White Paper “Demonstrable Governance”

“Turning Organisational Policies into Verifiable Guarantees”

Designed and written by Bruno URBAN
Content produced with the assistance of an LLM (AI)
Date: September 2026
Contact : integration@sotead.com

Foreword

This white paper arose from four questions which, although they concern different areas, ultimately raise the same fundamental issue: **how can we build an information system capable not only of applying an organisation's decisions, but also of demonstrating that they are actually being complied with?**

Cyber Defence

Automated attacks are increasing, and the use of artificial intelligence is progressively enhancing their adaptability and effectiveness. At the same time, automation and AI are playing an increasingly important role in detection and response mechanisms.

This development nevertheless raises two fundamental questions:

- What happens when the first line of defence is circumvented or overcome?
- What guarantees then remain within the information system itself?

Perimeter defence, however effective, cannot constitute the final line of protection on its own. Protected functions must themselves have mechanisms capable of controlling the conditions under which they agree to execute, the components they use and the data they consume.

When a guarantee required for execution can no longer be established, a decision must then be made: should continuity of processing be prioritised, at the risk of propagating a compromise, or should execution be refused, at the risk of causing unavailability?

This decision is not merely technical. **It is a matter of governance and must be capable of being explicitly defined by the organisation.**

Cyber Resilience

No organisation can assume that it will always be able to prevent a cyberattack.

Cyber resilience therefore requires preparation not only for defence, but also for the ability to maintain, rebuild or restore essential functions following a major incident.

This requires the organisation to be able to answer several questions:

- What are the organisation's critical functions?
- What is their actual scope, and on which resources, data and other functions do they depend?
- How can they be rebuilt or restored quickly within an environment in which trust can be established?
- What loss of data, capability or functionality is acceptable?
- What degraded mode of operation is the organisation prepared to accept?

Answering these questions requires, among other things, an up-to-date inventory of critical functions and their dependencies, controlled deployment mechanisms, backed-up and recoverable

data, and periodic exercises capable of demonstrating the organisation’s actual restoration capability.

Resilience therefore depends heavily on the knowledge an organisation has of its own information system.

Yet this knowledge is often maintained independently of the system it describes and can rapidly become incomplete or obsolete.

Ideally, the information system itself should contribute to producing and maintaining an exploitable representation of its functions, their dependencies and the guarantees associated with them.

Such knowledge could then become not merely documentary, but also usable for impact analysis, restoration, control and decision-making.

Governance

Organisations define numerous policies intended to govern their information systems. These stem from regulatory obligations, security requirements, data governance policies and operational constraints, but also from strategic choices, particularly in relation to resilience or sovereignty.

Defining these policies is a first challenge. Actually integrating them into the information system and demonstrating their application constitutes a second, often far more complex one.

Even when an organisation directly develops fewer and fewer of the solutions that make up its information system, the services it uses still need to be integrated with one another. This integration necessarily requires technical mechanisms: code, integration components, platforms or orchestration solutions.

The question then becomes how these integration points can be used to apply the organisation’s policies and produce the evidence required to demonstrate compliance with them.

A decision-maker should therefore be able to answer the following questions:

- How are the organisation’s policies actually applied within the information system?
- How can it be demonstrated that the corresponding controls and actions were actually executed?
- How can a strategic decision or a new policy be propagated rapidly when numerous business functions are affected?
- How can this ability to evolve be preserved without imposing identical life cycles on the different teams involved?

Answering these questions requires reducing the dependency between business logic and the mechanisms responsible for applying governance.

It also requires moving from governance based primarily on prescriptions and retrospective controls towards governance capable of producing, during the very execution of the information system, the evidence required to demonstrate that it has been applied.

Sovereignty

Modern information systems rely on a growing number of external services, technologies and providers. This development brings agility, but it also creates dependencies that may limit an organisation's ability to evolve its information system according to its own choices.

Sovereignty therefore cannot be reduced to the location of data or the nationality of a provider. It also concerns the organisation's ability to retain control over its processing, its data, its policies and the evidence required for their governance.

Several questions therefore arise:

- To what extent does a business function depend on the technology used to store or transport its data?
- Can an infrastructure or provider be replaced without calling the business logic into question?
- Does the organisation retain control over the elements required to understand, rebuild and control its processing?
- Does the evidence used to establish trust remain usable independently of the solutions that produced it?

Sovereignty does not mean the absence of dependencies. Rather, **it means knowing those dependencies, controlling them and preserving the ability to replace them when necessary.**

Conclusion to the Foreword

These four issues are generally addressed separately. Yet they share the same underlying difficulty: **the organisation must know what it executes, control the conditions under which it executes it, and possess objective evidence enabling it to establish trust.**

Defence is not sufficient if no guarantees remain once the defences have been breached.

Restoration is not sufficient if trust in what has been restored cannot be established.

Governance is not sufficient if the application of decisions cannot be demonstrated.

Finally, choosing technologies is not sufficient to ensure sovereignty if business functions become inseparable from those technologies.

This white paper explores an approach intended to make these guarantees **explicit, executable and demonstrable**, while preserving, as far as possible, the independence of business functions from the technical mechanisms that support them.

Chapter 1 — Why Do We Need a New Paradigm?

1.1 The Current Situation

For several decades, software development has focused primarily on delivering business functionality.

The quality of software was mainly measured by its ability to correctly perform the expected business operations.

Today, this perspective is no longer sufficient.

Modern software is no longer simply a collection of business functions.

It operates within a distributed information system, connected to numerous services, processing data of many different kinds, and subject to growing requirements relating to security, regulatory compliance, operational resilience, data governance and digital sovereignty.

As a result, the value of software no longer depends solely on **what it does**, but also on **how it does it**.

1.2 A Change in Nature, Not Merely in Technology

Distributed architectures, virtualisation, cloud computing, APIs, SaaS platforms and large-scale data processing have progressively removed the traditional boundaries of enterprise information systems.

Software no longer controls its entire execution environment.

Data is constantly moving between organisations, geographical regions, cloud providers and heterogeneous technical platforms.

In such an environment, the responsibility carried by software extends far beyond its own functional scope.

1.3 A Growing Number of Responsibilities

Modern software must now satisfy several categories of requirements simultaneously.

The **vertical requirements** directly support business objectives:

- manage customers;
- calculate prices;
- issue invoices;
- schedule field operations.

The **horizontal requirements** concern the information system as a whole:

- security;

- data governance;
- regulatory compliance;
- auditability;
- observability;
- resilience;
- operational governance;
- enterprise strategy.

Unlike business functionality, these requirements apply across the entire information system, irrespective of the purpose of each individual application.

1.4 The Current Challenge

Today, these horizontal requirements are expressed primarily through:

- policies;
- recommendations;
- procedures;
- standards.

However, they remain largely disconnected from their technical implementation.

In practice:

- security teams define security policies;
- data governance teams define rules relating to data;
- compliance teams establish regulatory requirements;
- enterprise architects publish architectural guidelines;
- development teams attempt to implement some of these requirements within their applications.

This organisation has several limitations:

- responsibilities become fragmented;
- the actual level of compliance is difficult to assess;
- audits become the main verification mechanism.

Ultimately, implementation depends to a large extent on how each development team interprets the organisation’s requirements.

1.5 The Limitations of Traditional Approaches

Most organisations have responded to these growing requirements by introducing additional processes.

- development methodologies have evolved;
- quality gates have multiplied;
- code reviews have become systematic;
- static and dynamic code analysis tools have been introduced;
- operational monitoring has improved considerably.

Although each of these initiatives addresses part of the problem, they generally operate independently.

As a result, organisations often accumulate controls without being able to demonstrate that the expected guarantees are actually maintained throughout the execution of a business process.

The problem is therefore no longer the absence of controls.

It is the absence of a coherent model capable of linking organisational policies, technical controls and verifiable evidence.

1.6 Trust Can No Longer Be Assumed

Traditionally, trust is established through assumptions.

Software is considered trustworthy because:

- it was developed by an internal team or by a team managed internally;
- it passed acceptance testing;
- it was reviewed;
- it was deployed using approved procedures.

These assumptions were generally acceptable when systems evolved slowly and remained relatively static.

Modern information systems evolve continuously.

Software may be deployed several times a day.

Dependencies are updated automatically.

Infrastructure is provisioned dynamically.

Artificial intelligence is increasingly contributing to code generation.

In such an environment, trust can no longer rely solely on prior validation.

It must be established continuously throughout execution.

This is one of the fundamental principles of a **Zero Trust** approach.

Nothing is considered trustworthy simply because it has been trusted before.

Each important property must be verified when it matters.

1.7 From Trust to Demonstration

This observation naturally leads to a fundamental question:

How can an organisation ensure that its policies are not merely documented, but are actually applied?

Traditional governance generally addresses this question through audits.

Audits remain essential, but they usually take place after deployment and assess samples of evidence collected over time.

The architecture proposed in this document follows a different philosophy.

Rather than assuming that organisational policies have been respected, the approach verifies their application before, during and after each business execution.

Trust therefore no longer rests on declarations.

It is established through **demonstrable guarantees**.

Each guarantee is achieved by at least one technical action and/or control.

Each action and/or control generates verifiable evidence.

Together, these items of evidence form a verifiable chain demonstrating that the expected policies were actually applied.

1.8 A Different Perspective

This document does not present yet another integration platform.

Nor does it propose another orchestration engine.

Its objective is fundamentally different.

It introduces an execution model in which organisational policies become **executable, verifiable and auditable guarantees**.

Business processing is no longer executed simply because it has been requested.

It is executed only when the conditions required to establish trust have been demonstrated.

Execution therefore becomes the consequence of verified guarantees rather than the starting point of the process.

1.9 Digital Sovereignty: Beyond Infrastructure

Digital sovereignty is often reduced to the choice of a cloud provider or the location of data.

While these aspects are important, they represent only part of the issue.

An organisation also loses part of its sovereignty when its business processing becomes dependent on a particular technology, a proprietary data format, an integration engine or a service provider whose replacement requires changes to the software itself.

The architecture proposed in this document aims to reduce these dependencies by separating business logic from the technical mechanisms used to access data, transport it, apply governance policies and produce the evidence required to demonstrate compliance.

Business components manipulate logical representations of data rather than physical representations.

Interactions with storage, transport and exchange infrastructures are delegated to specialised components.

This separation reduces technological dependencies and facilitates infrastructure evolution without calling business processing into question.

Technological changes, such as replacing a file system with object storage, introducing an HTTP API or migrating to a new platform, can therefore be carried out with limited impact on business processing.

This approach does not eliminate infrastructure dependencies, which remain necessary for execution, but it isolates them within specialised components.

The organisation therefore retains control over its business logic, its policies, its execution evidence and the knowledge produced about its information system.

Sovereignty therefore does not arise solely from owning infrastructure.

It also relies on the organisation’s ability to evolve its processing, its data and its governance mechanisms independently of the technologies that support them.

1.10 Chapter Conclusion

The growing complexity of modern information systems requires a different approach to software architecture.

Business functionality alone is no longer sufficient.

Organisations must also be able to demonstrate that their software complies with security policies, governance requirements, regulatory obligations and operational constraints.

The approach presented in this white paper proposes moving from **declarative governance** to **demonstrable governance**.

Rather than relying mainly on procedures and periodic audits, it introduces a model in which organisational policies are translated into technical actions and/or controls that produce verifiable evidence throughout execution.

This approach deliberately favours **denial of service over uncontrolled execution**: whenever a required guarantee cannot be demonstrated, execution is stopped.

Although this security posture may not be appropriate for every organisation, it reflects a fundamental principle: **trust must never be assumed when it cannot be demonstrated**.

The growing use of AI in both code development and cyberattacks further reinforces this need.

As attack cycles accelerate and become increasingly automated, preventive measures alone are no longer sufficient.

Each layer of defence must operate within a framework in which trust is continuously established rather than implicitly granted.

The next chapter presents the conceptual foundations of this approach by explaining how organisational policies can be transformed into demonstrable guarantees.

Chapter 2 — Turning Organisational Policies into Demonstrable Guarantees

2.1 Foundations of the Proposed Architecture

When a business team designs a processing operation, its concern is relatively straightforward: transforming input data into output data.

Whatever the business domain, this processing can generally be summarised in three steps:

1. obtain the data required for processing;
2. apply the business logic;
3. publish the results.

From a business perspective, this representation is sufficient. It describes the purpose of the processing without burdening it with technical or organisational concerns.

Yet the same processing is also subject to numerous requirements owned by other teams within the organisation.

- Data governance needs to ensure that data complies with established contracts, that it is traceable and that its quality is controlled.
- Security teams need to verify the integrity of executed components, control the origin of data, protect secrets and have evidence available to detect any alteration.
- Operations teams require recovery, logging and monitoring mechanisms.
- Architects need to control dependencies between processing operations and facilitate the evolution of the information system.
- Finally, senior management needs a reliable view of the processing being performed and the ability to demonstrate that the organisation’s policies are actually being applied.

All these expectations relate to the same business processing, but each has its own lifecycle, priorities and objectives.

Traditionally, these requirements have mainly been addressed through application-specific development. Developers are expected to incorporate the requirements of each governance domain into their code, progressively resulting in increasingly complex processing components that are difficult to maintain and whose behaviour depends heavily on the practices adopted by each team.

This approach has several limitations:

- the same mechanisms are developed multiple times;
- policies are interpreted differently across projects;
- changing a policy often requires modifications to several applications;
- audits rely heavily on documentary reviews or retrospective controls;
- the lifecycles of the different teams become highly dependent on one another.

Modern organisations, however, require precisely the opposite.

Processing operations need to evolve rapidly, be deployed independently, move between infrastructures, integrate new partners or respond to new regulatory requirements without calling the entire system into question.

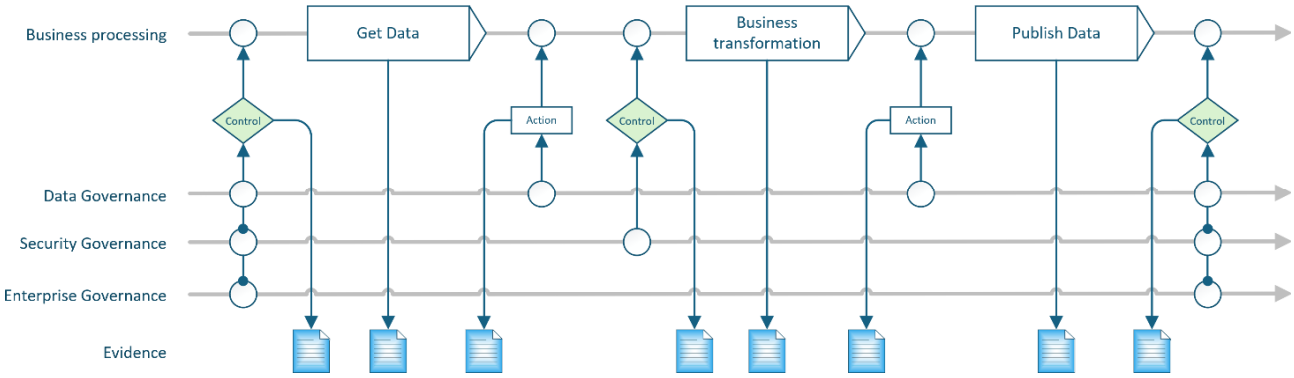
The objective is therefore no longer to embed organisational policies within software, but to provide a mechanism through which they can be applied automatically, independently of business development.

The proposed architecture is based on a simple principle: **each governance domain expresses its objectives in the form of "controls" and/or "actions" that can be executed at specific stages of business processing.**

These controls and actions are injected into the processing flow without modifying the business logic code itself.

Business processing can therefore remain focused exclusively on data transformation, while the injected controls and/or actions address governance requirements by providing the necessary evidence.

This approach is illustrated in the figure below.



Each governance domain retains its autonomy and its own lifecycle.

The policies it defines are translated into reusable technical mechanisms that remain independent of business processing.

These mechanisms may be controls, such as verifying data integrity or validating a Data Contract, or actions, such as signing a result, logging an event or preserving a transactional state.

The same mechanism may simultaneously address the needs of several governance domains.

For example, a data integrity control contributes to:

- **data governance**, by ensuring trust in exchanged information;
- **security**, by detecting any alteration;
- **audit**, by providing verifiable evidence;
- **enterprise strategy**, by strengthening confidence in decisions based on that data.

The value of this approach therefore lies not merely in performing additional controls.

Each control and each action produces **evidence**, which becomes the objective basis for demonstrating that the organisation’s policies have actually been applied.

Governance therefore no longer relies solely on procedures or declarations, but also on verifiable facts.

This separation between business logic, governance mechanisms and the evidence they produce forms the foundation of the approach presented in this document.

The following chapters will show how these mechanisms can progressively establish demonstrable guarantees relating to data, processing and, more broadly, the governance of the information system.

2.2 A Different Interpretation of Zero Trust

The term **Zero Trust** is frequently associated with identity management, network segmentation or continuous authentication.

Although these aspects remain essential, they address only part of the problem.

This document adopts a broader interpretation.

Rather than asking whether an identity or a system should be trusted, it asks a different question:

Can the property required by the organisation be demonstrated at the point when it matters?

Trust is no longer attached to an identity, a piece of software or an infrastructure component.

Instead, it is attached to the demonstration of specific properties.

For example:

- is the input data authentic?
- has the business processing component been validated?
- does the data comply with its contract?
- has the output been signed?
- can the transaction be recovered?
- has the **Business Process Definition** been approved?

Each of these questions can be answered objectively.

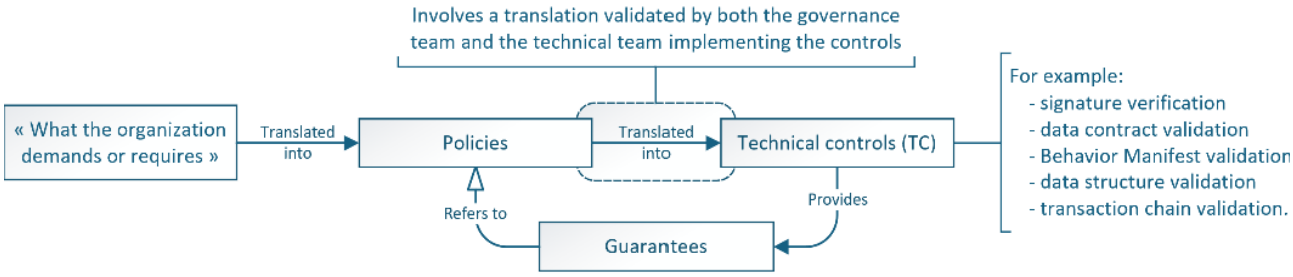
Each answer can be supported by evidence.

Trust then becomes the result of demonstrated guarantees.

2.3 The Trust-Building Model

We have seen that the approach follows a simple but rigorous reasoning model:

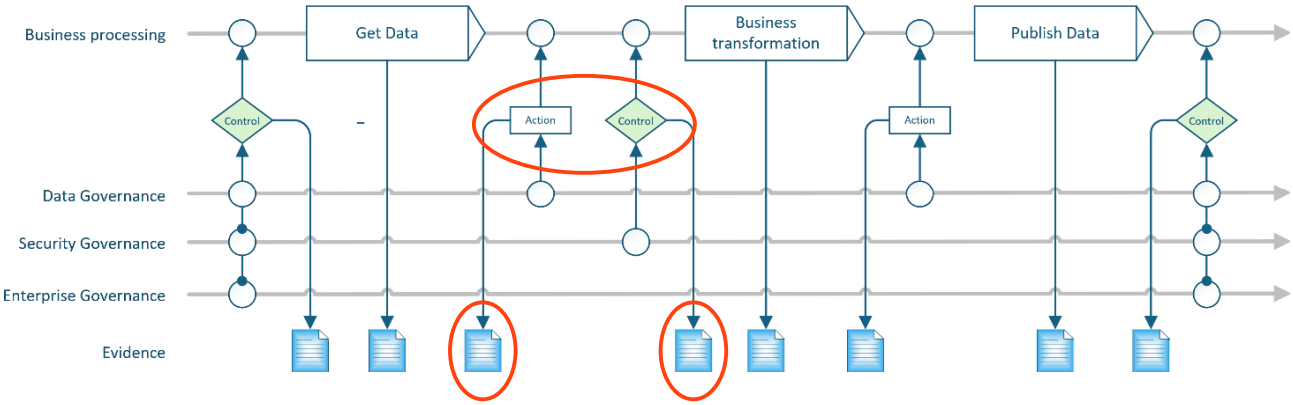
- each organisational requirement is progressively translated into technical actions and/or controls;
- each action and/or control is implemented by a specialised component that produces evidence; this evidence provides guarantees and, ultimately, contributes to trust.



Trust-Building Model – Actionable Governance Lifecycle – V1.0

A **Guarantee Orchestrator** coordinates the chain of actions and controls when a business processing request is received.

Actions and controls provide evidence that enables the Guarantee Orchestrator to decide whether execution may continue.



These sequences form the foundation of the entire architecture.

They establish a direct relationship between the organisation's intent and execution behaviour.

Rather than relying on assumptions, each execution decision is supported by verifiable evidence.

A concise representation of the model is therefore:

Organisational Objective → Policy → Action and/or Control → Evidence → Guarantee → Trust

2.4 Separation of Responsibilities

One of the major principles of the proposed architecture is the separation of responsibilities.

Each stakeholder contributes to the trust model without requiring detailed knowledge of the system as a whole.

For example:

- business teams define business rules and data ownership;
- data governance teams define **Data Contracts**;
- security teams define security policies and their associated controls;
- risk management validates acceptable technical capabilities;
- operations teams define execution policies;
- developers implement the business logic;
- the proposed architecture applies the resulting guarantees.

Each stakeholder signs or approves only those artefacts that fall within their area of responsibility.

The proposed architecture then combines these independent commitments into a coherent execution model.

Trust is therefore not delegated to a single team; it is built through the combination of independently validated guarantees.

2.5 Demonstrable Governance

The preceding sections have shown that:

- policies are translated into actions and/or controls;
- actions and/or controls produce evidence;
- evidence builds guarantees.

Governance therefore no longer relies solely on:

- procedures;
- documentary audits;
- declarations.

It also relies on technical evidence produced automatically during execution.

This approach is called **Demonstrable Governance**.

Any organisation can progressively adopt this approach without replacing its existing software.

The transition can begin with straightforward practices such as:

- formally describing **Data Contracts**;
- digitally signing critical artefacts;
- validating software behaviour before deployment;
- producing verifiable evidence during execution.

Each additional guarantee reduces uncertainty.

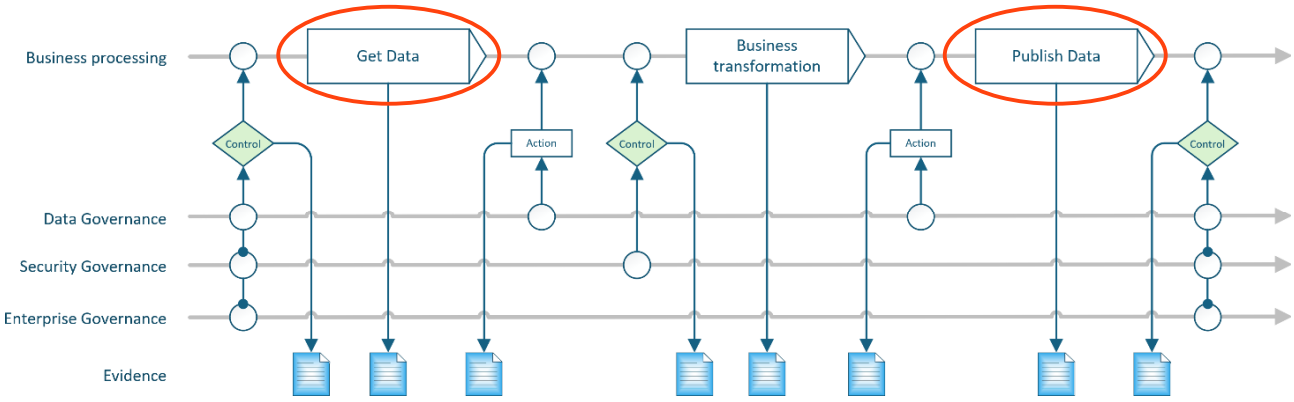
Each new item of evidence strengthens trust.

The objective is therefore not to establish trust once and for all, but to establish it continuously throughout the lifecycle of each business process.

The following chapters describe the guarantees that make this continuous trust possible, beginning with trust in the data itself.

Chapter 3 — Establishing Trust in Data

This chapter examines the actions and controls considered necessary to establish trust in the data used by business processing and in the data it creates.



3.1 Why Trust in Data Matters

Even the most reliable business algorithm can produce trustworthy results only if its input data is itself trustworthy.
This principle is well known, but it is often underestimated.

Organisations generally invest considerable effort in securing software while assuming that the data used by that software can be trusted.

In reality, this assumption can rarely be demonstrated.

Data may originate from databases, file systems, APIs, intermediaries such as message brokers, or partner organisations.

Whatever its origin, business processing must never assume that the data it uses is authentic, complete and structurally valid.

Instead, trust must be established progressively before any business processing begins.

3.2 Verifying Data Integrity

The first property to establish is integrity, which addresses the following question:

Has the data been modified since it was produced?

A simple hash is not sufficient to answer this question.

Although hashing detects modifications, an attacker capable of modifying the data can generally also recalculate the hash.

Integrity therefore cannot rely on hashing alone.

Cryptographic authentication is required.

The recommended approach is therefore to digitally sign the data.

A digital signature provides more information than a hash alone.

It enables the organisation to determine:

- who produced the data;
- when it was signed;
- which cryptographic algorithm was used;
- which signing key was involved;
- whether that key was subsequently revoked or compromised.

The proposed approach verifies these properties before execution whenever a signature is available.

When input data cannot be signed — for example because it originates from external systems — the approach cannot establish cryptographic trust.

In such cases, responsibility is transferred to the business domain, which must determine whether the information received remains acceptable despite the absence of cryptographic guarantees. This acceptability may itself be associated with specific controls.

3.3 Data Contracts

Data governance requires organisations to understand:

- where data comes from;
- where it is consumed;
- who owns it;
- how it evolves;
- which business processes transform it.

For organisations subject to regulations such as the **General Data Protection Regulation (GDPR)**, maintaining this knowledge is no longer optional.

A **Data Contract** therefore serves several purposes simultaneously:

- for developers, it defines the structural guarantees required before execution;
- for business processing, it defines the expected logical representation of the data;
- for governance teams, it documents how information flows through the information system.

These Data Contracts are valid only if they are versioned and signed.

They then also become governance artefacts whose integrity can be continuously verified.

3.4 Structural Validation

Authentic data is not necessarily usable by business processing.

Before execution, the proposed architecture verifies that the data received complies with the structure expected by the business processing component.

This verification is performed using properties defined in the associated **Data Contract**.

A Data Contract formally describes:

- the expected format;
- the data structure;
- mandatory attributes, such as headers;
- data types;
- validation rules;
- optional strictness policies.

Unlike business validation, structural validation remains entirely independent of business logic.

Rather than assessing business validity, structural validation ensures that the component receives data that complies with the agreed contract.

This separation considerably simplifies business implementations while improving reusability and sovereignty.

3.5 Abstracting the Physical Representation of Data

Traditional business software closely couples business logic with technical concerns.

A business function is frequently responsible for:

- locating data;
- authenticating with external systems;
- understanding storage formats;
- parsing files;
- converting character encodings;
- managing serialisation libraries.

These responsibilities have nothing to do with the business logic itself and merely reflect technical constraints.

This coupling reduces reusability and sovereignty.

Any change in storage technology often requires changes to business code, even when the business rules remain unchanged.

The proposed architecture follows a different approach: business processing executes within a **logical rather than physical data environment**.

It never directly manipulates CSV files, SQL result sets, JSON documents or XML documents.

Instead, the proposed architecture converts physical representations into standardised logical abstractions.

The resulting objects are of types such as:

- `iTableData` for a tabular representation of data;
- `iTreeData` for a hierarchical representation of data;
- `iBinaryData` for binary data such as an image;
- `iTextData` for data represented as unstructured text;
- `iStreamData` for data streams.

Business processing therefore becomes independent of storage technologies and serialisation formats, making it easier to adapt to strategic changes — and consequently improving sovereignty.

Its responsibility is limited to implementing business rules.

3.5.1 Normalisation and Denormalisation

Converting physical data into logical abstractions requires more than simple structural validation.

The proposed architecture must also normalise the information.

Normalisation may include:

- character encoding conversion;
- parsing numbers independently of locale;
- standardised date and time representations;
- logical data typing;
- canonical representations.

This normalisation ensures that each business processing component receives consistent data, regardless of the technologies used to produce it.

Conversely, once processing has been completed, the proposed architecture performs the reverse operation.

Logical data is converted back into its physical representations by applying the output Data Contracts before publication.

3.5.2 Data Transport Abstraction

Knowing what the data should look like is only part of the problem.

The proposed architecture must also know how to obtain and publish it.

This responsibility belongs to the **Data Transport** abstraction.

Transport implementations provide storage-specific read and write operations for file systems, databases, APIs, cloud storage and intermediaries such as message brokers.

Business processing is entirely unaware of these implementation details.

This separation improves portability while preventing sensitive credentials from being exposed to business code.

3.6 Additional Data

Business processing may also require:

- reference data;
- configuration parameters;
- secrets.

3.6.1 Reference Data

Reference data is accessed through dedicated **Resource Managers**.

This data may change between executions and is generally read-only.

Business processing is responsible for interpreting any functional consequences arising from changes to this data.

3.6.2 Configuration Parameters

In the proposed architecture, configuration parameters are stored in the **Business Process Definition**.

Because the definition itself is signed, these parameters inherit the same integrity guarantees.

3.6.3 Secrets

Secrets follow a different model.

In the proposed architecture, business processing never directly stores passwords or API keys.

Instead, it receives secret identifiers.

The proposed architecture resolves these identifiers through an `iSecretProvider`, retrieving the actual credentials from a secure storage mechanism such as a digital vault.

Wherever possible, transports consume these secrets directly, thereby minimising their exposure within business code, reducing their lifetime in memory and promoting independence and reusability.

3.7 Data Classification

Data governance increasingly relies on classification models based on enterprise taxonomies.

Each business process and each exchanged dataset may therefore be associated with one or more classifications describing properties such as:

- confidentiality;
- sensitivity;
- regulatory scope;
- retention requirements;

- business criticality.

These classifications form part of the **Business Process Definition** rather than the business code itself.

Since the proposed architecture already analyses each dataset during normalisation and denormalisation, it may also optionally perform automatic classification.

Regular expressions, dictionaries or specialised classifiers may identify characteristics such as personal information, payment data, healthcare identifiers or other regulated content.

The resulting classifications become additional evidence recorded during execution.

They can subsequently be used by governance platforms, data catalogues or compliance tools without requiring the business processing component itself to implement classification logic.

3.8 Data Lifecycle

The proposed architecture verifies data integrity when the data is consumed.

This verification ensures that processing relies on trustworthy information.

However, it does not remove the need for ongoing monitoring while data remains stored on media that may be modified or altered.

Such alteration may result from:

- disk corruption;
- a backup error;
- an incorrect restoration;
- partial deletion;
- malicious modification;
- an operational error.

Data integrity is not a property that is acquired once when data is produced.

It must be preserved throughout the data lifecycle, from creation through to archival or destruction.

The proposed architecture can perform periodic integrity checks through dedicated processing in order to identify alteration as early as possible and respond accordingly.

A distinction must therefore be made between **controls associated with the execution of processing** and **long-term information asset controls**, which are performed independently of any business execution.

The former establish the trust required for an immediate decision; the latter help preserve trust in the organisation’s information assets over time.

Data governance policies may also differentiate the controls to be performed and their frequency according to the nature of the data and whether it resides on mutable or immutable storage.

For example, they may specify:

- **Mutable storage:** periodic checks recommended;
- **Immutable storage:** verification only when data is retrieved.

3.9 Chapter Conclusion

Establishing trust in business processing begins with establishing trust in the data.

The proposed architecture therefore separates concerns traditionally associated with:

- integrity validation;
- structural validation;
- normalisation and denormalisation of physical representations;
- transport abstraction;
- secret protection;
- classification.

Business processing no longer needs to concern itself with where data comes from or how it is represented.

It simply receives data for which the required guarantees have already been established.

This separation not only improves the reusability and portability of business processing, but also transforms data governance into an **operational capability rather than a purely documentary exercise**.

The next chapter extends the same philosophy by establishing trust in business processing itself.

Chapter 4 — Establishing Trust in Execution

Trust in data is a necessary condition for reliable processing, but it is not sufficient. Data whose integrity, origin and compliance have been established may nevertheless be processed by a component other than the one expected, by an unauthorised version, or under conditions that no longer comply with the policies defined by the organisation.

Establishing trust in execution therefore means answering another question:

How can we demonstrate that processing was performed by the expected components, under the intended conditions and in accordance with the required guarantees?

This question is particularly important in information systems composed of numerous services, libraries and dependencies whose lifecycles evolve independently.

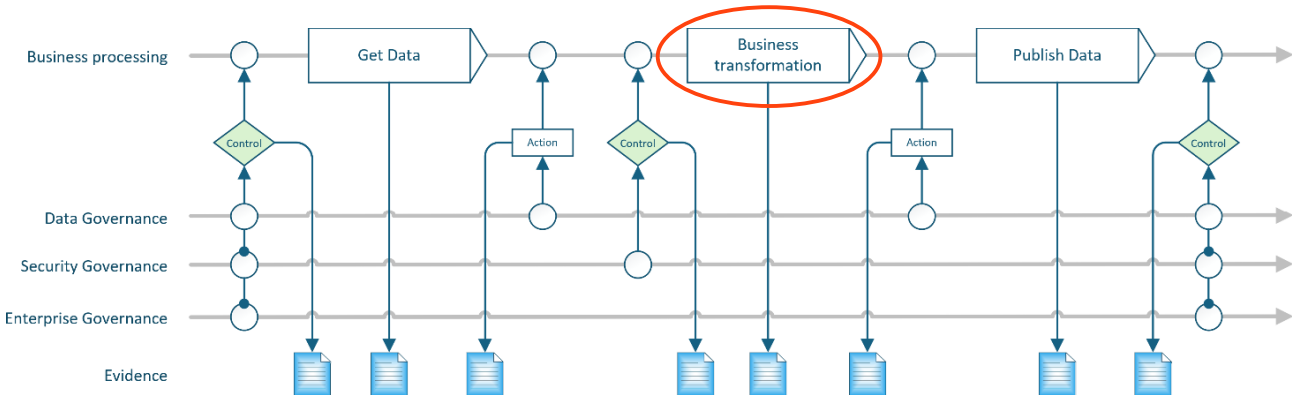
Authorising processing can no longer rely solely on the fact that a component has been deployed or belongs to an environment considered trustworthy. Trust must be established when execution is requested and maintained throughout the stages that make up that execution.

This requires precise knowledge of what is about to be executed, verification of its identity and integrity, and assurance that its expected behaviour is compatible with the applicable policies. The conditions of execution must then be controlled, its various stages monitored, and the elements required to demonstrate retrospectively what actually happened must be retained.

Execution therefore becomes an object of governance in its own right. Its authorisation no longer implicitly results from trust placed in a server, an application or a team; it results from verifiable guarantees established before, during and after processing.

This approach also leads to an essential rule: when an indispensable guarantee cannot be established, execution must not be considered implicitly authorised. Depending on the policy defined by the organisation, interrupting execution may then be preferable to continuing processing for which the conditions of trust can no longer be demonstrated.

This chapter presents the mechanisms used to build trust in execution: identifying and controlling components, controlling their behaviour and dependencies, transactional management, traceability of state changes, and production of the evidence required to reconstruct and verify execution.



4.1 From Code to the Behaviour Manifest

Several approaches are currently used to establish trust in software execution.

Organisations generally rely on:

- static application security testing (**SAST**);
- dynamic application security testing (**DAST**);
- manual or AI-assisted code reviews;
- containerisation and sandboxing technologies.

Each of these approaches addresses part of the problem.

Static and dynamic analyses attempt to identify vulnerabilities.

Note: Static and dynamic analyses are generally performed only when changes occur and/or when an application is deployed, rather than during its execution.

Containers attempt to isolate the consequences of those vulnerabilities.

4.1.1 A Behaviour-Based Approach

The proposed architecture follows a different philosophy.

Its objective is not to prove that business code is free from defects.

Nor does it seek to replace SAST or DAST solutions.

Instead, **it aims to demonstrate that the code being executed exhibits only behaviours that have previously been identified, validated and accepted.**

4.1.2 Generating the Behaviour Manifest

Before publication, each business processing component is analysed to identify its technical capabilities.

The analysis focuses on observable behaviour rather than on business logic.

The analysis includes:

- referenced libraries;
- namespaces used;
- use of reflection;
- native operating system calls (P/Invoke);
- process creation;
- network access;
- file-system access;
- HTTP calls;
- any other capability considered relevant by the organisation.

The result of this analysis is a **Behaviour Manifest**.

The resulting Behaviour Manifest characterises the technical capabilities of the component; it is a descriptive artefact, not a security assessment.

Once digitally signed and referenced by the **Business Process Definition**, it becomes the authoritative description of the expected execution behaviour.

4.1.3 Runtime Verification

Whenever the proposed architecture needs to load a business processing component, it performs the same behavioural analysis again.

It then compares the newly generated manifest with the published Behaviour Manifest.

Loading and execution are authorised only if the two manifests are identical.

Trust is therefore never placed solely in a declaration; it is established through agreement between an independent analysis and a previously validated and signed manifest.

Any discrepancy immediately prevents the business processing component from being loaded, in accordance with the principle described in Section 2.2, **“A Different Interpretation of Zero Trust”**.

4.1.4 Supporting Risk Assessment

The Behaviour Manifest can also provide valuable support to the team responsible for risk management.

Two versions of the same business processing component may implement different business logic while relying on exactly the same technical capabilities.

Conversely, apparently minor code changes may introduce entirely new technical behaviours.

The Behaviour Manifest enables organisations to distinguish objectively between these situations.

If the manifest remains unchanged, the technical risk perimeter has not changed.

Depending on organisational policy, deployment approval may therefore follow a simplified validation process.

If, on the other hand, the manifest changes, the component’s technical capabilities have changed.

This represents a change in the trust perimeter and justifies a new risk assessment before deployment.

Automatically generating a report describing behavioural differences between successive versions may allow risk management teams and **Change Advisory Boards (CABs)** to focus their reviews exclusively on significant technical changes rather than inspecting the entire source code.

4.1.5 Benefits for Stakeholders

The Behaviour Manifest serves three complementary purposes:

- **For developers**, it provides immediate feedback on the technical capabilities used by their implementation. They commit to the behaviour of the business processing component.
- **For the proposed architecture**, it provides a trusted behavioural reference used to validate execution.
- **For governance teams**, it becomes an objective artefact supporting change management and risk analysis.

4.2 Resource Managers

Executing business logic requires more than simply reading and writing data.

Business processing takes place in distributed environments where failures are inevitable.

Network interruptions, concurrent execution, unavailable storage systems, unexpected process termination and partial publication must all be anticipated.

For this reason, the proposed architecture relies on **Resource Managers**, which themselves rely on **Data Transports** responsible for the specifics of the underlying storage environments.

A Resource Manager is responsible for maintaining consistency between the proposed architecture and an associated storage technology.

Its responsibilities include:

- discovering available work items;
- reserving resources to prevent collisions during concurrent processing;
- reading input data;
- writing output data;
- participating in transactional coordination of input and output resources, within the limits of the guarantees provided by the systems concerned;
- restoring previous states when recovery is required;
- releasing resources after successful completion;
- archiving processed items.

The proposed architecture delegates all storage-specific operations to Resource Managers.

Conversely, Resource Managers never make execution decisions.

Their role is limited to implementing the operations requested by the **Guarantee Orchestrator**, while remaining entirely independent of business logic.

Once data has been read, the Guarantee Orchestrator progressively establishes trust by verifying signatures, validating contracts and normalising information before invoking the business processing component.

After execution, the same principle applies in reverse.

Output contracts are validated, data is converted back into its physical representation, signed where required, and finally delegated to the appropriate Resource Managers for publication.

4.3 The Guarantee Orchestrator

Before examining the execution model itself, it is useful to define the terminology used throughout the remainder of this document.

- A **Guarantee** is a property that the proposed architecture undertakes to demonstrate.
- A **Guarantee Controller** is responsible for verifying a specific guarantee and producing the associated evidence.
- The **Guarantee Orchestrator** coordinates these controllers and determines whether execution may proceed.
- Finally, the **Evidence Journal** records every demonstrated guarantee, enabling execution to be reconstructed and audited at any time.

The proposed architecture does not execute business processing simply because it has been requested.

It executes business processing only when the required guarantees have been successfully established.

The Guarantee Orchestrator therefore maintains the complete execution state.

It is the only component with an overall understanding of:

- the current execution state;
- guarantees already demonstrated;
- guarantees still required;
- possible recovery strategies.

Once a stage has been validated, it initiates the next stage.

This may involve acquiring data through a Resource Manager, performing a control through a Guarantee Controller, invoking the business processing component, or invoking a data converter.

Although the Guarantee Orchestrator does not know how a Resource Manager accesses data, which control a Guarantee Controller performs, or what the business processing component or data converter actually does, it knows how to invoke them and receive their results through established implementation contracts.

Its responsibility is exclusively orchestration.

Execution progresses through a deterministic state machine.

Each transition requires:

- preconditions;
- evidence;
- postconditions;
- recovery rules.

Only once all required guarantees have been demonstrated does the Guarantee Orchestrator authorise the next transition.

For example:

Read completed (+ Guarantee persisted)
▼
Signature verified (+ Guarantee persisted)
▼
Contract verified (+ Guarantee persisted)
▼
Normalisation completed (+ Guarantee persisted)
▼
Business processing (+ Guarantee persisted)
▼
(…)

The execution flow therefore never progresses merely because an operation has completed. It progresses because the guarantees required for that operation have been demonstrated and persisted.

4.4 Evidence Journal

The preceding sections introduced the components responsible for establishing trust.

We have seen that the Guarantee Orchestrator coordinates a state machine in which each transition corresponds to a newly established and persisted guarantee.

Each successful transition is immediately retained as a signed **evidence record**, cryptographically linked to the preceding record.

Together, these records form the **Evidence Journal** for a business execution.

The Evidence Journal serves two complementary purposes:

- it provides a verifiable history explaining **what happened** and **why the Guarantee Orchestrator considered it safe to continue**;
- it enables complete reconstruction of an interrupted execution, allowing the processing context to be resumed or restored without relying on assumptions.

Rather than depending on process memory, execution continuity depends entirely on persisted evidence.

4.4.1 Nominal Execution

Under ideal conditions, execution progresses according to the following sequence:

Step	Action
01	**Discover (**identify tasks to be executed)

02	**Lock input data (**reserve the input data associated with the task)
03	**Read input data** (retrieve the content associated with the task)
04	**Verify input data** (verify signature and structure)
05	**Normalise input data** (convert content into a logical format and validate values)
06	**Prepare** (invoke the business processing component's `Prepare` action)
07	**Verify output data** (verify output structure)
08	**Denormalise output data** (convert outputs into their physical format and validate values)
09	**Write output data** (write it to final storage while keeping it locked)
10	**Sign output data**
11	**Business commit** (invoke the business processing component's `Commit` action)
12	**Release** (release output-data locks)
13	**Archive** the task and the Evidence Journal associated with the execution

Each transition follows exactly the same pattern.

The required verification is performed by a specialised **Guarantee Controller**.

The resulting evidence is retained by the evidence recorder within the Evidence Journal.

The Guarantee Orchestrator validates that the transition requirements have been satisfied.

Only then may execution continue.

The Guarantee Orchestrator therefore never progresses because an operation merely appears to have succeeded; it progresses only after the corresponding guarantee has been formally demonstrated and persisted.

4.4.2 Error Handling

Not all failures have the same meaning.

The proposed architecture distinguishes three categories of error because they require fundamentally different responses.

Error type	Meaning and examples	Behaviour
Technical	A technical error indicates that execution is temporarily unable to continue. Examples include unavailable storage, network failures, infrastructure failures and temporary resource conflicts.	These errors do not invalidate the business request itself. Execution therefore applies the retry policy defined in the Business Process Definition. Input resources remain reserved while retries are performed in order to prevent concurrent execution. If the retry strategy is exhausted, execution is transferred to manual handling.
Security	Security errors correspond to failures to establish mandatory guarantees. Examples include invalid digital signatures, corrupted artefacts, a modified application component, manifest behavioural	Whenever such a situation occurs, execution is aborted and the request is transferred for security investigation. Unlike technical failures, security errors deliberately prevent automatic restart until the root cause has been understood. This reflects one of the architecture's fundamental principles:

	inconsistencies or invalid execution evidence.	**“Whenever trust cannot be demonstrated, execution must stop.”**
Business	Business errors indicate that processing cannot continue because business requirements have not been satisfied. Examples include missing reference data, violation of business rules or inconsistent business information.	Execution cannot automatically resolve such a situation. It is suspended and transferred to manual handling. Manual handling does not necessarily imply cancellation: correcting the underlying business information may allow the request to be processed successfully.

Note: In this context, an artefact is a product or document generated during an execution or required for an execution.

4.4.3 Recovery

Unexpected process termination is not, in itself, considered a business failure.

It simply interrupts the demonstration of guarantees.

When the Guarantee Orchestrator restarts, it analyses the Evidence Journal in order to reconstruct the **Execution Context**.

Recovery never attempts to infer what might have happened.

Instead, it resumes from the last demonstrated state.

Depending on the recorded transition, several situations may arise:

- if execution stopped before any external side effect occurred, execution resumes from the pre-execution state;
- if resources remain reserved, the appropriate Resource Managers first restore any outstanding reservations;
- if temporary outputs were written but never validated:
    - if the business processing was not committed, they are removed in accordance with the recovery capabilities provided by the corresponding Resource Managers;
    - if the business processing was committed, their publication is completed by the corresponding Resource Managers.

Restarting therefore consists of **rebuilding trust rather than blindly replaying operations**.

The Evidence Journal remains the sole authoritative source describing the state of execution.

4.4.4 Business Processing Models

Recovery capabilities also depend on the behaviour of the business processing component itself.

The implementation contract for a business processing component includes three distinct actions (**`Prepare`**, **`Commit`** and **`Rollback)`** which allow the component to participate in the distributed transaction encompassing the entire execution.

This transaction aims to ensure that the execution as a whole is either completed correctly or cancelled without impact on the data.

Depending on its behaviour during `Prepare`, `Commit` and `Rollback`, business processing can be classified into the following four implementation models:

Model	Operations			Comment
	1. Prepare	2. Commit	3. Rollback	
Pure function	Performs calculation only. Produces no external side effects.	Applies the results.	Does nothing.	Execution can safely be replayed whenever necessary. This is the preferred implementation model.
Compensatable transaction	Reserves external resources.	Commits those reservations.	Releases the resources.	Many modern APIs naturally follow this model. As long as reservations remain valid, interrupted executions can safely resume.
Idempotent transaction	Already produces observable effects.	Commits the effects.	Reverses the effects.	If `Prepare` produces effects, repeating the operation produces the same final result. Execution can therefore be replayed without introducing inconsistencies.
Irreversible transaction	Immediately performs an irreversible external action.	Does nothing.	Does nothing.	In this situation, it is not possible to demonstrate independently that re-execution is safe. The request is therefore transferred to manual handling. This is not a limitation of the proposed architecture but a direct consequence of its philosophy: **when safety cannot be demonstrated, the architecture deliberately refuses to make assumptions.**

4.4.5 Limitations

The Guarantee Orchestrator never directly manipulates business data.

Instead, it coordinates **Guarantee Controllers**, **Resource Managers**, a data converter and the business processing component.

Each participant takes part in a distributed transaction through precisely defined implementation contracts.

As we have seen, regardless of the underlying storage technology, each Resource Manager must be capable of:

- discovering available tasks;
- reserving a task;

- releasing reservations;
- marking success;
- marking failure;
- archiving processed tasks;
- participating in recovery.

This common contract enables heterogeneous storage technologies to participate in a single distributed execution model while remaining entirely independent of one another.

One important limitation must nevertheless be recognised: **the Guarantee Orchestrator cannot invent transactional guarantees that external systems do not provide.**

The execution environment can coordinate business processing only within the transactional guarantees provided by external systems.

Therefore, if a business processing component performs several external commits, those operations must themselves be protected by an appropriate transactional mechanism.

Within a single relational database, this is generally straightforward.

Across several databases — or when interacting with external web services — it can become considerably more complex.

When no distributed transaction mechanism exists, the preferred solution is to decompose the processing into several smaller business processing components.

Each component can then be executed independently within its own guarantee boundary.

Rather than attempting to coordinate irreversible operations, the proposed architecture coordinates a sequence of demonstrable guarantees.

This decomposition significantly improves resilience while reducing the operational impact of failures.

4.5 Chapter Conclusion

Trust in execution cannot be reduced to verifying a digital signature or validating a software component.

It requires every execution transition to be supported by demonstrable guarantees.

The proposed architecture therefore does not merely orchestrate business processing; **it orchestrates the construction of trust**.

Each guarantee builds upon the previous one.

Each decision is supported by verifiable evidence.

Each interruption can be analysed and recovered using that evidence alone.

Ultimately, execution itself becomes the consequence of an uninterrupted chain of demonstrated guarantees.

Chapter 5 — Establishing Trust through Declarative Process Descriptions

5.1 Why Describe a Process?

A process description is not simply a configuration artefact.
It is a declarative representation of business processing whose primary purpose is to enable the proposed architecture to demonstrate the guarantees associated with an execution.

Rather than describing only **what** must be executed, it describes the **trust model** governing that execution by specifying:

- Data Contracts;
- transport mechanisms;
- secrets;
- guarantees;
- required digital signatures;
- the Behaviour Manifest;
- responsibilities;
- publication policies;
- recovery policies;
- retention policies.

The process description therefore becomes the foundation upon which trust can be established, verified and ultimately demonstrated.

In most execution engines, a process description answers two fundamental questions:

- What must be executed?
- Where can the required data be found?

Although this information is sufficient to automate business processing, it is not sufficient to demonstrate that execution complies with the organisation’s governance policies.

In the proposed architecture, the **Business Process Definition** serves a broader purpose.

It describes not only the business process itself, but also identifies the guarantees that must be established before, during and after execution.

The objective is therefore not simply to automate execution, but to make that execution **demonstrably trustworthy.**

5.2 A Guarantee-Driven Description

The **Business Process Definition** contains all the information required by the actors participating in an execution so that the required guarantees can be established.

These actors include Resource Managers, Guarantee Controllers and the business processing component.

The proposed architecture provides generic execution capabilities, but it has no intrinsic knowledge of business policies, security rules, regulatory requirements or organisational governance.

All such decisions originate exclusively from declarative descriptions.

Rather than embedding governance logic within the Execution Engine, the proposed architecture interprets the guarantees defined by the Business Process Definition.

Governance policies can therefore evolve freely.

5.3 Splitting the Description into Multiple Documents

Not every aspect of a Business Process Definition evolves at the same pace.

For example:

- the identity of a process changes rarely;
- runtime parameters evolve as the infrastructure evolves;
- Behaviour Manifests evolve whenever the behaviour or scope of business processing changes;
- data exchanges evolve as Data Contracts and business processing evolve.

Grouping all these concerns into a single document would artificially force them to share the same lifecycle, even though each belongs to a different governance domain.

By separating them into independent artefacts, each discipline can evolve according to its own governance processes without affecting the others.

This independence is essential to preserving organisational agility while maintaining trust.

Each fragment of the Business Process Definition has its own:

- owner;
- digital signature;
- lifecycle;
- validation process.

The proposed architecture then assembles these independently governed artefacts into a trusted execution environment.

5.4 Digital Signatures

Each artefact participating in the execution model is digitally signed using asymmetric cryptography.

Each governance authority is responsible for protecting its own private signing key.

In practice, signatures are generally applied as part of the organisation’s deployment or publication process.

Different artefacts may be signed by different authorities, according to the organisation’s governance model.

When the proposed architecture executes, it first verifies the signature of each artefact using the trusted public key associated with its governance domain.

Typical trust domains include:

- Business;
- Operation;
- Development;
- Security;
- Platform;
- Data;
- Enterprise.

A typical mapping might be:

Artefact	Primary responsibility	Trust domain
Identity	Business team	Business
Runtime	Operations	Operation
Business Processing	Development team	Development
Behaviour Manifest	Development team + Security	Security
Data Exchanges	Data Governance	Data

A digital signature serves a purpose far beyond protecting file integrity.

It represents a verifiable commitment made by the authority responsible for that artefact.

The proposed architecture therefore does not merely verify that an artefact has not been modified; it also verifies which domain assumes responsibility for its content.

Trust consequently becomes an organisational property, not merely a cryptographic one.

5.5 Implications of Artefact Publication

When the proposed architecture executes, it always loads the latest version of the artefacts and verifies the integrity of the published set before any transaction begins.

This integrity check ensures that the publication process completed successfully and that no artefact was modified after publication.

During publication, the platform seals the complete version by digitally signing the collection of artefacts with the platform’s private key.

This platform signature certifies that the published version is complete, consistent with the deployment process and ready for execution.

Once execution has started, the proposed architecture continues to use the same immutable set of artefacts throughout the lifetime of its **Execution Context**.

If a new version of an artefact is published while transactions are still being processed, the proposed architecture detects the change, completes all transactions currently in progress and then shuts down cleanly.

When restarted, it creates a new Execution Context from the updated set of artefacts.

This behaviour ensures that every transaction is executed within a stable and fully verifiable execution environment.

5.6 The State-Machine Execution Engine

The proposed architecture does not dictate how an organisation governs its business processes.

Each organisation defines its own governance policies, control mechanisms and operational guarantees.

These evolve continuously in response to new regulations, emerging threats, internal policies and changing business objectives.

If these governance rules were embedded directly within the Execution Engine, every policy change would require modifications to the engine itself.

The Execution Engine would become tightly coupled to organisational governance and would lose its ability to serve as a generic execution environment.

The proposed architecture takes a different approach.

It contains no embedded governance policy.

Instead, it interprets a declarative state machine describing the transaction lifecycle, the guarantees to be established and the controls to be performed.

This description is called the **Guarantee Pipeline Definition**.

Because governance is expressed declaratively rather than implemented programmatically, policies can evolve independently of the Execution Engine.

The Execution Engine therefore acts as a **policy interpreter**, rather than a **policy implementation**.

5.6.1 Defining the State Machine

At start-up, the proposed architecture loads both the **Business Process Definition** and the **Guarantee Pipeline Definition**.

Although these two descriptions are interpreted together, they serve fundamentally different purposes.

The Business Process Definition describes **what the business process does**.

The Guarantee Pipeline Definition describes **how that business process must be governed**.

The Guarantee Pipeline Definition specifies, among other things:

- the stages of the transaction lifecycle;
- the Resource Managers to invoke;
- the Guarantee Controllers to execute;
- the execution sequence;
- the Evidence that must be produced.

This definition belongs to the organisation’s governance team, which translates governance policies into an executable chain of actions and controls.

Its lifecycle is therefore independent from that of the business process itself.

This separation allows governance policies to evolve without modifying business logic and, conversely, allows business processes to evolve without requiring changes to governance policies.

5.6.2 A Chain Rather than a Graph

Unlike a traditional workflow engine, the state machine does not describe business logic.

Instead, it describes the technical governance activities accompanying each transaction.

The execution model is intentionally linear.

A transaction progresses through a deterministic sequence of operations involving:

- Guarantee Controllers;
- Resource Managers;
- business processing;
- data transformers, where necessary.

The absence of a complex execution graph is a deliberate architectural choice.

A linear execution model is considerably easier to understand, easier to audit and significantly simpler to recover after a failure.

More importantly, it ensures that every transaction follows the same fully deterministic and fully traceable execution path.

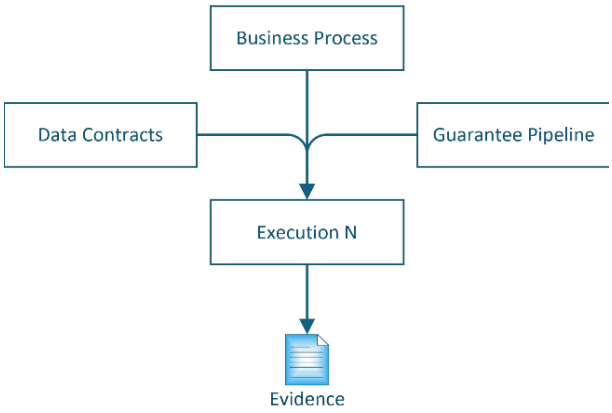
Business decisions remain entirely the responsibility of the business component.

The state machine is responsible only for the governance activities required to establish and demonstrate trust throughout the transaction lifecycle.

5.6.3 At Least Four Independent Lifecycles

Each transaction represents the convergence of at least the following four independent lifecycles:

- the Business Process Definition;
- the Guarantee Pipeline Definition;
- the Data Contracts;
- the current business request being processed.



Each evolves independently.

The same business process can therefore be executed under successive versions of a governance policy without requiring any modification to the business process itself.

Conversely, several versions of a business process may be governed by the same policy.

The **Evidence Journal** links these dimensions together.

For each transaction, it records:

- the version of the Business Process Definition;
- the version of the Guarantee Pipeline Definition;
- the version of the Data Contracts;
- the associated Execution Context;
- the Evidence produced throughout execution.

Years after execution, it therefore remains possible to reconstruct the exact environment in which a business decision was made, provided of course that the relevant journals have been retained.

This capability provides a level of traceability that is difficult to achieve when governance policies are embedded directly in application code.

5.7 Execution Quality Assurance

The proposed architecture does not create trust.

Nor does it define governance policies, validate business decisions or guarantee the correctness of an implementation.

Its responsibility is more fundamental.

It verifies that each actor involved in executing a transaction has fulfilled the commitments falling within its own area of responsibility.

Trust therefore emerges from the convergence of independently governed artefacts rather than from assumptions embedded within the Execution Engine.

5.7.1 Independent Responsibilities

Modern enterprise systems are developed and operated by several specialised teams.

- Business experts define business processes.
- Development teams implement business processing.
- Security teams authorise software behaviour.
- Data governance teams publish Data Contracts.
- Operations teams define execution environments.
- Platform teams deploy instances of the proposed architecture.

Each discipline has its own body of knowledge and applies its own governance processes.

Attempting to centralise these responsibilities within a single artefact or a single approval process would not only create organisational bottlenecks, but would also weaken accountability.

Instead, each discipline publishes its own trust artefact.

Each artefact follows its own lifecycle, validation process, publication flow and digital signature.

Ownership remains distributed, while trust becomes composable.

5.7.2 Creating a Trusted Execution Context

Before a transaction begins, the Execution Engine assembles all published artefacts into a single immutable **Execution Context**.

This context represents the complete set of assumptions under which the transaction will be executed.

Before accepting the context, the Execution Engine verifies that:

- every required artefact is present;
- every digital signature is valid;
- every artefact originates from a trusted authority;
- all declared versions are mutually compatible;
- all dependencies can be resolved;
- the assembled context is internally consistent.

Only once these verifications have succeeded does execution begin.

If any verification fails, the transaction is rejected before any business processing takes place.

This principle ensures that trust is established **before** execution rather than inferred afterwards.

5.7.3 Execution Quality Assurance

The Execution Engine performs what may be described as **Execution Quality Assurance**.

Unlike traditional Quality Assurance, which evaluates software before deployment, Execution Quality Assurance evaluates the conditions under which each individual transaction is executed.

Its purpose is not to determine whether a business decision is correct.

Nor does it attempt to validate the quality of software design, data modelling or governance policies.

Those responsibilities remain with the disciplines that produced the corresponding artefacts.

Instead, the Execution Engine verifies that the transaction executes within an environment that satisfies every declared commitment.

Execution Quality Assurance therefore answers questions such as:

- was the approved version of the business processing component executed?
- were the applicable Data Contracts available?
- was the required governance pipeline applied?
- were all mandatory controls completed successfully?
- was all required Evidence produced?

If the answer to any of these questions is no, the transaction cannot be considered fully trustworthy.

Execution Quality Assurance certifies that the transaction was executed under verified and demonstrable conditions.

5.7.4 An Architectural Principle

This approach reflects a broader architectural principle.

Enterprise trust cannot be delegated to a single component.

It must emerge from independently governed and objectively verifiable commitments.

The proposed architecture is therefore not the source of trust.

It is the mechanism that assembles trusted artefacts, verifies their integrity and compatibility, executes their declarative descriptions and produces the Evidence required to demonstrate that every declared commitment has been fulfilled.

Trust is not built into the proposed architecture; it is established through verifiable commitments and demonstrated through execution.

5.8 Beyond Execution: The Value of Declarative Descriptions

Declarative descriptions are often perceived simply as configuration artefacts consumed by an Execution Engine.

In reality, they represent something far more valuable because, by formally describing both business intent and operational governance, they become reusable information assets extending well beyond execution.

The Execution Engine of the proposed architecture is merely their first consumer.

The same descriptions can also support architecture, governance, security, compliance, operations and business analysis.

Documentation is no longer produced separately from the system.

It becomes a natural by-product of the descriptions required for execution.

5.8.1 The Business Process Definition as an Enterprise Asset

The Business Process Definition provides a structured and authoritative representation of business intent.

Because it is both executable and declarative, it becomes a trusted source of information for a broad range of stakeholders.

- Architects can analyse dependencies between business capabilities and technical components.
- Operations teams can understand how business activities are executed without inspecting application code.
- Data governance teams can identify which information is consumed, produced or referenced by each process.
- Security teams can determine which business processing components participate in critical operations.

Impact analysis becomes considerably simpler.

When business processing, a Data Contract or an external dependency changes, every affected business process can be identified directly from its declarative description.

The Business Process Definition therefore serves not only as an execution artefact, but also as a living and continuously maintained source of architectural knowledge.

Unlike traditional documentation, it remains synchronised with the deployed system because it is required for execution itself.

5.8.2 The Guarantee Pipeline Definition as a Governance Asset

The Guarantee Pipeline Definition provides the same level of visibility for operational governance.

Rather than documenting governance policies in narrative documents, the organisation publishes an executable description of the controls governing transaction execution.

This makes it possible to analyse governance with the same rigour as software architecture.

For example, organisations can:

- visualise the complete chain of controls applied to each transaction;
- compare governance policies across versions;
- identify the impact of introducing or removing a control;
- verify compliance with internal or regulatory requirements;
- detect missing or inconsistent governance rules;
- simulate policy evolution before deployment.

Because governance is expressed declaratively, it becomes inspectable, analysable and versionable.

Operational governance evolves from documentation into an enterprise asset that can be continuously managed and improved.

5.9 From Execution to Enterprise Knowledge

As declarative descriptions accumulate, they naturally form a body of interconnected knowledge.

Execution Contexts link all these artefacts into immutable transaction environments.

Evidence links each executed transaction to the exact versions of the artefacts that governed it.

These relationships already exist within the proposed architecture.

Representing them explicitly transforms a collection of technical artefacts into an **Enterprise Knowledge Graph**.

Such a graph enables organisations to answer questions that would traditionally require extensive manual investigation.

For example:

- which business processes depend on a particular Data Contract?
- which transactions were executed under a specific governance policy?
- which business processing components were authorised by a given Behaviour Manifest?
- which governance rules applied to a particular business decision?
- which deployed artefacts would be affected by a proposed change?

Because each relationship originates from trusted declarative descriptions, the resulting knowledge graph remains objective, traceable and continuously synchronised with the operational platform.

Note: The graph can be objective only relative to the completeness and correctness of what has been declared and observed. An undeclared or unobservable external dependency may therefore be absent.

The proposed architecture does not explicitly generate this knowledge.

It simply interprets declarative descriptions whose relationships naturally form an interconnected representation of the enterprise.

The **Enterprise Knowledge Graph** is therefore not a separate product.

It is an emergent property of a declarative architecture based on trusted descriptions.

5.10 Chapter Conclusion

Traditional execution platforms focus primarily on automation.

The architecture proposed in this white paper pursues a broader objective.

By separating business intent from operational governance and expressing both through independently governed declarative descriptions, the proposed architecture establishes a **trusted execution environment** rather than merely an execution engine.

Each participating discipline contributes its own trusted artefacts.

The proposed architecture assembles them, verifies their integrity and compatibility, interprets their intent and records the Evidence required to demonstrate that every declared commitment has been fulfilled.

Trust is therefore not embedded within the proposed architecture; it results from verifiable commitments established by independent authorities.

The result is an execution environment that not only automates business processes, but also provides demonstrable trust, complete traceability and a continuously evolving body of organisational knowledge.

In this model, declarative descriptions become much more than execution artefacts.

They become the foundation upon which trustworthy enterprise execution is built and enable the emergence of higher-level capabilities such as the **Enterprise Knowledge Graph**.

Chapter 6 — Adopting Demonstrable Governance

6.1 A Progressive Transformation

The architecture proposed throughout this white paper should not be regarded as an all-or-nothing solution.
The objective is to present an innovative way of thinking about information systems.

The proposed principles can be adopted progressively.

Each guarantee introduced reduces uncertainty.

Each item of evidence strengthens trust.

Organisations can therefore evolve at their own pace according to their priorities, regulatory requirements and technical maturity.

Demonstrable Governance is not a product.

It is an architectural approach.

6.2 Start by Describing

Most organisations already have business applications.

Replacing them is rarely realistic.

The first step is therefore to **describe rather than modify**.

Business processes should progressively be documented through structured and usable artefacts describing:

- business processing;
- exchanged data;
- expected contracts;
- technical dependencies;
- execution policies.

At this stage, no Execution Engine is required.

Simply making implicit knowledge explicit already represents a significant improvement.

The **Business Process Definition** becomes the first governance artefact.

6.3 Introduce Digital Signatures

Once critical artefacts have been identified, they should progressively become verifiable.

Digital signatures represent one of the simplest ways of establishing trust.

Organisations can begin by signing:

- Business Process Definitions;
- business processing components, such as DLLs and executables;
- Data Contracts.

At this stage, signatures may simply be verified during deployment.

Execution itself remains unchanged.

Nevertheless, organisational commitments become objectively verifiable.

6.4 Characterise Behaviour

Traditional deployment processes generally validate software through testing.

The proposed architecture introduces an additional perspective.

Before deployment, each **Application Component** is characterised.

Its technical capabilities are summarised within a **Behaviour Manifest**.

The objective is not to reject software automatically, but to understand what the software is technically capable of doing and formalise those capabilities within a governed artefact — the Behaviour Manifest.

Risk assessments therefore become significantly more objective.

Review efforts can focus on changes that genuinely modify the technical trust perimeter.

6.5 Produce Evidence

Many organisations already collect operational logs.

However, operational logs primarily describe events.

The proposed architecture recommends producing **Evidence** instead.

Rather than merely recording that a component executed, the execution environment records evidence that the required execution conditions were satisfied.

This distinction fundamentally changes the purpose of operational records.

Evidence becomes the foundation of governance.

Audits no longer rely exclusively on documentation.

They rely on verifiable execution evidence.

6.6 Orchestrate Guarantees

Only once guarantees, actions, controls and evidence have been introduced does the Execution Engine become truly valuable.

Its role is to coordinate guarantees that have already been defined.

The Execution Engine does not replace governance.

It **operationalises it**.

It continuously verifies that all required guarantees have been demonstrated before authorising the next execution stage.

Execution therefore becomes the consequence of governance rather than an independent technical activity.

6.7 Govern Guarantees Rather than Applications

Traditional governance reviews, audits and certifies applications.

The proposed architecture shifts this perspective.

Rather than governing applications directly, organisations govern the guarantees expected from those applications.

Business processing components become interchangeable.

What truly matters is that each execution demonstrates:

- data integrity;
- code integrity;
- compliance with Data Contracts;
- validated technical behaviour;
- controlled publication;
- recoverable execution.

The object of governance is therefore no longer the software itself, but the set of guarantees that software must continuously demonstrate.

6.8 A Maturity Model

Organisations rarely transform overnight.

Demonstrable Governance can therefore be adopted progressively.

One possible maturity model is shown below:

Level	Objective
1	Describe business processes
2	Introduce Data Contracts
3	Digitally sign critical artefacts
4	Characterise software behaviour
5	Produce execution evidence

6	Automatically orchestrate guarantees
7	Govern the enterprise through verifiable evidence

Each level provides independently measurable value.

Organisations are therefore free to stop at any point or continue towards fully demonstrable governance.

6.9 Beyond the Proposed Architecture

The proposed architecture is technology-independent.

Any execution platform could progressively adopt similar concepts, including:

- container platforms;
- integration platforms;
- workflow engines;
- microservice orchestrators;
- cloud-native execution environments.

The essential objective remains unchanged:

“Turn organisational policies into demonstrable guarantees supported by verifiable evidence.”

6.10 Chapter Conclusion

For decades, information systems were primarily designed to execute business functionality.

Growing regulatory requirements, increasingly sophisticated cyber threats and the accelerating pace of software delivery now require a different objective.

Organisations must no longer merely execute business processes correctly.

They must also be able to demonstrate that those processes were executed in accordance with the policies they defined.

This white paper proposes an architectural model in which governance, architecture, security and execution form a single continuous chain:

- policies become guarantees;
- guarantees are verified by specialised Guarantee Controllers;
- Guarantee Controllers produce Evidence;
- Evidence guides execution;
- execution, in turn, continuously demonstrates governance.

Ultimately, trust is no longer assumed.

It is **built, verified, recorded and continuously demonstrated**.

Chapter 7 — Limitations of the Model

An unobservable property cannot be demonstrated.

A **Behaviour Manifest** can characterise observable capabilities; it cannot demonstrate the absence of every vulnerability.

Evidence demonstrates a fact within a given context. It does not automatically guarantee the business validity of the result.

The architecture cannot create guarantees that the underlying infrastructures do not provide.

Demonstrable Governance complements organisational governance; it does not replace it.

Some decisions remain human decisions, including risk acceptance, policy qualification, behaviour approval and degraded-mode operation.

Annex A — Vocabulary of Fundamental Concepts

A.1 Execution Engine

Definition	Software component responsible for executing business processing within a controlled environment.
Description	The Execution Engine provides the resources required for processing — data, secrets and reference data — applies governance controls and actions, manages the transaction, produces execution evidence and ensures recovery following an incident.
Responsibilities	• load the business processing component; • provide data; • provide secrets; • manage transports; • apply policies; • produce evidence.

A.2 Guarantee Orchestrator

Definition	Component responsible for applying the controls and actions required to satisfy governance policies before, during and after the execution of business processing.
Responsibility	It orchestrates guarantees, not business processing.

A.3 Control

Definition	A verification that produces a decision. Examples: • integrity is verified; • contract requirements are met; • signature is valid.
Responsibility	Answer **yes** or **no**.

A.4 Action

Definition	An operation performed in order to produce a guarantee or evidence. Examples: • sign data;

	• record a state; • archive an artefact.
Responsibility	Modify the state of the system.

A.5 Policy

Definition	A set of rules defining the organisation’s governance objectives Example: All personal data must be signed. The Execution Engine does not interpret the objective; it applies the controls that give effect to it.

A.6 Guarantee

Definition	A demonstrable result obtained through one or more controls and actions. Example: Data integrity guarantee.

A.7 Evidence

Definition	An objective item demonstrating that a guarantee has been established. Examples: • signature; • journal; • transactional state; • control result.

A.8 Trust

Definition	A level of assurance obtained through the accumulation of demonstrable guarantees. Trust is not a mechanism. **It is a consequence.**

A.9 Manifest

Definition	A signed description of an artefact, processing component or key, used to verify its compliance and identity.

A.10 Data Contract

Definition	Describes the expected structure of data.

A.11 Business Contract

Definition	Describes the services provided by an application.

A.12 Execution Contract

Definition	Describes the guarantees provided by the Execution Engine.

A.13 Domain

Definition	A scope of responsibility associated with a policy, keys or artefacts. Examples: • Enterprise • Platform • Business • Data

A.14 Chain of Trust

Definition	A set of certification relationships used to demonstrate that an artefact originates from a recognised authority.

A.15 Demonstrable Governance

Definition	An approach that transforms organisational policies into technical mechanisms that automatically produce the evidence required to demonstrate that those policies have been applied.