IT Project Guidance – On Non-Prod Environment Data

Version: 1.0

Purpose

This document defines responsible, compliant, and sustainable practices for avoiding the misuse of production data in non-production data environments. It highlights common organisational failings, explains the legal and technical risks, and offers practical guidance for designing durable test data frameworks. It asserts that pre-production (PP) environments must also comply with the same standards unless they are fully governed and auditable at the level of production systems.

Synopsis

Many organisations remain immature in their data handling practices, especially when it comes to test and pre-production data environments. Despite privacy laws, sector-specific regulations, and growing scrutiny, production data continues to be reused in development and testing. This document identifies the core failure points, debunks misconceptions such as obfuscation, and advocates for intentional test data design based on domain principles. It affirms that any environment using production data must be held to production standards—something that almost never happens in practice.

Contents

Purpose and Audience

This document is for system architects, development leads, product owners, privacy and security officers, and project governance teams. It provides clear direction for avoiding the misuse of production data, designing appropriate test data models, and ensuring that pre-production data environments are not used as a loophole to bypass data controls.

Scope

The guidance applies across all environments that are not explicitly designated and operated as production. This includes development, build, test, integration, user acceptance testing, and pre-production data environments. It also covers operational areas such as automated deployment systems and backup management. Test code and functional testing frameworks are out of scope.

Background

Most organisations do not treat data as an architectural concern until an incident occurs. The default development culture is reactive: bugs arise in production, and developers resort to inspecting or reusing that data in lower environments to diagnose and test fixes. These habits persist despite the presence of privacy regulations and internal security frameworks.

Global privacy laws such as GDPR, CCPA, and the NZ Privacy Act impose strict requirements on data usage. Sector obligations—for example, public sector compliance with the New Zealand Information Security Manual (NZISM)—further restrict how and where data can be processed.

Penalties can include prosecution, fines, sector and public reputational damage, and service interruption. Loss of trust and consumers is a natural consequence. Personnel changes are likely—ranging from formal warnings to demotion or termination—particularly where negligence, scale of impact is large, or repeated violations are involved.

The majority of breaches are human in origin: accidental inclusion of data in logs, misrouted emails, credential exposure, or deliberate exfiltration. System penetrations are far less common than failures of process and design. Against this backdrop, weak test and deployment practices represent a persistent risk.

Production Data is Not Test Data

The fundamental error is the belief that production data is appropriate for testing.

Even when obfuscated, subsetted or anonymised, production data reflects historical states, not test intent. It is reactive, fragile, and poorly suited to uncovering defects at the edge of logic. Worse, it conditions teams to rely on happenstance rather than deliberate test coverage.

In environments such as pre-production, this behaviour becomes institutionalised. Tests "only work" when production data is used. This signals that the testing is broken. If a test cannot pass using designed, intentional test data, then either the test is invalid or the code is unfit for release.

Production Issue Data is not Test Data

A widely overlooked risk is the use of incident-related records as test data. Tier 2 or Tier 3 support staff often access production to investigate defects or inconsistencies, isolating a record that illustrates the issue and forwarding it to developers. This act informally converts live production data into test data by reference or direct reuse.

Developers or testers may then use that record in test suites or duplicate it into non-production data environments for debugging. This embeds real, identifiable user data into test datasets. Over time, the data’s origin and sensitivity are forgotten, and it becomes part of backups, logs, or test artefacts.

Such practices are harmful. They breach privacy, create inconsistent test conditions, and encourage poor analytical discipline. Analysts may misinterpret incidental data as representative. Developers may base features on unrepresentative quirks rather than domain logic.

Instead of reusing production data, the proper response is to abstract from it. If a production case reveals a flaw, teams should model synthetic data that reproduces the error class, not the individual record. Lower environments must be built on lawful, synthetic data that is comprehensible and traceable. Using real data for debugging—no matter how well intended—compromises both design integrity and governance.

Subsetting Production Data is not Test Data

Subsetting is often treated as a safer compromise—extracting only part of production data by sampling, filtering identifiers, or narrowing the population. But this approach is unreliable and erodes the purpose of testing.

First, subsets lack clear definition. What do they represent? Were edge cases, behaviours, exceptions, and user types intentionally included? Most subsets are chosen for convenience or randomness, which rarely captures the full scope of system logic or business rules.

Second, production data changes constantly. Even a well-chosen subset becomes outdated quickly. Tests built on it lose repeatability and mislead teams. Using only expired data makes it stale; using live data reintroduces privacy and operational risk.

Third, no subset can cover invalid or edge-case scenarios unless synthetic data is added. But once you mix synthetic and real records, you must manage them as a separate schema—essentially recreating a test dataset. It is more honest and effective to start with deliberate test data.

Subsetting typically arises when no proper test data strategy exists. It is used because it's easy, not because it's right. Once adopted, it becomes embedded in scripts and systems, reinforcing the false belief that production data is needed for testing.

Subsetting is not a compromise—it's a quiet surrender of rigour. A better approach is to build test data from first principles: versioned, traceable, and designed around specific use cases. Production data serves reality. Test data must serve clarity, intent, and verifiability.

Putting in Test Data in Production Data

A less common but equally serious risk is the reverse of copying prod data into test: inserting test data into production. Though often done with good intentions—verifying features, simulating edge cases, or demoing under load—it quickly compromises the integrity of the production data environment and its analytics.

Test data, unlike real records, is often malformed, contradictory, or deliberately incomplete. Its presence distorts reports, alerts, and monitoring tools that assume all production data is genuine. This is especially harmful in fields like education or health, where operational data informs audits, funding, and compliance.

For instance, inserting test enrolments into a live school system may result in incorrect capacity calculations or trigger alerts based on fake thresholds. A fictitious learner or teacher may appear in reports or public dashboards. The errors embedded in such data—intended for testing—are now treated as real, damaging trust in the system.

Moreover, it erodes professional discipline. Developers may start viewing production as a testbed, weakening change control boundaries. These test records are rarely fully removed, and the habit can persist, making future governance even harder.

There is no safe justification for placing test data in production. It violates the core principle that production systems must reflect authentic, auditable state. If real-world scenarios must be trialled, they should occur in pre-production systems that faithfully replicate architecture but use only synthetic, safe data.

Test data belongs in test environments. Never in production.

Preprod Data is not Production Data

A common but dangerous misconception is that pre-production data environments are equivalent to production, and thus safe for using production data. While pre-prod may share architecture—such as integration with a live IdP—it usually lacks the governance, monitoring, and legal oversight required of true production systems.

Production status is defined by more than infrastructure. It includes full auditability, role-based access, incident response, threat detection, operational logging, and validated user entitlements. Pre-prod typically does not meet these standards. It is intended for final-stage validation or deployment rehearsal, not for handling live customer data.

Introducing production data into pre-prod effectively turns it into a production system—without any of the protections. If an environment isn't governed as production, it must not contain production data. Doing so risks legal exposure, reputational harm, and blurs critical distinctions between testing and operations.

Using real data in pre-prod also damages test integrity. It breeds false confidence, erodes compliance, and encourages teams to design tests that only work on live data—defeating the purpose of testing and masking flawed logic.

Pre-production is still non-production. Unless governed at production standards, using real data in it is a breach. The correct approach is to model test data that is versioned, traceable, and scenario-specific. Production data reflects reality. Test data must serve clarity, reproducibility, and verification. Always—and only—in test environments.

Backup Data Is Production Data

Prod Backup data is also a production data environment asset. It's an integral part of the environment, simply using a different device than the operational data store. It is *not* a separate legal environment. Hence, not acceptable as a source of test data.

It should NOT be made accessible to developers or testers in any environment.

It is only to be used by automated recovery systems under strict deployment controls.

Any access to production backup must be audited, approved, and justifiable under operational incident response.

Essentially, allowing access is allowing employees to hack the organisation's systems – just a month ago. 99% of the records are still. The same. Leakage of the backup data can materially impact privacy and ongoing contractual discussions.

Obfuscated Production Data Is Not Test Data

Obfuscation is often promoted as a way to make production data “safe” for use in test environments, by masking names, emails, and identifiers. This belief is both false and dangerous.

Obfuscation is not anonymisation. It alters visible details but retains the underlying structure, behaviour, and statistical traits of the data. Modern techniques—including AI-driven correlation and metadata analysis—can re-identify users, especially where unique patterns or outliers exist.

Crucially, obfuscated data is still production data. It wasn’t created for testing purposes and lacks the deliberate coverage, edge cases, and controlled conditions that proper test data requires. At best, it imitates real data without clarity. At worst, it provides misleading signals, weakening both test accuracy and governance.

Like subsetting, obfuscation adds ambiguity. Reliable testing depends on crafted inputs with clear, verifiable outcomes. Obfuscated data undermines this—it’s opaque, difficult to trace, and leaves testers uncertain whether failures stem from code or data flaws.

Obfuscation also misleads stakeholders. It creates audit gaps, falsely reassures project teams, and discourages investment in proper test data design. Critically, it does not release organisations from legal obligations. If privacy is breached, masked data is still personal—and liabilities remain.

In the end, obfuscation conceals risk rather than resolving it. It delays the adoption of versioned, schema-aware, lawful test data strategies and embeds poor practice into delivery workflows. It is not a real solution—and no mature system or organisation should treat it as one.

Anonymised Data Is Not Test Data

A common defence for using production data in test environments is to anonymise it—removing names, identifiers, or shifting dates to reduce privacy risk. While this may lower disclosure concerns, it does not make the data suitable for testing. Anonymised production data is still production data. It reflects what worked, not what must be tested.

By nature, it contains valid, already-accepted inputs. It lacks malformed, incomplete, or intentionally invalid cases that testing requires. Systems tested on anonymised data may appear robust while silently failing to reject flawed inputs—because none are present.

Moreover, anonymisation strips out context. Learner enrolments may no longer match school terms. Roles may no longer align with permissions. Cross-entity relationships often break or become incoherent, making integration testing unreliable.

It also offers no version control, traceability, or alignment to expected outcomes. It is a frozen snapshot, not a reusable or evolving test dataset. It cannot support regression testing or demonstrate that validation rules catch failures.

In short, anonymisation avoids risk but also avoids rigour. It delays the real work: crafting test data with known edge cases, failure states, and business logic that must be verified. Systems should be tested against what might go wrong—not just what already went right. Anonymised data can't do that.

A Defensible Approach

Production data must never be used outside the production data environment unless the non-production data environment is:

- Explicitly governed at the same level
- Covered by audit and access controls
- Logging all access to the data
- Included in the same incident response plan
- Operated using the same credentials, monitoring, and protection standards

Almost no organisations meet these conditions. Therefore, any environment—*including pre-production,* including *integration test environments* —must not use production data. This includes credentials, transaction records, or any dataset containing personal, commercial, or operational information.

Designing Durable Test Data

Intentional test data must be crafted to reflect domain logic rather than tied to specific schemas. It should validate core, exceptional, and edge-case behaviours; be version-controlled, portable, and maintained independently; and remain free of system-specific identifiers or configurations.

In long-lived organisations, systems are short-term. Testing must therefore focus on enduring domains and their entities—Persons, Personas, Groups, Relationships, Resources—using data that can be transformed as systems evolve. This demands data shaped by domain understanding[1], not copied schemas.

In education, for example, tests should represent enrolments, teacher assignments, learner-artefact interactions, support roles, achievements, and identity changes across

[1] Use mature Domain Driven Design (DDD) methodologies.

providers. These cannot be faked using system-derived fields—they must be designed through domain analysis.

As a baseline, each core entity should have at least 42 records to enable pagination testing—two full pages plus a partial. These records must represent the full spread of actors and states: learners, teachers, parents, support roles, with conditions including active, pending, rejected, or edge-case transitions like aged-out or dual enrolment.

Test data should represent concepts that persist even as systems change. It must be designed with intent, so that its use verifies workflows and behaviour, not just technical correctness.

Regarding Integrations

Integration is among the hardest areas for responsible test data management, particularly where systems vary in maturity. Even if a project team uses abstract or synthetic data, it often has to interact with systems still reliant on real or obfuscated production data.

This leads to misalignment. In education, for instance, a test enrolment system may link test learners to schools, but if the integration layer only accepts real Ministry IDs, teams must either contaminate the test environment with production codes or allow the link to fail.

The problem deepens in multi-system setups where identity brokering, messaging, or reference data like qualifications or funding codes aren't coordinated across test environments. Ad hoc solutions—shared stubs, partial mirrors, hand-built lookups—add fragility and drift, increasing the chance of error and rework.

These issues are common in federated environments like education, health, or local government, where no single system governs all data. Integration points may seem safe because they carry system messages, but they become contamination risks when any payload includes live or derived production data.

True integration testing with synthetic data requires system-wide alignment. That means defining shared test identifiers, maintaining test-specific mappings, and designing datasets that are realistic but unmistakably synthetic.

This effort is often postponed because it’s complex. But skipping it leads to real data use by default—undermining any privacy or compliance claims.

To solve this, integration datasets must be treated with the same care as internal models. If live reference data must be used, it should be mirrored into read-only, test-specific form. Systems unable to support test data should be accessed via sandbox endpoints or emulated interfaces.

Without deliberate coordination, integration testing lacks control. Once one system uses convenience over caution, trust breaks down. Maturity must be shared—otherwise, so is the risk.

Regarding Integrations to IdPs

The use of a production data environment Identity Provider (IdP) is not, in itself, justification for using production data.

A PP environment may use production authentication to confirm real user identity, but those identities are used to access non-production data. The users must still be protected (e.g. logging, entitlements), but the datasets accessed must remain either be a copy of test data as a starting point, or synthetic demonstration/end user usable (e.g. similar to what was used earlier in UT) data.

Note that this is yet another reason to not embellish or pad identity tokens in IdPs or their brokers with information gleaned from other systems. Identity tokens should be compact, purpose-specific, and limited to what is needed to authenticate a user *to* the system—not to authorise them within the system. These are separate concerns. Authentication confirms identity and initial access to the system; authorisation within the system (e.g., what data they can view, modify, or create) must be managed by the system's own access control logic, not injected prematurely via upstream claims.

Mixing these layers not only creates brittle, opaque access models but also introduces serious operational complications. In particular, it becomes difficult or impossible to align production identities—packed with production-derived attributes—with synthetic test data and roles generated by the system itself. This mismatch results in broken testing, unclear permissions, and cross-contamination of environment assumptions. Including derived or enriched attributes from other systems—such as roles, group memberships, profile data, or inferred relationships—also introduces cross-system coupling and significantly increases the surface area for privacy breaches.

In non-production data environments especially, where datasets are not subject to the same level of scrutiny or protection, enriching tokens with production-derived metadata creates multiple risks. These include:

- Unintentional leakage of sensitive information through logs or test UI elements
- Confusion between synthetic and real data due to presence of hybrid tokens
- Increased likelihood of access violations when assumed roles or entitlements span multiple domains
- Loss of traceability and control, especially when tokens are reused or cached

Such practices also reduce architectural clarity. Tokens should contain what the identity provider is accountable for—not a mixture of upstream or system-derived knowledge. If additional claims are required, they should be retrieved through secure APIs with proper access controls, not baked into the token itself. This approach maintains separation of concerns, simplifies auditing, and reduces accidental overexposure.

In summary, padding identity tokens with context from other systems undermines both data protection principles and system integrity. Tokens should remain as lean and isolated as possible, especially in environments handling synthetic test data.

Designing Durable Cross System Test Data

To ensure integration is sustainable and test coverage meaningful, test data must be based not on the systems an organisation operates, but on the *business services* it delivers. Systems are transient. Business services are enduring. Test data designed around business logic and domain behaviours remains valid even as IT systems evolve or are replaced.

The mistake many teams make is defining test data within the constraints of current applications or IT service boundaries—tightly coupled to schemas, IDs, and internal logic. But those systems are merely tools. They support business services; they do not define them.

Take education as an example. The organisation may run separate systems for managing providers, learners, and enrolments. Each has its own schema, APIs, and identifiers. But these are IT services. The *business service* is far older and enduring: enabling people to form relationships with institutions that provide learning. That service predates the internet—and it will outlast the current generation of software.

To build test data that reflects that, organisations should define their datasets in neutral, structured formats—typically JSON or a schema that can be exported as JSON—that express concepts like people, places, groups, roles, relationships, and life events. These are the core entities that business services manage, regardless of the IT systems used to track them.

By defining test data in this way, the organisation avoids coupling to system internals. It reflects what actually happens in the service: people apply late, change identity, receive support, or drop out. This way, the data remains relevant whether it is loaded into today's platforms or tomorrow's.

Loading Durable Test Data into Services

For production or migrated data, records must *always* be loaded through the system's public APIs. Direct-to-database ETL bypasses validation, authorisation, and logging. It breaks audit trails, allows invalid state combinations, and leads to data integrity failures that are difficult to detect or recover from. That practice should be avoided entirely.

However, *test data is a special case*. It must include edge cases and deliberately invalid records—those that would be rejected by validation layers.

In ideal circumstances, the service would expose APIs specifically designed for test data upload that bypass validation, with strict safeguards to ensure such endpoints are

disabled in production data environments. This preserves traceability while supporting the controlled insertion of non-standard data.

However, in most organisations, such facilities do not exist or are infeasible to implement across all systems. In these cases, ETL directly to the system datastore remains the only remaining workable approach. Where this occurs, it must be treated as a deliberate and tightly controlled exception, confined to non-production data environments -- with a long-term view on building safer and more transparent alternatives.

Conclusion

Test data is not a technical detail—it is an architectural asset. Its design, governance, and use must be treated with the same seriousness as source code or security configuration.

The use of production data outside production data environments is never acceptable unless that environment is governed identically to production. Since that is rarely achieved, the default stance must be prohibition.

Organisations must move beyond obfuscation, sub-setting, and historical crutches. They must adopt purposeful, domain-aligned test data strategies, treat their test data as system-independent artefacts, and maintain compliance with privacy and sector standards at all times. Anything less risks privacy, safety, trust, punitive enforcement, and future operability of the service.

Appendices

Appendix A - Document Information

Authors & Collaborators

- Sky Sigal, Solution Architect

Versions

0.1 Initial Draft

1.0 First Release

Images

No table of figures entries found.

Tables

No table of figures entries found.

References

There are no sources in the current document.

Review Distribution

The document was distributed for review as below:

Identity	Notes
Alan Heward, Senior Advisor, Digital Assurance	

Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

Structure

Where possible, the document structure is guided by either ISO-* standards or best practice.

Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

Acronyms

API : Application Programming Interface.

DDD : Domain Driven Design

GUI: Graphical User Interface. A form of UI.

IdP : Identity Provider, a remote service for issuing tokens representing users within it, in return for them providing correct credentials, such that the token is trustable by a another service.

IT : acronym for Information, using Technology to automate and facilitate its management.

UI : User Interface. Contrast with API.

Terms

Refer to the project’s Glossary.

42 : a mythical number amongst geeks that has no specific meaning. However, useful as a minimum number of entities to return to demonstrate correct paging.

Application Programming Interface : an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability : a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain : a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Environment : a logically or physically distinct deployment context used during the development, validation, or operation of software and services. Environments serve different purposes along the delivery lifecycle and may share code, infrastructure, or credentials, but must maintain clear data and access boundaries. The typical environment types used within a service delivery pipeline include:

- **Build Test (BT):** Automated pipeline stage used to compile, unit test, and validate code integrity on commit.

- **Dev Test (DT):** Used by developers for debugging and local verification during active development.
- **System Test (ST):** Used by test analysts for structured exploratory testing of internal business logic and workflows.
- **User Test (UT):** Used by stakeholders, sponsors, or product owners to assess usability, user flow, and acceptance criteria.
- **Integration Test (IT):** Used to test connectivity and behaviour between systems, typically across organisational or service boundaries.
- **Training (TR):** Used for staff or end-user training purposes. Should contain only synthetic, representative data designed for instructional use.
- **Pre-Production (PP):** A full-fidelity clone of production architecture, used for final smoke testing and last-stage validation before deployment. It may integrate with live identity providers but must not hold production data.
- **Production (PR):** The live, operational environment where services are consumed by end users and external systems. All data in this environment is considered authoritative and subject to full privacy, security, and availability obligations.

Each environment must have clearly defined governance, access controls, and data sourcing rules. Informal or ambiguous environment use increases the risk of security breaches, data mishandling, and system misbehaviour.

Non-Production Data Environment : an environment that does not contain live, authoritative, or operational production data. These environments are used for development, testing, training, or pre-release validation. They may use synthetic data, anonymised samples, or representative datasets designed to simulate production scenarios, but must not contain real customer, stakeholder, or operational data unless governed to production standards. Non-production environments must be clearly separated, auditable, and protected to avoid cross-contamination with production systems.

Production Data Environment : commonly referred to as a “Production Environment,” but this is a misnomer. It is not the environment’s infrastructure that defines its classification, but the nature of the data it contains. Any environment holding live, authoritative, or operational production data—regardless of its purpose, accessibility, or hosting context—is a production data environment. These environments must be governed, audited, and protected to the same standard as frontline systems, even if not user-facing.

Test Data : data created or selected to validate system behaviour under known conditions. It includes both valid records (expected to succeed) and deliberately invalid records (expected to fail). Test data is synthetic, domain-aligned, and not derived from

production. It must be versioned, traceable to test scenarios, and safe for use in non-production environments only.

Appendix B – Pattern for Deferred Attribute Resolution via Trusted APIs

A sound architectural pattern separates authentication from authorisation by keeping identity tokens minimal and resolving additional user attributes only when needed, via secure API calls. This avoids premature or over-broad inclusion of cross-system context in identity tokens.

In this pattern, the consuming service redirects the end user to an external Identity Provider (IdP) to authenticate. The resulting token asserts identity and includes a stable, scoped identifier. Upon return, the service starts a new session and either looks up or creates an internal user record linked to the IdP identifier.

At this moment—and not before—the service may call an external profile or attribute service, using its own access credentials or delegated authorisation (see *Appendix C*), to retrieve further information needed to populate roles, profiles, or contextual associations. What data is fetched depends entirely on the service's own needs: for one system it may be classroom entitlements; for another, reporting lines or support relationships. This avoids assumptions or unnecessary coupling.

Crucially, this puts responsibility where it belongs: on the consuming system to determine what it needs, when, and from where. While it may require more effort on that system's part, this is appropriate. Reusability, traceability, and clean separation of concerns outweigh the convenience of overloaded tokens.

A common pushback is that SaaS vendors "can't be told" what their IdP tokens must contain. That's correct—but irrelevant. If a system *requires* certain information to function, it must already have logic and dependencies to obtain that information, which can be met. If not, it had no business relying on its presence in a token in the first place. The pattern described doesn't restrict capability; it simply ensures capability is obtained deliberately, through clear boundaries.

Appendix C - How the Consuming Service Gets Authorised to Query the Remote System

When a service needs to retrieve user-specific attributes from an external system (e.g. a profile, roles, or context service), it must be authorised to do so. This does **not** happen through the user's identity token. Instead, the consuming service itself becomes a recognised client of the remote system, and is granted permission to access specific APIs on behalf of the organisation.

This typically works through **OAuth 2.0 Client Credentials flow**, which is designed for machine-to-machine communication. The process is as follows:

1. **Client Registration**: The consuming service is registered as a client in the remote system's identity or access control platform (e.g. Azure AD, Auth0, or the SaaS vendor's developer portal). This may involve an admin or support engineer configuring a "machine account" or "service principal."
2. **Credentials Issued**: As part of registration, the consuming service is issued a **client ID and secret** (or sometimes a certificate or private key) which it stores securely. These are used to authenticate the service itself—not the user.
3. **Token Request (Client Credentials Grant)**: When the consuming service needs to make an API call, it authenticates with the remote service's token endpoint using these credentials. The service receives an access token that authorises it to call specific APIs.
4. **Scope and Permissions**: The token issued will include **scopes or claims** that define what the client is allowed to do. These scopes are configured by an **admin of the remote system** during setup. They define what parts of the API the consuming app can access and under what constraints. For example, the consuming app might have permission to call /users/{id}/profile or /entitlements but not write to those records.
5. **API Call with Token**: The consuming service includes the access token in the Authorization: Bearer header when calling the remote API. The remote service validates the token and responds accordingly.
6. **Caching and Renewal**: The consuming service may cache the data locally for session duration and renew its token periodically via the same OAuth flow.

This model means that **the consuming app is explicitly authorised** to access only what it needs, through an API contract and a secured token, independent of any claims embedded in a user token.

By isolating the **authentication of the user** (via IdP) from the **authorisation of the system** (via OAuth credentials and scopes), this approach maintains clean boundaries, aligns with the principle of least privilege, and avoids tightly coupling identity systems across domains.