Politecnico di Torino

Politecnico di Torino

CYBERSECURITY

A.a. 2025/2026

Graduation Session July 2026

Open-Source Solutions for Centralized Identity and Access Management

Implementation with Ansible and Podman on a Containerized Samba AD DC

Relatori:

Fulvio Valenza

Davide Piumatti

Andrea Galletto

Candidato:

Simone Scirpoli

Abstract

For years the Laboratorio di Informatica (LABinf) at the Politecnico di Torino ran on a Samba NT4 domain sitting on top of an OpenLDAP directory. That single setup handled authentication, file sharing, printing and FTP access for around ninety workstations, and for the most part it worked. The problems were underneath. User provisioning was split across separate, non-atomic steps with no real recovery if one failed; the management tools were built on smbldap-tools, which has been effectively unmaintained for a long time; old security workarounds from past Samba upgrades were still sitting in the production configuration; and nothing was isolated, since every service shared the same kernel and process tree on an operating system that was already well past its expiry date. This thesis describes the design and implementation of a test environment for migrating those services from the legacy NT4 model to a Samba Active Directory deployment. The first step was less about technology and more about figuring out what the existing system actually did. I went through the services in use, the management scripts, and the constraints coming from a mixed client base where Windows and Linux machines have to authenticate against the same identity source without anyone noticing the difference. From there the new architecture took shape: three Podman containers on a single host, one acting as the Active Directory Domain Controller, one as a Samba member file server, and one running NFS. The entire deployment is provisioned with Ansible, which means the infrastructure can be torn down and rebuilt instead of being patched by hand, and the configuration lives in version control rather than in someone's memory. Keeping the existing POSIX identities intact during the cutover turned out to be the part that needed the most care. UID and GID values are written straight into Active Directory as RFC 2307 attributes at account creation, then read back consistently by Winbind on the SMB side and by SSSD on Linux clients. The authentication paths differ by platform, Kerberos for domain-joined Windows machines, LDAP over STARTTLS for Linux logins, but they resolve to the same numbers, which is what makes the shared NFSv4 exports behave correctly. The daily operations that smbldap-tools used to cover were rewritten as a set of Python scripts targeting the new domain. The environment was then put through a series of functional and resilience tests: domain health, password policy, login from both platforms, end-to-end UID consistency across SMB and NFS, and the idempotency of the Ansible provisioning. The thesis ends with what the migration actually taught me about moving a production-like laboratory onto a containerized, automation-driven identity platform, including the things that only became obvious once the containers were running.

Summary

Context and motivation

The Laboratorio di Informatica (LABinf) of the Politecnico di Torino provides authentication, file sharing, printing and FTP access to roughly ninety workstations. For years this rested on a Samba NT4 domain backed by an OpenLDAP directory. The setup worked, in the sense that students logged in and found their files, but it had aged badly underneath.

User provisioning was the clearest symptom. Creating an account meant separate, non-atomic operations against LDAP and Samba with no recovery path if one of them failed halfway. The tooling around it was built on smbldap-tools, a project that has been effectively unmaintained for years. On top of that, the production configuration still carried workarounds inherited from past Samba upgrades: NTLMv1 had been re-enabled and SMB pinned to the NT1 dialect to keep older Windows clients working, and a known remote code execution issue (CVE-2017-7494) was sitting unpatched, documented in the live `smb.conf` with a comment to that effect. Nothing was isolated either. Every service ran on the same kernel and process tree, on a Debian release that was already well past its supported life. The migration was therefore not an in-place upgrade but a cutover: a new domain, a new directory backend, and a new operating model built alongside the old one.

Goals

The target was a modern Samba Active Directory deployment that fixed the structural problems rather than papering over them. Three requirements drove the design. Services had to be isolated from one another, so that a fault or compromise in one did not put the whole host at risk. The infrastructure had to be reproducible, described as code rather than configured by hand and recovered from memory. And every user had to find their files again on the other side of the cutover, with ownership intact.

That last point deserves emphasis. The file trees on disk are owned by numeric UIDs and GIDs, and the migration does not try to carry those numbers across. Each account is recreated with a fresh sequential UID, and its home directory, identified by the matriculation number, is re-owned to the new value. The stable key is therefore the matricola and the folder that bears its name, not the UID, and keeping that link intact, so that a returning student lands back on their own files, was the hard constraint the import had to satisfy.

Proposed architecture

The new environment is built from three Podman containers running on a single host, `cclix1`. One container is the Active Directory Domain Controller, one is a Samba member server acting as the file server, and one runs NFS. The two Samba containers sit on separate internal networks and are published to the laboratory through specific host addresses and ports, so the DC and the file server are reachable where the clients expect them without exposing everything.

Ansible owns the state of this infrastructure. The containers, their networks, the Samba configuration and the supporting files are all generated from a single set of variables, which means the deployment can be destroyed and rebuilt instead of being repaired in place. Day-to-day user operations, the things smbldap-tools used to handle, were not folded into Ansible. They were rewritten as a family of Python scripts that talk to the new domain directly, because account creation and password changes are routine actions and do not belong in an infrastructure-provisioning tool.

Key technical decisions

The choice that makes everything else hold together is how identities are mapped. UID and GID values are written straight into Active Directory as RFC 2307 attributes at the moment an account is created, using `idmap_ad` rather than the algorithmic `idmap_rid` backend. The distinction matters: `idmap_rid` would derive each number from the object's RID rather than letting the administrator set it explicitly, leaving the value hostage to the directory's internal allocation across a reprovision. By storing them explicitly, the same `uidNumber` is read back by every component that needs it.

Those components differ by platform. On the SMB side Winbind resolves the RFC 2307 attributes for the file server; on Linux clients SSSD reads the same attributes over LDAP. The authentication paths are not identical either, domain-joined Windows machines go through Kerberos, while Linux logins use LDAP over STARTTLS, but both resolve to the same numeric identity. This is what lets

the NFSv4 exports work correctly: NFS authorises purely on UID and GID, so correctness depends entirely on every client agreeing on the numbers, which they do because they all read them from the same object. The file server and the NFS server share a single physical directory tree, exported once and reachable both over SMB and over NFS.

Provisioning the domain controller for the first time needed some care, since `samba-tool domain provision` cannot run against an already-running `samba` process. The container is started idle, the domain is provisioned, and the container is then recreated with Samba as its main process. To keep this idempotent, the playbook checks for the presence of the provisioned database before acting, so a second run does not wipe an existing domain. Small detail, but it is the difference between a playbook you can re-run safely and one you cannot.

Validation

The environment was tested rather than just assembled. The checks covered the health of the domain controller, enforcement of the password policy, and login from both a Windows and a Linux client against the new domain. The central test was end-to-end identity consistency: confirming that a user receives the same UID when seen through SMB and through NFS, and that file ownership therefore stays coherent across both protocols. The idempotency of the Ansible provisioning was verified by running it repeatedly and confirming it converged without damaging existing state, and a set of resilience checks looked at how the deployment behaved when containers were restarted.

Conclusions

The result is a containerized, automation-driven identity platform that replaces an ageing NT4 domain while keeping the laboratory's existing identities intact. The work confirmed that the hard part of this kind of migration is rarely the directory software itself; it is the surrounding agreement on identity that has to hold across protocols and operating systems. Several things only became clear once the containers were actually running, and those observations, together with the limits of the current setup and the directions worth pursuing next, are discussed in the closing chapters.

Acknowledgements

Thank you to my dear colleagues in LABINF, Davide, Andrea and Luca for their support and collaboration throughout this journey.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Laboratorio di Informatica (LABinf) of the Politecnico di Torino is a teaching laboratory of about ninety workstations, used throughout the academic year for programming courses, lab exams, and supervised exercises. For a student sitting down at one of those machines, the experience is meant to be simple: log in with a single account, find a personal home directory already mounted, reach a network printer, and not think about any of it. Behind that simplicity sits a shared server infrastructure that has been authenticating users and holding the laboratory's identity database together for well over a decade.

That infrastructure is the subject of this thesis. Almost everything the lab depends on runs on a single host, `cclix1`, which is at the same time the domain controller, the file server, and the print server. Its authentication core is a Samba NT4-style domain backed by an OpenLDAP directory, an architecture that was perfectly reasonable when it was first deployed, and that has slowly aged into something nobody really wants to touch.

The reasons are partly technical and partly organisational. The NT4 domain model has been deprecated for years, and keeping it alongside modern clients has required compromises that weaken its security posture. Account management leans on tools that are no longer maintained, so every change carries the quiet risk of breaking something that cannot easily be fixed. And the documentation that should explain how all of this fits together has drifted out of sync with reality, to the point where the most reliable source of truth is the running configuration itself. None of this prevents the lab from working day to day. It does mean the system has become fragile, hard to reason about, and uncomfortable to maintain, the kind of setup that keeps running mostly because nobody dares to change it.

1.1 Proposed Solution

The work described here replaces that setup with a Samba Active Directory Domain Controller, deployed inside Podman containers and provisioned through Ansible. Moving to Active Directory brings the lab onto a model that is still actively supported and that integrates cleanly with both Windows and Linux clients. Containers undo the single-host entanglement of the old design, where every service shared the same kernel and process space: each piece now runs in its own isolated environment that can be rebuilt from scratch rather than patched in place. Ansible carries the configuration, so the whole deployment is described in code, reproducible, and safe to re-run without damaging a domain that has already been provisioned.

The migration is a clean cutover rather than an in-place upgrade. A central constraint is continuity: students and their files must survive the switch, so the existing user identities have to be carried over faithfully and the ownership of years of accumulated data on the shared storage must remain intact. Much of the engineering effort goes into guaranteeing exactly that.

1.2 Contribution

This thesis is not a survey of identity management options, nor a generic tutorial on Active Directory. It documents one real migration of one real laboratory, with the actual configuration, the decisions taken along the way, and the reasoning behind them. The value lies less in describing the happy path than in the parts that did not work the first time and in the constraints imposed by a system that has to keep serving students while it is being replaced. The artefacts produced, the Ansible roles, the container definitions, and the Python tooling for day-to-day account management, are meant to be usable, not merely illustrative.

1.3 Structure of the Thesis

The chapters move from background, through analysis and design, into implementation and validation. The arc roughly follows a *what* / *why* / *how* logic: first what the relevant technologies are and what the current system looks like, then why the new design takes the shape it does, and finally how it was built and confirmed to work.

Chapter 2 reviews directory services and identity and access management, from LDAP and Kerberos to the Active Directory model. Chapter 3 covers the implementation technologies: containerisation with Podman and automation with

Ansible. Together these two chapters form the background and can be skipped by a reader already familiar with the material.

Chapter 4 turns to the specific case, analysing the existing `cclix1` infrastructure and the weaknesses that justify replacing it. Chapter 5 presents the logical architecture of the target system, so the container layout, the identity schema, the storage model, and the authentication flows.

Chapter 6 is where the design choices are argued explicitly, weighing the alternatives that were considered. Chapters 7 and 8 then describe the concrete implementation: the provisioning of the domain controller and its container, and the file server with its Winbind identity resolution and NFS integration.

Chapter 9 reports the testing that verifies authentication, identity mapping, and file access end to end across both client platforms.

Chapter 10 steps back from the working system to the experience of building it, collecting the operational sharp edges and the places where the design turned out to be more fragile than it looks. Chapter 11 draws the work together, restates what the migration achieved and at what cost, and sketches the directions in which it could reasonably be taken further.

Chapter 2

Directory Services and Identity Management

2.1 Directory Services

A directory service is a network information store designed around one basic assumption: reads happen far more often than writes. Applications query a directory to resolve user identities, look up configuration parameters, or verify group memberships; they rarely modify entries. This asymmetry distinguishes a directory from a general-purpose relational database, and explains the design choices that make directory servers fast for lookups, hierarchically organised, and often distributed with replica copies across multiple sites.

2.1.1 Historical Evolution from X.500

The ITU ratified the X.500 standard in 1988, and then revised it in 1993 and 2001 [1]. The goal was ambitious: a single, globally distributed white-pages service that any organisation could publish to and any user could query. X.500 introduced the *Directory Information Tree* (DIT) for organising entries hierarchically, the *Directory System Agent* (DSA) as the server process, and the *Directory Access Protocol* (DAP) for client-server communication. DAP was a full OSI-layer protocol, and therein lay the problem. Implementing it over the TCP/IP stacks of the early 1990s was painful enough that most network operators simply did not bother.

Tim Howes, Wendy Yeong, and Mark Smith at the University of Michigan published the Lightweight Directory Access Protocol in 1993 as a TCP/IP-native shortcut to X.500 DSAs [2]. Early LDAP stripped out the OSI machinery and kept only what applications actually needed: bind, search, add, modify, delete. Later standardisation rounds progressively decoupled LDAP from X.500 entirely.

LDAPv3, defined across RFC 4510 through RFC 4519 [3], is the version in use today and the basis for every serious directory product, open source or commercial.

2.1.2 Directory Information Tree Structure

Every entry in an LDAP directory sits at a node in the DIT. Its position in the tree is encoded by its *Distinguished Name* (DN), built by concatenating a sequence of *Relative Distinguished Names* (RDNs) from the root downward. For most organisations the root is expressed using Domain Component attributes that mirror the DNS name: the base DN for `example.com` is `dc=example,dc=com`, and a user Alice within an organisational unit for staff would carry the DN `cn=Alice,ou=Staff,dc=example,dc=com`.

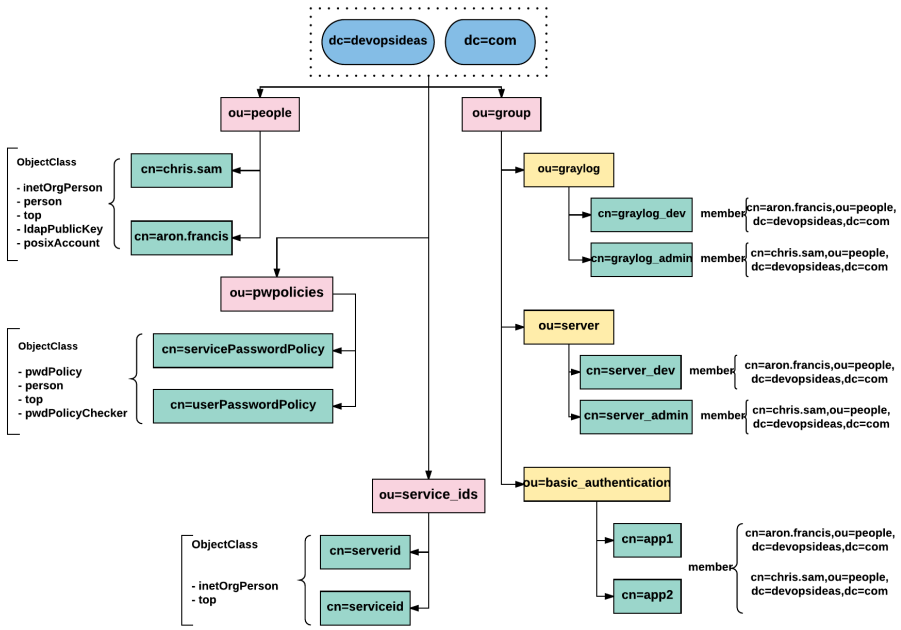

Figure 2.1: Example DIT showing the hierarchical organisation from the domain root down to individual user and group objects.

The tree structure is not merely cosmetic. Subtrees can be searched with scoped queries that avoid touching unrelated branches. Administrative authority can be delegated at any point in the hierarchy. Replication can be configured to synchronise only selected subtrees to replica servers that serve a particular geographic site. Poor DIT design like a flat structure with tens of thousands of objects under a single container, for example produces real performance degradation for both searches

and replication.

2.1.3 Objects, Attributes, and Schema

Every entry belongs to one or more object classes, and those classes decide which attributes it must carry and which it may. The classes themselves come from schema files the server loads at startup. `person` demands `cn` and `sn`. `inetOrgPerson` builds on `person` with the attributes that ordinary internet directories ended up needing, and organisations layer their own subclasses on top for whatever they have to represent locally.

Attribute types have associated *syntaxes* that constrain what values are valid, and *matching rules* that determine how values are compared during searches. The choice of matching rule is not academic: a telephone number stored with an equality matching rule cannot be found by a substring search, which matters when an application needs to look up users by partial string. Schema design errors of this kind are difficult to correct retroactively in a production directory.

2.1.4 Replication and Scalability

No serious directory runs on one server. LDAPv3 leaves replication out of the standard, so every product solved it differently: OpenLDAP built Syncrepl on top of the LDAP Content Synchronisation extension [4], Active Directory uses a proprietary multi-master protocol over RPC, and Samba 4 reimplements the Active Directory model wholesale. None of it is portable. Move between two directory products and the replication topology has to be replanned from scratch.

The two replication models pull in opposite directions. Single-master is the easy one to reason about, since every write lands on one authoritative server and fans out from there, but that one server is also a write bottleneck, and if it is down nobody can write anything anywhere. Multi-master lifts the bottleneck and pays for it in conflict resolution: when the same entry is edited on two replicas before they sync, something has to decide who wins. Active Directory settles those at the level of individual attributes, usually by the later timestamp.

Referrals are the older answer to the same scale problem. A server that does not hold a given subtree can point the client at one that does, a trick inherited straight from X.500 that lets independent administrative domains federate without any one server holding everything. Modern deployments lean on replication instead and referrals show up less, but they are still in the LDAPv3 spec and still turn up in the field, particularly across large academic federations.

2.1.5 Role in Enterprise Infrastructures

Authentication systems, mail servers, VPN gateways, file sharing services, and configuration management platforms all depend on a shared directory as the source of truth for identities and attributes. This central position has two sides. On one side, the directory is a multiplier: keeping account information in one place eliminates the drift that builds up when user data is duplicated across independent stores, each updated at different times by different teams. On the other, the directory is a single dependency whose unavailability can propagate failures across every service that relies on it.

Operationally significant attributes include POSIX values like `uidNumber`, `gidNumber`, and `homeDirectory`, which allow Linux and Unix systems to resolve accounts from the directory rather than from a local `/etc/passwd`. Mail attributes allow routing without a separate alias database. The directory also stores group memberships that govern access to filesystems, printers, and applications, making it a de facto access control database even when no formal access management system has been deployed.

2.2 Identity and Access Management

IAM is the discipline concerned with ensuring that the right individuals access the right resources at the right times. That definition is easy to state and genuinely hard to implement. Provisioning a new employee with an email address and file server access is trivial. Doing it consistently across a hundred systems, keeping it accurate as the employee's role changes over several years, and removing all of it completely on the day they leave, is not that easy as it seems.

2.2.1 Digital Identities

A *digital identity* is a collection of claims about an entity, a person, a device, or a service that systems use to make decisions about how that entity should be treated. The claims might be as sparse as a username and a credential, or as rich as a profile including organisational role, device health, and geographic location. How trustworthy those claims are depends on the care taken during identity proofing (verifying who the person actually is before creating their account) and on the integrity of the systems that store and convey the claims thereafter.

It is worth separating the concepts of *identity* and *account*. An identity is an abstract representation of an entity; an account is a concrete instantiation of that identity within a specific system. One employee typically has a single identity but accounts in many systems: HR software, version control platforms, cloud providers, internal web applications. One of the primary problems that IAM infrastructure

exists to solve is ensuring that those accounts remain consistent with each other, and that changes to the identity as a name change, a department transfer, a termination, propagate correctly to all of them.

2.2.2 Authentication, Authorisation, and Auditing

Authentication is the verification that an entity is who it claims to be. The claim can be supported by something the entity knows (a password or PIN), something it has (a hardware token or smart card), or something it is (a biometric). Multi-factor authentication requires at least two of these, and raises the cost of credential compromise substantially.

Authorisation is logically separate: knowing who a user is does not by itself determine what they may do. Authorisation decisions can be enforced locally by individual applications or delegated to a centralised policy engine. Modern architectures increasingly favour the latter, on the grounds that local enforcement creates as many inconsistencies as the distributed account stores that centralised IAM infrastructure was meant to replace.

Auditing closes the loop. It records what was done, when, and under whose identity, in a form somebody can go back and read, for a compliance check, a forensic investigation, or plain operational curiosity about why something broke. In healthcare, finance and critical infrastructure the law requires those logs to exist and to be tamper-evident, so there is nothing optional about them.

2.2.3 Identity Lifecycle Management

An identity is not static. From initial provisioning through role changes, departmental transfers, temporary access grants, suspension, and eventual deprovisioning, each account goes through a lifecycle that must be managed explicitly. Unmanaged lifecycles cause predictable problems: accounts that are not deprovisioned when an employee leaves remain potential entry points; permissions that are not updated when a user changes roles accumulate into privilege creep.

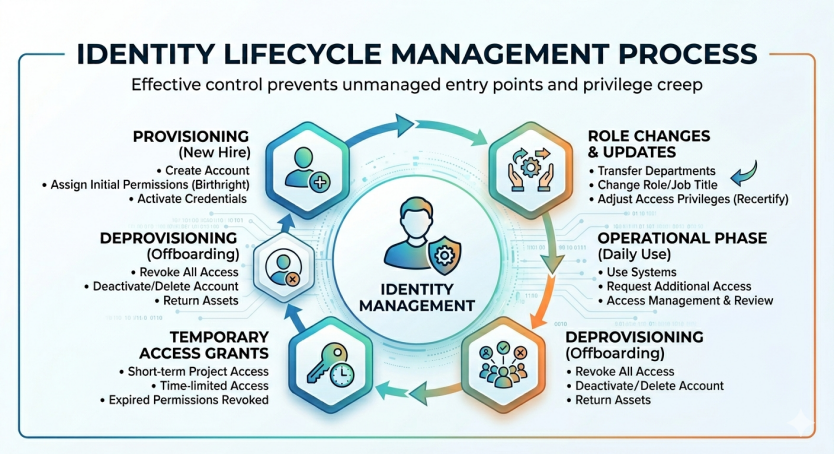

Figure 2.2: IAM identity lifecycle from initial provisioning through role transitions and eventual deprovisioning.

Joiner-Mover-Leaver (JML) workflows codify which actions must be taken at each of these lifecycle events. In a mature IAM implementation, the workflows are automated: a record created in the HR system triggers account creation in the directory and downstream applications; a department transfer triggers group membership changes; a termination triggers immediate suspension followed by scheduled deletion. The degree to which this can be automated depends on the quality of authoritative data sources,most often HR systems and on whether downstream applications expose provisioning interfaces at all.

2.2.4 Role-Based and Attribute-Based Access Control

RBAC organises permissions around roles rather than individual identities. Users are assigned to roles; roles carry permissions. The practical advantage is administrative scale: adding a new employee to a role grants them all of the permissions appropriate to that function in a single step, rather than enumerating resources individually. Role hierarchies allow senior roles to inherit the permissions of subordinate ones. Ferraiolo and Kuhn formalised the model in 1992 [5]; NIST subsequently published a reference model [6] that remains widely cited.

ABAC, described in NIST SP 800-162 [7], evaluates access decisions against a policy that can reference any attribute of the subject, the resource, the action, or the environment. A policy might read: *allow read access to documents whose sensitivity level is at most the subject's clearance level, during business hours, from devices with current compliance status.* The expressiveness comes at a price. ABAC policies are more complex to write and audit than role assignments, and centralised policy evaluation can introduce latency that some applications cannot tolerate.

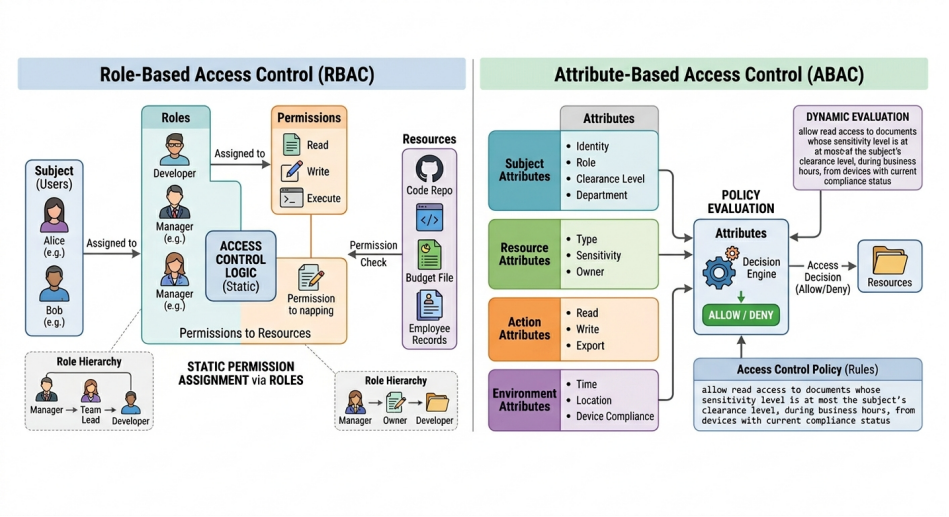

Figure 2.3: RBAC assigns permissions through static role memberships; ABAC evaluates dynamic policies against subject, resource, action, and environment attributes.

In practice, large organisations rarely implement either model in pure form. RBAC handles stable, well-understood access patterns; ABAC policies cover edge cases and context-sensitive exceptions. The two are not mutually exclusive.

2.2.5 IAM and Directory Services

The relationship is symbiotic. The directory provides the persistent store for identity data like accounts, group memberships, role assignments and the query interface through which that data is read. IAM processes sit above this layer, driving the creation and modification of directory entries and consuming directory data to make access decisions. A well-designed directory with no surrounding IAM processes will accumulate stale data within months; an IAM system built on an unreliable directory will produce inconsistent access control regardless of how carefully the processes are designed.

2.3 Open Source Alternatives

The commercial directory market is dominated by Microsoft Active Directory and Oracle Directory Services, but a range of open source alternatives are used widely in academic, research, and public sector environments. They differ significantly in architectural scope, from bare LDAP servers to fully integrated identity management suites.

2.3.1 OpenLDAP

OpenLDAP [8] is the most widely deployed open source LDAP server. It descends from the University of Michigan LDAP implementation; the OpenLDAP Project was created in 1998 to continue that work under an open licence. The server implements LDAPv3 thoroughly and is built around a modular overlay architecture: each functional layer, access control, synchronisation, attribute value constraints, referential integrity, is a separately loadable module that can be enabled or disabled without recompiling the server.

The current backend storage uses MDB, derived from the LMDB library, which replaced the older BDB backend in version 2.4. MDB provides ACID transactions and better write throughput than its predecessor. Replication is handled by Syncrepl, which supports both provider-push and consumer-pull topologies and can replicate only changed attributes rather than full entries, reducing bandwidth on busy directories.

OpenLDAP provides no graphical management interface. Administration is done through command-line tools (`ldapadd`, `ldapmodify`, `ldapsearch`) or through third-party GUIs. This is both a strength and a practical obstacle: the absence of a web console eliminates an attack surface and avoids a layer of abstraction that sometimes obscures what is actually happening in the directory, but it places a higher burden on administrators who are not comfortable reading and writing LDIF directly.

2.3.2 FreeIPA

FreeIPA [9] is not a directory server but an integrated identity management suite. It bundles a 389 Directory Server instance, a Kerberos KDC, a BIND DNS server, a Dogtag certificate authority, and a web management console into a single, opinionated deployment. Red Hat sponsors the project; the downstream product is Red Hat Identity Management.

The design goal is to provide a Linux-native alternative to Active Directory covering the same functional scope: centralised authentication, DNS-integrated service discovery, host enrolment, and group policy. Kerberos integration is deep: every enrolled host receives a Kerberos principal, and service tickets can be issued for registered services within the IPA realm. SSSD is the recommended client daemon.

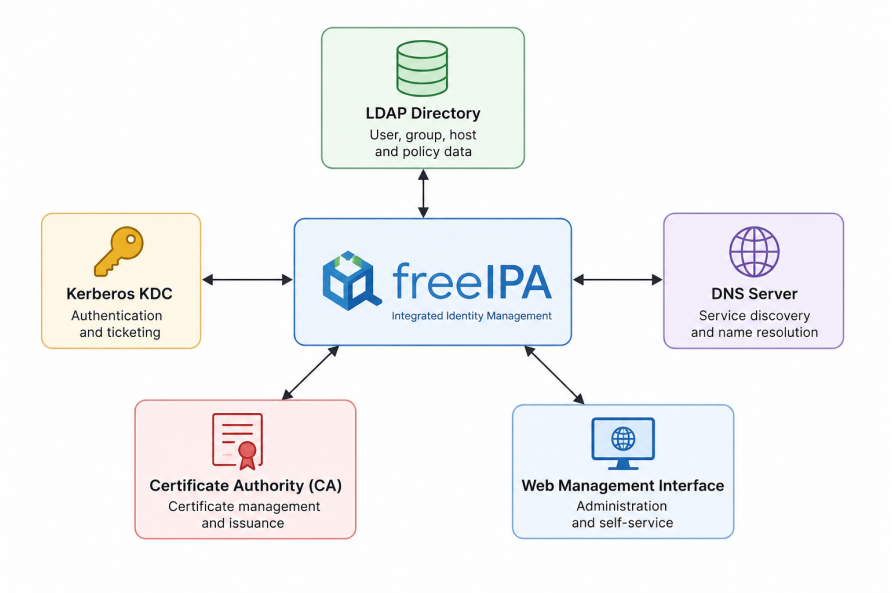

Figure 2.4: FreeIPA architecture: integrated components comprising an LDAP directory, Kerberos KDC, DNS server, certificate authority, and web management interface.

The trade-off of that integration is reduced flexibility. Each component is configured to work with the others, and stepping outside the expected topology requires careful manual intervention. FreeIPA also assumes a homogeneous Linux/Unix client base; Windows client support exists but is significantly weaker than what Samba AD provides.

2.3.3 389 Directory Server

389 Directory Server [10] is the LDAP engine inside FreeIPA, but it is also packaged and used independently. It traces its lineage to the Netscape Directory Server, one of the earliest commercial LDAP products. Relative to OpenLDAP, 389 Directory Server has historically had stronger built-in support for multi-supplier replication, a topology where multiple servers can accept writes simultaneously and a somewhat more accessible administration console. At high scale, millions of entries with thousands of concurrent connections, it has been competitive with commercial products.

2.3.4 Keycloak

Keycloak [11] occupies a different position in the IAM landscape. It is an identity broker, not a directory server. Its focus is modern authentication protocols: OpenID

Connect, OAuth 2.0, and SAML 2.0. The typical deployment puts Keycloak in front of an existing identity store like an LDAP directory, an Active Directory domain, or social identity providers and uses it to issue tokens consumed by web applications and APIs.

Keycloak does not replace a directory; it federates one. This makes it relevant in environments that need to expose enterprise identities to cloud services or microservice architectures without granting those services direct LDAP access or requiring them to understand Kerberos.

2.3.5 SSSD

The System Security Services Daemon [12] is a client-side component that sits between the Linux PAM and NSS subsystems and one or more identity backends. SSSD supports LDAP, Active Directory, and FreeIPA as backends, handling protocol details transparently from the operating system's perspective. It caches user and group information locally, so authentication can succeed even when the directory server is temporarily unreachable. On a laptop or a machine at a remote site with intermittent connectivity, this is practically significant.

SSSD has largely replaced older approaches like `nss-ldap` and `pam-ldap` on modern Linux distributions. Its responder and provider processes are separated: the responders handle PAM and NSS queries; the providers communicate with the identity backend. Either side can be restarted independently, which helps in practice when debugging authentication issues without interrupting active sessions.

2.4 Protocols

Directory services and identity management rely on a small number of network protocols whose properties shape the security and operational characteristics of the whole system. Each protocol has a history worth knowing, because design decisions made decades ago still surface as constraints in contemporary deployments.

2.4.1 LDAP

LDAPv3 operates over TCP, using port 389 for cleartext or StartTLS-upgraded connections and port 636 for LDAP over TLS. A session begins with a *bind* operation that establishes the identity under which subsequent requests will be executed. LDAPv3 supports multiple bind mechanisms through SASL [13]: PLAIN (a username and password sent over an already encrypted connection), GSSAPI (Kerberos), EXTERNAL (certificate-based), and others. An anonymous bind, which requires no credentials, is permitted by default in many servers and grants

access to whatever attributes the server's access control policy marks as publicly readable.

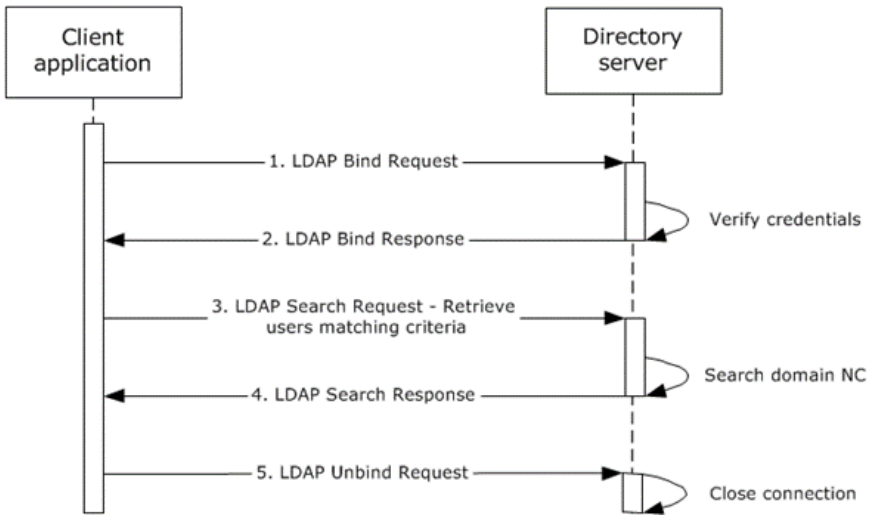

Figure 2.5: LDAP communication flow: a client binds to establish identity, issues a search request, receives matching entries, and unbinds.

The search operation is the most used and most complex of the LDAP operations. It takes a base DN, a scope (base, one-level, or subtree), a filter, and a list of requested attributes. Filters combine equality, substring, presence, ordering, and extensible matching assertions with AND, OR, and NOT. A filter like `(&(objectClass=inetOrgPerson)(uid=al*))` finds all `inetOrgPerson` entries whose `uid` attribute starts with *al*.

LDAPv3 introduced *controls* as a general extension mechanism. A control is an opaque block attached to any request or response that modifies how that operation is handled. The paging control limits result sets by returning entries in batches, which matters for searches over large directories. The server-side sorting control requests that the server sort results before returning them. The password policy control communicates password expiry warnings back to the client. Extensions work similarly but add entirely new operation types: the StartTLS extended operation upgrades a cleartext connection to TLS without opening a second TCP connection.

2.4.2 Kerberos

Kerberos was designed at MIT in the mid-1980s as part of Project Athena [14]. The problem it addresses is specific: how can two parties on an untrusted network prove their identities to each other without transmitting passwords, and do so efficiently enough to support single sign-on across many services during a workday?

The answer is a trusted third party called the Key Distribution Centre (KDC) and a ticket-based credential system.

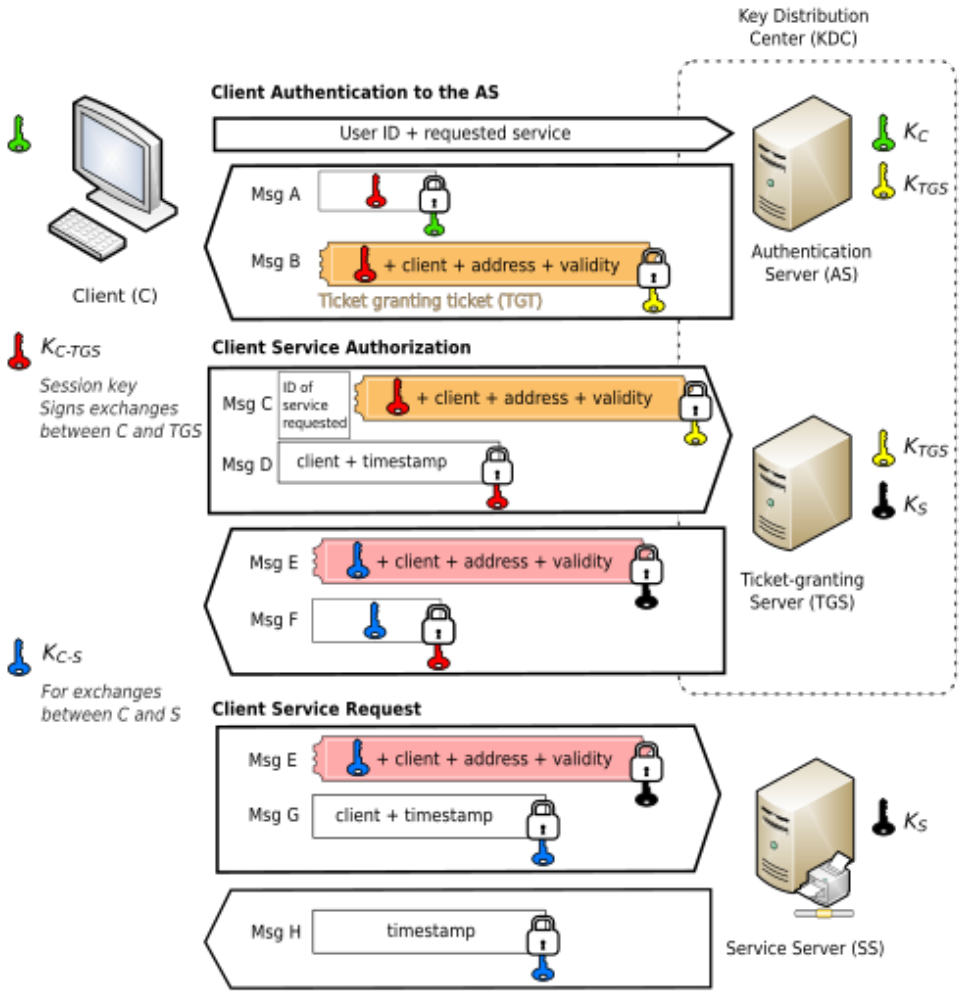

Figure 2.6: Kerberos V5 authentication workflow: the client obtains a TGT from the Authentication Service and exchanges it for service tickets at the Ticket Granting Service.

Kerberos V5 is specified in RFC 4120 [15]. The protocol has two phases. First, a client authenticates to the Authentication Service (AS) component of the KDC, typically by deriving a key from the user's password and using it to prove identity. The AS returns a Ticket Granting Ticket (TGT), encrypted with the KDC's own key, alongside a session key for communicating with the second component, the Ticket Granting Service (TGS). When the client needs to access a service, it presents

its TGT to the TGS and requests a service ticket for the target. The TGS issues a service ticket encrypted with the service's long-term key. The client forwards this to the service, which decrypts it and verifies the client's identity. No password is transmitted over the network at any point.

Kerberos cares about the clock to an unusual degree. Every ticket is timestamped, and the KDC rejects it if the client's clock and its own differ by more than the configured tolerance, five minutes out of the box. That makes NTP part of the security design rather than housekeeping: let the clocks drift and authentication starts failing for reasons that look nothing like a clock problem.

Service Principal Names are how the model names a particular service instance. An SPN is a service class plus a hostname, optionally with a port or instance tacked on. When a client asks for a service ticket, the KDC uses the SPN to find the right encryption key. Active Directory stores SPNs as attributes on machine and service account objects and adds them automatically when a computer joins the domain. The trouble starts when they have to be set by hand. A mistyped or missing SPN is one of the most common reasons Kerberos quietly fails in a mixed environment.

2.4.3 NTLM and SMB/CIFS

NTLM came before Kerberos on Windows and still lingers behind it. It works by challenge and response: the server throws a random nonce at the client, the client hashes the user's password together with that nonce and sends the result back, and the server checks it. The catch is that the server has no password to check against, since it stores none, so it forwards the challenge and the response to a domain controller and waits for a verdict. Every single network authentication makes that round trip to the DC, which is exactly why NTLM falls apart under load.

The bigger problem is pass-the-hash. Capture an NTLM response and you can replay it against other servers without ever knowing the password behind it. NTLMv2 folds client and server nonces into the hash and closes off some of the replay paths, but the attack class survives. Microsoft's own advice is to turn NTLM off wherever Kerberos can do the job. In practice it is hard to kill outright, because plenty of old applications and protocols drop back to it the moment Kerberos is unavailable, and they do it silently.

SMB (Server Message Block), also called CIFS in its older incarnations, is the network file sharing protocol used throughout Windows environments [16]. Sessions are established over TCP port 445 and authenticated with either Kerberos or NTLM. SMB1, the version dominant through the early 2000s, had well-documented security vulnerabilities and was deprecated by Microsoft following the WannaCry ransomware incident in 2017. SMB2 (Windows Vista) and SMB3 (Windows 8) introduced request compounding, improved caching, mandatory signing options, and end-to-end encryption. Samba implements all three versions; SMB1 is disabled

by default in current releases.

2.4.4 DNS and SRV Records

In Active Directory and Kerberos deployments, DNS does more than map hostnames to addresses. The SRV resource record, specified in RFC 2782 [17], allows services to advertise their location in a standardised, discoverable way. A client that needs to find a Kerberos KDC for the realm `EXAMPLE.COM` queries for the SRV record `_kerberos._tcp.EXAMPLE.COM` and receives the hostnames and port numbers of available KDCs, along with priority and weight values that the client uses for load distribution and failover.

This SRV-based discovery means DNS is a security-critical component of the authentication infrastructure, not just a naming convenience. An attacker able to inject false DNS responses can redirect clients to a rogue KDC or a rogue domain controller. Active Directory addresses this partly by integrating DNS directly: domain controllers are also authoritative DNS servers for the domain zone by default, and the zone data is stored inside the directory itself, replicated alongside other domain data. Changes to domain membership automatically update DNS.

2.4.5 Protocol Comparison

Table 2.1 summarises the protocols discussed above.

Table 2.1: Comparison of core directory and authentication protocols.

Protocol	Purpose	Transport	Port(s)	Auth model
LDAP	Directory access	TCP	389	Simple / SASL
LDAPS	Directory access	TCP+TLS	636	Simple / SASL
Kerberos	Authentication	TCP/UDP	88	Ticket-based
NTLM	Authentication	Embedded	N/A	Challenge-response
SMB2/3	File/resource sharing	TCP	445	Kerberos / NTLM
DNS (SRV)	Service discovery	UDP/TCP	53	N/A

2.5 The NT4 Domain Model

Before Active Directory, Windows networks used the NT 4 domain model for centralised authentication and resource management. The model is no longer deployed in new environments, but its concepts: PDC, BDC, SAM database, NetBIOS names; persist in documentation, configuration files, and the compatibility

layers built into Active Directory. Understanding what NT 4 domains were, and what their limitations were, helps explain why Active Directory was designed the way it was.

2.5.1 Architecture

An NT 4 domain is a flat administrative boundary with no hierarchy and no concept of delegation. Every user, group, and computer account is stored in the *Security Account Manager* (SAM) database, a proprietary binary file on the domain controllers. There is exactly one writable copy of the SAM, held by the *Primary Domain Controller* (PDC). Any number of *Backup Domain Controllers* (BDCs) hold read-only replicas and can authenticate users, but all account modifications must go to the PDC.

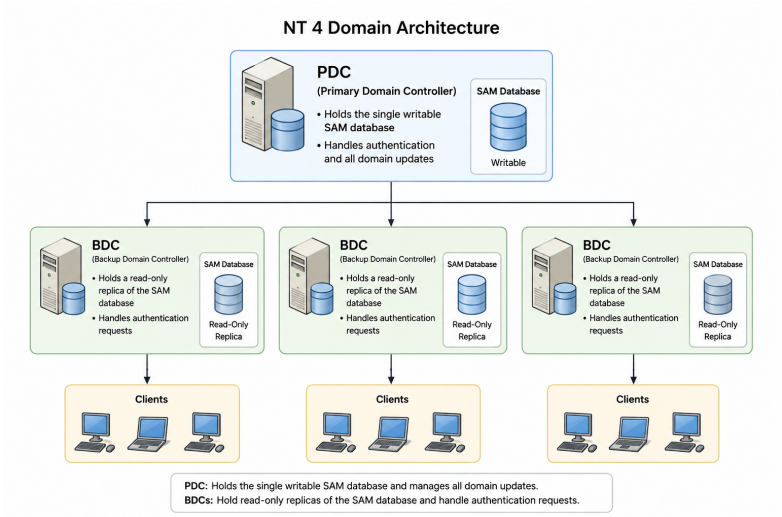

Figure 2.7: NT 4 domain architecture: the PDC holds the single writable SAM database; BDCs hold read-only replicas and handle authentication requests.

If the PDC became unavailable, no changes to the domain could be made until it was restored, or until an administrator manually promoted a BDC, a disruptive procedure that was a fairly common source of operational pain in environments that relied on NT 4 through the late 1990s.

Name resolution used NetBIOS rather than DNS. Machines either broadcast their names on the local segment or registered them with a WINS server. NetBIOS names are limited to 15 characters, are flat, and are scoped to a subnet by default. Large, routed enterprise networks had to use WINS servers to bridge subnets, adding infrastructure and a single point of failure in the name resolution path.

2.5.2 Authentication

Domain logons use NTLM challenge-response. When a user logs onto a workstation, the workstation exchanges a challenge-response sequence with a domain controller. Member servers that need to verify a domain user's identity forward the exchange to a DC through a *pass-through authentication* channel, the server does not store password data locally, which is better than earlier peer-to-peer models, but every network authentication still requires DC involvement.

Trust relationships between NT 4 domains are one-way and non-transitive. If domain A trusts domain B, users from B can access resources in A, but not the reverse, and the trust does not extend to domains that B trusts. A multi-domain organisation needed explicit trust relationships between every pair of domains it wanted to connect, a mesh that grew quadratically and had to be maintained manually.

2.5.3 Limitations and Decline

Microsoft's own documentation recommended keeping the SAM database below roughly 40,000 objects. In practice, organisations with large numbers of workstations hit this ceiling and had to split into multiple domains, which reintroduced the trust mesh problem. Delegation was impossible below the domain level; the whole domain was the minimum administrative unit. There were no organisational units, no scoped group policies, no hierarchical administration.

Windows 2000 replaced the NT 4 domain model with Active Directory. NT 4 domain controllers could participate as BDCs in mixed-mode Active Directory domains during migration, giving organisations an upgrade path, but the model itself was effectively end-of-life from the year 2000. NTLM authentication survived as a backward-compatibility mechanism, and it remains supported in all current versions of Windows.

2.6 Active Directory and Samba

Active Directory, shipped with Windows 2000 Server, replaced everything the NT4 model lacked: hierarchy, delegation, Kerberos, DNS-native naming, and a replication model that did not require a single master [18]. It is today the dominant directory and identity platform in enterprise environments.

2.6.1 Active Directory Architecture

The fundamental unit of organisation in Active Directory is the *domain*. A domain has its own LDAP namespace, its own Kerberos realm, its own KDCs, and its

own group policy scope. Multiple domains sharing a contiguous DNS namespace form a *tree*; multiple trees joined by trust relationships form a *forest*, which is the outermost security boundary. Trust relationships within a forest are two-way and transitive by default, which means that any domain in a forest can, by policy, be made accessible to users from any other domain, without the manual trust meshes required in NT 4.

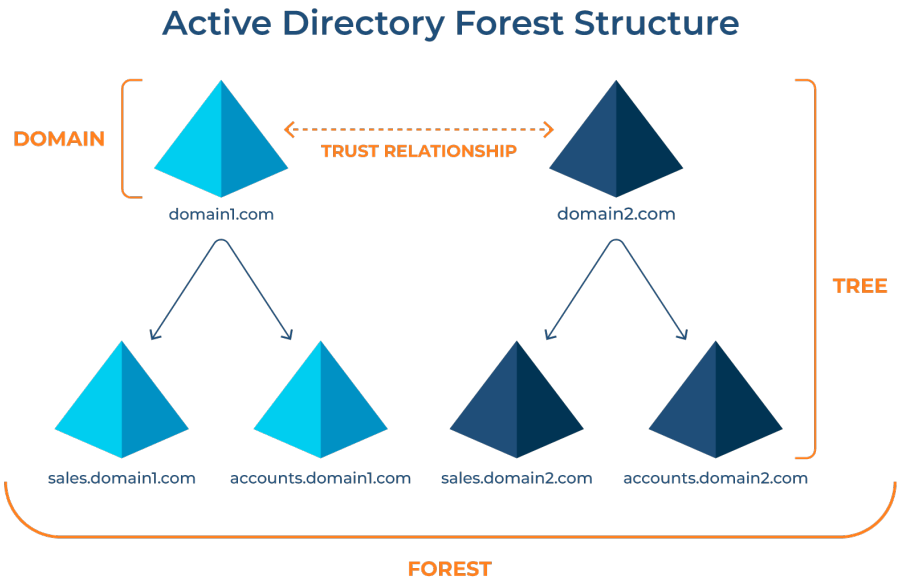

Figure 2.8: Active Directory logical structure: domains with shared DNS namespaces form trees; trees connected by two-way transitive trusts form a forest.

Within a domain, the flat SAM database is replaced by a full LDAP DIT. Objects are organised into *Organisational Units* (OUs), which serve as both structural containers and the granularity at which Group Policy Objects (GPOs) are applied. An administrator can delegate control over a specific OU, for example, the right to reset passwords for users in that OU, without granting any domain-wide privilege. This was one of the most operationally significant improvements over the NT 4 model.

All domain controllers participate in multi-master replication. Any DC can accept writes, and changes propagate through the domain using the *Knowledge Consistency Checker* (KCC), which builds and adjusts the replication topology automatically. Certain privileged operations, schema changes, RID pool allocation, PDC emulation, are assigned to specific DCs through the *Flexible Single Master Operations* (FSMO) role system. Losing a FSMO holder does not take the domain down; it only blocks the specific operation that role governs.

Group Policy is one of Active Directory's most operationally influential features.

GPOs are collections of configuration settings applied to OUs, domains, or sites: software installation, security baselines, desktop restrictions, logon scripts. Policy is evaluated hierarchically at logon time, and it can be scoped to specific groups with security filtering or conditioned on system properties with WMI filtering.

2.6.2 LDAP, Kerberos, and DNS Integration

Active Directory exposes the directory as an LDAP server implementing LDAPv3 with Microsoft extensions. The schema is fixed but extensible through the schema master; standard attributes cover user accounts (`sAMAccountName`, `userPrincipalName`, `userAccountControl`) and POSIX interoperability (`uidNumber`, `gidNumber` via RFC 2307 [19]).

Kerberos V5 is the default intra-domain authentication protocol. The domain DNS name is the Kerberos realm. Every domain controller runs a KDC; SPNs are stored as directory attributes and are registered automatically when a machine joins the domain.

DNS is integrated at the data level: Active Directory-integrated DNS zones store their data inside the directory, and zone updates replicate through the standard DC replication mechanism rather than through a separate DNS zone transfer. Domain controllers automatically register their SRV records for LDAP, Kerberos, and other services when they start, so clients discover available DCs through DNS queries without hardcoded address lists.

2.6.3 Samba Active Directory

Samba has implemented SMB/CIFS on Unix and Linux systems since the early 1990s. For most of that time it could act as a member server in a Windows NT domain or emulate an NT 4 PDC, but could not implement the Active Directory domain controller role. Samba 4.0, released in December 2012, changed this [20].

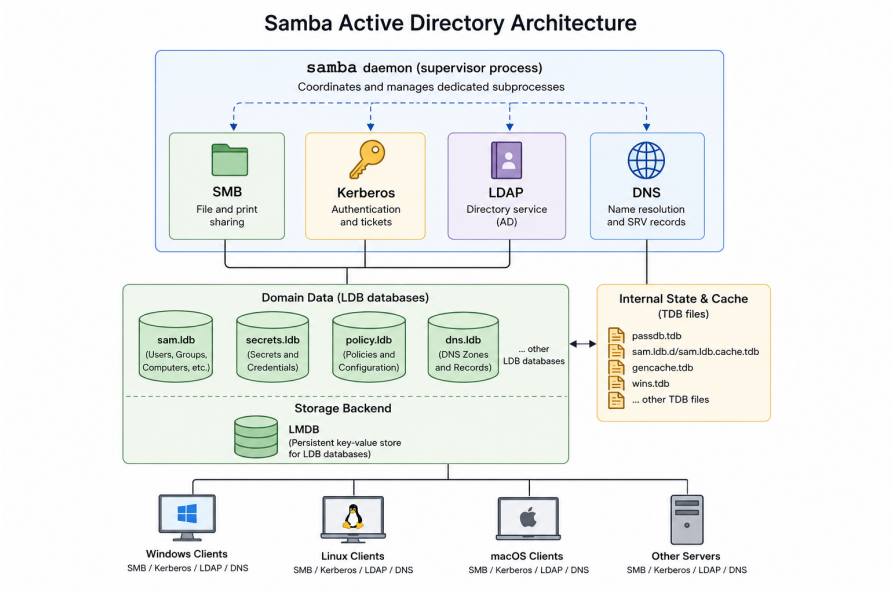

Figure 2.9: Samba Active Directory architecture: the `samba` daemon acts as a supervisor process coordinating dedicated subprocesses for SMB, Kerberos, LDAP, and DNS services. Domain data is stored in LDB databases backed by LMDB, while TDB files handle internal caching and state.

Samba AD uses LDB (a schema-aware, LDAP-like storage library) and TDB (a simple key-value store) rather than an external database engine. The Kerberos implementation is Heimdal, modified to integrate closely with the Samba AD backend. DNS can be served by the internal Samba server or delegated to a BIND installation loaded with a Samba plugin. Configuration lives in `smb.conf`, and directory operations are performed with `samba-tool`.

Samba AD supports RFC 2307 POSIX attributes without additional plugins, which means Unix and Linux clients can obtain UID, GID, and home directory information from the same LDAP query that returns Windows identity data. Domain provisioning is a single command: `samba-tool domain provision` creates the initial DIT, configures Kerberos, and registers DNS records. Samba AD is the only mature open source implementation of the Active Directory domain controller role, and it is commonly used in environments where Active Directory compatibility is required but Windows Server licensing is not an option.

2.7 Integration in Hybrid Environments

Most production environments of any scale are heterogeneous. Windows desktops and servers, Linux servers, macOS workstations, network storage appliances, and

embedded systems coexist and are expected to share a single identity namespace. Managing this consistently requires understanding where the Windows and POSIX identity models diverge, because those divergences create practical problems in every layer of the stack.

2.7.1 Identity Model Differences

Windows uses Security Identifiers (SIDs): variable-length, globally unique values assigned to every user, group, and computer at creation time. ACLs on Windows objects reference SIDs, not names. Renaming a user account or moving it between OUs has no effect on access rights because the SID does not change. This property makes Windows identities stable across organisational restructurings in a way that name-based systems are not.

POSIX uses 32-bit numeric identifiers: a UID for users, a GID for groups. On a standalone machine, UIDs are assigned locally and mean nothing outside that machine. Once filesystems are shared over NFS, UIDs must be coordinated across all machines that mount those filesystems, because the kernel uses the UID from the process making the request to check permissions against the UID stored in the inode, there is no name resolution at that layer.

Table 2.2: Comparison of Windows and POSIX/Linux identity models.

Aspect	**Windows**	**Linux / POSIX**
Identity representation	SID (variable length)	UID / GID (32-bit integer)
Identity assignment	Domain-wide unique	Machine-local (coordination needed)
Group membership	AD group (SID)	POSIX group (GID)
Authentication protocol	Kerberos V5 / NTLM	PAM (pluggable, backend-agnostic)
Name service	DNS + LDAP (AD)	NSS (files, LDAP, SSSD)
Access control	ACL with SID entries	DAC (mode bits) + optional ACL
Password policy	Enforced by KDC / domain	Local (/etc/pam.d) or via LDAP

2.7.2 SID to UID/GID Mapping

When a Linux system joins an Active Directory domain, it must translate between Windows SIDs and POSIX UIDs/GIDs. There are three approaches, each with different consistency properties.

Algorithmic mapping [21] derives a UID or GID deterministically from the SID using a fixed formula. No coordination is required, and results are consistent across machines without any shared state. The UIDs produced are large and numerically arbitrary, but they are stable as long as the SID does not change.

RFC 2307 mapping [22] stores POSIX attributes directly in the Active Directory object [19]. When a Linux client resolves a user, it reads `uidNumber` and `gidNumber` from the directory and uses them as given. This requires the directory to be provisioned with explicit POSIX attributes for every account, which is additional administrative work, but the results are predictable and the same value appears on every machine that reads from the same directory.

Persistent database mapping relies on Winbind to allocate a UID from a configured range the first time it encounters a given SID, and to store that mapping permanently in a local TDB file. The mapping is stable as long as the database is intact, but it is local: different machines will allocate different UIDs for the same SID unless the databases are synchronised, which is not automatic.

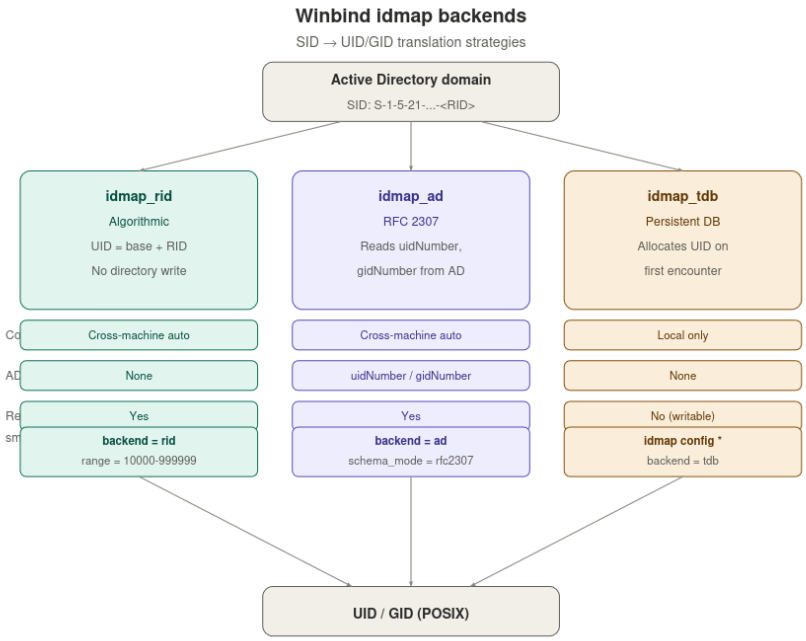

Figure 2.10: SID to UID/GID mapping strategies in a hybrid environment. RFC 2307 stores POSIX attributes in the AD directory; algorithmic mapping derives values locally from the SID; the Winbind persistent database stores per-machine allocations.

2.7.3 Winbind

Winbind is the Samba component that resolves Windows domain identities on Linux hosts. It implements the NSS and PAM interfaces, so standard tools like `id` or `ls -l` can resolve domain users and groups as if they were local accounts. Behind the scenes, Winbind queries the Active Directory LDAP server for user and group attributes and contacts the KDC for Kerberos authentication. It maintains a local cache to reduce latency and to allow authentication to proceed when a domain controller is temporarily unreachable.

The `wbinfo` command provides diagnostic access to the Winbind daemon: listing known domains, querying individual users and groups, and testing domain trust relationships. It is typically the first tool to run when debugging Winbind behaviour.

In environments where SSSD is preferred over Winbind, the functional role is similar, but the implementation is architecturally different and the range of supported backends is wider. SSSD is generally reported to handle high concurrency more gracefully, and its per-backend daemon model makes it easier to isolate failures.

2.7.4 Kerberos and PAM Integration

On a Linux machine enrolled in a Kerberos realm, user logins can be authenticated directly by the KDC rather than by a local password database. The `pam_krb5` or `pam_sss` module performs the Kerberos AS exchange when the user enters their password. On success, it populates a credentials cache with a TGT. Services that understand Kerberos, NFS with RPCSEC_GSS, SSH with GSSAPI, and others can then use this TGT to obtain service tickets silently, giving the user single sign-on behaviour across Kerberised services for the lifetime of the ticket.

The `/etc/krb5.conf` file specifies the realm name, KDC addresses, and admin server. Setting `dns_lookup_kdc = true` delegates KDC discovery to DNS SRV records, which avoids hardcoding domain controller addresses and allows the client to adapt automatically when controllers are added or removed.

2.7.5 Shared File Services

Nothing makes the identity-mapping problem as concrete as file sharing. NFS checks permissions down in the kernel, purely on UID and GID. If the UID a user carries on the client does not match the UID stored in the file's ownership on the server, the check fails, and it fails no matter how friendly the user's display name is or how valid their Kerberos principal is. So the same user has to resolve to the same number on every machine that touches the share. That is not tidiness for its own sake; it is the difference between the share working and not working.

SMB takes a different route to the same place. Samba does its access control on Windows SIDs and only drops to UID and GID when it has to touch the POSIX filesystem underneath. For Windows and Linux clients to share the same files over SMB, the SID-to-UID mapping has to line up on both sides, so that one on-disk UID always means one Windows identity.

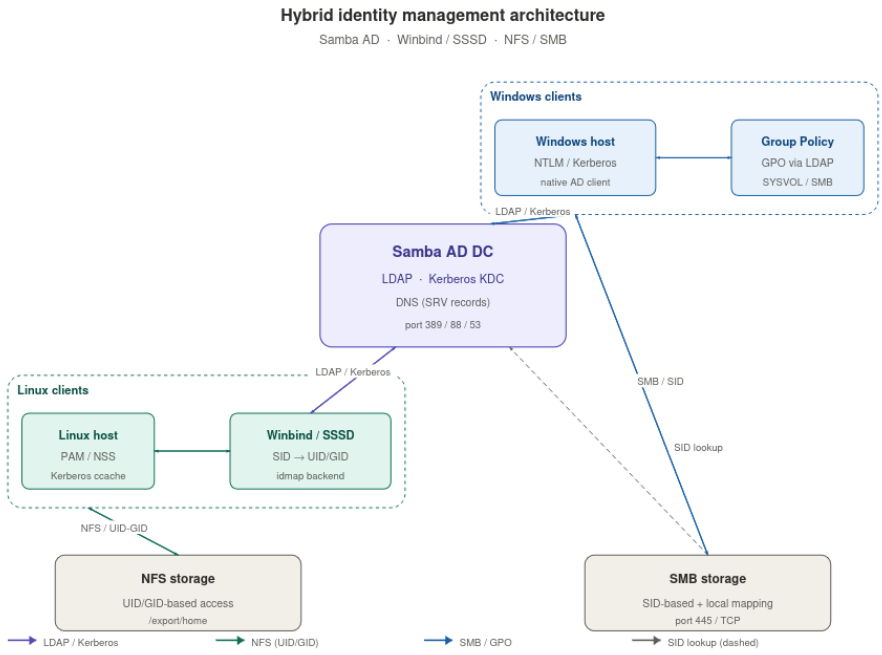

Figure 2.11: Hybrid identity management architecture: a Samba AD domain controller provides LDAP and Kerberos to both Windows and Linux clients; Winbind or SSSD mediates identity resolution on the Linux side; shared storage is accessed via NFS (UID/GID-based) or SMB (SID-based with local translation).

The choice of file sharing protocol, UID mapping strategy, and client-side daemon are interdependent. A consistent hybrid design requires making these choices together and understanding how each constrains the others, particularly at the boundary between Windows and Linux access patterns.

Chapter 3

Containerization, Service Management and Infrastructure Automation

The deployment and management of software services in networked environments has undergone a substantial transformation over the past two decades. What was once accomplished through manual configuration of physical machines has gradually given way to increasingly abstract layers of automation, virtualisation, and declarative specification. This chapter surveys the technical foundations and contemporary approaches that underpin modern infrastructure management, with particular attention to operating-system-level virtualisation, container technologies, the challenges of running stateful infrastructure services in isolated environments, and the principles of infrastructure automation. The identity and directory service layer on which these services depend is treated separately in Chapter 2; this chapter focuses on how services are packaged, deployed, and managed rather than on the protocols and data models they implement.

3.1 Operating-System-Level Virtualization

3.1.1 Evolution of Virtualization

Virtualization is older than the operating systems most people associate it with. The first working versions showed up in the 1960s, when IBM engineers on the CP-67/CMS system worked out how to split one mainframe into several isolated logical machines [23]. The goal was economic: hardware was expensive, workloads were variable, and idle cycles represented waste.

The same motivation would resurface, in different guise, some four decades later with the commercial popularisation of hypervisor-based virtualisation.

Throughout the 1980s and 1990s, the dominant model shifted away from hardware partitioning. The rapid decline in the cost of commodity x86 servers made it simpler and cheaper to dedicate physical machines to individual services. This approach introduced its own inefficiencies: servers ran at low utilisation, provisioning new capacity required physical intervention, and reproducing a software environment across machines was largely a manual, error-prone process.

Full-system virtualization came back into fashion in the 2000s, this time aimed at that same commodity hardware. VMware Workstation, Xen and later KVM showed that one physical host could run several complete operating systems at once, none of them aware of the others, with a hypervisor sitting underneath [24]. The appeal was real: consolidate workloads, clone an image to provision in minutes, snapshot for backup and recovery, contain a fault inside one guest. Cloud providers built their whole foundation on it, and metered access to virtual server capacity arrived at a scale that had not been possible before.

3.1.2 Virtual Machines versus Containers

Despite their success, hypervisor-based virtual machines carry significant overhead. Each guest instance must run a complete kernel, maintain its own device drivers, and reserve memory and CPU time for operating system processes that perform no work specific to the application being hosted. The time required to boot a virtual machine, measured in tens of seconds at minimum, further constrains the agility that modern deployment workflows demand.

Operating-system-level virtualisation takes a different approach. Rather than emulating hardware and running independent kernels, it leverages isolation mechanisms within a single shared kernel to partition user space into multiple independent execution environments. These environments, commonly called *containers*, share the host kernel and interact with it through the standard system call interface, but are restricted in what resources they can observe and access. The result is a much lighter abstraction: containers typically start in milliseconds, impose negligible CPU overhead, and avoid the memory duplication inherent in running many separate kernels [25].

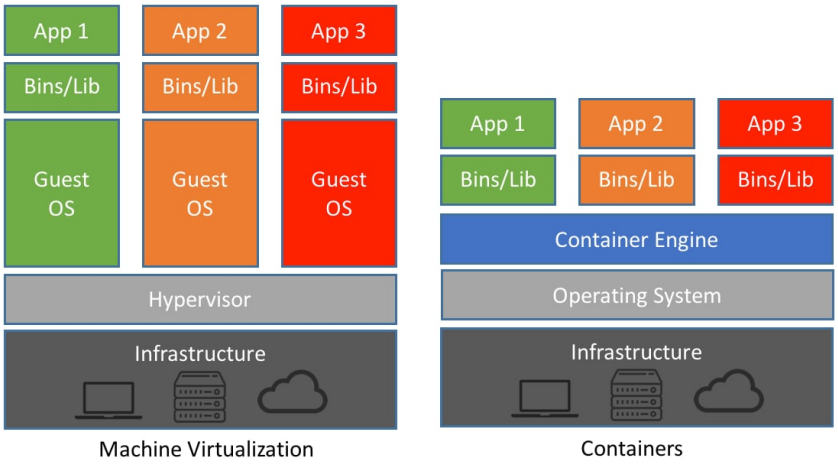

Figure 3.1: Architectural comparison between a hypervisor-based virtual machine stack and an operating-system-level container stack. Virtual machines include a full guest OS per instance; containers share the host kernel and isolate only user-space resources.

The trade-off is isolation depth. A virtual machine presents a complete hardware abstraction, making it substantially more difficult for a compromised guest to affect the host or other guests. A container relies on software-enforced boundaries within the kernel; a kernel vulnerability can in principle be exploited to escape the container context. This consideration has driven both ongoing hardening of container runtimes and the development of hybrid approaches such as gVisor [26] and Kata Containers [27], which combine container-like interfaces with stronger isolation guarantees at the cost of some of the performance advantage containers otherwise provide.

3.1.3 Linux Namespaces

The primary isolation primitive in the Linux kernel is the *namespace*. A namespace wraps a particular category of global system resource so that processes within the namespace have an independent view of that resource, isolated from processes in other namespaces. The kernel currently defines eight distinct namespace types, each controlling a different aspect of the operating environment [28].

The namespaces did not arrive all at once, and they do not all matter equally. The mount namespace (`CLONE_NEWNS`) came first, in kernel 2.4.19. It gives a process its own view of the filesystem hierarchy, with private mounts that never leak out to the host or to other namespaces, and containers use it to stand up isolated root filesystems out of layered images. UTS (`CLONE_NEWUTS`) is smaller: it lets

a container hold its own hostname and NIS domain without touching the host's. IPC (`CLONE_NEWIPC`) walls off System V semaphores, message queues and shared memory, so two containers cannot step on each other's IPC objects by accident.

PID (`CLONE_NEWPID`) gives each container its own process-ID space. The first process in a fresh PID namespace becomes PID 1 and plays init for that namespace, and anything outside the namespace is simply invisible to it. The network namespace (`CLONE_NEWNET`) is the one that matters most for serving anything. It hands each container its own interfaces, routing tables, firewall rules and sockets, and the runtime wires containers together with virtual Ethernet pairs and software bridges, so services reach each other across container boundaries without those interfaces ever showing up on the host network.

The user namespace (`CLONE_NEWUSER`) is the one that changed the security story. Stabilised in kernel 3.8, it lets an unprivileged user create containers by mapping a range of host UIDs and GIDs into a private namespace. A process inside can look like UID 0, root, while really being an ordinary unprivileged UID on the host. That is what cuts the attack surface of container operations down so far, and it is the foundation rootless runtimes are built on. The last two, cgroup (`CLONE_NEWCGROUP`) and time (`CLONE_NEWTIME`), fill in the corners, giving isolated views of the control-group hierarchy and of the system clock.

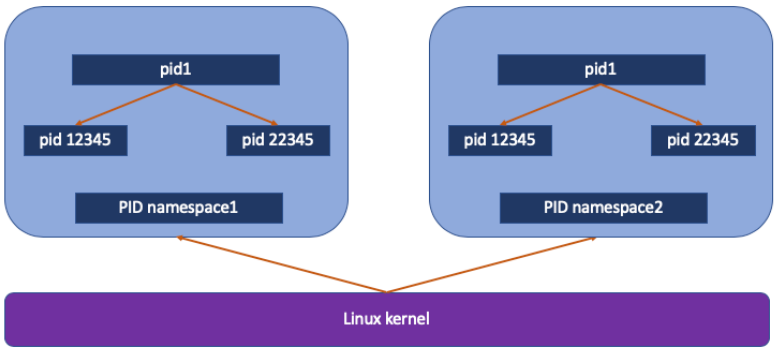

Figure 3.2: Overview of Linux kernel namespaces and the system resources each one isolates. Container runtimes typically create all or most of these namespaces for each container instance.

3.1.4 Control Groups

Namespaces govern what a process can *see*; control groups (cgroups) govern what it can *consume*. Introduced in the mainline Linux kernel with version 2.6.24 following work at Google [29], cgroups provide a hierarchical mechanism for partitioning

processes into groups and applying resource limits and accounting to those groups.

The original cgroup interface (cgroup v1) organised controllers, CPU, memory, block I/O, network bandwidth, and others into separate, independent hierarchies. This design proved flexible but complex to administer: a process could occupy different positions in different controller hierarchies, and tooling had to navigate multiple trees simultaneously.

Cgroup v2, merged into the kernel with version 4.5 and progressively adopted by distributions from around 2019 onward, unifies all controllers under a single hierarchy. The unified design simplifies resource accounting and, critically, enables delegation of a subtree to an unprivileged process, a capability essential to rootless container runtimes [30]. For container workloads the practical consequences are direct: memory limits prevent a misbehaving container from exhausting host memory and triggering the out-of-memory killer against unrelated processes; CPU quota ensures that latency-sensitive services receive adequate processor time even when co-located with compute-intensive batch jobs; block I/O throttling prevents any single container from saturating storage bandwidth. The combination of namespace-based isolation and cgroup-based resource control constitutes the foundational mechanism on which all major container runtimes are built.

3.1.5 Advantages and Limitations of Containers

The appeal of containers for service deployment is well established in the literature [31]. Their low startup latency and minimal overhead make them well suited to deployment pipelines where rapid scaling and frequent updates are requirements rather than conveniences. The ability to package an application together with its runtime dependencies into a self-contained, portable image addresses a persistent operational problem: the same image runs identically on a developer workstation, a continuous integration agent, and a production host.

Containers also encourage a discipline of immutability. Rather than modifying a running service, operators replace containers with updated images. Rollback becomes a matter of running the previous image; configuration drift is structurally less likely because the container file system is reset on each restart. The layered image format, described in Section 3.2, further promotes reuse: base images are shared across many derived images, reducing both storage requirements and the surface area affected by a security patch to the base.

Containers are not without limitations. The shared-kernel model means the host kernel version and configuration constrain what can run inside: a container requiring a kernel feature absent from the host will fail, often with an obscure error. The security boundary provided by namespaces and cgroups, while meaningful, is thinner than that of a hypervisor, and the record of container escape vulnerabilities is not short [32]. For infrastructure services that maintain persistent state, the

subject of Section 3.3, containers introduce additional complexity around storage, identity, and network addressing that must be handled explicitly by the operator rather than absorbed by the underlying operating system's service management layer.

3.2 Container Technologies

3.2.1 Standardization and the Open Container Initiative

The early history of container tooling was marked by fragmentation. Docker, which popularised containers for a broad developer audience after its public release in 2013, initially bundled all necessary components, image format, runtime, networking, and registry, into a single, tightly integrated product. As adoption grew and competing implementations emerged, the absence of common specifications threatened interoperability.

The Open Container Initiative (OCI), established in 2015 under the auspices of the Linux Foundation, addressed this through two core specifications: the OCI Runtime Specification [33] and the OCI Image Specification [34]. A third specification, the OCI Distribution Specification [35], was later added to standardise the protocol used by image registries.

The Runtime Specification defines the interface between a compliant runtime and the operating system, describing how to unpack an OCI image bundle and execute the contained process with the required namespaces, cgroups, and other configuration. The Image Specification defines the format of container images, layered file system tarballs combined with a JSON manifest and configuration, in a way that is both portable and content-addressable. Together, these specifications enable images built by one tool to be executed by any compliant runtime, and registries to serve images to any compliant client.

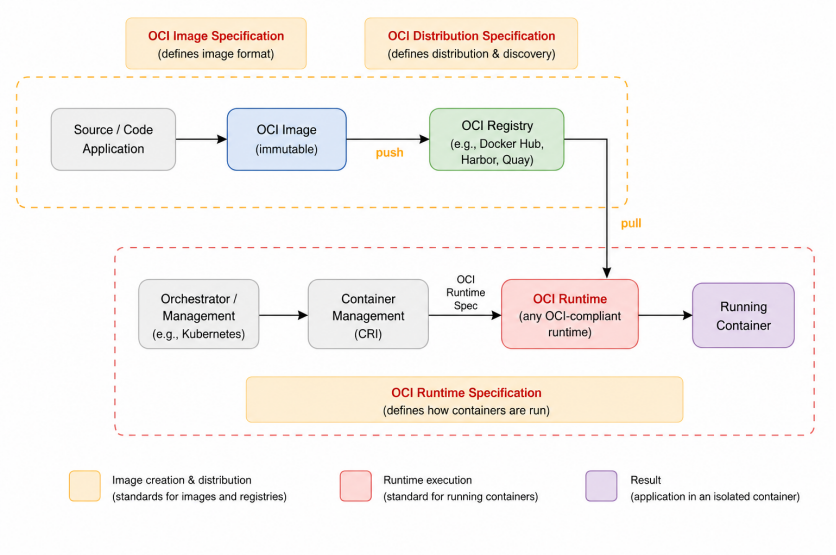

Figure 3.3: OCI architecture showing the relationship between the image specification, runtime specification, and distribution specification. Compliant tools at each layer are interchangeable without modification to adjacent layers.

The low-level reference runtime produced by the OCI project is `runc` [36], a Go implementation that translates a Runtime Specification bundle into the appropriate sequence of Linux system calls. Higher-level runtimes and container engines build on `runc` or compatible alternatives such as `crun` [37], a lighter C implementation, to provide the image management, networking, and user-facing interfaces that operators interact with directly.

3.2.2 Docker

Docker remains the most widely recognised name in container tooling, and its influence on both the technical community and popular understanding of containerisation is difficult to overstate. Released as open source in March 2013 by dotCloud (later renamed Docker, Inc.), it combined LXC-based process isolation, later replaced by its own `libcontainer`, with a layered union file system image format, a public registry (Docker Hub), and a developer-friendly command-line interface.

The Docker architecture in its current form is structured around a daemon process, `dockerd`, which manages images, containers, networks, and volumes on the host. A separate command-line client communicates with the daemon over a Unix socket or TCP connection using a RESTful API. An intermediate component, `containerd`, handles the lower-level lifecycle of containers and image transfers, delegating actual container execution to `runc` or another OCI runtime. This layered

architecture resulted from a progressive refactoring of an originally monolithic codebase, driven partly by contributions to OCI and partly by the need to integrate with orchestration systems such as Kubernetes [38].

The daemon-centric model has practical implications. Because `dockerd` runs as root by default, all container operations require root privileges at the daemon level even when the client is an unprivileged user. The daemon is also a single point of failure: if it crashes, all running containers are orphaned until it restarts. These characteristics have motivated the development of rootless Docker as a production feature, though its adoption has been uneven in comparison to purpose-built daemonless alternatives [39].

3.2.3 Podman

Podman (Pod Manager) was developed by Red Hat as a daemonless alternative to Docker, designed from the outset to operate without a persistent privileged background process [40]. Each Podman command is a self-contained invocation that interfaces directly with `conmon` (a per-container monitor process) and an OCI runtime, without routing through a centralised service. This eliminates the single point of failure and the privilege inheritance issues associated with a root daemon.

Podman's command-line interface is intentionally compatible with Docker's, enabling users to alias `docker` to `podman` in many workflows without modification. It supports the same OCI image format, the same registry protocols, and the same volume and networking semantics. Beyond compatibility, Podman adds native support for *pods*, groups of containers that share a network namespace, mirroring Kubernetes pod semantics, and integrates with `systemd` more naturally than Docker [41]. It can generate `systemd` unit files directly from running containers or pods, which is practically significant in environments that rely on `systemd` for service lifecycle management and have not adopted a dedicated orchestration platform.

A defining feature of Podman is its comprehensive support for rootless operation. By leveraging user namespaces and cgroup v2 delegation, an unprivileged user can create, run, and manage containers entirely within their own user context, without any interaction with a privileged daemon. Files created inside rootless containers are owned by the invoking user on the host, reducing the risk of privilege escalation through container escape. Rootless mode imposes constraints, notably on binding ports below 1024 and on certain Linux capabilities that require elevated privilege even within a user namespace, but these are generally manageable for production workloads through appropriate system configuration.

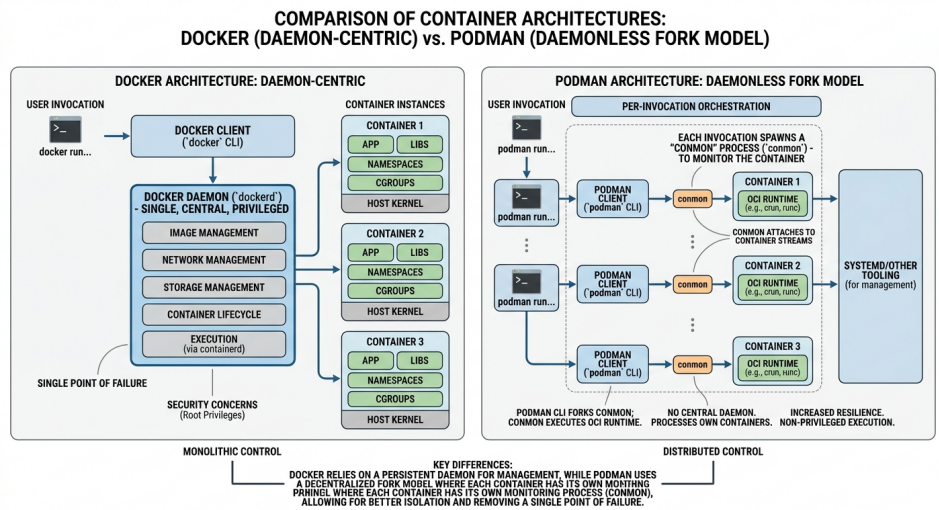

Figure 3.4: Architectural comparison between Docker and Podman. Docker routes all operations through a root daemon; Podman executes each command directly through a per-container monitor without an intermediary service.

Table 3.1: Comparison of major container engines across key dimensions.

Feature	**Docker**	**Podman**	**LXC/LXD**
Architecture	Daemon-based	Daemonless	Daemon-based
Rootless support	Partial	Full	Partial
OCI compliant	Yes	Yes	No
Pod support	No	Yes	No
Image format	OCI / Docker	OCI	LXC-specific
CLI Docker-compatible	Yes	Yes	No
Primary use case	Dev / Prod	Prod / Rootless	System containers

3.2.4 Container Images and Registries

A container image is an ordered collection of file system layers, each representing a set of changes relative to the layer below, combined with a JSON configuration file that specifies environment variables, the default command, exposed ports, and other metadata [34]. The layered model serves two purposes: it enables efficient storage, since layers common to multiple images are stored only once, and it facilitates incremental builds, where only layers affected by a change need to be rebuilt.

Images are distributed through *registries*, content-addressed stores accessible over HTTP that implement the OCI Distribution Specification. Docker Hub is the most prominent public registry, but organisations commonly deploy private registries, using software such as Harbor [42], the Docker Distribution project, or cloud-provided equivalents, to control access, enforce image signing, and reduce dependency on external network paths.

Image names follow a structured syntax: `[registry/][repository/]name[:tag]`. Tags are mutable labels that can be updated to point to different image digests, a design that has historically led to reproducibility issues when the same tag resolves to different images across environments. The OCI Image Specification's content-addressable digest, a SHA-256 hash of the image manifest, provides an immutable reference that can be used where exact reproducibility is required.

3.2.5 Security Considerations in Container Deployments

Security in container environments is a multi-layered concern. At the image level, the practice of starting from minimal base images like Alpine Linux, distroless images, or scratch images reduces the attack surface by eliminating software not required at runtime. Regular vulnerability scanning of images, integrated into CI/CD pipelines, helps detect known CVEs before images are promoted to production [32].

At the runtime level, the principle of least privilege applies directly. Containers should run as non-root users wherever possible; the Linux capability system allows granting only the specific capabilities required by the application rather than the full set associated with UID 0. Kernel security modules such as AppArmor and SELinux provide mandatory access control policies that can confine container behaviour beyond what cgroups and namespaces enforce directly. Seccomp profiles further restrict the set of system calls available to container processes, limiting the kernel attack surface exposed to potentially malicious code.

The networking model also carries security implications. Containers within a shared network namespace can communicate without restriction; explicit segmentation through separate virtual networks is necessary to enforce service boundaries and limit lateral movement in the event of a container compromise.

3.3 Containerized Infrastructure Services

3.3.1 Stateful versus Stateless Services

A foundational distinction in service architecture is between stateless and stateful services. A stateless service retains no information between successive requests: each interaction is independent, and any instance of the service can handle any

request without coordination with other instances. Web application servers that retrieve all necessary context from an external database are a canonical example. Stateless services are trivially scalable, additional instances can be added or removed without affecting correctness, and trivially replaceable, since no persistent data is lost when a container stops.

Stateful services, by contrast, maintain durable data that must survive across container restarts and, in many cases, across migrations to different hosts. A relational database, a directory service, or a network file server all fall into this category. The conventional container deployment model, in which a container is ephemeral and its writable layer is discarded when it stops, is fundamentally at odds with statefulness unless external persistence mechanisms are explicitly provided.

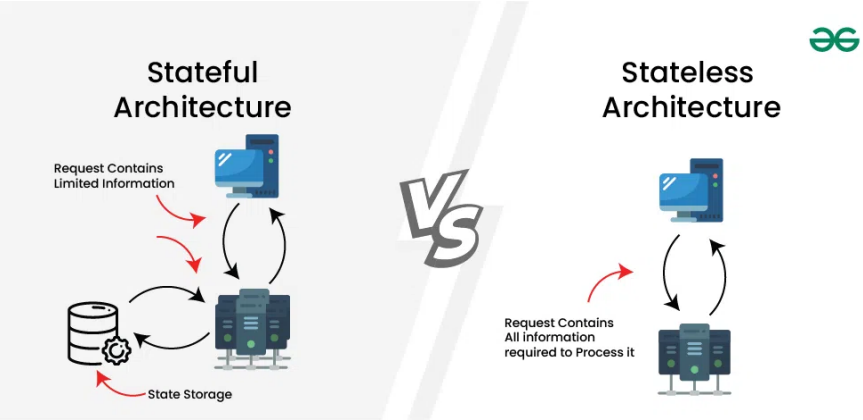

Figure 3.5: Conceptual model contrasting stateless service replicas, which share no persistent state and can be freely replaced, with stateful services that maintain durable data requiring explicit persistence management.

Container runtimes address this through the concept of *volumes*, directories or files from the host file system (or from dedicated storage plugins) mounted into the container's namespace at specified paths. Data written to mounted volumes persists independently of the container lifecycle. The OCI Runtime Specification provides the `mounts` field in the container configuration to express this binding, and all major runtimes expose volume management through their APIs and CLIs.

3.3.2 Challenges of Containerizing Infrastructure Services

Containerising infrastructure services like directory servers, authentication servers, file-sharing servers, DNS resolvers differs substantially from containerising application-tier services. Several recurring challenges characterise this domain.

Elevated kernel capabilities. Services such as a Samba Active Directory Domain Controller require Linux capabilities (`CAP_SYS_ADMIN`, `CAP_NET_BIND_SERVICE`, and others) and direct interaction with kernel subsystems that are normally absent from standard container profiles. These requirements sit in direct tension with the security principle of minimal capabilities and have historically made such services difficult to run in default container configurations. The operator must grant capabilities explicitly and verify that doing so does not violate site security policies.

Well-known port conflicts. Infrastructure services bind to ports that are both low-numbered and potentially already claimed by host services: 53 for DNS, 88 for Kerberos, 389 for LDAP, 445 for SMB/CIFS. Port publishing, mapping a specific host IP and port to a container port, resolves the conflict, but requires careful coordination, particularly when multiple containers on the same host need to expose services on different IP aliases.

Service interdependence and startup ordering. Infrastructure services rarely operate in isolation. An Active Directory domain controller provides Kerberos authentication and DNS resolution that are prerequisites for every domain-member service that runs alongside it. Containerising these services individually requires explicit management of startup ordering and readiness checking. A file server container that attempts to join the domain before the domain controller container is ready will fail in ways that are not always easy to diagnose from logs alone.

Persistent identity across replacements. Services that participate in Kerberos or LDAP depend on stable hostnames, service principal names (SPNs), and cryptographic keying material. When a container is replaced, whether due to an image update or a host migration, this material must remain consistent. The container boundary that makes stateless services easily replaceable becomes an operational complication for services whose identity is registered in a directory and whose keys are referenced by other services' ticket-granting logic.

3.3.3 Authentication and Directory Services in Containerized Environments

The identity and access management architecture relevant to this domain, LDAP, Kerberos, Samba Active Directory, Winbind, NSS, and PAM, is discussed in depth in Chapter 2. The concern here is not the protocols themselves but the additional constraints that the container boundary introduces when these services are run inside containers rather than directly on the host.

The most immediate constraint is that containers sharing a host do not automatically inherit the host's identity configuration. A container running a Samba file server must be explicitly provided with a Kerberos configuration, a resolver configuration pointing to the domain controller, and an `nsswitch.conf` that routes

identity lookups through the appropriate backend. These files are typically generated from templates and mounted into the container at known paths, rather than being embedded in the image, since their content depends on network topology and domain parameters that vary between deployments.

The Winbind daemon, when run inside a file server container, maintains its own credential cache and SID-to-UID mapping database independently of any Winbind instance on the host. This has a practical consequence: the UID space seen by the container may differ from the UID space on the host, which matters for file system permissions on directories exposed to the container through bind mounts. Coordinating the idmap configuration between host and container is therefore a deployment concern, not just a protocol concern.

Domain join operations, through which a Samba member server registers itself in the directory and acquires a machine account, must be performed inside the container rather than on the host, because the resulting keying material is stored in the container's volume. If the container is destroyed without preserving that volume, the machine account in the directory becomes stale and must be cleaned up manually before a new join can succeed.

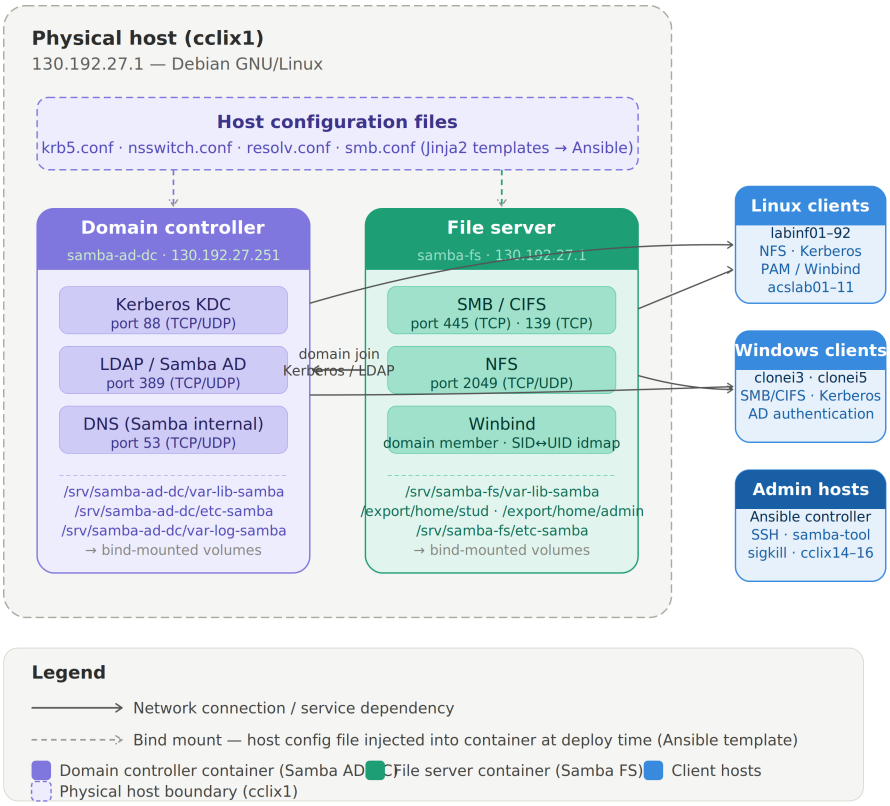

Figure 3.6: Example topology of containerised infrastructure services showing the dependency relationships between a domain controller container (Kerberos, LDAP, DNS), a file server container (SMB, NFS), and client hosts. Dashed lines indicate configuration files mounted from the host into each container.

3.3.4 File Services and Persistent Storage

Containerising a file server, whether NFS-based or SMB-based, introduces a specific storage architecture concern. The service itself is stateful in two distinct senses: it maintains runtime state (Kerberos credentials, Samba's TDB databases, lease and lock records) and it exports external storage (user home directories, shared volumes) to network clients. These two categories of data have different persistence requirements and should be managed through separate volume mounts.

Runtime state must survive container restarts but is local to the container instance and need not be shared across hosts. It is typically placed in a host directory that is bind- mounted into the container, ensuring that a container restart does not trigger a fresh domain join or discard cached authentication state.

The exported storage, by contrast, must persist entirely independently of the

container lifecycle and may need to remain accessible during container maintenance. Exposing host file system subtrees to the container through bind mounts is the standard approach. Access control on those subtrees must be evaluated carefully: the UID and GID values seen by the container's kernel operations are the values after any user namespace mapping, not the original host UIDs, and mismatches between the two can produce permission errors that are difficult to trace without understanding the full mapping chain.

3.3.5 Service Decomposition: Monolithic versus Multi-Container Approaches

A recurring architectural decision in containerised infrastructure is whether to package multiple related services into a single container or to decompose them across multiple containers, each with a narrowly defined responsibility.

The monolithic approach, running several services within one container, simplifies intra-service communication and reduces the operational overhead of managing multiple lifecycle entities. It aligns naturally with the model of traditional system administration, where a service like Samba runs alongside Winbind and Kerberos on the same host. The downside is that it sacrifices the fault isolation and independent update capability that containers are otherwise meant to provide. A configuration error in one service can take down co-located services, and updating the image requires restarting all of them simultaneously.

The multi-container approach assigns each logical service to its own container, communicating over virtual networks. This design aligns with the microservice philosophy but requires explicit management of inter-container dependencies, service discovery, and shared configuration. For tightly coupled infrastructure services such as a Samba AD DC whose DNS, LDAP, and Kerberos subsystems are implemented by the same process and share the same internal databases, decomposing them into separate containers introduces coordination complexity that offers little operational benefit. Practitioners in this space therefore often adopt a middle ground: separating services with clearly independent lifecycles (a domain controller from a file server, for example) while keeping components that must share internal state co-located within the same container.

3.4 Infrastructure as Code and Automation

3.4.1 Principles of Infrastructure as Code

Infrastructure as Code (IaC) denotes the practice of managing and provisioning computing infrastructure through machine-readable configuration artefacts rather than through manual procedures or interactive tooling [43]. The term encompasses a

spectrum of activities: configuring operating system settings and installed packages, deploying application services, defining network topology and firewall rules, and provisioning cloud resources. What unifies these activities under the IaC umbrella is the treatment of infrastructure specification as a first-class software artefact, subject to version control, code review, automated testing, and continuous integration practices.

The motivations for IaC are closely tied to the operational challenges of scale and repeatability. In an environment with a handful of servers, experienced administrators can apply consistent configurations through careful manual procedure. As the number of managed hosts grows, whether to tens, hundreds, or thousands, manual consistency becomes untenable. The probability of divergence between hosts increases with each unrecorded change; the effort required to apply a patch grows linearly with host count; and undocumented decisions accumulate into a body of tacit knowledge that is easily lost when personnel change. IaC addresses these problems by externalising infrastructure knowledge into explicit, versioned artefacts that define the desired state of the system.

A concept central to IaC is *idempotency*: an operation applied multiple times should produce the same result as applying it once. Idempotent operations allow operators to converge a system to its intended state without accounting for the current state or the history of previous operations, which simplifies both initial provisioning and ongoing drift remediation.

3.4.2 Configuration Management: Declarative versus Imperative Approaches

Configuration management tools operate on a model of *desired state*: the operator specifies what a system should look like, and the tool determines what changes are necessary to achieve that state from the current one. This contrasts with imperative scripting, in which the operator specifies a sequence of commands to execute, without built-in knowledge of the current state. A shell script that installs a package with `apt install` is not idempotent but a configuration management task that ensures a package is present is. Declarative specifications are generally more robust to partial failures and re-execution, since they converge toward the target rather than blindly replaying a fixed sequence that may have already been partially applied.

In practice the boundary between declarative and imperative is not sharp. Most declarative tools provide escape hatches, shell command execution, raw script blocks for operations that cannot be expressed cleanly through the module system. The discipline of minimising the use of these escape hatches, and ensuring that any imperative steps are themselves written to be idempotent, is part of what distinguishes well-maintained IaC from infrastructure automation that merely

happens to be written in YAML.

3.4.3 Ansible

Ansible, initially released in 2012 by Michael DeHaan and subsequently acquired by Red Hat in 2015, has become one of the most widely used configuration management and provisioning tools in the industry [44]. Its design philosophy emphasises simplicity and low barrier to entry: configuration is expressed in YAML; the control machine requires only an SSH client and a Python interpreter; managed nodes require only Python and an SSH server, with no persistent agent process.

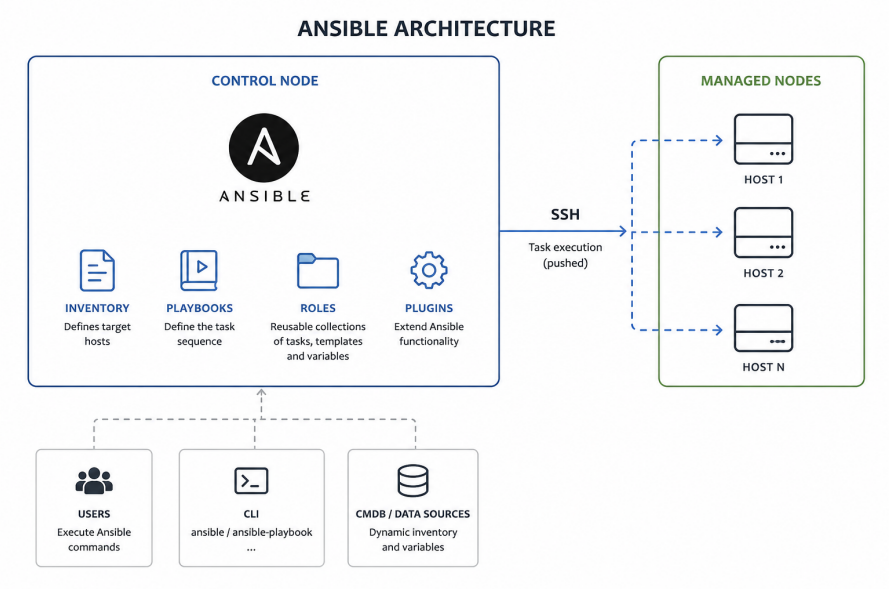

Figure 3.7: Ansible architecture: the control node pushes task execution to managed nodes via SSH. The inventory defines the target hosts; playbooks define the task sequence; roles encapsulate reusable collections of tasks, templates, and variables.

Ansible's central abstraction is the *module*: a self-contained unit of automation logic that encapsulates a specific idempotent operation, such as creating a file, installing a package, starting a service, or managing a container. Modules communicate their result as JSON, reporting whether any modification to the system was made through a `changed` flag that Ansible interprets for reporting and handler notification. Hundreds of built-in modules cover common system administration tasks; community collections extend this with support for cloud providers, network devices, and specialised applications such as Podman container management.

Playbooks are YAML files that define an ordered sequence of tasks to be applied to a set of hosts. Tasks are grouped into *roles* that are reusable, self-contained units that package tasks, templates, variables, and handlers together enabling code sharing across playbooks and projects [45]. The *inventory* defines the set of hosts that Ansible manages, organising them into groups that can be targeted selectively by playbooks. Inventories may be static YAML or INI files, or dynamic scripts that query an external source such as a cloud provider API.

Handlers are tasks triggered only when notified by another task, typically to restart a service after its configuration file changes. A handler is invoked at most once per playbook run regardless of how many tasks notify it, which avoids unnecessary service restarts. *Facts* are host-specific data collected automatically at the start of a playbook run, CPU count, memory size, network interface addresses, and so on, and are available as variables within subsequent tasks and templates.

Templates, rendered through the Jinja2 engine, allow configuration files to be generated dynamically from variables. The same template file produces different output for each host or group, depending on the variables in scope, a mechanism heavily used when deploying services whose configuration depends on network topology or domain parameters that differ between environments.

The `ansible-vault` facility encrypts sensitive data as passwords, private keys, API tokens within playbooks and variable files, allowing secret material to be stored in version control repositories without exposure in cleartext [46].

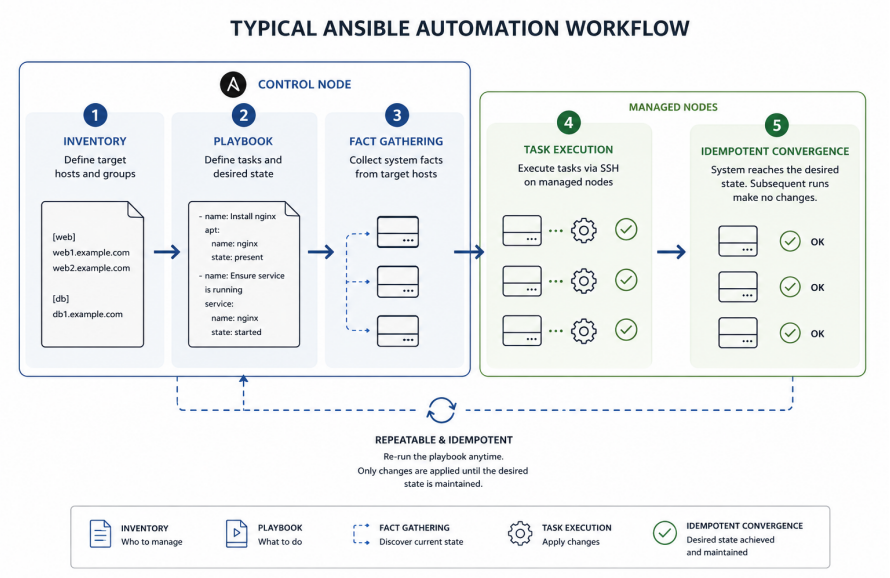

Figure 3.8: Typical Ansible automation workflow: from inventory and playbook definition, through fact gathering and task execution on managed nodes, to idempotent convergence of the target state.

3.4.4 Alternative Automation Tools

While Ansible enjoys broad adoption, several alternative tools merit consideration, each embodying different trade-offs in architecture, language, and operational model.

Puppet, developed by PuppetLabs (now Perforce) from 2005 onward, follows an agent-based, declarative model [47]. A Puppet agent runs on each managed node, periodically requesting a compiled *catalogue* from a central Puppet Server. The catalogue specifies the desired state of all resources on the node, and the agent applies any necessary changes independently of the control machine. Puppet uses its own domain-specific language for expressing resource configurations, which offers expressive power at the cost of a steeper learning curve than YAML.

Chef, originally released in 2009 by Opscode, also follows an agent-based model but expresses configuration in Ruby through constructs called *recipes* and *cookbooks* [48]. The use of a full programming language gives operators fine-grained control over configuration logic but can complicate adoption in teams without Ruby expertise and makes static analysis of the configuration harder.

Salt (SaltStack), first released in 2011, occupies a middle ground [49]. It supports both an agent-based (minion) model and an agentless SSH-based mode analogous to Ansible's approach. Salt uses ZeroMQ for high-performance communication between master and minions, which gives it advantages in large fleets where polling latency or fan-out performance matters. State definitions are expressed in YAML with Jinja2 templating, making the syntax familiar to anyone who has worked with Ansible.

Table 3.2: Comparison of major infrastructure automation tools.

Feature	**Ansible**	**Puppet**	**Chef**	**Salt**
Architecture	Agentless (push)	Agent (pull)	Agent (pull)	Agent or agentless
Config language	YAML / Jinja2	Puppet DSL	Ruby (DSL)	YAML / Jinja2
Idempotent	Module-dependent	By design	Resource-based	By design
Agent on nodes	Not required	Required	Required	Optional
Learning curve	Low	Medium	High	Medium
Scalability	Medium	High	High	Very high
Secrets	ansible-vault	Hiera / r10k	Chef Vault	Pillar / vault
Community	Very large	Large	Large	Moderate

Each of these tools reflects different assumptions about the scale and structure of the environments for which it is intended. Ansible's agentless architecture and shallow learning curve make it a natural starting point for teams seeking rapid adoption at moderate scale. Puppet and Chef's agent-based models offer stronger

guarantees of continuous convergence and scale more gracefully to large node counts, since agents apply configuration locally without waiting for a push from the control machine. Salt's performance characteristics make it attractive in environments with thousands of nodes where latency on configuration delivery is a real operational concern.

3.5 Modern Infrastructure Management

3.5.1 Service Lifecycle Management

Contemporary infrastructure management extends well beyond the initial deployment of services. The operational lifecycle of a service that includes provisioning, configuration, monitoring, upgrading, and eventual decommissioning must be managed explicitly, and the tooling choices made at each phase influence what is feasible at the others.

In environments that combine containerised services with traditional host-level services, lifecycle management is complicated by the need to coordinate across multiple management planes. Container runtimes manage container processes; init systems such as `systemd` manage host-level services; configuration management tools ensure that both levels remain in the intended state. A common integration point is the generation of `systemd` unit files from container definitions: Podman, for example, can produce a unit file for any running container or pod, which `systemd` then uses to start, stop, and monitor the container as it would any other service. This approach avoids reliance on a separate container daemon while still giving operators a familiar interface for service supervision [41].

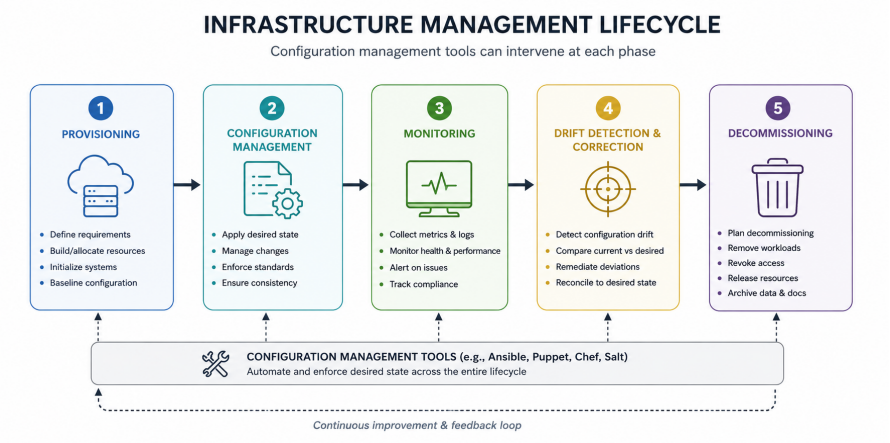

Figure 3.9: Infrastructure management lifecycle from initial provisioning through ongoing configuration management, monitoring, and drift correction to controlled decommissioning. Configuration management tools can intervene at each phase.

3.5.2 Monitoring and Observability

The modern approach to infrastructure monitoring distinguishes between three categories of signal. *Metrics* are quantitative, time-series measurements of system behaviour: CPU utilisation, memory pressure, request throughput, error rates. *Logs* are structured or unstructured records of discrete events, useful for diagnosing specific failures after the fact. *Traces* record the path a request takes through distributed system components, enabling latency attribution across service boundaries. The conjunction of all three is sometimes described as *observability*, the ability to infer the internal state of a system from its external outputs without requiring prior knowledge of which failure modes to look for [50].

For containerised services, metrics collection must account for the container boundary. Prometheus [51] and its ecosystem of exporters provide a pull-based model in which a central metrics server scrapes endpoints exposed by services, including runtime exporters that surface cgroup-level resource consumption for individual containers. Log aggregation systems such as the ELK stack (Elasticsearch, Logstash, Kibana) or Loki can centralise logs from multiple containers and hosts, enabling cross-service correlation that would otherwise require manually inspecting log files on multiple systems.

The challenge specific to infrastructure services is that many of them like LDAP servers, Kerberos KDCs, file servers, were not designed with metrics export in mind. Monitoring them at the service level often requires external probes: LDAP search latency measurements, Kerberos ticket acquisition timing, SMB connection counts. These must be instrumented separately from the container runtime metrics, and

the interpretation of their results requires domain knowledge of what constitutes normal and abnormal behaviour for the specific service version and workload.

3.5.3 Reproducibility and Configuration Drift

A well-maintained IaC codebase acts as executable documentation of the infrastructure's intended state. Changes are tracked through version control commits, proposed through merge requests, and auditable through history. When infrastructure is defined entirely in this way, reproducing a deployment on a new host, or verifying that an existing host matches its specification is a matter of running the automation against it.

Full reproducibility is harder to reach than it sounds. The automation captures the configuration you meant to have, but not everything that accumulates around it: data that piles up, the emergency fix someone made at 2am, the change pushed through a tool that never touched the IaC workflow. The gap between what the system actually is and what the code says it should be is configuration drift, and it does not close on its own. Agent-based tools like Puppet and Chef lean on continuous convergence to keep it in check. An agent wakes on a schedule, compares the host against the catalogue, and quietly corrects whatever has slipped since last time. Ansible has no agent doing that, so drift only gets caught when someone schedules a run; leave the runs out and it builds up unseen.

Secrets are their own headache here. Service-account passwords and private keys have to be reachable from the infrastructure code for provisioning to be fully automated, and they absolutely cannot sit in cleartext in version control. The usual answers are encryption at rest, Ansible Vault for secrets embedded in playbooks, HashiCorp Vault [52] for centralised storage and credentials minted on demand, which let secrets live alongside the code while keeping access control and an audit trail. Rotating them through those mechanisms instead of baking them statically into images or config files has stopped being an advanced trick and become the baseline expectation.

3.5.4 Hybrid Linux/Windows Environments

Most institutional and enterprise environments are heterogeneous. The operational challenges that arise at the boundary between Linux and Windows systems are well documented [53], and they do not disappear when services are containerised, they are in some cases amplified, because the container boundary adds another layer of identity and configuration that must be kept consistent with both sides.

The identity management aspects of hybrid environments, SID-to-UID mapping, Winbind and SSSD configuration, Kerberos realm integration, and the divergence between POSIX and Windows ACL models are discussed in Chapter 2. From the

infrastructure management perspective, the relevant concern is how to provision and maintain these configurations consistently across many hosts using IaC tooling. Template- based configuration management is particularly well suited to this: a Jinja2 template for `krb5.conf` or `nsswitch.conf`, parameterised by domain name and domain controller address, can generate host-specific configuration files that are correct by construction, eliminating the inconsistencies that accumulate when files are maintained manually on each host.

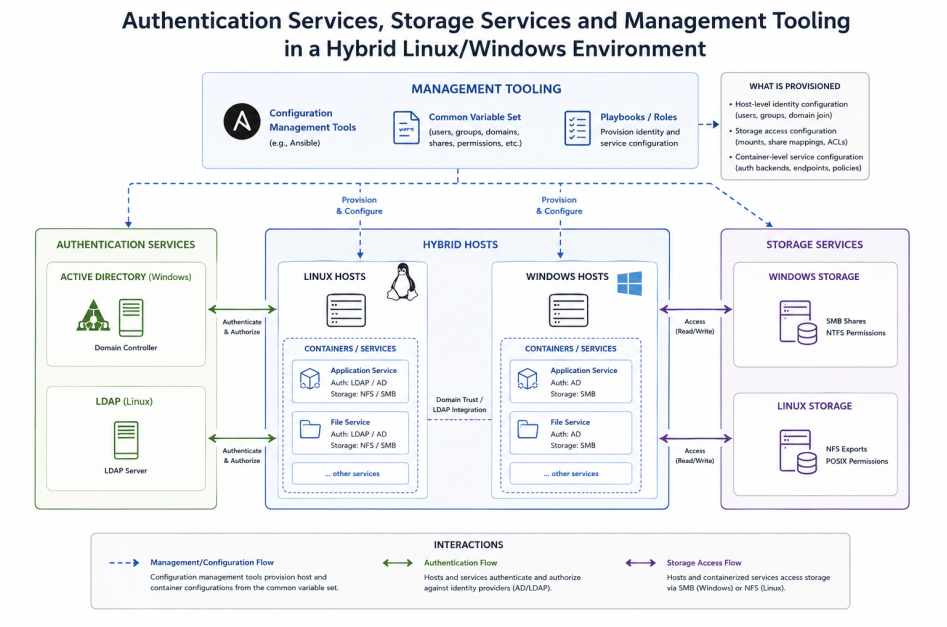

Figure 3.10: Interaction between authentication services, storage services, and management tooling in a hybrid Linux/Windows environment. Configuration management tools provision both the host-level identity configuration and the container-level service configuration from a common variable set.

Container deployments in hybrid environments also introduce the question of where domain join operations and certificate management should happen. If a containerised file server must be a domain member, the join and the keying material it produces must be handled as part of the container provisioning workflow, not as a manual post-deployment step. IaC tooling that orchestrates container creation and domain join in sequence, checking for an existing machine account before attempting a new join, is the approach most consistent with reproducible deployments.

3.5.5 Scalability, Maintainability, and Automation Maturity

The literature on infrastructure automation distinguishes between environments that use automation as a convenience, to reduce typing, and those that treat it as an architectural constraint, where no change to the infrastructure is considered valid unless it can be expressed in the automation codebase and applied through it. The latter model, sometimes called *automation-first* or *infrastructure as code in earnest*, is associated with lower rates of configuration drift, faster incident response (because the remediation path is already encoded), and easier onboarding of new administrators [43].

Maintainability of the automation codebase itself is a concern that receives less attention in the literature but is operationally significant. Ansible playbooks and container image definitions accumulate technical debt just as application code does. Without clear module boundaries, well-named variables, and documented rationale for non-obvious decisions, a playbook written under time pressure becomes difficult to modify safely six months later. The separation of host-level configuration from service-level configuration, achieved structurally through the container boundary, supports maintainability by limiting the scope of changes: a modification to a container image affects only the services running from that image, while a host-level playbook change affects all hosts to which it is applied but does not touch the services running inside containers.

Testing infrastructure code is an area where practice has matured significantly in recent years. Tools such as Molecule [54] allow Ansible roles to be tested in ephemeral container or VM instances, verifying that tasks produce the expected state without requiring access to a production environment. Container image builds can be similarly tested through CI pipelines that execute the build, run the resulting image, and verify service availability. The adoption of these practices is associated with reduced regression rates when infrastructure changes are applied to production systems [43].

Chapter 4

Analysis of the Existing Infrastructure and Migration Goals

4.1 System Inventory

The Laboratorio di Informatica (LABinf) at the Politecnico di Torino supports practical courses in computer engineering, hosting roughly ninety student workstations distributed across classroom-style rows plus a smaller set of administrative machines. All of them depend, for login, file access, and printing, on a shared server infrastructure whose architecture has not changed substantially since it was first deployed.

Two servers play a peripheral role. `cclix4` serves as a lightweight HTTP and FTP frontend for course material distribution. `cclix8` is a QNAP NAS unit used for secondary storage and occasional backup operations. Neither is part of the authentication chain, and neither is within the scope of this migration.

The machine that everything else depends on is `cclix1`, reachable at 130.192.27.1. It is a Sun Fire x4140, physically present in the lab, running Debian GNU/Linux in a pre-12 release. Its single NIC (MAC F8:F2:1E:17:05:C0) carries all network traffic for all the services it exposes. A port scan of the machine gives an immediate sense of how much is running: SSH on 22, the RPC endpoint mapper on 111, NetBIOS and Samba on 139 and 445, LDAP and LDAPS on 389 and 636, IPP printing on 631, NFS on 2049, and a Nagios NRPE monitoring agent on 5666. This is not a dedicated file server or a dedicated domain controller; it is all of those things simultaneously, on one host, with no boundary between them.

4.2 Service Architecture on cclix1

The authentication domain is a Samba NT4-style domain, not an Active Directory deployment. The distinction is significant: the NT4 domain model, described in chapter 2, uses a simplified Windows NT protocol stack rather than Kerberos-based authentication, and its capabilities for access control, delegation, and schema extension are correspondingly limited. The workgroup name in `smb.conf` is `LABINF-NG`, and the server identifies itself as an "LDAP-based domain controller", which captures the other structural choice in this architecture: the password backend is not Samba's native TDB store but an OpenLDAP instance running on the same host.

The relevant `smb.conf` directives are:

```
passdb backend      = ldapsam:"ldap://localhost"
domain logons       = yes
domain master       = yes
ldap suffix         = dc=labinf,dc=polito,dc=it
ldap user suffix    = ou=People
ldap group suffix   = ou=Groups
ldap idmap suffix   = ou=Idmap
ldap machine suffix = ou=Computers
```

Samba delegates all account lookups, group membership queries, and password verifications to OpenLDAP. The LDAP tree at `dc=labinf,dc=polito,dc=it` organises records into separate OUs: `ou=People` holds user accounts with both POSIX and Samba-specific attributes, `ou=Groups` holds group entries, `ou=Idmap` maintains the mapping between Windows SIDs and UNIX numeric identifiers, and `ou=Computers` stores machine accounts for domain-joined workstations.

The tool chain that coordinates writes across Samba and OpenLDAP is `smbldap-tools` [55]. When the Ansible-equivalent manual provisioning adds a user, the `add user script` directive points to `smbldap-useradd`, which creates the LDAP entries with the correct object classes. Analogous scripts handle group operations and machine account management. The Samba daemon itself never writes to OpenLDAP directly; it goes through these scripts.

Beyond authentication, cclix1 runs several other services that serve the lab daily. NFS exports home directories and course content to Linux clients across the subnet. CUPS listens on port 631 and handles print jobs for the lab's printers, with a small NFS re-export of printer-related content. An NTP daemon acts as time source for the subnet, which becomes relevant in an Active Directory environment where Kerberos ticket validity depends on clocks being synchronised. Network access control relies on TCP Wrappers: `/etc/hosts.allow` enumerates which services accept connections from which addresses, and `/etc/hosts.deny` sets the default-deny policy. Remote monitoring is provided by a Nagios NRPE agent on

port 5666, which exposes health checks to an external Nagios server.

All of these services share the same process tree, the same filesystem namespace, and the same network interface. There is no isolation between them: a problem in one propagates freely to the others.

4.3 Identity Management and User Provisioning

The identity model is split across two stores that must remain consistent by design but have no mechanism to enforce that consistency. OpenLDAP holds the canonical records: POSIX attributes (`uid`, `uidNumber`, `gidNumber`, `homeDirectory`, `loginShell`) combined with the Samba-specific `sambaSamAccount` object class, which carries the NT and LM password hashes and the user's Windows SID. Samba's own TDB files maintain runtime state that partially overlaps with some of this information. The `ou=Idmap` subtree holds an additional mapping table that Samba consults when it needs to translate a Windows SID to a UNIX UID.

The domain group structure maps to POSIX GIDs in the expected way: Domain Admins at GID 512, Domain Users at 513, Domain Guests at 514, Domain Computers at 515, and the built-in Administrators group at 544. Two lab-specific groups extend this: `netadmins` at GID 516, which includes lab staff accounts, and `netusers` at GID 517, which includes student accounts. This distinction between staff and students drives access control decisions for the shared file areas.

Provisioning a new student account is a sequence of steps with no transactional boundary. A Python script calls `ldapmodify` to create the user entry under `ou=People`, then calls `ldapmodify` again to add the user to the appropriate group entries, then invokes `smbpasswd` to set the initial Windows password hash. Each of these operations touches a different part of the system, and none of them knows whether the others have succeeded. If the `smbpasswd` call fails after the LDAP entries have been written, the account exists for POSIX and NFS purposes but cannot authenticate to the Windows domain. Detecting the inconsistency requires querying both systems separately.

The Python scripts themselves are a layered accumulation of workarounds written over time to handle edge cases encountered in production. They call `ldapmodify` and `smbpasswd` as subprocess invocations, parse their output with ad-hoc string matching, and have no formal error model. There is no input validation layer and no abstraction between the script and the raw LDAP schema. Adding a field to the user object, for example, requires editing the scripts directly.

4.4 File and Network Service Access

The way a client accesses its resources depends entirely on the operating system it is running. Windows workstations join the `LABINF-NG` domain and authenticate via NTLM. Once authenticated, they receive access to SMB shares defined in `smb.conf`: `[homes]` maps each user to their personal directory, `[netlogon]` delivers the logon script (`netlogon.bat`) that maps network drives at session start, `[msdn]` serves course software, `[admin]` is accessible to lab staff, and `[corsi]` holds course-specific materials. From the Windows client's perspective, these appear as ordinary network drives. The NFS layer underneath is completely invisible.

Linux clients follow a different path. Authentication is handled by PAM and NSS configurations that bind to the local OpenLDAP instance and resolve user accounts from the same LDAP tree that Samba uses. Home directories are mounted over NFS from `/export/home/stud`, course content from `/export/home/corsi`, and administrative files from `/export/home/admin`. A Xilinx software directory is also exported for specific machines. NFS access control is enforced through TCP Wrappers: the allowed subnets are `130.192.27.0/24` and `192.168.1.0/24`, with additional per-host rules for clone machines and administrative workstations.

Both access paths, SMB for Windows, NFS for Linux, land on the same physical directory tree under `/export`. A file written through an SMB session and a file written through an NFS mount exist in the same directory. This convergence is what makes the single-host architecture coherent: there is no synchronisation between storage backends because there is only one storage backend. The risk, as mentioned in the previous section, is that the UID/GID mapping must agree between the two protocols, and that agreement is maintained manually rather than enforced structurally.

Printing is accessible from both operating systems through CUPS on port 631. The lab's printers are managed centrally on cclix1 and queued through the IPP interface. The `/export/cups` NFS path provides ancillary print-related files to specific hosts.

4.5 Weaknesses of the Existing System

The authentication protocol stack is the most visible security problem. After a Samba upgrade from version 4.2 to 4.5, Windows 10 2016LTSB clients lost the ability to log in to the domain. The immediate fix was to force the server's maximum SMB dialect to `NT1` and explicitly re-enable NTLMv1 authentication with `ntlm auth = Yes`. Both directives appear in the production `smb.conf`. The practical consequence is that every client in the lab authenticates using a protocol whose weaknesses have been extensively documented [56]: NTLMv1 is vulnerable

to offline dictionary attacks against captured challenge-response exchanges, and pass-the-hash techniques allow an attacker who has extracted a hash from one machine to authenticate as that user elsewhere on the network without knowing the password. The upgrade that triggered the regression was never addressed properly; security was degraded in exchange for restored functionality and left that way.

The situation around CVE-2017-7494 is arguably worse, because it is documented inside the production configuration itself. That vulnerability allows a client with write access to an SMB share to upload a shared library and cause the server to load it, resulting in remote code execution [57]. A writable share, an unpatched remote code execution vulnerability, and NTLMv1 authentication together represent a risk profile that is difficult to describe charitably.

The split identity store is a structural problem that manifests operationally. Samba and OpenLDAP each maintain records about users and groups, and those records must agree for the system to behave correctly. The smbldap-tools package that coordinates them has not seen active development in years, which means that any incompatibility introduced by a future Samba release will not be fixed in the toolchain. Provisioning operations are not atomic: a user creation that writes the LDAP entries successfully but fails before the `smbpasswd` call completes leaves an account that is visible to NFS and PAM but cannot authenticate to the Windows domain. There is no recovery procedure for this state other than identifying the affected accounts manually and completing the missing steps by hand.

Service isolation does not exist in any form. The domain controller, file server, NFS daemon, CUPS server, and NTP process all run on the same kernel, in the same process tree, sharing the same filesystem. A runaway log file that fills the disk affects every service simultaneously. A Samba configuration error that crashes `smbd` takes the NT4 domain down; but because NFS authentication also goes through LDAP on the same host, the NFS sessions for Linux clients may also become unstable. The blast radius of any failure is the entire server.

The underlying Debian release is outdated by more than one major version. This is not just a cosmetic issue. Certain package versions needed for a modern Samba or Kerberos deployment are simply not available from the repositories pinned to this release. Security patches for the kernel, for OpenSSL, and for the Samba libraries arrive through the distribution’s backport schedule, which lags behind upstream. And some packages cannot be upgraded independently because their dependencies are tied to the version of glibc that ships with the installed Debian release.

The lab maintains a wiki intended to record configuration changes and operational decisions. In principle this provides continuity across staff turnover and a reference when troubleshooting. In practice, the wiki was kept current during some periods and neglected during others, with the result that its coverage is uneven and its reliability cannot be assumed. The gap between what is documented there and

what is actually running on the server is unknown without direct inspection of both. This is not a criticism of the individuals involved, maintaining documentation in parallel with operational work requires sustained discipline, and that discipline is difficult to preserve under time pressure. It is, rather, an argument that documentation which depends entirely on manual effort and goodwill is structurally fragile. An Ansible-managed infrastructure sidesteps the problem: the playbooks that deploy the system are its documentation, and they are necessarily current because they are what produces the running state. There is no separate act of documentation that can be skipped.

A final constraint shapes the migration strategy fundamentally: moving from the NT4 domain to an Active Directory domain requires changing the LDAP base DN from `dc=labinf,dc=polito,dc=it` to `dc=labinfad,dc=polito,dc=it`. Samba Active Directory provisions its own internal database under the new DN and has no import path from the old OpenLDAP tree. This means the migration is a cutover, not an upgrade. The existing domain is decommissioned, the new domain is provisioned from scratch, and user accounts are recreated in the new schema. There is no partial migration and no rollback once clients begin joining the new domain.

4.6 Migration Objectives

The objectives defined here are direct responses to the problems identified above. Each one targets a specific operational failure mode rather than a general preference for newer technology.

The primary objective is replacing the NT4 domain with a Samba Active Directory domain using Kerberos-based authentication [15]. This eliminates the NTLM dependency at the root. Kerberos mutual authentication removes the attack surface that NTLMv1 exposes, and the AD protocol stack enables modern SMB dialects without the compatibility compromises that triggered the protocol downgrade. Active Directory also provides a richer access control model, Group Policy infrastructure, and a schema that can be extended for POSIX attribute storage.

Service isolation is the second priority. Running the domain controller, file server, and NFS daemon as separate Podman containers on the same physical cclix1 hardware, requiring no procurement, introduces containment boundaries without additional cost. As discussed in chapter 3, containers provide process and filesystem isolation within the bounds of what the host kernel allows. A misconfiguration in the file server container does not affect the domain controller container; a container restart does not propagate across service boundaries. The blast radius of any individual failure becomes bounded and predictable.

The entire deployment must be driven by Ansible. This is a direct response to the absence of any reproducible procedure for the current system. The playbooks and templates that Ansible uses to deploy the infrastructure are also its documentation: running them on a fresh host should produce the same result as running them on the production server. Every configuration choice becomes visible, version-controlled, and auditable.

Identity management needs to be consolidated into a single authoritative store. Rather than keeping Samba's TDB files and OpenLDAP's records in approximate synchronisation via smbldap-tools, the new architecture stores all identity information inside the Active Directory database, extended with RFC 2307 POSIX attributes [19]. A single store means no synchronisation problem and no atomicity gap. User management operations happen through `samba-tool`, which writes all required attributes in one operation.

Directory queries from Linux clients must be encrypted. The current system allows plaintext LDAP on port 389 from within the lab subnet. In the new architecture, the domain controller exposes a CA-signed TLS certificate, and SSSD on client machines is configured to require `STARTTLS` before binding. This protects both credentials and directory contents from interception on the local network.

The operating system on cclix1 needs to be updated to a supported Debian release as a prerequisite for the above: modern Samba versions, current Kerberos libraries, and the Podman packages that the container architecture depends on are not available from the repositories that ship with the installed release.

Finally, the new architecture must maintain full compatibility for existing clients. Windows workstations continue to join a domain and access SMB shares with the same names they already know. Linux clients continue to mount home directories over NFS. From the end user's perspective the transition should be transparent.

Chapter 5

Logical Project Architecture

5.1 Architectural Overview

Everything in the new design still runs on cclix1. No new hardware was bought, and none was needed, the migration is about how the machine is carved up internally, not about buying capacity. The single process tree of chapter 4, where one Samba instance did everything, gives way to three Podman containers. Each owns one service layer. They run side by side on the same host but cannot see into each other, kept apart by container boundaries and by two separate bridge networks.

`samba-ad-dc` is the Active Directory domain controller. `samba-fs` is a Samba member server that joins the domain `samba-ad-dc` provides and serves the SMB shares. `nfs` exports the same files over NFS to the Linux clients. The two Samba containers are built from a Debian Bookworm image trimmed to what each role needs; the NFS container is built on erichough/nfs-server image.

The domain controller is not exposed through cclix1's primary IP address. A secondary IP alias, 130.192.27.251, is assigned to the physical NIC at boot time, and Podman's port forwarding routes AD-protocol traffic arriving at that address into the DC container's internal network. The primary address, 130.192.27.1, continues to carry SMB and NFS traffic. This separation means that DNS, Kerberos, and LDAP traffic from domain members never competes with file access traffic for the same port space.

Ansible owns all of it: the images, the networks, the provisioning, the DNS records, the accounts, the password policy, down to the last template. The playbooks are not documentation kept beside the system. They are the system's only definition, and running them against a clean Debian install rebuilds production from nothing.

Figure 5.1 shows the overall layout.

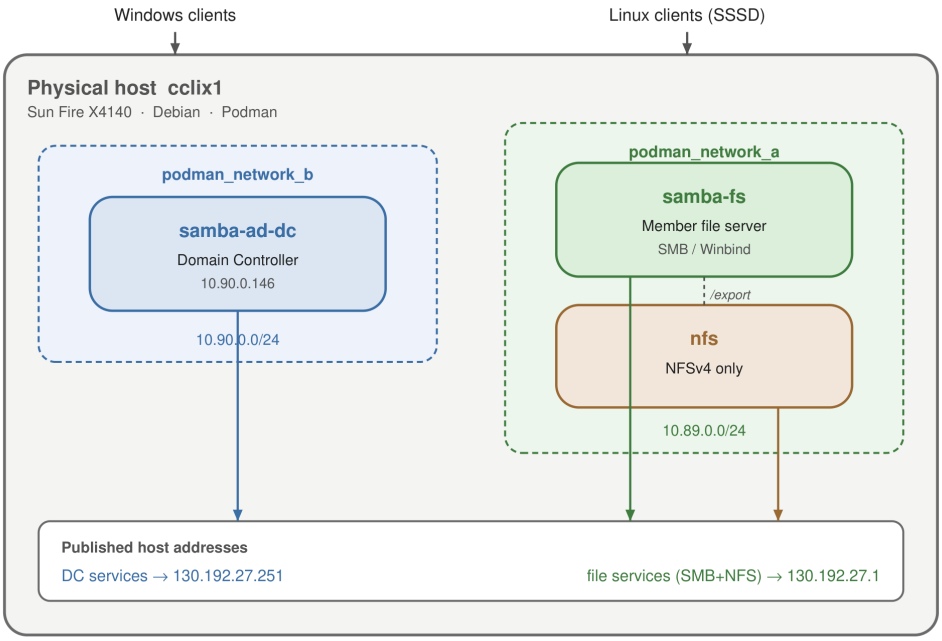

Figure 5.1: High-level architecture of the new infrastructure. Three Podman containers run on cclix1, isolated through two internal bridge networks. External clients reach the domain controller at 130.192.27.251 and all file services at the primary address 130.192.27.1.

5.2 Service Decomposition

The decision to run three separate containers rather than a single combined Samba instance came from operational considerations, not from principle.

A Samba AD domain controller and a Samba member file server have different runtime characteristics and different failure modes. The DC is the authentication root for every client in the lab: restarting it, even briefly, disrupts Kerberos ticket issuance for all domain members at once. The file server can be restarted independently, for a configuration change, a Samba upgrade, or a share redefinition, without affecting the DC at all, as long as the DC remains up during the restart. In the old system, where everything was a single Samba process on a single host, this kind of targeted restart was impossible. Any intervention touched everything.

Separating the NFS container from the file server container addresses a different concern. The NFS daemon inside the container needs to interact with the kernel's NFS filesystem layer to register export tables and manage mounts, which requires the `SYS_ADMIN` capability and a privileged container mode. Bundling this level of privilege into the same container as the Samba file server would expand the attack surface of the privileged process across the file serving stack unnecessarily. The NFS container is privileged; the Samba containers are not.

The three containers share no process namespace and no user namespace. Volume mounts into a shared host directory (`/export`) are the only intentional data channel between them, and those mounts give each container a view of the same physical storage without any coordination mechanism beyond the host filesystem's own consistency guarantees.

5.3 Network Layout and Addressing

Two Podman bridge networks are defined. `podman_network_b` covers the `10.90.0.0/24` subnet and is used exclusively by `samba-ad-dc`. `podman_network_a` covers `10.89.0.0/24` and is shared by `samba-fs` and `nfs`. The DC container is deliberately not on the same virtual network as the other two.

This split has a practical motivation. During domain provisioning, `samba-tool domain provision` creates a DNS A record that maps the DC's NetBIOS name to whatever IP the container has at that moment. If the container received a dynamic IP from the Podman bridge, this record would point to an address on the `10.90.0.0/24` internal network, one that is not routable from the lab subnet and that clients would never be able to reach. By assigning a static IP to the container (`10.90.0.146`, configured with the `ip` parameter in the Ansible `podman_container` task), Ansible knows exactly which DNS record to delete after provisioning completes. The cleanup is deterministic: delete the A record pointing

to 10.90.0.146, add the correct record pointing to 130.192.27.251. Without the static assignment, the internal IP could vary across container restarts, making the cleanup fragile.

The IP alias on the host is configured in /etc/network/interfaces via the Jinja2 template rendered by Ansible:

```
up   ip addr add 130.192.27.251/24 dev <iface>
down ip addr del 130.192.27.251/24 dev <iface>
```

This secondary address exists only on the physical interface; there is no separate NIC or VLAN. Podman's port-forwarding rules map traffic arriving at 130.192.27.251 on the AD-related ports 53 (DNS), 88 (Kerberos), 135 (RPC endpoint mapper), 137–139 (NetBIOS), 389 (LDAP), 445 (SMB/AD), 464 (Kerberos password change), 3268 (Global Catalog LDAP), and the dynamic RPC range 55000–55100/tcp into the container through podman_network_b. From any client on the lab network, 130.192.27.251 behaves as the domain controller's address. The forwarding indirection is invisible.

File traffic goes to the other address. The file server's SMB ports (139 and 445/tcp) and the NFS daemon's 2049 (tcp and udp) are published on 130.192.27.1, which is what Windows clients hit for shares and Linux clients hit for mounts. LDAPS on 636 and the TLS Global Catalog on 3269 sit in the playbook commented out: directory traffic is encrypted with STARTTLS on 389 rather than a dedicated TLS port, matching how SSSD is set up on the workstations.

Figure 5.1 illustrates the complete addressing scheme and the port forwarding relationships between the host's external interfaces and the container networks.

5.4 Authentication Flows

Walking through a login sequence from a client's perspective makes the role of each component concrete, since the container and network boundaries are invisible to the client but affect every step of the process.

When a Windows workstation attempts a domain login, it first performs a DNS SRV lookup to locate the domain controller for LABINFAD.POLITO.IT. That query reaches Samba's internal DNS server running inside the samba-ad-dc container, arriving at port 53 on 130.192.27.251. The DNS response points the client to the same address. The client then sends a Kerberos AS-REQ to port 88 on that address, which Podman forwards to the KDC inside the container. Assuming valid credentials, the KDC returns a Ticket Granting Ticket. The mechanics of the Kerberos exchange itself are not specific to this architecture and are described in chapter 2; what matters here is that the entire exchange stays on the 130.192.27.251 address

and that the client has no visibility into the fact that it is talking to a containerised process rather than a physical server.

To access an SMB share, the Windows client requests a service ticket for the Service Principal Name `cifs/cclix1.labinfad.polito.it` from the KDC. This SPN was registered in the AD database by Ansible during the setup of the file server container, using `samba-tool spn add`. The KDC issues the ticket, the client presents it when connecting to port 445 on `130.192.27.1`, and the `samba-fs` container validates it. Inside `samba-fs`, Winbind translates the user's AD SID into a POSIX UID/GID pair using the RFC 2307 attributes stored in the AD object, and file system operations proceed with the resulting Unix identity. The share definitions, `[homes]`, `[admin]`, `[corsi]`, are defined in the `smb.conf` deployed by Ansible, and the share paths map into the `/export` directory tree mounted from the host.

The Linux side is shorter but leans on the DC just as hard. A workstation running SSSD queries the DC's LDAP on port 389 over STARTTLS at PAM login, pulls the user's POSIX attributes and groups, and caches them for a while. The NFS mount goes straight to 2049 on 130.192.27.1, which Podman routes to `nfs`. That container authenticates nothing. It runs `sec=sys`, so access is decided by the UID and GID the request carries and by nothing else. The whole thing holds together only because SSSD resolves the user to the very UID that owns their home directory on disk, and it does because both SSSD and the account-creation scripts read that number from the same `uidNumber` on the same AD object.

Figure 5.2 shows both paths side by side.

5.5 Identity Schema in the New Design

The biggest shift from the old setup is where POSIX identity lives and who reads it.

Before, those numbers sat in OpenLDAP. `ou=People` and `ou=Groups` held the POSIX attributes, smbldap-tools bolted on the Samba-specific ones, and `ou=Idmap` kept a separate table mapping Windows SIDs to Unix IDs. Three subtrees, plus Samba's own TDB state, all of which had to stay in agreement with nothing enforcing it.

Now there is one database: the Active Directory instance Samba runs. Passing `-use-rfc2307` to `samba-tool domain provision` extends the AD schema with the RFC 2307 attributes [19], so `uidNumber`, `gidNumber`, `unixHomeDirectory` and `loginShell` become first-class fields on every user object, sitting next to the ordinary Windows attributes. No second tree, no sync script. On the DC, `idmap_ldb:use rfc2307 = yes` in `smb.conf` switches the extension on at the Samba level.

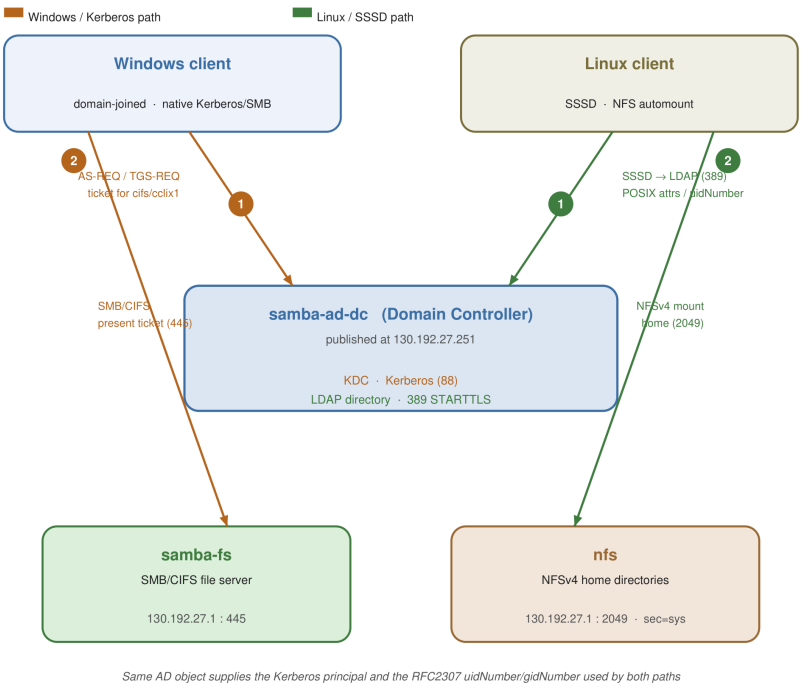

Figure 5.2: Authentication flows for Windows and Linux clients. Windows clients use Kerberos to obtain service tickets from the DC and then connect to the Samba file server via SMB/CIFS. Linux clients use SSSD to resolve identity from the DC's LDAP interface and mount home directories directly via NFS.

Two GIDs are fixed at provisioning time: Domain Users gets 100513, Domain Admins gets 100512. Every account keeps Domain Users as its POSIX primary group, 100513 is the `gidNumber` written into the object at creation and the GID that owns the home directory tree. On the AD side that is the only primary group an account has, and it does not change.

Then there is `domusers`, GID 101513, added during the initial deployment. It is a service group, not a stand-in for Domain Users, and the difference matters. The Linux clients only grant interactive login to members of `domusers`, so an account that resolves perfectly but is missing from that group gets turned away at the prompt. There is also a subtler reason it has to exist. When an AD user's primary group is itself a domain group, the membership is implicit in the user object rather than written out as an explicit member entry, and some NSS and PAM paths on Linux do not surface it the way they surface a normal supplementary group. The symptom is petty but real: tools that enumerate group membership sometimes fail to list a student inside Domain Users, even while authentication and file ownership work fine. A real, explicitly populated `domusers` covers both problems at once.

The mechanism, documented in `all.yml`, is to add every `Domain Users` member to `domusers` as an ordinary member:

```
for user in $(samba-tool group listmembers "Domain Users"); do
    samba-tool group addmembers domusers "$user"
done
```

and the account-creation tooling repeats it for each new account. So a student carries Domain Users (100513) as the real primary group and `domusers` (101513) as an explicit supplementary group with a known GID. How this is handled at creation time is covered in Chapter 7. The arrangement is not pretty, but it answers a genuine corner of mixed-environment identity mapping that the current tooling has no cleaner way to solve.

User UIDs start at 100000 and are handed out one at a time at creation. The last value used is kept in a plain text file,

```
/srv/samba-ad-dc/var-lib-samba/private/last_uid.txt
```

on a host volume that outlives container restarts. The number goes into the object's `uidNumber` when the account is made and never moves afterward. Because the file shares the `var-lib-samba` volume with Samba's LDB databases, it survives any recreation that leaves the volume in place.

Winbind in `samba-fs` is configured with the `idmap_ad` backend for the `LABINFAD` domain:

```
idmap config LABINFAD : backend = ad
idmap config LABINFAD : schema_mode = rfc2307
idmap config LABINFAD : unix_nss_info = yes
```

This backend reads `uidNumber` and `gidNumber` directly from the RFC 2307 attributes in the AD object, performing no algorithmic derivation from the RID and maintaining no local mapping table. The UID that Winbind presents to the filesystem is exactly the value stored in AD. For SIDs that `idmap_ad` does not recognise built-in groups, objects from other domains, a TDB backend is configured as the wildcard fallback with a separate range:

```
idmap config * : backend = tdb
idmap config * : range   = 3000-7999
```

This is a significant simplification over the old model, where the `ou=Idmap` LDAP subtree maintained an explicit SID-to-UID table that had to be kept in sync with `ou=People` and was consulted separately on every Windows-to-Unix identity translation.

5.6 Storage and Data Persistence

Container filesystems are ephemeral by design: when a container is removed and recreated, anything written inside it that is not on a bind-mounted volume is lost. The application data that must survive this lifecycle like Samba's LDB databases, Kerberos keytabs, TLS private keys, log files, per-share configuration is stored on the host and mounted into the containers through Podman bind volumes.

The persistence layer on the host is the `/srv/` directory tree. For `samba-ad-dc`, three subdirectories are created and mounted:

- `/srv/samba-ad-dc/var-lib-samba` → `/var/lib/samba` inside the container: Samba's LDB databases (including `sam.ldb`, the main AD store), the Kerberos keytab, the SysVol directory tree, TLS certificates, and the `last_uid.txt` counter.
- `/srv/samba-ad-dc/etc-samba` → `/etc/samba`: the `smb.conf` and any supplementary configuration.
- `/srv/samba-ad-dc/var-log-samba` → `/var/log/samba`: per-client log files, written by Samba at the configured log level.

The same pattern applies to `samba-fs`, with additional bind mounts for `/etc/nsswitch.conf` (which inside the file server container specifies `files winbind` for password and group resolution) and `/etc/resolv.conf` (which points to `130.192.27.251` as the primary nameserver, then to the subnet gateway, then to `8.8.8.8`). These two files must be managed by Ansible rather than baked into the container image, because the addresses they contain are infrastructure variables that may change.

`samba-fs` and `nfs` both mount `/export` from the host at the same path. This is the tree with the home directories and the course content, `/export/home/stud`, `/export/home/corsi`, `/export/home/admin`, and the template directory used when an account is created. Because both containers see the same host path, a file written over SMB through `samba-fs` shows up immediately over an NFS mount served by `nfs`. There is no second copy and nothing to replicate. The host filesystem is the one source of truth, with two protocol layers reading and writing on top of it.

Splitting `var-lib-samba` from `etc-samba` under `/srv` is deliberate. Ansible regenerates configuration from Jinja2 templates and drops it into `etc-samba`. It never touches `var-lib-samba` after the first provision, the LDB databases Samba builds and maintains are off limits to it. That line is what makes it safe to push a config change and restart a container without putting the AD databases at risk: Ansible rewrites configuration, never internal storage.

The NFS container needs one more thing handled on the host. The host's own `nfs-server` and `rpcbind` are stopped and disabled by Ansible before the container starts. `nfs-kernel-server` is installed anyway, purely because it brings in the `nfs` and `nfsd` kernel modules; the host daemon itself is never allowed to run. Ansible loads the modules with modprobe and makes them stick through `/etc/modules-load.d/nfs.conf`. The containerised server then uses them from inside a privileged container with `SYS_ADMIN`, which is what it takes to reach the kernel's NFS layer.

5.7 Automation and Reproducibility

As discussed in Chapter 3, the infrastructure-as-code principle holds that the system configuration should be expressed in a form that is executable, version-controlled, and repeatable. In this architecture, Ansible is not a convenience layer over manual steps; it is the only mechanism through which the infrastructure is built or modified.

All variables like IP addresses, domain names, container names, Podman network subnets, LDAP base DNs, UID base, GID assignments, path conventions, password policy parameters are centralised in `all.yml`. Every Jinja2 template rendered by Ansible draws from this file. Changing the domain name, or the DC's IP alias, or the UID base, means editing one variable and re-running the playbook; the change propagates to `smb.conf`, `krb5.conf`, `resolv.conf`, `/etc/hosts`, `sssd.conf`, the NFS exports, the `/etc/network/interfaces` stanza for the IP alias, and anything else that references the variable, all consistently, with no manual search-and-replace.

Sensitive values are stored in an Ansible Vault file and never appear in plaintext in the repository. The Vault holds the domain Administrator password, the LDAP query account credential, and the default initial password for new user accounts.

The Ansible controller decrypts the Vault at runtime using a passphrase held by the lab staff.

The first-run provisioning of the AD domain is handled through a sequence of conditional tasks. Ansible checks whether `/var/lib/samba/private/sam.ldb` exists inside the mapped host volume. If it does not, which is the case on a fresh deployment, the playbook starts the DC container with the entrypoint `["tail", "-f", "/dev/null"]` to keep it alive without launching Samba, then runs `samba-tool domain provision` inside it via `podman exec`:

```
samba-tool domain provision          \
    --realm=LABINFAD.POLITO.IT       \
    --domain=LABINFAD                \
    --server-role=dc                 \
    --use-rfc2307                    \
    --dns-backend=SAMBA_INTERNAL
```

Once provisioning completes, the container is recreated with its normal entrypoint (`["samba", "-F"]`), which runs Samba in foreground mode. Every later run skips the whole block because `sam.ldb` is now there.

DNS record management demonstrates a slightly more complex idempotency pattern. Provisioning creates a DNS A record pointing the DC's name to the internal Podman IP `10.90.0.146`, which is not reachable from the lab network. Ansible deletes this record using `samba-tool dns delete` and adds correct records pointing to `130.192.27.251`. Both operations are run with `failed_when: false`, so re-running the playbook on an already-provisioned system does not fail if the record to be deleted is already gone or the record to be added already exists.

User management for day-to-day lab operations is handled by a separate set of Python scripts rather than directly by Ansible. These scripts were rewritten to use `samba-tool user create` as their primary operation, which writes all required attributes, including the RFC 2307 POSIX fields `uidNumber`, `gidNumber`, `unixHomeDirectory`, and `loginShell`, to the AD database in a single call. This eliminates the two-step write sequence of the old system and the inconsistency window it created between OpenLDAP and Samba's TDB. The Python layer above `samba-tool` handles the lab-specific logic: sequential UID allocation from `last_uid.txt`, group membership assignment, email notification, and the PDF regulation document that students must acknowledge when their account is created.

The division of responsibility between Ansible and the Python scripts is intentional. Ansible owns infrastructure state: network configuration, container lifecycle, domain parameters, certificate generation, and the initial set of administrative accounts. The Python scripts own operational activity: creating and managing student accounts on a daily basis. Keeping these concerns separate prevents the playbooks from accumulating lab-specific user management logic that would make

them harder to understand, and it prevents the management scripts from needing to know anything about container orchestration.

Chapter 6

Design Rationale

The previous chapter described the architecture as it ended up: three Podman containers on `cclix1`, the identity schema built on RFC 2307 attributes, the two authentication paths for Windows and Linux clients. What it did not do was explain why any of those decisions were made instead of the obvious alternatives. That is the purpose of this chapter.

None of these choices were made in a vacuum or against a clean slate. The lab already had a working domain, a fixed user population, a known set of client machines, and exactly three people who would administer the result. Several of the decisions that look like best-practice recommendations in a textbook were, in this case, forced by constraints that had nothing to do with theory. I have tried to keep that distinction visible throughout: where a choice was genuinely a trade-off, I say so; where it was effectively dictated by the environment, I say that too.

6.1 Identity Backend: Samba AD DC versus FreeIPA and Plain OpenLDAP

The single most consequential decision in the whole project was which directory technology would hold the identities. Everything else, container layout, idmap configuration, the shape of the authentication flows, follows from it. Three serious candidates were on the table: Samba acting as an Active Directory Domain Controller, FreeIPA, and a hand-assembled stack built on plain OpenLDAP. Their internals were already discussed in Section 2.3; here the question is narrower, which one fits *this* lab.

The deciding factor is the client population. The LABinf workstations that students actually sit at run Windows 10. They expect to join a domain, receive Group Policy, authenticate with Kerberos, and reach SMB shares the way any Windows machine does in any Active Directory environment. That expectation

is not a preference we could negotiate away; it is the behaviour the operating system implements natively. A directory backend that speaks Active Directory protocols end to end means those clients work with no middleware, no agent, no schema translation layer. Samba AD DC does exactly that, because it *is* an AD implementation rather than something pretending to be one.

FreeIPA was the more interesting alternative, and it deserved a real look. On the Linux side it is arguably nicer to operate than Samba: integrated certificate authority, host-based access control, a coherent management UI, sane defaults. The problem is the Windows side. FreeIPA is fundamentally a Linux-centric identity system; to serve Windows clients properly it either relies on its own embedded Samba component or, in the more common production pattern, establishes a cross-realm trust with an actual Active Directory forest [58]. Both options reintroduce, from a different direction, the very thing we were trying to avoid: an AD-speaking component glued onto a non-AD core. If we were going to run Samba's AD machinery anyway to keep the Windows clients happy, running it as the primary domain controller was the more honest architecture than burying it inside FreeIPA as a satellite.

Plain OpenLDAP was the weakest fit, and the reasoning is short. OpenLDAP is a directory server and nothing more. It stores objects; it does not authenticate Kerberos tickets, it does not understand Group Policy, it does not register the SRV records a Windows client uses to find a domain controller. To approximate what AD gives out of the box, we would have had to stand up a separate MIT Kerberos KDC, wire it to the LDAP backend, build the GPO story ourselves (which in practice means there is no GPO story), and accept that Windows clients would never feel like domain members. That is a large amount of custom integration to end up with something less capable than what Samba provides as a single coherent product.

There is also a constraint that does not appear in any feature comparison: the lab is run by three people. Operational simplicity is not a luxury in that situation, it is the difference between an infrastructure that stays maintained and one that quietly rots because nobody remembers how the pieces fit together. FreeIPA's deployment footprint is heavier and its administration model is different enough from classic AD that the existing institutional knowledge, which is all AD-shaped, would not transfer cleanly. Samba AD DC let us reuse what the administrators already understood about domains, Kerberos realms, and SMB.

I want to be explicit that this was not an academic benchmark. We did not score the three options on a matrix and pick the winner. The Windows client requirement collapsed the decision early: any solution that did not present a native AD face to those machines was disqualified before the finer comparisons mattered. Samba AD DC was less a clever choice than the only one that respected the constraints already in place.

6.2 Containerization: Podman on Bare Metal versus the Alternatives

Once the directory technology was fixed, the next question was how to host it. The legacy setup ran every service on the same host, sharing one kernel and one process tree, which was one of the weaknesses catalogued in Chapter 4. Isolation was the goal. The candidates were virtual machines, Docker, LXC system containers, and Podman. The mechanics of containers, namespaces, and the Podman/Docker distinction were covered in Chapter 3, so I will stay on the level of why one option beat the others for `cclix1` specifically.

Start with the hardware, because it frames everything. `cclix1` is a physical Sun Fire server with finite, non-elastic resources. There is no cloud underneath it to absorb overhead. That single fact made full virtualization unattractive. VMs would have delivered the strongest isolation, genuinely separate kernels, but at a cost that hardware of this class feels: memory reserved per guest, a hypervisor layer to maintain, and a separate patching lifecycle for each virtual machine on top of the host. For three services on one constrained box, that is a lot of overhead bought to solve a problem that container isolation already solves well enough for this threat model.

Docker was ruled out for a reason that is easy to state and was a deliberate security decision rather than a matter of taste. Docker's traditional architecture centres on a long-running daemon that runs as root. That daemon is a persistent, privileged process and, by extension, a persistent attack surface and a single point of failure for every container it manages. Podman has no such daemon. Containers run as direct child processes and integrate with `systemd` the way any other service unit would, which means the host's own init system supervises them, restarts them, and logs them through the same machinery as everything else [59]. On a server whose whole point was to reduce standing privileged surface, choosing the daemonless model over a root daemon was straightforward.

It is worth being honest about one thing the rootless story does *not* apply here. Podman's headline feature is often rootless operation, but this deployment does not use it. The AD DC container needs `CAP_SYS_ADMIN` and publishes privileged ports, which puts it outside the comfortable rootless envelope. So the argument for Podman in this project is not "rootless"; it is "daemonless and `systemd`-native." Those are different properties, and conflating them would misrepresent why the choice was made.

LXC sits closer to a lightweight VM than to an application container, and it would have worked. But it pulls toward managing each container as a small, full system, with its own init and its own package set to keep current, which is more administrative weight than we wanted for what are essentially three single-purpose

service containers. Podman's OCI-image workflow let us treat each container as a disposable, rebuildable artifact described by a `Containerfile`, which fits an Ansible-driven, declarative deployment far better.

The last point concerns why there are three containers and not one. Bundling the domain controller, the file server, and the NFS server into a single image would have been simpler to write and would have thrown away most of the isolation we were trying to gain. The split follows real service boundaries. The domain controller is the source of truth for identity and runs the most security-sensitive code. The file server is a Samba member that joins the domain like any other client and serves SMB. The NFS server speaks a different protocol entirely and, as discussed later in Section 6.5, needs privileges the other two should never hold. Each of the three has a distinct lifecycle, a distinct privilege requirement, and a distinct failure mode. Keeping them separate means a compromise or a crash in one does not automatically take the others with it, and it means each can be rebuilt or restarted without disturbing the rest.

There is one more reason for keeping the domain controller apart from the file server, and it is not an inference from container theory but an explicit recommendation from the people who write Samba. The official documentation advises against using an AD DC as a file server at all, and states plainly that the DC should be dedicated to authentication and authorization while file and print services belong on member servers joined to the domain [60, 61].

6.3 idmap_ad versus idmap_rid: RFC 2307 with Explicit Attributes

This is the most technical decision in the design, and it is the one where getting the reasoning wrong would have produced a system that appeared to work and then corrupted file ownership in ways that are painful to diagnose. The two mechanisms, `idmap_rid` and `idmap_ad`, were both described in Section 2.7; the choice between them is what matters here.

The fundamental difference is where the POSIX numbers come from. `idmap_rid` computes a UID and GID algorithmically from the RID portion of each object's SID, offset into a configured range. It is attractive because it requires nothing extra on the directory objects, no POSIX attributes at all, the numbers are derived on the fly. `idmap_ad` does the opposite: it reads `uidNumber` and `gidNumber` directly off the AD object, exactly as RFC 2307 specifies. That difference looks small until you consider what each implies over the life of the directory.

With `idmap_rid`, a user's UID is a function of their RID. RIDs are stable in normal operation, but they are an internal artifact of the directory, not a property the administrator controls or guarantees. The real danger surfaces across rebuilds

and migrations. This project is precisely such a migration: a cutover to a new domain with a new base DN, as established in Chapter 4. Algorithmically derived UIDs are only as stable as the RID allocation underneath them, and in a fresh provision there is no reason the same human ends up with the same RID. A UID that silently changes is not a cosmetic problem when that UID is stamped into the ownership of every file in a user's home directory served over NFS.

`idmap_ad` removes the derivation entirely. The UID is whatever value we wrote into `uidNumber` when the account was created, and it stays that value regardless of RID, regardless of which OU the object lives in, regardless of a future reprovision as long as the attribute is preserved. The number becomes an explicit, administrator-owned fact rather than an emergent side effect. For an infrastructure whose entire premise was a clean migration onto a new directory, with home directories full of files owned by specific UIDs, that stability was not optional.

The cost is real and worth stating plainly. Choosing `idmap_ad` means every account *must* carry complete POSIX attributes from the moment it exists. There is no automatic fallback: an object without a `uidNumber` simply does not resolve to a Unix identity. This is the reason accounts are created the way they are, with all attributes written in a single `samba-tool user create` invocation rather than created first and decorated with POSIX attributes afterward. The atomic write is not a stylistic preference; it is what `idmap_ad` demands, because a half-created account, valid in AD but lacking `uidNumber`, is a Linux account that does not work. The legacy system's non-atomic provisioning was one of its documented weaknesses, and `idmap_ad` turns atomic creation from a nicety into a correctness requirement.

The payoff extends past Samba itself, and this is the part that ties the whole identity story together. The file server's Winbind reads `uidNumber` from the AD object to map an SMB session to a Unix identity. On the Linux side, SSSD reads `uidNumber` from the *same* AD object to resolve the same user. Because both consult one authoritative source, an SMB login through Winbind and a console or SSH login through SSSD land on identical UID and GID values. That coherence is what makes NFS with `sec=sys` correct in this design. NFSv4 with `sec=sys` authorizes purely on numeric UID and GID; it trusts the client's notion of who the user is. Normally that is a fragile assumption. Here it holds because there is exactly one place the numbers come from, and every component reads it. The figure below sketches that single-source relationship.

If we had chosen `idmap_rid`, Winbind would have invented UIDs from RIDs while SSSD, configured for real POSIX attributes, read different numbers, or we would have had to force both onto the same algorithm and pray the ranges never collided with anything. The explicit-attribute approach makes the consistency structural instead of coincidental.

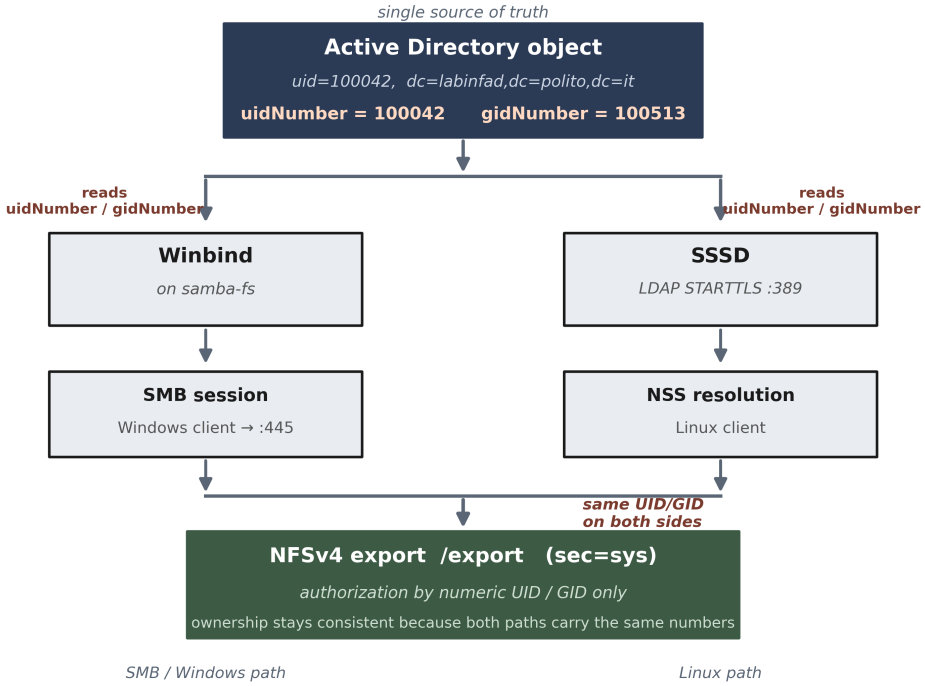

Figure 6.1: Both Winbind (SMB path) and SSSD (Linux path) resolve identity by reading the same `uidNumber`/`gidNumber` attributes from the same AD object. The shared source is what makes NFS `sec=sys` ownership consistent across protocols.

6.4 STARTTLS on Port 389 versus LDAPS on Port 636

Linux clients reach the directory over LDAP to read POSIX attributes, and that traffic has to be encrypted, carrying as it does the attributes that decide identity. There are two standard ways to get TLS on an LDAP connection. LDAPS opens a TLS session immediately on port 636, encrypted from the first byte. STARTTLS begins as a plaintext connection on the ordinary LDAP port 389 and then upgrades to TLS through a negotiated command. We use STARTTLS on 389.

The honest version of this decision is that it was settled at the level of which ports the deployment exposes. In the Ansible-published port set for the DC, 636 and the TLS global catalog port 3269 are commented out; 389 is open. STARTTLS was not selected after weighing cryptographic subtleties against LDAPS, it is what the exposed surface supports, and the surface was kept deliberately narrow. Encryption still happens; it just happens as an upgrade on the standard port rather than on a second dedicated one.

That said, STARTTLS carries a well-known weakness that has to be addressed rather than waved away. Because the connection starts in plaintext, a client that does not insist on completing the TLS upgrade can be tricked, by a network attacker who strips or blocks the upgrade, into continuing in clear, a downgrade attack [62]. STARTTLS is only as safe as the client's enforcement of it. The protection therefore lives on the client. SSSD on every Linux machine is configured with `ldap_id_use_start_tls = true` and, critically, `ldap_tls_reqcert = hard`. The first makes the upgrade mandatory; the second makes the client validate the server certificate against a trusted CA and refuse the connection if validation fails. With `hard` in force, a stripped or substituted TLS handshake does not silently fall back to plaintext, it fails closed.

That guarantee depends on the clients actually trusting the right certificate authority. The PKI here is self-managed: an internal CA produced by the `genera_certificati.sh` script signs the directory's certificate. For `reqcert = hard` to mean anything, every client has to hold that CA certificate in its trust store, which is handled as part of the deployment by fetching the CA cert from the DC and installing it into the system trust store on each client. The chain is closed on both ends: the server presents a certificate from our CA, and the client has been told, in advance and out of band, to trust exactly that CA and nothing weaker. Within a controlled environment where we operate both the directory and the client images, a self-managed CA with strict client validation is a defensible posture, and arguably a cleaner one than exposing an additional LDAPS port for no functional gain.

6.5 NFSv4-Only: Disabling NFSv3

The file store is shared to clients over NFS, and the deployment runs NFSv4 exclusively. NFSv3 is switched off, explicitly, through `NFS_DISABLE_VERSION_3=1` in the container's environment, with `rpcbind` disabled on the host as well. The question here is why version 3 was deliberately removed rather than left available for compatibility.

The cleanest argument is about surface area. NFSv3 is not self-contained. It leans on `rpcbind` listening on port 111 to advertise where its various sub-services live, plus a separate `mountd`, plus the ancillary lock and status daemons. Each of those is another listening process, another port, another thing to keep patched and firewalled. NFSv4 folds all of that into a single well-known port, 2049, and dispenses with the portmapper entirely. Fewer moving parts is fewer ways in. For a service whose job is simply to serve home directories on a LAN, the multi-daemon RPC apparatus of v3 was pure liability with no benefit we needed.

NFSv4 is also the better protocol on its own terms for this use. It is stateful,

which gives it stronger and more coherent file-locking semantics than v3's stateless model, and that matters for home directories where the same user may have a session open from more than one machine. Consolidating onto 2049 also made the container's published-port story trivial, which is visible in how the NFS container is configured.

The one point that invites a raised eyebrow is the use of `sec=sys`. That security flavour authenticates with nothing more than the numeric UID and GID the client claims, in the clear, with no cryptographic proof of identity. In a hostile network that would be indefensible. It is acceptable here for a specific structural reason: the security boundary for this infrastructure is the network, not the NFS layer. The export is reachable only within the controlled academic LAN, the client machines are ones we provision and manage, and access is constrained at the export and network level rather than trusted to NFS authentication. The UID/GID values that `sec=sys` relies on are, as argued in Section 6.3, guaranteed consistent because every component reads them from the same AD object. So the model is coherent: NFS does not try to be the place where identity is proven, that job belongs to Kerberos and to the directory; NFS only enforces ownership, and the ownership numbers are trustworthy because of how identity is managed upstream. Layering Kerberized NFS (`sec=krb5`) on top would have added real operational complexity, principal management, keytab distribution, clock-skew sensitivity, to defend a boundary that the network perimeter already defends. For this environment that trade was not worth making, and the decision to skip it was made with the perimeter assumption stated openly rather than assumed silently.

Chapter 7

Domain Controller and Container Provisioning

The previous chapters described what the new infrastructure looks like and explained the reasons behind the technologies that were chosen. This chapter is about the part that was actually difficult to get right: turning that design into a running domain controller, idempotently, from an Ansible playbook that can be re-executed any number of times without breaking the live database. Most of the engineering effort in the project did not go into deciding *that* Samba would run inside a Podman container, but into the order in which things had to happen so that the container could provision itself once and then keep running as a normal service afterwards.

Throughout the chapter I refer to the actual tasks of the `samba-ad-dc` role and to the configuration files that feed them. The variables are the ones centralised in `all.yml`; here they appear in the form in which the playbook consumes them.

7.1 The bootstrap sequence

A Samba Active Directory domain controller needs a provisioned database before it can start. The provisioning step, `samba-tool domain provision`, is what creates `sam.ldb` and the rest of the `private/` directory together with the internal DNS zone, the Kerberos KDC material and the default directory objects. The problem is a chicken-and-egg one. The container image is built with `samba -F` as its command, so that on every normal restart the daemon comes up in the foreground and logs to stdout. But on the very first deploy there is no `sam.ldb` yet, and `samba -F` against an unprovisioned tree exits almost immediately. With `restart_policy: always` that turns into a restart loop, and the container never stays alive long enough to be provisioned through `podman exec`.

The way out is to decouple the two states. On the first run the container is

started with a command that does nothing but keep the main process alive, the provisioning is performed against the now-stable container, and only then is the container recreated with its real entrypoint. The whole thing is gated on the presence of `sam.ldb`, which is the single fact that tells the playbook whether the domain already exists.

```
- name: Controlla se Samba e' gia' stato provisionato
  ansible.builtin.stat:
    path: /srv/{{ samba_ad_dc_data_dir }}/var-lib-samba/private/sam.ldb
  register: samba_provisioned
```

Listing 7.1: Idempotency guard on the provisioned database

Because `/srv/samba-ad-dc/var-lib-samba` is bind-mounted into the container at `/var/lib/samba`, the `stat` runs on the host and inspects exactly the file that the container would populate. There is no need to exec into the container to ask whether it is provisioned: the database lives on the host filesystem, and the host can see it directly. Every subsequent task that belongs to the first-deploy path carries the same condition, `when: samba_provisioned.stat.exists == false`, so on a second run the three bootstrap tasks are simply skipped.

The first of those tasks creates the temporary container.

```
- name: Crea container Samba AD DC
  containers.podman.podman_container:
    name: "{{ samba_ad_dc_container_name }}"
    image: "{{ samba_ad_dc_image_name }}"
    state: started
    hostname: "{{ netbios_name_ad_dc }}"
    recreate: yes
    detach: true
    restart_policy: always
    init: true
    cap_add:
      - SYS_ADMIN
    command: ["tail", "-f", "/dev/null"]
    network: "{{ podman_network_name_b }}"
    ip: "{{ podman_ip_ad_dc }}"
    volume:
      - "/srv/{{ samba_ad_dc_data_dir }}/var-lib-samba:/var/lib/samba"
      - "/srv/{{ samba_ad_dc_data_dir }}/etc-samba:/etc/samba"
      - "/srv/{{ samba_ad_dc_data_dir }}/var-log-samba:/var/log/samba"
    publish:
```

```
21        - "{{ IP_DC }}:53:53/tcp"
22        # ... full port list, see Section 7.1.1
23    when: samba_provisioned.stat.exists == false
```

Listing 7.2: Holding container for the first deploy

The command `["tail", "-f", "/dev/null"]` is the conventional trick to keep a container running with no real workload. It blocks forever on an empty file, PID 1 stays alive, and the container is therefore a stable target for `podman exec`. Here `recreate: yes` is appropriate: if a previous half-finished attempt left a broken container behind, we genuinely want it discarded and rebuilt from the image, because at this point there is no state worth preserving, `sam.ldb` does not exist, so there is nothing to lose.

`SYS_ADMIN` is added because Samba performs operations during provisioning and at runtime that a default container is not permitted to do, and `init: true` gives the container a proper init process so that the eventual `samba` supervisor and its child processes (discussed in Chapter 2) are reaped correctly rather than accumulating as zombies.

With the container alive, provisioning is a single `podman exec`.

```
1  - name: Provision Samba AD DC (solo se non esiste)
2    ansible.builtin.command: >
3      podman exec {{ samba_ad_dc_container_name }} samba-tool
       domain provision
4        --realm={{ realm }}
5        --domain={{ workgroup }}
6        --adminpass='{{ dc_admin_password }}'
7        --server-role=dc
8        --use-rfc2307
9        --dns-backend=SAMBA_INTERNAL
10   when: samba_provisioned.stat.exists == false
```

Listing 7.3: Domain provisioning

The flags map directly onto the identity schema fixed in Chapter 5: `-realm` resolves to `LABINFAD.POLITO.IT` and `-domain` to the NetBIOS workgroup `LABINFAD`. `-use-rfc2307` is the important one. It tells the provisioning to extend the schema with the POSIX attributes (`uidNumber`, `gidNumber`, `unixHomeDirectory`, `loginShell`) that the whole Linux side of the deployment depends on; without it the directory would be a pure Windows AD and the `idmap_ad` backend on `samba-fs` would have nothing to read. `-dns-backend=SAMBA_INTERNAL` keeps DNS inside the same process rather than delegating to BIND, which is the right call for a single-DC site and avoids one more moving part.

The admin password is interpolated from `dc_admin_password`, which is a Vault-protected variable. It never appears in the playbook in clear and it is not echoed

in the task output.

Once the database exists, the container is recreated with its definitive command.

```
- name: Crea container Samba AD DC
  containers.podman.podman_container:
    name: "{{ samba_ad_dc_container_name }}"
    image: "{{ samba_ad_dc_image_name }}"
    state: started
    hostname: "{{ netbios_name_ad_dc }}"
    recreate: false
    detach: true
    restart_policy: always
    init: true
    cap_add:
      - SYS_ADMIN
    command: ["samba", "-F"]
    network: "{{ podman_network_name_b }}"
    ip: "{{ podman_ip_ad_dc }}"
    volume:
      - "/srv/{{ samba_ad_dc_data_dir }}/var-lib-samba:/var/lib/samba"
      - "/srv/{{ samba_ad_dc_data_dir }}/etc-samba:/etc/samba"
      - "/srv/{{ samba_ad_dc_data_dir }}/var-log-samba:/var/log/samba"
```

Listing 7.4: Final container with the real entrypoint

This task has no `when` guard, so it runs on every execution of the playbook, and the difference from the first one is `recreate: false`. That single line carries most of the idempotency logic of the role. On the first deploy the container is currently running `tail`, so the declared command differs from the actual one and the Podman module recreates it with `samba -F`. On every later run the container is *already* running `samba -F` with the same image, the same network and the same volumes, so the module finds nothing to change and leaves it untouched. If `recreate: yes` had been used here instead, every single playbook run would tear the container down and rebuild it. The database survives because it lives on the bind-mounted host volume, but the restart is still gratuitous: it drops every active SMB and LDAP session, it briefly takes the KDC offline, and on a busy lab that is a self-inflicted outage for no benefit. `recreate: false` is therefore not a detail, it is the thing that makes the role safe to re-run during the working day.

`-F` is chosen over the interactive `-i` for a practical reason: in foreground mode Samba writes its logs to stdout, which Podman then captures, so `podman logs samba-ad-dc` gives the full daemon output without having to mount and tail `/var/log/samba` by hand. During the early debugging phase this mattered a lot.

7.1.1 Why the published-port list lives in the container definition

The `publish` block (abbreviated in Listing 7.2) maps the AD service ports from the host alias `130.192.27.251` into the container: 53 for DNS, 88 and 464 for Kerberos and its password-change service, 135 plus the dynamic range `55000-55100` for RPC, 137–139 for NetBIOS, 389 for LDAP, 445 for SMB, and 3268 for the global catalog. The list is duplicated verbatim in both container tasks. That duplication is ugly and I am aware of it, but it is deliberate: the two tasks are not always both executed, and factoring the port list into a YAML anchor would have made the first-deploy and steady-state paths share state in a way that is harder to reason about than two explicit, self-contained task bodies. The full rationale for which ports are open, and for the two that are commented out, is the subject of Section 7.7.

Note that `636` (LDAPS) and `3269` (GC over TLS) are present but commented in the published list. The deployment uses STARTTLS on the ordinary LDAP port rather than a separate TLS port, so those two never need to be reachable from outside the container.

7.2 Post-Provisioning Steps

Once the domain has been provisioned the container is restarted under `samba -F`, but a few things still have to fall into place before the playbook can talk to the directory it just built. The first is Kerberos configuration. The provisioning step writes a working `krb5.conf` into `/var/lib/samba/private/`, and rather than symlink it, which the Samba documentation advises against, the playbook copies it into `/etc/krb5.conf` inside the container so that the standard Kerberos client path finds it.

```
1 - name: Copia krb5.conf generato da Samba
2   ansible.builtin.command: >
3     podman exec {{ samba_ad_dc_container_name }}
4       cp /var/lib/samba/private/krb5.conf /etc/krb5.conf
```

Listing 7.5: Copying the generated krb5.conf

The second is timing. Right after a restart the daemon is up as a process but not yet answering on every port, and the tasks that follow, the DNS edits and the SPN registration, will fail if they race ahead of it. Two `wait_for` tasks gate on this: one blocks until LDAP is listening on 389, the other until the KDC answers on 88. Each has a thirty-second ceiling, which on this host is comfortably more than the daemon needs but still short enough that a genuinely dead container surfaces as a failure rather than a hang.

```
- name: Attendi che LDAP sia in ascolto sul container AD DC
  ansible.builtin.wait_for:
    host: "{{ IP_DC }}"
    port: 389
    timeout: 30
    state: started

- name: Attendi che KDC Kerberos sia pronto
  ansible.builtin.wait_for:
    host: "{{ IP_DC }}"
    port: 88
    timeout: 30
    state: started
```

Listing 7.6: Waiting for the DC to be ready

7.3 DNS after provisioning

Provisioning leaves the internal DNS in a state that is correct from Samba's point of view and wrong from the network's. `samba-tool domain provision` registers the host records for the new domain controller using the address the container actually has on its own interface, which is the Podman network address `10.90.0.146`. That address is private to `podman_network_b` and means nothing to a client on the lab subnet. A Windows machine that performs the usual SRV lookup, gets back `10.90.0.146`, and then tries to reach the KDC there will simply time out.

This is the reason the container is pinned to a static Podman address with `ip: 10.90.0.146` in both container tasks. The choice is not about wanting a predictable internal IP for its own sake, it is that we need to know, ahead of time and exactly, which bogus record provisioning is going to create, so that we can delete it again programmatically. A dynamically assigned address would have forced the playbook to first discover what the DC had registered and then remove it, which is fragile. Pinning the address turns a discovery problem into a constant.

Two tasks clean this up. The first deletes the records pointing at the Podman address.

```
- name: Rimuovi voce DNS default
  ansible.builtin.expect:
    command: "podman exec -it {{ samba_ad_dc_container_name
   }}
       samba-tool dns delete 127.0.0.1 {{ domain_name }}
       {{ item.name }} A {{ item.ip }} -U {{ dc_admin_user }}
   "
```

```
6       responses:
7         Password: "{{ dc_admin_password }}"
8     loop:
9       - { name: "{{ netbios_name_ad_dc }}", ip: "{{
        podman_ip_ad_dc }}" }
10      - { name: "@", ip: "{{ podman_ip_ad_dc }}" }
11    failed_when: false
12    changed_when: false
```

Listing 7.7: Removing the Podman-internal DNS records

The second adds the records that clients should actually see: the file server `cclix1` at `130.192.27.1`, the DC alias `cclix1dc` at `130.192.27.251`, and the zone apex `@` also at the DC alias.

```
1   - name: Aggiungi voci al DNS interno di samba
2     ansible.builtin.expect:
3       command: "podman exec -it {{ samba_ad_dc_container_name
        }}
4           samba-tool dns add 127.0.0.1 {{ domain_name }}
5           {{ item.name }} A {{ item.ip }} -U {{ dc_admin_user }}
        "
6       responses:
7         Password: "{{ dc_admin_password }}"
8     loop:
9       - { name: "{{ server_name }}", ip: "{{ IP_SERVER }}" }
10      - { name: "{{ dc_name }}",     ip: "{{ IP_DC }}" }
11      - { name: "@",                 ip: "{{ IP_DC }}" }
12    failed_when: false
```

Listing 7.8: Adding the externally reachable records

`samba-tool dns` expects an interactive password, which is why these tasks go through `ansible.builtin.expect` rather than `command`: the module sends the Vault-held admin password in response to the `Password:` prompt. Both tasks set `failed_when: false`. On the first run the delete removes a record that exists and the add creates one that does not, both succeed. On every later run the situation is reversed, the bogus record is already gone so the delete fails with "record not found", and the correct record already exists so the add fails with "already exists". Neither of those is a real error in an idempotent context; the desired end state is reached regardless, and forcing the playbook to treat them as failures would only produce noise. The delete additionally carries `changed_when: false` because, conceptually, after the first run it never changes anything.

The query against `127.0.0.1` inside the `exec` is deliberate: the command talks to the DNS service of the very DC it is running on, through the loopback inside

the container, rather than going back out through the published port. This keeps the operation independent of whether the external alias is reachable at the exact moment the task runs, which during a fresh provisioning it may not yet be.

7.4 Password policy: domain default and a fine-grained override

Out of the box, a freshly provisioned Samba AD enforces the familiar Windows defaults: complexity on, a minimum length of seven, and a maximum age. Those defaults do not fit LABinf. The user population is students who log in occasionally and who, in practice, write their password on a sticky note the moment it becomes hard to remember. Forcing Windows-style complexity on that population produces weaker security in the room, not stronger, because it pushes the secret onto paper. The decision was a minimum length of ten characters with complexity disabled, trading character-class rules for a slightly longer string that is still memorable.

Two layers implement this. The first sets the domain-wide default.

```
- name: Imposta permessi Password Utenti default
  ansible.builtin.command:
    podman exec {{ samba_ad_dc_container_name }}
    samba-tool domain passwordsettings set {{ item }}
  loop:
    - --min-pwd-length={{ min_pwd_length }}
    - --min-pwd-age=0
    - --max-pwd-age=365
    - --complexity={{ complexity }}
```

Listing 7.9: Domain-wide password settings

With `min_pwd_length` set to 10 and `complexity` to `off` in `all.yml`, this already produces the behaviour we want for the domain as a whole. The second layer is a Password Settings Object (PSO), Samba's implementation of fine-grained password policies. A PSO lets a different policy be attached to a specific group rather than to the whole domain, and it carries a precedence value that resolves conflicts when a user is subject to more than one. Here the PSO is attached to `Domain Users` so that the student-facing policy is bound to the group every student belongs to, independently of the domain default, which keeps the intent explicit and survives any later change to the domain-wide settings.

```
- name: Controlla se la PSO esiste
  ansible.builtin.command: >
    podman exec {{ samba_ad_dc_container_name }}
    samba-tool domain passwordsettings pso show
   pso_domain_users
```

```
5     register: pso_check
6     failed_when: false
7
8   - name: Crea PSO solo se non esiste
9     ansible.builtin.command: >
10      podman exec {{ samba_ad_dc_container_name }}
11      samba-tool domain passwordsettings pso create
      pso_domain_users 5
12      --min-pwd-length={{ min_pwd_length }}
13      --min-pwd-age=0
14      --max-pwd-age=365
15      --complexity={{ complexity }}
16    when: pso_check.rc != 0
17
18  - name: Associa PSO al gruppo Domain Users
19    ansible.builtin.command: >
20      podman exec {{ samba_ad_dc_container_name }}
21      samba-tool domain passwordsettings pso apply
      pso_domain_users "Domain Users"
22    register: pso_apply
23    failed_when: "'already applies' not in pso_apply.stderr
      and pso_apply.rc != 0"
```

Listing 7.10: Creating and applying the PSO

The precedence value passed to `pso create` is `5`. With a single PSO in the domain its absolute value is irrelevant, but it has to be there, and keeping it low leaves room for higher-numbered policies later without reordering.

The interesting part is the idempotency of the `apply` task. `samba-tool ... pso apply` is not idempotent: applying a PSO that already applies to the group exits with a non-zero return code and the message `ERROR: PSO 'pso_domain_users' already applies to 'Domain Users'`. A naive task would fail on the second run. The condition

```
1 failed_when: "'already applies' not in pso_apply.stderr and
    pso_apply.rc != 0"
```

inverts that: the task is considered failed only when the return code is non-zero *and* the error is something other than "already applies". The already-applied case is recognised by its stderr text and swallowed. This pattern, match the tool's own error string and treat that specific one as success, recurs throughout the role, because `samba-tool` was designed to be driven by a human at a prompt, not by an idempotent automation engine, and it signals "this is already the case" the same

way it signals real failure. The creation step avoids the same trap differently, by checking with `pso show` first and only creating when the check returns non-zero.

7.5 Unix attributes on the built-in groups

RFC 2307 mapping needs a `gidNumber` on the groups, not just a `uidNumber` on the users. After provisioning, the built-in groups `Domain Users` and `Domain Admins` exist as directory objects but carry no POSIX attributes, they are Windows groups with a SID and no GID. Until they get one, `idmap_ad` on the file server cannot resolve them to a numeric GID, and a student's primary group on the Linux side would fall back to whatever the wildcard backend invented.

```
1  - name: Controllo se "Domain Users" ha gia' gidNumber
2    ansible.builtin.shell: >
3      podman exec {{ samba_ad_dc_container_name }}
4      samba-tool group show 'Domain Users' | grep gidNumber
5    register: domain_users_gid
6    changed_when: false
7    failed_when: false
8
9  - name: Aggiunta del parametro Unix al Domain Users
10   ansible.builtin.command: >
11     podman exec {{ samba_ad_dc_container_name }}
12     samba-tool group addunixattrs "Domain Users" {{
     gid_number }}
13   when: "'gidNumber' not in domain_users_gid.stdout"
```

Listing 7.11: Adding gidNumber to the built-in groups

`Domain Users` is assigned `gid_number` = 100513 and `Domain Admins` `gid_number_admins` = 100512, both fixed in `all.yml`. The check uses `group show` piped through `grep gidNumber`: if the attribute is already there the `addunixattrs` step is skipped, so the task is naturally idempotent without needing to interpret an error string. `grep` returning non-zero when it matches nothing is exactly why this guard task sets `failed_when: false`, an absent `gidNumber` is not a failure, it is the signal to proceed.

7.5.1 The `domusers` group: a primary-group workaround

There is a third group, `domusers`, with `gidNumber` 101513, and it deserves a paragraph because it looks at first like a mistake. It is not a duplicate of `Domain Users`; it is a compensation for an edge case in how the primary group resolves on the Linux clients.

When a student account is created, its POSIX primary group points at `Domain Users` (GID 100513). On Windows this is exactly right. On Linux, going through Winbind on the file server and through SSSD on the workstations, the primary-group resolution for an AD user whose primary group is itself a domain group occasionally does not surface the way ordinary supplementary groups do, the membership is implicit in the user object rather than carried as an explicit `member` entry, and some NSS and PAM paths expect to find the user listed as a member of their own primary group. The symptom is small and annoying: tools that enumerate group membership do not always see the student inside `Domain Users`, even though authentication and file ownership work. I tested this personally. The Linux clients restrict interactive login through `access.conf`, and the natural rule, allow only members of `Domain Users`, did not work.

The fix is to maintain `domusers` (101513) as a real, explicitly populated group, and to add every student to it as an ordinary member at account-creation time and to point the `access.conf` rule at `domusers` rather than `Domain Users`. The account-management script does this as part of `add_user`, calling `samba-tool group addmembers` for the `domusers` group right after the user is created. The student therefore has `Domain Users` as the directory primary group and `domusers` as an explicit supplementary group with a known GID, and the cases where the primary group alone was not enough are covered. It is a workaround, not an elegant solution, but it is the kind of thing that only shows up once real users start logging in on both operating systems.

7.6 Creating the administrative users

The day-to-day student accounts are not created by Ansible. They are created on demand by the `cclpy` Python tooling, which owns user lifecycle operations. What the playbook does create, once, is the small set of administrative accounts for the people who run the lab, because those need to exist before anyone can do anything and they belong to the infrastructure state that Ansible is responsible for.

```
1  - name: Creo utenti Domain Admins
2    ansible.builtin.command: >
3      podman exec {{ samba_ad_dc_container_name }}
4      samba-tool user create {{ item.name }} {{ item.password
       }}
5         --use-username-as-cn
6         --uid='{{ item.name }}'
7         --uid-number='{{ item.uid_number }}'
8         --gid-number='{{ item.gid_number }}'
9         --given-name='{{ item.given_name }}'
10        --surname='{{ item.surname }}'
```

```
11        --gecos='{{ item.given_name }} {{ item.surname }}'
12        --login-shell='/bin/bash'
13        --home-directory='\\{{ server_name }}\{{ item.name }}'
14        --home-drive='Z:'
15        --unix-home='{{ home_stud_directory }}/{{ item.name }}
      '
16        --must-change-at-next-login
17    loop:
18      - { name: 'andrea', given_name: 'Andrea', surname: '
      Galletto',
19          uid_number: '99001', gid_number: '{{
      gid_number_admins }}',
20          password: '{{ default_admin_password }}' }
21      - { name: 'davide', given_name: 'Davide', surname: '
      Piumatti',
22          uid_number: '99002', gid_number: '{{
      gid_number_admins }}',
23          password: '{{ default_admin_password }}' }
24      - { name: 'luca', given_name: 'Luca', surname: 'Prevato'
      ,
25          uid_number: '99003', gid_number: '{{
      gid_number_admins }}',
26          password: '{{ default_admin_password }}' }
27    register: create_admins
28    failed_when: "'already exists' not in create_admins.stderr
        and create_admins.rc != 0"
29    changed_when: "'already exists' not in create_admins.
      stderr"
```

Listing 7.12: Administrative user creation

A few of the flags are worth reading carefully because they are the same ones the student-creation script uses, and they encode the link between the Windows view and the POSIX view of an account.

`-use-username-as-cn` forces the directory Common Name to equal the username. This matters because SSSD on the clients is configured with `ldap_user_name = uid`, and keeping CN, username and `uid` aligned removes a whole class of "the account exists but the lookup returns nothing" problems that come from AD's habit of using a display-name CN by default.

`-uid-number` and `-gid-number` write the POSIX identity directly into the AD object. This is the whole point of the RFC 2307 approach defended in Chapter 6: the UID is stored, not derived. The admins get fixed UIDs in the 99000 range so they never collide with the student range, which starts at 100000 and is allocated sequentially.

`-home-directory` and `-unix-home` are two different homes for two different worlds, and conflating them is a classic error. `-home-directory` is the UNC path `\\cclix1\username` that a Windows session maps to drive `Z:`; `-unix-home` is the POSIX path `/export/home/stud/username` that a Linux session uses as `$HOME`. The same account therefore lands in the same physical place, the shared `/export` tree, by two different routes depending on the operating system it logs in from.

`-must-change-at-next-login` forces a password change on first use, so the shared `default_admin_password` from Vault is only ever a one-time bootstrap secret.

The idempotency handling mirrors the PSO case. `samba-tool user create` on an existing user fails with `already exists`; the `failed_when` treats that one error as success, and the matching `changed_when` ensures that a run which only encounters already-existing users is reported as *unchanged* rather than green-but-changed, which keeps the playbook's change reporting honest. A separate task then adds the three accounts to `Domain Admins`, using the same trick against the `member already` message.

7.6.1 The `ldapuser` service account

The administrators are not the only account the playbook has to create up front. There is one more, and it has nothing to do with a person who logs in. The Linux workstations do not talk to Active Directory the way the file server does. The file server is a domain member with a machine account and a keytab, so Winbind authenticates as the host itself. The Ubuntu clients run SSSD instead, and SSSD needs an identity to bind with before it can read anything out of the directory: a plain, unprivileged account whose only job is to answer LDAP searches. That account is `ldapuser`.

It is deliberately ordinary. No POSIX attributes, no group memberships, no shell that matters, nothing that would let it own files or start a session. The only thing it is ever used for is the bind that SSSD performs on every `getent` and every login lookup, resolving usernames, UIDs and group membership through `ldap_default_bind_dn` on the client side. Its distinguished name, `cn=ldapuser,cn=Users,dc=labinfad,dc=polito,dc=it`, is exactly the value that reappears in the client `sssd.conf` discussed in Chapter 8, Listing 8.7. Creating it here is what makes that configuration valid; without this account the bind on the client would have nowhere to land.

```
1 - name: Controllo se l'utente ldapuser esiste gia'
2   ansible.builtin.command: >
3     podman exec {{ samba_ad_dc_container_name }}
4     samba-tool user show {{ ldap_query_user }}
5   register: ldapuser_check
```

```
6     changed_when: false
7     failed_when: false
8
9   - name: Creazione dell'utente ldapuser
10    ansible.builtin.command: >
11      podman exec {{ samba_ad_dc_container_name }}
12      samba-tool user create {{ ldap_query_user }} {{
      ldap_query_password }}
13    when: ldapuser_check.rc != 0
```

Listing 7.13: Creating the LDAP bind account

The idempotency pattern is the by-now familiar one, but turned around. Where the admin task lets `samba-tool` fail on `already exists` and then forgives that specific error, here a guard task runs `user show` first and the creation is gated on its return code. If `ldapuser` is already in the directory the show succeeds with `rc` equal to zero, the `when: ldapuser_check.rc != 0` condition is false, and the create never runs. Either approach reaches the same end state; this one was used wherever a clean `show` check existed and reading an error string felt more fragile than reading an exit code.

One detail worth noting is that the password comes from `ldap_query_password`, a Vault variable, and the account never has its password rotated or expired by the PSO logic from section 7.4. A service account that SSSD binds with on every lookup cannot be subject to `-must-change-at-next-login` or a short maximum age, because there is no interactive session in which a change could ever happen. It sits outside the human password policy on purpose, and the only thing protecting it is that it can read the directory and do nothing else.

7.7 TLS material and CA distribution

The Linux side of the deployment binds to LDAP over STARTTLS, not over LDAPS on port 636. That decision is argued in Chapter 6; what concerns this chapter is producing the certificates and getting the CA onto the clients so that `ldap_tls_reqcert = hard` in `sssd.conf` can actually be satisfied.

The DC's `smb.conf` points Samba at three files under the bind-mounted private directory:

```
1     tls enabled  = yes
2     tls keyfile  = /var/lib/samba/private/tls/samba.key.pem
3     tls certfile = /var/lib/samba/private/tls/samba.cert.pem
4     tls cafile   = /var/lib/samba/private/tls/ca.cert.pem
```

Listing 7.14: TLS directives in the DC smb.conf

Those paths resolve, on the host, to `/srv/samba-ad-dc/var-lib-samba/private/tls/`, which is the directory the certificate-generation script writes into. The script, `genera_certificati.sh`, builds a small local CA and issues the server certificate from it, with a validity of `tls_days` (3650 days, ten years, which for an internal lab CA is a reasonable trade between security hygiene and not having to remember to rotate it mid-thesis).

```
- name: Eseguo lo script per il certificato
  ansible.builtin.command: >
    {{ script_altro_path }}/genera_certificati.sh
  register: script_output
  tags: [tls]

- name: Scarica il certificato CA del Samba AD DC
  ansible.builtin.fetch:
    src: "{{ samba_tls_dir }}/ca.cert.pem"
    dest: ca.cert.pem
    flat: yes
  tags: [tls]
```

Listing 7.15: Running the certificate script and fetching the CA

`ansible.builtin.fetch` pulls the CA certificate back from the managed host to the control machine, from where it is distributed into the client systems' trust store. On the clients the CA ends up in the bundle referenced by SSSD as `ldap_tls_cacert = /etc/ssl/certs/ca-certificates.crt`, and `ldap.conf` points at the same bundle with `TLS_CACERT`. The chain is therefore closed: the DC presents a certificate signed by the lab CA, the client has that CA in its system trust store, and the STARTTLS handshake on port 389 validates with `reqcert = hard` instead of being downgraded to `allow`.

Because the TLS material is only meaningful once Samba reloads it, the role finishes with an explicit stop-and-start of the DC container. This is the one place where a restart is intentional rather than accidental, the certificates were written after the daemon started, so the daemon has to be bounced to read them. Both tasks carry the `tls` tag, which means the certificate work, the fetch and the restart can be re-run as a unit with `-tags tls` without touching provisioning, DNS or user creation. That selective re-run was used repeatedly while getting the certificate chain right, and it is a good illustration of why every task in the role is either guarded by a state check or made safe to repeat: the role is not a one-shot installer, it is something you run again and again as you converge the system, and only the parts that genuinely changed should do any work.

7.8 Summary

The provisioning of the domain controller comes down to a single ordering insight and a consistent discipline around it. The ordering insight is that a Samba DC cannot start before it is provisioned and cannot be provisioned before it starts, which is resolved by bringing the container up with an inert command, provisioning through `podman exec`, and only then recreating it with `samba -F`, with `recreate: false` on that final step so the database and the live sessions survive every later run. The discipline is idempotency: a `stat` on `sam.ldb` gates the whole first-deploy path, condition checks gate the group attributes and the PSO creation, and everywhere `samba-tool` signals "already done" with a non-zero exit, a `failed_when` matches that specific message and lets it pass. The DNS rewrite, the password policy, the Unix attributes on the built-in groups, the administrative accounts, and the TLS chain are each implemented as small tasks that reach the same end state whether they run on a bare server or against a system that has been live for months. How those identities are then consumed on the file server and the NFS side, the domain join, Winbind, the export ACLs, and the kernel-module handling, is the subject of Chapter 8.

Chapter 8

File Server, Winbind and NFS Integration

The previous chapter dealt with the part of the infrastructure that owns identity: the domain controller, its database, the password policy, the service accounts. This chapter moves to the part of the infrastructure that actually serves data. Two containers are involved here, `samba-fs` and `nfs`, and a third actor that does not run in any container at all, the SSSD stack living on `cclix1` and on the Linux workstations.

What ties these three together is that they all consume the same identities, but they reach them through different paths. The file server resolves users through Winbind. The Linux clients resolve users through SSSD. The NFS server resolves nothing at all and trusts whatever UID arrives on the wire. Getting these paths to agree on the same numbers for the same people is the whole point of the work described below, and most of the configuration in this chapter exists to keep them aligned rather than to add features.

The design of all this was already laid out in Chapter 5. Here the focus is on the configuration files as they were actually written, on the directives that needed a specific value rather than a default, and on a handful of things that did not work on the first attempt.

8.1 The file server `smb.conf`

The file server runs as a member server, not a domain controller, and its configuration reflects a single concern: it must map AD identities to POSIX UIDs and GIDs in exactly the way the rest of the system expects, and it must never invent a number on its own.

8.1.1 The idmap block

The relevant part of the global section is the idmap configuration:

```
idmap config * : backend = tdb
idmap config * : range = 3000-7999

idmap config LABINFAD : backend = ad
idmap config LABINFAD : range = 99000-1099999
idmap config LABINFAD : schema_mode = rfc2307
idmap config LABINFAD : unix_nss_info = yes
```

Listing 8.1: idmap and Winbind directives in the file server smb.conf

There are two idmap backends declared here, and the distinction between them matters. The domain-specific block, `idmap config LABINFAD : backend = ad`, tells Winbind that for any SID belonging to the `LABINFAD` domain it must not compute an identity but read it. The `schema_mode = rfc2307` option points Winbind at the POSIX attributes stored on the AD object itself, `uidNumber`, `gidNumber`, `loginShell`, `unixHomeDirectory`. These are the same attributes `samba-tool user create` writes atomically at account creation, as described in Chapter 7. The reasoning behind choosing `idmap_ad` over `idmap_rid` was already argued in Chapter 6; what is shown here is the directive-level consequence of that decision.

`unix_nss_info = yes` is the directive that is easy to overlook and important to get right. Without it, Winbind would still resolve UID and GID from the AD object, but it would fall back to the `template shell` and `template homedir` values for the login shell and home directory, ignoring `loginShell` and `unixHomeDirectory` on the object. With `unix_nss_info = yes`, Winbind exposes the home directory and shell that are actually stored in AD through NSS as well. For this deployment that is not cosmetic: the home directory recorded on each account is the one the NFS automounter expects, and a mismatch there means a user logs in to a working session whose `$HOME` points nowhere.

The wildcard block, `idmap config * : backend = tdb`, handles every SID that does not belong to `LABINFAD`. These are the built-in and well-known SIDs, `BUILTIN\Administrators`, `NT AUTHORITY\SYSTEM`, and similar, which have no `uidNumber` attribute and never will. For those, a locally allocated number from a `tdb` file is perfectly acceptable, because nothing outside this single host ever needs to agree on what they are. The range `3000-7999` was kept deliberately far below the student range, which starts at 100000. There is no functional overlap between a built-in account squatting at 3001 and a real student at 100xxx, and keeping the two ranges separated by a wide gap makes it obvious when reading a `ls -ln` listing which numbers came from AD and which were allocated locally.

8.1.2 Winbind behaviour directives

```
winbind nss info = rfc2307
winbind use default domain = yes
winbind enum users = no
winbind enum groups = no
winbind refresh tickets = yes
```

Listing 8.2: Winbind NSS and enumeration directives

`winbind nss info = rfc2307` is the companion to the idmap block. It forces Winbind to answer NSS queries using the RFC 2307 attributes rather than deriving anything algorithmically. With `idmap config LABINFAD : backend = ad` already in place this is the only consistent choice; setting it to anything else would put the NSS-facing side of Winbind at odds with its own idmap backend, and the symptom of that is the kind of bug where `getent passwd mario` returns one UID and `id mario` returns another.

`winbind use default domain = yes` strips the domain prefix from usernames. Without it, every identity surfaces as `LABINFAD\mario`; with it, the same user is simply `mario`. This is mostly a convenience on the Windows side, but it has a concrete effect on the Linux side too, because PAM stacks and shell tools deal much more gracefully with a bare username than with one containing a backslash. Since this is a single-domain environment there is no ambiguity to lose by dropping the prefix.

The two enumeration directives are set to `no` on purpose. Enumeration is what lets a command like `getent passwd` with no argument return the entire population of users. With roughly ninety accounts, turning it on would not be a performance disaster, so the choice is not really about speed. It is about not generating periodic full-scan LDAP queries against the DC for something nothing in the workflow needs. Enabling enumeration encourages tools to walk the whole directory on a hunch, and on a domain controller that is also serving live authentication that is unnecessary traffic. Account creation and lookup in this deployment always go through the `cclpy` scripts and targeted `ldapsearch` filters, never through a blind enumeration, so disabling it costs nothing and removes a class of avoidable load.

8.1.3 Protocol floor

```
server min protocol = SMB2_02
client min protocol = SMB2_02
```

Listing 8.3: SMB protocol minimum

This single line is the most direct answer to one of the weaknesses identified in the analysis of the legacy system (Chapter 4). The old domain had been forced

down to SMB1 (`NT1`) to keep an obstinate Windows 10 LTSB image working after a Samba upgrade, and that downgrade dragged the whole infrastructure back to a protocol that should have been dead years earlier. On the new file server, `SMB2_02` is the floor and there is no negotiation path back to SMB1. Nothing in the current client population needs it, and the few machines that did were precisely the ones the migration was meant to retire. Setting both `server min protocol` and `client min protocol` closes the door from both directions, so the file server will neither accept an SMB1 session nor open one when it acts as a client during the domain join.

8.2 Joining the file server to the domain

The `smb.conf` above describes a member server, but writing `server role = member server` does not make it one. The container becomes an actual member only when it joins the domain, and that join is the point at which `samba-fs` stops being a standalone Samba install and starts being something Active Directory knows about. The join is what produces the machine account `CCLIX1$` in the directory and the Kerberos keytab that `smbd` later uses to validate tickets. Everything in §8.1 is inert configuration until this step has run.

Like the provisioning of the DC, the join is guarded so it does not run on every execution. `net ads testjoin` answers the only question that matters: is this machine's secret still valid against the DC?

```
- name: Verifica se il server e' gia' joinato al dominio
  ansible.builtin.command: >
    podman exec -it {{ samba_fs_container_name }} net ads
   testjoin
  register: join_check
  changed_when: false
  failed_when: false

- name: join del server cclix1 al dominio
  ansible.builtin.expect:
    command: "podman exec -it {{ samba_fs_container_name }}
      net ads join -U {{ dc_admin_user }} -S {{
   netbios_name_ad_dc }}"
    responses:
      Password: "{{ dc_admin_password }}"
  when: "'Join is OK' not in join_check.stdout"
```

Listing 8.4: Conditional domain join

If `testjoin` prints `Join is OK` the join task is skipped; otherwise the join runs, authenticating as the domain administrator and targeting the DC by its NetBIOS

name. The join itself goes through `ansible.builtin.expect` rather than `command` because `net ads join` prompts for a password interactively, and there is no flag that feeds it non-interactively in the way the rest of the playbook would prefer.

Joining is necessary but not sufficient for Windows clients to mount the shares over Kerberos. When a Windows machine connects to `\\cclix1`, it asks the KDC for a service ticket for the SMB service on that host, identified by the Service Principal Name `cifs/cclix1.labinfad.polito.it`. If no object in the directory claims that SPN, the KDC cannot issue the ticket. The client then silently falls back to NTLM or fails outright, depending on how it is configured, and either way the failure is the kind that looks like a permissions problem and is actually a naming one. The machine account created by the join already registers an SPN under the container's NetBIOS name, but the clients resolve and connect to `cclix1.labinfad.polito.it`, so the SPN for that exact name has to be present too.

```
- name: Aggiungi SPN cifs via expect
  ansible.builtin.expect:
    command: >
      podman exec -it {{ samba_fs_container_name }}
      samba-tool spn add cifs/{{ netbios_name_fs }}.{{
   domain_name }}
      {{ netbios_name_fs }}$
      -H ldap://{{ dc_full_name }} -U {{ dc_admin_user }}
    responses:
      Password: "{{ dc_admin_password }}"
  failed_when: false
```

Listing 8.5: Registering the CIFS SPN

This writes `cifs/cclix1.labinfad.polito.it` onto the `CCLIX1$` machine account, through an explicit `ldap://` connection to the DC. The trailing `$` is the machine-account naming convention and is easy to drop, at which point the command quietly targets a non-existent user account instead of the machine. `failed_when: false` covers the re-run case, where the SPN is already registered and `samba-tool` objects.

There is one ordering detail that does not show up in the listings but matters in practice. Both the keytab from the join and the freshly written SPN are read by `smbd` at startup, so after the join and the SPN registration the container is stopped and started again. A running `smbd` would not pick up the new machine credentials on its own, and skipping the restart produces a server that authenticated fine during the join and then refuses tickets for reasons that are genuinely hard to read from the logs. The restart is cheap and it removes the ambiguity, so it is done unconditionally at the end of this part of the role.

8.3 NSS inside the `samba-fs` container

The `nsswitch.conf` mounted into the `samba-fs` container is short and deliberately different from the one on the host:

```
passwd:         files winbind
group:          files winbind
shadow:         files
gshadow:        files

hosts:          files dns
```

Listing 8.6: nsswitch.conf inside samba-fs

On `cclix1` the corresponding lines read `files sss ldap`. Inside the container they read `files winbind`. This is not an inconsistency to be smoothed over; it is the point.

Inside `samba-fs`, the only thing that needs to resolve AD identities is Samba itself, and Samba resolves them through Winbind. There is no SSSD daemon running in this container, and there is no reason to put one there. When a Windows client connects and authenticates over Kerberos, `smbd` asks Winbind to turn the authenticated SID into a UID and GID, and Winbind answers from the AD object using the RFC 2307 attributes. The NSS `winbind` module is the bridge that lets the rest of the container's libc see those same identities, which matters when Samba needs to `chown` a file to a domain user or evaluate `valid users` against a group.

The host is a different story. `cclix1` and the Linux workstations are end-user login systems, and there Linux logins go through SSSD against LDAP, as described in the next section. So the infrastructure ends up running two identity stacks side by side: Winbind inside `samba-fs` serving Samba, and SSSD on the hosts serving interactive Linux logins. They are parallel paths. They read from the same AD objects and they agree on the same numbers, they have to, because RFC 2307 gives both of them the same `uidNumber`, but they exist for different consumers and neither one is a substitute for the other. Trying to collapse them into a single stack was considered and rejected: Samba member servers are happiest with Winbind, and Linux desktops are happiest with SSSD, and forcing either to use the other's mechanism buys nothing but friction.

8.4 The Linux authentication stack: `sssd.conf` on the host

SSSD on `cclix1` and on the workstations is configured as a plain LDAP client, not as an AD client. This is a deliberate choice and worth stating clearly, because

SSSD has a dedicated `id_provider = ad` mode that would have been the obvious default.

```
[domain/labinfad.polito.it]
id_provider = ldap
auth_provider = ldap
access_provider = ldap

ldap_referrals = false

ldap_uri = ldap://cclix1dc.labinfad.polito.it
ldap_search_base = dc=labinfad,dc=polito,dc=it
ldap_default_bind_dn = cn=ldapuser,cn=Users,dc=labinfad,dc=
   polito,dc=it
ldap_default_authtok = <vault>

ldap_id_use_start_tls = true
ldap_tls_reqcert = hard
ldap_tls_cacert = /etc/ssl/certs/ca-certificates.crt
```

Listing 8.7: Core provider and TLS configuration in sssd.conf

Choosing `id_provider = ldap` over `id_provider = ad` keeps SSSD reading the directory as a directory. The `ad` provider would pull in its own Winbind-style machinery and its own assumptions about ID mapping, and the goal here was the opposite: treat AD as an LDAP server that happens to hold POSIX attributes, and read those attributes directly. The whole infrastructure is built around the idea that the `uidNumber` written into AD is authoritative, so the client stack that consumes those numbers should read them verbatim and do nothing clever.

That intent is made explicit by `ldap_id_mapping = false`:

```
ldap_id_mapping = false
ldap_schema = ad

ldap_user_name = uid
ldap_user_uid_number = uidNumber
ldap_user_gid_number = gidNumber
ldap_user_home_directory = unixHomeDirectory
ldap_user_shell = loginShell
```

Listing 8.8: POSIX attribute mapping in sssd.conf

With `ldap_id_mapping = false`, SSSD does not derive UIDs from SIDs; it reads `uidNumber` and `gidNumber` off the object. This is the SSSD-side equivalent of `schema_mode = rfc2307` on the Winbind side, and the two settings are what make the two stacks land on identical numbers for the same user. The explicit

attribute map underneath is there because `ldap_schema = ad` changes some default attribute names, and a couple of them needed pinning. `ldap_user_name = uid` is the one that mattered most in practice: SSSD's AD schema would otherwise key the username off `sAMAccountName`, and in this directory the field that holds the login name as the rest of the system understands it is `uid`.

Two directives in this file are the result of debugging rather than design. `ldap_referrals = false` is set because with referrals left on, SSSD simply failed to find users in this setup, the search would chase a referral and come back empty. The configuration file carries a one-line comment to that effect next to the directive, left in place precisely so that nobody flips it back on six months from now. It is the kind of setting that looks wrong until you have watched it fail with the other value.

The TLS directives close out the security posture for the Linux path. `ldap_id_use_start_tls = true` forces STARTTLS on port 389; combined with `ldap_tls_reqcert = hard`, there is no plaintext fallback and no tolerance for an unverified certificate. The decision to use STARTTLS on 389 rather than LDAPS on 636 was made at the architecture level (Chapter 6); the consequence here is that SSSD must validate the DC certificate against the CA bundle, and the CA certificate generated during provisioning is distributed into the system trust store on every client so that `hard` can actually succeed.

Finally, `enumerate = false` and `cache_credentials = false` are both intentional and both restrictive. No credential is cached locally, which is a security policy decision: a stolen or cloned workstation disk should not carry usable offline logins for domain accounts. The cost is that a client with no path to the DC cannot authenticate a domain user at all, which in a lab where the DC and the workstations share the same rack is an acceptable trade. Enumeration is off for the same reason it is off on the file server, nothing in the workflow needs to list every user, and leaving it on only invites tools to scan the whole directory.

8.5 NFS: kernel modules, the container, and exports

The NFS server is the one component that does no identity resolution whatsoever. NFSv4 with `sec=sys` authenticates nothing beyond the UID and GID numbers carried in each request, and it trusts those numbers implicitly. The entire correctness of the NFS layer therefore rests on something external to it: that every client presenting UID 100123 has resolved that number from the same AD object, through SSSD, and that the files on disk are owned by exactly that number. The file server and the SSSD stack described above are what make that assumption hold. NFS just moves bytes for whoever claims a UID.

8.5.1 Kernel modules and the host

The container image is `erichough/nfs-server`, and it runs a user-space NFS server. That server still depends on kernel facilities, it cannot serve NFS without the `nfs` and `nfsd` modules loaded in the host kernel, because the modules are what actually implement the protocol in the kernel's network stack. Containers share the host kernel, so the modules have to be present on `cclix1` itself before the container starts.

The Ansible `pre_task` handles this in a way that looks contradictory until the reasoning is clear:

```
- name: install kernel modules
  ansible.builtin.apt:
    name: nfs-kernel-server
    state: present

- name: disable NFS server
  ansible.builtin.systemd:
    name: nfs-server
    state: stopped
    enabled: false

- name: disable RPCbind
  ansible.builtin.systemd:
    name: rpcbind.service
    state: stopped
    enabled: false

- name: disable RPCbind.socket
  ansible.builtin.systemd:
    name: rpcbind.socket
    state: stopped
    enabled: false
```

Listing 8.9: NFS host preparation in pre_task.yml

The `nfs-kernel-server` package is installed not to run it, but to obtain the kernel modules and supporting bits that ship with it. Immediately afterward the host's own `nfs-server` daemon and `rpcbind` are stopped and disabled, along with `rpcbind.socket`. The intent is to keep the kernel-side capability and discard the user-space server on the host, because the user-space server is going to run inside the container instead. If both the host daemon and the container tried to bind the NFS ports the result would be a conflict on 2049, and disabling the host services up front avoids that race entirely. Leaving `rpcbind` running would also be pointless, NFSv4 does not use the portmapper, which is the same reason port 111 is left

unpublished on the container.

The modules are then loaded explicitly and made persistent:

```
- name: Carica i moduli del kernel per il container NFS
  community.general.modprobe:
    name: "{{ item }}"
    state: present
  loop:
    - nfs
    - nfsd

- name: Rendi persistenti i moduli del kernel
  ansible.builtin.lineinfile:
    path: /etc/modules-load.d/nfs.conf
    line: "{{ item }}"
    create: yes
  loop:
    - nfs
    - nfsd
```

Listing 8.10: Loading and persisting kernel modules

The `modprobe` task loads them for the current boot; the `lineinfile` task writes them into `/etc/modules-load.d/nfs.conf` so they come back after a reboot. Both are needed. Without the `lineinfile` step the container would start cleanly the first time and then fail silently after the next host reboot, which is exactly the sort of failure that does not show up during a deployment and surfaces weeks later when someone restarts the machine for an unrelated reason.

8.5.2 Why the container is privileged

```
- name: Create and run NFS container
  containers.podman.podman_container:
    name: "{{ nfs_container_name }}"
    image: "{{ nfs_image_name }}"
    privileged: true
    cap_add:
      - SYS_ADMIN
    env:
      NFS_VERSION: "4"
      NFS_DISABLE_VERSION_3: 1
```

Listing 8.11: Privileged NFS container

This container runs with `privileged: true` and an explicit `SYS_ADMIN` capability. That is a wide grant, and it goes against the general preference in this

project for keeping containers as constrained as possible, so it deserves a justification rather than a shrug. The user-space NFS server inside the container has to perform mount operations, it needs to mount the NFSv4 pseudo-filesystem and manipulate the export table, and those are privileged kernel operations. There is no reduced-capability configuration that lets a container do this. Attempts to run it with a curated capability set rather than full `privileged` ran into the server failing at startup when it tried to set up its mounts. For a container whose only job is to export a single tree to a trusted lab subnet, accepting the privilege was the pragmatic outcome; the isolation that matters here is provided by the network boundary and the export ACLs, not by capability dropping.

The two environment variables pin the protocol. `NFS_VERSION: "4"` and `NFS_DISABLE_VERSION_3: 1` make this an NFSv4-only server, which is what lets the whole RPC/portmapper apparatus stay disabled and keeps the published port list down to 2049 alone.

8.5.3 `/export` as a single shared tree

The most important structural fact about the file storage is that `samba-fs` and `nfs` both bind-mount the same host directory, `/export`. A file written by a Windows client over SMB lands in `/export` through `samba-fs`, and it is visible immediately to an NFS client reading through the `nfs` container, because there is no second copy and no replication. It is literally the same filesystem seen through two doors. A student who saves a file on a Windows session and then logs into a Linux session finds it already there.

That immediacy has a sharp edge. There is no asynchronous sync to go wrong, but there is also nothing arbitrating concurrent access between the two protocols. If an SMB client and an NFS client hold the same file open at the same time, the locking story is weak: `sec=sys` NFSv4 does not provide strong cross-protocol lock semantics, and Samba's own locking does not extend to clients that never went through Samba. In practice this is handled at the application level and by the simple reality that a given user is almost never logged in to both a Windows and a Linux session editing the same file simultaneously. It is a known limitation rather than a solved problem, and it was judged acceptable for a lab workload where the access patterns are overwhelmingly single-writer.

8.5.4 The export ACLs

The `exports.j2` template is where operational requirements that have nothing to do with identity end up shaping the configuration. The root export is straightforward:

```
1 /export 130.192.27.0/255.255.255.0(rw,fsid=0,no_subtree_check,sync
    ,crossmnt)
```

Listing 8.12: NFSv4 root export

`fsid=0` marks this as the root of the NFSv4 export tree, `crossmnt` lets clients descend into anything mounted beneath it, and the whole lab subnet gets read-write access at this level. The interesting rules are in the sub-exports, and the most revealing one is `/export/home/admin`:

```
/export/home/admin \
  130.192.27.240(ro,no_root_squash,subtree_check,sync) \
  130.192.27.241(ro,no_root_squash,subtree_check,sync) \
  ... \
  130.192.27.223(ro,no_root_squash,subtree_check,sync) \
  130.192.27.14(rw,root_squash,subtree_check,sync) \
  130.192.27.15(rw,root_squash,subtree_check,sync) \
  130.192.27.16(rw,root_squash,subtree_check,sync) \
  130.192.27.251(rw,root_squash,subtree_check,sync) \
  130.192.27.0/255.255.255.0(ro,no_root_squash,subtree_check,sync)
```

Listing 8.13: Differentiated admin export (rendered)

This single export has three different access classes, and each one exists because of a real operational need rather than a policy abstraction.

The `ro,no_root_squash` entries for the clone machines (the `.240`–`.245` range and `.223`) are the ones that look alarming at first glance. `no_root_squash` means a request arriving as UID 0 is left as UID 0 instead of being mapped down to `nobody`. Normally that is exactly what you do not want to grant. But these addresses are the imaging machines used to clone the lab workstations, and the cloning process reads system-level files out of the admin tree as root, the imaging tooling needs to preserve ownership and permissions on files it copies, and `root_squash` would mangle that by rewriting every root-owned file as `nobody`. The export is read-only for these hosts, so the privilege they are handed is the ability to read root-owned data faithfully, not to write anything. That containment, root identity but no write, is what makes the grant tolerable.

The `.14`, `.15`, `.16` and `.251` entries are the management addresses, and they get `rw,root_squash`: they can write into the admin tree, but a runaway root process on one of them is squashed to `nobody` and cannot trample root-owned files. The general subnet at the end gets `ro`, so an ordinary workstation can read shared admin material but nothing more.

The remaining exports are plainer and follow from the access model:

```
/export/home/stud  130.192.27.0/255.255.255.0(rw,root_squash,
    subtree_check,sync)
/export/home/corsi 130.192.27.0/255.255.255.0(ro,root_squash,
    subtree_check,sync)
```

Listing 8.14: Student and course exports (rendered)

Student homes are read-write for the whole subnet with `root_squash` in force, because a student session has no business presenting root and the actual file ownership is enforced by the per-home directory permissions described in Chapter 7, not by NFS. The course tree is read-only for everyone, with write access reserved to administrators through the Samba share rather than through NFS. The clone-machine carve-out in the admin export is the clearest example in the whole deployment of a configuration line that only makes sense once you know what the lab actually does on a reimaging day; without that context it reads as a misconfiguration, and with it, it reads as the only thing that works.

8.6 Startup sequence of `samba-fs`

The domain controller container starts with `samba -F` and that single command brings up the full AD stack. The file server cannot do the same, because a member server is not one process; it is `winbindd` and `smbd` running together, and they have to come up in the right order. The container therefore uses a small `start.sh` rather than a bare command:

```
#!/bin/bash
set -e

echo "[INFO] Starting Winbind..."
/usr/sbin/winbindd -D

echo "[INFO] Starting smbd..."
/usr/sbin/smbd -D --configfile=/etc/samba/smb.conf

tail -f /dev/null
```

Listing 8.15: start.sh for the samba-fs container

The ordering is not arbitrary. `winbindd` has to be running before `smbd`, because `smbd` leans on Winbind to turn authenticated SIDs into the UIDs and GIDs it needs while a session is being set up. Start `smbd` first and the first authentication that arrives before Winbind is ready resolves to nothing, and the client gets an access failure that looks like a permissions problem but is really a startup race. Both daemons are launched with `-D` so they background themselves, and then `tail -f /dev/null` keeps PID 1 alive so the container does not exit the moment the two daemons have detached. It is a crude foreground keeper, but it is exactly the right amount of machinery for a container that has no init beyond what Podman's `-init` provides, and it makes the daemon logs land in the container's standard streams where `podman logs` can reach them.

With the file server and NFS layer in place, every path that an identity can take through the system is now wired: a Windows client reaches its files over SMB

through Winbind, a Linux client reaches the same files over NFS after resolving its identity through SSSD, and both arrive at the same bytes in `/export` owned by the same UID. Whether all of this actually behaves as intended under real use is the subject of the validation work in Chapter 9.

Chapter 9

Validation and Testing

There is no test harness behind this chapter. No CI pipeline runs after every commit, no automated suite asserts a list of green checkmarks, and nothing about the validation that follows was machine-driven. For a laboratory deployment that one person provisions by hand and that serves a known, finite set of workstations, building a continuous-integration apparatus around it would have been effort spent in the wrong place. What the system needed instead was proof, gathered once against the real deployment on `cclix1`, that each chain it is supposed to guarantee actually closes end to end.

So the method here is functional and deliberately low-tech. For every promise the architecture makes I reproduced the behaviour of a real client using the native command of whatever protocol was involved, and I read the output. A Windows machine is supposed to get a Kerberos ticket and mount a share, so I joined one and looked at its ticket cache. A Linux machine is supposed to resolve an identity through SSSD and hit NFS with the same UID, so I ran `id` and then created a file across the mount and checked who owned it. The criterion throughout was binary and uninterpreted: a command either produces the output that proves the point or it does not. Where a command dumps far more than the relevant line, I have trimmed to what matters and said so.

A note on what this chapter is not. It does not re-argue any design decision, that work belongs to Chapter 6, and it does not re-describe how anything was configured, which is the subject of Chapters 7 and 8. Here the configuration is treated as given and the only question asked is whether it produces the behaviour it was meant to. Where a test brushes against a genuine limitation, I point at it and move on; the discussion of what was awkward to operate is held back for Chapter 10.

9.1 Health of the domain controller

Nothing downstream means anything if the DC itself is not what it claims to be. The first thing worth confirming is that the provisioning step from 7.1 actually produced a domain controller and not, through some misconfiguration, a member server pointed at nothing. `samba-tool domain info` answers this directly:

```
# samba-tool domain info 127.0.0.1
Forest           : labinfad.polito.it
Domain           : labinfad.polito.it
Netbios domain   : LABINFAD
DC name          : cclix1dc.labinfad.polito.it
DC netbios name  : CCLIX1DC
Server site      : Default-First-Site-Name
Client site      : Default-First-Site-Name
```

Listing 9.1: Domain controller self-report

The realm, the NetBIOS domain, and the DC's own name are the values written by the provision command in 7.3, read back out of the running directory. The role line is the one that matters: this is an active directory domain controller answering on the loopback inside its own container, which is exactly what the bootstrap was supposed to leave behind.

Because the entire deployment rests on a single `sam.ldb` living on a bind-mounted volume, the integrity of that database is not a detail. A corrupt directory would fail in ways that are hard to diagnose from the outside, so I ran the offline consistency check before trusting anything else:

```
# samba-tool dbcheck
Checking 263 objects
Checked 263 objects (0 errors)
```

Listing 9.2: Database consistency check

Zero errors across every object in the directory. This is cheap to run and it is the closest thing the deployment has to a structural assertion that the provisioning wrote a coherent database rather than a damaged one.

The DNS check is more interesting, because it is the one that directly validates the post-provisioning rewrite from 7.3. The whole reason that section exists is that `samba-tool domain provision` registers the container's internal Podman address, `10.90.0.146`, against the host records, and a client outside the container must never be handed that address; it has to receive the published `130.192.27.251` for the DC and `130.192.27.1` for the file server. Querying the zone after the rewrite shows whether the cleanup in 7.7 and 7.8 actually took:

```
# samba-tool dns query 127.0.0.1 labinfad.polito.it @ ALL
  Name=, Records=4, Children=0
```

```
3     SOA: serial=..., ... (flags=600000f0...)
4     NS: cclix1dc.labinfad.polito.it. (flags=600000f0...)
5     A: 130.192.27.251 (flags=f0...)
6   Name=cclix1, Records=1, Children=0
7     A: 130.192.27.1 (flags=f0...)
8   Name=cclix1dc, Records=1, Children=0
9     A: 130.192.27.251 (flags=f0...)
```

Listing 9.3: DNS records after the rewrite

The zone apex and `cclix1dc` resolve to `130.192.27.251`, `cclix1` resolves to `130.192.27.1`, and nowhere does `10.90.0.146` appear. That absence is the actual result being checked. Had the rewrite been skipped or failed silently, a domain-joined client performing an SRV lookup would have come back with the internal address and then tried to reach a Kerberos KDC on an IP that is not routable from the lab, and the failure would have looked like a Kerberos problem rather than a DNS one. Confirming the records here closes that off at the source.

9.2 Kerberos and the password policy

With the directory healthy, the next question is whether the internal KDC actually issues tickets. The plainest way to ask is to authenticate as the domain administrator and inspect the resulting cache:

```
1 # kinit Administrator@LABINFAD.POLITO.IT
2 Password for Administrator@LABINFAD.POLITO.IT:
3 # klist
4 Ticket cache: FILE:/tmp/krb5cc_0
5 Default principal: Administrator@LABINFAD.POLITO.IT
6
7 Valid starting       Expires              Service principal
8 06/10/2025 14:22:09  06/11/2025 00:22:09  krbtgt/LABINFAD.POLITO.
    IT@LABINFAD.POLITO.IT
```

Listing 9.4: Obtaining a TGT from the internal KDC

A ticket-granting ticket for the right principal in the right realm, with a sane lifetime, is proof that the KDC running inside the DC container is live and minting credentials. This is the foundation every Windows login later depends on, so it is worth establishing on its own before any client is involved.

The fine-grained password policy is the other piece of the authentication setup that needs verifying, and the distinction worth drawing is between confirming that the policy task ran and confirming that the policy is in force. The first is uninteresting; Ansible reports success on the task whether or not the result is meaningful. The second is what the deployment actually relies on. The PSO

created in 7.10 should exist with precedence 5 and the relaxed settings the lab chose, and it should be applied to `Domain Users`:

```
# samba-tool domain passwordsettings pso show pso_domain_users
Password information for PSO "pso_domain_users"

  Precedence (lowest is best)            : 5
  Minimum password length                : 10
  Password complexity                    : off
  Minimum password age (days)            : 0
  Maximum password age (days)            : 365

PSO applies to the following objects:
  CN=Domain Users,CN=Users,DC=labinfad,DC=polito,DC=it
```

Listing 9.5: Effective PSO on Domain Users

This reads the policy out of the directory and shows it landing on the group it was meant for, which validates the behaviour independently of how the task achieved idempotency. The numbers match the design: a ten-character floor, complexity switched off, and a one-year maximum age.

A policy that is merely declared is not the same as a policy that is enforced, and the stronger test is to try to violate it. I attempted to set a password below the ten-character floor for an ordinary student account in `Domain Users`:

```
# samba-tool user setpassword s123456 --newpassword='short1'
ERROR: Failed to set password for user 's123456':
Password doesn't meet the requirements ... (length)
```

Listing 9.6: A short password is rejected

The directory refuses it. That refusal is the evidence that matters: the PSO is not a value sitting unused in an object, it is being consulted and applied at the moment a password is set. A test that succeeds in being rejected is more convincing here than any number of successful reads, because it exercises the enforcement path rather than the declaration.

9.3 Authentication and access from a Windows client

This is the chain the actual users see. A student sits at a Windows 10 workstation, logs in with a domain account, and expects a home directory to be there. Validating it starts one layer down, with the file server's own trust relationship to the DC. The member server's machine secret has to be valid against the domain, and the same command used as a guard in 8.4 reports on it:

```
# net ads testjoin
Join is OK
```

Listing 9.7: Verifying the member-server trust

`Join is OK` means the secret held by `samba-fs` still authenticates against the DC, which is the prerequisite for the file server to broker any SMB session for a domain user. With that confirmed, a Windows 10 machine joined to the domain and a login with AD credentials produces a working session, and the user's home is mapped from the `[homes]` share on `\\cclix1`.

The more revealing check is what the Windows client's ticket cache looks like after that login, because it validates the SPN registration from 8.2. When the client connects to the file server over SMB, it asks the KDC for a service ticket naming `cifs/cclix1.labinfad.polito.it`, and the KDC can only issue that ticket if the SPN was registered against the account, which is precisely what 8.5 does. Running `klist` on the workstation after a successful file-share access:

```
C:\> klist

Cached Tickets: (2)

#0> Client: s123456 @ LABINFAD.POLITO.IT
    Server: krbtgt/LABINFAD.POLITO.IT @ LABINFAD.POLITO.IT
#1> Client: s123456 @ LABINFAD.POLITO.IT
    Server: cifs/cclix1.labinfad.polito.it @ LABINFAD.POLITO.IT
```

Listing 9.8: Service ticket for the CIFS SPN on the Windows client

The presence of that second ticket is the direct evidence the SPN is both registered and resolvable. Without it the client would have had no Kerberos service principal to request, and the connection would either have collapsed to NTLM or failed outright; seeing a CIFS service ticket in the cache means the Kerberos path ran the whole way through, from TGT to service ticket to authenticated SMB session.

One more thing worth confirming on the wire is the protocol the session negotiated. The file server's `smb.conf` sets a minimum of SMB2 in 8.3, and that floor is meaningful only if real sessions honour it. A server-side `smbstatus` shows the dialect each connection settled on:

```
# smbstatus | grep -A1 dialect    # output trimmed
PID    Username  Group     Machine            Protocol Version
...    s123456   domusers  130.192.27.84      SMB3_11
```

Listing 9.9: Negotiated SMB dialect, server side

The session came up on SMB3, comfortably above the SMB2 floor, and there are no NT1 sessions in the listing at all. That is consistent with the minimum

protocol set on the member server, and it confirms that the old NT1 fallback the legacy domain depended on (see Chapter 4) is genuinely gone rather than merely deprecated in configuration.

9.4 Authentication and access from a Linux client

The Linux path runs in parallel, and it goes through SSSD rather than Winbind, as laid out in 8.4. The single most important property of the whole system surfaces here, so it is worth being deliberate about it. When a student created with `samba-tool user create` had a `uidNumber` and `gidNumber` written onto their AD object in Chapter 7, the requirement is that SSSD on an Ubuntu workstation reads back exactly those numbers, the same object, resolved by a different stack:

```
$ id s123456
uid=100123(s123456) gid=100513(Domain Users) groups=100513(Domain
    Users),101513(domusers)
```

Listing 9.10: Identity resolution through SSSD

The UID is the one stored on the object, not a number SSSD invented, because `ldap_id_mapping` is false and SSSD is reading the RFC 2307 attributes directly. The primary GID is `100513`, and the student also appears in `domusers` at `101513`, which is the supplementary group from the workaround in 7.5.1. That `domusers` entry showing up in the group list is the validation of that workaround: it is the explicit membership that the primary-group resolution alone did not reliably surface, and here it is present.

The full account entry confirms SSSD is pulling the POSIX attributes and not just a username:

```
$ getent passwd s123456
s123456:*:100123:100513:Surname Name:/export/home/stud/s123456:/
    bin/bash
```

Listing 9.11: Full passwd entry from SSSD

The login shell and the home directory path are the `loginShell` and `unixHomeDirectory` values from the AD object, which is only possible because the SSSD attribute mapping in 8.8 is wiring those attributes through. And the built-in group resolves to the expected GID:

```
$ getent group "Domain Users"
Domain Users:*:100513:s123456,s123457,...
```

Listing 9.12: Group resolution for Domain Users

Then there is the test I trust most in this section, because it works by failing. SSSD is configured with `ldap_tls_reqcert = hard` and mandatory STARTTLS,

the enforcement that 6.4 and 8.4 both lean on to make the plaintext-to-TLS upgrade safe. The claim is that the LDAP channel is genuinely encrypted and certificate-verified, with no quiet fallback to cleartext if verification cannot happen. The way to prove there is no fallback is to remove the thing that makes verification possible. On a client where the CA certificate had *not* been distributed to the trust store, identity resolution should not degrade to an unverified connection, it should break:

```
$ id s123456
id: 's123456': no such user

# /var/log/sssd/sssd_labinfad.polito.it.log (relevant line)
... Failed to initialize TLS: (Peer's certificate issuer is not
    recognized) ...
```

Listing 9.13: Identity resolution fails without the CA certificate

The user is simply not found, and the SSSD log says why: the TLS handshake could not validate the server certificate against any trusted issuer, so the connection was refused rather than continued in the clear. This is the test that proves the encryption is real. A positive test, where resolution works on a properly provisioned client, tells you the happy path functions; it tells you nothing about whether an attacker stripping the TLS upgrade would be silently tolerated. This negative result demonstrates the absence of that tolerance directly: with `hard` in force and no trusted CA, the client fails closed exactly as 6.4 argued it must.

9.5 End-to-end UID consistency and NFS access

This is where the argument that runs through the entire thesis either holds or does not. SMB and NFS are two different protocols served by two different containers, and the whole identity scheme is built on the premise that they resolve the same user to the same number. The NFS layer, as Chapter 8 makes plain in 8.5, authenticates nothing past the UID and GID in each request; it trusts them implicitly. Its correctness is therefore entirely borrowed from the resolution layer above it.

First, the exports are visible. The portmapper is disabled on the host and NFSv3 is not in play, so the classic `showmount` is not the right instrument; the NFSv4 pseudo-filesystem is queried by mounting the root export and listing it. The exports defined in 8.12 and the rendered student and course exports in 8.14 should be reachable under `/export`:

```
# mount -t nfs4 130.192.27.1:/ /mnt/test
# ls /mnt/test/home
admin  corsi  stud
```

Listing 9.14: NFSv4 exports visible to the client

The `home` subtree with its three branches is served as expected. With the root reachable, a direct mount of the student tree:

```
# mount -t nfs4 130.192.27.1:/home/stud /mnt/stud
# mount | grep stud
130.192.27.1:/home/stud on /mnt/stud type nfs4 (rw,...,sec=sys
    ,...)
```

Listing 9.15: Mounting the student export over NFSv4

Now the test that ties everything together. A student with `uidNumber` 100123 mounts their home, creates a file, and the question is who owns that file on the underlying tree. If the resolution layers did their job, the file is owned by exactly 100123 on both sides of every protocol:

```
# as the student, over the NFS mount:
$ touch /mnt/stud/s123456/probe.txt

# on the host, looking at the physical /export tree:
# ls -n /export/home/stud/s123456/probe.txt
-rw-r--r-- 1 100123 100513 0 Jun 10 15:04 /export/home/stud/
    s123456/probe.txt

# and the same file, reached over SMB through Winbind:
# smbcacls //cclix1/homes /probe.txt -U s123456    # owner line
    only
OWNER:LABINFAD\s123456
```

Listing 9.16: A file created over NFS is owned by the AD-assigned UID

The file is owned by UID `100123` and GID `100513` on the physical tree, and the same file accessed over SMB resolves its owner to the same student through Winbind. This is the empirical confirmation of the claim made in 8.5: because NFS with `sec=sys` trusts the UID and nothing else, the only reason the ownership is correct is that SSSD on the client and Winbind on the file server both resolved the student to `100123` from the same AD object. Had `idmap_ad` and the RFC 2307 attributes not lined up, had the two stacks derived different numbers, this is the test that would have caught it, with the file landing under some other UID or appearing as `nobody`. Its success validates `idmap_ad`, the RFC 2307 schema, and the atomic attribute write of `samba-tool user create` all at once, because all three have to be right for one number to survive the whole round trip.

The version restriction deserves its own check. NFSv3 is meant to be unavailable, a consequence of `rpcbind` being stopped and disabled on the host in 8.9 and of port 111 being left unpublished, since NFSv4 has no use for the portmapper. Probing for it and forcing a v3 mount should both come up empty:

```
# rpcinfo -p 130.192.27.1
```

```
2 rpcinfo: can't contact portmapper: RPC: Remote system error -
    Connection refused
3
4 # mount -t nfs -o vers=3 130.192.27.1:/home/stud /mnt/v3
5 mount.nfs: requested NFS version or transport protocol is not
    supported
```

Listing 9.17: NFSv3 is unavailable

There is no portmapper to answer on 111, and a forced v3 mount is rejected. That is the behaviour the host preparation was meant to produce: the only door open is NFSv4.

The differentiated access on the exports is worth a quick confirmation too, since the student and course trees are deliberately not equal. From a workstation on the lab subnet, the student tree is read-write and the course tree is read-only, matching 8.14:

```
1 # echo test > /mnt/stud/s123456/ok.txt        # succeeds
2 # mount -t nfs4 130.192.27.1:/home/corsi /mnt/corsi
3 # echo test > /mnt/corsi/probe.txt
4 bash: /mnt/corsi/probe.txt: Read-only file system
```

Listing 9.18: Read-only enforcement on the course tree

A write to a student home goes through; the identical write to the course tree is refused at the export level. Write access to the course material is reserved for administrators through the Samba share rather than through NFS, which is the split 8.5 describes, and the export ACL is enforcing it.

9.6 Idempotency of the playbook

Chapter 7 makes a great deal of idempotency, and the honest way to back that up is to measure it rather than assert it. A second full run of the playbook against an already-provisioned system should change nothing:

```
1 PLAY RECAP
    *************************************************************
2 cclix1    : ok=47   changed=0   unreachable=0   failed=0   skipped
    =9
```

Listing 9.19: Second run reports no changes

`changed=0` with no failures is the result that matters, because it means the second run was a no-op and the playbook can be re-run safely without disturbing a live system. The skipped tasks are the first-deploy steps that the idempotency guards correctly stepped over.

The recap alone is not quite enough, though, because the value of that zero rests on the specific tasks where `samba-tool` is not naturally idempotent and idempotency had to be engineered with `failed_when` and `changed_when`. Those are the places where a careless re-run would either error out or report a spurious change. None of them did. The PSO task in 7.10, the administrative-user creation in 7.12, the DNS delete and add in 7.7 and 7.8, and the SPN registration in 8.5 all came back `ok` on the second pass; the directory already held what each of them would have written, and the custom result handling translated that into `ok` rather than failure or change.

The most consequential of these is the container recreation in 7.4, which carries `recreate: false`. On a second run this must report `ok` and leave the running container untouched, because recreating it would tear down the live AD stack and break every open session. The way I confirmed it left the container alone is indirect but conclusive: an SMB session and an LDAP query that were open before the run were still alive after it. If the container had been recreated, both would have dropped. They did not, which is the proof that `recreate: false` held and the second run genuinely changed nothing about the running infrastructure.

9.7 Resilience

The last thing to establish is that the system survives the two events it will actually face in a lab: a container restart and a host reboot.

Restarting the `samba-ad-dc` container should bring Samba back up under `samba -F` with the directory intact, and it should emphatically *not* re-provision. The guard in 7.1 keys on the presence of `sam.ldb`, and on restart that file is already there on the bind-mounted volume, so the entire first-deploy path is skipped:

```
# podman restart samba-ad-dc
samba-ad-dc
# samba-tool domain info 127.0.0.1 | grep -i netbios
Netbios domain   : LABINFAD
```

Listing 9.20: The DC comes back without re-provisioning

The DC answers with the same identity after the restart, and no provisioning ran, because the database it needs persisted on the volume rather than living inside the container's ephemeral layer. That persistence is precisely what the bind-mount of `var-lib-samba` was for, and the restart is what demonstrates it doing its job.

The host reboot is the more demanding test, and it is the one that validates a specific note from Chapter 8. Kernel modules do not survive a reboot on their own; if `nfs` and `nfsd` were merely loaded by hand during provisioning, the NFS container would come back up after a reboot with no kernel support underneath it and fail silently. The `lineinfile` task in 8.10 writes both modules

into `/etc/modules-load.d/nfs.conf` to prevent exactly that. After rebooting `cclix1`:

```
# after the host has rebooted, with no manual intervention:
# lsmod | grep nfsd
nfsd                    ...  ...
# podman ps --format '{{.Names}} {{.Status}}'
samba-ad-dc  Up 2 minutes
samba-fs     Up 2 minutes
nfs          Up 2 minutes

# from a client, with no remount command issued by hand:
$ ls ~/   # home is served again
```

Listing 9.21: Modules and containers recover after a host reboot

The modules are loaded because `modules-load.d` put them back, the three containers are running because their `restart_policy` is `always`, and a client can reach its home without anyone touching the server. The whole sequence ran with no manual step, which is the point: the persisted module configuration is what closes the gap that would otherwise have left NFS dead after a reboot in a way that gives no obvious error, only a mount that hangs.

9.8 Summary

Taken together these checks walk a request all the way through the system and confirm it arrives intact at every layer: the DC is authoritative and its DNS is correct, the KDC issues tickets and the password policy is enforced rather than merely declared, a Windows client gets a CIFS service ticket and a Linux client resolves the same identity over a channel that fails closed without its CA, and a file written over NFS is owned by the same AD-assigned UID it would carry over SMB. The playbook re-runs as a no-op and the deployment comes back on its own after both a container restart and a host reboot. What the validation does not address is what any of this cost to get working, where the configuration is fragile, and which corners only revealed themselves once real accounts were logging in on two operating systems at once. That is the subject of the next chapter.

Chapter 10

Operational Experience and Lessons Learned

The previous chapter showed the system doing what it was meant to do. This one is about everything that had to go wrong first. None of the checks in Chapter 9 pass on the first attempt in practice; they pass after the configuration has been wrestled into a shape that produces them, and most of that wrestling left no trace in the final playbook. What follows are the parts of the work that were genuinely awkward, the places where the design is more fragile than it looks from the outside, and a few things that only became visible once real accounts were logging in on two operating systems at the same time.

10.1 The primary group that would not behave

The single most time-consuming problem in the whole migration had nothing to do with Kerberos, TLS, or containers. It was a group ID.

The design, described in 7.5.1, expects a student's POSIX primary group to be `Domain Users` at GID `100513`. That part works exactly as written: the `addunixattrs` task in Chapter 7 stamps the `gidNumber` onto the built-in group without complaint, the account is created with `-gid-number 100513`, and `getent passwd` reports the right primary GID. Where it got awkward was not in assigning the number, it was in how that primary group resolves once a student is actually logging in.

The trouble is that `Domain Users` is itself a domain group, and an AD user's membership in their own primary group is implicit in the user object rather than carried as an explicit `member` entry. UID and file ownership are unaffected, those flow from the `uidNumber` and the `gidNumber` directly, but the tools further up the stack that enumerate group membership do not always see the student listed

inside `Domain Users`. Some NSS and PAM paths expect to find a user among the members of their own primary group and quietly do not. On its own this would be a cosmetic annoyance, except that the lab's Linux clients gate interactive access on membership of a specific group, so a membership that does not reliably surface is not cosmetic at all.

The supplementary group `domusers` at `101513` is the answer to that, and the reason it works where the primary group alone did not is precisely that it is an ordinary, explicitly populated group: every student is added to it as a real `member` at creation time, with a second `samba-tool group addmembers` call in `cclpy_functions.py` immediately after the user is created. That explicit membership is the thing the implicit primary-group membership failed to provide, so the group enumeration the clients depend on now finds the student where it expects to. The home directory is `chown`-ed to `uidNumber:gidNumber` on the physical tree for good measure, so on-disk ownership never depends on whatever the higher layers happen to report.

None of this is elegant. It is two groups where one ought to have sufficed, and the split reads as a mistake until you know that `Domain Users` carries the identity and `domusers` carries the membership that the clients can actually see. If I were starting again I would still do it this way, but I would write the reasoning down at the point of creation rather than rediscovering it, because the second time someone reads `cclpy_functions.py` the two groups look accidental. The `domusers` entry showing up in 9.10 is the visible residue of the whole thing.

10.2 UID assignment and the matricola as the stable key

The decision to assign UIDs sequentially from 100000 and track the high-water mark in `last_uid.txt`, rather than deriving them algorithmically as `idmap_rid` would, was the right one for this lab. What it implies for the migration is that identity continuity rests on the matriculation number, not on the UID itself.

The legacy domain had its own UID assignments, but those numbers were not carried across. `cclpy_add_old_accounts.py` reads a CSV of matriculation numbers and recreates each account through the ordinary `add_user` path. That path always calls `get_next_uid()`, which reads `last_uid.txt`, increments it, and hands back the next value in sequence; there is no code path that reads or reuses a legacy UID. The home directory is identified by the matricola, `/export/home/stud/s123456`, and is then `chown`-ed recursively to the freshly assigned `uidNumber:gidNumber`. The old on-disk numbers are overwritten. What stays constant across the cutover is the folder name, and that is what reconnects a returning student to their files.

This is clean in one respect that is worth stating, because it is the opposite of

a problem one might expect. Every UID in the new domain comes from a single counter, so there is no second source of numbers to collide with: no preserved range to keep clear of the sequential allocation, no shared namespace where a reused value could land on top of a generated one. The cost sits elsewhere. The scheme depends on the matricola being a genuinely stable key, since a student who returned under a different identifier would not be matched to their old directory, and it depends on the recursive `chown` reaching every file under each home. For a tree of this size that is a one-off cost, paid once during the import against the real `/export`, and it is the reason on-disk ownership ends up coherent with what `idmap_ad` reports rather than tracking whatever numbers the legacy directory happened to hold.

10.3 Where container isolation stops

Chapter 3 argued for containers on the grounds of isolation, and that argument holds for the two Samba containers. The NFS container is where the isolation story gets honest about its limits.

NFS in the kernel is not something a container can provide for itself. The `nfsd` module lives in the host kernel, is shared by every namespace on the machine, and has to be loaded on `cclix1` itself for the containerised server to have anything to talk to. That is why the NFS container runs privileged with `SYS_ADMIN`, and why `rpcbind` and the host's own `nfs-server` had to be stopped so they would not contend for the same kernel resources. The container is a packaging and lifecycle convenience here, not a security boundary; an attacker who broke out of that particular container would be looking at the host kernel with elevated capabilities. I think the trade was worth making for the operational uniformity of having all three services managed the same way, but it should be named for what it is rather than folded quietly into the same isolation claim that genuinely applies to the AD containers.

The reboot behaviour is the second half of this lesson, and it taught it the hard way. A hand-loaded kernel module does not survive a reboot. The first time `cclix1` was rebooted during testing, the containers all came back, their `restart_policy` is `always`, and NFS appeared to be running, but no client could mount anything, and there was no error to point at. The container was up, the export configuration was present, and clients simply hung. The cause was that `nfsd` was no longer loaded in the host kernel and the running container had nothing underneath it. The fix is the `lineinfile` task that writes the modules into `/etc/modules-load.d/` so the host loads them at boot before the containers start; Chapter 9 verifies that it works. What the validation chapter cannot convey is how long that silent failure took to diagnose the first time, precisely because every layer reported healthy. A failure that announces itself is a nuisance; a failure that looks like success is a genuine

hazard, and this was the one place in the deployment that produced one.

10.4 Cutover without a net

The migration is a cutover, not an upgrade, because the new deployment uses a different base DN (`dc=labinfad,dc=polito,dc=it`) from the legacy domain. This was a deliberate choice for a clean schema, and Chapter 4 explains why an in-place upgrade of the old NT4 domain was not attractive. The operational consequence is the part worth recording: there is no gradual rollback. Once workstations are repointed at the new domain, the old one is not running in parallel waiting to take them back. A mistake discovered after the cutover is a mistake fixed forward, on the live system, with users already on it.

In practice this raised the cost of every validation in the previous chapter, because each of those checks had to pass before the switch, not after. The idempotency testing in particular earned its place here: a playbook that can be re-run safely is not a luxury when re-running it on a live system is the only way to apply a correction. The thing I underestimated going in was how much of the engineering effort would go not into making the system work but into making it safe to touch again after it was already serving users. For a one-person lab deployment with no staging environment that mirrors production, that property turned out to matter more than almost anything else.

Chapter 11

Conclusions

The starting point of this work was a domain that worked. Using that as the premise for a migration sounds odd, but it is the point. The old LABinf NT4 domain answered requests, authenticated users, mounted home directories, and had done so for well over a decade. What was not visible from the outside was the price paid to keep it standing. Every weakness examined in Chapter 4 told the same story from a different angle: decisions taken under pressure and never revisited, settled one on top of another until that configuration had simply become the one nobody dared touch.

NTLMv1 re-enabled and SMB forced down to NT1, because otherwise the Windows 10 2016LTSB machines stopped connecting. CVE-2017-7494 left open, flagged in the production `smb.conf` by a comment that was direct and not at all reassuring. Account provisioning split across separate `ldapmodify` and `smbpasswd` calls with no recovery path if something failed halfway. The `smbldap-tools` long unmaintained, an outdated operating system underneath, every service sharing the same process tree. None of these on its own justified discarding everything. Together they described an infrastructure that had reached the end of its life.

So the question was never *how to upgrade*, but how far it made sense to rebuild rather than keep patching. The answer reached here was the most demanding of the available ones. A new Active Directory domain with a different base DN (`dc=labinfad,dc=polito,dc=it`), provisioned from the ground up and populated through a controlled migration of the existing accounts. A clean cutover, not an in-place upgrade. The cost was real, both in effort and because a cutover leaves little room for error on the day of the switch, but it made it possible to leave the entire accumulated layer of compromises behind without carrying a single one into the new system.

11.1 What was built

The result is a Samba Active Directory domain running entirely inside Podman containers on `cclix1`, across three distinct components: the domain controller, the file server joined to the domain, and the NFS server. The reasoning for splitting these into containers was argued in Chapter 6, and the consequences of that split for the AD services, in particular the ability to rebuild a container from a clean state, are taken up again from the operational side in Chapter 10. On top of the containers sits Ansible, which owns the state of the infrastructure, while the family of `cclpy.*` Python scripts handles the day-to-day operations on user accounts. Keeping those two concerns apart, infrastructure in Ansible and daily account management in Python, was one of the better structural decisions, and it is part of why neither side grew into something unmanageable.

The technical core of the work is not the containerization, though. It is the way identity stays consistent as it crosses from the Windows world into the Linux one. The POSIX attributes defined by RFC 2307 are written explicitly into the domain objects when the user is created, atomically, and `idmap_ad` reads them rather than deriving them from an algorithm. The practical effect is that the same `uidNumber` comes out of the same AD object whether Winbind reads it on the file server or SSSD reads it on a Linux client mounting a home over NFS. That is exactly the kind of consistency the old system held together by hand, and that here falls out of the structure of the data itself. Chapter 9 confirmed it end to end, following a single user's identity all the way to a file written over NFS and owned by the AD-assigned UID it would also carry over SMB.

11.2 What the work shows

Beyond the specific deployment, the migration makes a fairly general point about laboratory infrastructure of this kind. The old system was not bad engineering when it was built; it became fragile because it accumulated state that lived only in the running configuration and in the memory of whoever last touched it. The thing that actually changed with this work is not that users authenticate, they did before, but that the reason they authenticate is now written down as code, reproducible, and rebuildable from scratch. An infrastructure described in Ansible and populated by versioned tooling can be torn down and brought back up deliberately, which is a different relationship with the system than one where every change is a small act of courage.

The validation in Chapter 9 backs this up where it matters most: the deployment recovers on its own from both a container restart and a host reboot, and the provisioning playbook re-runs as a no-op rather than damaging an already-provisioned

domain. For a setup that has to be maintained by different people over several years, that property is worth more than any single feature.

11.3 Limitations

The honest limits of the system fall into two groups. The operational sharp edges, the ones discovered by living with the deployment rather than by reading its design, are the subject of Chapter 10: the privileged NFS container and where its isolation genuinely stops, the primary-group resolution that needed an explicit supplementary group to behave, and the fact that a cutover offers no gradual rollback. Those belong to the experience of running the thing, and they are recorded there rather than repeated here.

What remains to state are the limits that are architectural rather than experiential. The first is structural and the most consequential: `cclix1` is a single host, and so is the domain controller. There is no secondary DC and no replication, so the availability of the entire domain rests on one machine. For a teaching laboratory this is an acceptable and probably proportionate choice, but it is a single point of failure and should be named as one.

Two smaller limits are worth recording alongside it. LDAP connections use STARTTLS on port 389 rather than a dedicated LDAPS endpoint on 636; the choice is documented and defensible, but it is a compromise relative to a cleaner separation of channels. And the NFS service depends on the third-party `erichough/nfs-server` image: it works and it is convenient, but its maintenance and future updates are not under our control.

11.4 Future work

The obvious first step is redundancy for the domain controller. A second DC would remove the single point of failure noted above, and since Samba supports multi-DC replication the path exists; what it needs is to be designed together with the provisioning side, which today assumes exactly one DC and would have to learn not to.

The NFS container deserves a calmer second look to decide whether `privileged` is truly unavoidable or whether a narrower set of capabilities could carry it. The discussion in Chapter 10 explains why the requirement exists at all, and it may turn out to be irreducible, since NFS in a container is awkward on precisely this point, but it is the limit most worth a second attempt. Beyond that, the infrastructure would benefit from a more structured layer of monitoring and alerting than it has now, something that surfaces the state of the containers and services instead of leaving it to be discovered by hand, which is exactly the failure mode that made

the silent NFS-after-reboot problem so hard to diagnose the first time. And if the approach proves itself on LABinf, the same shape, Ansible for state, containers for isolation, Python for management, is general enough to be carried to other laboratories in the department with contained adjustments.

There is then a more ambitious direction that steps outside a single laboratory. Today LABINFAD is a fully autonomous domain, with its own users, its own RFC 2307 schema, and its own policies. One could imagine connecting it to the university's Microsoft Active Directory through a *trust* relationship, so that institutional users could authenticate against laboratory resources without replicating their accounts. Samba does support trusts towards Active Directory domains and forests, so the idea is not far-fetched.

It is, though, more complicated than the phrase *just set up a trust* suggests. Trust support in Samba has historically been less complete than on Windows Server, with known limitations in some transitivity and name-routing scenarios. There is also a knot specific to this infrastructure: accounts coming from the university domain would not carry the POSIX attributes written into our schema, because that directory is managed elsewhere, so their mapping on the Linux clients could not rely on `idmap_ad` but would need an algorithmic backend such as `idmap_rid` or `idmap_autorid` dedicated to the trusted domain. That means two different mapping logics living under one roof, one for local users and one for external ones, which is the sort of thing that has to be designed up front rather than discovered later. And it is a decision that does not rest with the laboratory alone: it would require coordination with whoever runs the central university AD, on both the technical and the policy side. I record it as an interesting scenario, university-wide single sign-on on the LABinf machines, rather than a step achievable in the short term.

11.5 Final remarks

Going back to where this started: the old domain worked, and the new one works better in ways that do not all show at a glance. The difference is not that users log in. It is that the system they log into can now be reasoned about, re-run, and rebuilt without depending on knowledge that exists only in one person's head or in a configuration nobody wrote down. For a laboratory that has to last years and pass between hands, that legibility is the most durable result of the whole effort, more than any individual piece of the stack that produces it.

Bibliography

[1] *Information Technology – Open Systems Interconnection – The Directory: Overview of Concepts, Models and Services*. Tech. rep. X.500. ITU-T Recommendation X.500 (originally published 1988, revised 1993, 2001, 2008, 2012, 2016, 2019). International Telecommunication Union, 1993 (cit. on p. 4).

[2] Wengyik Yeong, Tim Howes, and Steve Kille. *X.500 Lightweight Directory Access Protocol*. RFC 1487. Obsoleted by RFC 1777. Internet Engineering Task Force, July 1993 (cit. on p. 4).

[3] Kurt Zeilenga. *Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map*. RFC 4510. Internet Engineering Task Force, June 2006 (cit. on p. 5).

[4] Kurt Zeilenga and Jong Hyun Choi. *The Lightweight Directory Access Protocol (LDAP) Content Synchronization Operation*. RFC 4533. Internet Engineering Task Force, June 2006 (cit. on p. 6).

[5] David F. Ferraiolo and D. Richard Kuhn. «Role-Based Access Controls». In: *Proceedings of the 15th National Computer Security Conference*. National Institute of Standards and Technology / National Computer Security Center. Baltimore, MD, 1992, pp. 554–563 (cit. on p. 9).

[6] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. *Proposed NIST Standard for Role-Based Access Control*. Tech. rep. 3. The formal NIST RBAC standard is ANSI INCITS 359-2004. ACM Transactions on Information and System Security, Aug. 2001, pp. 224–274. DOI: 10.1145/501978.501980 (cit. on p. 9).

[7] Vincent C. Hu, David Ferraiolo, D. Richard Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Special Publication 800-162. National Institute of Standards and Technology, Jan. 2014. DOI: 10.6028/NIST.SP.800-162 (cit. on p. 9).

[8] The OpenLDAP Project. *OpenLDAP Software 2.6 Administrator's Guide*. 2024 (cit. on p. 11).

[9] The FreeIPA Project. *FreeIPA Documentation*. 2024 (cit. on p. 11).

[10] The 389 Directory Server Project. *389 Directory Server Documentation*. 2024 (cit. on p. 12).

[11] Red Hat, Inc. and the Keycloak community. *Keycloak Documentation*. 2024 (cit. on p. 12).

[12] The SSSD Project. *SSSD – System Security Services Daemon: Documentation*. 2024 (cit. on p. 13).

[13] Alexey Melnikov and Kurt Zeilenga. *Simple Authentication and Security Layer (SASL)*. RFC 4422. Internet Engineering Task Force, June 2006 (cit. on p. 13).

[14] B. Clifford Neuman and Theodore Ts'o. «Kerberos: An Authentication Service for Computer Networks». In: *IEEE Communications Magazine*. Vol. 32. 9. IEEE, Sept. 1994, pp. 33–38. DOI: 10.1109/35.312841 (cit. on p. 14).

[15] Clifford Neuman, Tom Yu, Sam Hartman, and Kenneth Raeburn. *The Kerberos Network Authentication Service (V5)*. RFC 4120. Internet Engineering Task Force, July 2005 (cit. on pp. 15, 57).

[16] Microsoft Corporation. *[MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3 – Specification*. Tech. rep. Open Specifications documentation, updated periodically. Microsoft Corporation, 2024 (cit. on p. 16).

[17] Arnt Gulbrandsen, Paul Vixie, and Levon Esibov. *A DNS RR for Specifying the Location of Services (DNS SRV)*. RFC 2782. Internet Engineering Task Force, Feb. 2000 (cit. on p. 17).

[18] Microsoft Corporation. *[MS-ADTS]: Active Directory Technical Specification*. Tech. rep. Open Specifications documentation, updated periodically. Microsoft Corporation, 2024 (cit. on p. 19).

[19] Luke Howard. *An Approach for Using LDAP as a Network Information Service*. RFC 2307. Internet Engineering Task Force, Mar. 1998 (cit. on pp. 21, 24, 58, 63).

[20] The Samba Team. *Samba Documentation: Setting Up Samba as an Active Directory Domain Controller*. 2024 (cit. on p. 21).

[21] The Samba Team. *Idmap config rid* (cit. on p. 24).

[22] The Samba Team. *Idmap config ad* (cit. on p. 24).

[23] Robert J. Creasy. «The Origin of the VM/370 Time-Sharing System». In: *IBM Journal of Research and Development* 25.5 (1981), pp. 483–490. DOI: 10.1147/rd.255.0483 (cit. on p. 28).

[24] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. «Xen and the Art of Virtualization». In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. New York, NY, USA: ACM, 2003, pp. 164–177. DOI: 10.1145/945445.945462 (cit. on p. 29).

[25] Ann Mary Joy. «Performance Comparison between Linux Containers and Virtual Machines». In: *International Journal of Computer Science & Engineering Survey* 6.1 (2015), pp. 1–10. DOI: 10.5121/ijcses.2015.6101 (cit. on p. 29).

[26] Andy Young. *True Sandboxed Containers with gVisor*. 2019. URL: https://gvisor.dev/docs/ (cit. on p. 30).

[27] Kata Containers Community. *Kata Containers Architecture*. 2019. URL: https://katacontainers.io/docs/ (cit. on p. 30).

[28] Michael Kerrisk. *Namespaces in Operation, Part 1: Namespaces Overview*. 2013. URL: https://lwn.net/Articles/531114/ (cit. on p. 30).

[29] Tejun Heo. *Control Group v2*. 2015. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html (cit. on p. 31).

[30] Michael Kerrisk. *Control Groups Series by LWN.net*. 2021. URL: https://lwn.net/Articles/604609/ (cit. on p. 32).

[31] Dirk Merkel. «Docker: Lightweight Linux Containers for Consistent Development and Deployment». In: *Linux Journal* 2014.239 (2014), p. 2. ISSN: 1075-3583 (cit. on p. 32).

[32] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. «Container Security: Issues, Challenges, and the Road Ahead». In: *IEEE Access*. Vol. 7. 2019, pp. 52976–52996. DOI: 10.1109/ACCESS.2019.2911732 (cit. on pp. 32, 37).

[33] Open Container Initiative. *OCI Runtime Specification*. 2016. URL: https://github.com/opencontainers/runtime-spec (cit. on p. 33).

[34] Open Container Initiative. *OCI Image Format Specification*. 2016. URL: https://github.com/opencontainers/image-spec (cit. on pp. 33, 36).

[35] Open Container Initiative. *OCI Distribution Specification*. 2021. URL: https://github.com/opencontainers/distribution-spec (cit. on p. 33).

[36] Open Container Initiative. ***runc**: CLI Tool for Spawning and Running Containers According to the OCI Specification*. 2015. URL: https://github.com/opencontainers/runc (cit. on p. 34).

[37] Giuseppe Scrivano. ***crun**: Fast and Low-Memory Footprint OCI Container Runtime Fully Written in C*. 2019. URL: https://github.com/containers/crun (cit. on p. 34).

[38] James Turnbull. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014. ISBN: 978-0-9888978-2-9 (cit. on p. 35).

[39] Docker, Inc. *Run the Docker Daemon as a Non-Root User (Rootless Mode)*. 2020. URL: `https://docs.docker.com/engine/security/rootless/` (cit. on p. 35).

[40] Containers Community. *Podman: A Tool for Managing OCI Containers and Pods*. 2018. URL: `https://github.com/containers/podman` (cit. on p. 35).

[41] Daniel J. Walsh. *Podman in Action: Secure, Rootless Containers*. Shelter Island, NY, USA: Manning Publications, 2022. ISBN: 978-1-617-29975-7 (cit. on pp. 35, 47).

[42] Harbor Project. *Harbor: An Open Source Trusted Cloud Native Registry Project*. 2016. URL: `https://goharbor.io/docs/` (cit. on p. 37).

[43] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud*. Sebastopol, CA, USA: O'Reilly Media, 2016. ISBN: 978-1-491-92420-8 (cit. on pp. 42, 51).

[44] Lorin Hochstein and René Moser. *Ansible: Up and Running*. 2nd. Sebastopol, CA, USA: O'Reilly Media, 2017. ISBN: 978-1-491-97925-3 (cit. on p. 44).

[45] Red Hat, Inc. *Ansible Documentation*. 2023. URL: `https://docs.ansible.com/` (cit. on p. 45).

[46] Red Hat, Inc. *Protecting Sensitive Data with Ansible Vault*. 2023. URL: `https://docs.ansible.com/ansible/latest/vault_guide/index.html` (cit. on p. 45).

[47] James Loope. *Pulling Strings with Puppet: Configuration Management Made Easy*. New York, NY, USA: Apress, 2011. ISBN: 978-1-430-22530-6 (cit. on p. 46).

[48] Joshua Cannon. *Learning Chef: A Guide to Configuration Management and Automation*. Sebastopol, CA, USA: O'Reilly Media, 2014. ISBN: 978-1-491-94935-5 (cit. on p. 46).

[49] Colton Rhode. *SaltStack for DevOps: Extremely Fast and Simple IT Automation and Configuration Management*. Leanpub, 2015. URL: `https://leanpub.com/saltstackfordevops` (cit. on p. 46).

[50] Charity Majors, Liz Fong-Jones, and George Miranda. *Observability Engineering: Achieving Production Excellence*. Sebastopol, CA, USA: O'Reilly Media, 2022. ISBN: 978-1-492-07644-0 (cit. on p. 48).

[51] Prometheus Authors. *Prometheus Documentation*. 2023. URL: `https://prometheus.io/docs/` (cit. on p. 48).

[52] HashiCorp, Inc. *HashiCorp Vault Documentation*. 2023. URL: https://developer.hashicorp.com/vault/docs (cit. on p. 49).

[53] Samba Team. *Setting Up Samba as an Active Directory Domain Controller*. 2012. URL: https://wiki.samba.org/index.php/Setting_up_Samba_as_an_Active_Directory_Domain_Controller (cit. on p. 49).

[54] Ansible Community. *Molecule Documentation: Testing Framework for Ansible Roles*. 2023. URL: https://ansible.readthedocs.io/projects/molecule/ (cit. on p. 51).

[55] Jérôme Tournier and IDEALX. *smbldap-tools: Scripts for Managing Samba User Accounts via LDAP*. Free Software / Open Source Project. Originally released under the GNU GPL. Repository: https://gitlab.com/samba-team/smbldap-tools. 2003 (cit. on p. 53).

[56] Microsoft Corporation. *Mitigating Pass-the-Hash (PtH) Attacks and Other Credential Theft Techniques*. Tech. rep. Version 2. Microsoft Corporation, 2014. URL: https://download.microsoft.com/download/7/7/A/77ABC5BD-8320-41AF-863C-6ECFB10CB4B9/Mitigating-Pass-the-Hash-Attacks-and-Other-Credential-Theft-Version-2.pdf (cit. on p. 55).

[57] Samba Team. *CVE-2017-7494: Remote code execution from a writable share*. Samba Security Announcement. May 2017. URL: https://www.samba.org/samba/security/CVE-2017-7494.html (cit. on p. 56).

[58] FreeIPA Project. *Active Directory Trust Setup*. FreeIPA Documentation. Describes the cross-forest Kerberos trust required for FreeIPA to serve Windows Active Directory clients, relying on Samba components on the IdM side. 2024. URL: https://www.freeipa.org/page/Active_Directory_trust_setup (visited on 06/15/2026) (cit. on p. 71).

[59] Red Hat, Inc. *Podman and systemd: Running Containers as System Services*. Podman Documentation. Daemonless container model; container lifecycle managed through systemd units rather than a persistent root daemon. 2024. URL: https://docs.podman.io/en/latest/markdown/podman-systemd.unit.5.html (visited on 06/15/2026) (cit. on p. 72).

[60] The Samba Team. *Setting up Samba as an Active Directory Domain Controller*. SambaWiki. 2026 (cit. on p. 73).

[61] Canonical Ltd. *Provisioning a Samba Active Directory Domain Controller*. Ubuntu Server Documentation. 2026 (cit. on p. 73).

[62] The OpenLDAP Project. *Using TLS: StartTLS and Connection Security Considerations*. OpenLDAP Software Administrator's Guide. StartTLS upgrades a plaintext LDAP connection on port 389 to TLS; without strict client enforcement the negotiation is exposed to downgrade (TLS-stripping) attacks. 2024. URL: `https://www.openldap.org/doc/admin24/tls.html` (visited on 06/15/2026) (cit. on p. 76).