependence and says nothing about the probability of any failure.

Proof. Part (i): if a retrieves and verifies x, then the procedure returned x, and by the assumption x was decoded from an existing state, so Phys(x, t'). Part (ii) is immediate from Definition 2.3, which includes Ref as a conjunct. For part (iii), the countermodels below are given. □

Countermodel 2.5 (Present but not locatable)**.** Let x be a data file on a disk image stored in a box, and let the only record of the box's existence be the memory of an agent who has since died. Then Phys(x, t') holds, since a decoding of the disk exists, but no agent has any retrieval procedure of bounded cost that reaches it. This is the case that archivists describe as a collection without a finding aid.

Countermodel 2.6 (Locatable but not usable)**.** Let x be a compiled program with a recorded hash, stored in a known location, written for a hardware platform and a library version that no longer exist and with no source or documentation retained. Then x can be found and verified, so Ref holds, but an agent who wishes to modify the behavior of x lacks the information needed to do so within any feasible budget. The class K containing modification is not satisfied, so Op fails for that K even though it may hold for the class consisting only of copying.

The third persistence notion is relative to K, and this is not a defect but a feature. The same artifact may be operationally persistent for one class of operations and not another. A compiled binary that cannot be modified may still be executed on an emulator, a spreadsheet whose macros are broken may still be read as data, and a text file in a forgotten encoding may still be opened and examined as bytes. The relevant questions are always of the form: persistent for whom, within what budget, for what operations.

2.3 Failure modes as a classification

The three notions yield a classification of failures that is more useful than the undifferentiated complaint that something has been lost. A *loss* is a failure of physical persistence: the bits are gone. A *disappearance* is a failure of referential persistence in the presence of physical persistence: the bits exist but cannot be found or cannot be recognized as the sought object. A *decay of intelligibility* is a failure of operational persistence in the presence of referential

persistence: the artifact is in hand and unusable. Each has different causes and different remedies, and a program of preservation that addresses only one will produce the others as byproducts. A common instance is the migration of a collection to a new format, which secures physical persistence and often destroys referential persistence by changing identifiers, and sometimes operational persistence by dropping features of the original that were not represented in the new format.

Example 2.7 (A migrated mailbox)**.** A mailbox archive is converted from a proprietary format to plain text files in a directory per month. Physical persistence is improved, because plain text has no dependence on a vendor application. Referential persistence is damaged for any agent who formerly located messages by folder hierarchy and flag state, because those are not carried by the new format. Operational persistence with respect to the class of operations *reconstruct a conversation thread* is damaged if message identifiers are not preserved. The migration is a net benefit or a net loss only relative to a stated agent and a stated class of operations.

2.4 The role of verification

The verification clause deserves a closer look, because it marks the place where content addressing enters the story. An agent who holds a hash of an artifact can verify any candidate by recomputing the hash and comparing, so that referential persistence for that agent reduces to the problem of finding any copy of the artifact, wherever it may be. This gives hashes a distinctive role: they separate the identity of an artifact from its location, and so make referential persistence robust against changes of location. They do not provide any of the context needed for operational persistence, and a hash is useless to an agent who has lost the hash. These observations are developed formally in Part VI, where content-addressed object graphs are treated in full.

Chapter 3

Recovery as a Probabilistic Problem

The following model abstracts from established information-retrieval models, which evaluate the recovery of documents relative to a collection, a query set, and relevance judgments [41]. It concerns a single artifact and a single delay, and its two channels stand in for the whole apparatus of indexing and query formulation.

3.1 A simple model of delayed recovery

The independence results of the previous chapter are qualitative. Practical decisions about how much effort to spend on naming, documentation, and indexing require a quantitative picture, and a model need not be accurate to be useful for the purpose of exposing the structure of the tradeoff. The following model isolates the principal dependencies.

Let an artifact be produced at time t and an encounter occur at $t' = t + \Delta$. Let m denote the metadata recorded with the artifact (names, comments, indexes, documentation) and c the retrieval context in which the later agent operates (available search tools, search vocabulary, the agent's knowledge of the project). The quantity of interest is

$$R(x, \Delta \mid m, c) = \mathbb{P}\big[\text{the agent recovers and correctly interprets } x \text{ after delay } \Delta\big].$$

Definition 3.1 (Two-channel recovery model)**.** Recovery can occur through two channels that are assumed to succeed independently. The *memory channel* succeeds with probability $p(\Delta) = e^{-\lambda \Delta}$, where $\lambda > 0$ is a forgetting rate; it represents the agent's unaided recollection of location and purpose. The *record channel* succeeds with probability $s = s(m, c) \in [0, 1]$, which depends on the recorded metadata and the retrieval context but not on Δ directly. Then

$$R(\Delta) = 1 - \big(1 - e^{-\lambda \Delta}\big)\big(1 - s\big).$$

The independence assumption is strong, and in practice the channels interact: a person's memory of an artifact's existence prompts a search that uses the metadata. The formula should be read as a benchmark and not as a bound. If the channels are positively dependent, so that success of one makes success of the other more likely, the probability that at least one succeeds is lower than the independent value; if they are negatively dependent, it is higher. The direction of the error therefore depends on the dependence, which is an empirical matter. The value of the model is that it makes certain qualitative consequences explicit, and the consequences of Proposition 3.2 should be checked for robustness against dependence before being relied on.

Proposition 3.2 (Behavior of the two-channel model)**.** *Under Definition 3.1 with s constant in Δ: (i) R is nonincreasing in Δ, with $R(0) = 1$ and $\lim_{\Delta \to \infty} R(\Delta) = s$; (ii) The marginal value of the record channel, $\partial R / \partial s = 1 - e^{-\lambda \Delta}$, is increasing in Δ and vanishes at $\Delta = 0$; (iii) If an investment*

of cost κ raises s by δs, then the investment increases the recovery probability by $\delta s\,(1 - e^{-\lambda\Delta})$, so it is worthwhile at horizon Δ with benefit-per-recovery V only when $V\,\delta s\,(1 - e^{-\lambda\Delta}) > \kappa$. [PROVED UNDER STATED ASSUMPTIONS]

Proof. Write $p = e^{-\lambda\Delta}$, so $R = 1 - (1-p)(1-s) = p + s - ps = s + p(1-s)$. Then $dR/d\Delta = (1-s)\,dp/d\Delta = -\lambda(1-s)e^{-\lambda\Delta} \leq 0$, with equality only when $s = 1$, which gives monotonicity. At $\Delta = 0$, $p = 1$ and $R = 1$; as $\Delta \to \infty$, $p \to 0$ and $R \to s$. For (ii), $\partial R/\partial s = 1 - p$. For (iii), a small change δs yields $\delta R = (1-p)\,\delta s$ to first order, and this is exact because R is linear in s. Multiplying by V and comparing with κ gives the condition. □

The proposition reproduces a familiar experience in formal terms. Over short horizons, the record channel is nearly worthless, because memory suffices; over long horizons it is nearly everything. A writer who judges the value of documentation from the standpoint of the present, when memory is intact, will rationally underinvest, and this underinvestment is a feature of the decision problem and does not need to be attributed to carelessness.

3.2 Non-monotonicity in realistic settings

The monotone decline in Proposition 3.2 depended on s being constant. In realistic settings s depends on the retrieval context c, which changes. Search tools improve, so that an artifact that was hard to find in one decade is trivially found in the next. Indexes are rebuilt, new naming conventions are adopted, and new contexts supply interpretive keys. It is therefore wrong to suppose that recoverability is a decreasing function of time. A correct statement is that recoverability is the outcome of two processes, the decay of what the agent knows and the evolution of the tools and records the agent can use, and either can dominate.

Proposition 3.3 (Recoverability can increase with delay)**.** *There exist time-varying record-channel functions $s(\Delta)$ for which $R(\Delta)$ is strictly increasing on some interval.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. With $R(\Delta) = s(\Delta) + e^{-\lambda\Delta}(1 - s(\Delta))$ we have $R'(\Delta) = s'(\Delta)(1 - e^{-\lambda\Delta}) - \lambda(1 - s(\Delta))e^{-\lambda\Delta}$. Choose $s(\Delta) = \frac{1}{2}$ for $\Delta \leq \Delta_0$ and then increase it linearly with slope μ on $[\Delta_0, \Delta_0 + \epsilon]$. At Δ_0+ we have $R' = \mu(1 - e^{-\lambda\Delta_0}) - \frac{\lambda}{2}e^{-\lambda\Delta_0}$, which is positive whenever $\mu > \frac{\lambda e^{-\lambda\Delta_0}}{2(1-e^{-\lambda\Delta_0})}$. Such a μ exists for every $\Delta_0 > 0$. □

The example is not exotic. A scanned lab notebook that was unreadable in 1990 becomes searchable after optical character recognition improves; a text file in an obscure format becomes readable when a converter is written. The lesson for preservation is that the retrieval context is not a constant of nature and that preserving the raw artifact in its original form keeps open the possibility that a later context will recover what an earlier one could not. Premature normalization, in which an artifact is converted into a convenient form and the original discarded, closes this possibility.

3.3 What the record channel consists of

The function $s(m, c)$ was left abstract. It is useful to decompose it according to the structure of the retrieval task. To recover an artifact, an agent must formulate a query, apply it to some

index, select the correct item from the results, and interpret the item. The probability of the whole is a product of the probabilities of the steps conditional on their predecessors. Names and directory structure contribute chiefly to the second and third steps. Documentation contributes chiefly to the fourth. Indexes and search tools contribute to the second. Tests contribute to the fourth, by providing a mechanical check that the agent's interpretation matches the author's. Writing $s = s_1 s_2 s_3 s_4$ for the four conditional success probabilities, any practice that increases one factor increases s by an amount proportional to the product of the others. This multiplicative structure has a notable implication: the factor with the lowest value limits the benefit of improvements to the others. A project with excellent documentation and an unsearchable file structure gains little from further documentation.

Remark 3.4 (Calibration). No numerical values are assumed for λ or for the components of s. These would have to be estimated for specific populations of agents and artifacts, and the appendix on experimental protocols outlines how such estimates could be obtained from retrospective studies of real projects. The resulting relationships are structural: the shape of the dependence on delay, the multiplicative form of the record channel, and the increasing value of metadata with horizon.

Chapter 4

The Continuation Problem

4.1 From persistence to capability

The persistence notions of Chapter 2 are relative to a class K of operations that the later agent wishes to perform. It is now time to give this class a structure. The structure should capture the intuition that opening a file requires less preserved context than running a script, running a script less than modifying it safely, and modifying it safely less than reproducing a scientific result that depends on it.

Definition 4.1 (Operation classes)**.** Fix a universe Op of operations that an agent might perform on an artifact. An *operation class* is a subset $K \subseteq \mathsf{Op}$. Let the following classes be distinguished for artifacts that are programs or program-like, with the understanding that analogues exist for other kinds of artifact: (a) *Inspection*, K_{ins}: obtaining the artifact's representation and reading it as a sequence of symbols; (b) *Execution*, K_{exe}: running the artifact on some input and obtaining its output; (c) *Correct use*, K_{use}: running the artifact on inputs within its intended domain and interpreting the output as intended; (d) *Safe modification*, K_{mod}: changing the artifact while preserving specified properties; (e) *Reproduction*, K_{rep}: reconstructing a reported result produced by the artifact; (f) *Evaluation*, K_{evl}: judging whether the artifact or the result it supports is correct for the purpose claimed.

The classes are not nested as sets of operations, but they are ordered by the demands they make on context in typical cases. Evaluation of a result presupposes the ability to reproduce it, since one cannot judge a result one cannot regenerate; reproduction presupposes correct use; and correct use presupposes execution, which presupposes inspection. The ordering is a hierarchy of *requirements*, and the next definition makes this precise.

Definition 4.2 (Context and sufficiency)**.** Fix a finite universe I of *information items*, each representing a unit of context that might be preserved with an artifact: a build instruction, a dependency version, a statement of the intended input domain, a rationale for a design choice, a statement of the statistical model, and so on. A *context* is a subset $m \subseteq I$. For an artifact x, agent a, and operation class K, the predicate $P_{x,a,K}(m)$ asserts that a, given x, m, and its own background knowledge, can perform K on x correctly. The predicate is assumed *monotone*: if $P_{x,a,K}(m)$ and $m \subseteq m'$ then $P_{x,a,K}(m')$. The set of *sufficient contexts* is

$$\mathrm{Suff}(x, a, K) = \{m \subseteq I : P_{x,a,K}(m)\}.$$

Monotonicity says that additional preserved information never makes an operation impossible. This is an idealization, because in practice irrelevant or misleading information can degrade performance, and a comprehensive archive without a finding aid can be less usable than a short one. The idealization is made so that the structure of sufficiency can be studied separately from the problem of presentation; the effect of noise is treated in Chapter 12.

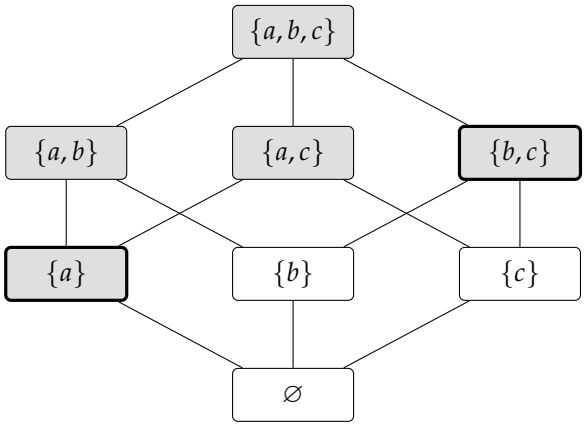


Figure 4.1: An up-set of sufficient contexts in $(\mathcal{P}(I), \subseteq)$ for $I = \{a, b, c\}$, as in Proposition 4.3. Shaded vertices are sufficient. The two heavy outlines are the minimal sufficient contexts $\{a\}$ and $\{b, c\}$: they are incomparable and of different sizes, so a minimal sufficient context is neither unique nor of fixed cardinality. The example is constructed for illustration and does not describe a particular artifact.

4.2 Structure of sufficient contexts

Proposition 4.3 (Basic structure of sufficiency)**.** *Let I be finite and let $P = P_{x,a,K}$ be monotone with* $\mathrm{Suff}(x, a, K) \neq \varnothing$*. Then: (i)* $\mathrm{Suff}(x, a, K)$ *is an up-set in $(\mathcal{P}(I), \subseteq)$, and every sufficient context contains a minimal sufficient context; (ii) If $P_{x,a,K'}(m)$ implies $P_{x,a,K}(m)$ for all m (so that K' is at least as demanding as K), then* $\mathrm{Suff}(x, a, K') \subseteq \mathrm{Suff}(x, a, K)$*; (iii) Minimal sufficient contexts need not be unique, and need not have equal size.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) That Suff is an up-set is the monotonicity assumption. Given a sufficient m, if m is not minimal there is a sufficient proper subset; repeating the step yields a descending chain of proper subsets, which terminates by finiteness at a minimal sufficient context contained in m. (ii) is immediate from the hypothesis. (iii) is shown by example: let $I = \{r, t, s\}$ and let $P(m)$ hold iff $m \supseteq \{r\}$ or $m \supseteq \{t, s\}$, where r is a README stating the interface, t is a test suite, and s is a sample invocation. Both $\{r\}$ and $\{t, s\}$ are minimal sufficient, with sizes 1 and 2. □

Part (iii) corresponds to a familiar fact: there are typically several independent routes by which a later reader can come to understand an artifact. A reader can learn how to use a function from its documentation, from its tests, from its callers, or from reading its body. The preservation problem is not to preserve one route but to choose which routes are worth the cost of preserving.

Definition 4.4 (Cost of preservation)**.** A *cost function* is a map $\kappa : I \to \mathbb{R}_{\geq 0}$ extended additively to contexts by $\kappa(m) = \sum_{i \in m} \kappa(i)$. The *minimum preservation cost* for the family $\{K_1, \ldots, K_n\}$ of operation classes is

$$\min\Big\{\kappa(m) : m \in \bigcap_{j=1}^{n} \mathrm{Suff}(x, a, K_j)\Big\}.$$

For a single operation class, the problem is easy when the sufficient contexts are given explicitly as a list of alternatives: take the cheapest. But the interesting problems involve several

classes at once, since an author typically wishes to support inspection, use, modification, and reproduction together, and different classes have overlapping routes. Overlap is what makes the combined problem hard.

Proposition 4.5 (Hardness of multi-class preservation)**.** *The decision problem* given a finite family of operation classes, each specified by a list of alternative minimal contexts, a cost function, and a budget, does a context of cost at most the budget lie in all the sufficiency sets? *is NP-complete, even when every alternative minimal context is a singleton and every item has unit cost. The reduction is from Set Cover* [20]. [PROVED UNDER STATED ASSUMPTIONS]

Proof. Membership in NP is clear, since a proposed context can be checked against each list in polynomial time. For hardness, reduce from Set Cover. Given a universe $U = \{e_1, \ldots, e_n\}$, a family of sets $S_1, \ldots, S_k \subseteq U$, and an integer B, let the information items be $I = \{1, \ldots, k\}$ with unit costs. For each e_i define an operation class K_i whose alternative minimal contexts are the singletons $\{j\}$ for every j with $e_i \in S_j$. A context $m \subseteq I$ is sufficient for all K_i simultaneously exactly when for each i it contains some j with $e_i \in S_j$, that is, exactly when $\{S_j : j \in m\}$ covers U. A context of size at most B exists iff a cover of size at most B exists. □

The result is a precise sense in which perfect documentation policy is computationally intractable in general. It does not imply that practical policies are intractable, since real structures have regularity that arbitrary instances lack. What it does imply is that no universal procedure for choosing what to preserve can be expected, and that conventions, which fix the family of routes in advance, are valuable partly because they replace an intractable optimization with a lookup.

4.3 The ladder and its limits

The hierarchy of Definition 4.1 suggests a ladder. At the bottom, inspection requires only that the artifact be retrievable and its encoding known. Execution adds the requirement that the environment can run it: interpreter or processor, libraries, system calls. Correct use adds the intended input domain and the meaning of the outputs. Safe modification adds the invariants the author intended to preserve and the rationale for the design, because a modification is safe only relative to properties, and the properties are not always visible in the code. Reproduction adds the data, the parameters, the randomness, and the measurement procedure for the scientific result. Evaluation adds the criteria by which the result is to be judged, which are partly empirical and partly conventional.

Remark 4.6 (The ladder is not universal)*.* Climbing the ladder is not a developmental inevitability. An artifact may support evaluation in the absence of safe modification, as when a mathematical result is checked by a reader who has no ability to extend the method. An artifact may support execution but not inspection, as when a binary is run on an emulator and nobody can read its source. The ordering describes typical growth in contextual demand and not a strict partial order on capabilities. In particular, evaluation of a scientific result does not presuppose its reproduction: a reader can check a derivation, identify a confounding variable, or detect an invalid inference without executing anything. The requirements of the operation classes are better described as a partial order with several routes to evaluation, of which the ladder is one chain. Later chapters will repeatedly meet the same caution in the setting of developers (Chapter 22), where growth in awareness of available operations does not entail growth in actual ability, and neither entails competence.

4.4 The unifying claim

All the practices considered in this book can be read as attempts to preserve enough structure that a future agent can recover meaning, evaluate validity, and continue development. A comment above a function and a warrant attached to a scientific claim are different objects, but each preserves information that a later interpreter needs in order to judge whether an operation or a conclusion is legitimate. The remaining parts take this common function and follow it through increasingly institutional settings: from the account on a personal computer to the namespace of a public repository, from a note to oneself to a review process, from a file to a claim.

Definition 4.7 (Continuation)**.** Let x be an artifact produced at time t with preserved context $m \subseteq I$, let a' be an agent at time $t' > t$, and let K be an operation class. Let $A_{t'}$ denote the information that a' can acquire at time t' without access to the circumstances of the production of x: public sources, documentation of tools and standards, subsequent research, and the agent's own further investigation. The relation $\mathrm{Cont}(a', x, m, t', K)$ holds when a' can perform every operation in K on x correctly using x, m, the agent's own knowledge, and information from $A_{t'}$, and without requiring any information from the circumstances of the production of x that is neither contained in m nor recoverable from $A_{t'}$.

The definition does not require the future agent to be epistemically isolated. What it excludes is dependence on inaccessible historical context, such as the unrecorded intention of a producer, as distinct from the legitimate acquisition of new information. Enlarging $A_{t'}$ weakens the demand on m, so that a context m may suffice for an agent in a well-documented field and fail for the same artifact in a field with no accessible literature; sufficiency is accordingly relative to the information environment as well as to the operation class. The final clause is what makes continuation distinct from mere use. A former collaborator who can be asked can often use an artifact whose preserved context is empty, but the dependence on the collaborator is precisely what continuation is meant to remove. Continuation is the property that a work can outlive the availability of its producers.

Part II

Identity, Filesystems, and Rights

Chapter 5

Accounts, Principals, and Namespaces

5.1 The first institutional distinction

A person who first sits at a multi-user operating system meets an elementary institutional fact before meeting any program: the machine is not merely a collection of programs but an environment populated by identities, resources, and rules governing what the former may do with the latter. The session begins with a name and a credential. The name selects a principal; the credential supports a claim that the person is entitled to act as that principal. Everything that follows, including the files created, the processes started, and the permissions exercised, is attributed to the principal and not to the person. This separation between the human being and the identity under which the human acts is a long-standing feature of multi-user computing, and it predates Unix: time-sharing systems of the 1960s, among them CTSS and Multics, already attached accounts, credentials, and access rules to users, and Multics in particular developed access control lists and protection rings. Unix simplified and popularized this inheritance. Every later construction in this book, from repository accounts to claim authorship, inherits the separation.

In Unix-like systems a principal is represented internally by a numeric user identifier, and a user may belong to several groups, each represented by a numeric group identifier. The textual user name that appears at the login prompt is a label for the number; the kernel authorizes operations by comparing numbers. In the traditional account database, each user entry carries a name, a numeric identifier, a primary group identifier, a descriptive field, a home directory path, and a login shell. In Windows systems the corresponding construct is the security principal, identified by a security identifier, with a token constructed at logon that lists the principal’s identifier, group memberships, and privileges. The two designs differ in many respects, but they agree on the structural point: authorization is decided by identifiers, and labels are for humans.

5.2 Identity as a relation

It is tempting to identify an account with its name. The tempting identification fails for the same reason that identifying a document with its title fails: names can be changed, reused, and shared, while the thing named must remain recognizable across such changes if records about it are to remain meaningful.

Definition 5.1 (Namespace and principal)**.** A *namespace* is a pair (L, ν) in which L is a set of labels and $\nu : L \rightharpoonup \mathsf{Id}$ is a partial resolution function to a set of identifiers. A *principal* is an element $p \in \mathsf{Id}$ for which authorization predicates are defined. A namespace is *injective* if distinct labels never resolve to the same identifier, and *stable* under an operation if the operation does not change $\nu(\ell)$ for any label ℓ outside the operation’s declared scope.

Definition 5.2 (Identity as a relation)**.** The *identity* of an agent in an institution is not a single identifier but a relation $\mathrm{Id} \subseteq L \times \mathsf{Id} \times \mathcal{H} \times \mathcal{R}$ among labels, identifiers, histories of recorded actions, and recognitions by other agents. The label tells others what to call the agent, the identifier tells the system whom to authorize, the history records what has been done under the identifier, and the recognition records who treats the agent as the author of that history.

The four components can vary independently. An agent may change its label while keeping the identifier and history, as when an account is renamed. An agent may lose its identifier while retaining its history in records that mention the label, as when an account is deleted. An agent may acquire a history without recognition, as when contributions are made pseudonymously and no one links them to a person. And a recognition may attach to a label that, over time, has been controlled by several different agents. A theory of identity that collapses these components cannot describe, for instance, the difference between a renamed account and a transferred one.

Proposition 5.3 (Rename invariance and reuse hazard)**.** *Let a filesystem record ownership as a numeric identifier. Then* (*i*) *renaming the label of a user does not change the ownership relation on existing files;* (*ii*) *deleting a user and later assigning the same numeric identifier to a different user transfers ownership of all files formerly owned by the deleted user to the new user;* (*iii*) *consequently, ownership by identifier is a faithful record of authorship only under an additional assumption that identifiers are never reused.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Ownership is stored as the pair of numeric identifiers in the file's metadata, and authorization compares numbers. (i) The label resolves to the number through the account database; changing the label changes the database entry and not the numbers stored with files. (ii) After deletion, files retain the old number. When a new account is created with that number, resolution of the stored number now yields the new account. (iii) follows: the ownership relation records an identifier, and the claim that the identifier names the same agent over time is an additional assumption. □

The hazard is no mere curiosity. Hosted repository services face the same structure when an account name is released and claimed by another party: any link, citation, or attribution that depended on the label now points to a different agent, though the numeric identifier used internally by the service differs. Whether the service treats the identifier or the label as the public handle determines which of the persistence notions of Chapter 2 the public record enjoys. The matter returns in Chapter 18 and, with formal consequences for portability, in Chapter 29.

5.3 The home directory and the persistent address

The home directory is the first place at which a principal acquires a persistent association with resources. It is a location in the namespace of files that is, by convention, the default working area of the principal and the repository of that principal's configuration. The association is established by the account database and protected by the permission system, and it persists across sessions. This persistence is what allows a person to accumulate scripts, notes, and configuration over years, and it is also what makes the later problem of recall acute: the home directory is a private institution whose only member may forget its contents.

The conventional division of the home directory into visible working material and hidden configuration, the latter marked by a leading period in file names, is an instance of classification by convention. Hidden files do not receive special protection; they are simply omitted from default listings. The convention is nevertheless significant, because it separates artifacts that the person produces from artifacts that the person's tools produce about the person's preferences, a distinction that becomes the topic of Chapter 9.

Chapter 6

Permissions as an Authorization Relation

6.1 The permission check

Authorization in the traditional Unix model is determined by a short algorithm applied to a file's metadata. Each file has an owner, a group, and a mode word of which the lowest nine bits are three triples of read, write, and execute bits for the owner class, the group class, and all other users. When a process requests an operation on a file, the system selects exactly one class: the owner class if the process's effective user identifier equals the file's owner, otherwise the group class if the process belongs to the file's group, otherwise the other class. It then checks whether the selected class's bit for the requested operation is set. A privileged user, conventionally identified by the zero identifier, is exempt from most of these checks.

The selection of exactly one class is the feature of the algorithm that most often surprises newcomers. It means that permissions are not cumulative across classes: the owner is not granted whatever the group and others are granted. The following consequence illustrates this.

Example 6.1 (An owner denied what others may do)**.** Let a file have mode 007 in octal notation, so that the owner and group classes have no permissions and the other class has read, write, and execute. A process whose effective user identifier equals the file's owner is assigned to the owner class and is denied every operation, while any unrelated user is permitted all of them. The owner may restore access by changing the mode, since the right to change the mode belongs to the owner independently of the mode bits.

The example exhibits a structural feature that is the seed of governance: the right to change the rules is distinct from the rights the rules confer. A model of authorization must therefore have two levels.

Definition 6.2 (Authorization system with meta-permissions)**.** An *authorization system* consists of a set P of principals, a set O of objects, a set Op of operations, a set Π of policies, an authorization relation $\Gamma \subseteq P \times O \times \mathsf{Op} \times \Pi$ read as *principal p may perform op on o under policy π*, and a *policy-update operation* $\mathsf{chpol} \in \mathsf{Op}$ whose effect on an object o is to replace the policy governing o by another. The system has *meta-permissions* when the authorization of chpol is itself determined by Γ.

In the traditional model, the authorization of chpol on a file is hard-wired to the owner (and to the privileged user), which means the system does not allow the policy-update right to be delegated except by transferring ownership. Access control lists in Windows and in POSIX extensions generalize this by making the right to change permissions an entry like any other, so that it may be granted to several principals or withheld from the owner. The distinction

has a long reach: the history of repository systems is largely a history of increasingly fine meta-permissions, from the owner of a repository to organization roles, protected branches, and required reviews.

6.2 Expressivity limits of mode bits

Proposition 6.3 (Class limitation)**.** *Under the traditional mode-bit model, for a fixed file and a fixed operation, the principals are partitioned into three classes determined by the file's owner and group* (*owner, other members of the group, everyone else*)*, and the authorized set is a union of some of these classes, apart from the privileged user. Hence at most eight authorized sets are expressible for the operation on the file. In particular, a policy under which two principals in the same class are treated differently, or under which the authorized set is not a union of the three classes, cannot be expressed by mode bits alone, and a policy family that requires more than eight distinct authorized sets for one operation on one file cannot be expressed by mode bits on that file.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. The principals are partitioned by the algorithm into the owner class, the group class (principals other than the owner who belong to the file's group), and the remainder. The permitted set for each operation is the union of those classes whose bit for the operation is set. Every expressible authorized set is therefore determined by a choice of a subset of three classes, giving at most $2^3 = 8$ possibilities. Two principals in the same class satisfy exactly the same permission checks, so no policy that separates them is expressible, and a requirement for more than eight distinct authorized sets cannot be met. The statement concerns a fixed file and operation; the count of three classes is a property of the model and does not bound the number of principals. □

The practical remedy, creation of groups, moves complexity from the file to the group database. This remedy has costs that foreshadow later organizational problems: group membership must be maintained, the semantics of group names must be remembered, and a group created for one purpose is liable to be reused for another. Access control lists remove the bound at the cost of greater opacity: a file's effective policy is no longer visible in a single short string.

6.3 Directories, traversal, and the semantics of write

The permission bits take on distinctive meanings for directories. Read permission on a directory allows its entries to be listed; write permission allows entries to be added, removed, or renamed, provided search permission is also held; execute permission, called search permission, allows entries to be resolved by name when the name is already known. These meanings produce asymmetries that bear on discoverability. A directory with search permission but no read permission allows a person who already knows a file name to open the file but does not allow the person to find out what names exist. In effect, such a directory withholds referential persistence for agents who lack prior knowledge while preserving it for those who possess the name. The mechanism is a rudimentary form of capability: possession of the name is the authority.

A further device is the restricted deletion flag, commonly called the sticky bit, which on a shared directory restricts removal and renaming of an entry to the entry's owner, the

directory's owner, and the privileged user. It exists because the write permission on a directory, ordinarily sufficient to delete any entry within it, would otherwise allow any participant in a shared scratch space to delete the files of every other. The sticky bit is thus a small governance rule: it distinguishes the right to contribute to a shared space from the right to alter the contributions of others, a distinction that reappears in every collaboration system.

Principle 6.4 (Contribution is not control)**.** A system that allows many principals to add to a shared space must distinguish, in its authorization relation, the right to add from the right to remove or alter what others have added. Systems that do not make this distinction either forbid shared spaces or expose them to destruction by any participant. [DESIGN ARGUMENT]

Chapter 7

Paths, Files, and Classification

7.1 Files, names, and the identity of content

Early students of filesystems sometimes suppose that a file is its name. In Unix-like systems the file is an index node, a record carrying metadata and pointers to content, and the name is an entry in a directory that associates a string with an index node number. A single index node can have several names, which are called hard links, and removing a name does not remove the node until the last name is gone and no process holds it open. Renaming within a filesystem changes the directory entry and leaves the index node untouched. Symbolic links add another kind of association: a symbolic link is a small file whose content is a path, resolved each time it is followed, and which may therefore dangle if the target is moved or removed.

Definition 7.1 (Filesystem as a labelled directed graph)**.** A *filesystem* is a tuple (N, D, λ, c) in which N is a set of nodes, $D \subseteq N$ is the set of directory nodes, $\lambda : D \to (\Sigma^* \rightharpoonup N)$ assigns to each directory a partial map from names to nodes, and $c : N \setminus D \to \mathsf{Content}$ assigns content to non-directory nodes. A *path* is a finite sequence of names, and the *resolution* of a path $s_1/\ldots/s_k$ from a root $r \in D$ is $\lambda(\cdots\lambda(\lambda(r)(s_1))(s_2)\cdots)(s_k)$ when every step is defined, and undefined otherwise.

The definition models a filesystem without symbolic links: in it every non-directory node is content-bearing, and resolution is a recursive descent through directory maps. Symbolic links make resolution depend on the content of intermediate nodes, which are interpreted as paths to be resolved in turn, so that resolution may fail to terminate and a path may be valid at one moment and dangling at the next. The results below are stated for the restricted model unless a symbolic link is mentioned.

The definition makes plain that resolution is a partial function whose value depends on the whole chain of directory maps, so that a path is meaningful only relative to the state of every directory it traverses. A path is therefore a fragile reference. Moving any ancestor directory invalidates every path through it, though not the identity of the files beneath. This is the formal content of the old advice not to rely on locations, and it explains why content-addressed systems, in which the reference is derived from the content itself, behave differently under reorganization (Chapter 16).

Proposition 7.2 (Reorganization and references)**.** *Let ϕ be a rename of a directory entry. (i) Resolution of every path that does not traverse the renamed entry is unchanged. (ii) Resolution of every path that traverses the renamed entry through the old name becomes undefined, unless a new entry with the old name is introduced. (iii) The node, its content, and the resolution of paths through any hard link not involving the renamed entry are unchanged.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Each claim follows directly from Definition 7.1: resolution depends only on the maps

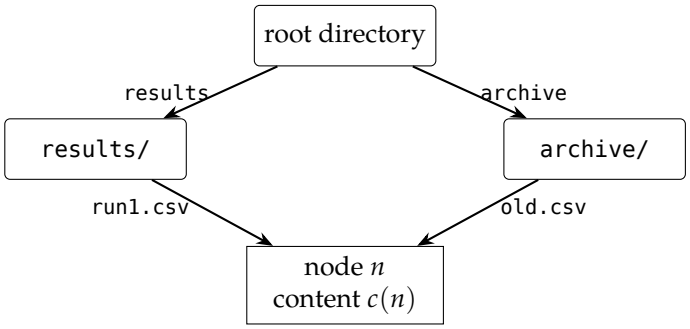


Figure 7.1: A filesystem in the model of Definition 7.1 with one node reachable under two names, which is the situation created by a hard link. The paths `results/run1.csv` and `archive/old.csv` resolve to the same node n. Renaming either entry changes the resolution of paths through it (Proposition 7.2) and leaves n and its content unchanged. The model contains no symbolic links.

$\lambda(d)$ for directories d along the path, and the rename changes exactly one of these maps at exactly one name. □

7.2 Classification by hierarchy

A directory hierarchy is a classification scheme with the restriction that each artifact receives one primary class. The restriction matters. Real artifacts have several attributes, and a hierarchy forces a choice of which attribute to use first. A scientific project may wish to organize its files by experiment, by data type, by date, or by the stage of analysis, and a tree can honor only one of these as the top-level partition. The remaining attributes are relegated to lower levels or to the file names, where they are accessible only by scanning.

Example 7.3 (Two attributes, one tree)**.** Let artifacts be indexed by pairs (u, v) with $u \in \{1, \dots, A\}$ and $v \in \{1, \dots, B\}$, one artifact per pair. A tree whose top level partitions on u and whose second level partitions on v has A directories each containing B files. A searcher who knows u opens one directory. A searcher who knows only v must open directories until finding one with the sought entry, which takes on average $(A+1)/2$ directory openings. The tree ordered by v first reverses the costs.

Proposition 7.4 (Axis ordering)**.** *Suppose the searcher knows u but not v with probability q, and knows v but not u with probability $1-q$, and never both. The expected number of directory openings for the u-first tree is $q + (1-q)\frac{A+1}{2}$, and for the v-first tree is $q\frac{B+1}{2} + (1-q)$. The u-first tree is strictly better exactly when $(1-q)(A-1) < q(B-1)$.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. For the u-first tree, a searcher who knows u opens exactly one directory. A searcher who knows only v opens directories in some order until success; with the location uniformly distributed, the expected number of openings is $(A+1)/2$. The v-first case is symmetric with A and B exchanged and the roles of the probabilities swapped. Subtracting, the difference of costs (u-first minus v-first) is

$$q\Big(1 - \tfrac{B+1}{2}\Big) + (1-q)\Big(\tfrac{A+1}{2} - 1\Big) = -\tfrac{q(B-1)}{2} + \tfrac{(1-q)(A-1)}{2},$$

which is negative exactly when the stated inequality holds. □

The condition says that the first axis should be the one the searcher is more likely to know, weighted by the fan-out of the other axis, which accords with intuition: organize first by what future searchers will be able to say. The result also shows why the problem has no universal solution. The probabilities q are properties of future searchers, which the person creating the hierarchy can estimate but not know. Systems that supplement hierarchies by tagging or by search indexes dissolve the choice of axis at the cost of building and maintaining a second structure, and that second structure is itself an artifact subject to the three kinds of persistence.

7.3 Context contributed by the path

The path contributes a substantial portion of the meaning of a file, and empirical studies of how people organize and recover files report that location serves as a reminder of the activity with which a file was associated [37]. The name `results.csv` says almost nothing, but `percolation-screening/experiments/chokepoint/results.csv` says a good deal, and this meaning is not stored in the file. It is stored in the directory structure, and if the file is copied elsewhere the meaning is lost. This is an important example of information that is held in the *relations* among artifacts and not in the artifacts themselves, and it recurs at every scale of the book. A commit's meaning depends on its position in a history; a claim's standing depends on its position in an argument; a repository's authority depends on the recognition it receives. In each case, a copy of the item without its relations is a different and poorer object.

Principle 7.5 (Context may live outside the artifact)**.** When context relevant to a class of operations is held in relations to other artifacts (directory position, links, history, citation), operations that move or copy the artifact without those relations do not preserve operational persistence for that class, even if the artifact's bytes are preserved exactly. [DESIGN ARGUMENT]

Archival practice has long recognized this under the heading of original order, the doctrine that a collection's arrangement as received carries information that should not be altered casually. The mathematical content is that any reorganization is a map on the relational structure and that preservation questions must be asked of the map, not only of the objects.

Chapter 8

Naming as Prediction

8.1 The retrieval loss of a naming policy

A name is a message to a future searcher who cannot be consulted. The message must be composed under uncertainty about the searcher's vocabulary, knowledge, and intent. The problem has a natural formalization as the minimization of expected retrieval loss.

Definition 8.1 (Naming policy and retrieval loss)**.** Let X be a finite set of artifacts, Λ a set of labels, and Q a set of queries. A *naming policy* is a map $N : X \to \Lambda$. A *retrieval procedure* for N is a map $\mathrm{Retrieve}(\cdot, N) : Q \to \mathcal{P}(X)$ returning a set of candidates. Given a joint distribution on pairs (X, Q) of target artifact and query, and a loss function $\ell(x, S)$ that penalizes failure to include x in the returned candidate set S and the size of S, the *expected retrieval loss* is

$$\mathcal{L}(N) = \mathbb{E}_{(X,Q)}\big[\ell\big(X, \mathrm{Retrieve}(Q, N)\big)\big].$$

The loss penalizes two kinds of error. Retrieving the wrong artifact, or failing to retrieve any, is the obvious kind. Retrieving a large set of candidates among which the right one must be identified at cost is the other, and in practice it is often the dominant cost, since the searcher's time is consumed by sifting. A good name does not merely allow the artifact to be found; it allows it to be recognized as the artifact being sought, among its neighbors.

8.2 Bounds on label-only retrieval

The most basic limit on naming concerns the size of the label space. A name carries information, and a naming policy that uses few distinct labels cannot distinguish many artifacts.

Proposition 8.2 (Label capacity)**.** *Let X have n elements, let the target be drawn uniformly, and let $N : X \to \Lambda$ be a naming policy that uses exactly L distinct labels. A searcher who knows the label of the target and returns a single guess succeeds with probability exactly L/n, whatever the sizes of the classes sharing a label. In particular no policy with at most L labels in use supports label-only retrieval with success probability above L/n.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Given the label, the searcher's best single guess is any element of the class carrying that label; it is correct with probability $1/|C_\ell|$ conditional on the label ℓ, which occurs with probability $|C_\ell|/n$. The unconditional success probability is $\sum_\ell \frac{|C_\ell|}{n} \cdot \frac{1}{|C_\ell|} = L/n$, where the sum is over the L labels in use. This holds for any partition into classes, equal or not, so the success probability is the number of distinct labels over n, and it cannot exceed L/n for a policy with at most L labels in use. □

The relevant quantity is the number of labels actually in use and not the size of the label alphabet. A policy that gives every artifact a unique label, so that $L = n$, supports perfect

label-only retrieval, but only for searchers who know the unique label. Unique labels are cheap to generate (a hash will do) and useless to a searcher who cannot reconstruct them. The value of a name lies in its reconstructibility from what the searcher is likely to know, which is why descriptive names are used in preference to arbitrary identifiers, and why arbitrary identifiers are used in addition to descriptive names when exactness is needed.

Proposition 8.3 (Names must be reconstructible)**.** *For a searcher with knowledge state κ and a naming policy N, let E be the event that the searcher derives $N(x)$ from κ, and let $\rho = \Pr[E]$. Assume (a) the sequential model: retrieval succeeds only if the label is first derived, so that the success event S satisfies $S \subseteq E$; and (b) given E, the searcher's success is that of a searcher who knows the label, so that $\Pr[S \mid E] \leq L/n$ by Proposition 8.2. Then $\Pr[S] \leq \rho\, L/n$.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By (a), $\Pr[S] = \Pr[S \cap E] = \Pr[E] \Pr[S \mid E]$, and by (b) the second factor is at most L/n. □

Both assumptions matter. Without (a), a searcher who fails to derive the label may still succeed by browsing or by search on content, and the bound fails. Without (b), derivation of the label may be correlated with the target in a way that favors the searcher, as when the searcher derives the label because the target is a typical member of its class. The proposition says that, within the sequential model, a naming policy cannot do better than balance label capacity against reconstructibility; whether real searchers follow the sequential model is an empirical question.

8.3 Conventions as constraints on the search space

If the future searcher cannot know the writer's naming habits, shared conventions can substitute for knowledge. A convention is a commitment, observed by many writers, to derive labels from attributes by a stated rule. A searcher who knows the rule and the attributes can reconstruct the label without remembering it.

Proposition 8.4 (Value of a convention)**.** *Let attributes $\alpha(x) \in \mathcal{A}$ be injective on X and let a convention fix a deterministic map $g : \mathcal{A} \to \Lambda$ that is injective, with $N(x) = g(\alpha(x))$. A searcher who knows $\alpha(x)$ and g retrieves x with probability one. A searcher who knows only a proper subset of the attributes succeeds with probability equal to the reciprocal of the number of artifacts consistent with the known attributes.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. With full knowledge, the label $g(\alpha(x))$ is computable and, by injectivity, identifies x. With partial knowledge, the consistent artifacts form a set S; if the target is uniform within S and the searcher guesses one element, the success probability is $1/|S|$. □

The proposition shows what a convention does and what it does not do. It converts memory of a label into computation of a label, which is a gain whenever the attributes are more memorable than the label. It does nothing for a searcher who knows too few of the attributes. For this reason the design of conventions is a matter of choosing which attributes to encode, and the choice has the same structure as the choice of tree axes in Proposition 7.4.

8.4 Names in code, in commits, and in headings

The same analysis applies, with changes of detail, to function names, variable names, commit messages, section headings, and file names. A function name is a retrieval key for a reader who is scanning a codebase for the procedure that does a thing; a good name encodes the operation and its domain. A commit message is a retrieval key for a reader who is searching history for the change that introduced a behavior. A heading is a retrieval key for a reader skimming a document for the part that answers a question.

Names also embody a theory of how the artifact will be described later. A name chosen from the writer's present vocabulary encodes the writer's present concerns; a name chosen from the vocabulary of the problem domain encodes concerns that outlive any particular author. Naming from the domain is accordingly more durable, but also more demanding, since it requires the writer to know the domain's vocabulary. Naming from the implementation is less durable, because implementations change while the problem endures. The familiar advice to name things for what they are and not for how they are done is a heuristic for increasing the reconstruction probability of Proposition 8.3 over long horizons.

Remark 8.5 (Names and the stability of reference). A name that must remain stable to support external references, such as a published function name in a library, is an interface, and changing it is an act with consequences for every dependent. Part IV shows how package ecosystems formalized this through version numbering conventions, and Part VI treats name stability as a component of institutional continuity.

Part III

The Discipline of Reuse

Chapter 9

Scripts, Templates, and Portable Configuration

9.1 The script as externalized procedure

A script is a procedure that has been written down so that it need not be performed from memory. Its first function is labor saving: a sequence of commands that would be typed repeatedly is typed once. Its second function, which is more important for the argument of this book, is that it records a procedure in a form that can be inspected. A script that renames a thousand files according to a rule documents the rule more exactly than any prose description could, and it does so in a notation whose meaning is fixed by an interpreter. The record is executable, and so it is checkable by running it. This dual character, being both a tool and a statement of method, is why scripts occupy a central place in the passage from personal practice to reproducible science.

A script nevertheless inherits all the persistence problems of Chapter 2. It may exist (physical persistence) while its location is forgotten (referential persistence); it may be found while the assumptions about the environment in which it ran have been lost (operational persistence). A script that reads from a hard-coded path, depends on an environment variable set in a long-closed shell session, or invokes a tool that has changed its option syntax may be inspectable and not executable. The common cause is that the script's text records the *operations* while the *conditions* under which the operations were valid are held elsewhere.

Definition 9.1 (Implicit and explicit preconditions)**.** For a script s and an environment e, let $\mathrm{Pre}(s)$ be the set of conditions on e under which s executes as intended. The *explicit preconditions* $\mathrm{Pre}_{\mathrm{exp}}(s) \subseteq \mathrm{Pre}(s)$ are those recorded in the script or in artifacts bundled with it, and the *implicit preconditions* are the remainder. The script is *self-describing* when $\mathrm{Pre}_{\mathrm{exp}}(s) = \mathrm{Pre}(s)$.

Self-description is an ideal, seldom attained, but it is the measure against which practices can be compared. Checks at the head of a script that verify the presence of required tools and exit with an informative message convert implicit preconditions into explicit ones; declarations of required versions do the same; and strict modes in which undefined variables cause errors turn some implicit preconditions into detectable failures.

9.2 Idempotence and the safe rerun

A procedure intended for repeated use should behave well when run more than once, since a person who is unsure whether it has been run is likely to run it again. The mathematical notion is idempotence.

Definition 9.2 (Idempotent operation)**.** An operation f on states of an environment is *idempotent* if $f \circ f = f$. A family of operations $\{f_i\}$ is *commuting* if $f_i \circ f_j = f_j \circ f_i$ for all i, j.

Proposition 9.3 (Composition of idempotent operations)**.** *(i) If f and g are idempotent and commute, then $f \circ g$ is idempotent. (ii) Without commutation, the composition of two idempotent operations need not be idempotent. (iii) If f is idempotent, then for every state e the state $f(e)$ is a fixed point of f, and iterating f never changes the outcome after the first application.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) $(fg)(fg) = f(gf)g = f(fg)g = (ff)(gg) = fg$ using commutation and idempotence. (iii) is the identity $f(f(e)) = f(e)$. For (ii), take states $\{0, 1, 2\}$ with f defined by $f(0) = 1$, $f(1) = 1$, $f(2) = 2$, and g by $g(0) = 0$, $g(1) = 2$, $g(2) = 2$. Both are idempotent. Then $fg(0) = f(0) = 1$, $fg(1) = f(2) = 2$, so $(fg)(fg)(0) = fg(1) = 2 \neq fg(0) = 1$. □

The failure of idempotence under non-commuting composition explains why setup scripts that are composed of individually safe steps can still be unsafe to rerun: the order of steps matters, and the second run starts from a state that the first run altered. The practical discipline is to make each step a check followed by an action, so that a step that has already taken effect becomes a no-op, and to order steps so that later steps do not invalidate the checks of earlier ones.

9.3 Dotfiles and the portable environment

Configuration of a working environment accumulates in files that tools read at startup: shell initialization, editor settings, version control preferences, terminal behavior. In Unix-like systems these are conventionally hidden files in the home directory, and collectively they are called dotfiles. They constitute the personal institution of a user in an unusually literal sense: they record that user's standing decisions about how work is to be done.

Dotfiles are among the first artifacts a user places under version control, because loss of them is felt immediately and because their small size and textual form make them well suited to version control. The practice of keeping dotfiles in a repository and installing them on new machines by a bootstrap script is, in the vocabulary of this book, an environment-reconstruction procedure. Its quality is judged by how fully it restores the environment from the repository alone.

Definition 9.4 (Reconstruction map)**.** Let Rep be the set of states of a configuration repository and Env the set of environment states. A *reconstruction map* is a function $\beta : \mathsf{Rep} \times \mathsf{Env}_0 \to \mathsf{Env}$ that applies the repository's bootstrap procedure to a base environment $e_0 \in \mathsf{Env}_0$. The repository is *sufficient* for a target environment e^* relative to a class Env_0 of base environments if $\beta(r, e_0) \simeq_K e^*$ for all $e_0 \in \mathsf{Env}_0$, where $\simeq_K$ is observational equivalence for an operation class K: for a family of observations $\mathrm{Obs}_K(e)$ (the outputs and visible effects of each operation of K on each input in a stated test family), $e \simeq_K e'$ holds when $\mathrm{Obs}_K(e) = \mathrm{Obs}_K(e')$. Exact equality $e = e^*$ implies $e \simeq_K e^*$ for every K and is strictly stronger; two environments with different installed but unused packages may be equivalent for K and unequal.

The base class Env_0 is the quantity that matters. A bootstrap script that works from a freshly installed system of a particular distribution, with network access, is sufficient for a narrow class; a script that also works when a package index has changed, a tool has been removed,

or a different operating system is in use is sufficient for a wide one. Preference for default configurations, with as little customization as the work allows, is a strategy that widens the class: the less the environment departs from defaults, the more machines already satisfy the target, and the fewer steps the bootstrap needs to carry. This is a kind of conservatism with a precise justification, namely that every deviation from defaults is an item of context that must be preserved and that a future environment may lack.

9.4 Templates and the reuse of structure

A template is a script's cousin for documents and projects: a pre-built structure with places to fill in. Templates carry conventions, so that a new project begins with a README, a license file, a layout, and a place for tests. They are the practical means by which conventions propagate. A template does for the structure of an artifact what a naming convention does for its label (Proposition 8.4): it replaces a decision to be remembered with a decision to be recognized.

The risk of templates is cargo-cult adoption, in which the structure is retained while its function is forgotten. A project may carry a contributing guide that nobody maintains and a test directory that tests nothing. The distinction between structure and function is again the distinction between physical or referential persistence and operational persistence: the file exists and can be found, but the practice it was meant to support has lapsed. The discipline of reuse thus consists not in the accumulation of templates but in the continual checking that the practices they encode are still operative.

Chapter 10

Documentation as Communication with an Absent Agent

10.1 The manual page tradition

The manual page is one of the most successful documentation genres in computing. Its fixed sections, which in the conventional layout include a name line, a synopsis, a description, a list of options, examples, related commands, and notes on known defects, give the reader a predictable route to the information needed: what the command is called and does, how it is invoked, what each option means, and where to look next. The predictability is the point. A reader encountering an unfamiliar command can navigate its manual page without having to read it, because the structure of every page is the same.

The structure of the genre embodies a theory of what a user of a tool needs to know. The synopsis answers *how do I call this*; the description answers *what does it do*; the options answer *what can I vary*; the examples answer *what does typical use look like*; the related-commands section answers *what else is relevant*. These are different kinds of information, and an undifferentiated essay that mixes them forces the reader to extract each from a narrative. In the vocabulary of Definition 4.2, the manual page is a partitioning of context items by type, so that a reader seeking a particular type can find it without reading the rest.

10.2 Contracts and what the code does not say

Function-level documentation conventions, including documentation comments with sections for arguments, return values, and exceptions, serve a similar role at smaller scale. The strongest form is the specification of a procedure as a contract: a statement of what the caller must ensure (the precondition), what the procedure guarantees on return (the postcondition), and what remains unchanged (the frame). In Hoare's notation [19], a correctness statement has the form $\{P\}\ C\ \{Q\}$, meaning that if the precondition P holds before the command C is executed and C terminates, then the postcondition Q holds afterward.

The importance of contracts for the present argument lies in what they leave out. A contract describes an interface and says nothing about how the interface is realized. This is valuable for use, since a caller needs only the interface, and troublesome for modification, since a maintainer needs to know why the implementation takes the form it does. Two different implementations may satisfy the same contract while differing in running time, memory use, numerical stability, ordering of side effects, and behavior outside the specified domain, and a modification that preserves the contract may destroy properties the contract never mentioned.

Proposition 10.1 (Contracts underdetermine safe modification)**.** *Let C_1 and C_2 be two programs satisfying the same contract $\{P\}\ \cdot\ \{Q\}$ and differing in some property Φ not implied by the contract*

(*that is, C_1 has Φ and C_2 does not, and neither P nor Q entails Φ*). *Then replacing C_1 by C_2 preserves all properties entailed by the contract and may fail to preserve Φ. Consequently, the contract alone is not a sufficient context for the operation class* modify while preserving Φ. [PROVED UNDER STATED ASSUMPTIONS]

Proof. Every property entailed by the contract holds for any program satisfying it, hence holds for C_2 if it held for C_1. By hypothesis Φ holds for C_1 and not C_2, so the replacement does not preserve Φ. A context that does not mention Φ cannot distinguish the safe replacement from the unsafe one, so it is not sufficient for a class of operations whose correctness criterion includes Φ. □

This is the formal form of the observation that documentation must record the *why* as well as the *what*. The reasons an implementation was chosen are the properties Φ that the contract omits: that an algorithm was selected for numerical stability, that a loop order was chosen for cache locality, that a data structure was kept because an external consumer depends on its ordering. These are rationale items in the information universe I, and they are the items most easily lost, since they are not needed to make the code run.

10.3 History as documentation

Version control adds a second documentary channel: the record of changes. A commit consists of a snapshot or difference together with a message, and the message is the place for explaining why the change was made. But the difference itself does not determine the explanation.

Proposition 10.2 (Intent is not a function of the change)**.** *Let* Intent *denote the set of intentions with which a change may be made, and let δ be a textual difference between two versions. There exist differences δ and distinct intentions $\iota_1 \neq \iota_2$ such that both are consistent with δ. Hence no function from differences to intentions is correct in general.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Consider the difference that replaces the constant 0.05 by 0.01 in a threshold. One author may intend to tighten the statistical significance level, another to correct a transcription error from a source that gives 0.01, a third to tune a parameter against a validation set. All three produce the identical difference. A function from differences to intentions would assign a single value to this difference, hence be wrong for at least two of the three authors. □

A commit message that says *change threshold* restates the difference. A commit message that says *use the significance level stated in the protocol; the previous value was a transcription error* records the intention, and the intention is exactly what the difference cannot supply. This is the reason that disciplined projects ask for descriptive messages and that tools which generate messages automatically from differences are of limited documentary value: they add no information beyond what the difference contained.

10.4 Documentation decay

Documentation can be wrong. A comment that describes code as it was before a change is worse than no comment, because it misleads a reader who trusts it. The risk is greatest for documentation that is physically separate from the thing described, since there is no

mechanical connection forcing the two to change together. Remedies fall on a spectrum. At one end, documentation embedded in code, as comments or documentation strings, is edited when the code is edited, with some probability. Further along, documentation that is executable, in the form of examples that are run as tests, fails loudly when it diverges from the behavior it describes. At the far end, the specification and the implementation are derived from a single source, so that divergence is impossible by construction.

Definition 10.3 (Documentary coupling)**.** The *coupling* of a documentation item d to an artifact x is the probability that a change to x affecting the content of d is accompanied by a corresponding change to d. Documentation is *mechanically coupled* when coupling is one by construction (for example, generated from the artifact), *checked* when divergence triggers a failure signal (for example, a doctest), and *free* otherwise.

The three levels have increasing reliability and increasing cost in flexibility. Mechanically coupled documentation can only say what the artifact determines, which excludes rationale; checked documentation can say more, but only things that tests can verify; free documentation can say anything and is trusted only as far as its maintainers are diligent. A project that wishes its documentation to survive long horizons will use all three, assigning each kind of information to the level that suits it: signatures and types to the mechanical level, usage examples to the checked level, and rationale to the free level, with the understanding that the free level is the most fragile and the most valuable.

Chapter 11

Dependencies and Environments

11.1 The hidden context of imported code

An artifact that imports another artifact has acquired a dependency, and the dependency is part of the context needed for any operation on the first. A script that uses a numerical library cannot run without the library, and the behavior of the script depends on the library's version, on the versions of the libraries the library itself uses, and on the system on which they are all installed. The dependency graph of a nontrivial program is large, deep, and mostly unexamined by its author. The result is a distinctive failure mode: a program that worked for years stops working, not because it changed, but because something it depends on, directly or transitively, changed beneath it.

Definition 11.1 (Dependency graph and closure)**.** A *dependency graph* is a directed graph $G = (V, E)$ whose vertices are artifacts together with version identifiers and whose edges $u \to v$ record that u requires v. For a set $S \subseteq V$ of artifacts, the *closure* $\mathrm{cl}(S)$ is the set of vertices reachable from S. A deployment is *closed* if every artifact that it uses at run time belongs to $\mathrm{cl}(S)$ for the declared set S of direct requirements. The *undeclared dependencies* of a run are the artifacts it reads that lie outside $\mathrm{cl}(S)$.

The distinction between declared and actual dependencies is critical. A script may import a module that happens to be installed on the author's machine through an unrelated project; the dependency is real, since the script fails without it, and undeclared. Undeclared dependencies are among the most common reasons why work that functions on its author's machine fails elsewhere.

11.2 Version constraints and resolution

Dependencies are rarely stated as exact versions. Most package managers allow ranges, such as any version at least a given number but below the next major release. The practice trades precision for flexibility: a range allows the ecosystem to supply bug fixes without requiring each consumer to update its declarations. The cost is that the meaning of a declaration depends on the state of the registry at the time of resolution.

Definition 11.2 (Resolution function)**.** Let Reg_τ be the state of a package registry at time τ, and let Req be a set of requirements with version constraints. A *resolution procedure* is a function ϱ that maps $(\mathsf{Reg}_\tau, \mathsf{Req})$ to a set of pinned artifacts satisfying the constraints, or to failure.

Proposition 11.3 (Resolution depends on time)**.** *In general,* $\varrho(\mathsf{Reg}_\tau, \mathsf{Req}) \neq \varrho(\mathsf{Reg}_{\tau'}, \mathsf{Req})$ *for* $\tau \neq \tau'$*. A pinned specification that names each artifact by content hash is invariant: it resolves to*

the same artifacts at every time at which those artifacts remain obtainable. [PROVED UNDER STATED ASSUMPTIONS]

Proof. For the first claim, let Req require a library at least version 1.0. If at time τ the registry's latest release is 1.0 and at τ' it is 1.1, a procedure that selects the latest satisfying version returns different results. For the second, a content hash determines the artifact (under collision resistance), so the specification does not depend on registry state, only on the availability of an artifact with that hash. □

The proposition is the formal content of the practice of lock files, which record the resolved versions, and, in stronger form, of hash-pinned requirements. A lock file converts a resolution that depends on time into one that does not, at the price of freezing the ecosystem's improvements until the lock is deliberately refreshed.

11.3 The computational difficulty of resolution

The difficulty of dependency resolution is not merely organizational. When versions can conflict, so that two packages cannot both be installed, the problem of finding a consistent installable set is computationally hard. The hardness is a classical result in the literature on package-management systems.

Theorem 11.4 (Hardness of installability)**.** *Deciding whether a given package can be installed from a repository in which packages declare dependencies as conjunctions of disjunctions of versioned packages, together with pairwise conflicts, is NP-complete.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Membership in NP holds because a proposed installation set can be checked for dependency satisfaction and absence of conflicts in polynomial time. For hardness, reduce from 3-SAT. Given a formula with variables $x_1, \dots, x_k$ and clauses $C_1, \dots, C_n$, introduce for each variable a package pair $x_i^{\top}$ and $x_i^{\bot}$ declared in conflict, so that at most one of them is installed. For each clause introduce a package c_j whose dependency is the disjunction of the packages corresponding to its literals. Introduce a root package that depends on all variable choices (a disjunction $x_i^{\top} \vee x_i^{\bot}$ for each i) and on all clause packages. The root is installable iff there is an assignment (choice of one package per variable) satisfying every clause. □

The result is attributed, in various forms, to work on package-based distributions and on dependency solving as a separate concern (see [23, 1]). It explains why modern package managers incorporate satisfiability solvers, and why the existence of a compatible set of versions is a different matter from its discoverability by naive procedures. It does not imply that practical instances are hard, since real dependency graphs have structure that SAT solvers exploit, but it does set the baseline: a theory of reproducible environments cannot assume that a consistent dependency assignment can be found cheaply by inspection, or that, once found, it is unique.

11.4 Environments as artifacts

Because the behavior of a program depends on its environment, the environment itself must be treated as an artifact if the program's behavior is to be reproduced. Environment descriptions can be of several kinds: a list of installed packages with versions; a container image that

bundles the user-space filesystem; a declarative description from which an environment is built; or a complete system image. They differ in how much of the environment they capture, in how much they depend on external registries at rebuild time, and in how readable they are. The container image is the most complete and the least inspectable; the declarative description is the most readable and the most dependent on external state.

Proposition 11.5 (Completeness-inspectability tradeoff)**.** *Among environment representations, one that stores the complete binary content of the environment is independent of external registries but supports inspection only through additional tooling, while one that stores only requirements is directly inspectable but is subject to the time dependence of Proposition 11.3. Hybrid representations that store a declarative description together with content hashes of the resolved artifacts recover time independence for those artifacts that remain obtainable while preserving inspectability.* [DESIGN ARGUMENT]

The design argument is simple and its force is practical. A representation that records both the intended requirements and the resolved artifacts supports two classes of operations: reproduction of the original environment where possible, and informed reconstruction where it is not. The second class is the one that matters when an artifact has been withdrawn from a registry, since the declarative layer says what was needed even when the binary layer cannot be satisfied.

Chapter 12

Conventions, Noise, and the Economy of Reading

12.1 More context can mean less usability

Definition 4.2 assumed that sufficiency is monotone: adding information to a context never harms. That assumption is false in practice, because readers have finite budgets. A repository with ten thousand undifferentiated files and a one-line description is not better documented than one with a short README, even though the first contains strictly more information. The additional information is noise, in the sense that its presence raises the cost of finding what is needed.

Definition 12.1 (Budgeted reading)**.** Let a preserved context be a finite sequence $m = (i_1, \ldots, i_k)$ of items, each with reading cost $\gamma(i_j) > 0$, and let a reader have budget β. The reader processes the items in order until the budget is exhausted. A class of operations K requires a set $\mathrm{Need}(K) \subseteq I$ of items; the reader *succeeds* if every item of $\mathrm{Need}(K)$ is processed within the budget.

Proposition 12.2 (Noise can destroy sufficiency)**.** *Under budgeted reading, adding items to a context can cause a reader who previously succeeded to fail. In particular, if irrelevant items are placed before a needed item, then for a sufficiently large total cost of irrelevant items, a budget that sufficed before the addition no longer suffices.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let $m = (r)$ with r needed and $\gamma(r) = 1$, and let $\beta = 1$. The reader succeeds. Let $m' = (n_1, \ldots, n_k, r)$ with irrelevant items of unit cost. The reader exhausts the budget after processing n_1 and fails for any $k \geq 1$. □

The monotone idealization is thus recovered only when the reader has an effective way of skipping irrelevant items, which is to say, when the context is structured so that relevant items can be located without processing the rest. Indexes, tables of contents, section headings, and naming conventions are mechanisms for providing this.

12.2 Ordering by expected relevance

If the order of items matters, the question arises of which order is best. In the simplest case, in which a reader needs exactly one item, drawn from a known distribution, and processes items in sequence, the optimal ordering has a classical form.

Proposition 12.3 (Optimal ordering for single-item need)**.** *Let the needed item be item i with probability w_i, with unit reading costs and budget B (an integer). A reader processing items in the order π succeeds with probability $\sum_{j \leq B} w_{\pi(j)}$, and this is maximized by ordering items by nonincreasing w_i. The maximum equals the sum of the B largest weights.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Success occurs iff the needed item is among the first B in the order, which has probability equal to the sum of the weights of those items. The sum of any B distinct weights is at most the sum of the B largest, and ordering by nonincreasing weight places the B largest first. □

The result justifies the near-universal practice of putting a summary first. A README is a convention that places the most frequently needed items at a standard position. Its authority comes from its being the first thing a reader opens, and its usefulness from the high weight of orienting information in the distribution of needs. Conventions of this sort effectively fix the prior w across a community, so that writers need not estimate it separately for each artifact.

12.3 Standard files as institutional organs

Repositories of software have developed a small set of conventional files whose names and places are recognized by readers and by tools: a README that orients the reader, a license that states the terms of reuse, a changelog that records notable changes between releases, a contributing guide that states how outsiders may propose changes, and a code of conduct that states expectations of participants. Each answers a question that a prospective user or contributor will ask, and each does so at a predictable place. Taken together they are the organs by which a repository communicates with newcomers.

From the standpoint of the formalism, each such file is a context item of a distinct type, placed at a conventional position to reduce search cost. The license file deserves particular note: it is a context item without which a large class of operations, including copying, modification, and redistribution, is not legally permitted, regardless of whether it is technically possible. Technical possibility and legal permission are independent, and the license is the artifact that carries the latter. An archive of code without licenses is physically and referentially persistent and, for many classes of operations, not operationally persistent in the legal sense.

12.4 Closing the first three parts

The preceding chapters have established a vocabulary: three kinds of persistence, the recovery model, the hierarchy of continuation classes with minimal sufficient contexts, the structure of namespaces and permissions, the economics of naming, the layering of documentation, and the dependence of artifacts on environments. The claims made so far are modest, and several are elementary, but they have the useful property of being stated sharply enough to be applied to institutions. The next part turns from the formal vocabulary to its historical development, tracing how the problems identified here were met, imperfectly, by successive generations of tools, until they became the problems of repository infrastructure.

Part IV
The History of Software Continuity

Chapter 13

Software Before Persistent Distribution

13.1 Programs as physical objects

The earliest programs were configurations of hardware. Plugboards, switches, and punched media held them, and a program could be transferred only by moving or reproducing the medium. Stored-program machines changed the representation but not at first the economics: programs were written for a specific machine, held on cards, paper tape, or magnetic tape, and distributed within user communities by physical exchange. Cooperative user groups for early commercial machines circulated program libraries on media, and the practice of sharing source was ordinary because the programs were written for particular installations by the people who operated them. Source and executable were not separated in the way that later commercial practice would separate them: the thing that was shared was typically what could be read.

Successive media improved density, cost, and ease of copying. Magnetic tape allowed large collections to be moved; floppy disks, introduced in the early 1970s and refined through several formats, put a duplicable medium within reach of individuals; cassette tape served early hobbyist machines. The consequence for continuity was ambivalent. Cheaper, more durable media increased the number of copies in circulation and the probability that some copy survived, and at the same time they multiplied versions, since each copying event was an opportunity for modification, corruption, and loss of the information about origin.

13.2 Networks of exchange

The next layer was electronic. Bulletin board systems, of which the best known early example was the computerized bulletin board begun in Chicago in 1978, allowed users to dial a machine and download programs. File transfer protocols, whose early specifications date to the first years of the ARPANET and whose long-lived standard was published in 1985, supported archives of software on networked hosts. Distributed discussion systems carried source code posted to special groups, some moderated so that postings were collected as complete, buildable packages. Compilation archives, shareware collections, and later commercial and noncommercial software repositories on optical discs extended the pattern. The network made a program discoverable, in the sense that a person who knew where to look could obtain it, but it did little for provenance. A file named for a program on an archive host might be the latest version, an old version, a modified version, or an unrelated file.

The commercial counterpart was the separation of the executable from the source. Software sold as a product was typically sold as an executable, with source retained by the vendor. The user possessed a thing that could be run and could not be read in the sense that matters for maintenance. This distinction is crucial for the theory of continuity, because the map from source to executable is many-to-one.

Proposition 13.1 (Executables underdetermine sources)**.** *Let* $c : \mathsf{Src} \to \mathsf{Exe}$ *be a compilation map. If* c *is not injective, there is no function* $d : \mathsf{Exe} \to \mathsf{Src}$ *with* $d \circ c = \mathrm{id}$*. Compilation maps of practical compilers are not injective: they discard comments, identifier names (in stripped binaries), formatting, and distinctions erased by optimization.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Suppose $c(s_1) = c(s_2)$ with $s_1 \neq s_2$. A left inverse d would give $s_1 = d(c(s_1)) = d(c(s_2)) = s_2$, a contradiction. Two sources differing only in a comment compile to the same executable under any compiler that discards comments, so c is not injective. □

The proposition has a historical corollary. A collection of executables is a collection of artifacts for which the highest operation class available to an ordinary user is execution (Definition 4.1), and perhaps inspection of a representation that is not meant for human reading. Safe modification and reproduction require the source and the build context, which were not distributed. Decompilation can recover a program of equivalent behavior in some cases, but it recovers neither the author's identifiers nor the comments nor the rationale, which is to say, nearly the entire contextual layer that matters for continuation.

13.3 The problem of identification

When software circulates through informal channels, the question of what a given file is becomes nontrivial. Identification under incomplete provenance is the problem of deciding, for a given artifact and a given claim about its identity, whether the claim is true.

Definition 13.2 (Provenance record)**.** A *provenance record* for an artifact x is a tuple $\pi(x) = (\mathrm{prod}, \tau, \mathrm{src}, \mathrm{der})$ recording the producing agent, the time of production, the artifacts from which x was derived, and the derivation procedure. A record is *incomplete* if one or more components are unknown or unverifiable.

An identification policy decides whether two artifacts or an artifact and a description are the same. The simplest policy identifies by name. Name-based identification fails in two opposite ways: the same artifact may be known under several names, and different artifacts may share a name.

Proposition 13.3 (Names are neither sound nor complete)**.** *Let* Univ *be a universe of artifacts that allows* (*a*) *renaming, in which one artifact acquires an additional name, and* (*b*) *name reuse, in which one name is applied to different artifacts at different times or places. Then no identification policy based on name alone is both sound* (*never identifying distinct artifacts*) *and complete* (*always identifying copies of the same artifact*)*.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By (b) there are distinct artifacts $x \neq y$ bearing the same name n; a policy that is complete with respect to names identifies x and y, which violates soundness. By (a) there is an artifact z with names $n_1 \neq n_2$; a policy that is sound for names, identifying only artifacts with the same name, fails to identify the copy under n_1 with the copy under n_2, which violates completeness. Both properties cannot hold. □

The way out, which becomes decisive later, is to identify by content. A checksum or cryptographic hash identifies an artifact by a function of its bytes, so that identical content gives identical identifiers regardless of name, and differing content gives different identifiers up to the collision resistance of the function. Content identification solves the identity problem

for byte-level equality at the cost of making any change, however trivial, a change of identity. It says nothing about authorship, purpose, or derivation, which are the rest of provenance.

Chapter 14

The Dependency and Version Problem

14.1 Shared libraries and the single-slot namespace

As software grew in size, programs came to be assembled from components, and economy of storage and update suggested that common components be shared. A shared library is installed once and used by many programs; fixing a defect in it fixes all programs that use it. The arrangement has a cost, which became widely known in the era of dynamic link libraries on desktop systems: a program is written against a version of a library, and installing a program may replace the shared copy with another version that other programs cannot tolerate. The colloquial name for the resulting class of failures was DLL hell, and the phenomenon was not specific to one vendor's platform.

The structure of the problem is a namespace in which the library is addressed by name alone. A single slot exists for each name, and installers compete for it.

Proposition 14.1 (Single-slot conflict and its resolution)**.** *Let programs $P_1, \dots, P_k$ each require a library of name ℓ at a version in a set V_i of acceptable versions.* (*i*) *If the system admits one installed version per name and $\bigcap_i V_i = \varnothing$, then the programs cannot all function.* (*ii*) *If the system keys installed libraries by the pair* (*name, version*)*, then each program can be supplied with a version in V_i, subject only to the availability of that version.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) A single installed version v must belong to every V_i for all programs to function, which requires $v \in \bigcap_i V_i$, impossible when the intersection is empty. (ii) With composite keys, each program is directed to the entry for a version in its own V_i, and entries for different versions do not collide. □

The remedies that emerged corresponded to the two halves of the proposition. Side-by-side installation, in which the version becomes part of the key, resolves conflicts at the price of storage and of multiplying the number of copies that require security updates. Isolation, in which each program bundles its own copies of libraries, achieves the same effect by a different route and is the basis of modern container images. Both abandon the single-slot sharing that created the problem. A third approach, the package manager, keeps shared installation but computes a consistent selection, which is where the complexity result of Theorem 11.4 applies.

14.2 Registries and package managers

Two components of the modern ecosystem are often conflated. A *registry* is a service that stores published packages and metadata, and answers queries about available versions. A *package manager* is a client tool that reads a project's declared requirements, queries one or more registries, resolves a consistent set of versions, retrieves them, and installs them. The

Python Package Index and the tool commonly used with it, the Node ecosystem's public registry and its client, and the Rust crates registry and its build tool all illustrate the division. System-level package managers of Unix-like distributions combine both roles within the distribution's own infrastructure, with curated repositories maintained by the distribution.

The separation matters because the registry carries a risk that the client cannot eliminate: it determines what exists. A package that is withdrawn from a registry disappears from the world of all projects that did not retain a copy. The best-known illustration is an incident of 2016 in which the author of a very small and very widely used package removed it from the public Node registry and builds across the ecosystem failed. The package consisted of a few lines. Its removal demonstrated that a dependency graph is only as stable as its least-controlled node, and that the transitive closure of a project may contain components that the project's authors have never heard of. Registry policy was subsequently changed to restrict the removal of packages that others depend on, an example of institutional response to an infrastructural failure.

Definition 14.2 (Registry dependence)**.** A project has *registry dependence* on a registry $\mathcal{G}$ if some artifact in the closure of its requirements is obtainable only from $\mathcal{G}$. The project has *unmirrored registry dependence* if, in addition, no copy of that artifact is retained in the project's own storage.

Registry dependence is not in itself a defect, since it is the point of a registry. Unmirrored registry dependence is a vulnerability in the sense of Chapter 2: it leaves referential persistence in the hands of a third party. Vendoring dependencies, mirroring registries, and hash-pinning with local caches are the corresponding mitigations.

14.3 Semantic versioning as a convention

Version numbers began as labels and became a communication channel. A widely adopted convention, proposed in the early 2010s, gives numbers three components, a major, a minor, and a patch number, and prescribes that the major number is incremented for incompatible changes to the public interface, the minor for backward-compatible additions, and the patch for backward-compatible corrections. The convention allows a consumer to state a requirement like *any version with major number two*, relying on the promise that no such version will break the consumer.

The promise is a claim about behavior, and claims about behavior of programs are subject to Rice's theorem [29]. The convention is therefore a social contract and not a verified guarantee.

Proposition 14.3 (Compatibility cannot be decided mechanically)**.** *For programs in a Turing-complete language, there is no algorithm that decides, given two versions p and p' of a library, whether p' is backward compatible with p, in the sense that every client program that terminates correctly with p also terminates correctly and with the same observable result with p'.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Reduce the halting problem. Given a Turing machine M and input w, let p be a library exposing a function f that returns the constant 0. Let p' be identical except that f first simulates M on w and then returns 0. A client that calls f and prints the result behaves identically under p and p' exactly when M halts on w (when M does not halt, the client under p' does not terminate while under p it does). A decision procedure for compatibility would decide the halting problem. □

The proposition does not make the convention worthless; it places the responsibility for compatibility on the humans who publish and on the test suites that approximate the semantics. Type systems and interface checkers verify a decidable subset, namely syntactic compatibility of signatures, which is necessary and far from sufficient. A mature ecosystem combines the convention, the checkers, and the experience of maintainers, with the understanding that none of them replaces the others.

Chapter 15

Version Control Before Git

15.1 The first systems

The earliest widely used systems for tracking changes to source files appeared in the 1970s and early 1980s. One system developed at a large telephone-company research laboratory [30] stored successive versions of a file as a compact record from which any version could be reconstructed, together with who made each change and when. Another, developed in the early 1980s [35], reversed the direction of storage: the newest version of a file was stored in full and older versions as differences, on the grounds that the newest version is the one most often requested. These systems operated on individual files, with locking to prevent simultaneous editing: a user who wished to change a file checked it out under a lock, edited it, and checked it in, at which point the lock was released.

The per-file model and the locking discipline reflect the working conditions of their period: small teams on shared machines, expensive storage, and the expectation that two people would not edit the same file at the same time. As teams grew and machines were networked, the locking discipline became a bottleneck, and the per-file model hid something important, namely that the unit of meaningful change is often a set of coordinated modifications across several files.

15.2 Concurrent versioning and the repository state

A system that became the dominant open-source tool of the late 1980s and 1990s replaced exclusive locking with concurrent editing and merging [5]. Each developer worked on a private copy, updated it from a central server, and resolved conflicts when changes overlapped. The model scaled better. Its limitation was that the server still tracked revisions per file, and a commit that touched several files was executed as a sequence of separate operations. Between the first and last, the repository was in a state that no author had intended.

Proposition 15.1 (Intermediate states under non-atomic commit)**.** *Suppose a commit consists of k independent file updates applied one at a time to a shared repository. (i) If the order of application is fixed, an observer can see $k+1$ distinct repository states associated with the commit, of which exactly two correspond to states intended by the author. (ii) If the order is unconstrained, an observer can see any of 2^k subsets of the updates applied, of which exactly two correspond to intended states. (iii) If the commit is atomic, exactly two states are visible, before and after.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. In (i), after applying the first j updates for $j = 0, 1, \ldots, k$ the repository is in state S_j, giving $k+1$ states; only S_0 and S_k were intended. In (ii), any subset of the updates may have been applied when the observation is made, giving 2^k states, again with only the empty subset and the full set intended. In (iii), by definition, no intermediate state is observable. □

The consequences are practical and historical. Builds from a repository in an intermediate state fail or, worse, succeed with a mixed program. Revision numbers attached to files, which differed from file to file, made it impossible to name the state of the project as a whole except by a symbolic tag applied to every file. A successor designed in the 1990s and 2000s to address these shortcomings introduced atomic commits and global revision numbers, so that a single integer identified a complete repository state. This was a real advance in referential persistence: a person could say *revision 1432* and have a determinate meaning, provided that the repository itself remained available and unrewritten.

15.3 Centralized coordination and centralized storage

All the systems of this era shared a design feature: a central server held the authoritative repository. Developers held working copies, but the history lived in one place. This coupled two functions that are logically distinct. The first is *coordination*: deciding what the official history is and who can append to it. The second is *storage*: holding the bytes of the history and serving them. In a centralized system both reside in the server. A server failure threatens storage, a server administrator controls coordination, and a person without commit rights cannot keep a history of local work that is integrated with the official one.

Definition 15.2 (Coordination and storage)**.** For a version control system, *storage centralization* is the property that a single host holds the only complete copy of the history. *Coordination centralization* is the property that a single party decides which histories are authoritative. The two properties are independent: a system may have centralized storage and distributed coordination, or distributed storage and centralized coordination.

The distinction is easy to lose and is crucial for what follows. The distributed systems that appeared in the 2000s, among them systems named Monotone, Darcs, BitKeeper, Mercurial, and Git, eliminated storage centralization by giving every working copy the full history. They did not eliminate coordination centralization, and in practice most projects using them continued to designate a single repository as authoritative. The designation became a matter of convention and social recognition and not of technical necessity. The next two chapters examine how this convention arose and what supports it.

15.4 The circumstances of Git

The distributed system that became dominant was created in 2005 by the lead developer of the Linux kernel after a dispute over the license of the proprietary system that the kernel project had been using. The new system was designed rapidly, with goals that included speed, distributed operation, strong integrity guarantees, and the ability to handle a very large project. It was released in the spring of that year and reached its first stable release before the year ended. Its design choices, which are the subject of the next chapter, reflect the working practices of a project with thousands of contributors, many maintainers, and a hierarchical pattern of patch review and integration.

A detail of the history deserves emphasis. The dispute that precipitated Git was a dispute about the terms on which a tool could be used to hold the project’s own history. A community that had built a body of work found that the infrastructure holding the record of the work was

subject to the licensing decisions of a third party. The response, to create an infrastructure whose license and design allowed no such dependence, is an instance of the pattern examined in Chapter 29: infrastructural independence is valued most when it has recently been lost.

Chapter 16

Git and the Content-Addressed Object Graph

16.1 Objects and identifiers

Git stores a project's history as a collection of immutable objects [7], each identified by a cryptographic hash of its contents. There are four kinds. A *blob* holds the bytes of a file. A *tree* lists the entries of a directory, each entry giving a name, a mode, and the identifier of a blob or another tree. A *commit* records the identifier of a tree (the snapshot), the identifiers of zero or more parent commits, an author and committer with timestamps, and a message. A *tag* object, in its annotated form, names another object and carries a message and optionally a signature. The identifier of an object is the hash of a header consisting of the object's type and length followed by its content.

Definition 16.1 (Object store)**.** Let $H : \{0,1\}^* \to \mathsf{Id}$ be a hash function. An *object* is a pair (τ, u) with type $\tau \in \{\mathrm{blob}, \mathrm{tree}, \mathrm{commit}, \mathrm{tag}\}$ and content $u \in \{0,1\}^*$. Its *identifier* is $\iota(\tau, u) = H(\mathrm{enc}(\tau, u))$, where enc is an injective encoding that prefixes the type and length. An *object store* is a finite partial map $\Sigma : \mathsf{Id} \rightharpoonup \mathsf{Obj}$ such that $\Sigma(\iota(\tau, u)) = (\tau, u)$ for every identifier in its domain, and such that every identifier referenced in the content of a stored tree, commit, or tag object lies in the domain (the store is *closed under reference*).

A set of named references, called refs, maps human-readable names such as branch names to commit identifiers. Branches in Git are nothing more than movable pointers of this kind. This is a notable design choice: the history is a graph of immutable objects, and the notion of a branch is an external annotation on the graph. The same graph can be named in any number of ways without altering it.

16.2 The Merkle property

The structure of an object store is a Merkle directed acyclic graph, named after the construction of hash trees [24] in which each node's identifier depends on the identifiers of its children. The property that makes it useful is that an identifier commits to everything reachable from it.

Theorem 16.2 (Identifier determines ancestry)**.** *Assume H is injective on the image of* enc (*collision-freeness*)*. Let c be a commit identifier in a closed object store Σ. Then c determines uniquely* (*i*) *the tree and parent identifiers recorded in the commit,* (*ii*) *by recursion, the complete set of ancestor commits, with their trees and the contents of every blob reachable from those trees, and* (*iii*) *the partial order of ancestry among these commits. Two closed stores containing the identifier c therefore agree on all of these.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Since H is injective on encodings and enc is injective, c determines the pair (τ, u) with $\iota(\tau, u) = c$, in particular the content u of the commit. The content lists the tree identifier and the parent identifiers. Apply the same argument to each. By induction on the length of the longest path from c (finite, since the graph is acyclic and the store is finite), every reachable object is determined. Blobs have no outgoing references, so the induction terminates. The ancestry order is the reachability relation along parent edges, determined by the parent lists. For two closed stores both containing c, the above determination yields the same reachable objects and the same relations. □

The theorem is the foundation for the lineage results of Chapter 25. Its hypothesis is not innocent. The hash function originally used by Git was SHA-1, for which a practical collision was demonstrated in 2017 [33], and the theorem holds only to the extent that collisions are not constructible by an adversary relevant to the use case. Git has added detection of known collision attack patterns and a transition to a stronger hash function. The correct reading is that the theorem is conditional on collision resistance and that such conditions are properties of the cryptographic world and not of the repository.

16.3 Operations and what they do to identifiers

The theorem allows a precise description of what common operations do to the identity of history.

Proposition 16.3 (Behavior of operations on identifiers)**.** *Let Σ be a closed object store satisfying the hypothesis of Theorem 16.2.* (*i*) Cloning *copies the objects reachable from the cloned refs and preserves every identifier;* (*ii*) Relocation *of a repository* (*copying the object store to another host*) *preserves every identifier;* (*iii*) Rewriting *a commit c so that it has a different tree, different parents, or different message produces a commit c' with $c' \neq c$, and every descendant of c that is reproduced on top of c' also receives a new identifier. Ancestors of c keep their identifiers;* (*iv*) Cherry-picking *the changes introduced by c onto a different parent produces a commit with a different identifier, even when the introduced changes are identical.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) and (ii) follow because identifiers depend only on content and the objects are copied bytewise. (iii) The identifier of c' differs from that of c since its encoding differs, and by injectivity of H. A descendant d of c records the identifier of its parent; if d is reproduced with parent c', its content differs, hence so does its identifier; by induction along the descent, every reproduced descendant has a new identifier. Ancestors are unchanged since they do not reference c. (iv) The commit records its parent identifier, which differs from the original's; therefore its content and identifier differ. □

The proposition reveals that Git's notion of history has two layers of identity. The graph of objects is rigid: an identifier names exactly one history. Yet the notions that practitioners care about, such as *the same change* or *the same branch*, live above this rigid layer and are not captured by identifiers. A rebased branch has entirely new commit identifiers, yet a human regards it as the same branch with the same changes. The tool provides some heuristics, such as a patch identifier computed from the textual change independently of its parent. These heuristics are approximate, and the gap between rigid identity and the human notion of sameness is where the questions of lineage in Part VI take their shape.

16.4 What the object graph does not contain

The object store is not the whole of what a project means by its repository. It contains no record of who may write to the repository, no issues or discussions, no review comments, no notion of a pull request, no record of what automated checks passed, no list of contributors beyond the names in commits, and no assertion about which branch is authoritative. A clone of a repository obtained from a hosting service contains the objects and the refs, but not the hosting service’s records of reviews, issues, and permissions. This is the technical basis of the distinction between portability of history and portability of the institution, which is developed formally in Chapter 29: the former is guaranteed by the architecture, and the latter is not.

Chapter 17

The Emergence of a Recognized Upstream

17.1 Convention built over symmetry

Technically, every clone of a Git repository is equal. A clone has the same objects, can create commits, can serve as a source for further clones, and can accept changes from any other. No clone is intrinsically authoritative. Yet projects in practice have a canonical repository, and the existence of a canonical repository is so familiar that it is easy to attribute it to the technology. It is not a property of the technology. It is a convention, supported by social recognition and by the integration of hosting services, and it can be created, challenged, and changed without any change to the underlying objects.

The convention operates through a small number of recognitions. Contributors send changes to the canonical repository's maintainers. Users install from it. Package registries are pointed at it. Documentation links to it. Search engines index it. Each of these is a decision by some agent to treat one repository as the continuation of the project; collectively they constitute the project's authority. The authority is distributed across the recognizing agents and has no location other than in their dispositions.

Definition 17.1 (Recognition and upstream)**.** For a project, let $\mathcal{C}_t$ be the set of repositories (equivalently, of refs) that continue the project's history at time t, and let A be the set of relevant agents. A *recognition* is a function $U_t : \mathcal{C}_t \to \mathcal{P}(\mathsf{A})$ assigning to each candidate continuation the set of agents that treat it as authoritative at time t. A candidate $r \in \mathcal{C}_t$ is a *recognized upstream* at t if $U_t(r)$ is sufficiently large relative to $U_t(r')$ for every other r', under a stated criterion of sufficiency.

The definition leaves the criterion of sufficiency open on purpose. Different communities use different criteria: a project governed by a single maintainer may treat the maintainer's repository as authoritative by fiat; a project governed by a foundation may treat the repository designated by the foundation as authoritative; a community may follow whichever fork the largest number of packagers adopt. The mathematical analysis of such criteria is the topic of Chapter 26.

17.2 Forks as challenges

Because every clone can continue independently, any dissatisfied participant can initiate a competing continuation. The possibility is the community's ultimate check on maintainers: a maintainer who acts against the interests of the contributors risks a fork that the contributors will follow. Historical instances of forks that displaced the original, and of forks that withered, are numerous and instructive. A fork succeeds when it acquires recognition, which depends on the fork's technical quality, on the reputation of those who maintain it, and on the cost to

Table 17.1: Functions provided by successive generations of infrastructure. A mark in a cell indicates that the generation provides the function as a native feature, though not that it provides it well.

Generation	Distribution	Source pres.	Depend. resol.	Collaboration	Institutional coord.
Media exchange	•				
Networked archives	•	•			
Package registries	•		•		
Centralized version control		•		•	
Distributed version control		•		•	
Hosting platforms	•	•		•	•

others of switching. A fork fails when it cannot attract the recognition necessary to sustain development. Neither outcome is dictated by the technology.

17.3 Hosting as consolidation

Hosting services integrate, around the repository, several functions that were previously separate: browsing and search of the code, an issue tracker, a mechanism for proposing and reviewing changes, automated build and test, release publication, and an identity system with profiles and notifications. The first widely used services for open-source project hosting, which began appearing in the late 1990s, provided a subset: space for repositories, mailing lists, and download areas. The services that appeared in the late 2000s distinguished themselves by emphasizing the social dimension of development. Developers had profiles; projects could be followed; contributions were attributed on a public page; and a changed copy of another's repository could be created with one action and proposed back with another. These were conveniences, and they changed the effective structure of open-source development by lowering the cost of creating forks and proposing changes to near zero.

The consolidation had effects beyond convenience. Because the services provided a single place where code, discussion, review, and release lived, they became the default location of record for software projects. The issue tracker and the review history, which exist only in the service's database and not in the Git objects, became part of the project's memory, and the service became part of the project's institution. A project that is moved elsewhere must either migrate these records or accept their loss, and the cost of migration is a component of the lock-in analyzed in Chapter 46.

17.4 A taxonomy of functions

Table 17.1 organizes the preceding chapters by the functions that each generation of infrastructure provided. Five functions are distinguished. *Artifact distribution* is the delivery of copies to those who want them. *Source preservation* is the retention of source in a form from which earlier states can be recovered. *Dependency resolution* is the computation of consistent sets

of components. *Collaborative development* is the coordination of concurrent contributions by several agents. *Institutional coordination* is the maintenance of identity, authority, governance, and public record. The table makes visible that the last function was historically a byproduct of the others. Hosting platforms did not set out to provide institutional coordination; they provided hosting, issue tracking, and review, and institutional coordination emerged as a property of the combination. This observation is the historical premise of the rest of the book. If institutional coordination emerges from certain combinations of capabilities, then it is a legitimate question which combinations suffice and which are merely customary, and the answer should be stated independently of any provider.

Remark 17.2 (On technological determinism). It would be a mistake to read the history above as the progressive solution of a sequence of problems by a sequence of inventions. Each system solved some problems and exposed others; several of the solutions coexisted for decades; and the dominance of any one was due in part to historical contingencies, including the licensing dispute, the timing of hosting services, and the existence of large communities that adopted particular tools. The taxonomy classifies functions; it does not rank technologies or assert inevitability.

Part V
The Formation of the Developer

Chapter 18

Account Creation and Persistent Identity

18.1 From principal to public identity

The accounts studied in Chapter 5 were local: a principal on a machine, known to the system and a few colleagues. A hosting service extends the same structure to a public setting. An account is created by choosing a handle in a global namespace, supplying contact information, and setting a credential. A profile may be added, with a display name, a picture, a short biography, a stated affiliation, and links elsewhere. The handle becomes part of every address in the service: repositories, contributions, discussions, and notifications all carry it. The account is thus a persistent address that accumulates a history, and the history is public by default.

Persistence is the operative property. The value of an account lies in what accrues to it, and what accrues depends on the account's staying recognizably the same. A person who changes handles loses the continuity of links, citations, and recognition associated with the old one, and so the choice of a handle is a choice with long consequences made at a moment when the person has little information about what the account will become. This is naming under uncertainty (Chapter 8) in an especially acute form, because the name cannot be tested against a future vocabulary and is difficult to change after the fact.

Definition 18.1 (Public identity)**.** The *public identity* of a developer on a service at time t is the tuple $(\ell_t, \iota, \eta_t, \rho_t)$ consisting of the current handle ℓ_t, the immutable internal identifier ι of the account, the public history η_t of actions performed under ι up to t, and the recognition ρ_t comprising the set of agents who attribute η_t to the developer. The handle and the recognition can change independently of the history; the history grows monotonically; and the internal identifier is fixed for the life of the account.

This is a specialization of Definition 5.2. Two features deserve emphasis. First, in services that expose only the handle publicly and use the internal identifier privately, external references to the account refer to the label and are therefore subject to the reuse hazard of Proposition 5.3. Second, the recognition component is outside the service's control: it resides in the minds of readers, in citations, in other services' links.

18.2 Authorship in the object graph is a claim

A subtle point concerns the relation between the account and the commits made by its holder. Git records an author name and email address in each commit, but these are values supplied by the committer's local configuration. The hash function binds them to the commit content, but nothing binds them to any real person or to any account.

Proposition 18.2 (Commit authorship is unauthenticated)**.** *For an unsigned commit in a Git repository, the author and committer fields are arbitrary strings chosen by the party who created the commit. Consequently, for any account and any commit content, a party with no relation to the account can create a commit that names the account's holder as author, and the resulting object is indistinguishable by its content alone from a commit made by the holder.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. The commit content is the concatenation of the tree identifier, parent identifiers, author line, committer line, and message. All of these are inputs to the commit-creation operation. The identifier is the hash of this content. The creation operation does not consult any authentication mechanism, so any strings may be supplied. Two commits with the same field values and the same other content are the same object by Definition 16.1. □

The services respond in two ways. They associate the email addresses in commits with accounts by a verification process, so that the interface displays a commit as made by an account only if the address has been verified by that account; and they support cryptographic signatures on commits and tags, so that a holder of a registered key can make a verifiable claim of authorship. Both mechanisms add authentication to the object graph from outside it, and the first of them lives in the service's database, not in the repository. A clone of the repository does not carry the service's association. The distinction reappears in Chapter 29.

18.3 Pseudonyms, handles, and recognition across time

Developers, like authors in other fields, often work under more than one name, and the same person may maintain several accounts for different purposes. A single researcher may use one account for publicly archived scientific work, another for personal tools, another for a collaboration. From the standpoint of the identity relation this is unproblematic: each account has its own history and recognition. It does raise a question for the theory of attribution: when is a body of work to be considered the work of one agent? A convention is that attribution follows the account, and the grouping of accounts into persons is left to the recognition component. Services that offer organizational accounts add a further layer, in which actions are performed by individuals acting for an organization; the identity of the organization is then a relation among its members' actions and the governance that authorizes them.

Remark 18.3 (Identity and responsibility). Identity matters institutionally because it is the carrier of responsibility. A maintainer who is responsible for a project's security fixes must be identifiable as the maintainer, and users who depend on the project must be able to find the maintainer. When the maintainer's account is unreachable, deleted, or compromised, the responsibility is orphaned. Public identity is therefore not mere branding. It is the surface through which obligations attach.

Chapter 19

Attention Before Contribution

19.1 Five relations that are often conflated

The first interactions most people have with a hosting service are not contributions. They are acts of attention: searching, browsing, reading code and issues, starring projects, following developers, watching repositories for activity. These acts leave traces that the service aggregates into counts, and the counts are then read as signals. The danger is a conflation of relations that are conceptually distinct.

Definition 19.1 (Relations between agents and projects)**.** For an agent a and a project r, five relations are distinguished. *Attention*: a has viewed or is notified about r. *Endorsement*: a has publicly signaled approval of r. *Dependency*: a has an artifact or process whose operation requires r. *Contribution*: a has proposed or made changes to r that were integrated into r or considered for integration. *Authority*: a is authorized to decide what is integrated into r. The relations are logically independent: no one of them implies any other.

Each implication that readers are tempted to draw is invalid. Attention need not imply endorsement; a project may be watched because it is wrong or a competitor. Endorsement need not imply dependency; a reader may admire a project and never use it. Dependency need not imply contribution; most users never contribute. Contribution need not imply authority, since contributions are routinely accepted from people with no further standing. And authority need not imply any of the others: a maintainer may hold authority over a project they no longer watch.

19.2 Counts as signals

A star is a one-click act. It may mean that the agent finds the project useful, wishes to bookmark it, intends to read it later, supports the author, or wishes to signal participation in a trend. The service aggregates all of these into one integer. The integer is a count of heterogeneous events and cannot be inverted to recover any single cause.

Proposition 19.2 (Aggregate counts are not invertible)**.** *Let each of n agents have one of $m \geq 2$ motives for starring a project, and let the count S be the number of agents who star. Then the map from the vector of motive-assignments to S is not injective, and for any value s with $0 < s < n$ there exist distinct motive assignments producing it. No function of S alone recovers the proportion of any given motive.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Fix s with $0 < s < n$ and two assignments in which exactly s agents star: in the first all starring agents have motive 1, in the second all have motive 2. Both give $S = s$, and the assignments differ since $m \geq 2$. A function of S takes the same value on both, so it cannot recover the proportion of motive 1. □

The result is deliberately elementary. It is stated because its practical consequence is often overlooked: ranking by counts is ranking by an aggregate whose composition is unknown.

19.3 Cumulative advantage

A deeper problem arises from feedback. Services display counts and sort by them, and agents are more likely to view, and therefore to star, what is already prominent. The process is a form of cumulative advantage, studied in the literature on preferential attachment and on cultural markets ([4, 31]). The simplest mathematical model is an urn process.

Proposition 19.3 (Cumulative advantage with equal quality)**.** *Consider two projects of identical quality. Starting from one star each, at each step a new agent stars one project, choosing project* 1 *with probability proportional to its current star count. After n steps the number k of stars added to project* 1 *is uniformly distributed on* $\{0, 1, \dots, n\}$*. In particular, the limiting share of attention is uniformly distributed on* $[0, 1]$ *and carries no information about quality, which is identical.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. A particular sequence of n choices containing k choices of project 1 has probability

$$\frac{(1 \cdot 2 \cdots k)\,(1 \cdot 2 \cdots (n-k))}{2 \cdot 3 \cdots (n+1)} = \frac{k!\,(n-k)!}{(n+1)!},$$

since at each choice of project 1 the numerator factor is its current count (which runs from 1 up to k over the choices) and similarly for project 2, while the denominator factors are the total count, which runs from 2 to $n+1$. This probability depends only on k. There are $\binom{n}{k}$ such sequences, so $\mathbb{P}[k] = \binom{n}{k}\frac{k!(n-k)!}{(n+1)!} = \frac{1}{n+1}$. The share k/n therefore converges in distribution to the uniform law on $[0, 1]$. □

Strict equality of quality is of course an idealization, but the point is robust to departures from it: when advantage is cumulative, the outcome is highly sensitive to early events, and the final distribution of attention can differ greatly from the distribution of merit. Experimental studies of artificial cultural markets found that the best and worst items tended to retain their extreme positions while intermediate items were widely unpredictable, and that the introduction of social influence increased inequality and unpredictability ([31]). The relevance for repository ecosystems is a caution against reading visibility as epistemic standing.

Principle 19.4 (Visibility is not standing)**.** Counts of attention, endorsement, and following are properties of the attention economy of a platform. They are not measures of correctness, quality, or authority, and an institution that relies on them for those purposes inherits the arbitrariness of cumulative advantage. [DESIGN ARGUMENT]

The principle motivates a later requirement: an infrastructure for scientific claims should record standing in terms of explicit support, objection, and scope (Part VIII) and not in terms of counts.

19.4 Attention as the beginning of a trajectory

None of this makes attention unimportant. It is the mechanism by which a newcomer acquires the knowledge needed to act. Observing how a project is organized teaches the conventions

of a community: where discussion happens, what a good issue looks like, how maintainers respond. A person who has watched many projects acquires a model of the institution. The transition from observer to participant depends on the acquisition of this model, and the acts of attention are its raw material. The key point for the theory of developerhood is that attention enlarges the set of operations a person *knows about* without changing what the person *can do*, a distinction made precise in Chapter 22.

Chapter 20

Forking and Independent Continuation

20.1 The discovery of the fork

Among the operations that a hosting service presents to a newcomer, forking has a special status. It is the discovery that another project's history can become the basis of an independent continuation, one that the newcomer controls, that proceeds from the same point, and that need not seek permission to diverge. The operation is technically trivial, since it copies a repository; its institutional significance lies in the fact that it creates a new locus of authority from a shared past.

Definition 20.1 (Future histories and forks)**.** For a commit c in an object store, let $\mathrm{Fut}(c)$ be the set of histories that extend c by finite sequences of admissible commits. A *fork* of a repository r at c is a repository r' that contains the ancestry of c and is controlled by an agent distinct from the controller of r. The fork has its own set $\mathrm{Fut}_{r'}(c)$ of futures determined by the governance of r'.

The key property is that forking adds possibilities without removing any.

Proposition 20.2 (Forking does not constrain the original)**.** *Let r' be a fork of r at c. Then the set of futures available to the controller of r is unchanged by the creation of r', and the set of futures reachable from c by some agent is the union of the sets available to the controllers of r and r'. In particular, the number of independent continuations of the history up to c is nondecreasing under forking.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Creating r' copies objects and does not modify r, since object stores are immutable and the controller of r retains all its refs and permissions. The set of continuations accessible to each controller is determined by its own repository and governance. The union is the sum of independently available sets, and each fork contributes a new member to the family of continuations. □

This is the formal sense in which forking is the ultimate check on a maintainer: it is an exit option that does not require the maintainer's cooperation. Hirschman's classification of responses to deteriorating organizations into voice, exit, and loyalty provides a vocabulary ([18]). Forking lowers the cost of exit to nearly zero in the technical sense. The social costs of exit remain, since a fork must attract users and contributors, and these costs are why forks are rare relative to their availability.

20.2 Technical copying and legal permission

The ease of copying a repository can conceal the question whether the copy may lawfully be redistributed or modified. Technically, any readable repository can be cloned. Legally, the

right to copy, modify, and distribute software is by default reserved to the copyright holder, and others may do so only to the extent that a license grants it. A repository without a license is, in many jurisdictions, one whose contents may be viewed on the hosting service under the service's terms but may not be redistributed or used to create derivative works without the author's permission. The terms of service of at least some hosting providers grant limited rights to view and fork public repositories within the service, but these rights do not extend to redistribution elsewhere.

Proposition 20.3 (Technical possibility and legal permission are independent)**.** *For an artifact x and an operation op, write* $\mathrm{Tech}(x, op)$ *for the proposition that the operation is technically possible and* $\mathrm{Legal}(x, op)$ *for the proposition that it is legally permitted. All four combinations occur.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. It suffices to exhibit instances. Both hold: copying a file released under a permissive license. Technical only: cloning an unlicensed public repository in order to redistribute it. Legal only: using a work whose license permits modification when the source has been lost, so that modification is permitted but not feasible. Neither: modifying a work whose source is unavailable and whose license forbids it. □

Licenses form a small number of families. Permissive licenses allow reuse with few conditions, typically retention of a notice. Copyleft licenses require that derivative works be distributed under the same or compatible terms. Dedications to the public domain, where available, and equivalent instruments, remove the author's rights as far as the jurisdiction allows. For the continuity of a project, the license is the artifact that determines which forks are lawful and under what conditions they may be merged back.

Definition 20.4 (Derivative-license constraint and compatibility)**.** For a license L, let $A(L)$ be the set of licenses under which a work derived from material under L may be distributed. Two licenses L_1, L_2 are *jointly compatible* if $A(L_1) \cap A(L_2) \neq \varnothing$. A combined work incorporating material under L_1 and L_2 may be lawfully distributed only if the licenses are jointly compatible.

Proposition 20.5 (Joint incompatibility occurs)**.** *There exist pairs of licenses with $A(L_1) \cap A(L_2) = \varnothing$. For such a pair, material under L_1 and material under L_2 cannot lawfully be combined into a single distributed work, even though both may be freely used separately.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let L_1 require that derivatives be distributed under L_1 only, and let L_2 require that derivatives be distributed under L_2 only, with $L_1 \neq L_2$. Then $A(L_1) = \{L_1\}$ and $A(L_2) = \{L_2\}$, which are disjoint. Real instances include pairs of copyleft licenses whose conditions are mutually exclusive; the abstract construction shows that nothing in the logic of licensing precludes it. □

The consequence for forks is that two independently developed continuations under incompatible licenses cannot be merged, even if the textual merge would be trivial. Licensing therefore constrains the topology of the space of possible reconciliations, an observation developed in Chapter 28.

Chapter 21

Contribution, Refusal, and Maintainer Authority

21.1 The pull request as a protocol

A proposed change to a project is, in the practice of hosted collaboration, a request that the project's maintainers pull a set of commits from another repository or branch. The mechanism has several stages. The contributor prepares the change in a fork or a branch. The contributor opens a request, with a title and a description. The maintainers and other participants discuss and review it. Automated checks may be run. The maintainers decide to integrate it, to request changes, or to close it without integration. The outcome is recorded in the service.

Each stage embodies a distinct kind of decision. Whether the change can be merged without textual conflict is a technical matter. Whether it conforms to the project's stated policies is a matter of rule-following. Whether the maintainers want it is a matter of discretion. Whether it is correct is an epistemic matter that the maintainers can judge only imperfectly. These four are formalized in Chapter 27, where it is shown that none entails any other.

21.2 Mechanism and decision

A frequent error in discussions of hosted platforms is to attribute decisions taken within the platform to the platform. When a maintainer declines a contribution, the platform has not declined it. The platform provides a mechanism by which a maintainer can record a decision; the decision is the maintainer's. The distinction is not a quibble. It determines to whom a person should address a complaint, whom a person should hold responsible, and which interventions are likely to change the outcome.

Definition 21.1 (Mechanism and discretion)**.** A platform specifies a set $\mathcal{M}$ of *mechanisms*, the operations it makes available, and a set of *platform rules* Γ_{plat} that constrain their use. A project owner specifies *project policies* Γ_{proj} within the space permitted by Γ_{plat} and makes *decisions* d by exercising mechanisms in accordance with these policies and with discretion. A decision is *attributable to the owner* if, holding the platform rules fixed, it varies with the owner's choices, and *attributable to the platform* if it is determined by Γ_{plat} alone.

Proposition 21.2 (Decisions are not determined by mechanisms)**.** *There exist two projects hosted on the same platform, with the same platform rules and the same submitted contribution, in which the contribution is accepted in one and declined in the other. Hence the outcome is not determined by the platform rules, and the acceptance or refusal is attributable to the project owners.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let both projects be governed by Γ_{plat}, which permits both acceptance and refusal of any request by the owner. Let the contribution δ be submitted to each. Owner o_1 with policy

Γ_1 accepts it and owner o_2 with policy Γ_2 declines it, both exercising permitted mechanisms. The outcomes differ while Γ_{plat} and δ are the same, so the outcome is not a function of these alone. □

The proposition is almost too simple to state, which is a measure of how frequently it is forgotten. It is the formal version of a distinction every legal and institutional system draws between an institution that makes tools available and the actors who use them. It has a corollary for design. A platform may be criticized for its mechanisms, such as default settings that make hostile behavior easy or an interface that discourages explanation of refusals. It may not be criticized for particular decisions of individual maintainers without further argument that the platform's mechanisms determined them.

21.3 Refusal as information

A refusal carries less information than it appears to. A closed request may reflect that the change is wrong, that it is out of scope for the project, that it conflicts with a policy, that the maintainers lack capacity to review it, that it duplicates other work, that the contributor has not followed the procedures, or that the maintainers dislike the contributor. These reasons have very different implications for the contributor and for the epistemic evaluation of the contribution.

Proposition 21.3 (Refusal underdetermines reason)**.** *Let $\rho_1, \dots, \rho_k$ with $k \geq 2$ be the reasons for which a contribution may be refused, let π_i be the prior probability that ρ_i is the operative reason (exactly one reason operates), and suppose at least two reasons have positive prior and each causes refusal. If refusal is recorded without a stated reason, the posterior probability of reason ρ_i given refusal is $\pi_i / \sum_j \pi_j$, where the sum ranges over refusal-causing reasons. The posterior is a point mass only if the prior is itself concentrated on a single refusal-causing reason.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By Bayes' rule, $\mathbb{P}(\rho_i \mid \text{refusal}) = \mathbb{P}(\text{refusal} \mid \rho_i)\pi_i / \mathbb{P}(\text{refusal})$. For refusal-causing reasons $\mathbb{P}(\text{refusal} \mid \rho_i) = 1$, and $\mathbb{P}(\text{refusal}) = \sum_j \pi_j$ over those reasons, which gives the stated formula. If at least two reasons have positive prior, each has posterior strictly between 0 and 1. □

The practical content is that a project that wishes its refusals to carry information must state reasons, and that observers should refrain from drawing conclusions about the quality of a contribution from the fact of refusal alone. This is one of the places in which the claim graph of Part VIII improves on a platform's record: it makes the reason for a decision a first-class item.

21.4 Governance mechanisms available to owners

Hosting services provide a range of governance mechanisms that owners may use. Branch protection can require that changes to designated branches be made only by merged requests, that a minimum number of approvals be obtained, that automated checks pass, and that the history on the branch not be rewritten. Code ownership files can designate who must approve changes to particular paths. Roles distinguish read, triage, write, and administrative

permissions. Organizations aggregate repositories and members under shared policies. Each mechanism adds to the repertoire of Γ_{proj} and each is a point at which an institution's procedure is encoded in software. The sum of these mechanisms is a considerable fraction of what a small institution requires: a way to say who may do what and a way to require evidence of review before a change is accepted. What is missing from the list, and what later parts address, is a way to represent why a decision was taken and what evidence supported it.

Chapter 22

From Developer to Institutional Organizer

22.1 Creating a first repository

A person who has attended to projects, forked some, contributed to some, and experienced refusals eventually does something categorically different: creates a project of their own. The act has a different structure from the previous ones. In forking, the history is inherited; in contributing, the governance belongs to someone else. In creating a repository, the person is the author of the history and of the governance. Each of the questions that other maintainers have answered must now be answered by the new maintainer: which license, what structure, what to document, what to accept.

The transition is gradual and passes through several stages: maintaining personal configuration files in a repository; publishing small tools; preserving a development environment so that it can be reconstructed; creating repositories for original work; writing contribution guidelines for those who might contribute; and establishing organizations that group repositories under a common identity. At each stage, new responsibilities are assumed and new operations become relevant.

22.2 A developmental model

The progression can be modeled without presupposing a ladder of competence. The model distinguishes three sets associated with a developer at a time.

Definition 22.1 (Developmental state)**.** The *developmental state* of a developer at time t is a triple $\mathcal{D}_t = (K_t, C_t, R_t)$ of subsets of a universe Op of operations and obligations, where K_t is the set of operations the developer *knows to exist*, C_t is the set of operations the developer *can actually perform* (that is, is authorized to perform and is able to carry out), and R_t is the set of *responsibilities* the developer has assumed or been assigned (maintenance of published artifacts, obligations to dependents, commitments to collaborators). A *competence function* $\chi_t : C_t \to [0, 1]$ assigns to each performable operation the quality with which the developer performs it.

The first observation is that the three components are independent.

Proposition 22.2 (Independence of knowledge, capability, and responsibility)**.** *For each of the sets K, C, R there exist developmental states in which an element belongs to that set and to neither of the others. In particular, $K \setminus C$ (aspiration or awareness without ability), $C \setminus K$ (latent capability), and $R \setminus (K \cup C)$ (responsibility without awareness or ability) can all be nonempty.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. For $K \setminus C$: a developer knows that branch protection rules exist but lacks administrative permission on any repository. For $C \setminus K$: a developer is granted write access to a repository and can perform an operation, such as publishing a release, of which the developer is unaware. For $R \setminus (K \cup C)$: a developer inherits maintenance of a repository on which users depend, without knowing how to respond to a security report and without the permission to publish a fix. Each situation is consistent with the definitions. □

The consequence is that no linear scale of developerhood captures the progression. Two developers can be incomparable under the componentwise inclusion order, one with greater awareness and the other with greater capability. Competence adds a further dimension: a large C_t with low χ_t describes a developer with wide permissions and poor results. A model that treated developerhood as a universal ladder would conflate these.

Proposition 22.3 (No total order)**.** *The set of developmental states ordered by componentwise inclusion is a partial order that is not total when* $|\mathsf{Op}| \geq 2$*.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Take $\mathsf{Op} \supseteq \{o_1, o_2\}$ and states $\mathcal{D} = (\{o_1\}, \varnothing, \varnothing)$ and $\mathcal{D}' = (\{o_2\}, \varnothing, \varnothing)$. Neither is contained in the other componentwise. □

22.3 Responsibility and irreversibility

Responsibility has a feature that distinguishes it from knowledge and capability: it can be acquired involuntarily and cannot always be shed. Publishing a repository that others depend on creates obligations that persist after the publisher stops attending to it. The dependents have no way to know that the maintainer's attention has lapsed, and the maintainer, by remaining the nominal authority, blocks others from taking over.

Proposition 22.4 (Responsibility is not discharged by deletion)**.** *Let a project r have a set* $\mathrm{Dep}(r)$ *of dependents with* $|\mathrm{Dep}(r)| > 0$*. Removing r from its host does not remove the dependency relations. It converts them into dangling references unless copies exist, and in either case the responsibility to dependents is not discharged but displaced.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. A dependency is a relation $u \to r$ held in the dependents' own artifacts and configurations; deleting r changes only the resolution of the reference (Proposition 7.2), not the relation. If no copies exist, resolution fails, and the dependents bear the cost; if copies exist, resolution proceeds, and the maintainer's decision to delete has changed who is responsible for the copies. □

The proposition ties the developmental model to the earlier incident in which a withdrawn package disrupted builds (Chapter 14). It also explains why institutions that host critical dependencies have developed practices for transferring maintainership and for archiving projects in a state that signals their status.

22.4 Governance levels

The assumption of institutional agency can be described in terms of the governance an organizer establishes. A project with no stated terms is at the lowest level. A project with a license establishes the terms of reuse. A project with contribution guidelines establishes the

Table 22.1: Governance levels of a repository and the questions each answers. The levels describe what has been specified, not how well.

Level	Specified	Question answered for a newcomer
0	Nothing beyond content	What is here?
1	License	May this be reused, and on what terms?
2	Contribution guidelines	How may changes be proposed?
3	Review and acceptance rules	What must happen before a change is accepted?
4	Roles and succession	Who decides, and who decides after them?
5	Dispute and revision procedures	What happens when decisions are contested?

terms of participation. A project with review requirements establishes the terms of acceptance. A project with a charter and designated roles establishes the terms of succession. The levels are cumulative in the sense that each presupposes the capacity to specify the previous ones, but not in the sense that every project needs every level.

Table 22.1 makes visible a pattern that Part VI formalizes. A repository acquires institutional character as it climbs this table: not by adding files, but by adding procedures that answer the questions a stranger would need to have answered in order to rely on it, contribute to it, or continue it.

Chapter 23

Continuity Debt, Orphaning, and Succession

The developmental state of Definition 22.1 describes one agent. A project is maintained by several agents over a span of years, and the questions that matter for continuation concern the project: how the work that future agents will need to do accumulates, who is both entitled and able to do it, and what happens when those who were are no longer present. The developmental model connects to the preservation framework of Chapter 4, the authorization relation of Chapter 6, and the governance results of Part VI. The results are elementary consequences of the definitions; their value is to state exactly which quantities move, and which do not, when a project grows or its maintainers leave. The empirical question of how often each mechanism operates in real projects is not addressed (Appendix H).

23.1 Continuity debt

Fix an artifact x, a successor agent a, and the finite universe I of information items of Definition 4.2, with an additive nonnegative cost c on items extended to sets by summation. Let $\mathcal{K}$ be a finite family of operation classes that future agents are expected to need, the *demanded classes*. A context $m \subseteq I$ is *sufficient for the demand* if $m \in \bigcap_{K \in \mathcal{K}} \mathrm{Suff}(x, a, K)$, and write $\mathrm{Suff}(\mathcal{K})$ for this intersection.

Definition 23.1 (Continuity debt)**.** For a preserved context m_0 and a demanded family $\mathcal{K}$, the *continuity debt* is

$$D(m_0, \mathcal{K}) = \min\{\, c(m \setminus m_0) : m \supseteq m_0,\ m \in \mathrm{Suff}(\mathcal{K}) \,\},$$

the least additional cost of preserved information needed to make the project continuable for every demanded class, with the value $+\infty$ if $\mathrm{Suff}(\mathcal{K})$ is empty.

The debt is the gap between what has been preserved and what the demand requires. It is a debt in the sense that it is incurred without any decision to incur it and is paid by acts of preservation (Definition 4.4, Proposition 4.5).

Proposition 23.2 (Properties of continuity debt)**.** *Assume sufficiency is monotone, as in Definition 4.2. Then: (i) $D(m_0, \mathcal{K}) = 0$ iff $m_0 \in \mathrm{Suff}(\mathcal{K})$. (ii) If $m_0 \subseteq m_1$ then $D(m_1, \mathcal{K}) \le D(m_0, \mathcal{K})$ (preserving more never increases the debt). (iii) If $\mathcal{K} \subseteq \mathcal{K}'$ then $D(m_0, \mathcal{K}) \le D(m_0, \mathcal{K}')$ (enlarging the demand never decreases the debt). (iv) For two families, $D(m_0, \mathcal{K} \cup \mathcal{K}') \le D(m_0, \mathcal{K}) + D(m_0, \mathcal{K}')$, with strict inequality possible when the two repairs share items.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) If m_0 is sufficient, take $m = m_0$ at cost zero; conversely, if the debt is zero and costs of items are positive, the minimizing m adds nothing, so $m = m_0$ (if some items have

cost zero the statement holds for the context extended by such items, which is sufficient by monotonicity). (ii) Let $m \supseteq m_0$ attain $D(m_0, \mathcal{K})$. Then $m \cup m_1$ is sufficient because $\mathrm{Suff}(\mathcal{K})$ is an up-set, contains m_1, and $c((m \cup m_1) \setminus m_1) = c(m \setminus m_1) \le c(m \setminus m_0)$ because $m_0 \subseteq m_1$. (iii) $\mathrm{Suff}(\mathcal{K}') \subseteq \mathrm{Suff}(\mathcal{K})$, so the minimum for $\mathcal{K}'$ is over a smaller set. (iv) Let m, m' attain the two minima. Then $m \cup m'$ is sufficient for both families and $c((m \cup m') \setminus m_0) \le c(m \setminus m_0) + c(m' \setminus m_0)$, with strict inequality when $(m \cap m') \setminus m_0$ has positive cost. □

Part (iii) is the formal content of the remark that debt accrues passively. A project whose users, dependents, and expected uses multiply enlarges $\mathcal{K}$ with no change in m_0, and its debt does not decrease and may increase. Part (iv) records the reason that shared documentation is economical: a single item that serves several classes is paid for once.

23.2 Qualified agents and orphaning

Let $\mathrm{Auth}_t(K)$ be the set of agents authorized at time t to perform the operations of K on the project (in the sense of Definition 6.2), and $\mathrm{Cap}_t(K)$ the set of agents a for whom $\mathrm{Cont}(a, x, m_t, t, K)$ holds, where m_t is the context preserved at t together with whatever the agents themselves retain.

Definition 23.3 (Qualified agents and orphaning)**.** The *qualified agents* for K at t are $Q_t(K) = \mathrm{Auth}_t(K) \cap \mathrm{Cap}_t(K)$. The class K is *orphaned* at t if $Q_t(K) = \varnothing$. The project is orphaned for a demanded family $\mathcal{K}$ if some $K \in \mathcal{K}$ is orphaned.

Proposition 23.4 (Routes to orphaning)**.** *A class K can become orphaned by each of the following mechanisms, and the mechanisms are independent: (i)* departure*: $Q_t(K) = \{a\}$ and a ceases to act; (ii)* mismatch*: $\mathrm{Auth}_t(K) \neq \varnothing$ and $\mathrm{Cap}_t(K) \neq \varnothing$ with no agent in both, so that nobody has left; (iii)* drift*: a class K newly enters the demanded family for which $m_t \notin \mathrm{Suff}(x, a, K)$ for every agent, so that $\mathrm{Cap}_t(K) = \varnothing$ although the same agents are present and authorized.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By construction. (i) Let $Q = \{a\}$ and remove a. (ii) Let $\mathrm{Auth} = \{a\}$ and $\mathrm{Cap} = \{b\}$ with $a \neq b$, the situation of a nominal owner who no longer understands the project and a competent contributor who holds no rights, as permitted by the independence of knowledge, capability, and responsibility (Proposition 22.2). (iii) Let all agents be authorized and let a class be demanded whose sufficiency sets contain no subset of m_t. □

The second mechanism shows why a count of active committers does not measure the risk of orphaning, and the third why a project can be orphaned by growth alone.

Proposition 23.5 (Redundant coverage)**.** *Let M be the set of maintainers and let R be a set of responsibilities, each $\rho \in R$ being a class whose qualified agents are the maintainers in $Q(\rho) \subseteq M$. No single departure orphans a responsibility if and only if $|Q(\rho)| \ge 2$ for every $\rho \in R$. The responsibilities orphaned by the departure of a are exactly those with $Q(\rho) = \{a\}$, and the fraction of R that they form is the key-person dependence of Definition 45.3 at the level of qualified agents.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Removing a leaves $Q(\rho) \setminus \{a\}$, which is empty iff $Q(\rho) \subseteq \{a\}$, and $Q(\rho) = \varnothing$ already would be an existing orphaning. Hence no departure orphans ρ iff $|Q(\rho)| \ge 2$. □

For example, suppose a project has maintainers a and b and the responsibilities of making releases, reviewing changes, and fixing security defects. If a alone holds the release keys and understands the release process, and b reviews and also understands the security process but lacks the keys, then $Q(\text{release}) = \{a\}$, $Q(\text{review}) = \{b\}$ (since only b is both authorized and able), and $Q(\text{security}) = \varnothing$ because b holds the knowledge but not the rights while a holds the rights but not the knowledge. The security responsibility is already orphaned although both maintainers are present, and the departure of either orphans a second responsibility. Counting maintainers (two) would not have shown any of this.

23.3 Succession

Proposition 23.6 (Succession criterion)**.** *Let a' be an agent at time t' for whom $\mathrm{Cont}(a', x, m, t', K)$ holds. Then a' can perform K as an authorized agent of the existing regime iff either $a' \in \mathrm{Auth}_{t'}(K)$ or some agent holds the meta-permission to grant a' that authority and does so. If $\mathrm{Auth}_{t'}(K) = \varnothing$ and no agent holds a meta-permission over K, then every continuation of K by a' is a fork, and whether it is the recognized continuation is not determined by the record of the existing regime.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. The first statement is the definition of authorization with meta-permissions (Definition 6.2): a right exists in the regime only if it is held or granted by an agent entitled to grant it. For the second, if no right and no meta-permission over K exists, a' cannot acquire one by any action within the regime, and an action taken outside it produces a new history whose standing relative to the original depends on recognition (Chapter 20, Theorem 26.2). □

The criterion separates two requirements that are often run together. Capability of the successor is a matter of the preserved context and is addressed by documentation and the other devices of Part III. Authority of the successor is a matter of the governance regime and is addressed by recorded succession, delegated meta-permissions, and threshold authority (Proposition 43.3). A project can satisfy the first and fail the second, in which case a competent successor can only fork, and it can satisfy the second and fail the first, in which case the authority passes to an agent who cannot use it.

23.4 Overload and the maintainability horizon

The preceding results concern which agents are qualified. A separate mechanism concerns whether the qualified agents have the effort to meet the obligations that arrive. Let $a_t \geq 0$ be the effort demanded by obligations arriving in period t (reviewing contributions, answering issues, repairing breakage caused by changes in dependencies), let $c_t \geq 0$ be the maintainers' effort available in period t, and let the backlog satisfy $B_{t+1} = \max(0, B_t + a_t - c_t)$ with $B_0 = 0$.

Proposition 23.7 (Backlog dynamics)**.** (*i*) *If $a_t \leq c_t$ for all t then $B_t = 0$ for all t.* (*ii*) *If $a_t - c_t \geq \varepsilon > 0$ for all $t \geq t_0$ then $B_t \geq \varepsilon(t - t_0)$ for $t \geq t_0$, so the backlog is unbounded.* (*iii*) *If maintainers depart at time τ so that $c_t = 0$ for $t \geq \tau$ and $a_t \geq \alpha > 0$, then $B_t \geq \alpha(t - \tau)$ for $t \geq \tau$.* (*iv*) *If $a_t = a_0 + w\,|\mathrm{Dep}_t|$ with $|\mathrm{Dep}_t| = \lfloor gt \rfloor$ for constants $w, g > 0$ and $c_t = c > a_0$ is constant, then the backlog is first positive at the least t with $a_0 + w\lfloor gt \rfloor > c$.* [PROVED UNDER STATED ASSUMPTIONS] (*FOR THIS DETERMINISTIC MODEL ONLY*)

Proof. (i) $B_t + a_t - c_t \leq B_t$ and induction. (ii) $B_{t+1} \geq B_t + a_t - c_t \geq B_t + \varepsilon$ for $t \geq t_0$. (iii) is (ii) with $\varepsilon = \alpha$. (iv) Until the first t with $a_t > c$ the backlog stays zero by (i), and at that t it becomes positive. □

The model is deliberately crude. Real effort is neither deterministic nor fungible across tasks, and obligations differ in urgency. It serves to make one point precise: because the arrival of obligations is driven by dependents (Proposition 22.4) and not by the maintainers' choices, a project that succeeds in attracting dependents moves toward the overload condition (iv) unless capacity grows with it. This is the quantitative counterpart of the passive accrual of continuity debt in Proposition 23.2(iii). Whether real projects follow such dynamics is an empirical question listed in Appendix H, and the model should be read as stating the mechanism whose strength is to be measured.

23.5 What the chapter contributes

The contribution is the identification of three quantities that the developmental model alone does not separate: the information that must be preserved (continuity debt), the agents who are both entitled and able to act (qualified agents), and the effort available against arriving obligations (backlog). A project is continuable when all three are adequate, and each fails by its own mechanism. The orphaning of a class need not involve any departure, and a project that is growing can be moving toward orphaning with no change in its personnel.

23.6 Result of Part V

The part has described, without presupposing a ladder, a model of the progressive acquisition of capability and responsibility by an individual within a repository ecosystem. Three results organize the discussion. Knowledge, capability, and responsibility are independent components of a developer's state (Proposition 22.2). Developerhood is a partial order and not a scale (Proposition 22.3). Responsibility, once assumed toward dependents, cannot be discharged by removal (Proposition 22.4). Applied to a project, the model yields a notion of continuity debt that accrues passively as demand grows (Proposition 23.2), three independent routes by which a class of operations becomes orphaned (Proposition 23.4), and a criterion that separates the capability of a successor from the authority of a successor (Proposition 23.6). The next part turns to the mathematical structure of the systems in which these individuals operate.

The preceding parts followed one agent. Part VI changes the unit of analysis from the developer to the institution formed by many developers, accounts, histories, and rules. The independence of knowledge, capability, and responsibility reappears there at the level of contributions, where mergeability, policy admissibility, acceptance, and correctness are also independent, and the individual's relation to a project becomes a question of who may authorize which transformation.

Part VI

The Mathematics of Repository Institutions

Chapter 24

Typed Repository Graphs

24.1 Why a typed graph

The preceding parts used an informal notion of a repository as a structure holding artifacts, histories, agents, and relations. The mathematical core of the book requires a precise object. The object is a typed directed graph, chosen because the relations that matter in a repository institution come in different kinds that must not be conflated. A parent edge between commits is not a review edge between a person and a commit, and a theory in which both are edges of one sort cannot say what an admissible transformation is. The typing disciplines are the places where definitions bear weight.

The design avoids dependence on a hosting provider by defining everything in terms of a small number of generic types, with the understanding that any provider implements some subset of them. The model is a proposal, not a characterization of any existing system. Its purpose is to distinguish the technical properties of a repository from the social and epistemic structures that develop around it.

Definition 24.1 (Typed repository graph)**.** Let T_V be a finite set of vertex types and T_E a finite set of edge types, with a signature $\mathrm{sig} : T_E \to \mathcal{P}(T_V) \times \mathcal{P}(T_V)$ assigning to each edge type a set of permitted source types and a set of permitted target types. A *typed repository graph* is a tuple $\mathcal{G} = (V, E, \theta_V, \theta_E)$ in which V is a finite set of vertices, $E \subseteq V \times V \times T_E$ is a set of typed edges, $\theta_V : V \to T_V$ assigns types to vertices, and θ_E is the projection of an edge to its type. The graph is *well typed* if for every edge $(u, v, \epsilon) \in E$ we have $\theta_V(u) \in \pi_1(\mathrm{sig}(\epsilon))$ and $\theta_V(v) \in \pi_2(\mathrm{sig}(\epsilon))$.

The typed vocabulary is a structural device. It does not claim that every hosting service implements the same institutional mechanisms, and the platform layer should be read as the generic category of records that services of this kind keep around the Git objects.

A concrete choice of types, adequate for the remainder of the book, is displayed in Table 24.1. The vertex types fall into four layers: the Git layer, which contains the objects of Definition 16.1; the reference layer, containing named movable pointers; the platform layer, containing the records that hosting services maintain around the objects; and the epistemic layer, containing claims, evidence, and attestations. Each layer's edges point into the layers below it, never the reverse, with the exception of attestations, which refer upward to claims and downward to objects.

24.2 The native layer and the hosted layer

Definition 24.2 (Native projection)**.** Let $T_{\mathrm{git}} = \{\mathrm{blob}, \mathrm{tree}, \mathrm{commit}, \mathrm{tag}\} \subseteq T_V$. The *native projection* $\pi_{\mathrm{git}}(\mathcal{G})$ is the subgraph induced on the vertices of types in T_{git}, together with the

Table 24.1: A generic vertex and edge vocabulary for repository institutions. Types in the first block are native to Git; the remainder are supplied by platforms or by epistemic infrastructure.

Layer	Vertex types	Representative edge types
Git objects	blob, tree, commit, tag	entry (tree to blob or tree), snapshot (commit to tree), parent (commit to commit), target (tag to object)
References	branch, release	points-to (branch to commit), publishes (release to tag or commit)
Platform	agent, issue, review, policy	authored-by (claimed), reviews (review to commit), references (issue to commit or issue), governs (policy to branch)
Epistemic	claim, evidence, attestation	supports, objects-to, scopes, attests (attestation to any vertex)

edges among them. A typed repository graph is a *hosted extension* of a native graph $\mathcal{N}$ if $\pi_{\text{git}}(\mathcal{G}) = \mathcal{N}$.

The projection forgets everything except the object graph. It is surjective onto native graphs, since every native graph is itself a typed repository graph, and it is far from injective.

Proposition 24.3 (The projection loses information)**.** *For every native graph $\mathcal{N}$ with at least one commit, there exist infinitely many pairwise non-isomorphic hosted extensions of $\mathcal{N}$. In particular, π_{git} is not injective and no function of the native graph determines the hosted graph.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let c be a commit in $\mathcal{N}$. For each $n \geq 0$ define $\mathcal{G}_n$ by adding to $\mathcal{N}$ exactly n issue vertices each with a references edge to c. The graphs $\mathcal{G}_n$ are well typed, project to $\mathcal{N}$, and are pairwise non-isomorphic because they have different numbers of issue vertices. □

The proposition is trivial and will be used often. Its content is the observation that the object graph is a strict substructure of the institution, and that questions about review, permission, or discussion are questions about the other layers.

24.3 Admissible transformations

A repository evolves. The theory needs a notion of admissible change, a class of graph transformations that respect the invariants the system is supposed to maintain. For the native layer the invariants are immutability of existing objects, closure under reference, and acyclicity of the parent relation.

Definition 24.4 (Native extension step)**.** An *extension step* on a native graph $\mathcal{N} = (V, E, \theta_V, \theta_E)$ adds one vertex v of a native type together with outgoing edges of permitted types to vertices already in V (so that closure under reference is preserved), and adds no edge with source in V. An *admissible native transformation* is a finite composition of extension steps.

Proposition 24.5 (Preservation by extension steps)**.** *Admissible native transformations (i) preserve well-typedness, (ii) preserve every vertex and edge of the original graph (so existing objects are immutable), (iii) preserve closure under reference, and (iv) preserve acyclicity of the parent relation.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) The new vertex and its outgoing edges are chosen with permitted types. (ii) Nothing is removed and no edge with source in the old vertex set is added. (iii) New outgoing edges lead to existing vertices, and no old vertex gains an edge. (iv) A cycle in the parent relation of the extended graph would have to pass through the new vertex v, since the old graph is acyclic. But v has no incoming edges in the extended graph (no edge is added with target v, since old vertices gain no edges), so v cannot lie on a cycle. □

The proposition says in a few lines why object graphs are append-only in the abstract model. In the concrete system the same property is secured by the hash. To construct a cycle among objects, one must find identifiers $x_1, \ldots, x_n$ such that each x_i equals the hash of an encoding that includes x_{i+1}, indices modulo n, which requires a fixed point of a composition of hash evaluations.

Proposition 24.6 (Cycles are infeasible in the random-oracle idealization)**.** *Model H as a random oracle with b-bit outputs. An adaptive adversary makes at most q oracle queries, each on an input string that cites at most c identifiers of b bits (the identifiers are the substrings of the input that the object format interprets as references). The probability that the objects whose identifiers were obtained from the queries contain a reference cycle is at most $q^2c/2^b$.* [PROVED UNDER STATED ASSUMPTIONS] (*IN THE RANDOM-ORACLE MODEL ONLY*)

Proof. Suppose the created objects contain a cycle $o_1 \to o_2 \to \cdots \to o_n \to o_1$, where $o_i \to o_{i+1}$ means that the content of o_i cites the identifier x_{i+1} of o_{i+1} (indices modulo n, with $n = 1$ allowed). Consider the object o_j on the cycle whose identifier was obtained by the earliest query, say query t. Its input cites x_{j+1}, and x_{j+1} is the output of some query $t' \geq t$ (with $t' = t$ only if $n = 1$). The input of query t is fixed before the output of query t' is drawn, and the output of the first query on a given input is uniform on $\{0,1\}^b$ and independent of all earlier queries and outputs, whatever the adversary's strategy. For a fixed pair (t, t') and a fixed cited position, the probability that the output equals the cited value is 2^{-b}. There are at most q choices of t, at most c cited positions, and at most q choices of t', so a union bound over all of them gives at most $q^2c/2^b$. The adaptivity of the adversary is handled as follows. Fix a pair (t, t') with $t \leq t'$ and a cited position k in the input w_t of query t. The input w_t is a function of the transcript of queries $1, \ldots, t-1$ and of the adversary's coins, so it is determined before query t' is answered, and in particular it cannot depend on the output $y_{t'}$ when $t' > t$. Conditioned on the transcript up to query $t' - 1$, the output $y_{t'}$ of a first-time query is uniform on $\{0,1\}^b$, and the cited value in position k of w_t is a fixed string, so $\Pr[y_{t'} = \text{cited value}] = 2^{-b}$. For $t' = t$ the cited value is part of the input w_t and the output y_t is drawn after the input is fixed, so the same bound holds. A repeated query returns an earlier output and cannot create a coincidence that the first occurrence did not. Summing the conditional probabilities over the at most q^2c triples, and using that a cycle requires at least one coincidence of this form, gives the bound. □

The bound is of the usual collision-style form and has been rederived here under an explicit query model; an independent check by a reader familiar with random-oracle proofs is still advisable. The caveat remains that the real hash function is not a random oracle. The qualification matters: the status of the proposition is limited to the idealization.

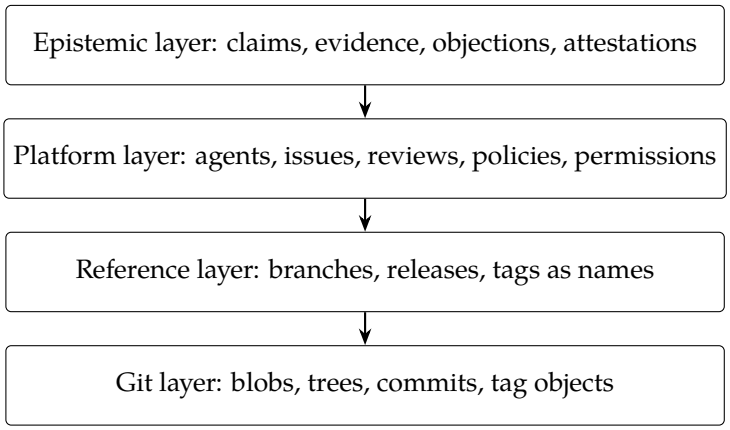


Figure 24.1: Layers of a repository institution. An arrow from one layer to another indicates that vertices of the upper layer refer to vertices of the lower layer and not conversely.

24.4 Layering and the direction of dependence

Figure 24.1 depicts the layers. The direction of the arrows reflects an architectural principle that will be used throughout: higher layers depend on lower layers, never the reverse. A review refers to a commit; a commit does not refer to a review. A claim refers to evidence, which may be a commit or a release; the commit does not refer to the claim. The principle is the structural reason why a Git history can be migrated without the platform records, while the platform records cannot be migrated without the histories to which they refer.

Principle 24.7 (Downward reference)**.** In a well-designed repository institution, edges between layers point from the more institutional layer to the less institutional layer. Consequences: the lower layer can be preserved, copied, and verified without the upper; the upper cannot be interpreted without the lower; and migrations should be planned from the bottom up. [DESIGN ARGUMENT]

Chapter 25

History, Lineage, and Continuation

25.1 Histories as partial orders

Within the commit layer, the parent edges define a partial order: a commit c' is an ancestor of c if there is a directed path from c to c' along parent edges. The relation is reflexive and transitive by construction and antisymmetric because the graph is acyclic (Proposition 24.5).

Definition 25.1 (History)**.** A *history* is a finite poset $(C, \preceq)$ in which C is a set of commits and $c' \preceq c$ means that c' is an ancestor of c (including $c' = c$). Write $c' \prec c$ for the strict relation. The *ancestor set* of c is $\mathrm{Anc}(c) = \{c' : c' \preceq c\}$. Two commits are *comparable* if one is an ancestor of the other and *divergent* otherwise. A *merge base* of divergent commits x, y is a maximal element of $\mathrm{Anc}(x) \cap \mathrm{Anc}(y)$.

Merge bases are the data used by three-way merges (Chapter 28). A pleasant property of linear histories is that the merge base is unique. In general it is not.

Proposition 25.2 (Merge bases need not be unique)**.** *For divergent commits x, y the set of merge bases is an antichain, and it is nonempty if and only if x and y have a common ancestor (in particular, when the history has a single root). When nonempty it can contain more than one element.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. The set $\mathrm{Anc}(x) \cap \mathrm{Anc}(y)$ is finite, and it is nonempty exactly when x and y have a common ancestor; the merge bases are its maximal elements, which exist whenever the set is nonempty and always form an antichain. If the histories are disconnected the set is empty and there is no merge base, which occurs for repositories that have been combined from unrelated roots before being merged. For non-uniqueness, take a root r and commits a, b each with parent r; let x have parents a, b in that order and y have parents b, a (two merges of the same pair of branches, created independently). Then $\mathrm{Anc}(x) \cap \mathrm{Anc}(y) = \{r, a, b\}$ whose maximal elements are a and b. □

The configuration of Figure 25.1 is the source of the long-standing difficulty with merging in the presence of repeated cross-merges. Tools resolve it by recursive strategies that first merge the multiple bases into a virtual common ancestor. The existence of multiple merge bases is a reminder that a history is a partial order and not a line, and that "the common ancestor" is a convention that fails on the general case.

25.2 Which properties of lineage survive which operations

The practical question is what is preserved when a history is copied, rewritten, or moved. The answers fall into three groups, corresponding to three kinds of invariant.

The first kind is *identifier-level*: statements about the identity of commits and objects. The second is *order-level*: statements about the ancestry relation within the history. The third is

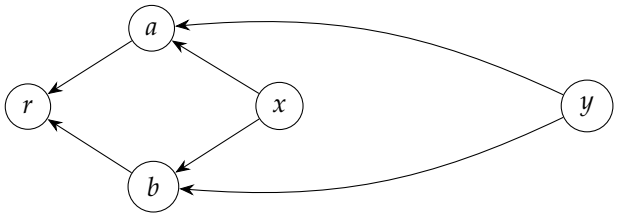


Figure 25.1: A criss-cross configuration. The commits x and y each merge a and b, so their merge bases are a and b, two maximal common ancestors.

change-level: statements about the textual changes that commits introduce, which are what humans often mean by the same change.

Definition 25.3 (Patch equivalence)**.** For a commit c with a single parent p, let $\Delta(c)$ denote the change from the tree of p to the tree of c, represented as the set of path-level modifications with the textual hunks at each path. Two single-parent commits c, c' are *patch-equivalent*, written $c \sim c'$, if $\Delta(c) = \Delta(c')$.

Patch equivalence is an equivalence relation on single-parent commits, since it is defined by equality of a function's values. It is coarser than identifier equality and incomparable with tree equality: two patch-equivalent commits may have different trees (because their parents differ) and two commits with the same tree may have different changes.

The two hypotheses on the base in part (ii) are independent. The condition $b \neq \text{parent}(c_1)$ excludes the trivial rebase of the segment onto its own parent, which changes nothing, and the condition $c_1 \not\preceq b$ excludes a base that already contains c_1, where the operation reduces to a fast-forward and the invariants differ. The theorem concerns an abstract replay operation and not every behavior of a real tool. A *hunk-preserving replay* of a commit c onto a commit p produces a commit with parent p whose change Δ equals that of c (the same paths with the same textual hunks), and it is *conflict-free* when it is defined, that is, when each hunk applies to the tree of p without modification. Real rebase and cherry-pick implementations approximate this operation: they may use surrounding context to locate hunks, apply a rename-detection heuristic, or fall back to a three-way merge that alters hunks. Where they do, the equivalence $\sim$ of the theorem holds only to the extent that the hunks are in fact unchanged, and results that depend on it should be read as applying to the abstract operation.

Theorem 25.4 (Invariants of lineage under standard operations)**.** *Let $(C, \preceq)$ be a history in a closed object store satisfying the hypothesis of Theorem 16.2. (i) Under cloning and relocation, identifier-level, order-level, and change-level properties are all preserved: the map on commits is the identity; (ii) Let $c_1 \prec c_2 \prec \cdots \prec c_k$ be a linear segment of single-parent commits and let a conflict-free rebase onto a base b with $b \neq \text{parent}(c_1)$ and $c_1 \not\preceq b$ produce $c'_1 \prec \cdots \prec c'_k$. Then $c'_i \prec c'_j$ iff $i < j$ (order within the segment is preserved), $c_i \sim c'_i$ for all i (change-level equivalence), and $c_i \neq c'_i$ for all i (identifier-level identity is not preserved). Moreover $c_i \not\preceq c'_j$ for all i, j, so the old commits are not ancestors of the new ones; (iii) Cherry-picking a single-parent commit c onto a commit $p' \neq \text{parent}(c)$ without conflict produces c' with $c' \sim c$, $c' \neq c$, and with no ancestry relation between c and c' unless $c \preceq p'$.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) Cloning and relocation copy objects bytewise (Proposition 16.3), so the identity map on commits preserves identifiers, parent relations, and the changes recorded.

(ii) A rebase constructs c'_1 with parent b and tree obtained by applying $\Delta(c_1)$ to the tree of b, and c'_{i+1} with parent c'_i and tree obtained by applying $\Delta(c_{i+1})$ to the tree of c'_i. Conflict-freeness means each application succeeds without altering the hunks, so $\Delta(c'_i) = \Delta(c_i)$. By construction the parent chain is $b \prec c'_1 \prec \cdots \prec c'_k$, so order within the segment is preserved. The commit c'_1 has parent $b \neq \text{parent}(c_1)$, so its encoded content differs from that of c_1 and, by injectivity of H, $c'_1 \neq c_1$. Inductively, c'_{i+1} records the parent $c'_i \neq c_i$ and therefore differs from c_{i+1}. Finally, the ancestors of c'_j are $\{c'_1, \ldots, c'_j\} \cup \text{Anc}(b)$. If some c_i were in $\text{Anc}(b)$ then $c_1 \preceq c_i \preceq b$, contradicting $c_1 \not\preceq b$; and no c_i equals any c'_l.

(iii) is proved as in (ii) with $k = 1$; the new commit's parent is p', so its ancestors are $\{c'\} \cup \text{Anc}(p')$, which contains c only if $c \preceq p'$. □

The theorem makes the earlier remark precise. Rebase and cherry-pick preserve change-level structure but destroy identifier-level structure, and consequently any external record keyed to identifiers is invalidated. This includes pointers from other branches, references in issue discussions, citations in publications, and anything else that recorded the old identifiers. The cost is not borne by the Git objects themselves, which remain valid, but by the relational context in which they were embedded, in line with Principle 7.5.

25.3 Squashing and the compression of history

A third operation, squashing, replaces a sequence of commits by a single commit whose change is the composition of the sequence. It is widely used to keep history tidy, and it is worth recording what it does to the three kinds of invariants. Order-level information within the squashed segment is destroyed, since the segment becomes a point. Change-level information is preserved in aggregate, since the composite change equals the composition of the segment's changes, but the decomposition into steps is lost. Identifier-level information is destroyed in the same way as for rebase. A history that has been aggressively squashed thus records *what* changed between states and loses a good deal of *how* the change was developed, including false starts and the reasoning visible in intermediate commit messages. The tradeoff is real. Squashing lowers the noise of Chapter 12 at the price of the documentary channel of Chapter 10.

Proposition 25.5 (Squashing is lossy)**.** *Let a segment $c_1 \prec \cdots \prec c_k$ with $k \geq 2$ be replaced by a single commit s whose change is $\Delta(c_k) \circ \cdots \circ \Delta(c_1)$. Then there exist distinct segments with the same composite change, and hence no function of s alone recovers the original segment.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Take two segments of length two over a single file: one that first inserts a line and then edits it, and one that inserts the final edited line directly together with a no-op first commit (or inserts the line in a different order of operations leading to the same text). Both compose to the same overall change. A function from s to segments would have to return both. □

25.4 Continuation of a history

The word continuation has been used informally. A precise notion is needed for the question of what it means for one repository to continue another.

Definition 25.6 (Continuation of a history)**.** A history $(C', \preceq')$ is a *strict continuation* of $(C, \preceq)$ if $C \subseteq C'$ and $\preceq'$ restricted to C equals $\preceq$ and no element of $C' \setminus C$ is an ancestor of an element of C. It is a *change-level continuation* if there is a map $\phi : C \to C'$ with $c \sim \phi(c)$ for all c and ϕ order-preserving.

The strict notion is what a fork or a clone followed by new commits satisfies. The change-level notion is satisfied also by rebased histories. A good deal of confusion about *whether a rewritten repository is still the same project* dissolves once the two notions are distinguished: a rewritten history is a change-level continuation and not a strict one, and external references that depend on strictness break.

Chapter 26

Authority Without Centralized Storage

Studies of open-source communities describe authority as something that emerges and is legitimated through procedures and not as a consequence of technical ability to modify shared code [42]; the chapter formalizes only a structural part of that picture.

26.1 Technical existence versus recognition

Every commit in a distributed repository exists independently of any claim about its standing. A branch in someone's clone is technically a legitimate continuation of the history it extends. Nothing in the object graph says that one continuation is the project and another is a variant. The statement that one is authoritative is a statement about the world, about the dispositions of agents, and cannot be read from the graph.

Recall the recognition function U_t of Definition 17.1, which associates to each candidate the set of agents that treat it as authoritative at time t. The questions to be settled are what such a function can look like, whether the structure of the history constrains it, and how a socially stable upstream can emerge from individual dispositions.

26.2 History alone does not select an authority

Consider the strongest conceivable claim that history determines authority: that there is a rule F which, given the ancestry structure of a set of candidate continuations, selects the authoritative one. Such rules are not fanciful. The longest-chain rule used in some distributed ledger systems is one; selecting the candidate with the greatest number of commits is another; selecting the candidate with the most recent common ancestry to the largest number of contributors is a third.

Definition 26.1 (Structural authority rule)**.** Let a *candidate configuration* be a pair (G, L) where G is a finite directed acyclic graph (the ancestry graph) and $L \subseteq V(G)$ is a nonempty set of candidate tips. A *structural authority rule* is a function F assigning to each candidate configuration a nonempty subset $F(G, L) \subseteq L$, such that F is invariant under isomorphism: if $\sigma : G \to G'$ is a graph isomorphism with $\sigma(L) = L'$ then $F(G', L') = \sigma(F(G, L))$.

The invariance condition says that the rule uses only the shape of the ancestry graph and is blind to names, identifiers, contents, authors, and timestamps. It is a natural formalization of the idea that authority could be read from the history itself.

Theorem 26.2 (Structural rules cannot break symmetry)**.** *Let (G, L) be a candidate configuration and let σ be an automorphism of G with $\sigma(L) = L$. Then for every structural authority rule F, the set $F(G, L)$ is invariant under σ, so it is a union of σ-orbits. In particular, if $\sigma(a) = b \neq a$ for tips $a, b \in L$, then $a \in F(G, L)$ iff $b \in F(G, L)$. Hence if every tip of L lies in a σ-orbit of size at least*

two, $F(G, L)$ contains at least two tips and F cannot select a unique one. [PROVED UNDER STATED ASSUMPTIONS]

Proof. Apply the invariance condition with $G' = G$, $L' = L$, and the automorphism σ: $F(G, L) = \sigma(F(G, L))$. Hence $F(G, L)$ is a union of σ-orbits. If a is in it then so is $\sigma(a) = b$. If every tip belongs to an orbit of size at least two, any nonempty union of orbits has at least two elements. □

Countermodel 26.3 (A symmetric fork)**.** Let G consist of a root r and two children a and b, both with parent r, and let $L = \{a, b\}$. The map swapping a and b is an automorphism preserving L. By Theorem 26.2, no structural authority rule selects exactly one of the two tips. The configuration is realized by any project in which two maintainers each add one commit to a shared base.

The theorem shows a limitation that is robust, since it holds for every structural rule at once, but also a narrow one: the symmetric configurations are special. Real forks usually differ in contents, authors, timing, or volume of activity, and rules using such information can break the symmetry. This is the point. The information that breaks the symmetry is not intrinsic to the ancestry structure and has to be supplied from outside it, and the choice of what to supply is a choice of governance. A rule that prefers the more active fork encodes one theory of authority, a rule that prefers the older encodes another, a rule that prefers the one endorsed by a designated body encodes a third. None is forced by the history.

Corollary 26.4 (Authority nonuniqueness)**.** *Agreement among all parties on the history, and on all contents of the objects, does not determine a unique authoritative continuation. For every configuration with two incomparable tips there exist recognition functions U, U' consistent with the same history in which different tips are recognized as dominant.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Recognition is a function on candidates to sets of agents and is not constrained by the graph. Take any recognition under which tip a receives the recognition of all agents and b none, and another in which the roles are reversed. Both are consistent with the history. □

26.3 Recognition without unanimity

If history does not select an authority, social recognition must, but recognition need not be exclusive or unanimous. Agents may recognize several continuations as legitimate, with different roles: one as the development upstream, another as the stable branch, a third as a long-term maintenance fork. The mathematics of dominance depends on how a threshold is set.

Proposition 26.5 (Dominance under overlapping recognition)**.** *Let a candidate r be θ-*dominant *at time t if $|U_t(r)| \geq \theta|\mathsf{A}|$. (i) If recognition is exclusive (each agent recognizes at most one candidate) and $\theta > 1/2$, at most one candidate is θ-dominant. (ii) If recognition may overlap, then for every $\theta < 1$ and every number m of candidates there are recognition functions under which all m candidates are θ-dominant simultaneously.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) Under exclusivity the sets $U_t(r)$ are pairwise disjoint. Two θ-dominant candidates would contribute at least $2\theta|\mathsf{A}| > |\mathsf{A}|$ agents, impossible. (ii) Let every agent recognize every candidate, so $|U_t(r)| = |\mathsf{A}| \geq \theta|\mathsf{A}|$ for each. □

The case of overlapping recognition is common in practice: a downstream project depends on an upstream and also recognizes a patched fork; a packager ships one version and tracks another. The existence of simultaneous dominance means that a single "official" repository is a sufficient but not a necessary state of affairs, and a theory that required uniqueness would misdescribe much of real practice.

26.4 How a recognized upstream emerges

A stable convention of recognition is an equilibrium of a coordination problem. Each agent benefits from coordinating with others on the same upstream, because shared infrastructure, shared review, and a shared record reduce costs, and each agent also has preferences about the quality or governance of the candidates. The following model captures the interplay and shows that coordination benefits can dominate quality differences.

Definition 26.6 (Coordination model)**.** There are n agents and two candidates $1, 2$. If a fraction x_k of agents choose candidate k, an agent choosing k receives payoff $\alpha x_k + q_k$, where $\alpha > 0$ is the value of coordination and q_k is the intrinsic quality of candidate k as seen by the agents. A *Nash equilibrium* is a profile in which no agent can raise its payoff by changing its choice unilaterally.

Proposition 26.7 (Lock-in)**.** *Let n be large enough that one agent's deviation changes shares by $1/n$, and let αn^{-1} be negligible, so that a deviator to an unchosen candidate sees share approximately 0. The profile in which all agents choose candidate 1 is a Nash equilibrium iff $\alpha + q_1 \geq q_2$ up to terms of order $1/n$; all choose candidate 2 iff $\alpha + q_2 \geq q_1$. Both are equilibria iff $\alpha \geq |q_1 - q_2|$. In that case an inferior candidate can be the unique recognized upstream in equilibrium.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. In the all-1 profile an agent receives $\alpha + q_1$; by deviating to 2 it receives $\alpha \cdot \frac{1}{n} + q_2 \approx q_2$. The profile is an equilibrium iff $\alpha + q_1 \geq q_2$ (up to $O(1/n)$). Symmetrically for all-2. Both inequalities hold iff $\alpha \geq q_2 - q_1$ and $\alpha \geq q_1 - q_2$, that is, $\alpha \geq |q_1 - q_2|$. When this holds and $q_1 < q_2$, the all-1 profile is an equilibrium although candidate 2 is better. □

The result is a standard feature of coordination games and is invoked here for its implications. The upstream that emerges is selected by history of play, by early adoption, by focal points, and by switching costs, as much as by quality. This is the formal counterpart of the remark that the dominance of any repository convention is partly a contingent matter, and it prepares the discussion of network effects in Chapter 46.

Chapter 27

Contribution as an Admissibility Problem

27.1 Four predicates

A proposed change to a history can be assessed on several grounds. The grounds are conflated in ordinary speech, because the same word, *accepted*, is used for results of different processes. These grounds are logically independent.

Definition 27.1 (Proposals and the four predicates)**.** Let h be a history with tip t and let δ be a proposed transformation, represented as a change to the tree at t. Let G be a governance regime and σ a specification against which correctness is judged. Define the following predicates. (a) $\mathsf{M}(h, \delta)$, *technical mergeability*: δ can be applied to the tree at t without textual conflict by the merge procedure in force; (b) $\mathsf{P}_G(h, \delta)$, *policy admissibility*: δ satisfies the mechanical rules of G (required checks pass, required trailers or signatures are present, required approvals are recorded); (c) $\mathsf{A}_G(h, \delta)$, *maintainer acceptance*: an agent authorized by G decides to integrate δ; (d) $\mathsf{C}_\sigma(h, \delta)$, *epistemic correctness*: the history extended by δ satisfies σ. The *integration decision* is $\Gamma_G(h, \delta) \in \{0, 1\}$, with $\Gamma_G(h, \delta) = 1$ meaning that δ is integrated.

The integration decision is a function of the governance regime and in general depends on the first three predicates; whether it also depends on the fourth depends on whether the regime has a means of assessing correctness. A regime in which an authorized agent may override policy gives $\Gamma_G = \mathsf{A}_G$ alone. A regime in which acceptance requires both policy satisfaction and the agent's decision gives $\Gamma_G = \mathsf{P}_G \wedge \mathsf{A}_G$. In neither is C_σ a determinant, because correctness is not directly available to the regime.

Theorem 27.2 (Independence of the four predicates)**.** *For every assignment of truth values $(m, p, a, c) \in \{0, 1\}^4$ there exist a history h, a proposal δ, a governance regime G with an override right for authorized agents, and a specification σ such that $\mathsf{M} = m$, $\mathsf{P}_G = p$, $\mathsf{A}_G = a$, and $\mathsf{C}_\sigma = c$.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Take a history whose tip tree contains a function `add(x,y)` in a file `f.py` and a separate file `g.py`. Let σ state that `add` returns the sum. Construct δ from four independent choices.

Mergeability m: if $m = 0$, let a concurrent commit on the base branch modify the same line of `g.py` that δ modifies, so that applying δ produces a textual conflict; if $m = 1$, let there be no such concurrent commit.

Policy p: let G require a sign-off trailer in the proposal's commit message as a mechanical check. If $p = 1$, include the trailer; if $p = 0$, omit it.

Acceptance a: let the authorized agent's decision be an input of the construction, set to a. Since G has an override right, acceptance can be set independently of p.

Correctness c: let δ also modify `f.py`. If $c = 1$ the modification leaves `add` correct; if $c = 0$ it introduces an off-by-one error so that `add` returns a value one too large.

The four choices affect disjoint aspects of the construction (which line of which file is edited, the text of the message, the decision of the agent, the body of the function), so all sixteen combinations arise. □

The theorem explains many ordinary experiences. A contribution that merges cleanly may still be unacceptable. One that satisfies every mechanical rule may be rejected. One that is accepted may be wrong. One that is correct may be refused. No one of these facts implies another. An institution that wishes to act on any of the predicates must therefore provide an independent mechanism for establishing it.

Example 27.3 (Clean merge, broken program)**.** On a base commit, a function `f` is defined in one file and called in several others. One branch renames `f` to `g` everywhere. A concurrent branch adds a new file with a call to `f`. The two sets of changes touch disjoint paths, so the merge is textually clean ($\mathsf{M} = 1$), but the merged program contains a call to a function that no longer exists ($\mathsf{C} = 0$). The error is a semantic conflict, undetectable at the level at which the merge operates.

27.2 Governance regimes and review

Governance regimes differ in who is authorized to decide and under what conditions. Three canonical forms may be distinguished. In a *sole-maintainer* regime one agent decides. In a *quorum* regime a decision requires approval by at least k of n designated reviewers. In a *hierarchical* regime subordinate maintainers decide for parts of the project and a higher authority for the whole. These are not exhaustive, but they cover the usual cases and permit simple quantitative statements about review.

Proposition 27.4 (Review quorum and common-cause failure)**.** *Let a defective proposal be reviewed by k reviewers. Suppose each reviewer independently detects the defect with probability d, except that with probability β a common blind spot affects all reviewers, in which case none detects it. The probability that the defective proposal receives all k approvals is*

$$\Lambda(k) = \beta + (1 - \beta)(1 - d)^k.$$

This is decreasing in k and bounded below by β for every k. The structure, a floor set by a common cause plus a term that decays with independent replication, recurs in Proposition 47.1, where it governs correlated support for a claim, and it is one of the recurring forms of the book. [PROVED UNDER STATED ASSUMPTIONS]

Proof. With probability β the blind spot is present and all reviewers approve. With probability $1 - \beta$ reviewers act independently and each approves with probability $1 - d$, so all k approve with probability $(1 - d)^k$. Total probability gives the formula. The derivative with respect to k (treating k as continuous) is $(1 - \beta)(1 - d)^k \ln(1 - d) < 0$ for $0 < d < 1$ and $\beta < 1$, and $\Lambda(k) \geq \beta$ since the second term is nonnegative. □

The proposition formalizes a familiar caution. Adding reviewers helps against idiosyncratic oversight and does nothing against shared blind spots. Review in a homogeneous community, where reviewers share assumptions, tools, and training, has a floor of failure set by the shared blind spot, and the practical path to reducing the floor is diversity of reviewers and

independent verification mechanisms, such as tests written by different people and automated checks that do not share the reviewers' assumptions.

27.3 The role of automated checks

Automated checks occupy a position between policy and correctness. They are mechanical, so their pass or failure feeds the predicate P_G. They are also intended as partial evidence for C_σ. The relation between the two is the central topic of Chapter 33, where it is shown that passing checks is evidence for correctness relative to the checks' coverage and no more. For present purposes it suffices that the independence in Theorem 27.2 extends to the checks: there are proposals that pass every check and are wrong, and proposals that fail a check and are right, because checks have finite coverage and may themselves be defective.

Chapter 28

Forks, Federation, and Governance Divergence

28.1 The three-way merge as a partial operation

When two histories diverge from a common base and later wish to be reconciled, the standard tool is the three-way merge. It takes a merge base and two descendants and produces, when possible, a tree combining the changes of both. Formally, a tree is a partial function from paths to contents, and the changes made by a descendant are the paths at which it differs from the base.

Definition 28.1 (Trees and change sets)**.** Let Path be a set of paths and Cont a set of contents together with a distinguished element $\perp$ for absence. A *tree* is a function $T : \mathsf{Path} \to \mathsf{Cont} \cup \{\perp\}$ with finite support. For trees T_0 (the base), T_1, T_2, the *change sets* are $D_i = \{p : T_i(p) \neq T_0(p)\}$ for $i = 1, 2$. The *pathwise merge* $\mathrm{M}(T_0, T_1, T_2)$ is defined when for every $p \in D_1 \cap D_2$ we have $T_1(p) = T_2(p)$, by

$$\mathrm{M}(T_0, T_1, T_2)(p) = \begin{cases} T_1(p) & p \in D_1 \\ T_2(p) & p \in D_2 \setminus D_1 \\ T_0(p) & \text{otherwise,} \end{cases}$$

and is undefined otherwise.

Real merges operate below the level of whole files, merging line-level hunks within a file, but the pathwise merge captures the structure that matters for the following results, and the generalization to finer granularity changes details and not the logical form.

Theorem 28.2 (Properties of the pathwise merge)**.** *Let T_0, T_1, T_2, T_3 be trees with change sets D_i relative to T_0. (i) The merge is symmetric: $\mathrm{M}(T_0, T_1, T_2) = \mathrm{M}(T_0, T_2, T_1)$ whenever either is defined; (ii) If $D_1 \cap D_2 = \varnothing$ the merge is defined, and its change set is $D_1 \cup D_2$; (iii) If D_1, D_2, D_3 are pairwise disjoint, then merging in either order gives the same result: $\mathrm{M}(T_0, \mathrm{M}(T_0, T_1, T_2), T_3) = \mathrm{M}(T_0, T_1, \mathrm{M}(T_0, T_2, T_3))$; (iv) If some $p \in D_1 \cap D_2$ has $T_1(p) \neq T_2(p)$, then no tree T satisfies $T(p) = T_1(p)$ and $T(p) = T_2(p)$, and the merge is undefined. Mergeability is therefore not universal.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) On $p \in D_1 \cap D_2$ definedness gives $T_1(p) = T_2(p)$ so both orders return the same value; on $D_1 \setminus D_2$ both return $T_1(p)$; on $D_2 \setminus D_1$ both return $T_2(p)$; elsewhere $T_0(p)$. (ii) With empty intersection there is no constraint to check. At $p \in D_1$ the result is $T_1(p) \neq T_0(p)$; at $p \in D_2$ the result is $T_2(p) \neq T_0(p)$; elsewhere it equals $T_0(p)$. So the change set is $D_1 \cup D_2$. (iii) Let $T_{12} = \mathrm{M}(T_0, T_1, T_2)$, which has change set $D_1 \cup D_2$ by (ii). Its intersection with D_3 is empty, so the outer merge is defined. At $p \in D_1$ the result is $T_1(p)$; at $p \in D_2$, $T_2(p)$; at

$p \in D_3$, $T_3(p)$; elsewhere $T_0(p)$. The same description results for the other bracketing. (iv) is immediate since a function has one value at p. □

Part (iv) is the countermodel to universal mergeability: two histories that make incompatible changes to the same path have no common continuation that retains both. Example 27.3 shows that the converse also fails: disjointness of change sets does not guarantee that the merged program is correct. The two together say that textual mergeability is neither necessary nor sufficient for semantic compatibility.

28.2 Divergence of governance

Divergence of two histories is a technical fact. Divergence of governance is an institutional one. Two forks may disagree not about the content of a change but about whether such changes should be accepted at all, as when one fork adopts a stricter review policy or a different license, or declares different goals. Merging content across such a boundary is an act with institutional consequences: the importing repository adopts decisions made under another regime.

Definition 28.3 (Federation)**.** A *federation* is a family $\{(r_i, \Gamma_i)\}_{i \in J}$ of repositories each with its own governance regime, together with a set of *import operations* $\mathrm{imp}_{i \leftarrow j}$ by which repository r_i may incorporate commits or changes from r_j. An import is *gated* if its execution requires $\Gamma_i(h_i, \delta) = 1$ for the change δ being imported.

Proposition 28.4 (Governance sovereignty of gated federations)**.** *In a federation in which every import is gated, no sequence of operations performed by agents outside r_i changes the accepted history of r_i except through a decision $\Gamma_i = 1$ taken under the governance of r_i. Consequently, the accepted history of each member is determined by its own governance and the offers made to it, and not by the governance of other members.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By definition, the only operation by which changes enter r_i from outside is a gated import, and gating requires $\Gamma_i = 1$. Operations in other repositories do not modify the object store of r_i. Hence the history accepted by r_i is the result of decisions of Γ_i among offered changes. □

The proposition is a simple consequence of definitions, and it states the sense in which federation can preserve independent governance while still permitting reuse. Each member of the federation retains the exit option and the veto. The cost is that coordination between members requires explicit decisions at the boundary, which is slower than a single regime.

28.3 When separation is appropriate

It is tempting to treat reconciliation as always desirable. The foregoing analysis suggests that it is not. Four conditions favor preserving separation. The first is incompatible change sets in the sense of Theorem 28.2(iv), where the merge is undefined. The second is incompatible licenses (Proposition 20.5), where the merge is legally impermissible. The third is divergent governance, where the importing party would inherit decisions it did not endorse. The fourth is divergent goals, where the parties do not want the same artifact. In any of these cases the appropriate relation between the repositories is not merger but selective, gated exchange.

Conjecture 28.5 (Federation scaling)**.** *Let a federation of N repositories have pairwise import costs bounded below by a positive constant. Then the expected fraction of useful cross-repository changes that are actually imported decreases as N grows, unless the import cost is reduced by shared infrastructure such as common metadata conventions.* [CONJECTURAL]

The conjecture is offered because it is precise enough to be tested on archives of real repositories: one would measure the proportion of changes in related forks that were later carried across and how this varied with the number of related forks. No result is claimed.

Chapter 29

Portability, Dependence, and Institutional Capture

29.1 History is portable; the institution is not guaranteed to be

Most of the quantities that people value in a repository are not in its object store. The object store carries content and ancestry (Theorem 16.2). The rest, including the record of discussions, the review comments, the permissions, the identities attached to contributions, the links from outside documents, the release assets, the configuration of automated pipelines, and the project's social standing, is held in other components. A migration that moves the object store has moved the history and not the institution.

Definition 29.1 (Institution state)**.** The *state of a repository institution* is a tuple $\mathcal{I} = (O, R, B, P, \mathrm{Id}, X)$ in which O is the object store, R the set of refs, B the platform records (issues, reviews, discussions, release assets, pipeline configurations), P the permissions and governance settings, Id the bindings between accounts and the identities that appear in O, and X the external references, meaning the links, citations, and registry pointers held by outsiders that resolve to components of $\mathcal{I}$.

A migration is a map from the state of a source institution to that of a target. Its quality is assessed by the functions the institution performed and whether the target performs them. A function is a map from states to outputs; the following definition identifies the components on which it depends.

Definition 29.2 (Institutional function and its dependence set)**.** An *institutional function* is a map φ from institution states to outputs (for example, *the list of authors of a release*, *the set of approvals required for the next merge*, *the resolution of a given citation*). A component $\kappa \in \{O, R, B, P, \mathrm{Id}, X\}$ is in the *dependence set* $D(\varphi)$ if there exist states $\mathcal{I}, \mathcal{I}'$ differing only in κ such that $\varphi(\mathcal{I}) \neq \varphi(\mathcal{I}')$.

29.2 Lineage preservation and its limits

The first result concerns what a migration that moves only Git data does preserve.

Theorem 29.3 (Lineage preservation theorem)**.** *Let μ be a migration that copies the object store and the refs of a source institution into a target institution, byte for byte, and assume the hypothesis of Theorem 16.2. Then (i) every commit identifier of the source is a commit identifier of the target, with the same tree, parents, and message; (ii) the ancestry order of the source is the ancestry order of the target restricted to the copied commits, and the two coincide when the target contains no additional commits; (iii) every blob has the same identifier and content; (iv) every ref points to the same commit. On the other hand, (v) no function of (O, R) determines B, P, Id, or X, so no migration restricted to*

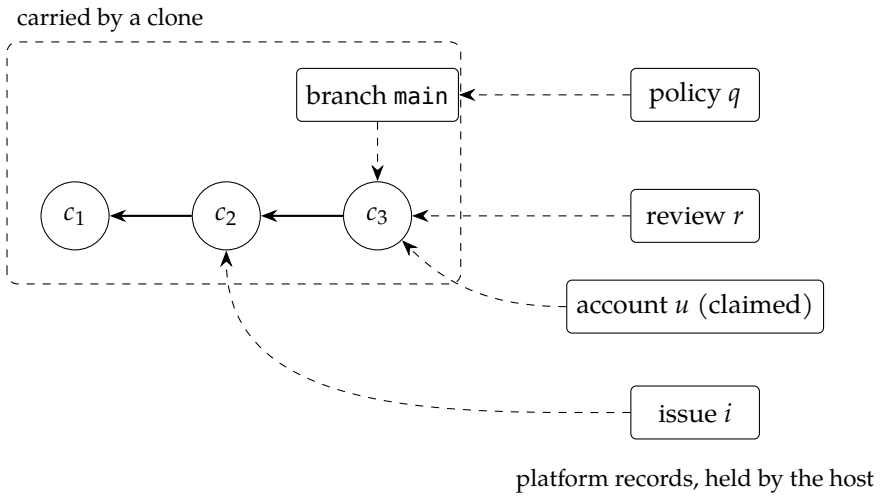


Figure 29.1: A Git graph with a hosted extension, in the vocabulary of Table 24.1. Solid arrows are parent edges. Dashed arrows are references from a reference or platform record to a lower vertex. The dashed rectangle encloses what a mirror of objects and refs carries; the platform records outside it are not determined by it (Theorem 29.3). The records shown are generic, not those of any particular hosting service.

(O, R) *can preserve any institutional function whose dependence set contains one of these.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Parts (i) to (iv) follow from Theorem 16.2 and Proposition 16.3(i)–(ii) since copying is bytewise. For (v), Proposition 24.3 constructs, for any native graph, hosted extensions that differ in the platform layer while sharing the native graph; extending the construction to permissions, identity bindings, and external references by adding or altering these components while leaving (O, R) fixed gives states that agree on (O, R) and differ elsewhere. If φ depends on a component other than O or R, such a pair of states gives different values of φ, but any migration that is a function of (O, R) maps them to the same target state, so it cannot preserve φ on both. □

29.3 The institutional portability criterion

The theorem has a converse in spirit: a migration preserves an institutional function when it transports, or reconstructs, everything on which the function depends.

Theorem 29.4 (Institutional portability criterion)**.** *Let μ be a migration with transported components $T \subseteq \{O, R, B, P, \mathrm{Id}, X\}$, and let φ be an institutional function. (i) If $D(\varphi) \subseteq T$, and the target environment implements each transported component faithfully with respect to φ, then $\varphi(\mu(\mathcal{I})) = \varphi(\mathcal{I})$. (ii) If $D(\varphi) \not\subseteq T$ and μ's output on each omitted component is a fixed default independent of the source, then there exist source states for which $\varphi(\mu(\mathcal{I})) \neq \varphi(\mathcal{I})$. Thus a migration preserves φ for all states iff it transports every component of $D(\varphi)$ faithfully.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) By definition of the dependence set, φ is determined by the components in $D(\varphi)$, provided the other components are held fixed in a way that does not matter, meaning that

changing them alone never changes φ. Since those components are transported faithfully, the target agrees with the source on $D(\varphi)$ and so gives the same value. (ii) Let $\kappa \in D(\varphi) \setminus T$. By definition there are states $\mathcal{I}_1, \mathcal{I}_2$ differing only in κ with $\varphi(\mathcal{I}_1) \neq \varphi(\mathcal{I}_2)$. Since the migration outputs the same default for κ and agrees on everything else, $\mu(\mathcal{I}_1) = \mu(\mathcal{I}_2)$, so $\varphi(\mu(\mathcal{I}_1)) = \varphi(\mu(\mathcal{I}_2))$. This common value cannot equal both $\varphi(\mathcal{I}_1)$ and $\varphi(\mathcal{I}_2)$. □

The criterion is nearly a tautology once definitions are in place. Its use is to organize the practical analysis of migrations as an inventory: list the institutional functions that matter, determine for each the components on which it depends, and check that each component is either transported or reconstructible. The common failure is to inventory the files and neglect the functions. Because the functions that matter most to outside parties, such as the resolution of citations and the stability of dependencies, depend on external references X, the stability of the identifiers by which others refer to the project determines in large measure whether a migration is experienced as continuity or as disappearance.

29.4 Degrees of dependence and capture

Dependence on a provider is a matter of degree. A project that uses a hosting service only to store a mirror of a repository whose primary copy is elsewhere has little dependence. A project whose issues, reviews, automation, releases, and documentation exist only in the service has much. The degree can be defined in terms of the institutional functions that would be lost under migration.

Definition 29.5 (Dependence index and capture)**.** Let Φ be a finite set of institutional functions the project relies on, each assigned a weight $w_\varphi > 0$ with $\sum w_\varphi = 1$. Let T_0 be the set of components that are portable at low cost (for the project, with its chosen tools and formats). The *dependence index* on a provider is

$$\Omega = \sum_{\varphi \in \Phi,\ D(\varphi) \not\subseteq T_0} w_\varphi,$$

the weight of functions that depend on components not freely portable. A project is *captured* by a provider with respect to a function φ if φ depends on a component that only the provider controls and for which no export exists that satisfies Theorem 29.4.

The index is a modeling device, not a measurement, and its weights are judgments. It serves to keep the discussion of lock-in tied to specific functions and to avoid vague assertions. The components are of different character. Git objects are portable by design. Issue text and comments can often be exported, but the links between them and the commit graph may be damaged, and the numbering schemes by which others refer to them do not transfer. Review state and approvals are generally exportable only in part. Stars, follows, and other attention records are not portable at all in practice, since they are relations among accounts on the provider's namespace. Automation configurations are frequently tied to the provider's execution environment and syntax. Account identity is the most thoroughly captured of all, because it is a namespace the provider controls.

29.5 Mitigations

Mitigations are known and fall under familiar headings. Regular mirroring of the object store to independent hosts addresses O and R. Periodic export of platform records into the repository itself, in open formats, addresses B partially by moving platform-layer content down into the Git layer, which is portable by Principle 24.7. Documenting governance in files in the repository addresses P by making it part of the artifact. Using identifiers that do not depend on a single host for citations and dependencies addresses X; intrinsic identifiers derived from content, in the style of those used by software archives, are an example, as are persistent identifiers resolved through independent services. Signing commits and tags with keys the author controls addresses part of Id by decoupling authorship from the provider's account namespace. None of the mitigations is complete, and each costs effort, which is exactly the tradeoff described in Definition 4.4.

Principle 29.6 (Pull the institution into the repository)**.** To the extent that the functions of an institution can be represented as files in the Git layer, they inherit the portability of the Git layer. A design that places policies, records of decisions, mirrors of discussions, and attestations in the repository itself reduces dependence on any host. [DESIGN ARGUMENT]

The principle is the guiding design idea of the architecture proposed in Part IX. Its limit is that some functions are inherently relational and public, such as being discoverable by strangers, and cannot be moved into a private store. Discovery requires an index maintained by someone, and in that sense every institution depends on infrastructure it does not entirely control. The question is therefore not whether to depend but on what, how many alternatives exist, and what exit costs.

29.6 Result of Part VI

Part VI has produced a typed-graph model of repository institutions, with the native Git layer as a proper substructure (Proposition 24.3); invariants of lineage under standard operations (Theorem 25.4); a symmetry result showing that structural rules cannot select an authoritative continuation (Theorem 26.2) together with a coordination model for the emergence of recognized upstreams (Proposition 26.7); a proof of the independence of four notions of acceptable contribution (Theorem 27.2); properties of merging that delimit when reconciliation is possible (Theorem 28.2); and a lineage preservation theorem and portability criterion (Theorems 29.3 and 29.4). Part VII applies the same vocabulary to the particular institution of computational science.

Part VII applies the machinery of this part to a particular kind of institution, the community that produces and evaluates scientific results. The lineage, authority, and portability results of this part concern the records of an institution and the functions that depend on them, and the next part asks which records a computational scientific claim needs in order to be reproduced, verified, and assigned priority, and in what sense the repository supports each of those functions.

Part VII

The Repository as Scientific Instrument

Chapter 30

The Incomplete Scientific Publication

30.1 What a paper can carry

The scientific article is a compression. It reports a question, a method, a result, and an interpretation, in a length determined by the conventions of a journal, and it assumes a reader who shares the discipline's background. For experiments in which the method is a procedure carried out by hand with standard equipment, the compression may be tolerable: a skilled reader can reconstruct the procedure from the description. For computational work the compression discards most of what a reconstruction needs. The method is a program of thousands of lines, run on data of unstated provenance, under library versions that determine its behavior, with parameters chosen after exploration that is not reported. The prose gives an account of the program, and the account is not the program.

Concern about this gap has a long history and a large literature. Early proposals that a computational article should be accompanied by the code and data that generated its figures, so that the article is merely the advertisement for the scholarship, date to the 1990s [6]. The tradition of literate programming, in which a program is written as an explanation to human readers with code interleaved, had earlier posed the same question from the side of the programmer [22]. The case for treating published computational results as reproducible artifacts has been made repeatedly [28]. A decade of surveys and replication attempts has documented how frequently results fail to be reconstructed: in one widely cited survey of researchers, a majority reported having failed to reproduce another scientist's experiment, and a substantial fraction reported failing to reproduce their own [3]. Evaluations of whether published computer-systems papers could be built from the authors' released code found that a minority could be built without effort [8]. The causes are mundane: missing files, unstated versions, hard-coded paths, deleted data.

30.2 The specification gap

The difficulty can be put formally. Let $\mathcal{X}$ denote the full collection of artifacts and conditions from which a result was produced, and let a publication be a map π from $\mathcal{X}$ to a short description.

Definition 30.1 (Publication as projection)**.** Let Exp be a set of fully specified computational experiments. A *publication format* is a function $\pi : \mathsf{Exp} \to \mathsf{Desc}$ to a set of descriptions, and the *specification gap* of a format at $\mathcal{X} \in \mathsf{Exp}$ is the set $\pi^{-1}(\pi(\mathcal{X})) \setminus \{\mathcal{X}\}$ of experiments that the format cannot distinguish from $\mathcal{X}$.

Proposition 30.2 (Prose publication has a nonempty specification gap)**.** *Suppose descriptions are finite strings over a finite alphabet with length at most ℓ, and suppose* Exp *contains more than $|\Sigma|^{\ell+1}$ elements, where Σ is the alphabet. Then the specification gap is nonempty for at least one*

experiment, and the proportion of experiments with empty gap is at most $|\Sigma|^{\ell+1}/|\mathsf{Exp}|$. [PROVED UNDER STATED ASSUMPTIONS]

Proof. There are fewer than $|\Sigma|^{\ell+1}$ strings of length at most ℓ, so π cannot be injective on a larger set, by the pigeonhole principle. An experiment has an empty gap iff its description is not shared by any other experiment; the number of such experiments is at most the number of descriptions. □

The result is a counting argument and has a corresponding limitation: real descriptions are not uniformly distributed over the possible experiments, and the interest is in whether the particular gap matters for the claim. Experiments that differ in irrelevant details yield the same result, and the gap is harmless. But the proposition identifies the issue: any bounded-length description leaves some distinctions undeclared, and the distinctions that matter cannot be known in advance by the author. Publishing the experiment itself as an executable artifact, in addition to the description, reduces important classes of this ambiguity, but it is not a complete solution: an executable artifact depends in turn on interpreters, hardware, environmental assumptions, and external resources, and it must be accompanied by a statement of which equivalence of outputs counts as agreement, since unique reconstruction of every irrelevant detail is not needed to reproduce a finding. The remainder of this part examines what executable publication requires.

30.3 Reproduction, replication, and the vocabulary of agreement

The terms in use for the relation between a later study and an earlier one are inconsistently applied across disciplines. A convention proposed in a report of a national academy distinguishes *reproducibility*, defined as obtaining consistent results using the same input data, computational steps, methods, code, and conditions of analysis, from *replicability*, defined as obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data [26]. A three-way refinement that matches the structure of the experiment defined in the next chapter.

Definition 30.3 (Three kinds of agreement)**.** Given a published computational result y obtained from an experiment $\mathcal{X}$: *Rerunning* is executing the released artifacts, unchanged, in some environment, to obtain an output; *reproducing* is obtaining an output that agrees with y within a stated tolerance by rerunning in a way that satisfies the declared specification; *replicating* is obtaining a result that agrees with the scientific finding by an independent study, with new data or an independently constructed implementation.

The three are logically distinct. A rerun can succeed without reproduction if the output differs from the published one. Reproduction can succeed without replication, since a program that contains an error reproduces its error faithfully. Replication can succeed without reproduction, as when an independent implementation confirms a finding that the original code failed to produce in a rerun. Failure at one level does not imply failure at another, and an institution that wishes to know whether a finding is reliable needs evidence at the level of replication, which no amount of reproducibility can provide.

Remark 30.4 (Reproducibility is necessary but not sufficient)*.* Computational reproducibility is a minimal requirement of transparency: it shows that the reported number follows from

the reported procedure. It does not show that the procedure is a good test of the hypothesis. A fully reproducible study can have a flawed design, an invalid measurement, or a model that does not fit the world. Part of the case for treating repositories as scientific instruments is that they make the first condition cheap to satisfy and so direct attention to the others.

Chapter 31

Reproducibility as a Dependency Structure

31.1 The experiment as a structured object

An experiment, for present purposes, is a six-component object.

Definition 31.1 (Computational experiment)**.** A *computational experiment* is a tuple $\mathcal{X} = (C, D, E, \Theta, S, M)$ where C is the code (including build scripts), D is the data (including input files and any external services queried), E is the execution environment (operating system, libraries, hardware where relevant), Θ is the parameter configuration, S is the specification of randomness (seeds and generators), and M is the measurement procedure that maps raw outputs to reported quantities. The *raw output* of a run is a function $\mathrm{out}(C, D, E, \Theta, S)$ and the *reported result* is $M(\mathrm{out}(C, D, E, \Theta, S))$.

The decomposition is a modeling choice, and other decompositions would serve. Its virtue is that each component is an item to be preserved in the sense of Chapter 4, and each has its own failure modes. Code fails through missing versions and undeclared files. Data fails through deletion, modification, and silent updates of external sources. The environment fails through version drift (Proposition 11.3). Parameters fail through reliance on interactive exploration whose results are not recorded. Randomness fails through unseeded generators and nondeterministic parallelism. Measurement fails through undocumented post-processing, such as the removal of outliers by hand.

31.2 Declared inputs and the closure criterion

A run reads from many places. Only some are declared in the specification, and the question is whether the declared ones suffice.

Definition 31.2 (Reads and declared closure)**.** For a run ρ of the experiment, let $\mathrm{Reads}(\rho)$ denote the set of artifacts (files, environment variables, network resources, system clock readings, hardware identifiers, and so on) from which the run reads during execution. Let Decl be the set of artifacts declared in the specification, and let $\mathrm{cl}(\mathrm{Decl})$ denote their closure under declared dependencies (Definition 11.1). The specification is *closed for* ρ if $\mathrm{Reads}(\rho) \subseteq \mathrm{cl}(\mathrm{Decl})$. It is *closed* if it is closed for every run permitted by the specification.

The definition requires a notion of what counts as reading. For practical purposes this includes any information obtained from outside the process's own declared inputs: the time of day, the machine's name, the order in which a directory is listed, a random device. The reads of a program can be recorded by instrumenting its interface to the operating system, and the experiment reported in Appendix F does so for a small example.

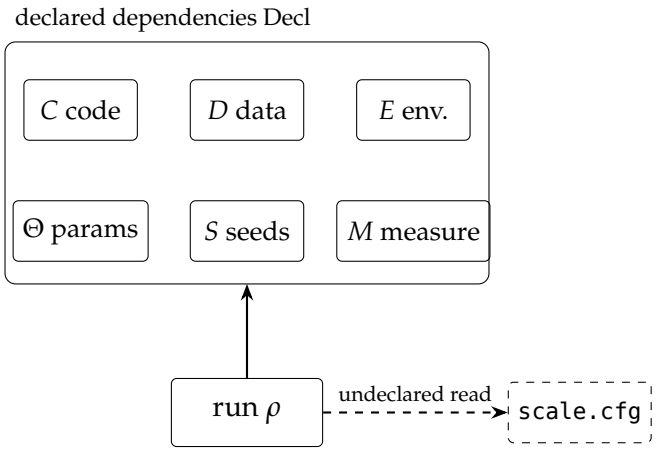


Figure 31.1: The six components of a computational experiment (Definition 31.1) and a run that reads an artifact outside the declared set, as in the situation of Theorem 31.3 and experiment E6. The solid arrow denotes the reads that fall in Decl; the dashed arrow is a read of an undeclared configuration file, and the specification is then not closed. The configuration file name is that of the toy experiment of Appendix F.

Theorem 31.3 (Reproducibility closure criterion)**.** *Let the raw output of a run be a deterministic function of the values of the artifacts it reads,* $\text{out} = F(\text{Reads}(\rho))$. (*i*) (*Sufficiency*) *If the specification is closed for a run* ρ *and the artifacts in* $\text{cl}(\text{Decl})$ *are reproduced exactly, then any rerun in which these artifacts take the same values yields the same raw output;* (*ii*) (*Necessity of closure for guaranteed reproduction*) *If the specification is not closed for* ρ*, in that* ρ *reads an artifact* $u \notin \text{cl}(\text{Decl})$ *on which* F *depends nontrivially, then there are reruns with identical values of all declared artifacts that yield different raw outputs;* (*iii*) *Closure of the specification is a property of the specification and does not imply that a rerun has been performed or that its output agrees with the published result.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) By closure, every read lies in $\text{cl}(\text{Decl})$, so the values read in the rerun coincide with those of the original, and determinism gives equal outputs. (ii) By nontrivial dependence there exist two values $u_1 \neq u_2$ of u with different outputs when all other reads are fixed. Rerunning with $u = u_1$ and $u = u_2$ and identical declared artifacts gives different outputs. (iii) Closure is a relation between the read set and the declared set, evaluated without reference to any execution's results; the agreement of outputs is a different proposition about values. □

The theorem separates three questions that are often run together. Is the specification complete in the sense of declaring all that is read? Has a reproduction been successfully carried out? Does the reproduced result support the claim? Closure is the first, a checkable structural property of the specification; success of a rerun is the second, an empirical fact about a particular execution; and support of the claim is the third, treated in Chapter 33. A reasonable aim for a published repository is that the first be verified mechanically, the second be demonstrated by an independent party, and the third be argued openly.

Remark 31.4 (Nondeterminism and tolerance). The determinism assumption fails for many scientific programs. Parallel reductions in floating-point arithmetic depend on the order of operations, which depends on thread scheduling; hardware differences produce different rounding; library versions change algorithms. For such programs exact agreement is not to

Table 31.1: Components of an experiment held fixed under three kinds of agreement.

	C	D	E	Θ	S	M
Rerun	fixed	fixed	fixed	fixed	fixed	fixed
Reproduction	fixed	fixed	varies within spec	fixed	fixed	fixed
Replication	independent	independent	varies	may vary	varies	construct fixed

be expected, and the appropriate notion is agreement within a tolerance ϵ determined by the sensitivity of the claim. The closure criterion extends: closure then guarantees that the declared inputs fix the output distribution, and reproduction is the agreement of distributions or summary statistics up to ϵ.

31.3 Rerunning, reproducing, replicating as levels of the dependency graph

The three kinds of agreement of Definition 30.3 correspond to which components of $\mathcal{X}$ are held fixed. A rerun holds all six fixed. Reproduction holds C, D, Θ, S, M fixed and allows E to vary within the specification. Replication varies D or C (new data or an independent implementation) and E, and holds only the scientific question and, ideally, the measurement construct fixed. The relationship is displayed in Table 31.1.

The table suggests why repositories are suited to the first two levels and not the third. A repository can preserve C, D (when permitted), Θ, S, M, and a description or a machine-readable specification of E. It cannot, by itself, produce independent data or independent implementations. Those require other researchers. The instrument supports the first two levels and facilitates the third by lowering the cost of the first.

31.4 A catalog of failure modes

The failure modes of reproduction can be organized by which component of the experiment is responsible. A catalog is given in Appendix D. The point to be made here is structural: every failure mode is a failure of one of the persistence properties of Chapter 2 for one of the six components. A missing file is a failure of physical persistence for D or C. An unlinked dataset is a failure of referential persistence. An undocumented parameter is a failure of operational persistence for Θ. The framework of the early chapters thus applies unchanged, and the problem of reproducibility is a case of the problem of continuation (Definition 4.7) with $K = K_{\text{rep}}$.

Chapter 32

Provenance and the Executable Scientific Record

32.1 What a hash, a tag, and a signature establish

A repository that serves as a scientific record uses a small set of instruments for provenance. Commits record states of the artifacts. Tags name particular commits, typically those corresponding to a release or to a version cited in a paper. Content hashes identify artifacts independently of location. Checksums published alongside downloads allow verification of integrity. Digital signatures bind a statement to a key. Build manifests list the inputs and steps of a build. Persistent identifiers, issued by archival services, give citable names that resolve over long periods. Each instrument establishes something, and each is regularly credited with establishing more.

Definition 32.1 (Integrity, authenticity, validity)**.** For an artifact x and a statement s about it: *Integrity* of x with respect to a reference identifier h is the property $H(x) = h$. *Authenticity* of s is the property that s was produced by a specified agent, evidenced by a signature verifiable under a key bound to that agent. *Validity* of x for a purpose is the property that x is correct for that purpose, in the sense of satisfying a specification that is itself adequate to the purpose.

Proposition 32.2 (The three properties are independent)**.** *Integrity does not imply authenticity or validity; authenticity does not imply integrity of any artifact not covered by the signature, nor validity; validity does not imply integrity or authenticity. All combinations occur.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. A hash of an arbitrary file shows integrity relative to that hash, with no signature and no assurance of correctness. A signed statement of the form *this program is correct* is authentic, but the signature covers the statement and not necessarily the program, and the program may contain errors. A correct program stored without a published identifier has validity with no integrity reference. For validity without authenticity, consider a correct program of unknown authorship. The remaining combinations arise by combining these cases, for example a signed hash of a correct program. □

The practical import is a caution against reading the instruments as certificates of correctness. The commit that is tagged as the version used in a paper proves that the repository contained that state at that time, according to its records. It proves nothing about whether the code does what the paper says it does.

32.2 Authorship and timestamps are claims

As observed in Proposition 18.2, the author fields of a commit are unauthenticated. The same is true of the commit timestamps, which are supplied by the committer's clock or explicitly set. A commit can be given any timestamp.

Proposition 32.3 (Commit timestamps are unauthenticated)**.** *For an unsigned commit, the author date and committer date are fields supplied at creation. For any commit content and any chosen date, a commit with that date can be created, and the resulting identifier is a function of the content including the chosen date. Consequently the timestamps in a repository, taken alone, do not bound the time at which the repository's objects came into existence.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. The creation operation accepts the date as an input, as it does the author. Two commits with different dates and otherwise identical content have different identifiers, both valid. No part of the data structure ties the date to a physical clock. □

Even signatures on commits and tags usually do not authenticate the time unless the signature scheme incorporates a trusted timestamp. A holder of a key can sign a commit bearing a false date. What a signature establishes is that a key-holder asserted the content, including the claimed date, not that the assertion is true.

32.3 External anchoring

The remedy for unauthenticated time is external anchoring: recording the identifier of an artifact in a record maintained by a party independent of the author, whose own timestamps are trusted by the audience. Examples include a timestamping service following a standard protocol, a publication in a journal or preprint server, an archive that independently crawls and stores a public repository, an entry in a public append-only log, and a message in a widely mirrored mailing list. Each places the identifier, or content from which it can be computed, into a record whose time the author cannot alter.

Theorem 32.4 (Anchoring bound)**.** *Let H satisfy two hypotheses:* (*i*) unpredictability*, meaning that the artifact class has high min-entropy relative to the submitter, so that no feasible submitter can name the artifact x before it has been produced except with negligible probability; and* (*ii*) preimage resistance*, meaning that given h no feasible procedure finds an x with $H(x) = h$ except with negligible probability. Suppose a record whose entries and times cannot be altered by the submitter contains, at time T, an entry citing the identifier h. If the submitter later exhibits an artifact x with $H(x) = h$, then with overwhelming probability x was in the submitter's possession no later than T. The entry thus gives an upper bound T on the time of existence of x in the submitter's possession.* [PROVED UNDER STATED ASSUMPTIONS] (*UNDER THE STATED CRYPTOGRAPHIC ASSUMPTIONS AND TRUST IN THE RECORD*)

Proof. The entry fixes h at time T. If x had come into the submitter's possession only after T, then the submitter would have produced a preimage of the previously fixed value h, either by search or by guessing. Preimage resistance (ii) excludes the search, and unpredictability (i) excludes the guess. Neither hypothesis can be dropped. Without (i), a submitter who can anticipate a short or formulaic artifact can commit to its hash in advance and write the artifact afterward, so that the entry certifies nothing about when the content existed. Without (ii), a submitter who can invert H can produce an artifact for a previously fixed h. The alternative

that x was obtained from another party who held it before T is consistent with the conclusion, which concerns possession and not authorship. □

The theorem gives only an upper bound on the time of possession. It does not show who created x, since the submitter may have received it from someone else. The high-entropy hypothesis is essential: a hash of a short, predictable text can be anticipated by an adversary who commits to the hash in advance and writes the text afterward. Nor does it establish that x was published, only that its hash was recorded; an entry with the hash alone does not make the content public. Priority claims therefore need two components that the theorem separates: an anchored hash showing that the content existed by T, and a public disclosure showing what the content was and who made it available. We return to the distinction in Chapter 34.

32.4 Persistent identifiers

For citation, the instrument of choice is a persistent identifier assigned by a service committed to resolving it for the long term. Services for research output, including generalist repositories, assign digital object identifiers to deposits, and integrations let a release of a repository be deposited and receive such an identifier. Archives of source code assign identifiers derived from the same content-addressing principles as Git, so that an identifier is computed from content and does not depend on the archive’s naming [10]. Content-derived identifiers are persistent in the sense of Chapter 2 for any holder of the identifier who can find any copy of the object. Assigned identifiers are persistent as long as the registry resolves them. These are different guarantees with different failure modes, and a reference to a software artifact in a publication is most robust when it includes both: an assigned, resolvable identifier for human use, and a content-derived identifier for verification.

Chapter 33

Continuous Verification and Scientific Revision

33.1 Three distinct activities

The words *testing*, *verification*, and *validation* are often used loosely. In the literature on scientific computing they have distinguishable meanings: verification is the assessment of whether a computational model implements its specification correctly, the question of whether the equations are solved right; validation is the assessment of whether the specification represents reality with sufficient accuracy for the intended use, the question of whether the right equations are solved [27]. To these a third is added here.

Definition 33.1 (Three levels of assessment)**.** For a computational study with code C, specification Spec, empirical claim κ, and world $\mathcal{W}$: *Verification* is the assessment that C satisfies Spec. *Validation* is the assessment that Spec is an adequate model of the relevant part of $\mathcal{W}$. *Inference* is the assessment that the output of C, together with the validated model, supports the claim κ.

Proposition 33.2 (Verification does not entail inference)**.** *There exist studies in which C perfectly satisfies* Spec *(verification succeeds with certainty) and the claim κ derived from the output is false.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let Spec describe a linear regression of an outcome on a predictor, and let C be a verified implementation. Let the data come from a process in which the outcome depends on an unmeasured confounder correlated with the predictor, so that the fitted coefficient does not equal the causal effect. The claim κ that the predictor causes the outcome with the fitted magnitude is false although C implements the regression exactly. □

The example explains the observation that a mathematically correct implementation can still fail to substantiate an empirical claim. The failure lies in validation, since the specification does not model the confounding, or in inference, since the step from regression coefficient to causal claim is unjustified. Neither failure is visible to tests of the code.

33.2 Tests as finite samples

The workhorse of verification in practice is the automated test: an input, an expected output, and a check. A collection of tests samples the behavior of a program, and passing the tests is evidence of correctness proportional to the coverage of the sample.

Proposition 33.3 (Passing tests does not establish a universal property)**.** *Let f be a program computing a function on an infinite domain and let $\mathcal{T}$ be a finite set of test inputs on which f agrees*

with a specification g. Then there exist infinitely many programs f' that agree with f on $\mathcal{T}$ and disagree with g elsewhere. [PROVED UNDER STATED ASSUMPTIONS]

Proof. For each input $z \notin \mathcal{T}$ define f'_z by $f'_z(w) = f(w)$ for $w \neq z$ and $f'_z(z) \neq g(z)$ (choose any value different from $g(z)$). Each f'_z agrees with f on $\mathcal{T}$ and disagrees with g at z. There are infinitely many choices of z. □

The proposition is the logical content of the old remark that tests can show the presence of errors and not their absence. It does not make tests useless. It identifies what a test suite is: a finite set of constraints narrowing the space of programs consistent with observed behavior. Better suites constrain more of the relevant space. Property-based tests generate many inputs from a stated property. Coverage measurements report which parts of the program the tests execute, which is necessary for tests to say anything about those parts and not sufficient for them to say it correctly.

Example 33.4 (Complete coverage, incorrect program)**.** A function intended to return the maximum of two numbers is implemented as returning the first argument if it exceeds the second and otherwise the second argument plus zero, and a single test checks that the maximum of 3 and 5 is 5. Line coverage is complete. An implementation that always returns the second argument passes the same test and is wrong.

33.3 Continuous integration and its evidential role

Continuous integration is the practice of running a project's checks automatically on every proposed change. Its principal benefit is the early detection of regressions: changes in behavior that the tests encode as unexpected. The evidential role of a passing build is limited by three considerations. First, the checks are chosen by the authors and share their assumptions (Proposition 27.4), so a shared blind spot passes every check. Second, the checks run in an environment that may differ from the environment of use, and a pass in one says little about another unless the environment is specified. Third, the checks verify consistency with the tests and not truth of the scientific claims. A green status badge is a statement that the specified checks ran successfully in the specified environment on the specified commit, and is a stronger and more precise statement than it may appear and a weaker one than it is often taken to be.

33.4 Proof assistants and formalized correctness

Where the object is mathematical, formal proof assistants offer a stronger instrument: a machine-checked proof that a statement follows from stated axioms in a given logic. Systems of this kind, among them a widely used programming language and proof assistant based on dependent type theory [9], are suitable for verification of finite models and of precisely specified transition systems. Their guarantee is exact and its scope is narrow. The guarantee is that the formal statement, as written, is derivable. Whether the formal statement captures the informal claim intended is a separate question, the fidelity of formalization, and it cannot itself be settled inside the proof assistant. A formal proof of a misformalized statement is a correct proof of the wrong theorem.

Principle 33.5 (The fidelity gap)**.** For every formal artifact that is offered as support for an informal claim, there are two questions: whether the artifact is internally correct, which tools can settle, and whether it means what the claim says, which only readers can settle. Institutions that rely on formal artifacts must provide for the second question in their procedures for review and objection. [DESIGN ARGUMENT]

33.5 Revision as a normal condition

A scientific record that includes executable artifacts will be revised. Bugs are found, data are corrected, libraries are updated, claims are qualified. The infrastructure should therefore treat revision as expected and support it with visible history rather than treat it as an embarrassment to be concealed. Version control supplies the mechanism. The institutional question is what a revision means for the claims that relied on the earlier state: whether they stand, are qualified, or are withdrawn, and how readers who relied on them are informed. The question belongs to the epistemic layer and is taken up in Chapter 39.

33.6 Symbolic derivations and the audit of formulas

A published model usually contains derivations: sequences of expressions in which each is obtained from the previous by a transformation justified by an algebraic rule. Such a derivation is an artifact in the sense of Chapter 2, and the operation of checking it is an operation class in the sense of Definition 4.1. It is a particularly instructive case because the information it requires is small and well defined, and because computer algebra systems make the check executable. The case also shows concretely that the operation of evaluating a scientific statement does not presuppose the operation of reproducing an experiment.

Definition 33.6 (Derivation with declared assumptions)**.** A *derivation* is a tuple $\delta = (e_0, e_1, \dots, e_n; \Gamma; \rho_1, \dots, \rho_n)$ in which each e_i is an expression over a set of symbols, Γ is a set of *assumptions* that restricts the values the symbols may take (for example, $x \neq 1$, $x > 0$, or that a symbol is real), and each ρ_i names the rule used to pass from e_{i-1} to e_i. Write $\mathrm{Adm}(\Gamma)$ for the set of assignments of values to symbols that satisfy Γ. The step $e_{i-1} \to e_i$ is *valid under* Γ if for every assignment in $\mathrm{Adm}(\Gamma)$ both expressions are defined and take equal values.

Proposition 33.7 (Validity is relative to the declared assumptions)**.** *(i) If a step is valid under* Γ *and* $\mathrm{Adm}(\Gamma') \subseteq \mathrm{Adm}(\Gamma)$ *then it is valid under* Γ'*. (ii) A step valid under* Γ *need not be valid under a* Γ'' *with* $\mathrm{Adm}(\Gamma'') \supsetneq \mathrm{Adm}(\Gamma)$*. (iii) A step that replaces an equation by its square is valid as an implication on* $\mathrm{Adm}(\Gamma)$ *but not in general as an equivalence, so that solutions of the derived equation need not be solutions of the original.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) Equality at every assignment of a set implies equality at every assignment of a subset. (ii) Take $e_{i-1} = (x^2 - 1)/(x - 1)$ and $e_i = x + 1$ with $\Gamma : \ x \neq 1$. The step is valid, but under $\Gamma'' : \ x \in \mathbb{R}$ the first expression is undefined at $x = 1$ while the second equals 2. (iii) The equation $\sqrt{x + 2} = x$ implies $x + 2 = x^2$, which has the solutions 2 and -1, and only 2 satisfies the original equation. □

The proposition is elementary, and its consequence for preservation is not. A derivation whose assumptions are not recorded can be checked only by a reader who guesses them, and a

guess that is too permissive makes a valid step appear invalid, while one that is too restrictive hides an invalid step. The assumptions Γ are therefore a necessary item of preserved context for the operation of checking, and they play the role of declared inputs in the closure criterion of Theorem 31.3. The same applies to a computer algebra system. Its output is a function of the input expression, of the symbols' declared properties, of the system's version, and of its defaults, such as whether undeclared symbols are taken to be real or complex and which branches of multivalued functions are used. If a run depends on any of these without declaring them, the run is not closed in the sense of Definition 31.2, and two systems, or two versions, may return different results for the same expression without either being in error. Experiment E9 exhibits the dependence for a standard case: in the system used there, the expression $\sqrt{x^2}$ is left unchanged for a symbol with no declared properties, becomes $|x|$ for a real symbol, and becomes x for a positive symbol. It also shows a cancellation that silently removes the point $x = 1$ from the domain, and the extraneous root produced by squaring.

Proposition 33.8 (Checking a derivation requires no interpretation of the symbols)**.** *Let* Chk *be the operation class of checking whether each step of a derivation δ is valid under its declared Γ, and let* Adq *be the operation class of judging whether the model that contains δ is empirically adequate for a stated system. There exist artifacts for which every minimal sufficient context for* Chk *is disjoint from every minimal sufficient context for* Adq*. In particular, a reader holding only the first context can perform* Chk *and has nothing toward* Adq*.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Take the artifact x to be the derivation itself, so that its expressions are the artifact and not items of context. Let $I = \{\Gamma,$ rule names, measurement protocol, data, interpretation of the symbols$\}$. By Definition 33.6, the validity of a step is a function of (e_{i-1}, e_i, Γ) and of the admissibility of the named rule, so $\{\Gamma,$ rule names$\}$ is a sufficient context for Chk, and it is minimal because dropping Γ leaves the domain undetermined (Proposition 33.7(ii)). Judging adequacy for a stated system requires comparing the model with observations under the stated interpretation, so $\{$measurement protocol, data, interpretation$\}$ is a minimal sufficient context for Adq, and it does not contain the items of the first. The two sets are disjoint. □

What the proposition makes precise is the independence of two competences that the monograph elsewhere distinguishes. A reader with no knowledge of the discipline that produced the formulas can check the algebra, and cannot judge whether the model is adequate, and conversely a specialist can judge adequacy and still make an error that a symbolic check would catch. The proposition is a particular case of Proposition 33.2: verification of a derivation does not entail inference about the world. It also supplies a concrete case for the remark in Chapter 4 that evaluation need not presuppose reproduction, since the check of a derivation requires neither the data nor the experiment.

The practical consequence is a record. A derivation preserved for later audit should carry the expressions, the declared assumptions, the name and version of any symbolic system used together with its relevant settings, and the checks applied, for example substitution of random admissible values (E9 applies this to the cancellation above) or an independent derivation by another route. A symbolic result is then a candidate consequence whose validity is conditional on the recorded assumptions and the correctness of the computation, and whose bearing on the scientific question requires the further evidence discussed in this chapter. [DESIGN ARGUMENT]

Two limits belong with this. Symbolic computation can be intractable, and exact results may grow beyond any useful size, so that a failure to complete a derivation is not evidence

against the claim. And a computer algebra system certifies nothing about the correspondence of the model with reality: it can find the roots of an equation without showing that they are realizable states, and a stationary point without showing that it is a global optimum. In the classical predator-prey system $\dot{x} = ax - bxy$, $\dot{y} = cxy - dy$, symbolic computation gives the interior equilibrium $(d/c, a/b)$, the Jacobian there, and its eigenvalues $\pm\sqrt{-ad}$, which are purely imaginary for positive a and d (E9). That identifies a linearized center. It does not by itself establish nonlinear stability, still less that any ecological system behaves as the model describes, and an auditor who stops at the eigenvalues has verified an algebraic fact and has not evaluated a scientific claim.

Chapter 34

Scientific Priority and Public Research Lineage

34.1 Priority as a social institution

Scientific priority, the recognition accorded to the first to announce a discovery, is an institution with a long history, a significant place in the motivations of researchers, and a record of dispute [25]. Its rules are conventions of the scientific community: priority generally goes to the first to publish, with publication understood as public communication in a form others can examine and build on. These conventions are distinct from the rules of intellectual property law, which differ among jurisdictions and have changed over time. A statement about priority in the scientific sense is not a legal claim, and this book makes no legal claims.

Repository histories contribute to the documentation of development in at least three ways. They record the sequence of changes in a body of work. They provide public timestamps, when the repository is public and the timestamps are corroborated. They support citations to released versions. Each contribution has limits that follow from the results of the previous chapters.

34.2 Evidence of prior publication and conclusive proof

The distinction to be drawn is between evidence that something was publicly available at a time and proof that it was first.

Definition 34.1 (Disclosure record)**.** A *disclosure record* for content x is a triple $(x, T, \mathcal{A})$ in which T is a time and $\mathcal{A}$ is an anchoring record satisfying the hypotheses of Theorem 32.4 for the identifier $H(x)$, together with evidence that x itself was accessible to the public at or after T (a public repository page, an archived copy, a published preprint).

Proposition 34.2 (What a disclosure record establishes)**.** *A disclosure record establishes an upper bound T on the time at which the content x was in the possession of the submitter and the fact that x was publicly accessible. It does not establish (i) that the submitter authored x, (ii) that no earlier disclosure of x or of equivalent content exists, or (iii) that x is correct or novel. Consequently a disclosure record can be refuted by an earlier disclosure record for the same or equivalent content, but cannot, by itself, establish that no such earlier record exists.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By Theorem 32.4 the record gives an upper bound on the time of possession. The statements (i) through (iii) are not implied by the data of the record: a copied artifact would produce an identical record, an earlier equivalent disclosure is consistent with the later record, and the record contains no assertion about truth or novelty. The refutation claim follows because an earlier verifiable record places the possession earlier. The impossibility of

establishing the nonexistence of earlier records is the general logical difficulty of proving a universal negative from a finite set of observations. □

The proposition reflects a general asymmetry: priority can be challenged by finding something earlier but cannot be confirmed in a final sense. This is a feature of the institution, not of the technology. A repository history with corroborating anchors strengthens a claim of early disclosure and cannot make it conclusive.

34.3 Public lineage as a research practice

A practice that responds to these considerations is the maintenance of a public research lineage: a repository or collection of repositories in which work is published as it is produced, with history preserved, with released versions identified by persistent identifiers, and with the identifiers anchored in independent records. The practice has costs. Work is exposed in unfinished form and ideas can be taken. It has benefits as well: the record is a time-ordered account of development that can be examined by anyone, errors are corrected openly, and the trail of dependencies among works is visible. The record is not a substitute for peer evaluation, and its credibility depends on corroboration. It is an additional instrument for documenting what was available when.

A hazard specific to the practice is the rewriting of history. Squashing or rebasing public history (Theorem 25.4, Proposition 25.5) replaces identifiers and discards intermediate states, which removes the very evidence that a lineage is supposed to supply. A repository that is rebased after anchors have been recorded retains the anchors in external records, but the anchored identifiers may no longer resolve to commits in the repository, and a reader cannot confirm that the current history descends from the anchored state. The practice therefore asks for a discipline: record anchors for released states, keep the anchored objects reachable by tags that are not rewritten, and treat rewriting as an act with consequences for the evidentiary value of the record.

34.4 Result of Part VII

The part has argued that the scientific publication can be treated as a versioned dependency structure, in which the paper is one artifact among several. It has formalized the experiment (Definition 31.1), established a closure criterion that separates specification from execution (Theorem 31.3), distinguished integrity, authenticity, and validity (Proposition 32.2), derived an anchoring bound for timestamps (Theorem 32.4), and shown the limits of tests and of priority evidence (Propositions 33.3 and 34.2). The repository supports the first two levels of agreement, rerunning and reproduction, and facilitates the third, replication. What it cannot supply is the independent judgment that evaluation of scientific claims requires, a matter taken up in Part VIII and in the final chapters.

The part so far has treated claims as conclusions attached to executable artifacts. Part VIII considers the content of those claims and of the documents around them, and asks what it would mean to version meaning and justification in the way Part VI versioned files. It does so through abstract interfaces for a semantic layer and a claim layer, and not through the complete specifications of any existing system.

Part VIII
Semantic and Epistemic Infrastructure

Chapter 35

The Limits of File-Centric Representation

35.1 What files, directories, commits, and diffs represent

The infrastructure examined so far represents work as files organized in directories, changes as commits, and differences as textual diffs. These choices are inherited from the operating-system and source-code settings in which they arose, and they fit that setting well: a program is a set of files, a change to a program is a change to files, and the meaning of a change is borne by the text. When the same infrastructure is applied to bodies of knowledge, such as a collection of mathematical results, a research program, or an argument, the fit degrades. The units of meaning in a body of knowledge are propositions, definitions, derivations, data, and objections, and these do not align with files. A theorem may occupy a few lines in the middle of a file; a definition may be used in a dozen files; an objection may concern a relation between two propositions neither of which is mentioned in the objection's text.

Three defects of file-centric representation can be isolated. First, the dependencies among propositions are not recorded in the structure: a theorem uses a lemma, but the file system has no representation of the use, and the only trace is the text of the proof, which a reader must interpret. Second, the identity of a proposition is tied to its location: moving a definition from one file to another is a deletion and an addition as far as the diff is concerned. Third, the relations that give a proposition its standing, such as the evidence for it and the objections to it, are outside the files altogether.

Proposition 35.1 (Diffs cannot distinguish a move from a deletion and an addition)**.** *Let a text file contain a block B at position p and let a change relocate B to position p'. The line-oriented difference between the two versions of the file is the same as for a change that deletes B at p and independently adds an identical block at p'. When B moves between files, the same holds for the pair of file-level differences.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. A difference is a function of the two texts (or the two trees), and the two operations described yield the same pair of texts. Hence they yield the same difference, and no function of the difference alone distinguishes them. □

Rename and move detection in version-control tools are heuristics applied after the fact, based on similarity of content. They succeed often and fail when the moved block is modified, when it is small, or when many similar blocks exist. The failures matter little for source code, where history rarely depends on the identity of a block, and matter much for a body of knowledge in which a claim's history is the record of its justification.

35.2 Co-change is not dependency

A natural attempt to recover dependencies mechanically uses the history: files that change together are presumed related. The idea has some empirical support in studies of software evolution. As a basis for inferring semantic dependency among propositions it is unreliable.

Proposition 35.2 (Co-change does not determine dependency)**.** *Let the co-change relation* CC *on propositions be defined by* p CC q *iff* p *and* q *are modified in a common commit. Neither inclusion between* CC *and the dependency relation* Dep *holds in general: there are propositions with* p Dep q *that never change together, and propositions that change together without any dependency.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let q be a lemma used by theorem p. A revision of the proof of p that does not affect q changes p alone, and a typographical correction in q changes q alone, so a history consisting of such commits contains no common commit even though p Dep q. Conversely, a global search-and-replace of a notation in a single commit modifies unrelated propositions together. □

The consequence is that a representation of knowledge needs its own structure for dependency, recorded by authors or computed from the content, and cannot rely on the structure of files and commits. The structure must also have its own identity conditions, since the identity of a proposition should survive moves and edits that do not change what it says.

35.3 Identity of content versus location of a document

The distinction can be put in terms already available. A path is a location. A content hash is an identity of bytes. Neither is an identity of a proposition. A proposition's identity is a more delicate matter: two textually different statements may express the same proposition, and two textually identical statements in different contexts may express different ones (the symbol f in two chapters). A system for knowledge versioning must therefore choose its identity conditions deliberately. Three options are available. Identify by byte content, which is exact and brittle: any edit creates a new object. Identify by assigned identifier, which is stable under edits and requires an authority to assign and a convention about when an edited object is still the same. Identify by a combination: a stable identifier for the proposition, with versions identified by content. The third is the usual compromise and corresponds to the practice of giving a work a stable name and each edition a hash or number.

Definition 35.3 (Atom, version, and anchor)**.** An *atom* is a unit of content with a stable identifier $a \in \mathsf{AtomId}$. A *version* of an atom is a pair (a, h) with $h = H(u)$ for the content u of that version. An *anchor* is a reference from another object to an atom, which may be to the stable identifier (referring to the atom as it evolves) or to a version (referring to particular content).

The distinction between the two kinds of anchor is of central importance for later chapters. An anchor to the stable identifier follows the atom through its revisions, so that an objection continues to attach to a claim as the claim is rephrased. An anchor to a version is immune to later revision, so that an objection continues to attach to the claim as it was when the objection was raised. Both are needed, and a system that offers only one cannot represent the difference between objecting to a thing and objecting to a version of it.

Chapter 36

Typed Semantic Hyperstructures

36.1 Semantic Structure and Preservation

Polyxan provides a framework for semantic hyperstructures, including their organization, reference relations, and transformations. The preservation properties of such structures can be investigated through an abstract interface based on reference integrity, well-founded expansion, scope, and typed relations. These properties establish conditions under which semantic transformations preserve the interpretation of structured content.

The formal results apply to structures satisfying the stated interface assumptions. Their applicability to the independent Polyxan specification depends on the correspondence between its definitions and the abstract constructions used here. The outstanding correspondence obligations are identified in Appendix E.

36.2 Content atoms, references, and transclusion

A semantic layer organizes content into atoms (Definition 35.3) connected by typed relations. Three features are of present interest. First, references between atoms are explicit objects and not text patterns: a statement that one atom uses another is a relation in the structure. Second, references are bidirectional in the sense that the structure supports the query *what refers to this atom*, as well as *what does this atom refer to*. Third, transclusion is supported: one atom can include the content of another by reference, so that a document is assembled from atoms and changes to an atom appear wherever it is included.

Definition 36.1 (Typed hypergraph of atoms)**.** A *typed hypergraph of atoms* is a tuple $\mathcal{H} = (\mathsf{At}, \mathsf{Ed}, \mathrm{type}, \mathrm{roles})$ in which At is a set of atoms, Ed a set of hyperedges, $\mathrm{type} : \mathsf{Ed} \to T_{\mathrm{rel}}$ assigns a relation type to each hyperedge, and roles assigns to each hyperedge a function from a set of role names to atoms, so that a hyperedge relates several atoms in distinguished roles. The graph is *well typed* if each hyperedge's roles match the signature of its type.

Hyperedges are used because relations among propositions are often not binary: a derivation relates several premises to one conclusion; a dataset, a method, and a result are related jointly by a computation. A binary graph can encode them by introducing intermediate nodes, at the price of making the intermediate node an atom whose status must be managed.

36.3 Transclusion and expansion

Transclusion makes a document a function of the structure. The rendered content of a document is the result of expanding its references.

Definition 36.2 (Reference terms and expansion)**.** Let *terms* over a set Lit of literal content and a set Nm of names be defined by $t ::= \ell \mid \mathrm{ref}(n) \mid \mathrm{seq}(t_1, \dots, t_k)$ with $\ell \in \mathsf{Lit}$, $n \in \mathsf{Nm}$. An *environment* is a partial function $\mathcal{E} : \mathsf{Nm} \rightharpoonup \mathsf{Terms}$. The *reference graph* of $\mathcal{E}$ has an edge $n \to m$ whenever $\mathrm{ref}(m)$ occurs in $\mathcal{E}(n)$. The *expansion* $\exp_{\mathcal{E}}$ is defined by $\exp(\ell) = \ell$, $\exp(\mathrm{seq}(t_i)) = \mathrm{seq}(\exp(t_i))$, and $\exp(\mathrm{ref}(n)) = \exp(\mathcal{E}(n))$, provided the right-hand sides are defined.

Proposition 36.3 (Expansion terminates uniquely iff the reference graph is well founded)**.** *Let $\mathcal{E}$ have finite domain and let every name occurring in a term of $\mathcal{E}$ lie in the domain. Then $\exp_{\mathcal{E}}(t)$ is defined for every term t over the domain iff the reference graph of $\mathcal{E}$ is acyclic, and in that case the expansion is unique and contains no references.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. If the graph is acyclic, define the height of a name as the length of the longest path from it. Expansion of a name of height 0 is its literal content (its term has no references); expansion of a name of height h uses only names of height less than h and is defined by induction. Uniqueness is clear since each clause determines the value. If the graph has a cycle $n_1 \to \cdots \to n_j \to n_1$, expanding $\mathrm{ref}(n_1)$ requires expanding $\mathrm{ref}(n_1)$ again after finitely many steps, so the recursive definition does not terminate; the expansion would be an infinite term. Hence it is undefined as a finite object. □

The proposition has a practical reading. A transclusion system must reject, or treat specially, cyclic inclusion. It also says that the rendered content of a document depends on the whole environment reachable from it, which gives another instance of the closure property (Definition 31.2): the closure of a document under references is the set of atoms its rendering reads.

36.4 Semantic embeddings and the limits of similarity

Representations of content as vectors in a high-dimensional space, learned so that semantically similar items lie near each other, are useful for retrieval and for suggesting relations. They are not identifiers. Two properties preclude it. First, similarity is approximate: distinct atoms can have identical embeddings, and an atom's embedding may change when the embedding model changes. Second, nearness is not a relation of the kind that authors assert: a pair of atoms may be close in the embedding space because they are about the same topic or because one is the negation of the other.

Proposition 36.4 (Embeddings do not identify)**.** *Let atoms be finite strings over a finite alphabet and let ε map them to vectors representable in finite precision. Then (i) ε is not injective on the set of all finite strings. (ii) If ε is L-Lipschitz, with respect to the ℓ_1 norm on the normalized word-frequency vector and some norm on the output, then a text of n words and the text obtained by replacing one word (for instance by inserting a negation in place of a word) have embeddings at distance at most $2L/n$. Nearness in the embedding space therefore does not determine the relations asserted by authors.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. (i) Representable vectors form a finite set, while finite strings are countably infinite. (ii) The normalized frequency vector of a text of n words has ℓ_1 norm 1, and replacing one word moves $1/n$ of mass from one coordinate to another, changing the vector by at most $2/n$ in ℓ_1. Lipschitz continuity bounds the change of the embedding by $2L/n$. □

The role of embeddings in the architecture is therefore auxiliary: they propose candidate relations for human or algorithmic assessment, and the relations that matter are asserted explicitly with types.

Chapter 37

Semantic Transformations and Preservation

37.1 Which transformations, which invariants

A semantic layer supports operations that reorganize content without intending to change its meaning: replacing a repeated passage by a reference to a single atom, moving a definition to a place from which it is visible to all its uses, normalizing the presentation of a derivation. The Polyxan corpus names families of such operations, among them Extraction, Hoisting, Curvature Reduction, and a Polycompiler that composes them. The definitions of these operations belong to that corpus. A meaning-preserving reorganization must satisfy explicit preservation conditions, states the preservation theorem for the two operations whose abstract structural content is determined by the notions of reference and scope, namely a replacement of repeated content by a reference and a relocation of a definition to an enclosing scope, and sets out the verification obligations that the remaining operations must discharge.

The preservation condition is

$$I(T(x)) = I(x)$$

for a transformation T and an invariant I that must be specified. The research task is to determine which invariants are meaningful and which transformations preserve them. A transformation that preserves no stated invariant is not meaning-preserving in any checkable sense, and an invariant that every transformation preserves is vacuous.

Definition 37.1 (Candidate invariants)**.** For a typed hypergraph with an environment (Definition 36.2), consider the following invariants. I_1, *expansion*: the map $t \mapsto \exp_{\mathcal{E}}(t)$ on terms of the pre-transformation graph, compared through a name correspondence. I_2, *reference integrity*: every reference resolves to an atom. I_3, *acyclicity*: the reference graph is acyclic. I_4, *relation structure*: the multiset of typed relations among exposed atoms, up to renaming of internal atoms introduced by the transformation. I_5, *sharing*: the number of distinct atoms that store a given piece of content.

The first four are intended to be preserved. The fifth is deliberately changed by extraction, whose purpose is to increase sharing. The point of listing it is that a notion of correctness that insisted on preserving every property would forbid the transformations it was meant to describe.

37.2 Extraction

Extraction replaces occurrences of a repeated piece of content by references to a single new atom that holds it.

Definition 37.2 (Extraction)**.** Let $\mathcal{E}$ be an environment, s a term with no free names (or whose free names resolve identically at all occurrences), and $n \notin \mathrm{dom}(\mathcal{E})$ a fresh name. The *extraction of s under n* is the environment $\mathcal{E}'$ with $\mathcal{E}'(n) = s$ and $\mathcal{E}'(m) = \mathcal{E}(m)[s := \mathrm{ref}(n)]$ for each $m \in \mathrm{dom}(\mathcal{E})$, where $[s := \mathrm{ref}(n)]$ replaces every occurrence of s as a subterm by $\mathrm{ref}(n)$.

Example 37.3 (Extraction, and a rejected cycle)**.** Let $\mathrm{Lit} = \{\ell_1, \ldots, \ell_5\}$ and let $s = \mathrm{seq}(\ell_4, \ell_5)$. Let $\mathcal{E}(a) = \mathrm{seq}(\ell_1, s, \ell_2)$ and $\mathcal{E}(b) = \mathrm{seq}(s, \ell_3)$, so that s is repeated. Extraction of s under a fresh name n gives $\mathcal{E}'(n) = s$, $\mathcal{E}'(a) = \mathrm{seq}(\ell_1, \mathrm{ref}(n), \ell_2)$, and $\mathcal{E}'(b) = \mathrm{seq}(\mathrm{ref}(n), \ell_3)$, and the expansions of a and b are the same in $\mathcal{E}$ and $\mathcal{E}'$. Now suppose an editor proposes to change $\mathcal{E}'(n)$ to $\mathrm{seq}(\ell_4, \mathrm{ref}(a))$. The reference graph acquires the edges $n \to a$ and $a \to n$, a cycle, so by Proposition 36.3 the expansion of a is no longer defined, and an interface that enforces well-foundedness rejects the change. The example is built from the interface of this chapter and is not drawn from any existing system.

Theorem 37.4 (Extraction preserves expansion, integrity, and acyclicity)**.** *Let $\mathcal{E}$ have an acyclic reference graph and resolvable references, and let $\mathcal{E}'$ be the extraction of s under a fresh n, where s contains no free names. Then for every $m \in \mathrm{dom}(\mathcal{E})$, $\mathrm{exp}_{\mathcal{E}'}(\mathcal{E}'(m)) = \mathrm{exp}_{\mathcal{E}}(\mathcal{E}(m))$. The reference graph of $\mathcal{E}'$ is acyclic and every reference in $\mathcal{E}'$ resolves.* [PROVED UNDER STATED ASSUMPTIONS] [EXPERIMENTALLY INVESTIGATED]

Proof. Since s contains no names, its expansion is the same in $\mathcal{E}$ and in $\mathcal{E}'$, so $\mathrm{exp}_{\mathcal{E}'}(\mathrm{ref}(n)) = \mathrm{exp}_{\mathcal{E}'}(s) = \mathrm{exp}_{\mathcal{E}}(s)$. Prove by structural induction on a term t of $\mathcal{E}$ that $\mathrm{exp}_{\mathcal{E}'}(t[s := \mathrm{ref}(n)]) = \mathrm{exp}_{\mathcal{E}}(t)$. If $t = s$, both sides equal the expansion of s. If t is a literal different from s or a reference $\mathrm{ref}(m)$, then $t[s := \mathrm{ref}(n)] = t$ and the claim reduces to the induction on $\mathrm{exp}_{\mathcal{E}'}(\mathcal{E}'(m)) = \mathrm{exp}_{\mathcal{E}}(\mathcal{E}(m))$, available because the reference graph is acyclic and the induction proceeds on the height of names. If $t = \mathrm{seq}(t_1, \ldots, t_k)$ then the substitution distributes over the sequence and the induction hypothesis applies to each t_i. The new name n has no outgoing references since s has no names, so it adds no cycle, and every new reference $\mathrm{ref}(n)$ resolves to s. The experiment E8 of Appendix F, run in an equivalent scoped-term formulation, checks this on 5045 random terms with repeated closed subterms and finds no violation. □

37.3 Hoisting

Hoisting moves a definition outward to an enclosing scope. Scope introduces a danger that extraction avoids: a definition moved outward may be captured by, or may capture, a name in the new scope.

Definition 37.5 (Scoped terms and hoisting)**.** Consider scoped terms $t ::= \ell \mid \mathrm{ref}(n) \mid \mathrm{seq}(t_i) \mid \text{let } n = d \text{ in } b$, with lexical scoping: the definition d is resolved in the scope surrounding the let, and b in that scope extended by n. The *hoisting* of the inner binding in $\text{let } x = d_1 \text{ in } (\text{let } y = d_2 \text{ in } b)$ is the term $\text{let } y = d_2 \text{ in } (\text{let } x = d_1 \text{ in } b)$.

Theorem 37.6 (Hoisting preservation under non-capture)**.** *Let $T = \text{let } x = d_1 \text{ in } (\text{let } y = d_2 \text{ in } b)$ and let T' be its hoisting. Suppose $x \neq y$, $y \notin \mathrm{FV}(d_1)$, and $x \notin \mathrm{FV}(d_2)$. Then T and T' have the same expansion in every environment. The side conditions are sufficient and not necessary: there are terms violating them whose expansions nevertheless coincide, and terms violating them whose expansions differ.* [PROVED UNDER STATED ASSUMPTIONS] [EXPERIMENTALLY INVESTIGATED]

Proof. Under the hypotheses, resolution of a reference inside d_1 cannot reach y (which does not occur free there), and a reference inside d_2 cannot reach x. In T, references within b to x resolve to d_1 in the outer scope and references to y resolve to d_2 in the scope extended by x; since d_2 does not mention x, extending the scope by x does not alter its expansion. In T', symmetrically, references to x resolve to d_1 in the scope extended by y, and d_1 does not mention y. Since $x \neq y$, lookups of names in b give the same expansions in both. For the failure of the converse, the terms with $y \in \mathrm{FV}(d_1)$ include cases where the outer binding of y is also defined to equal d_2's expansion (so nothing changes), and cases where it differs (so the hoisted term captures the reference). The second kind is exhibited next. □

Countermodel 37.7 (Capture)**.** Let an outer scope bind y to the literal A. Consider $T = \mathrm{let}\ x = \mathrm{ref}(y)\ \mathrm{in}\ (\mathrm{let}\ y = \mathsf{B}\ \mathrm{in}\ \mathrm{ref}(x))$. The reference in the definition of x resolves to the outer y, so $\exp(T) = \mathsf{A}$. Hoisting gives $T' = \mathrm{let}\ y = \mathsf{B}\ \mathrm{in}\ (\mathrm{let}\ x = \mathrm{ref}(y)\ \mathrm{in}\ \mathrm{ref}(x))$, in which $\mathrm{ref}(y)$ now resolves to B, so $\exp(T') = \mathsf{B}$. The two expansions differ because the hoisted binding captured a name.

The experiment E8 confirms the structure of the theorem on random terms: among terms satisfying the side conditions, every hoisting preserved the expansion (740 of 740); among terms violating them, some changed the expansion and some did not (994 of 2943), consistent with sufficiency and the absence of necessity. This is evidence for the proved statement and a check on the experiment's own construction; it does not extend the theorem.

37.4 Operations whose preservation properties must be established from their definitions

The remaining operations of the Polyxan corpus, Curvature Reduction and the Polycompiler that composes operations, are defined there in terms that this book does not restate. The obligations that follow from the framework can nevertheless be stated. For each operation T the following are to be established or refuted from its definition: that it preserves reference integrity (I_2); that it preserves acyclicity (I_3); that it preserves the expansion of every exposed atom (I_1), or else a precisely stated weaker equivalence; and that it preserves the typed relations among exposed atoms (I_4). For a composite operation built by composing operations, the fact that each component preserves an invariant implies that the composite does, since preservation of an invariant is closed under composition.

Proposition 37.8 (Closure of preservation under composition)**.** *If T_1 and T_2 each preserve an invariant I, in the sense $I(T_i(x)) = I(x)$ whenever T_i is defined at x, then $T_2 \circ T_1$ preserves I wherever it is defined.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. $I(T_2(T_1(x))) = I(T_1(x)) = I(x)$, the first equality because $T_1(x)$ is in the domain of T_2 and T_2 preserves I, the second because T_1 preserves I. □

The proposition is the reason a compiler of transformations can be verified piecewise. It does not say that a composite preserves *more* than its components do, and the failure of a component to preserve an invariant means that the composite is not guaranteed to preserve it either.

Conjecture 37.9 (Invariant lattice)**.** *The invariants of Definition 37.1, ordered by implication over the class of well-typed environments, form a lattice in which expansion* (I_1) *implies reference integrity* (I_2) *for the pre-transformation graph, and in which no nontrivial transformation of the class considered preserves sharing* (I_5) *while reducing storage.* [CONJECTURAL]

The conjecture is stated for the purpose of organizing the verification: it asks which invariants imply which, so that fewer need be checked. The second clause is nearly a restatement of the definitions. A proof of the first clause would require a precise account of how the invariants are formalized and cannot be given at the level of generality adopted here.

Chapter 38

Adversaria and the Public Claim Graph

38.1 Claims, Grounds, and the Persistence of Justification

The preservation of scientific knowledge requires more than the retention of propositions. A proposition may remain unchanged while the evidence supporting it is superseded, the conditions under which it holds are narrowed, or objections arise that alter its standing. Conversely, a proposition once rejected may become defensible through the discovery of new evidence or the correction of an earlier argument. The history of a claim is therefore not reducible to a succession of textual revisions. It includes the changing relations among what is asserted, what is offered in support, what is contested, and what remains unresolved.

Adversaria approaches this problem through the public claim graph, in which the fundamental unit is an argued proposition rather than an isolated statement. An argued proposition comprises a claim, the grounds offered for it, the warrant connecting those grounds to the claim, the scope within which the assertion is intended to hold, the objections directed against it, and its epistemic status. These components are distinguishable because they may change independently. New grounds may strengthen an existing claim without altering its wording; a revised warrant may change the interpretation of the same evidence; a scope restriction may preserve a qualified proposition while withdrawing a broader assertion; and an objection may remain part of the historical record even after it has been answered.

The governing distinction is between recording what has been asserted and determining what ought to be accepted. A public claim graph does not acquire epistemic authority merely by accumulating propositions or recording agreement among participants. Its purpose is to preserve the relations through which assertions can be examined, challenged, qualified, and revised. What matters is not simply whether a community currently accepts a proposition, but what grounds that acceptance, which objections remain operative, and what further evidence or argument would justify a change of status.

This distinction extends the account of continuation developed in the preceding parts. Just as a repository preserves more than the current contents of its files, an epistemic record must preserve more than the current wording or classification of its claims. The ability to reconstruct an earlier judgment, identify the grounds on which it depended, and determine how subsequent revisions altered its standing is a condition of independent evaluation. An append-only record of argumentative events, together with explicit rules for interpreting those events, provides one possible architecture for such continuity.

The formal treatment below isolates the relations required to investigate these properties. It employs an abstract claim-graph interface whose correspondence with the independent Adversaria specification is recorded in Appendix E. Results established for this interface apply to a concrete implementation insofar as that implementation satisfies the relevant assumptions. Where a particular method of assigning epistemic status is required, established formal argumentation semantics provide an illustrative construction; no identification of that

construction with Adversaria's own status rules is presumed.

38.2 Preserving a proposition and preserving the history of its justification

A proposition is a statement. Its justification is a structure: the grounds, the warrant, the scope, the objections and their answers. A system that stores only the statement preserves the proposition; a system that stores the structure preserves the justification. The two come apart in practice. An encyclopedia entry stores the proposition and, at best, links to sources. A journal article stores a proposition with an argument for it but does not store the objections later raised, so the record of how the claim fared is held elsewhere. The functional division of the older tradition of argument analysis into claim, data, warrant, backing, qualifier, and rebuttal [36] is a division of the justification into components that a reader can examine separately, and the point of a claim graph is to make those components first-class, linkable, and permanently recorded.

Definition 38.1 (Argued proposition: an abstract interface)**.** An *argued proposition* is a tuple $(c, G, w, \varsigma, O, \sigma)$ in which c is the claim, G a set of grounds (references to evidence, which may be atoms of the semantic layer), w the warrant, ς the scope, O the set of objections, and σ the status. A *claim graph* is a typed hypergraph (Definition 36.1) whose atoms include argued propositions, and whose hyperedges include the relations *supports*, *objects-to*, *answers* (an objection), and *scopes*.

38.3 An append-only record

The proposed discipline for such a graph is that it be built from an append-only sequence of events. An event adds a claim, adds grounds, states a warrant, restricts a scope, raises an objection, answers an objection, withdraws a claim, or records a status assertion. Events carry the identity of the agent who made them and an indication of when. The state of the claim graph at any point is obtained by replaying the events up to that point.

Definition 38.2 (Event log and replay)**.** An *event log* is a finite sequence $\mathbf{e} = (e_1, \dots, e_n)$ of events drawn from a set Ev. A *state space* St has a distinguished initial state s_0 and an update function $u : \mathsf{St} \times \mathsf{Ev} \to \mathsf{St}$. The *replay* of the first k events is $\mathrm{Rp}_k(\mathbf{e}) = u(\cdots u(u(s_0, e_1), e_2) \cdots, e_k)$. A *status function* $\mathrm{stand} : \mathsf{St} \to \mathsf{Status}^{\mathsf{Claim}}$ assigns to each state a status for each claim.

The log is the source of truth and the state is derived. This is the same architecture as the Git object graph, in which the history is the primary object and a working tree is derived from it, and it is chosen for the same reason: it makes the record immutable, auditable, and replayable.

38.4 An example status function

To fix ideas it is useful to have a concrete status function, even if only as an example. The abstract argumentation frameworks of Dung give one, in which claims are arguments, objections are attacks, and the status of a claim is determined by a labelling [11]. In the grounded

labelling, a claim is *in* if every one of its attackers is *out*; a claim is *out* if at least one of its attackers is *in*; and all remaining claims are *undecided*. The labelling is the least fixed point of a monotone operator, computed by iteration, and is unique.

Proposition 38.3 (Properties of the grounded labelling)**.** *For a finite attack relation, the grounded labelling exists, is unique, and is invariant under isomorphism of the claim graph: if ϕ is a bijection between two claim graphs preserving attacks, the labelling of $\phi(c)$ equals that of c. It is computed by iterating the labelling rules from the empty assignment until no change occurs, in at most as many rounds as there are claims.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let Φ map a pair (set I of claims labelled in, set Ω labelled out) to the pair obtained by adding to I every claim all of whose attackers are in Ω and adding to Ω every claim with an attacker in I. Φ is monotone for the componentwise inclusion order on pairs. The state space is a finite complete lattice, so by the Knaster–Tarski theorem [34] Φ has a least fixed point, reached by iterating from the empty pair; each round that changes the pair adds at least one claim, so there are at most as many rounds as claims. The definition refers only to the attack relation and not to the identities of claims, so a bijection preserving attacks commutes with Φ and with its iteration, giving invariance. Uniqueness is the uniqueness of the least fixed point. □

This choice of example is deliberate in being modest. It makes no use of numerical scores. The status is one of three values determined by the structure of attacks. It is not proposed as the correct status function for a public claim graph; indeed the question of what the right status function is, or whether a scalar summary is appropriate at all, is a question for the specification and is not settled here.

38.5 What the example shows about structure

Two properties of the example will be used. The first is isomorphism invariance: the status of a claim depends on its position in the structure and not on any intrinsic property of the claim text or its identifier. This is the property that makes it possible to migrate a claim graph, rename its identifiers, or merge two graphs along a bijection without altering any status. The second is non-monotonicity. Adding an objection to a claim can change its status from in to out; adding an objection to the objector can restore the original claim to in. The status of a claim is therefore not a quantity that accumulates. It is a function of the whole current structure, and the history of its changes is part of the record of what has been argued.

Chapter 39

Defeat, Revision, and Epistemic Continuity

39.1 What Revision Requires of a Record

Scientific claims are subject to revision as evidence accumulates, errors are discovered, warrants are challenged, and the scope of established conclusions is reconsidered. Such revisions need not invalidate everything that preceded them. A claim may be restricted to a narrower domain, an objection may be answered without being withdrawn, and a previously accepted conclusion may lose its standing without losing its historical significance. The problem is therefore not simply to preserve successive versions of a proposition, but to retain the evidentiary and argumentative relations through which its standing has changed.

A record that preserves only the latest judgment conceals the grounds on which earlier judgments were made. A record that preserves earlier judgments without accommodating subsequent correction mistakes historical fidelity for epistemic stability. Neither is adequate for the continuation of scientific inquiry. The relevant requirement is that revision alter the current interpretation of a claim without retrospectively altering the evidence, objections, and decisions from which that interpretation was derived.

An append-only record addresses the historical component of this requirement by representing corrections, qualifications, and withdrawals as new events rather than replacements of earlier entries. Its epistemic adequacy, however, depends on the rules by which those events are interpreted. The preservation of an event sequence does not itself establish that a revised status is justified, that competing objections have been resolved, or that the same history will yield the same judgment under different evaluation rules. These questions require an explicit account of claim identity, event ordering, revision semantics, and the relation between historical records and their evaluated states.

Theorem 39.1 (Epistemic revision theorem)**.** *Let* $\mathbf{e} = (e_1, \ldots, e_n)$ *be an event log with update function* u *and status function* stand *as in Definition 38.2, and let* $\mathbf{e}' = (e_1, \ldots, e_n, e_{n+1}, \ldots, e_m)$ *be an extension of* $\mathbf{e}$*. Then:* (*i*) (*Prefix stability*) *For every* $k \leq n$, $\mathrm{Rp}_k(\mathbf{e}') = \mathrm{Rp}_k(\mathbf{e})$*, hence the status of every claim as of time* k *is the same whether computed from* $\mathbf{e}$ *or* $\mathbf{e}'$*;* (*ii*) (*Representability of revision*) *The status of a claim at time* m *may differ from its status at time* n*; both are recoverable from* $\mathbf{e}'$ *as* $\mathrm{stand}(\mathrm{Rp}_n(\mathbf{e}'))$ *and* $\mathrm{stand}(\mathrm{Rp}_m(\mathbf{e}'))$*;* (*iii*) (*Determinism*) *Two parties who hold the same log and the same pair* (u, stand) *compute the same statuses at every time.* [PROVED UNDER STATED ASSUMPTIONS] [EXPERIMENTALLY INVESTIGATED]

Proof. (i) By definition Rp_k depends only on the first k events, which coincide. (ii) The two replays are defined by the same log, and the update function may map an event to a different status, as the example of the grounded labelling shows: with e_{n+1} an objection that attacks an undefeated claim, its status changes while all previous events are retained. (iii) The replay is

a composition of the same functions applied to the same arguments. □

The experiment E7 exercises the statements on 2000 random event logs of length 25, comparing the status maps computed at every prefix before and after the log is extended, and finds no violation of prefix stability; it also exhibits the reinstatement sequence (a claim that is in, then out upon an objection, then in again when the objector is itself objected to), observed in 1520 of the 2000 logs as a change of status at least twice for some claim. The experiment confirms the behavior of an implementation; the theorem is a statement about the abstract definitions.

39.2 Rules are part of the record

Part (iii) of the theorem carries a hypothesis that is easily missed: that the two parties hold the same pair (u, stand). A change in the update function or the status function changes the replay of the same log.

Proposition 39.2 (Replay is relative to the rule version)**.** *There exist an event log* **e** *and two status functions* $\mathrm{stand}_1, \mathrm{stand}_2$ *that disagree on the status of some claim at some time. Consequently an append-only record supports reproducible statuses only if the status function in force is itself recorded as part of the log, as a versioned object whose changes are events.* [PROVED UNDER STATED ASSUMPTIONS] [EXPERIMENTALLY INVESTIGATED]

Proof. Take the log (claim a, claim b, objection b attacks a, claim c, objection c attacks b). Under the grounded labelling, a is in at the end (reinstated by c's attack on b). Under a rule that labels a claim out whenever it has any attacker, a is out. The two status functions disagree on a. □

The experiment reported in Appendix F found that the grounded rule and the any-attacker rule disagreed on 1626 of 2000 random logs, which shows that disagreement between plausible rules is the norm and not the exception on dense random attack structures. The statistic describes random logs and says nothing about the logs that real communities would produce.

Example 39.3 (An objection, a scope restriction, and a revision)**.** Consider the log in which claim a asserts that a method works on a class of inputs, and objection b attacks a by exhibiting an input outside a stated subclass on which the method fails. Under the grounded labelling of the example status function, b is in and a is out. The author then records a scope restriction: a new claim a', linked to a as its revision, asserts the result for the subclass only. The objection b concerns an input outside the subclass and so does not attack a'. After the event, a' is in, b is in, and a remains out but present in the record, together with the link that explains why a' replaced it. A third party reading the log can see what was claimed, what was objected to, and what is now asserted, and can see that the claim was narrowed and not withdrawn. The example is built on the abstract status function of this chapter and says nothing about how any particular system would represent a scope restriction.

39.3 Retraction is an event, not a deletion

A claim that is later judged mistaken is not removed from the record. Its withdrawal is recorded as an event, and the replay excludes it from the live graph from that point onward

while retaining it in the log. This is the right treatment for two reasons. First, others may have relied on the claim, and the record of the reliance, including the citations that pointed to it, should not lose its referent. Second, the history of what was claimed and when is evidence for the study of the community's reasoning, and a rewritten history destroys it.

An append-only record has a limit that should be stated plainly. It can record that a judgment was erroneous and corrected. It cannot remove the effects that the earlier judgment had in the world while it stood, such as decisions taken on its basis, and it cannot guarantee that readers will consult the later correction. The record preserves what happened; consequences have their own course.

Proposition 39.4 (Append-only logs cannot reorder past events)**.** *If a log is append-only, there is no operation on the log that changes the relative order of two events already recorded, and consequently no way to represent a claim that an earlier event* should have been *recorded after a later one other than by a new event stating the claim.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By definition an append-only log admits only the operation of adding events at the end. Such an operation does not alter the positions of existing events. A statement about ordering must therefore be an additional event. □

39.4 Defeat of different kinds

The notion of defeat has several varieties that a record should distinguish. An objection may *rebut* a claim, asserting that it is false. It may *undercut* the warrant, asserting that even if the grounds hold they do not support the claim. It may *restrict the scope*, accepting the claim within a narrower range. It may *challenge the grounds*, asserting that the evidence is defective. It may *withdraw* the claim at the author's initiative. Each has a different effect on what a reader may rely on and should be represented as a different kind of event or relation. A system that records only the undifferentiated fact of attack loses the information that distinguishes a scope restriction, which leaves most of the claim intact, from a rebuttal, which does not.

Definition 39.5 (Kinds of defeat)**.** For an argued proposition $(c, G, w, \varsigma, O, \sigma)$, an objection is of one of the following kinds: *rebuttal* (attacking c directly), *undercutting* (attacking w), *ground challenge* (attacking an element of G), or *scope restriction* (replacing ς by a proper subset). A *withdrawal* is an event by the proposer retracting c.

The kinds correspond to different positions in the structure of the argued proposition, and so to different edges in the typed hypergraph. A status function that distinguishes them can say, for example, that a claim whose scope has been restricted retains standing within the narrower scope. Whether a particular status function should do so is a design decision for the specification. The interface requires only that the distinctions be recordable.

Chapter 40

The Semantic-Epistemic Interface

40.1 Two layers with different jobs

The semantic layer of Part VIII's earlier chapters manipulates content: it extracts repeated passages, relocates definitions, assembles documents by transclusion, and reorganizes without intending to change meaning. The epistemic layer records claims, grounds, warrants, scopes, and objections, and computes standing. The two layers have different jobs and different standards of correctness. A content transformation is correct when it preserves specified invariants. An epistemic judgment is correct, to the extent that anything is, when it is justified by evidence and argument. The first can be verified mechanically; the second cannot.

The design principle of the interface is that the layers be kept separate: content transformation must not adjudicate, and adjudication must not rewrite content. A transformation that altered the standing of a claim would be doing the work of an adjudicator without the means; an adjudication that altered the content of an atom would corrupt the object that others reference.

Principle 40.1 (Separation of transformation and adjudication)**.** No content transformation may change the standing of any claim except through events recorded in the epistemic layer, and no epistemic event may change the content of any atom. The only relation between the layers is the anchoring of epistemic objects to atoms and versions of atoms. [DESIGN ARGUMENT]

40.2 A candidate architecture

A typed hypergraph with node types for both content atoms and argued propositions makes the interface a matter of typed edges. Anchors (Definition 35.3) connect epistemic objects (grounds, scope descriptions, objections that cite specific passages) to atoms. A claim may itself be an atom, so that its text is subject to transformation, and its grounds may be atoms such as datasets, derivations, and releases. The semantic layer's transformations act on the atoms; the epistemic layer's events act on the claims and relations.

An anchor that is part of an objection or a ground is a reference of the kind studied in Part VI, and it inherits the closure and integrity requirements. If a transformation changes or removes an atom that an anchor refers to, the anchor either dangles or comes to refer to something else. The interface must therefore specify what a transformation owes to the anchors that point to the atoms it affects.

Definition 40.2 (Translation record)**.** A *translation record* for a transformation T is a partial function τ_T from anchors (to atoms or versions of atoms) in the pre-transformation graph to anchors in the post-transformation graph, recorded as an event in the log. A transformation is *anchor-preserving* if it comes with a translation record that is defined on every anchor that points to an atom changed or removed by the transformation, and that maps each such anchor

to an anchor whose target has the same expansion (or an explicitly recorded relation to it, such as a stated refinement).

40.3 Soundness of the interface

The central result is that standing is invariant under anchor-preserving transformations, because the status function depends only on the structure of the claim graph, and an anchor-preserving transformation preserves that structure up to isomorphism.

Theorem 40.3 (Interface soundness)**.** *Let $\mathcal{H}$ be a claim graph over a semantic layer with environment $\mathcal{E}$, and let T be an anchor-preserving transformation preserving expansion (I_1) and relation structure (I_4), with translation record τ_T. Suppose the status function is invariant under isomorphism of claim graphs (as in Proposition 38.3) and depends on the semantic layer only through anchored expansions. Then (i) every anchor of $\mathcal{H}$ resolves in $T(\mathcal{H})$ after translation; (ii) the claim graph $T(\mathcal{H})$ obtained by replacing anchors via τ_T is isomorphic to $\mathcal{H}$ as a claim graph; and (iii) for every claim c,* $\mathrm{stand}(T(\mathcal{H}))(c) = \mathrm{stand}(\mathcal{H})(c)$. [PROVED UNDER STATED ASSUMPTIONS] [EXPERIMENTALLY INVESTIGATED]

Proof. (i) By anchor preservation, τ_T is defined on every affected anchor, and unaffected anchors are untouched, so all anchors resolve in the post-transformation graph. (ii) The claim graph consists of epistemic objects and their relations to each other and to atoms. The relations among epistemic objects are unchanged by T, which acts only on atoms. The relations to atoms are mapped by τ_T to anchors with the same expansion, which, by hypothesis on how the status function depends on the semantic layer, makes them equivalent for status. The identity on epistemic objects together with τ_T on anchors defines the isomorphism. (iii) follows from (ii) and the isomorphism invariance of the status function. □

The compound hypothesis on the status function is stronger than it may appear. It excludes status functions that depend on the identifiers of atoms or anchors, for example one that counts a claim as supported only if its ground has a particular identifier. Such functions are unnatural, but a system whose status rule is of that kind would not satisfy the theorem, and the correspondence obligations of Appendix E include checking that the actual rule does not depend on identifiers.

The experiment E8 supplies a check on the semantic half of the argument by confirming that extraction preserves expansion on random terms, and E7 checks isomorphism invariance of the example status function on 2000 random logs. The combination is a test of the two ingredients of the theorem and not of the theorem, which is a deduction from them.

40.4 When a transformation changes what a claim is about

A transformation that does not preserve expansion changes the content to which a claim refers. In that case the claim does not thereby become a claim about the new content. The claim was made about the old content, and its standing was earned by arguments about that content. The interface must therefore treat such a transformation as the creation of a new version, to which no standing is automatically transferred.

Proposition 40.4 (Non-transfer of standing)**.** *Let T change the expansion of an atom a anchored by a ground of claim c, and let $a' = T(a)$. There exist status functions satisfying all the invariance*

properties of Theorem 40.3 under which the standing of c with ground a′ differs from the standing of c with ground a. Consequently standing cannot in general be transferred along a transformation that changes expansion without a new epistemic event. [PROVED UNDER STATED ASSUMPTIONS]

Proof. Let the ground *a* support *c* by supplying a datum that satisfies the warrant, and let a' change the datum to one that violates it. Under any status function that is sensitive to whether the warrant is satisfied by the ground, which every nontrivial one is, the standing differs. The invariance properties of the theorem concern isomorphism of graphs, which the replacement of *a* by a' is not, since expansions differ. □

The consequence for practice is a rule of procedure. When content changes materially, either an author records a new version and the claims that relied on the old are re-examined, or an explicit event records that the change is regarded as not affecting a claim and gives the reason. The reason is itself a claim, open to objection.

40.5 Result of Part VIII

The part has argued that the file-centric model is inadequate to represent bodies of knowledge (Propositions 35.1 and 35.2), has given an abstract interface for a semantic layer with expansion and invariants (Theorems 37.4 and 37.6), has given an interface and an example status function for an epistemic layer (Proposition 38.3), has proved that an append-only record can represent revision without altering history provided the rules are themselves recorded (Theorem 39.1, Proposition 39.2), and has proved an interface soundness result for anchor-preserving transformations (Theorem 40.3). The status of the two specifications themselves is outside the scope of these results; the obligations that they must meet in order for the results to apply to them are listed in the appendix on cross-reference mapping.

Part IX gathers the preceding results into architectural requirements and examines how far a single person or a small group can perform the functions of a scientific institution with this infrastructure. Its conclusions depend on the semantic and claim layers only through the interfaces of this part, and they are therefore conditional on the correspondence obligations of Appendix E.

Part IX

From Personal Memory to Public Institution

Chapter 41

Requirements for a General Scientific Repository

41.1 Method: requirements from functions

The preceding parts have defined the notions needed to state what an infrastructure for scientific work must provide. Requirements follow from the institutional functions that the infrastructure is meant to support, using the dependence sets of Definition 29.2, and to separate the components that these functions depend on from conveniences inherited from the interfaces of existing services. A component is indispensable for a set of functions if its removal makes some function in the set unachievable, and a convenience otherwise.

The set of functions to be supported is taken from the preceding analysis. A repository that serves software, mathematics, experimental research, semantic hypermedia, and public argument must support at least the following: the retention of artifacts and their identification independent of location; the recording of change with attributable agents; the authorization of changes under a stated governance; the reconstruction of experiments from declared inputs; the recording of provenance with independent anchoring; the maintenance of typed relations among units of content; the recording of claims, grounds, objections, and standing with revision; the discovery of work by strangers; and the migration of the whole to a new host without loss of the above.

Definition 41.1 (Indispensable component)**.** Let Φ be a set of required institutional functions. A component κ is *indispensable for* Φ if $\kappa \in D(\varphi)$ for some $\varphi \in \Phi$. The *minimal architecture* for Φ is the set $\bigcup_{\varphi \in \Phi} D(\varphi)$.

Proposition 41.2 (Minimal architecture)**.** *A system implements all functions in Φ for all states only if it implements every component of the minimal architecture, and implementing exactly the minimal architecture is sufficient to determine the value of every function in Φ.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Necessity: if a component $\kappa \in D(\varphi)$ were omitted, Theorem 29.4(ii) shows that there are states for which φ is not determined correctly. Sufficiency: the value of each $\varphi \in \Phi$ is determined by the components in $D(\varphi)$ by definition of the dependence set, hence by their union. □

The proposition is definitional and is stated because it gives the procedure for answering the question of what is essential: enumerate the functions, compute their dependence sets, and take the union. The substantive work lies in establishing the dependence sets, which are empirical matters about the functions and the representation.

Table 41.1: Requirements for a general scientific repository, with the principal chapters in which each is analyzed.

	Requirement	Basis in the text
R1	Content-addressed storage of artifacts with identity independent of location	Theorem 16.2, Proposition 13.3
R2	Versioned history with explicit ancestry and tolerance for rewriting only under recorded rules	Theorem 25.4
R3	Identity of agents independent of any one host namespace, with key rotation	Proposition 5.3, Proposition 18.2
R4	Governance recorded as data: who may change what, with changes to the policy themselves recorded	Definition 6.2, Chapter 27
R5	Declared-input closure for computational experiments, with environment capture by content hash	Theorem 31.3, Proposition 11.5
R6	Provenance with integrity, authenticity, and external time anchoring	Proposition 32.2, Theorem 32.4
R7	Typed relations among units of content with stable identity and versioned content	Definition 35.3, Proposition 36.3
R8	Append-only epistemic record with versioned status rules and anchored grounds	Theorem 39.1, Theorem 40.3
R9	Discovery by strangers without dependence on a single index	Chapter 46
R10	Complete export and import with verification that declared institutional functions are preserved	Theorem 29.4

41.2 Indispensable functions

The inventory in Table 41.1 lists requirements, each with the component it relies on and the part of the book that supports it. The entries are stated at the level of function and not of implementation.

41.3 Conveniences inherited from existing interfaces

Many features of hosting services are conveniences by the criterion of Definition 41.1: removing them does not make any required function unachievable. Profile images, reaction icons, activity graphs, and social counters belong here. So does the particular shape of a pull request interface, since the functions it supports, proposal and review of changes, are supported by the underlying requirements R2 and R4 in any number of interfaces. The classification does not say these features are worthless. They are valuable to users and affect adoption. The point is that an independent implementation need not reproduce them to be a scientific repository, and that a design which took them as requirements would be inheriting the accidents of an interface.

The criterion also marks the converse risk: some features that look like conveniences are in fact indispensable for a function that is not obvious. The visible record of who reviewed a change is a social feature of an interface, and it is also the carrier of the evidence needed for the policy-admissibility predicate. A procedure for classifying a feature is therefore to ask which institutional functions would lose their determination if it were absent.

Chapter 42

A Layered Repository Architecture

42.1 Seven layers

The requirements of the previous chapter suggest a separation into layers, each responsible for one group of functions and each depending only on layers below it (Principle 24.7). Seven layers are proposed.

Definition 42.1 (Layered architecture)**.** The layers, from lowest to highest, are: L_1, *durable artifact storage*, holding immutable content-addressed objects; L_2, *version histories*, holding ancestry and named references; L_3, *identity*, binding agents to keys and handles with rotation and revocation; L_4, *governance*, holding policies and the log of their changes; L_5, *semantic indexing*, holding the typed hypergraph of atoms and their relations; L_6, *epistemic records*, holding the append-only claim log; and L_7, *public discovery*, holding indexes, search, and presentation. A layer L_i *depends on* L_j if its interface refers to objects of L_j in its specification.

42.2 Independence of layers

Proposition 42.2 (Independent replaceability of layers)**.** *Suppose the dependence relation among the layers is acyclic. Then the layers can be ordered so that each depends only on lower layers, and for each k the layers $L_1, \dots, L_k$ form a self-contained system: their interfaces refer to no object outside them. Consequently a layer L_k can be replaced by another implementation of the same interface, with a migration of its data, without invalidating any lower layer, and the layers above L_k remain valid provided the replacement preserves the references they hold into L_k. If the dependence relation contains a cycle, the layers on the cycle cannot be replaced independently.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Acyclicity gives a topological order. The interface of L_i refers only to objects of layers it depends on, which are lower in the order, so $L_1, \dots, L_k$ refer only to objects within themselves. Replacing L_k changes no object that a lower layer refers to, since lower layers do not refer to L_k. Layers above refer into L_k and remain valid if the replacement keeps those references resolvable. If L_i and L_j lie on a cycle, each refers to the other, so replacing one invalidates references held by the other unless both are replaced together. □

42.3 What can be implemented over Git and what cannot

The lower two layers are native to the Git object model, and a good deal of the others can be represented as content in a Git repository, so that they inherit its portability and verification. Identity records, in the form of public keys and signed statements binding them to handles, can be stored as files. Governance can be represented as policy files whose changes are commits, with the rule that changes to the policy files must satisfy the policy then in force,

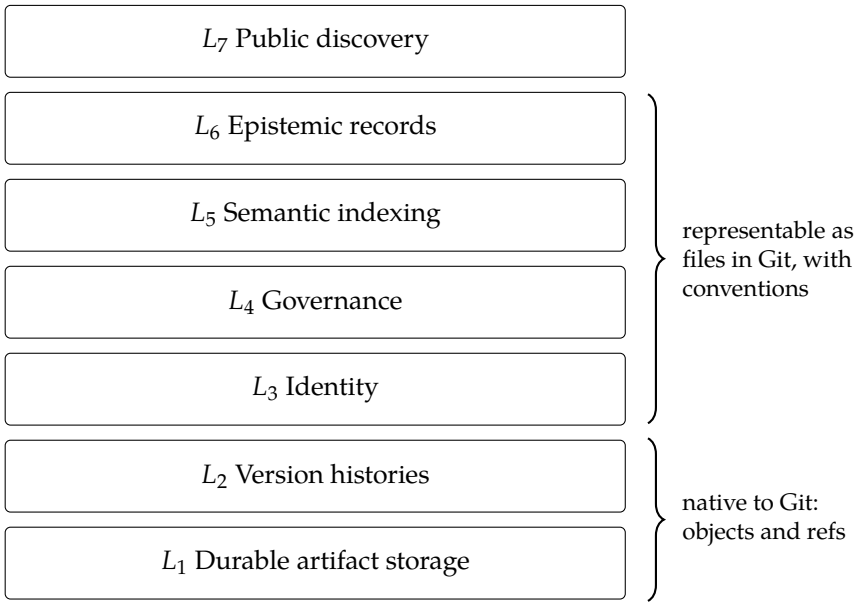


Figure 42.1: The seven layers of the proposed architecture. The lower two are native to Git; the remainder can be represented as files and logs stored in Git, with the caveats discussed in the text.

which can be checked mechanically by a hook. The semantic layer can be represented by files carrying atom identifiers and relation records. The epistemic layer can be represented by an append-only event log in which each event includes the hash of its predecessor.

Proposition 42.3 (Hash-chained logs are tamper-evident)**.** *Let an event log be stored so that event e_i includes $H(e_{i-1})$ as part of its content, with a fixed value for e_0. Assume H is collision resistant. If an event e_j is altered to $e'_j \neq e_j$, then the identifiers of all events e_i with $i > j$ computed from the altered log differ from those of the original, and any holder of the original identifier of the last event detects the alteration.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. By collision resistance, $H(e'_j) \neq H(e_j)$, so e'_{j+1}, which includes $H(e'_j)$, differs in content from e_{j+1}, and therefore $H(e'_{j+1}) \neq H(e_{j+1})$. Induction gives the claim for all later events. A holder of the original last identifier compares it with the one computed from the altered log and finds a mismatch. □

Several things cannot be implemented over Git alone. Git offers no mechanism for restricting who may push beyond what the host provides, so the enforcement of governance must be supplied by the host, by hooks, or by verifying clients; the representation of governance as data is only a necessary part of enforcement. Git has no concept of discovery, and *finding* requires indexes that are maintained by a party. Git has no mechanism for revoking a key retroactively or for establishing that a key was valid at a time, which the identity layer needs and which requires an external anchoring service. And Git’s treatment of large binary data is poor, so artifacts such as datasets and container images require separate storage with content addressing, the references to which are stored in Git.

Table 42.1: Implementation of each layer over Git and the supplements required.

	Layer	Representable in Git	Supplement needed
L_1	Artifact storage	Blobs, trees; small artifacts	External content-addressed store for large data
L_2	Histories	Commits, refs, tags	Policy on rewriting; mirrors
L_3	Identity	Key and handle records as signed files	External anchoring for key validity time
L_4	Governance	Policy files; checks in hooks and clients	Enforcement at write time by host or client
L_5	Semantic indexing	Atom and relation files	Index for query and expansion
L_6	Epistemic records	Hash-chained event files	Rules versioned in the log
L_7	Discovery	Not applicable	Independent indexes and crawlers

Chapter 43

Migration, Federation, and Independence

43.1 Institutional history is what must survive

The central question of migration is not whether source files can be copied. They can. It is whether a project can preserve its meaningful institutional history when its infrastructure changes. By Theorem 29.4, the answer is determined function by function: for each function the project relies on, either every component on which it depends is carried over faithfully, or the function is lost.

An inventory is a necessary first step. The functions of a mature project include the resolution of external citations; the attribution of contributions to persistent identities; the enforcement of the review policy; the evaluation of the standing of claims; the reconstruction of released artifacts; and the continuation of governance through changes of maintainers. For each, the dependence set can be listed, and for each component the portability can be assessed. Where a component is not portable, the options are to reconstruct it from other components, to export it into a portable representation in advance (Principle 29.6), or to accept the loss and say so.

43.2 Replication and common-cause failure

Mirroring the repository on several hosts protects against loss of any one. The protection is limited by the correlation of the hosts' failures, a structure already met in the analysis of review (Proposition 27.4).

Proposition 43.1 (Survival under replication with common-cause failure)**.** *Let r independent replicas each survive a horizon with probability p, and let a common-cause event destroy all replicas with probability β. The probability that at least one replica survives is*

$$(1-\beta)\big(1-(1-p)^{r}\big),$$

which is increasing in r and bounded above by $1-\beta$ for every r. [PROVED UNDER STATED ASSUMPTIONS]

Proof. With probability $1-\beta$ no common-cause event occurs, and the replicas survive independently, so at least one survives with probability $1-(1-p)^{r}$. With probability β none survives. The expression follows, and $1-(1-p)^{r}$ increases in r to a limit of 1 for $p>0$. □

The proposition gives the quantitative form of the advice to diversify: replicas in the same administrative domain, jurisdiction, or software stack share causes of failure, and the way to reduce β is to place replicas under different operators, in different jurisdictions, with different software. It also shows the limited value of the r-th copy once β dominates.

43.3 Governance continuity and the dead-hand problem

Governance has its own continuity problem. If the authority to change the policies of a project rests with one key and the key is lost, the policy can never be changed again. The project is stuck with its governance, which may include a permission structure designed for a circumstance that no longer exists.

Definition 43.2 (Threshold authority)**.** A *threshold authority* with parameters (k, n) is a set of n key holders such that a policy change is valid iff it is endorsed by at least k of them, $1 \le k \le n$.

Proposition 43.3 (Loss and compromise trade-off)**.** *A (k, n) threshold authority remains able to act after the loss of any $n - k$ key holders and fails to be able to act after the loss of $n - k + 1$. It remains safe against any coalition of $k - 1$ compromised holders and is unsafe against some coalition of k. The maximal number ℓ of tolerated losses and the maximal number c of tolerated compromises satisfy $\ell + c = n - 1$. Hence for fixed n, increasing tolerance of one decreases tolerance of the other.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. The authority can act iff at least k holders remain, so it tolerates $n - k$ losses and not $n - k + 1$. A coalition can force a change iff it has at least k members, so $k - 1$ compromises are tolerated and k are not. Then $\ell + c = (n - k) + (k - 1) = n - 1$. □

Secret-sharing schemes make threshold authorities implementable without a trusted dealer holding the whole secret at rest [32], and threshold signatures allow the endorsement to be verified as a single signature. The result of the proposition holds irrespective of the scheme: it is a statement about the logic of the threshold. Its meaning for institutions is that no choice of (k, n) removes the tension between the risk of paralysis and the risk of capture. A governance design chooses a point on the line and should record the choice and the reasons, as a decision in the governance layer.

43.4 Succession as data

The continuation of governance across the departure of any one person requires that succession be specified in the record. A project can record who holds which roles, the conditions under which roles transfer, the procedure for appointing successors, and the fallback in the case of inactivity. This is the content of the highest levels of Table 22.1. Recording succession as data in the repository means that the information survives migration along with the other portable components, and that an outsider can determine, from the record alone, who is currently authorized. Without it, a project with an absent maintainer is orphaned in the sense of Proposition 22.4: dependents cannot tell whether anyone is responsible.

Chapter 44

Plenumhub as a Candidate Implementation

44.1 Status and scope

This chapter is explicitly prospective. The authoritative specification of Plenumhub has not been established at the time of writing, and nothing in the chapter is a claim that an implementation exists or that any implementation satisfies the requirements of the preceding chapters. The chapter treats the design target as a set of requirements (Table 41.1) and a layered architecture (Definition 42.1), and sets out how a candidate implementation would be evaluated. Claims about an implementation remain conditional on verification.

44.2 Independence as a design requirement

The motivating requirement is independence. A system proposed as an alternative to hosted services must not depend, conceptually or operationally, on any one of them. It must not assume that its identities live in the namespace of an external provider, that its notifications are carried by an external service, that its automation runs on an external provider's infrastructure, or that its discovery depends on an external index. Each such assumption would reintroduce a component in the dependence set of some institutional function that the new system does not control (Definition 29.5).

Definition 44.1 (Independence of an implementation)**.** An implementation is *independent of a provider* $\mathfrak{p}$ with respect to a set Φ of institutional functions if, for every $\varphi \in \Phi$, the dependence set $D(\varphi)$ in that implementation contains no component controlled solely by $\mathfrak{p}$, and every external reference it issues resolves without $\mathfrak{p}$.

44.3 An acceptance plan

Verification of a candidate against the requirements proceeds through acceptance tests, each stated as a property that can be checked on a running system. Table 44.1 lists them. Many are direct generalizations of the experiments of Appendix F, which test the corresponding properties on small models; a reference test suite for an implementation would extend those experiments to the implementation's real interfaces. The status column records, as of writing, that none of them has been run against an implementation, because no implementation specification has been fixed.

The table is a list of obligations. Each could fail, and a failure is informative: the failure of R10 would show a component in some function's dependence set that the implementation

Table 44.1: Acceptance tests for a candidate implementation. Status refers to an implementation; none has yet been evaluated.

Req.	Acceptance property	Status
R1	Export and import preserve every object identifier; verify with a full object listing	Not evaluated
R2	Clone and relocation preserve ancestry; rewriting is refused or recorded by rule	Not evaluated
R3	An identity can be re-bound to a new handle and key with continuity of attribution	Not evaluated
R4	A change to a policy that violates the policy in force is rejected by a verifying client	Not evaluated
R5	A deliberately undeclared read is detected by a closure check	Not evaluated
R6	An anchored identifier can be verified by a party with no access to the host	Not evaluated
R7	Expansion is unchanged by extraction and by hoisting satisfying side conditions	Not evaluated
R8	Replay of the claim log under the recorded rules reproduces recorded statuses	Not evaluated
R9	A second independent index can be built from public data alone	Not evaluated
R10	A round trip through a second implementation preserves each declared function	Not evaluated

does not export, which is exactly the information the portability criterion is designed to bring out.

44.4 What the design cannot guarantee

A design that satisfies all of the properties still depends on matters outside the design. It depends on people operating it, on the existence of at least one host, on the willingness of others to recognize its identifiers and its record, and on legal and economic conditions. The properties are about what the technology can make possible, and the realization of possibility depends on adoption (Chapter 46). The most the chapter can responsibly state is that the architecture is consistent with the requirements derived from the analysis, that its acceptance conditions are checkable, and that the checking has not been done.

Chapter 45

The Individual as an Institutional Center

45.1 Functions and legitimacy

An individual with access to a repository system can now carry out functions that were once the work of organizations. The individual can publish work with a verifiable history, maintain an authoritative lineage of it, establish procedures for contributions, preserve a reproducible record, and invite objections in a public record. The question is what this changes and what it does not.

Definition 45.1 (Institutional functions and institutional legitimacy)**.** The *institutional functions* of a body of work are the five functions of Chapter 48: recognizing identities, authorizing transformations, preserving provenance, evaluating claims, and maintaining continuity through disagreement and change. *Institutional legitimacy* is the recognition of the body, by parties outside it, as entitled to perform those functions with consequences: accreditation, affiliation, standing in a recognized community, or legal status.

Proposition 45.2 (Functions and legitimacy are independent)**.** *All four combinations of exercising the institutional functions and possessing institutional legitimacy occur.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Both: an established university research group with a documented repository practice. Functions without legitimacy: an individual with a public repository, signed history, stated policies, and an open claim log, recognized by no accrediting body. Legitimacy without functions: a nominally accredited body that has ceased to maintain records or to review anything. Neither: an abandoned repository with no recognition. □

The proposition separates two things that discussions of independent research often run together. Performing the functions well does not confer legitimacy, and the absence of legitimacy does not prevent the functions from being performed. Legitimacy is conferred by others, through processes the individual does not control, and is not a technical property. What the technology changes is the cost of performing the functions and the verifiability of their performance. It lowers the barrier to the functions and leaves the barrier to legitimacy where it was.

45.2 Key-person dependence

An institution centered on one person has a specific vulnerability. Every function depends on the individual.

Definition 45.3 (Key-person dependence)**.** For an agent a and a set Φ of institutional functions, the *key-person dependence* of Φ on a is the fraction of functions $\varphi \in \Phi$ for which there exist states in which the removal of a's actions makes φ undeterminable. A body of work is *a-independent* if this fraction is zero.

For a one-person body of work, the dependence of nearly every function on the individual is one. The decisions of acceptance, the maintenance of the record, the authority over policy, and the continuation of the claim log all pass through the same agent. This is not a defect of the individual. It follows from the structure, and it is the reason the continuity mechanisms of Chapter 43 matter more for individuals than for groups. Documentation sufficient for continuation by a stranger (Definition 4.7), recorded succession, mirrors under different operators, and a governance authority that is not a single key are the mechanisms by which an individual's body of work becomes independent of the individual.

45.3 Public lineage as evidence

An individual who has no institutional affiliation has one resource that institutions provide only indirectly: a record. A public repository with a long history, reproducible artifacts, stated procedures, and open objections is evidence of the kind of work done. The evidence is of limited force. It documents what was done and when, within the limits of Proposition 34.2, and says little about whether the work is correct or significant. The evaluation of those matters is by others, and a record that invites it, by making objections easy to state and to anchor, is more useful than one that does not. The claim graph of Part VIII is the infrastructure for inviting evaluation, and its adoption by others is what would convert a private record into a public one in a deeper sense.

Chapter 46

Network Effects and De Facto Standards

46.1 Why technical alternatives do not displace incumbents

The foregoing chapters describe an independent architecture and argue that its components can be implemented. They do not show that such an architecture would be adopted. Infrastructure is subject to network effects: the value of using a system increases with the number of others using it, and a system with fewer users has less value to a prospective user even if it is technically superior [21]. Increasing returns to adoption can lock in a technology that is not the best available [2]. Proposition 26.7 gave a minimal model; the present chapter extends it to dynamics and to the effect of interoperability.

46.2 Critical mass

Let $x \in [0,1]$ be the fraction of a population using an alternative system, and let $F : [0,1] \to [0,1]$ be a continuous adoption response, giving the fraction who will adopt next period when the current fraction is x.

Proposition 46.1 (Critical mass)**.** *Suppose F is continuously differentiable and nondecreasing with $F(0) = 0$, $F(1) = 1$, $F'(0) < 1$, $F'(1) < 1$, and $F(x^*) > x^*$ for some $x^* \in (0,1)$. Then 0 and 1 are locally attracting fixed points of the iteration $x \mapsto F(x)$, and there is a fixed point $x_c \in (0, x^*)$ such that every initial adoption level in $(0, x_c)$ decreases monotonically to 0. In particular, adoption below the critical level x_c collapses.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Set $g(x) = F(x) - x$. Since $g(0) = 0$ and $g'(0) = F'(0) - 1 < 0$, there is $\epsilon > 0$ with $g < 0$ on $(0, \epsilon)$. The set $S = \{x \in (0, x^*] : g(x) \geq 0\}$ contains x^*, and $x_c = \inf S \geq \epsilon > 0$. By continuity $g(x_c) \geq 0$ if $x_c \in S$ and $g(x_c) \leq 0$ since $g < 0$ on $(0, x_c)$, so $g(x_c) = 0$ and x_c is a fixed point, with $g < 0$ on $(0, x_c)$. Take $x_0 \in (0, x_c)$. Then $F(x_0) < x_0$, and since F is nondecreasing and $F(0) = 0$ we have $F(x_0) \geq 0$ and $F(x_0) \in [0, x_0)$. The iterates form a nonincreasing sequence in $[0, x_0]$, which converges to a fixed point of F in $[0, x_0]$; as $g < 0$ on $(0, x_c)$ the only such fixed point is 0. The attraction of 0 and of 1 follows from $F'(0) < 1$ and $F'(1) < 1$ with $F' \geq 0$ by the standard linearization. □

The proposition formalizes the notion of critical mass. An alternative with adoption below the threshold decays; above it grows. The practical import is that the technical merits of an alternative affect the shape of F and hence the location of the threshold, and that reaching the threshold is a coordination problem and not a technical one.

46.3 Interoperability as threshold reduction

The coordination component of value can be reduced if the alternative can interoperate with the incumbent. In the model of Definition 26.6, suppose a fraction $\lambda \in [0, 1]$ of the coordination value α is independent of which platform a participant uses, because the platforms interoperate in the relevant respects (shared identifiers, import and export, cross-references). Then the effective coordination weight is $(1 - \lambda)\alpha$.

Corollary 46.2 (Interoperability removes lock-in)**.** *In the setting of Proposition 26.7, taken in the limit $n \to \infty$ so that the $O(1/n)$ terms of that proposition vanish, and with coordination weight $(1 - \lambda)\alpha$, the inferior candidate is an equilibrium of the all-adopt kind iff $(1 - \lambda)\alpha \geq q_2 - q_1$. Hence if $(1 - \lambda)\alpha < q_2 - q_1$ the only equilibrium is the adoption of the superior candidate. In particular, for $\lambda > 1 - (q_2 - q_1)/\alpha$, lock-in on the inferior candidate disappears.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. Replace α by $(1 - \lambda)\alpha$ in Proposition 26.7. The all-1 profile is an equilibrium iff $(1 - \lambda)\alpha + q_1 \geq q_2$. The stated threshold follows by solving for λ. When the inequality fails, all-1 is not an equilibrium, while all-2 remains one since $(1 - \lambda)\alpha + q_2 \geq q_1$ holds as $q_2 > q_1$. □

This is the formal justification for the portability requirement. A system that can import from and export to the incumbent, that issues identifiers resolvable by anyone, and that preserves citations lowers the effective lock-in. The argument also shows a limit: if the incumbent controls the references on which interoperability depends, interoperability can be withdrawn, and the argument weakens. For this reason the position taken in Principle 29.6 and in Proposition 41.2, that the institution's functions should depend on components under the institution's own control as far as possible, is not merely a matter of principle.

46.4 Cumulative value of established histories

A source of lock-in specific to repositories is the accumulated value of the history. A project with years of issues, reviews, releases, and citations has a stock of institutional record whose value is lost if it is not migrated. The cost of migration is the loss of the non-portable part of the record, and the dependence index of Definition 29.5 measures its size. A project may rationally stay on a platform that it does not otherwise prefer because its history is there. The remedy is to reduce the non-portable fraction in advance, by the practices of Principle 29.6, which lowers the cost of leaving and thereby also, in a market of platforms, the market power of the incumbent.

46.5 Professional signaling

A further factor is signaling. A profile on a widely used service functions as a credential: it is a public record that others can inspect, and its presence is read as evidence of competence. The value of the signal depends on its being widely read, and so again on adoption. The relevant observation is that signaling value is carried by records and not by infrastructure, and records can be made portable. A verifiable record of work, in the form of a signed history anchored in independent records, can be presented independently of the service that hosted

it. To the extent that readers learn to evaluate such records, the dependence of signaling on any one service declines.

Chapter 47

The Limits of Repository-Based Science

47.1 What repositories cannot supply

The earlier chapters may have given the impression that a sufficiently good repository would suffice for science. It does not. Several institutional functions lie outside what any repository can provide, and a treatise that did not say so would confuse infrastructural possibility with institutional sufficiency.

First, *reliable peer evaluation*. A repository can record objections and evaluations. It cannot ensure that qualified people produce them, that they are independent of the author, or that they are honest. Second, *long-term preservation guarantees*. Mirrors and persistent identifiers reduce risk (Proposition 43.1) and cannot remove it, and preservation over decades depends on institutions with durable funding and mandates. Third, *legal protection*. Licenses, trademarks, and liability are matters of law, and a repository can only record the license it is given. Fourth, *funding*. Work requires resources, and repositories do not produce them. Fifth, *specialized experimental facilities*. An experiment that requires a particle accelerator or a clinical trial is not made possible by version control, and the measurement procedure M of Definition 31.1 may involve instruments that cannot be shared. Sixth, *universally accepted standards of validity*. What counts as adequate evidence varies by field, and a claim graph can record arguments and not decide which standards apply.

47.2 Independence of evaluation

The first limit deserves a formal statement because it concerns the epistemic layer directly. A claim's standing in a record depends on the support and objections attached to it. Support that comes from sources that share an origin is not independent, and the number of such supports does not measure the strength of the case.

Proposition 47.1 (Correlated support has a floor of error)**.** *Suppose a false claim receives k supporting attestations, each of which wrongly supports a false claim with probability ϵ independently, except that with probability β all attestations derive from a common erroneous source and wrongly support it together. The probability that a false claim has all k supports is $\beta + (1-\beta)\epsilon^k$, which for any k is at least β.* [PROVED UNDER STATED ASSUMPTIONS]

Proof. With probability β the common source is erroneous and all supports are present; otherwise they occur independently, each with probability ϵ. Total probability gives the expression, and the second term is nonnegative. □

This is the same structure as Proposition 27.4 and Proposition 43.1, and its application is the same. Adding more supports from the same origin does not reduce error below the floor β. The record can make the origin visible, by recording the dependence of an attestation on

its source as a relation in the graph, and so allow a reader to discount correlated support. It cannot produce independence that does not exist in the world. That requires other people who have not copied the author's assumptions.

47.3 Infrastructural possibility is not institutional sufficiency

Let Q be the set of functions required of a scientific institution and P the set of functions that a repository system can provide. The preceding list shows $Q \setminus P \neq \varnothing$. The statement is almost trivially true and is nevertheless worth stating, since the opposite assumption is common among enthusiasts of any infrastructure. The correct claim is a more modest one: a repository system can provide a necessary part of what scientific institutions require, namely a means of recording and verifying what was done, claimed, objected to, and revised. The remaining parts are the work of people and of other institutions.

Principle 47.2 (Necessity without sufficiency)**.** Repository infrastructure is plausibly necessary for open, verifiable, and correctable science. It is not sufficient, and a design that claimed sufficiency would be unfalsifiable and therefore of little use. [DESIGN ARGUMENT]

Chapter 48

From Repository to Institution

48.1 The Criterion

A repository becomes institution-like not merely by accumulating files or participants, but by establishing durable procedures for recognizing identity, authorizing transformations, preserving provenance, evaluating claims, and maintaining continuity through disagreement and change. These five functions distinguish an institutional repository from a collection of artifacts whose continued intelligibility and operation depend on the knowledge, authority, or presence of particular individuals.

Definition 48.1 (Institution-like repository)**.** A repository system is *institution-like* to degree $d \in \{0, \ldots, 5\}$ if it has documented and operative procedures for d of the following five functions. F_1, *recognizing identity*: a binding between agents and identifiers that survives changes of handle and key, with a record of the binding's history (Definition 5.2, Definition 18.1). F_2, *authorizing transformations*: a governance regime recorded as data, including the authority to change it (Definition 6.2, Definition 27.1). F_3, *preserving provenance*: integrity, authenticity, and external anchoring of its artifacts, with declared-input closure for computational claims (Proposition 32.2, Theorem 32.4, Theorem 31.3). F_4, *evaluating claims*: an append-only record of claims, grounds, objections, and standing under versioned rules (Theorem 39.1). F_5, *maintaining continuity through disagreement and change*: procedures for forks, federation, migration, and succession (Theorem 29.4, Proposition 43.3). A procedure is *durable* if it continues to operate after the removal of any single agent or of any single provider component. Two predicates are distinguished. The repository is *institution-like to degree d* when it has d documented and operative procedures that are durable in the sense just stated; this is a property of the repository and does not depend on anyone's having observed it. The repository is *verifiably institution-like to degree d* when, in addition, the available record justifies attributing durability to d procedures: a history showing that the procedure operated during the absence of a given agent, or a mechanical check, such as a verifying client rejecting a change that violates the policy (compare Table 44.1). A procedure may be durable without having been tested by an absence, and one that survived a single absence may fail under another, so verification evidences durability and does not constitute it. An outsider can assess only the verifiable degree, which is why a procedure stated in documentation and never exercised does not raise it.

The durability clause is what distinguishes an institution from a practice. A practice persists as long as its practitioner does. A procedure that survives the removal of any one participant or provider has been separated from them, which is to say, has been written down, made checkable, and placed under a governance that does not depend on one person. The notion of key-person dependence (Definition 45.3) and of the dependence index (Definition 29.5) provide the means to assess durability.

Proposition 48.2 (Institution-likeness is independent of size)**.** *For each $d \in \{0, \ldots, 5\}$ there exist repositories of arbitrarily small size that are institution-like to degree d and repositories of arbitrarily large size, in numbers of files and participants, that are institution-like to degree* 0. [PROVED UNDER STATED ASSUMPTIONS]

Proof. A repository with a small number of files can contain the policy files, key records, anchored releases, claim log, and succession records that realize d of the functions; none of these needs many files. A repository with an arbitrarily large number of files and participants can lack all such procedures, for instance if governance is informal, identities are host accounts only, nothing is anchored or closed, claims are unrecorded, and succession is unspecified. □

48.2 Summary of the argument

The argument of the book can be restated in order. Part I introduced the prediction problem at the root of all organization of artifacts and distinguished three kinds of persistence and a hierarchy of continuation classes. Part II showed that identity, rights, classification, and naming are institutional devices addressing that problem at the scale of one machine. Part III showed that scripts, documentation, dependency declarations, and conventions are the practices by which artifacts remain usable. Part IV traced how the problems were met by successive technologies until hosted repositories consolidated several functions and institutional coordination emerged as a byproduct. Part V described the progressive acquisition of capability and responsibility by an individual. Part VI supplied the mathematical core: typed graphs, lineage invariants, the impossibility of structural selection of authority, the independence of the four notions of acceptable contribution, the properties of merging, and the lineage preservation and portability results. Part VII applied the framework to computational science, with the closure criterion and the anchoring bound. Part VIII extended it to knowledge versioning with interface-first treatments of a semantic layer and a claim graph. The present part drew the architectural consequences and stated the limits.

48.3 Open problems

Several problems are left open, and some of them may have negative answers. The first is the empirical estimation of the recovery parameters of Chapter 3: the forgetting rate and the components of the record channel for real populations of artifacts and agents, which retrospective studies of archived projects could provide. The second is the characterization of minimal sufficient contexts for the classes of Definition 4.1 in specific domains, beyond the general hardness result of Proposition 4.5. The third is the precise relation between the invariants of Definition 37.1 for the actual transformations of the semantic system, which must be settled against its own definitions. The fourth is the design of status functions for a claim graph that respect the distinctions of Definition 39.5 without reducing standing to a scalar, a question belonging to the specification and not settled here. The fifth is the federation-scaling conjecture (Conjecture 28.5), which can be tested on archives of related repositories. The sixth is the measurement of correlated support in practice (Proposition 47.1): how often supposedly independent attestations share sources. The seventh is the design of external anchoring and discovery services that are themselves independent, a problem whose solution is partly technical and partly a matter of organizations. The eighth, and the most general, is

whether the five functions of Definition 48.1 are the right ones, or whether a sixth, such as resource allocation, belongs in the list.

48.4 Closing

The book began with a person saving a file, and ends with a proposal for the architecture of public institutions. The continuity between them is the claim that organizing information so that a future agent can use, evaluate, and continue it is a single problem at different scales. The solutions at each scale have the same shape: names that can be reconstructed, contexts that survive their authors, histories that cannot be silently altered, procedures for deciding what to accept, and records of why. None of them is guaranteed by technology. All of them can be made cheaper, more checkable, and more widely available, and that is the sense in which a repository, given suitable procedures and enough time, can become an institution.

Appendix A

Notation Glossary

Table A.1: Principal notation, with the place of first definition.

Symbol	Meaning	Definition
$\mathrm{Phys}, \mathrm{Ref}, \mathrm{Op}$	Physical, referential, and operational persistence	Definitions 2.1 to 2.3
$R(x, \Delta \mid m, c)$	Probability of recovery and interpretation after delay Δ	Definition 3.1
$\mathrm{Suff}(x, a, K)$	Sufficient contexts for operation class K	Definition 4.2
$\mathrm{Cont}(a', x, m, t', K)$	Continuation relation	Definition 4.7
$\mathcal{L}(N)$	Expected retrieval loss of naming policy N	Definition 8.1
$\mathrm{cl}(S)$	Closure of a set of artifacts under dependency	Definition 11.1
$\iota(\tau, u)$	Identifier of an object of type τ and content u	Definition 16.1
$\preceq$	Ancestry order on commits	Definition 25.1
$\sim$	Patch equivalence of commits	Definition 25.3
U_t	Recognition function at time t	Definition 17.1
$\mathsf{M}, \mathsf{P}_G, \mathsf{A}_G, \mathsf{C}_\sigma$	Mergeability, policy admissibility, acceptance, correctness	Definition 27.1
Γ_G	Integration decision under governance G	Definition 27.1
$\mathcal{D}_t = (K_t, C_t, R_t)$	Developmental state	Definition 22.1
$\mathcal{I} = (O, R, B, P, \mathrm{Id}, X)$	State of a repository institution	Definition 29.1
$D(\varphi)$	Dependence set of an institutional function	Definition 29.2
$\mathcal{X} = (C, D, E, \Theta, S, M)$	Computational experiment	Definition 31.1
$\mathrm{Reads}(\rho)$	Set of artifacts read by a run	Definition 31.2
$\exp_{\mathcal{E}}$	Expansion of terms in an environment	Definition 36.2
$\mathrm{Rp}_k(\mathbf{e})$	Replay of the first k events of a log	Definition 38.2
stand	Status function	Definition 38.2
τ_T	Translation record of a transformation T	Definition 40.2

Appendix B

A Chronology of Software Distribution and Version Control

The dates in Table B.1 are approximate years of first public availability or description, collected for orientation. They have not been verified against primary sources for this work and should be checked before being cited.

Table B.1: Approximate chronology of systems discussed in the text.

Year	Development
1971	Eight-inch floppy disk introduced; early file transfer protocol specified for ARPANET hosts
1975	Source Code Control System described in the literature
1978	Early dial-up bulletin board system in operation
1979	Distributed discussion network (Usenet) begins
1982	Revision Control System described
1985	Standard file transfer protocol published; Free Software Foundation founded
1986–1990	Concurrent Versions System developed, then rewritten in C and described
1991	First release of the Linux kernel
1993–1995	Debian distribution and its package format; RPM; central archive for Perl modules
1999	Open-source hosting service SourceForge launched
2000–2004	Subversion project begun and first stable release
2003	Python Package Index begun; early distributed systems appear
2005	Git and Mercurial released
2008	GitHub launched; pip released
2010	Node package registry and client released
2015	Rust 1.0 and its package registry
2016	Withdrawal of a small, widely used package disrupts builds across a package ecosystem
2017	First practical collision for the SHA-1 hash function
2018	Acquisition of GitHub by Microsoft announced and later completed

Appendix C

Comparative Models of Git and Hosting-Service Objects

The comparison in Table C.1 is generic. It describes categories found in many hosting services and is not a specification of any one of them.

Table C.1: Objects of the Git model and of typical hosting services.

Object	Layer	Identity	Carried by a clone
Blob, tree, commit, tag object	Git	Content hash	Yes
Branch, tag name	Reference	Name pointing to a hash	Yes (selected refs)
Account	Platform	Host-assigned identifier and handle	No
Issue or discussion	Platform	Host-assigned number	No
Review or approval	Platform	Host-assigned identifier	No
Pull or merge request	Platform	Host-assigned number	Partly (via refs, where exposed)
Permission, role	Platform	Host configuration	No
Release record, attached asset	Platform	Host-assigned identifier	Tag only; assets no
Pipeline definition	Mixed	File in repository plus host configuration	File yes; host configuration no
Star, follow, watch	Platform	Relation among host accounts	No

Appendix D

A Catalog of Reproducibility Failure Modes

Table D.1 organizes failure modes by the component of the experiment of Definition 31.1 and by the persistence property that fails (Chapter 2).

Table D.1: Failure modes of reproduction by component.

Comp.	Failure mode	Property failed	Mitigation
C	Code not released or released in a different state than used	Physical, referential	Tag and archive the exact state; anchor its identifier
C	Hard-coded paths or machine-specific constants	Operational	Parameterize; check on a clean machine
C	Undeclared imports satisfied by the author's machine	Operational	Closure check of reads (Theorem 31.3)
D	Data deleted or withdrawn by the source	Physical	Deposit a copy where licensing allows; record hashes
D	Data silently updated at a stable address	Referential	Content-addressed references
D	Data inaccessible for legal or privacy reasons	Referential, operational	Provide synthetic or summary data and a specification
E	Library versions drift	Operational	Lock by content hash (Proposition 11.3)
E	Dependence on a withdrawn package	Physical, referential	Mirror dependencies (Definition 14.2)
E	Hardware-dependent numerical behavior	Operational	State tolerance; record hardware
Θ	Parameters chosen by interactive exploration and not recorded	Operational	Record every run's configuration as data
S	Unseeded generators; nondeterministic parallelism	Operational	Seed streams; declare reduction order or tolerance
M	Manual post-processing, such as removal of outliers	Operational	Script all steps from raw output to reported quantity
M	Measurement not valid for the construct	Validity	Validation argued in the open record (Chapter 33)
all	Narrative description diverges from released artifacts	Operational	Generate figures and tables from the repository

Appendix E

Cross-Reference Mapping to the Polyxan and Adversaria Specifications

The semantic and epistemic constructions of Chapters 36, 37, 38, 39, and 40 are defined through abstract interfaces. Their correspondence with the independent Polyxan and Adversaria specifications requires mappings between the respective definitions and verification that the relevant structural assumptions are satisfied.

Table E.1 identifies the required correspondences and their associated verification obligations. A completed correspondence must identify the source definition, its abstract counterpart, the assumptions required for transfer, and the evidentiary status of that transfer, whether proved, experimentally tested, or conjectural. Until these correspondences are established, the formal results apply to the abstract interfaces rather than necessarily to the existing systems.

Table E.1: Obligations for applying the abstract results to the Polyxan and Adversaria specifications.

Notion in this book	Where used	Hypothesis to check against the specification
Atom with stable identifier and versioned content	Definition 35.3	Identity of an atom survives edits; versions are content-identified
Reference and bidirectional lookup	Definition 36.1	References are explicit objects; reverse lookup is defined
Transclusion and expansion	Definition 36.2, Proposition 36.3	Reference graph is acyclic, or cycles are guarded with a stated semantics
Extraction	Definition 37.2, Theorem 37.4	Replacement of repeated content by a reference to a fresh atom, with the side condition of no free names
Hoisting	Definition 37.5, Theorem 37.6	Relocation of a definition to an enclosing scope, with scope rules as in the definition or a stated variant
Curvature Reduction	Chapter 37	Obligations I_1 to I_4 to be established from the specification's definition
Polycompiler	Proposition 37.8	Composition of operations; each component's preserved invariants listed
Argued proposition	Definition 38.1	Claim, grounds, warrant, scope, objections, and status are first-class and separately addressable
Append-only log and replay	Definition 38.2, Theorem 39.1	State is derived by replay; status rules are versioned in the log
Status function	Proposition 38.3, Proposition 39.2	Isomorphism invariance; dependence on the semantic layer only through anchored expansions
Kinds of defeat	Definition 39.5	Rebuttal, undercutting, ground challenge, scope restriction, and withdrawal are distinguishable
Anchors and translation records	Definition 40.2, Theorem 40.3	Transformations affecting anchored atoms emit translation records

Appendix F

Computational Appendix: Experiments

F.1 Purpose and limits

The experiments are small models run to check that definitions and proved statements behave as intended when implemented, and to expose implementation errors in the reasoning. They do not establish universal claims. Each ran on the author's workstation environment with the standard library of Python and, for the first two, the Git version installed there (2.43.0). The scripts are delivered with the manuscript in a directory named `experiments`.

F.2 Kinds of experiment

The nine experiments differ in kind and should not be given equal evidential weight. E9 is a demonstration with real software, a symbolic algebra system, on a few small examples. E1 and E2 test the behavior of actual Git, although the platform records of E2 are emulated and not obtained from a hosting service. E6 demonstrates a real class of failure, undeclared reads, using the interpreter's audit mechanism. E4 is a demonstration with a deliberately simplified resolver. E3, E5, E7, and E8 are implementation consistency checks of mathematical models: they confirm that code implementing a stipulated model agrees with the stated formula or theorem, and they do not independently confirm that the model describes real systems. None of the nine is an empirical study of real-world behavior, such as real repository migrations, real dependency failures, or retrieval tasks with human participants; those remain open, and the retrospective study of Appendix G is a substantive future research program and not an optional extension.

F.3 E1: lineage under clone, rebase, and cherry-pick

Script: `e1_lineage.py`. *Design:* a repository with a base commit, a feature branch of k commits touching distinct files, and a later commit on the main branch; the repository is mirrored; the feature branch is rebased onto the main branch; the first feature commit is cherry-picked onto the main branch. Properties tested: identifiers preserved by mirroring; identifiers of rebased commits differ from the originals; patch identifiers (as computed by Git's stable patch-identifier mechanism) of rebased commits equal those of the originals; order within the rewritten segment is preserved; the old commits are not ancestors of the new ones; the cherry-picked commit has a new identifier and the same patch identifier. *Result:* for $k \in \{2, 3, 5, 8, 13\}$ all properties held in all five runs. This is consistent with Theorem 25.4 and Proposition 16.3. [EXPERIMENTALLY INVESTIGATED]

F.4 E2: migration of Git data and the absence of platform records

Script: `e2_migration.py`. *Design:* a repository with twenty commits, an annotated tag, and a second branch is mirrored to a new location; a separate record store emulating platform data (an issue citing a commit, a review count, a permissions entry) is kept outside the repository. Compared: the sets of objects reachable from all refs, the set of refs with their targets, the result of a strict consistency check, and whether the mirror contains any trace of the platform records. *Result:* the object sets and the refs were equal, the consistency check passed, and the mirror contained no trace of the platform records, which existed only in the side store. This illustrates Theorem 29.3(i)–(iv) and the premise of (v); it is a demonstration that a mirror does not carry such records, and the platform store here is an emulation, so nothing is inferred about any real service. [EXPERIMENTALLY INVESTIGATED]

F.5 E3: properties of the pathwise merge

Script: `e3_merge.py`. *Design:* random trees over twelve paths with four content values and absence, with random modifications of a base; 20,000 trials. Checked: symmetry of the merge; definedness and change set when change sets are disjoint; equality of the two bracketings for pairwise disjoint triples; undefinedness when some path is changed differently on both sides. *Result:* no violations; 8,455 triples with pairwise disjoint change sets were checked for associativity and 4,164 pairs with a genuine conflict were checked for undefinedness. These agree with Theorem 28.2; the model is the pathwise merge and not a line-level merge. [EXPERIMENTALLY INVESTIGATED]

F.6 E4: dependency resolution over time

Script: `e4_resolution.py`. *Design:* a toy registry at two times with requirements stated as ranges; resolution selects the highest satisfying version; a lock records version and a hash for each resolved artifact. *Result:* the resolved sets differ between the two times (versions 1.1 and 2.0 at the first time; 1.2 and 2.1 at the second), and the pinned resolution is the same at both times. This illustrates Proposition 11.3. [EXPERIMENTALLY INVESTIGATED]

F.7 E5: Monte Carlo check of the recovery model

Script: `e5_recovery.py`. *Design:* for forgetting rates $\lambda \in \{0.1, 0.5, 1.0\}$, record-channel probabilities $s \in \{0, 0.3, 0.7\}$, and delays $\Delta \in \{0, 1, 5, 20\}$, 200,000 independent trials each of the two-channel model. *Result:* the maximum absolute difference between the empirical and the analytic probability was 0.0025, consistent with sampling error at this sample size. The experiment checks the algebra of Proposition 3.2 and says nothing about whether real recovery follows the model, which is an empirical question left open in Chapter 48. [EXPERIMENTALLY INVESTIGATED]

F.8 E6: closure check by recording reads

Script: `e6_closure.py`. *Design:* a small analysis reads a declared data file and, unannounced, a configuration file. File opens are recorded through the interpreter's audit mechanism and compared with the declared set. *Result:* the recorded reads were the data file and the configuration file; the undeclared read was the configuration file; the specification was therefore not closed; changing the content of the undeclared file changed the program output from 60 to 120 with all declared inputs unchanged. This realizes the situation of Theorem 31.3(ii). Detection through file opens captures only one class of reads; clock, network, environment, and hardware reads need other instruments. [EXPERIMENTALLY INVESTIGATED]

F.9 E7: append-only claim log, replay, and reinstatement

Script: `e7_claims.py`. *Design:* a log of events (add claim, add objection, retract) with replay and the grounded labelling as an example status function. A reinstatement scenario with three claims was run by hand. Then 2,000 random logs of 25 events were generated and the following were checked: prefix replays computed while the log was being built equal the prefix replays recomputed from the completed log; the labelling is invariant under a random renaming of claims; the grounded labelling was compared with a rule that labels a claim out whenever it has any live attacker; and the number of logs in which some claim changes status at least twice was counted. *Result:* in the scenario the status of the first claim over successive events was in, in, out, out, in. Across the random logs there were no prefix violations and no isomorphism violations; the two rules disagreed on the final status of some claim in 1,626 logs; and some claim changed status at least twice in 1,520 logs. These numbers characterize random attack structures and not real discourse. They agree with Theorem 39.1, Proposition 38.3, and Proposition 39.2. [EXPERIMENTALLY INVESTIGATED]

F.10 E8: extraction and hoisting

Script: `e8_transform.py`. *Design:* random scoped terms of depth up to five with literals, references, sequences, and local bindings; expansion by lexical scoping. Extraction replaced a repeated closed subterm by a reference to a fresh binding. Hoisting swapped two nested bindings. Terms that were closed were retained for the hoisting test, and the side conditions of Theorem 37.6 were evaluated for each. *Result:* extraction preserved the expansion in all 5,045 applicable terms; hoisting under the side conditions preserved it in all 740 applicable terms; hoisting in terms violating the side conditions changed the expansion in 994 of 2,943 cases. The pattern fits the sufficiency of the side conditions and their non-necessity. [EXPERIMENTALLY INVESTIGATED]

F.11 E9: symbolic derivations and declared assumptions

Script: `e9_symbolic.py`, which requires the SymPy package (version 1.14.0 was used). *Design:* five checks on small examples of the kind discussed in Section 33.6. The simplification of $\sqrt{x^2}$ is computed for a symbol with no assumption, a real symbol, and a positive symbol. The expression $(x^2 - 1)/(x - 1)$ is cancelled and the original and the result are evaluated

at $x = 1$. The equation $\sqrt{x+2} = x$ is squared, and the candidates are tested against the original. The Jacobian of the Lotka–Volterra system is evaluated at the interior equilibrium and its eigenvalues computed. A substitution audit compares the cancelled and original expressions at random rational values with $x \neq 1$. *Result:* the simplification of $\sqrt{x^2}$ was $\sqrt{x^2}$, $|x|$, and x respectively; the cancellation returned $x + 1$, the original was undefined at $x = 1$ while the result equalled 2; squaring gave the candidates -1 and 2 and only 2 satisfied the original equation; the Jacobian at $(d/c, a/b)$ was $\begin{pmatrix} 0 & -bd/c \\ ac/b & 0 \end{pmatrix}$ with eigenvalues $\pm\sqrt{-ad}$; and the substitution audit found no failure in 993 admissible points. These confirm the claims of Proposition 33.7 on examples, and the first shows that a symbolic result depends on declared assumptions. They concern one system and version and say nothing about others. [EXPERIMENTALLY INVESTIGATED]

Appendix G

Protocol for Estimating Recovery Parameters

The recovery model of Chapter 3 contains parameters that could be estimated from retrospective studies of real projects. A protocol follows. A sample of archived projects is drawn from a repository archive, stratified by field, size, and age. For each project, a set of artifacts is chosen at random from its history: scripts, notebooks, data files, and configuration. For each artifact, a retrieval task is posed to a participant who did not write it, and, where available, to the author: given a description of the artifact’s purpose, find it in the project and use it for a specified operation drawn from the classes of Definition 4.1. Outcomes recorded are the time to locate, whether the correct artifact was selected, whether the operation succeeded within a fixed time, and the metadata present at the time of creation (naming style, presence of comments, documentation, and tests). The delay is the age of the artifact. Fitting the two-channel model to the author’s results estimates the forgetting rate λ from the dependence on delay, and fitting to the stranger’s results, for whom the memory channel is absent, estimates the record channel s directly as a function of the recorded metadata. The comparison between author and stranger isolates the effect of memory from the effect of records. Limitations include selection (archived projects are survivors), the difficulty of matching task difficulty across artifacts, and the unobserved quality of the descriptions given to participants. The protocol is a design, and no data have been collected.

Appendix H

Empirical Agenda

The modeling claims of this book are conditional on their models, and none has been tested against data collected for this work. Table H.1 states, for the principal modeling claims, what observation would count for or against each. The column on current knowledge is limited to what this book itself provides.

Table H.1: Observations that would test the principal modeling claims.

Claim	Observation that would confirm or refute it	Currently available here
Two-channel recovery (Definition 3.1)	Retrospective retrieval tasks (Appendix G); fit of delay dependence; comparison of joint success with the product of channel successes to test independence	Simulation agreement only (E5)
Non-monotone value of added documentation (Chapter 12)	Controlled tasks varying the volume and organization of documentation for the same artifact and measuring time and success	None
Lock-in and critical mass (Proposition 26.7, Proposition 46.1)	Observed migration of projects under varying levels of interoperability and switching cost	None
Visibility as preferential attachment (Proposition 19.3)	Longitudinal records of recognition signals for projects of comparable quality	None
Review quorum with a common-cause floor (Proposition 27.4)	Seeded defects reviewed by varying numbers of reviewers; estimate of the correlation of detection	None
Closure of declared reads (Theorem 31.3)	Audit of published repositories that records reads during execution and compares them with the declared inputs	One toy case (E6)
Time-dependence of dependency resolution (Proposition 11.3)	Re-resolution of archived manifests at later dates and comparison with the originally resolved sets	One toy case (E4)
Developer model of knowledge, capability, and responsibility (Definition 22.1)	Longitudinal observation or surveys that measure the three components separately and test their independence	None
Continuity debt, orphaning routes, and backlog (Proposition 23.2, Proposition 23.4, Proposition 23.7)	Longitudinal data on projects recording dependents, maintainers with rights and with demonstrated ability, and the time to unaddressed obligations	None

Appendix I

Register of Results by Epistemic Status

This register is generated from the source of the book. Each entry gives the kind, number, title, and the status label attached to the statement. The label records the kind of support and not its strength: a result proved for a toy model, a result proved under a cryptographic idealization, and a result that is a direct consequence of a definition all carry the same label, and the qualification stated with each result should be read together with the register.

Counts by part

Part	Proved	Investigated	Conjectural	Design
I	5	0	0	0
II	7	0	0	0
III	7	0	0	1
IV	7	0	0	0
V	16	0	0	0
VI	16	0	1	0
VII	10	0	0	0
VIII	13	0	1	0
IX	10	0	0	0

Counts by mathematical function

The class describes what kind of statement a result is and not how important it is. Definitional consequences follow from definitions in a few lines; structural results establish properties or equivalences of the defined objects; impossibility or independence results exhibit constructions showing that something cannot be derived or that properties are independent; complexity results concern computational hardness; preservation results state which invariants survive a transformation; model-dependent quantitative results are exact for a stipulated model or idealization.

Class	Results
complexity	2
conjecture	2
definitional consequence	11
impossibility or independence	33
model-dependent quantitative	20
preservation	7
structural	19

proved (91)

Kind	No.	Title	Class
Proposition	Proposition 2.4	Ordering of the persistence notions	structural
Proposition	Proposition 3.2	Behavior of the two-channel model	model-dependent quantitative
Proposition	Proposition 3.3	Recoverability can increase with delay	model-dependent quantitative
Proposition	Proposition 4.3	Basic structure of sufficiency	structural
Proposition	Proposition 4.5	Hardness of multi-class preservation	complexity
Proposition	Proposition 5.3	Rename invariance and reuse hazard	impossibility or independence
Proposition	Proposition 6.3	Class limitation	impossibility or independence
Proposition	Proposition 7.2	Reorganization and references	definitional consequence
Proposition	Proposition 7.4	Axis ordering	model-dependent quantitative
Proposition	Proposition 8.2	Label capacity	model-dependent quantitative
Proposition	Proposition 8.3	Names must be reconstructible	model-dependent quantitative
Proposition	Proposition 8.4	Value of a convention	model-dependent quantitative
Proposition	Proposition 9.3	Composition of idempotent operations	definitional consequence
Proposition	Proposition 10.1	Contracts underdetermine safe modification	impossibility or independence
Proposition	Proposition 10.2	Intent is not a function of the change	impossibility or independence
Proposition	Proposition 11.3	Resolution depends on time	structural
Theorem	Theorem 11.4	Hardness of installability	complexity
Proposition	Proposition 12.2	Noise can destroy sufficiency	structural
Proposition	Proposition 12.3	Optimal ordering for single-item need	model-dependent quantitative
Proposition	Proposition 13.1	Executables underdetermine sources	impossibility or independence
Proposition	Proposition 13.3	Names are neither sound nor complete	impossibility or independence

Kind	No.	Title	Class
Proposition	Proposition 14.1	Single-slot conflict and its resolution	structural
Proposition	Proposition 14.3	Compatibility cannot be decided mechanically	impossibility or independence
Proposition	Proposition 15.1	Intermediate states under non-atomic commit	structural
Theorem	Theorem 16.2	Identifier determines ancestry	structural
Proposition	Proposition 16.3	Behavior of operations on identifiers	definitional consequence
Proposition	Proposition 18.2	Commit authorship is unauthenticated	impossibility or independence
Proposition	Proposition 19.2	Aggregate counts are not invertible	impossibility or independence
Proposition	Proposition 19.3	Cumulative advantage with equal quality	model-dependent quantitative
Proposition	Proposition 20.2	Forking does not constrain the original	definitional consequence
Proposition	Proposition 20.3	Technical possibility and legal permission are independent	impossibility or independence
Proposition	Proposition 20.5	Joint incompatibility occurs	structural
Proposition	Proposition 21.2	Decisions are not determined by mechanisms	impossibility or independence
Proposition	Proposition 21.3	Refusal underdetermines reason	impossibility or independence
Proposition	Proposition 22.2	Independence of knowledge, capability, and responsibility	impossibility or independence
Proposition	Proposition 22.3	No total order	impossibility or independence
Proposition	Proposition 22.4	Responsibility is not discharged by deletion	definitional consequence
Proposition	Proposition 23.2	Properties of continuity debt	structural
Proposition	Proposition 23.4	Routes to orphaning	structural
Proposition	Proposition 23.5	Redundant coverage	definitional consequence
Proposition	Proposition 23.6	Succession criterion	definitional consequence
Proposition	Proposition 23.7	Backlog dynamics	model-dependent quantitative
Proposition	Proposition 24.3	The projection loses information	impossibility or independence

Kind	No.	Title	Class
Proposition	Proposition 24.5	Preservation by extension steps	preservation
Proposition	Proposition 24.6	Cycles are infeasible in the random-oracle idealization	model-dependent quantitative
Proposition	Proposition 25.2	Merge bases need not be unique	structural
Theorem	Theorem 25.4	Invariants of lineage under standard operations	preservation
Proposition	Proposition 25.5	Squashing is lossy	impossibility or independence
Theorem	Theorem 26.2	Structural rules cannot break symmetry	impossibility or independence
Corollary	Corollary 26.4	Authority nonuniqueness	impossibility or independence
Proposition	Proposition 26.5	Dominance under overlapping recognition	model-dependent quantitative
Proposition	Proposition 26.7	Lock-in	model-dependent quantitative
Theorem	Theorem 27.2	Independence of the four predicates	impossibility or independence
Proposition	Proposition 27.4	Review quorum and common-cause failure	model-dependent quantitative
Theorem	Theorem 28.2	Properties of the pathwise merge	structural
Proposition	Proposition 28.4	Governance sovereignty of gated federations	definitional consequence
Theorem	Theorem 29.3	Lineage preservation theorem	preservation
Theorem	Theorem 29.4	Institutional portability criterion	definitional consequence
Proposition	Proposition 30.2	Prose publication has a nonempty specification gap	impossibility or independence
Theorem	Theorem 31.3	Reproducibility closure criterion	structural
Proposition	Proposition 32.2	The three properties are independent	impossibility or independence
Proposition	Proposition 32.3	Commit timestamps are unauthenticated	impossibility or independence
Theorem	Theorem 32.4	Anchoring bound	model-dependent quantitative
Proposition	Proposition 33.2	Verification does not entail inference	impossibility or independence
Proposition	Proposition 33.3	Passing tests does not establish a universal property	impossibility or independence

Kind	No.	Title	Class
Proposition	Proposition 33.7	Validity is relative to the declared assumptions	structural
Proposition	Proposition 33.8	Checking a derivation requires no interpretation of the symbols	impossibility or independence
Proposition	Proposition 34.2	What a disclosure record establishes	impossibility or independence
Proposition	Proposition 35.1	Diffs cannot distinguish a move from a deletion and an addition	impossibility or independence
Proposition	Proposition 35.2	Co-change does not determine dependency	impossibility or independence
Proposition	Proposition 36.3	Expansion terminates uniquely iff the reference graph is well founded	structural
Proposition	Proposition 36.4	Embeddings do not identify	impossibility or independence
Theorem	Theorem 37.4	Extraction preserves expansion, integrity, and acyclicity	preservation
Theorem	Theorem 37.6	Hoisting preservation under non-capture	preservation
Proposition	Proposition 37.8	Closure of preservation under composition	definitional consequence
Proposition	Proposition 38.3	Properties of the grounded labelling	structural
Theorem	Theorem 39.1	Epistemic revision theorem	preservation
Proposition	Proposition 39.2	Replay is relative to the rule version	impossibility or independence
Proposition	Proposition 39.4	Append-only logs cannot reorder past events	definitional consequence
Theorem	Theorem 40.3	Interface soundness	preservation
Proposition	Proposition 40.4	Non-transfer of standing	impossibility or independence
Proposition	Proposition 41.2	Minimal architecture	structural
Proposition	Proposition 42.2	Independent replaceability of layers	structural
Proposition	Proposition 42.3	Hash-chained logs are tamper-evident	model-dependent quantitative
Proposition	Proposition 43.1	Survival under replication with common-cause failure	model-dependent quantitative
Proposition	Proposition 43.3	Loss and compromise trade-off	model-dependent quantitative
Proposition	Proposition 45.2	Functions and legitimacy are independent	impossibility or independence

Kind	No.	Title	Class
Proposition	Proposition 46.1	Critical mass	model-dependent quantitative
Corollary	Corollary 46.2	Interoperability removes lock-in	model-dependent quantitative
Proposition	Proposition 47.1	Correlated support has a floor of error	model-dependent quantitative
Proposition	Proposition 48.2	Institution-likeness is independent of size	impossibility or independence

conjectural (2)

Kind	No.	Title	Class
Conjecture	Conjecture 28.5	Federation scaling	conjecture
Conjecture	Conjecture 37.9	Invariant lattice	conjecture

design argument (1)

Kind	No.	Title	Class
Proposition	Proposition 11.5	Completeness-inspectability tradeoff	structural

Bibliography

[1] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency Solving: A Separate Concern in Component Evolution Management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[2] W. B. Arthur. Competing Technologies, Increasing Returns, and Lock-In by Historical Events. *The Economic Journal*, 99(394):116–131, 1989.

[3] M. Baker. 1,500 Scientists Lift the Lid on Reproducibility. *Nature*, 533:452–454, 2016.

[4] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[5] B. Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the Winter 1990 USENIX Conference*, 1990.

[6] J. B. Buckheit and D. L. Donoho. WaveLab and Reproducible Research. In *Wavelets and Statistics*, Lecture Notes in Statistics 103, pages 55–81. Springer, 1995.

[7] S. Chacon and B. Straub. *Pro Git*. Apress, second edition, 2014.

[8] C. Collberg and T. A. Proebsting. Repeatability in Computer Systems Research. *Communications of the ACM*, 59(3):62–69, 2016.

[9] L. de Moura and S. Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction, CADE 28*, Lecture Notes in Computer Science 12699, pages 625–635. Springer, 2021.

[10] R. Di Cosmo and S. Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *Proceedings of the 14th International Conference on Digital Preservation (iPRES)*, 2017.

[11] P. M. Dung. On the Acceptability of Arguments and Its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artificial Intelligence*, 77(2):321–357, 1995.

[12] Flyxion. RSVP-Polyxan: A Unified Field Theory of Semantic Hyperstructures. Manuscript, 2025.

[13] Flyxion. Membrane Intelligence and Selective Semantics: A Category-Sheaf Formalization within RSVP-Polyxan. Manuscript, 2025.

[14] Flyxion. A Coherence-First Alternative to Social Media: Semantic Objects, Entropy-Bounded Evolution, and Provenance-Anchored Identity. Manuscript, January 2026.

[15] Flyxion. Manifold-Aligned Semantic Dynamics: From Philosophical Constraint to Geometric, Categorical, and Operational Unification. Manuscript, February 2026.

[16] Flyxion. The Geometry of Correction: Self-Reference, Reality-Anchoring, and the Design of Systems That Remain Answerable. Manuscript, July 2026.

[17] Flyxion. Adversaria: A Specification for a Public Claim Graph. Manuscript, September 2026.

[18] A. O. Hirschman. *Exit, Voice, and Loyalty: Responses to Decline in Firms, Organizations, and States*. Harvard University Press, 1970.

[19] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[20] R. M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[21] M. L. Katz and C. Shapiro. Network Externalities, Competition, and Compatibility. *The American Economic Review*, 75(3):424–440, 1985.

[22] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[23] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208, 2006.

[24] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology, CRYPTO '87*, Lecture Notes in Computer Science 293, pages 369–378. Springer, 1988.

[25] R. K. Merton. Priorities in Scientific Discovery: A Chapter in the Sociology of Science. *American Sociological Review*, 22(6):635–659, 1957.

[26] National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. National Academies Press, 2019.

[27] W. L. Oberkampf and C. J. Roy. *Verification and Validation in Scientific Computing*. Cambridge University Press, 2010.

[28] R. D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, 2011.

[29] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[30] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[31] M. J. Salganik, P. S. Dodds, and D. J. Watts. Experimental Study of Inequality and Unpredictability in an Artificial Cultural Market. *Science*, 311(5762):854–856, 2006.

[32] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[33] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The First Collision for Full SHA-1. In *Advances in Cryptology, CRYPTO 2017*, Lecture Notes in Computer Science 10401, pages 570–596. Springer, 2017.

[34] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[35] W. F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *Proceedings of the 6th International Conference on Software Engineering*, pages 58–67, 1982.

[36] S. E. Toulmin. *The Uses of Argument*. Cambridge University Press, 1958.

[37] D. Barreau and B. A. Nardi. Finding and Reminding: File Organization from the Desktop. *ACM SIGCHI Bulletin*, 27(3):39–43, 1995.

[38] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pages 1277–1286, 2012.

[39] W. Jones. *Keeping Found Things Found: The Study and Practice of Personal Information Management*. Morgan Kaufmann, 2007.

[40] W. Jones and J. Teevan, editors. *Personal Information Management*. University of Washington Press, 2007.

[41] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[42] S. O'Mahony and F. Ferraro. The Emergence of Governance in an Open Source Community. *Academy of Management Journal*, 50(5):1079–1106, 2007.

[43] E. S. Raymond. The Cathedral and the Bazaar. *First Monday*, 3(2), 1998.

[44] C. M. Schweik and R. C. English. *Internet Success: A Study of Open-Source Software Commons*. MIT Press, 2012.