Intrinsic Repository Resilience

A Community Architectural Proposal for Duplicati

Improving Repository Survivability Through Intrinsic Recoverability

Document Type

Community Architectural Proposal

Purpose

This document presents an architectural vision for introducing intrinsic repository resilience as a possible future capability within Duplicati.

The proposal focuses on repository survivability, recoverability and long-term integrity while preserving compatibility with existing backup strategies and architectural principles.

Scope

This document describes:

- the motivation for repository resilience;
- the architectural principles behind the proposal;
- possible behavioural models;
- user experience considerations;
- potential technical directions for future exploration.

This document intentionally does **not** define:

- a required implementation;
- a specific algorithm;
- a repository format change;
- a development roadmap.

Author Role

Prepared as a community contribution for discussion and evaluation by the Duplicati project maintainers and community.

The author does not propose to implement this capability directly, but to contribute an architectural perspective and a structured discussion framework.

Status

Proposal for Community Discussion

Version: 1.0

Date: July 2026

Guiding Principle

> A repository should not only store backup data.
>
> It should also possess the capability to preserve its own recoverability.

Intended Audience

- Duplicati maintainers;
- contributors;
- technically interested community members;
- users interested in long-term repository preservation.

Disclaimer

This document represents an independent architectural proposal.

All implementation decisions, technical approaches and development priorities remain entirely under the responsibility of the Duplicati project maintainers.

Abstract

Backup repositories are expected to preserve data reliably over long periods of time. However, repository integrity can still be affected by events such as accidental deletion, storage corruption or loss of individual repository objects.

Existing protection mechanisms, including multiple backup destinations, replication strategies and storage redundancy, address important failure scenarios but operate at different architectural layers.

This proposal introduces the concept of **Intrinsic Repository Resilience**: a repository capability designed to improve the ability of an individual repository instance to tolerate and recover from limited internal object failures.

The proposal does not define a specific implementation, redundancy algorithm or repository format. Instead, it defines architectural principles, behavioural objectives and user experience concepts that could guide future development.

The central idea is that repository resilience should become a continuously maintained property of the repository lifecycle. Users should define a desired resilience objective, while Duplicati remains responsible for managing the underlying mechanisms required to achieve and maintain that objective.

Intrinsic Repository Resilience is intended to complement existing backup strategies rather than replace them. It addresses a distinct failure domain: the ability of a repository to preserve its own recoverability independently from external copies or storage-level redundancy.

This document is presented as a community architectural proposal intended to stimulate discussion and exploration within the Duplicati project.

Executive Summary

Intrinsic Repository Resilience

A Community Architectural Proposal for Duplicati

Overview

Backup repositories are designed to preserve valuable data over time. However, even a valid and complete repository may become partially compromised by events such as accidental deletion, storage corruption or loss of individual repository objects.

Current backup protection strategies address many important scenarios, including multiple destinations, replication and storage redundancy. These mechanisms remain essential and continue to represent fundamental best practices.

This proposal explores an additional protection layer: **Intrinsic Repository Resilience**.

The Core Idea

The central idea is to introduce repository resilience as an intrinsic capability of the repository itself.

A resilient repository would not simply store backup data.

It would also maintain the ability to tolerate and recover from limited internal object loss.

The repository would therefore evolve from a passive storage structure into an actively preserved digital asset.

The Problem Addressed

Existing protection mechanisms typically operate outside the repository itself.

Examples include:

- additional backup copies;
- cloud replication;
- RAID systems;
- filesystem-level protection.

These approaches protect against important failure scenarios, especially complete storage failure or disaster events.

However, they do not directly address a different class of problems:

- accidental deletion of individual repository files;

- isolated corruption of repository objects;
- partial loss of storage content while the storage system remains available.

Intrinsic Repository Resilience focuses specifically on this internal repository integrity domain.

Architectural Principle

The proposal introduces a policy-driven model.

Users define the desired resilience objective.

Duplicati manages the mechanisms required to maintain that objective.

Users should not need to understand:

- redundancy algorithms;
- parity structures;
- recovery mathematics;
- internal repository organisation.

The complexity remains inside the software.

The confidence belongs to the user.

Relationship with Existing Backup Strategies

Intrinsic Repository Resilience is not intended to replace existing backup methodologies.

It complements them.

A complete protection strategy may include:

- repository resilience for internal recoverability;
- multiple destinations for copy redundancy;
- geographical separation for disaster recovery.

Each layer addresses a different failure domain.

Together, they provide stronger protection than any single mechanism alone.

Expected Benefits

Potential benefits include:

- increased repository survivability;
- reduced impact from limited object loss;
- improved confidence in long-term repositories;
- better preservation of repositories that outlive their original source systems;

- reduced dependency on external recovery sources for limited corruption scenarios.

Implementation Independence

This proposal intentionally avoids defining a specific technical solution.

Possible future implementations could involve different approaches, including redundancy techniques, erasure coding or other mechanisms.

The architectural objective remains independent from the implementation method:

> A repository should continuously maintain its configured ability to recover from limited internal failures.

User Experience Vision

From the user's perspective, repository resilience should remain simple.

The user chooses a resilience policy.

Duplicati maintains it.

Repository health should be visible through clear indicators, allowing users to understand whether their repository is:

- healthy;
- protected;
- degraded;
- recovering;
- requiring attention.

Purpose of This Document

This document is not a technical specification or an implementation request.

It is a community architectural proposal intended to stimulate discussion among maintainers, contributors and users.

Its purpose is to explore whether intrinsic repository resilience could represent a valuable future capability for Duplicati.

Final Statement

Backup software protects data.

A resilient repository protects the ability to continue protecting that data.

Intrinsic Repository Resilience proposes a possible evolution of the backup repository concept:

from a container that stores information
to a structure capable of preserving its own recoverability.

Table of Contents

Intrinsic Repository Resilience 1
A Community Architectural Proposal for Duplicati 1
Improving Repository Survivability Through Intrinsic Recoverability 1
Document Type 1
Purpose 1
Scope 1
Author Role 2
Status 2
Guiding Principle 2
Intended Audience 2
Disclaimer 2

Abstract 3

Executive Summary 4
Intrinsic Repository Resilience 4
A Community Architectural Proposal for Duplicati 4
Overview 4
The Core Idea 4
The Problem Addressed 4
Architectural Principle 5
Relationship with Existing Backup Strategies 5
Expected Benefits 5
Implementation Independence 6
User Experience Vision 6
Purpose of This Document 6
Final Statement 6

Chapter 1 — Introduction 13
1.1 Purpose of this Proposal 13
1.2 Motivation 13
1.3 Scope 14
1.4 Guiding Philosophy 14
1.5 About this Proposal 15
1.6 Reading this Document 15
Key Takeaway 15

Chapter 2 — Failure Domains 16
2.1 Introduction 16
2.2 What is a Failure Domain? 16
2.3 Failure Domain Classification 16
Object Failure 16
Repository Failure 17
Storage Failure 17
Site Failure 17
2.4 Existing Protection Mechanisms 18
2.5 Repository Replication versus Repository Resilience 18

2.6 Intrinsic versus Extrinsic Protection 19
2.7 Architectural Implications 19
2.8 Summary 19
Key Takeaway 20

Chapter 3 — Repository Resilience Principles 21
3.1 Introduction 21
3.2 RR-1 — Policy over Mechanism 21
3.3 RR-2 — Continuous Repository Resilience 21
3.4 RR-3 — Adaptive Repository Evolution 22
3.5 RR-4 — Intrinsic Recoverability 22
3.6 RR-5 — Repository Health Must Be Measurable 22
3.7 RR-6 — Backward Compatibility 23
3.8 RR-7 — Failure Domain Separation 23
3.9 RR-8 — Technology Neutrality 23
3.10 RR-9 — Incremental Adoption 24
3.11 RR-10 — Simplicity for the User 24
3.12 Architectural Consequences 24
3.13 Summary 24
Key Takeaway 25

Chapter 4 — Architectural Integration 26
4.1 Introduction 26
4.2 Architectural Philosophy 26
4.3 Repository Resilience Manager 26
4.4 Responsibilities 27
4.5 Repository Lifecycle Integration 27
4.6 Continuous Adaptation 27
4.7 Repository Health 28
4.8 Repository Resilience Policies 28
4.9 Repository Evolution 29
4.10 Technology Independence 29
4.11 Architectural Benefits 29
4.12 Summary 30
Key Takeaway 30

Chapter 5 — Relationship with Existing Features 31
5.1 Introduction 31
5.2 Complementing Existing Features 31
5.3 Multiple Backup Destinations 31
5.4 The 3-2-1 Backup Strategy 32
5.5 RAID and Storage Redundancy 32
5.6 Cloud Storage Replication 32
5.7 Repository Verification and Repair 33
5.8 External Synchronization Tools 33
5.9 Layered Protection 33
5.10 Architectural Compatibility 34
5.11 Summary 34
Key Takeaway 34

Chapter 6 — Behavioural Model 36

6.1 Introduction 36
6.2 The Repository as a Living System 36
6.3 Repository States 36
Initializing 36
Protected 36
Adapting 37
Degraded 37
Recovering 37
Healthy 37
6.4 Continuous Protection 37
6.5 Adaptive Protection 38
6.6 Repository Health Evolution 38
6.7 User Experience 38
6.8 Failure Response 39
6.9 Long-Term Evolution 39
6.10 Behaviour Before Implementation 39
6.11 Summary 40
Key Takeaway 40

Chapter 7 — Architectural Value and Expected Benefits 41
7.1 Introduction 41
7.2 A New Architectural Property 41
7.3 Increased Repository Survivability 41
7.4 Improved Operational Confidence 41
7.5 Long-Term Repository Preservation 42
7.6 Separation of Responsibilities 42
7.7 Simplicity for Users 42
7.8 Flexibility for Maintainers 43
7.9 Incremental Adoption 43
7.10 Community Value 43
7.11 Recognizing the Trade-offs 43
7.12 Summary 44
Key Takeaway 44

Chapter 8 — User Experience and Configuration Model 45
8.1 Introduction 45
8.2 Policy-Oriented Configuration 45
8.3 Advanced Configuration 45
8.4 Repository Health Dashboard 46
8.5 Health Indicators 46
8.6 Progressive Disclosure 47
8.7 Behaviour During Backup Operations 47
8.8 Behaviour After Repository Changes 47
8.9 Behaviour During Object Loss 47
8.10 User Confidence 48
8.11 Consistency with Duplicati Philosophy 48
8.12 Summary 48
Key Takeaway 49

Chapter 9 — Design Considerations and Implementation Flexibility 50
9.1 Introduction 50

9.2 Preserving Architectural Freedom 50
9.3 Evolution Rather Than Replacement 50
9.4 Independence from Specific Algorithms 51
9.5 Scalability 51
9.6 Incremental Maintenance 51
9.7 Compatibility with Repository Evolution 51
9.8 Extensibility 52
9.9 Implementation Neutrality 52
9.10 Community Collaboration 52
9.11 Summary 53
Key Takeaway 53

Chapter 10 — Discussion, Trade-offs and Open Questions 54
10.1 Introduction 54
10.2 Storage Overhead 54
10.3 Computational Cost 54
10.4 Repository Complexity 55
10.5 Backward Compatibility 55
10.6 Performance Considerations 55
10.7 Repository Growth 55
10.8 Interaction with Existing Maintenance Operations 56
10.9 Questions for Discussion 56
10.10 Community Evaluation 56
10.11 Scope Boundaries 57
10.12 Summary 57
Key Takeaway 57

Chapter 11 — Illustrative Operational Model 58
11.1 Introduction 58
11.2 Repository Objects 58
11.3 Resilience Metadata 58
11.4 Illustrative Recovery Objects 59
11.5 Continuous Protection 59
11.6 Minimum Resilience Objective 59
11.7 Adaptive Repository Evolution 60
11.8 Illustrative Example 60
11.9 Behaviour During Object Loss 60
11.10 Policy-Driven Operation 61
11.11 Summary 61
Key Takeaway 61

Chapter 12 — Conclusion 62
12.1 Introduction 62
12.2 From Storage Container to Resilient Repository 62
12.3 Complementing Existing Protection Strategies 62
12.4 The Core Principle 63
12.5 A User-Centric Vision 63
12.6 An Invitation to Discussion 63
12.7 Final Perspective 64
Final Key Takeaway 64

Appendix A — Illustrative Example of Repository Resilience 65
A.1 Purpose 65
A.2 Initial Repository State 65
A.3 Repository Growth 65
A.4 Object Loss Scenario 66
A.5 Resilience Restoration 66
A.6 Conceptual Interpretation 67

Appendix B — Possible Technical Directions 68
B.1 Purpose 68
B.2 Redundancy-Based Approaches 68
B.3 Recovery Objects 68
B.4 Dynamic Resilience Groups 68
B.5 Minimum Resilience Policy 69
B.6 Verification and Repair Integration 69
B.7 Technology Evolution 69
B.8 Closing Note 70

Chapter 1 — Introduction

1.1 Purpose of this Proposal

Duplicati has established itself as one of the most flexible and feature-rich open-source backup solutions available today. Its architecture combines client-side encryption, block-level deduplication, compression and support for a wide range of storage backends while maintaining a remarkably simple user experience.

Over the years, Duplicati has continuously evolved by improving reliability, storage efficiency and usability. Every backup operation ultimately produces a repository that becomes the persistent representation of the protected data.

In many real-world situations, that repository eventually becomes the only remaining copy of the original information.

At that point, preserving the repository is equivalent to preserving the data themselves.

This proposal originates from that observation.

Rather than suggesting another backup feature, it explores whether the repository itself could acquire an additional architectural property:

> **Intrinsic Repository Resilience**

The proposal deliberately avoids prescribing implementation techniques.

Instead, it attempts to describe the properties that a resilient repository could exhibit, leaving implementation choices entirely to the Duplicati maintainers.

1.2 Motivation

Current Duplicati repositories are extremely efficient.

They minimize storage consumption through block-level deduplication while providing excellent portability across local disks, NAS devices and cloud storage providers.

Within each repository, however, every repository object currently represents a unique component of the overall backup structure.

If one of those objects becomes unavailable—for example because of accidental deletion, filesystem corruption or silent media degradation—the repository may become partially or completely unusable unless:

- the original source data are still available, or
- another complete repository copy exists.

The recent introduction of multiple backup destinations significantly improves protection against storage failures and fully supports established backup strategies such as the 3-2-1 rule.

This proposal addresses a different problem.

It focuses on the ability of **each individual repository instance** to tolerate limited object loss without immediately losing its ability to restore data.

1.3 Scope

This document explores one possible architectural evolution of the Duplicati repository.

It does not propose modifications to backup scheduling, retention policies, encryption algorithms or storage backends.

Likewise, it does not attempt to replace existing protection mechanisms such as:

- RAID;
- mirrored storage;
- filesystem redundancy;
- cloud replication;
- multi-destination repositories;
- off-site backups.

Those technologies address different failure scenarios.

The objective of this proposal is substantially narrower:

> **to improve the intrinsic recoverability of an individual repository instance when a limited number of repository objects become unavailable.**

1.4 Guiding Philosophy

The central idea behind this proposal can be summarized in one sentence.

> **Users should configure the desired level of repository resilience, not the mechanism used to achieve it.**

This principle intentionally separates policy from implementation.

Users express an objective.

Duplicati determines how that objective should be maintained throughout the repository lifecycle.

This philosophy is already reflected in many existing Duplicati features.

Users do not decide how blocks are deduplicated.

They do not manually organize repository volumes.

They do not determine when repository compaction should occur.

Instead, users define high-level objectives while Duplicati remains responsible for the underlying implementation.

Repository resilience should follow the same design philosophy.

1.5 About this Proposal

This document should not be interpreted as a request for a specific implementation.

The author is not a member of the Duplicati development team and does not intend to propose code or implementation details.

Instead, this proposal represents an attempt to organize a possible architectural direction in a form that may facilitate technical discussion within the Duplicati community.

If any part of these ideas proves valuable, implementation decisions should naturally remain entirely under the responsibility of the Duplicati maintainers.

1.6 Reading this Document

The remaining chapters progressively develop the proposal.

The discussion intentionally starts from architectural principles before considering possible implementation strategies.

This ordering reflects the belief that architectural properties should drive implementation choices, rather than the opposite.

Readers primarily interested in implementation aspects may therefore find the accompanying Technical White Paper more appropriate, while this document focuses on motivations, design principles and high-level architecture.

Key Takeaway

This proposal is not about adding redundancy.

It is about exploring whether **repository resilience** should become an architectural property of Duplicati repositories, allowing each repository instance to preserve its own recoverability despite limited internal failures.

Chapter 2 — Failure Domains

2.1 Introduction

Before discussing possible solutions, it is essential to clearly identify the type of failures this proposal intends to address.

Different technologies are designed to protect against different classes of failures. Although these technologies may appear similar from a user's perspective, they operate at distinct architectural layers and therefore solve fundamentally different problems.

One of the key motivations behind this proposal is the observation that **repository object failures represent a failure domain that is not directly addressed by current repository replication strategies**.

Understanding this distinction is fundamental to correctly evaluating the scope of this proposal.

2.2 What is a Failure Domain?

A failure domain is the set of components that can be affected by a single failure event.

Different failure domains require different protection strategies.

For example, a corrupted file, a failed hard disk and the destruction of an entire storage site are all failures, but they occur at different architectural levels and therefore require different recovery mechanisms.

This proposal focuses exclusively on failures occurring **inside an otherwise existing repository**.

2.3 Failure Domain Classification

For the purpose of this proposal, repository-related failures can be classified into four distinct domains.

Object Failure

An Object Failure occurs when one or more repository objects become unavailable while the repository itself continues to exist.

Typical examples include:

- accidental deletion of a single repository object;
- silent media corruption (bit rot);
- filesystem inconsistencies;
- isolated storage errors;
- partial cloud synchronization failures;

- human error during manual repository maintenance.

In these situations, the storage device itself remains operational.

Only a limited portion of the repository has been lost.

This proposal primarily addresses this failure domain.

Repository Failure

A Repository Failure occurs when an entire repository instance becomes unavailable.

Examples include:

- complete disk failure;
- loss of a removable drive;
- NAS hardware failure;
- accidental deletion of the repository directory;
- cloud bucket deletion.

Repository replication, multiple backup destinations and traditional backup strategies effectively address this failure domain.

This proposal does not attempt to replace those mechanisms.

Storage Failure

A Storage Failure affects the underlying storage infrastructure.

Examples include:

- disk controller failures;
- RAID degradation;
- SSD wear-out;
- filesystem corruption affecting the entire volume.

These failures are typically mitigated through hardware redundancy, RAID, resilient filesystems or enterprise storage technologies.

Again, these mechanisms operate below the repository layer.

Site Failure

A Site Failure represents the loss of an entire physical location.

Typical examples include:

- fire;
- flooding;

- theft;
- power events;
- natural disasters.

Protection against this failure domain is provided through off-site replication, cloud storage and established backup strategies such as the 3-2-1 rule.

2.4 Existing Protection Mechanisms

Duplicati already integrates or interoperates with several technologies that address the failure domains described above.

These include:

- multiple backup destinations;
- cloud storage providers;
- RAID and mirrored storage;
- resilient filesystems;
- external synchronization tools;
- traditional disaster recovery strategies.

These mechanisms are extremely valuable and should continue to be regarded as best practices.

However, they primarily protect against storage-level or repository-level failures.

They do not necessarily provide intrinsic recoverability for each individual repository instance.

2.5 Repository Replication versus Repository Resilience

One of the most important distinctions introduced by this proposal is the separation between **repository replication** and **repository resilience**.

Repository replication answers the following question:

> “Is another complete copy of the repository available?”

Repository resilience answers a different question:

> “Can this specific repository instance continue to function despite losing a limited number of internal repository objects?”

Although both approaches improve data protection, they address different failure domains.

Consequently, they should not be considered competing solutions.

Instead, they complement each other.

2.6 Intrinsic versus Extrinsic Protection

Another useful distinction concerns the origin of the protection itself.

Repository replication provides **extrinsic protection**.

The repository survives because another copy exists elsewhere.

Repository resilience provides **intrinsic protection**.

Each repository instance possesses its own capability to tolerate limited internal failures without immediately depending on external copies.

This distinction represents one of the fundamental ideas behind this proposal.

2.7 Architectural Implications

Recognizing repository object failures as an independent failure domain has important architectural consequences.

Rather than treating repository integrity exclusively as a property of the storage infrastructure, the repository itself becomes responsible for maintaining a configurable level of recoverability.

This does not eliminate the need for replication, RAID or off-site backups.

Instead, it introduces an additional layer of protection operating at the repository level.

The resulting protection strategy becomes layered rather than monolithic.

Each architectural layer contributes to the overall reliability of the backup ecosystem while remaining responsible for a different class of failures.

2.8 Summary

This proposal intentionally focuses on **Object Failures**.

It neither replaces nor competes with existing storage protection mechanisms.

Instead, it complements them by introducing the concept of **intrinsic repository resilience**.

In other words:

- storage technologies protect storage;
- replication protects repository availability;
- disaster recovery protects geographical availability;
- repository resilience protects the integrity and recoverability of each individual repository instance.

These objectives are complementary and together contribute to a more robust backup architecture.

Key Takeaway

Repository replication and repository resilience solve different problems.

Replication ensures that another repository exists.

Repository resilience ensures that each repository instance remains recoverable even after limited internal failures.

Recognizing these as separate failure domains provides the conceptual foundation upon which the remainder of this proposal is built.

Site Failure (Off-site / 3-2-1 Strategy)

Repository Failure

(Multi-target / Replication)

Storage Failure

(RAID / ZFS / Filesystem)

Object Failure

(Repository Resilience Framework)

Each layer complements the others rather than replacing them.

Chapter 3 — Repository Resilience Principles

3.1 Introduction

The previous chapter established that repository object failures constitute a distinct failure domain that is not directly addressed by existing repository replication mechanisms.

Having identified the problem space, the next step is to define the architectural principles that should guide any future implementation.

These principles deliberately avoid prescribing specific technologies or algorithms.

Instead, they describe the behavioural properties that a resilient repository should exhibit, regardless of the implementation eventually adopted.

For this reason, the principles presented here should be regarded as architectural objectives rather than implementation requirements.

3.2 RR-1 — Policy over Mechanism

Users should configure the desired resilience objective rather than the mechanism used to achieve it.

This proposal intentionally separates **what** users expect from **how** Duplicati satisfies that expectation.

A user may simply express a resilience policy such as:

- Standard Resilience
- High Resilience
- Custom Resilience Policy

or specify a quantitative objective such as a desired resilience level.

The internal implementation remains entirely under the responsibility of Duplicati.

This design philosophy preserves implementation flexibility while maintaining a simple user experience.

3.3 RR-2 — Continuous Repository Resilience

Every repository object should become part of the repository resilience domain as soon as reasonably possible after being successfully committed.

No repository object should remain permanently unprotected while waiting for future repository operations.

This principle intentionally avoids implementation assumptions.

Whether resilience is achieved incrementally, periodically or through another strategy is considered an implementation detail.

The only architectural requirement is that the configured resilience objective be continuously maintained throughout the repository lifecycle.

3.4 RR-3 — Adaptive Repository Evolution

Repositories continuously evolve.

Backup operations create new repository objects.

Compaction reorganises repository contents.

Retention policies remove obsolete data.

Repair and verification modify repository metadata.

Repository resilience should evolve together with these operations.

Rather than being treated as a separate maintenance activity, resilience should become an intrinsic property of normal repository evolution.

The repository should continuously adapt itself in order to preserve the configured resilience objective.

3.5 RR-4 — Intrinsic Recoverability

Each repository instance should possess its own capability to tolerate limited internal failures.

Recoverability should not exclusively depend upon:

- the continued availability of the original source data;
- another repository replica;
- external recovery procedures.

Whenever reasonably possible, the repository should preserve its own recoverability using information already contained within the repository itself.

This principle represents one of the central ideas of this proposal.

3.6 RR-5 — Repository Health Must Be Measurable

Repository resilience should never become an invisible internal feature.

Users should always be able to determine the current resilience status of their repository.

Possible indicators may include:

- overall repository health;
- current resilience level;

- recoverability status;
- integrity verification status.

The proposal intentionally does not prescribe any particular user interface.

Its only objective is to make repository resilience observable.

3.7 RR-6 — Backward Compatibility

Backward compatibility should remain an absolute priority.

Existing repositories must continue to operate exactly as they do today.

Repository resilience should remain entirely optional.

Users choosing not to enable repository resilience should experience no behavioural differences compared with current Duplicati repositories.

This principle minimizes adoption barriers while preserving compatibility with existing installations.

3.8 RR-7 — Failure Domain Separation

Repository resilience should complement, never replace, existing protection mechanisms.

Storage redundancy.

Repository replication.

Disaster recovery.

Off-site backup.

These technologies address different failure domains and should continue to coexist.

Repository resilience simply introduces an additional protection layer operating inside each repository instance.

3.9 RR-8 — Technology Neutrality

This proposal intentionally avoids favouring any specific redundancy technology.

Erasure coding.

Parity-based approaches.

Forward error correction.

Future coding techniques.

Any of these may potentially satisfy the architectural objectives described in this document.

Consequently, the selection of a particular algorithm should remain entirely under the responsibility of the Duplicati maintainers.

Technology evolves.

Architectural principles should remain stable.

3.10 RR-9 — Incremental Adoption

Repository resilience should integrate naturally into the existing Duplicati ecosystem.

Users should be able to adopt repository resilience gradually without requiring repository redesign or disruptive migration procedures.

Whenever feasible, enabling repository resilience should feel like enabling any other optional Duplicati capability.

The objective is evolutionary improvement rather than architectural disruption.

3.11 RR-10 — Simplicity for the User

Repository resilience should increase reliability without increasing complexity.

Advanced algorithms should remain hidden behind a simple configuration model.

Users should not need to understand coding theory, redundancy schemes or storage mathematics.

They should simply decide the level of repository resilience appropriate for their needs.

Duplicati should remain responsible for translating that policy into concrete implementation decisions.

3.12 Architectural Consequences

Taken together, these principles define a significant conceptual shift.

Rather than introducing another maintenance feature, repository resilience becomes an intrinsic architectural property of the repository itself.

This distinction is important.

Features can often be enabled or disabled independently.

Architectural properties instead influence how the entire system behaves throughout its lifecycle.

Repository resilience should therefore be regarded not as another maintenance operation, but as a continuous repository capability.

3.13 Summary

The principles presented in this chapter intentionally avoid implementation details.

Instead, they establish the behavioural expectations that should remain valid regardless of future implementation choices.

This separation between architectural objectives and implementation mechanisms provides the flexibility necessary for the Duplicati maintainers to evaluate different technical solutions while preserving a coherent long-term vision.

Key Takeaway

Repository resilience is defined by its architectural properties, not by the redundancy algorithm used to implement them.

Users configure resilience objectives.

Duplicati determines how those objectives are achieved.

Chapter 4 — Architectural Integration

4.1 Introduction

The previous chapters introduced the motivation for repository resilience and established the architectural principles that should guide any future implementation.

The natural next question is therefore:

How could repository resilience integrate into the existing Duplicati architecture without fundamentally changing its design?

This chapter deliberately avoids proposing concrete classes, APIs or implementation details.

Instead, it introduces a logical architectural model intended to describe responsibilities rather than software components.

4.2 Architectural Philosophy

Duplicati already follows a clear architectural philosophy.

Users configure desired behaviours.

Duplicati determines how those behaviours are achieved.

Examples include:

- block deduplication;
- volume management;
- repository compaction;
- retention policy execution;
- backend abstraction.

Repository resilience should follow the same philosophy.

Users should configure a resilience objective.

Duplicati should remain responsible for continuously maintaining that objective throughout the repository lifecycle.

4.3 Repository Resilience Manager

For the purposes of this proposal, the responsibility for maintaining repository resilience is represented by a logical architectural component referred to as the **Repository Resilience Manager**.

The term *Manager* is intentionally conceptual.

It does **not** imply the existence of a dedicated service, process or software module.

Instead, it represents the architectural responsibility for ensuring that the repository continuously satisfies its configured resilience policy.

Any implementation chosen by the Duplicati maintainers may distribute this responsibility across existing components or introduce new ones.

The proposal intentionally leaves that decision open.

4.4 Responsibilities

Regardless of its implementation, the Repository Resilience Manager would conceptually be responsible for:

- evaluating repository resilience status;
- maintaining the configured resilience objective;
- reacting to repository evolution;
- updating resilience-related metadata;
- exposing repository health information.

These responsibilities define *what* the architecture should accomplish.

They intentionally avoid specifying *how* those objectives should be achieved.

4.5 Repository Lifecycle Integration

Repository resilience should not be considered an isolated maintenance activity.

Instead, it should naturally accompany the normal lifecycle of the repository.

Repository events that may influence resilience include:

- backup operations;
- repository compaction;
- retention cleanup;
- repair procedures;
- verification activities;
- repository import or recreation.

Each of these operations may modify the repository structure.

Consequently, repository resilience should evolve together with the repository itself.

4.6 Continuous Adaptation

One of the central concepts introduced by this proposal is continuous adaptation.

Rather than periodically rebuilding repository resilience from scratch, the repository should strive to preserve the configured resilience objective as it evolves.

This principle intentionally avoids prescribing whether adaptation occurs:

- immediately;
- incrementally;
- lazily;
- during maintenance operations;
- or through another strategy.

Those decisions remain implementation details.

The architectural objective remains simply that repository resilience should continuously reflect the current repository state.

4.7 Repository Health

A resilient repository should also be observable.

Users should not merely enable repository resilience.

They should be able to understand its current state.

For this reason, the proposal introduces the concept of **Repository Health**.

Repository Health represents a logical view of the repository's overall recoverability and resilience status.

Possible indicators may include:

- configured resilience policy;
- current resilience level;
- recoverability status;
- integrity status;
- pending resilience optimization.

The proposal intentionally refrains from defining a specific graphical interface.

Only the architectural concept is considered important.

4.8 Repository Resilience Policies

Rather than exposing implementation parameters, repository resilience should be configured through high-level policies.

Possible examples include:

- Disabled

- Standard
- Enhanced
- Maximum
- Custom

A custom policy could allow advanced users to specify quantitative resilience objectives.

The implementation strategy required to satisfy those objectives should remain entirely under Duplicati's control.

This separation preserves both usability and future implementation flexibility.

4.9 Repository Evolution

Repositories are dynamic systems.

They continuously change throughout their lifetime.

Consequently, repository resilience should also be regarded as a dynamic property rather than a static configuration established only once.

As the repository evolves, resilience should automatically evolve with it.

This adaptive behaviour represents one of the distinguishing characteristics of the proposed framework.

4.10 Technology Independence

The architectural concepts presented in this chapter intentionally remain independent from any particular redundancy technology.

Whether resilience is achieved through:

- erasure coding;
- parity techniques;
- distributed redundancy;
- future coding methods;
- or entirely different approaches,

does not affect the architectural responsibilities described here.

This separation between architecture and implementation preserves the long-term relevance of the proposal even as storage technologies continue to evolve.

4.11 Architectural Benefits

Treating repository resilience as an architectural responsibility rather than as a collection of implementation techniques provides several advantages.

It allows implementation strategies to evolve independently.

It maintains compatibility with existing repositories.

It simplifies the user experience.

Most importantly, it shifts repository resilience from being considered a maintenance operation to becoming a continuous property of the repository itself.

4.12 Summary

This chapter intentionally introduces only one new architectural concept:

Repository Resilience Manager.

The term should not be interpreted as a software component.

Instead, it represents the architectural responsibility for continuously maintaining repository resilience throughout the repository lifecycle.

The specific implementation remains entirely open and should naturally be determined according to the architectural principles, coding standards and long-term evolution of the Duplicati project.

Key Takeaway

Repository resilience should integrate into Duplicati as an architectural responsibility rather than as an isolated feature.

Users define resilience objectives.

Duplicati continuously maintains those objectives as the repository evolves.

Chapter 5 — Relationship with Existing Features

5.1 Introduction

Every mature software project gradually develops multiple layers of protection.

Each layer addresses a different class of problems while complementing the others.

Duplicati already provides a rich ecosystem of features designed to improve reliability, flexibility and long-term data preservation.

The purpose of this proposal is **not** to replace or redesign any of these existing mechanisms.

Instead, it introduces an additional architectural capability operating at a different level of the backup ecosystem.

Understanding this distinction is essential for correctly evaluating the scope of this proposal.

5.2 Complementing Existing Features

Repository resilience should be regarded as an additional architectural layer.

It neither competes with nor diminishes the value of the protection mechanisms already available within Duplicati and the surrounding storage ecosystem.

Each mechanism remains responsible for a different failure domain.

The resulting protection strategy becomes layered rather than redundant.

5.3 Multiple Backup Destinations

The recently introduced support for multiple backup destinations represents a major enhancement to Duplicati.

It enables a single backup job to maintain multiple independent repository instances, greatly simplifying implementation of widely recommended strategies such as the **3-2-1 backup rule**.

This proposal fully embraces that capability.

However, repository replication and repository resilience address different questions.

Multiple destinations answer:

> **“Does another complete repository exist?”**

Repository resilience instead asks:

> **“Can this specific repository continue to operate after losing a limited number of its own internal objects?”**

These objectives are complementary rather than competing.

Each replicated repository instance could itself benefit from intrinsic repository resilience.

5.4 The 3-2-1 Backup Strategy

The 3-2-1 strategy remains one of the most effective approaches for protecting data against catastrophic failures.

Maintaining multiple copies across different storage media and locations significantly reduces the risk of complete data loss.

Repository resilience should not be interpreted as an alternative to this strategy.

Instead, it strengthens each individual repository participating in that strategy.

In other words:

3-2-1 protects against losing repositories.

Repository resilience protects the integrity of each repository.

Together they create a more robust protection model.

5.5 RAID and Storage Redundancy

Hardware redundancy technologies such as RAID primarily protect storage devices against hardware failures.

Similarly, resilient filesystems provide protection against certain classes of storage corruption.

These technologies operate below the repository layer.

Repository resilience operates above them.

Consequently, both approaches remain valuable and complementary.

Storage redundancy protects the storage infrastructure.

Repository resilience protects the logical repository stored on that infrastructure.

5.6 Cloud Storage Replication

Cloud storage providers often provide excellent infrastructure availability and geographical redundancy.

These capabilities significantly reduce the probability of storage loss.

Nevertheless, they generally preserve the exact contents stored within the repository.

If repository objects become corrupted before synchronization, or if accidental deletions propagate across replicas, cloud replication alone cannot necessarily restore repository integrity.

Repository resilience addresses this different class of failures.

5.7 Repository Verification and Repair

Duplicati already provides verification and repair capabilities that help detect repository inconsistencies and recover from specific repository problems.

These tools remain essential components of repository maintenance.

Repository resilience should be regarded as complementary to them.

Verification determines whether problems exist.

Repair attempts to restore repository consistency.

Repository resilience aims to reduce the impact of certain failures before repair becomes necessary.

Together, these mechanisms provide stronger repository lifecycle management.

5.8 External Synchronization Tools

Many users replicate repositories using external synchronization software or filesystem-level replication.

These approaches remain excellent practices for increasing repository availability.

However, they replicate repository contents exactly as they exist.

Repository resilience instead concerns the internal ability of each replicated repository to tolerate limited object loss.

The two approaches therefore reinforce one another.

5.9 Layered Protection

The architectural vision proposed in this document can be summarized as a layered protection model.

Each layer contributes independently to the overall robustness of the backup ecosystem.

Protection Layer	Primary Objective
Off-site backup	Protect against site failures
Repository replication	Protect against repository loss
Storage redundancy	Protect against storage failures
Repository resilience	Protect against repository object failures

No single layer replaces another.

Instead, each contributes to reducing risk within its own failure domain.

5.10 Architectural Compatibility

One of the strengths of this proposal is that it requires no conceptual changes to the existing Duplicati philosophy.

Repositories continue to function exactly as they do today.

Backup workflows remain unchanged.

Storage backends remain unchanged.

Encryption remains unchanged.

Deduplication remains unchanged.

Repository resilience simply introduces an additional architectural capability that may be enabled when desired.

This evolutionary approach minimizes disruption while maximizing compatibility.

5.11 Summary

Repository resilience should not be viewed as an alternative to existing protection technologies.

Instead, it fills a gap between storage protection and repository replication by introducing intrinsic recoverability within each individual repository instance.

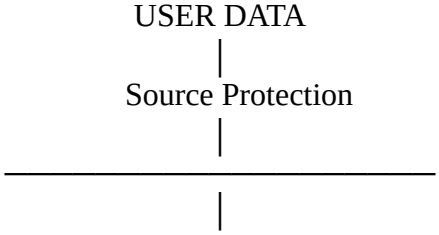
Rather than replacing existing best practices, it extends them.

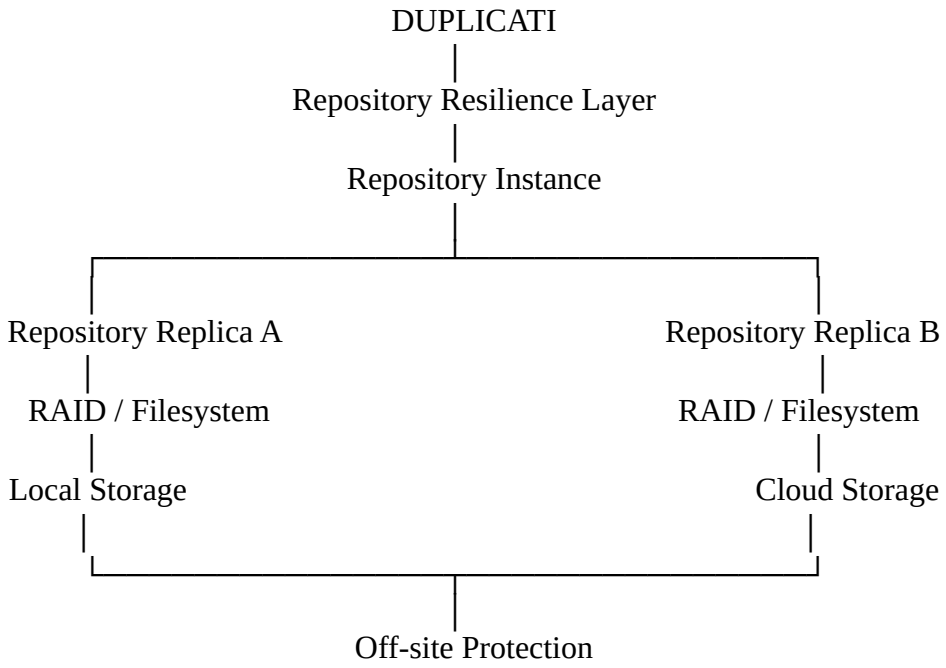
The proposal therefore strengthens the overall backup architecture while fully preserving the value of Duplicati’s existing capabilities.

Key Takeaway

Repository resilience complements existing protection mechanisms by addressing a failure domain that is currently outside the scope of storage redundancy, repository replication and disaster recovery strategies.

Its purpose is not to create additional copies of the repository, but to improve the survivability of each repository instance.




DUPLICATI
Repository Resilience Layer
Repository Instance
Repository Replica A
Repository Replica B
RAID / Filesystem
RAID / Filesystem
Local Storage
Cloud Storage
Off-site Protection

Chapter 6 — Behavioural Model

6.1 Introduction

The previous chapters introduced the motivation, architectural principles and integration strategy for repository resilience.

This chapter shifts the focus from architecture to behaviour.

Rather than describing implementation mechanisms, it explores how a resilient repository should behave throughout its entire lifecycle.

The objective is to define observable behaviours that remain valid regardless of the implementation technology eventually adopted.

6.2 The Repository as a Living System

Traditional backup repositories are often regarded as passive collections of storage objects.

This proposal suggests a different perspective.

A resilient repository should be considered an active system that continuously maintains its own recoverability.

As new repository objects are created, removed or reorganized, the repository should automatically preserve the resilience objective configured by the user.

Resilience therefore becomes a continuous property rather than a one-time operation.

6.3 Repository States

From a resilience perspective, every repository may be viewed as progressing through a number of logical states.

Initializing

The repository is being created.

Its resilience capability is progressively established.

The objective is to ensure that newly committed repository objects become part of the resilience domain as early as reasonably possible.

Protected

The configured resilience objective is fully satisfied.

The repository is capable of tolerating the level of object loss defined by its current resilience policy.

No user intervention is required.

Adapting

Repository evolution is temporarily modifying its internal structure.

Examples include:

- backup operations;
- compaction;
- retention cleanup;
- repository optimization.

During this phase, the resilience subsystem automatically adjusts itself in order to preserve the configured protection objective.

Degraded

One or more repository objects have become unavailable or corrupted.

Although the configured resilience objective may no longer be fully satisfied, the repository remains operational and recoverable.

The degradation should be measurable and clearly reported to the user.

Recovering

The repository is actively restoring its resilience capability.

Recovery may occur automatically or through explicit maintenance operations, depending on the implementation chosen.

The objective is to restore the configured resilience policy while minimizing disruption.

Healthy

A repository returns to the Healthy state once both its operational integrity and its configured resilience objective have been fully restored.

6.4 Continuous Protection

One of the most important behavioural objectives introduced by this proposal is continuous protection.

A repository should never remain permanently outside its configured resilience policy simply because repository evolution has not yet reached a predefined threshold.

In practical terms, repository protection should evolve together with repository growth.

The implementation strategy remains intentionally unspecified.

The behavioural expectation is simply that resilience should closely follow repository evolution.

6.5 Adaptive Protection

Repository resilience should continuously adapt to structural changes.

Adding new repository objects.

Removing obsolete objects.

Reorganizing repository contents.

Optimizing storage utilization.

Each of these events modifies the logical structure of the repository.

Rather than requiring explicit reconstruction of the resilience model, the repository should naturally adapt itself as part of normal operation.

6.6 Repository Health Evolution

Repository Health should represent more than a binary healthy/unhealthy indicator.

Instead, it should evolve dynamically as repository conditions change.

For example:

- resilience temporarily below the configured objective;
- pending optimization after compaction;
- degraded recoverability due to missing objects;
- restoration in progress.

This behavioural model provides users with a clearer understanding of repository status while avoiding unnecessary maintenance operations.

6.7 User Experience

The behavioural complexity described in this proposal should remain almost entirely invisible to users.

Users should continue interacting with Duplicati using familiar concepts.

From the user’s perspective, repository resilience should behave similarly to existing Duplicati capabilities:

configuration remains simple;

maintenance remains largely automatic;

advanced implementation details remain hidden.

The objective is to increase reliability without increasing operational complexity.

6.8 Failure Response

When repository object loss occurs, the repository should respond predictably.

Rather than immediately transitioning from a fully operational state to an unrecoverable condition, the repository should progressively degrade according to the resilience capability currently available.

Whenever possible, limited object loss should reduce resilience before reducing recoverability.

This behavioural distinction represents one of the fundamental objectives of the proposed framework.

6.9 Long-Term Evolution

Repositories often remain in service for many years.

During their lifetime they undergo thousands of backup operations, maintenance procedures and structural optimizations.

Repository resilience should therefore be regarded as a long-term architectural capability rather than as a temporary maintenance feature.

Its behaviour should remain stable regardless of repository age, storage backend or repository size.

6.10 Behaviour Before Implementation

One of the guiding ideas behind this proposal is that behavioural expectations should precede implementation decisions.

Different implementation techniques may exhibit different performance characteristics.

They may organize redundancy differently.

They may optimize storage differently.

Nevertheless, all implementations should preserve the same observable repository behaviour defined by this chapter.

This separation ensures that architectural consistency is maintained even as implementation techniques evolve over time.

6.11 Summary

This chapter intentionally defines how a resilient repository should behave rather than how resilience should be implemented.

The repository is viewed as a continuously evolving system that preserves its own recoverability while adapting to normal repository operations.

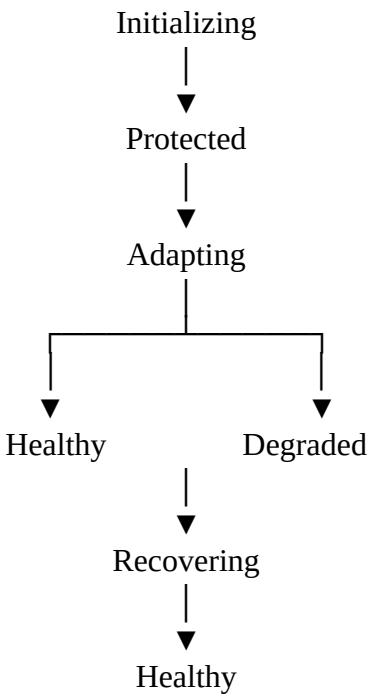
Implementation techniques remain entirely open.

Behavioural expectations remain stable.

Key Takeaway

Repository resilience should behave as a continuous architectural property of the repository lifecycle.

Users should experience a repository that automatically preserves its recoverability while evolving naturally over time, without requiring knowledge of the underlying resilience mechanisms.



Chapter 7 — Architectural Value and Expected Benefits

7.1 Introduction

Every architectural enhancement introduces both opportunities and responsibilities.

The purpose of this chapter is not to claim that repository resilience is universally beneficial, but rather to examine where it may provide meaningful value and under which circumstances.

The proposal should therefore be evaluated not only in terms of its technical feasibility, but also in terms of the architectural capabilities it may introduce into the Duplicati ecosystem.

7.2 A New Architectural Property

Most existing Duplicati features extend the system by introducing new functionality.

Repository resilience is different.

Rather than adding a new operation, command or maintenance task, it introduces a new architectural property.

Once enabled, resilience becomes part of the repository's normal behaviour throughout its entire lifecycle.

From this perspective, repository resilience resembles properties such as encryption or deduplication more than it resembles a maintenance feature.

It continuously influences how the repository behaves without requiring continuous user interaction.

7.3 Increased Repository Survivability

The most immediate benefit of repository resilience is improved survivability.

Limited loss of repository objects would no longer necessarily imply immediate loss of repository usability.

Instead, the repository could continue operating while its resilience capability progressively decreases.

This behavioural change represents a significant conceptual improvement.

Recoverability becomes a gradual property rather than an all-or-nothing condition.

7.4 Improved Operational Confidence

Many Duplicati users store repositories on external disks, NAS devices or cloud storage providers for extended periods.

Although these storage solutions are generally reliable, they remain susceptible to accidental deletion, silent corruption and operational mistakes.

Knowing that each repository instance possesses an intrinsic capability to tolerate limited object loss may increase user confidence, particularly for long-term archival repositories.

This proposal therefore aims not only to improve recoverability, but also to improve trust in repository preservation.

7.5 Long-Term Repository Preservation

Repositories often outlive the original systems that created them.

Hardware changes.

Operating systems evolve.

Applications disappear.

Source data may eventually no longer exist.

At that stage, the repository itself becomes the digital asset requiring preservation.

Repository resilience supports this long-term perspective by treating the repository as an object worthy of preservation in its own right.

7.6 Separation of Responsibilities

Another architectural benefit introduced by this proposal is a clearer separation of protection responsibilities.

Storage technologies remain responsible for storage reliability.

Replication remains responsible for repository availability.

Disaster recovery remains responsible for geographical protection.

Repository resilience becomes responsible for the internal recoverability of each repository instance.

This separation avoids unnecessary overlap while strengthening the overall protection model.

7.7 Simplicity for Users

One of the guiding objectives of this proposal is to improve reliability without increasing complexity.

Users should continue configuring backup jobs using familiar concepts.

Rather than selecting coding algorithms or redundancy strategies, they simply choose an appropriate resilience policy.

Duplicati remains responsible for translating that policy into implementation decisions.

This approach preserves usability while allowing implementation strategies to evolve over time.

7.8 Flexibility for Maintainers

Because this proposal intentionally focuses on architectural behaviour rather than implementation techniques, it leaves substantial freedom to future development.

Maintainers remain free to evaluate different algorithms, metadata formats and optimization strategies.

As long as the observable architectural properties remain consistent, implementation details may evolve independently.

This flexibility helps protect the proposal against technological obsolescence.

7.9 Incremental Adoption

Repository resilience is intentionally conceived as an optional capability.

Existing repositories continue functioning without modification.

Users who value the additional protection may enable repository resilience when appropriate.

This gradual adoption model minimizes migration barriers and allows the feature to mature over time through practical experience and community feedback.

7.10 Community Value

Architectural discussions often generate value beyond the specific feature being proposed.

Even if repository resilience were ultimately implemented differently—or not implemented at all—the concepts introduced by this proposal may still contribute to future discussions about repository evolution, long-term preservation and recoverability.

For this reason, the proposal should be viewed not only as a feature suggestion, but also as a contribution to the ongoing architectural dialogue surrounding Duplicati.

7.11 Recognizing the Trade-offs

No architectural enhancement comes without cost.

Repository resilience would almost certainly introduce additional storage overhead, computational work and implementation complexity.

These trade-offs are both expected and legitimate.

The purpose of this proposal is not to deny their existence, but to suggest that, for some users and some use cases, the additional resilience may justify those costs.

Determining the appropriate balance remains a design decision for the Duplicati maintainers.

7.12 Summary

Repository resilience introduces value not merely by increasing redundancy, but by extending the architectural capabilities of the repository itself.

Its principal contribution lies in shifting repository recoverability from an implicit consequence of external replication to an intrinsic property of each repository instance.

Whether this evolution proves desirable is naturally a matter for discussion within the Duplicati community.

Key Takeaway

The primary value of repository resilience is architectural rather than algorithmic.

It introduces a new property—intrinsic recoverability—that complements Duplicati’s existing strengths while preserving its philosophy of simplicity, flexibility and user control.

Chapter 8 — User Experience and Configuration Model

8.1 Introduction

One of the guiding principles of this proposal is that repository resilience should improve reliability without increasing user complexity.

Most Duplicati users are interested in protecting their data, not in understanding redundancy algorithms, coding theory or storage mathematics.

For this reason, repository resilience should be presented as a high-level repository capability rather than as a collection of technical configuration parameters.

The architectural sophistication should remain largely invisible.

8.2 Policy-Oriented Configuration

Users should not configure redundancy mechanisms.

Instead, they should express a resilience objective.

Examples of possible resilience policies include:

- Disabled
- Standard
- Enhanced
- Maximum
- Custom

Each policy represents a desired level of repository resilience rather than a specific implementation strategy.

Duplicati remains responsible for selecting the appropriate internal mechanisms required to satisfy the chosen policy.

This separation preserves simplicity while allowing implementation techniques to evolve independently.

8.3 Advanced Configuration

Although simplicity should remain the default, advanced users may require greater control.

A Custom Resilience Policy could therefore expose higher-level objectives such as:

- desired resilience level;
- acceptable storage overhead;

- preferred optimisation strategy;
- recovery priority.

Even in advanced mode, users should continue expressing goals rather than implementation details.

Parameters such as coding algorithms, parity layouts or redundancy structures should remain internal implementation decisions.

8.4 Repository Health Dashboard

Repository resilience should be observable.

Users should be able to understand the current protection status of their repository at a glance.

The proposal therefore introduces the concept of a Repository Health Dashboard.

Possible information presented may include:

- configured resilience policy;
- current resilience status;
- repository integrity;
- recoverability status;
- pending optimisation activities;
- resilience maintenance recommendations.

The precise presentation remains entirely open.

Only the architectural concept is important.

8.5 Health Indicators

Repository Health should communicate meaningful information without overwhelming users.

Possible health states might include:

- Healthy
- Protected
- Adapting
- Degraded
- Recovering
- Attention Required

These states intentionally describe repository behaviour rather than implementation details.

Users should understand the practical meaning of each state without requiring technical knowledge of the underlying resilience mechanisms.

8.6 Progressive Disclosure

Different users require different levels of information.

For most users, a simple indication that the repository is healthy should be sufficient.

Advanced users, however, may wish to inspect additional diagnostic information.

Repository resilience should therefore follow a principle of progressive disclosure.

Simple information remains visible by default.

Detailed technical information becomes available only when explicitly requested.

This approach preserves usability while satisfying the needs of experienced users.

8.7 Behaviour During Backup Operations

Repository resilience should integrate naturally into existing backup workflows.

Enabling repository resilience should not require users to modify their normal operating procedures.

During backup operations, Duplicati should automatically maintain the configured resilience objective while minimising disruption to normal repository activity.

Whenever resilience optimisation is temporarily incomplete, repository health should accurately reflect the current situation.

8.8 Behaviour After Repository Changes

Repositories evolve continuously.

Backup operations create new repository objects.

Compaction reorganises storage.

Retention policies remove obsolete data.

Users should not be expected to manually restore repository resilience after each of these operations.

Instead, resilience maintenance should occur automatically as part of normal repository evolution.

From the user’s perspective, repository resilience should feel like a continuously maintained capability rather than an occasional maintenance task.

8.9 Behaviour During Object Loss

One of the most visible benefits of repository resilience appears when repository objects become unavailable.

Rather than immediately reporting an unrecoverable repository, Duplicati should communicate the current repository health in a clear and understandable manner.

Possible examples include:

- repository remains fully recoverable;
- resilience temporarily reduced;
- maintenance recommended;
- resilience objective restored.

The objective is to communicate practical consequences rather than internal implementation details.

8.10 User Confidence

An often overlooked aspect of backup software is user confidence.

Users rarely inspect repository internals.

Instead, they trust that the repository will remain usable when restoration eventually becomes necessary.

Repository resilience contributes to this confidence by making repository health observable and by reducing the consequences of limited object loss.

In this sense, repository resilience improves not only recoverability, but also the perceived reliability of long-term repository preservation.

8.11 Consistency with Duplicati Philosophy

The configuration model proposed in this document intentionally follows the design philosophy already present throughout Duplicati.

Users define objectives.

Duplicati determines implementation.

Examples already exist in several areas of the software:

- compression;
- block deduplication;
- repository organisation;
- storage abstraction.

Repository resilience simply extends this philosophy to repository recoverability.

8.12 Summary

Repository resilience should remain simple to configure, easy to understand and largely automatic.

The user’s responsibility ends with defining the desired resilience objective.

Duplicati assumes responsibility for continuously maintaining that objective while presenting meaningful repository health information throughout the repository lifecycle.

Key Takeaway

Users should configure **how resilient they want their repository to be**, not **how resilience is implemented**.

The complexity belongs inside the software.

The confidence belongs to the user.

Chapter 9 — Design Considerations and Implementation Flexibility

9.1 Introduction

Throughout this proposal, implementation details have intentionally been kept separate from architectural principles.

This distinction is deliberate.

Architectural properties should remain stable over time, while implementation techniques may evolve as new technologies, algorithms and design approaches become available.

The purpose of this chapter is therefore not to recommend a particular implementation strategy, but to identify the qualities that any implementation should ideally preserve.

9.2 Preserving Architectural Freedom

One of the primary goals of this proposal is to avoid constraining future development.

Different implementation strategies may offer different advantages in terms of performance, storage efficiency, computational complexity or maintainability.

For this reason, the proposal intentionally refrains from recommending a specific redundancy algorithm or repository format.

Instead, it defines behavioural objectives while leaving implementation choices entirely to the Duplicati maintainers.

This approach preserves architectural freedom without sacrificing conceptual consistency.

9.3 Evolution Rather Than Replacement

Repository resilience should be regarded as an evolutionary enhancement of the existing repository architecture.

The proposal does not require a redesign of Duplicati’s core concepts.

Existing repository formats, storage backends and operational workflows may continue to function as they do today.

Wherever possible, repository resilience should integrate naturally into the current architecture rather than introducing parallel subsystems or incompatible repository models.

This evolutionary approach reduces adoption barriers and preserves long-term maintainability.

9.4 Independence from Specific Algorithms

Several technologies could potentially provide the recoverability objectives described in this proposal.

Examples might include erasure coding, parity-based techniques, forward error correction or future approaches not yet widely adopted.

The architectural model presented here intentionally remains independent of any particular algorithm.

This separation ensures that improvements in coding techniques can be adopted over time without altering the architectural principles established by this proposal.

9.5 Scalability

Repository resilience should remain applicable across a wide range of repository sizes.

Small personal repositories.

Large archival repositories.

Cloud-hosted repositories.

Repositories containing millions of blocks.

Although implementation strategies may differ, the conceptual model should remain consistent.

Users should interact with repository resilience in the same way regardless of repository scale.

9.6 Incremental Maintenance

Wherever practical, repository resilience should evolve incrementally together with repository activity.

Large-scale reconstruction procedures should ideally become exceptional rather than routine.

Incremental maintenance may reduce computational overhead, minimise operational disruption and improve long-term scalability.

Whether this objective is achieved immediately, periodically or through alternative techniques remains an implementation decision.

9.7 Compatibility with Repository Evolution

Duplicati continues to evolve.

Repository formats may change.

Metadata structures may be refined.

Storage optimisation techniques may improve.

Repository resilience should therefore be designed as a concept capable of evolving together with the repository rather than depending on fixed internal structures.

This flexibility increases the long-term sustainability of the proposal.

9.8 Extensibility

Although this proposal focuses primarily on repository object recoverability, the underlying architectural model may naturally support future extensions.

Potential future developments might include:

- enhanced repository diagnostics;
- richer repository health metrics;
- resilience analytics;
- predictive maintenance;
- repository ageing analysis.

These possibilities should be regarded as future opportunities rather than current objectives.

The proposal intentionally keeps its present scope limited to intrinsic repository resilience.

9.9 Implementation Neutrality

Different implementations may optimise different aspects of repository behaviour.

Some may prioritise storage efficiency.

Others may favour computational simplicity.

Others may optimise recovery speed.

The architectural model introduced by this proposal intentionally accommodates all of these possibilities.

Its objective is not to prescribe optimisation priorities, but to establish a stable behavioural framework within which different implementation strategies may coexist.

9.10 Community Collaboration

This proposal should be regarded as the beginning of a technical discussion rather than the conclusion of one.

Maintainers may identify constraints not considered here.

Contributors may propose more effective implementation strategies.

Users may identify additional use cases.

Such collaboration represents one of the principal strengths of open-source software development.

For this reason, architectural flexibility has intentionally been prioritised over implementation specificity.

9.11 Summary

The long-term value of this proposal lies in the distinction it establishes between architectural objectives and implementation techniques.

By defining desired repository behaviour while avoiding unnecessary implementation constraints, the proposal aims to remain relevant even as Duplicati continues to evolve.

This flexibility is intended not as a lack of specificity, but as a conscious architectural choice.

Key Takeaway

This proposal defines **what a resilient repository should achieve**, not **how resilience must be implemented**.

Preserving implementation freedom is considered an essential architectural principle rather than a limitation of the proposal.

Chapter 10 — Discussion, Trade-offs and Open Questions

10.1 Introduction

Every architectural enhancement involves trade-offs.

Repository resilience is no exception.

The purpose of this proposal is not to suggest that intrinsic repository resilience should necessarily become part of Duplicati.

Rather, it aims to encourage discussion around a possible architectural direction whose potential benefits may justify further exploration.

This chapter intentionally identifies questions, trade-offs and design considerations that deserve careful evaluation before any implementation could be considered.

10.2 Storage Overhead

Any mechanism capable of improving repository recoverability is expected to introduce additional storage requirements.

The exact overhead depends entirely on the implementation eventually chosen.

Different coding strategies may optimise for different objectives.

Some may prioritise storage efficiency.

Others may favour faster recovery or computational simplicity.

For this reason, this proposal deliberately avoids prescribing acceptable overhead values.

Instead, it recognises storage consumption as one of the principal design parameters that future implementations would need to balance.

10.3 Computational Cost

Maintaining repository resilience inevitably requires additional processing.

Depending on the implementation, this processing may occur:

- during backup operations;
- during repository maintenance;
- incrementally over time;
- or through a combination of these approaches.

Each strategy presents different advantages and disadvantages.

The proposal intentionally leaves these optimisation decisions open.

10.4 Repository Complexity

Introducing intrinsic repository resilience would increase the conceptual complexity of repository management.

Additional metadata.

Additional maintenance logic.

Additional recovery procedures.

These aspects should not be underestimated.

Any implementation should therefore strive to minimise operational complexity while preserving the simplicity that characterises Duplicati.

10.5 Backward Compatibility

Backward compatibility remains one of the most important architectural considerations.

Existing repositories should continue functioning exactly as they do today.

Repository resilience should remain optional.

Migration, if required, should be predictable, incremental and fully under user control.

Maintaining compatibility with existing backup workflows should remain a fundamental objective.

10.6 Performance Considerations

Repository resilience should not unnecessarily compromise normal backup performance.

Different implementation strategies may choose different optimisation priorities.

For example:

- minimising CPU usage;
- reducing storage overhead;
- accelerating recovery;
- balancing background maintenance activities.

Determining the most appropriate balance remains an implementation decision beyond the scope of this proposal.

10.7 Repository Growth

Repositories naturally evolve over time.

Consequently, repository resilience must remain effective regardless of repository size or age.

Large archival repositories present different operational characteristics compared with newly created repositories.

Any implementation should therefore scale gracefully while preserving the behavioural principles described throughout this document.

10.8 Interaction with Existing Maintenance Operations

Repository resilience would inevitably interact with existing maintenance procedures.

Repository compaction.

Verification.

Repair.

Retention cleanup.

These interactions require careful architectural consideration.

The proposal intentionally avoids prescribing how these integrations should be realised, recognising that they depend heavily on Duplicati’s internal architecture.

10.9 Questions for Discussion

Rather than attempting to answer every technical question, this proposal intentionally leaves several areas open for community discussion.

Examples include:

- Which implementation strategy best satisfies the architectural objectives?
- How should repository health be presented to users?
- What degree of configurability is appropriate?
- Should repository resilience be enabled by default for new repositories?
- Which repository objects should participate in resilience protection?
- How should maintenance operations preserve the configured resilience policy?

These questions are intended to stimulate constructive discussion rather than to define requirements.

10.10 Community Evaluation

The ultimate value of this proposal depends upon community feedback.

Maintainers may identify architectural constraints.

Contributors may propose alternative approaches.

Users may contribute additional real-world scenarios.

This collaborative evaluation is one of the principal strengths of the open-source development model.

For this reason, the proposal should be regarded as the beginning of a conversation rather than as a completed specification.

10.11 Scope Boundaries

It is equally important to define what this proposal intentionally does **not** attempt to address.

This proposal does not seek to:

- replace repository replication;
- replace RAID or resilient filesystems;
- replace disaster recovery strategies;
- redefine repository formats;
- prescribe redundancy algorithms;
- introduce mandatory architectural changes.

Maintaining these boundaries helps preserve the focus of the discussion and avoids unnecessary overlap with existing Duplicati capabilities.

10.12 Summary

Architectural proposals become stronger when they openly acknowledge their own uncertainties.

Repository resilience presents both opportunities and challenges.

The objective of this proposal is not to minimise those challenges, but to provide a structured framework within which they can be evaluated.

Ultimately, determining whether repository resilience represents a worthwhile architectural evolution remains a decision for the Duplicati maintainers and community.

Key Takeaway

Repository resilience should be evaluated as an architectural opportunity rather than as a predefined solution.

Its potential value lies not in eliminating trade-offs, but in offering a new architectural capability whose benefits may justify careful consideration.

Chapter 11 — Illustrative Operational Model

11.1 Introduction

The previous chapters intentionally focused on architectural principles and behavioural objectives while avoiding implementation details.

This chapter takes a different approach.

It presents one possible operational interpretation of those principles.

The purpose is purely illustrative.

Nothing described in this chapter should be interpreted as a recommended implementation or as a requirement for future development.

Instead, the examples simply demonstrate how the architectural concepts introduced throughout this proposal could naturally translate into repository behaviour.

11.2 Repository Objects

For the purpose of illustration, a Duplicati repository may be viewed as a collection of repository objects.

Current examples include:

- .dblock
- .dindex
- .dlist

Future repository formats may introduce additional object types.

From an architectural perspective, all repository objects represent persistent components of repository state.

Repository resilience therefore concerns the preservation of repository objects rather than the preservation of any particular file format.

This distinction allows the architectural model to remain valid even as repository formats evolve.

11.3 Resilience Metadata

One possible implementation strategy could introduce dedicated resilience metadata describing the current protection state of repository objects.

Such metadata might record information including:

- current resilience policy;

- protected object set;
- repository health status;
- recoverability information;
- pending resilience maintenance.

The specific representation of this information remains entirely open.

Its purpose is simply to allow the repository to evaluate and maintain its own resilience objectives.

11.4 Illustrative Recovery Objects

One possible implementation could introduce additional repository objects dedicated to recoverability.

For the purposes of illustration, these may be referred to as **Recovery Objects**.

Their internal organisation, storage format and coding strategy are intentionally left unspecified.

Recovery Objects would not represent user data.

Instead, they would contribute to preserving the recoverability of repository objects.

Whether such objects ultimately exist—or whether resilience is achieved through entirely different techniques—remains an implementation decision.

11.5 Continuous Protection

One of the central ideas introduced during the development of this proposal is that newly created repository objects should not remain permanently outside the configured resilience domain.

Whenever reasonably practical, protection should begin as soon as repository evolution allows.

The architectural objective is therefore continuous protection rather than delayed protection.

Exactly how this objective is achieved remains intentionally unspecified.

11.6 Minimum Resilience Objective

Repository resilience should be driven by policies rather than by implementation thresholds.

For example, a repository configured with a resilience objective equivalent to approximately 20% redundancy should continuously strive to maintain that objective throughout its lifetime.

As new repository objects are created, resilience maintenance should adapt accordingly.

The implementation should not require repository growth to reach predetermined object counts before any protection becomes available.

The configured resilience objective should instead be regarded as a continuously maintained property of the repository.

11.7 Adaptive Repository Evolution

Repositories naturally expand and contract over time.

Backup operations introduce new repository objects.

Retention policies remove obsolete objects.

Compaction reorganises repository contents.

The resilience model should evolve together with these changes.

Rather than rebuilding repository protection from scratch after each modification, resilience should adapt incrementally whenever practical.

This adaptive behaviour reflects the architectural principles established in previous chapters.

11.8 Illustrative Example

Consider a repository configured with a resilience policy corresponding to an approximate protection objective of twenty percent.

Initially, the repository contains only a single repository object.

Even at this early stage, the repository should begin moving towards its configured resilience objective rather than postponing protection until a larger number of repository objects has accumulated.

As additional repository objects are created, resilience maintenance continues adapting automatically.

At every stage of repository growth, the repository strives to remain as close as reasonably possible to its configured resilience objective.

The precise implementation strategy remains entirely open.

11.9 Behaviour During Object Loss

Suppose that one or more repository objects become unavailable due to accidental deletion or local storage corruption.

Repository Health would immediately reflect the reduced resilience state.

If the configured resilience objective still permits recovery, the repository remains operational while initiating resilience restoration according to its maintenance strategy.

Only after the available resilience capacity has been exhausted would repository recoverability become compromised.

This behaviour illustrates the distinction between resilience degradation and repository failure introduced earlier in this proposal.

11.10 Policy-Driven Operation

Throughout repository evolution, implementation mechanisms remain subordinate to policy objectives.

Users define the resilience policy.

Duplicati continuously evaluates repository state.

Whenever repository evolution causes resilience to fall below the configured objective, appropriate maintenance operations restore the desired protection level.

This policy-driven approach preserves both implementation flexibility and a consistent user experience.

11.11 Summary

The operational examples presented in this chapter intentionally remain illustrative.

Their purpose is not to define implementation details but to demonstrate that the architectural principles established throughout this proposal naturally lead to coherent repository behaviour.

Alternative implementations may differ substantially while preserving the same architectural objectives.

Key Takeaway

The proposed resilience model is **policy-driven**, **continuously adaptive** and **implementation-independent**.

The examples presented in this chapter illustrate one possible operational interpretation of these principles without constraining future development.

Chapter 12 — Conclusion

12.1 Introduction

This document has presented a proposal for introducing intrinsic repository resilience as a new architectural capability within Duplicati.

The proposal does not define a specific implementation.

It does not prescribe a redundancy algorithm.

It does not require a particular repository format.

Instead, it defines a vision:

A repository should not only store backup data.

A repository should also possess the capability to preserve its own recoverability.

12.2 From Storage Container to Resilient Repository

Traditional backup architectures often consider the repository as a passive storage container.

Data is written.

Data is replicated.

Data is restored when required.

This proposal introduces a different perspective.

A repository is not merely a collection of files.

It is a structured digital asset whose integrity and recoverability deserve continuous preservation.

Repository resilience represents the evolution from a passive storage model towards a more autonomous preservation model.

12.3 Complementing Existing Protection Strategies

Intrinsic repository resilience does not replace existing backup best practices.

It does not replace:

- multiple backup destinations;
- the 3-2-1 strategy;
- storage redundancy;
- disaster recovery planning;
- external replication.

Instead, it complements them.

Each protection layer addresses a different failure domain.

Together, these layers create a stronger and more comprehensive protection ecosystem.

12.4 The Core Principle

The central idea introduced by this proposal can be summarized as follows:

> A repository should continuously maintain its configured resilience objective throughout its entire lifecycle.

This principle intentionally remains independent from implementation.

Whether achieved through parity, erasure coding, future technologies or alternative mechanisms, the architectural objective remains the same.

12.5 A User-Centric Vision

From the user’s perspective, repository resilience should remain simple.

Users should not need to understand:

- coding algorithms;
- redundancy structures;
- recovery mathematics;
- repository internals.

They should only define how resilient they want their repository to be.

Duplicati should manage the complexity internally while providing clear visibility into repository health.

12.6 An Invitation to Discussion

This proposal is intentionally presented as an architectural discussion rather than a finished technical specification.

Its purpose is to encourage evaluation, refinement and collaboration.

The Duplicati maintainers and community are best positioned to determine:

- whether this capability aligns with the project’s long-term vision;
- how it could integrate with the existing architecture;
- which implementation strategies may be appropriate.

12.7 Final Perspective

Backup software exists because data has value.

However, the repository that preserves that data also becomes valuable over time.

As repositories grow larger, older and more independent from their original sources, their ability to preserve themselves becomes increasingly important.

Repository resilience proposes a simple conceptual evolution:

From:

> “The repository stores the backup.”

Towards:

> “The repository participates actively in preserving the backup.”

This document proposes that intrinsic repository resilience may represent a natural future capability for advanced backup preservation systems.

Final Key Takeaway

Repository resilience is not about creating another copy of the backup.

It is about making each repository instance stronger, more aware and more capable of surviving the limited failures that inevitably occur over long periods of time.

Appendix A — Illustrative Example of Repository Resilience

A.1 Purpose

This appendix provides a simplified example intended only to clarify the mental model behind repository resilience.

The example does not represent a proposed implementation.

It does not define file formats, algorithms or internal mechanisms.

Its only purpose is to illustrate the expected behaviour of a repository configured with a resilience objective.

A.2 Initial Repository State

Assume a repository configured with:

Resilience Policy:
Custom

Minimum Resilience Objective:
20%
The repository currently contains:

Data Objects:

D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
The repository also maintains an internal resilience capability sufficient to tolerate limited object loss.

The exact mechanism is irrelevant.

The important property is:

The repository can recover from a limited number of missing objects.

A.3 Repository Growth

A new backup operation creates:

D11
A resilient repository should not leave this object permanently outside the protection domain.

The desired behaviour is:

Before:

D1-D10 protected

After:

D1-D11 repository continuously adapting
The repository should automatically maintain the configured resilience objective as it evolves.

A.4 Object Loss Scenario

Assume accidental deletion or corruption of:

D5
The repository state becomes:

Repository Health:

Integrity:
Degraded

Recoverability:
Maintained

Resilience:
Below configured objective
The repository is not considered fully protected anymore.

However, it is not necessarily lost.

This distinction is fundamental.

A.5 Resilience Restoration

The repository continues operating.

During normal maintenance activities, it restores its resilience level.

The conceptual evolution is:

Configured objective:

20%

↓

Object loss:

Reduced resilience

↓

Maintenance:

Restoring resilience

↓

Target achieved:

20%
The implementation mechanism remains completely independent from this behavioural model.

A.6 Conceptual Interpretation

The important idea is not the existence of a specific recovery file or redundancy algorithm.

The important idea is:

The repository continuously strives to preserve its configured ability
to recover from limited internal failures.

This is the essence of repository resilience.

Appendix B — Possible Technical Directions

B.1 Purpose

This appendix explores possible technical directions that could theoretically support repository resilience.

These examples are provided exclusively as discussion material.

They are not requirements.

They are not implementation recommendations.

The final technical approach should be determined by the Duplicati maintainers according to architectural constraints and engineering priorities.

B.2 Redundancy-Based Approaches

One possible direction is the use of redundancy mechanisms capable of reconstructing missing repository objects.

Examples include:

- parity-based techniques;
- erasure coding;
- forward error correction.

Such approaches may provide the ability to recover missing repository objects without requiring access to the original source data.

B.3 Recovery Objects

A possible implementation model could introduce dedicated recovery objects.

These objects would not contain user data.

Their purpose would be to preserve sufficient information to reconstruct repository components when required.

The organisation, naming and storage format of such objects would remain implementation decisions.

B.4 Dynamic Resilience Groups

One possible strategy could involve dynamically managed groups of repository objects.

The objective would be to maintain a configured resilience ratio while allowing the repository to continuously grow.

Unlike fixed groups created only after a certain amount of data accumulation, a dynamic approach would aim to ensure that new repository objects progressively enter the resilience domain as soon as practical.

B.5 Minimum Resilience Policy

A policy-based model could allow users to specify a minimum resilience objective.

For example:

Minimum Repository Resilience:

20%

The repository would then continuously adapt its internal resilience structures to maintain that target.

The percentage itself is only an example.

The appropriate policy model would require further evaluation.

B.6 Verification and Repair Integration

Repository resilience could potentially integrate with existing verification and repair workflows.

Verification could evaluate:

- repository integrity;
- resilience status;
- recoverability state.

Repair operations could potentially restore both repository consistency and resilience level.

The exact integration approach would depend on the final architecture.

B.7 Technology Evolution

Storage and coding technologies continue to evolve.

A resilience architecture should therefore avoid dependency on a single algorithm.

The long-term objective should be preserving the capability:

Repository can recover from limited internal object failures.

The technology used to achieve this capability may change over time.

B.8 Closing Note

The technical directions presented here are intentionally exploratory.

Their purpose is to demonstrate that the architectural principles described in this document can translate into practical engineering approaches.

The final implementation, if pursued, should emerge from collaborative design, experimentation and evaluation by the Duplicati community.