AspectCompliance: Structural Modularization of LGPD Compliance through Declarative Aspect-Oriented Enforcement

Fernando Antonio Dantas Gomes Pinto[1], **Augusto Cesar Espíndola Baffa**[1], **Edward Hermann Haeusler**[1]

[1]Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Departamento de Informática
Rio de Janeiro – RJ – Brazil

{fpinto, abaffa, hermann}@inf.puc-rio.br

Abstract. *The Brazilian General Data Protection Law (LGPD) imposes strict requirements for accountability, traceability, and lawful personal data processing in public-sector systems. In practice, enforcing these requirements often leads to code scattering and tight coupling between regulatory policies and business logic. This paper introduces AspectCompliance, a lightweight aspect-oriented framework for Python that supports declarative enforcement of LGPD policies. Access control, audit logging, and data masking are treated as cross-cutting concerns, preserving their separation from domain services. A controlled experiment compares embedded compliance logic, procedural aspect attachment using* `aspectlib`*, and declarative weaving with AspectCompliance. The results indicate improved structural separation and policy centralization while maintaining acceptable runtime behavior.*

Resumo. *A Lei Geral de Proteção de Dados (LGPD) impõe requisitos rigorosos de responsabilização, rastreabilidade e tratamento lícito de dados pessoais em sistemas do setor público. Na prática, a aplicação desses requisitos frequentemente leva ao espalhamento de código e ao forte acoplamento entre políticas regulatórias e lógica de negócio. Este artigo apresenta o AspectCompliance, um framework leve orientado a aspectos para Python que oferece suporte à aplicação declarativa de políticas da LGPD. Controle de acesso, registro de auditoria e mascaramento de dados são tratados como preocupações transversais, preservando sua separação em relação aos serviços de domínio. Um experimento controlado compara lógica de conformidade embutida, acoplamento procedural de aspectos com* `aspectlib` *e weaving declarativo com AspectCompliance. Os resultados indicam melhora na separação estrutural e na centralização das políticas, mantendo comportamento aceitável em tempo de execução.*

1. Introduction

The Brazilian General Data Protection Law (LGPD) established a regulatory framework for the processing of personal data in Brazil [Presidência da República do Brasil 2018].

Public-sector information systems frequently handle sensitive citizen data, including identification, fiscal, and administrative records, and are therefore subject to strict requirements concerning accountability, traceability, and lawful processing. In digital government environments, compliance requirements are particularly critical due to the large-scale processing of citizen-sensitive information and the increasing demand for transparency, accountability, and auditability in public administration. Ensuring compliance with these requirements often means embedding access control checks, audit logging, and data masking procedures directly into application services, following privacy-oriented architectural principles such as Privacy by Design [Cavoukian 2009]. Although effective at a local level, this practice leads to code scattering and tangling, reduces maintainability, and makes regulatory evolution harder to manage. Aspect-Oriented Programming (AOP) was introduced to modularize cross-cutting concerns through abstractions such as join points, pointcuts, and advices [Kiczales et al. 1997, Laddad 2003]. These constructs support a systematic separation between functional logic and auxiliary enforcement behavior. However, to the best of our knowledge, structured aspect-oriented enforcement is still not commonly adopted in many public-sector software environments, particularly in Python-based systems.

To address this gap, we introduce *AspectCompliance*, a lightweight aspect-oriented framework for Python designed to support the declarative and non-intrusive enforcement of regulatory concerns. Unlike common decorator-based mechanisms that require explicit annotation of service methods [Maries 2026], *AspectCompliance* uses a dedicated DSL for pointcut specification, enabling compliance policies to be applied systematically without modifying domain code. Beyond the framework itself, this work proposes a structural characterization of compliance modularity. We model software systems under regulatory constraints as a compositional structure $\mathcal{S} = (D, C, W)$, where D denotes domain logic, C compliance rules, and W the mechanism that binds them. From this formulation, we analyze structural separation and reduced scattering of compliance rules, and compare three enforcement styles: embedded enforcement, procedural aspect attachment, and declarative weaving. Therefore, this work proposes an aspect-oriented compliance framework for public-sector systems operating under LGPD constraints, with emphasis on centralized enforcement, compliance modularity, and maintainability in digital government environments.

The remainder of this paper is organized as follows. Section 2 discusses LGPD-driven governance challenges in public-sector information systems. Section 3 reviews related work on aspect-oriented programming. Section 4 presents the proposed enforcement model and the *AspectCompliance* architecture, and details the experimental design, including evaluation metrics and the structural characterization of compliance modularity. Section 5 reports and analyzes the comparative results. Finally, Section 6 concludes the paper and outlines directions for future research.

2. LGPD and Governance Challenges in Public-Sector Systems

Data protection requirements manifest as cross-cutting concerns in software systems. They span multiple services, evolve independently from business logic, and require consistent traceability. In the context of LGPD [Presidência da República do Brasil 2018, Cristóvam et al. 2021], these requirements include access control, audit logging, data masking, and accountability mechanisms [Cavoukian 2009], all of which directly affect

digital governance in public administration.

As a concrete example, consider a municipal citizen service system that allows consultation of personal records and the generation of administrative reports. Each service that accesses sensitive information must verify permissions, log access events for accountability, and apply masking or anonymization depending on the user profile. In conventional implementations, these mechanisms are repeatedly embedded across multiple services, creating a structural pattern of code scattering. Traditional implementations typically incorporate compliance logic through middleware layers, decorators, or local validations inserted directly into service methods. This pattern can be illustrated in simplified form:

```
def get_citizen(cpf):
    check_permission()
    log_access(cpf)
    citizen = repository.find(cpf)
    citizen.cpf = mask(citizen.cpf)
    return citizen
```

Listing 1. Simplified service implementation example.

In AOP terminology, the execution of a service operation constitutes a `join point`. `Pointcuts` declaratively select such execution points, and `advices` specify additional behavior to be applied `before`, `after`, or `around` them [Laddad 2003]. Similar structures tend to be replicated across different services, increasing coupling between regulatory rules and functional logic. Figure 1 illustrates how compliance mechanisms become distributed across multiple application components.

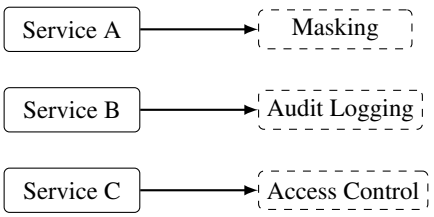

Figure 1. Distribution of compliance mechanisms across distinct services.

In governmental environments, where regulatory changes are frequent and systems typically have long operational lifecycles, this structural dependency compromises maintainability, complicates continuous auditing, and increases the cost of regulatory evolution. The introduction of a new legal requirement may demand modifications at multiple service points, increasing the risk of inconsistent enforcement. Aspect-Oriented Programming (AOP) offers an architectural paradigm aimed at modularizing cross-cutting concerns [Rashid 2004]. Through abstractions such as *join points*, which denote well-defined execution points in a program, *pointcuts*, which declaratively select sets of such execution points, and *advices*, which specify additional behavior to be executed at those points [Kiczales et al. 1997], AOP enables an explicit separation between functional logic and auxiliary enforcement behavior. Figure 2 presents the architectural contrast, highlighting the centralization of compliance policies within a dedicated aspect.

However, in many programming environments, structured support for aspect-oriented enforcement is either absent or limited to partial interception mechanisms. Such

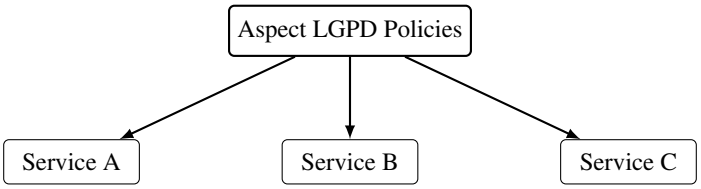

Figure 2. Centralization of compliance policies through aspect-oriented modularization.

solutions typically operate at architectural boundaries and do not provide an explicit transversal composition model. Although not exclusive to public-sector systems, it becomes particularly critical in contexts characterized by frequent regulatory change, institutional accountability, and long system lifecycles. In the case of LGPD, legal requirements such as access control, traceability, and accountability demand consistent and systematically enforceable mechanisms across multiple services. When compliance logic is dispersed within application code, ensuring uniform enforcement and adapting to new regulatory interpretations becomes costly and error-prone. The absence of explicit architectural support for modular compliance enforcement therefore constrains independent policy evolution and weakens digital governance capabilities.

3. Related Work

In the seminal AOP work, Kiczales [Kiczales et al. 1997] introduced aspect-oriented programming as a way to modularize cross-cutting concerns such as security, logging, and auditing. In the AspectJ framework and in Laddad's work [The AspectJ Team 2003, Laddad 2003], this paradigm is realized through explicit pointcuts and structured weaving, establishing a reference model for modular policy enforcement. In the Python ecosystem, approaches such as `aspectlib` [Maries 2026] support runtime interception of cross-cutting concerns, but rely on explicit decoration or procedural runtime binding rather than on declarative policy specification and explicit structural separation of compliance logic.

Thulnoon [Thulnoon et al. 2017] proposes an aspect-oriented approach to privacy preservation in distributed systems, combining dynamic weaving with cryptographic controls to regulate data forwarding among network nodes. Their work demonstrates that AOP can modularize privacy and security concerns without directly modifying the original application code, although their proposal is focused on distributed communication security and encryption orchestration rather than on declarative regulatory compliance enforcement in application-level services.

From the perspective of privacy regulation in digital government environments, the LGPD text [Presidência da República do Brasil 2018] and Privacy by Design [Cavoukian 2009] frame privacy compliance as a software engineering problem involving recurring enforcement mechanisms such as access control, audit logging, and data masking across service endpoints. In practice, Peixoto [Peixoto et al. 2023] reports recurring challenges related to privacy awareness, the distinction between privacy and security, and the practical adoption of privacy-preserving strategies in software projects. However, such studies remain primarily diagnostic, focusing on developers' perceptions and organizational practices rather than on reusable architectural mechanisms for embedding

privacy constraints into software systems.

Unlike these approaches, *AspectCompliance* combines declarative pointcut specification with an explicit structural account of compliance modularity. The proposed framework characterizes compliance enforcement as an architectural concern whose separation, centralization, and binding mechanisms can be made explicit and systematically analyzed.

4. Research Design and Methods

This section outlines the proposed enforcement approach and the experimental setup used to evaluate its impact on modularity and governance support.

4.1. Aspect-Oriented Enforcement Model

The proposed approach adopts an aspect-oriented enforcement model in which compliance policies are treated as cross-cutting concerns and applied dynamically at runtime. The model relies on four core abstractions: **Join Points**, which correspond to function calls in the target system; **Pointcuts**, declarative expressions that select sets of join points based on structural patterns (e.g., module path and function name); **Advices**, behavioral units associated with pointcuts that define logic to be executed before, after, or around the intercepted operation; and **Runtime Weaving**, a dynamic composition mechanism that binds advices to matching join points during execution. In this model, enforcement logic remains outside the business services. Policies such as audit logging, access control, and data masking are encapsulated in advices rather than embedded in service code. When a selected join point is reached, the corresponding advice intercepts execution through an explicit `proceed()` operation, allowing the original behavior to continue under controlled conditions. This arrangement enables compliance rules to evolve independently from functional logic, reducing code scattering and improving auditability.

4.2. Framework Architecture

The enforcement model is implemented as a lightweight framework that provides operational support for aspect-oriented concepts without modifying the syntax of the host language. The architecture comprises four main components: **DSL Parser**, which parses a domain-specific language for pointcut declaration using ANTLR [Parr 2013] and produces a structured `PointcutSpec` representation; **Matcher Engine**, which matches functions against pointcut specifications based on module and function name patterns; **JoinPoint Abstraction**, which encapsulates the execution context of intercepted calls, including the function reference, arguments, and a `proceed()` operation; and **Weaver**, which dynamically wraps matched functions and associates them with registered advices. Weaving occurs at runtime and does not require recompilation or source-code modification. This is particularly valuable in public-sector systems, where regulatory adjustments often need to be incorporated incrementally and with low disruption. The framework also supports centralized aspect registration through a global registry, enabling systematic enforcement across modules without invasive changes to application services.

4.3. Experimental Setup

To evaluate the structural and governance implications of the proposed approach, we designed a controlled comparative experiment based on a governmental information sys-

tem that processes sensitive citizen data. The experiment centers on a service operation, `get_citizen(cpf, role)`, which returns personal information. Three user roles are considered: **admin**, with full access; **auditor**, with CPF masking and name anonymization; and **guest**, whose access is denied. The regulatory requirements reflect typical LGPD obligations in public-sector systems, including access control verification, audit logging of sensitive operations (accountability), and conditional data masking and anonymization (privacy by design). Under these conditions, we developed three implementations with identical functional requirements: **Traditional Implementation**: compliance logic is embedded directly in the service method, intertwining business and regulatory concerns; **Procedural AOP Implementation**: compliance logic is externalized into an aspect using `aspectlib`, with explicit procedural attachment to the target service; **Declarative AOP Implementation**: compliance logic is externalized into an aspect using AspectCompliance, with declarative binding defined through a pointcut expression and applied by runtime weaving.

To make the differences between these approaches explicit, we present below minimal excerpts from the three implementations used in the experiment. The code excerpts in Listings 2, 3 and 4 are intentionally simplified to emphasize the structural differences among the implementations. The complete experimental artifact includes the three evaluated roles (`admin`, `auditor`, and `guest`), together with the corresponding authorization and transformation logic.

```python
def get_citizen_traditional(cpf, role="guest"):
    audit("BEGIN", {"op": "get_citizen", "role": role})
    if not has_permission(role, "citizen:read"):
        audit("DENY", {"op": "get_citizen", "role": role})
        raise PermissionError("Unauthorized")
    result = repository.find(cpf)
    if role == "auditor":
        result["cpf"] = mask_cpf(result["cpf"])
        result["name"] = anonymize_name(result["name"])
    audit("END", {"op": "get_citizen", "role": role})
    return result
```

Listing 2. Traditional implementation with embedded compliance logic.

```python
import aspectlib
def get_citizen_aspectlib_raw(cpf, role="guest"):
    return repository.find(cpf)
@aspectlib.Aspect(bind=True)
def lgpd_aspect(cutpoint, *args, **kwargs):
    role = kwargs.get("role", "guest")
    audit("BEGIN", {"op": cutpoint.__name__, "role": role})
    if not has_permission(role, "citizen:read"):
        audit("DENY", {"op": cutpoint.__name__, "role": role})
        raise PermissionError("Unauthorized")
    result = yield aspectlib.Proceed
    if role == "auditor":
        result["cpf"] = mask_cpf(result["cpf"])
        result["name"] = anonymize_name(result["name"])
    audit("END", {"op": cutpoint.__name__, "role": role})
    yield aspectlib.Return(result)
get_citizen_aspectlib = lgpd_aspect(get_citizen_aspectlib_raw)
```

Listing 3. Procedural AOP with explicit *aspectlib* attachment.

As shown in Listing 2, the traditional implementation embeds compliance logic directly in the service body, increasing coupling between regulatory rules and domain

```
def get_citizen_ac_raw(cpf, role="guest"):
    return repository.find(cpf)
pc = pointcut("execution: examples.* -> get_*")(lambda: None)
@aspect(pc)
def lgpd_aspect(jp):
    role = jp.kwargs.get("role", "guest")
    audit("BEGIN", {"op": jp.func_name, "role": role})
    if not has_permission(role, "citizen:read"):
        audit("DENY", {"op": jp.func_name, "role": role})
        raise PermissionError("Unauthorized")
    result = jp.proceed()
    if role == "auditor":
        result["cpf"] = mask_cpf(result["cpf"])
        result["name"] = anonymize_name(result["name"])
    audit("END", {"op": jp.func_name, "role": role})
    return result
```

Listing 4. Declarative AOP with *AspectCompliance*.

functionality. Listings 3 and 4, in contrast, externalize compliance behavior into aspects. In `aspectlib`, composition is achieved through explicit procedural attachment; in *AspectCompliance*, it is defined through a pointcut expression and applied by runtime weaving. All three listings implement the same functional requirements under identical regulatory constraints and serve as the experimental artifacts used in the comparative evaluation reported in Section 5. Structurally, the traditional implementation corresponds to an intertwined configuration $D' = \{d_i \oplus C_i\}$, whereas the two aspect-oriented variants preserve the distinction between domain logic D and compliance rules C, delegating composition to a mechanism W. In `aspectlib`, W is local and procedural; in *AspectCompliance*, it is declarative and independent. As a result, policy scope becomes more explicit and scattering is reduced during policy evolution.

4.3.1. Evaluation Metrics

To support systematic comparison, we define the following criteria. Let $LOC_{business}$ denote the number of lines in the business service body, LOC_{policy} the number of lines dedicated to regulatory enforcement, and $LOC_{binding}$ the number of lines required to connect policy and business logic through decoration, attachment, pointcut declaration, or weaving. In the traditional implementation, business and compliance logic occupy the same service body, so these LOC values are interpreted analytically rather than as physically disjoint segments.

(1) Separation Index: We define the *Separation Index* as a ternary structural indicator:

$$SI = \begin{cases} 0 & \text{if compliance logic is embedded directly in service methods;} \\ 1 & \text{if compliance logic is externalized but attached procedurally through decorators or wrappers;} \\ 2 & \text{if compliance logic is externalized and declaratively bound through an independent weaving mechanism.} \end{cases}$$

This metric distinguishes not only whether compliance logic is externalized, but also how the composition between policy logic and business logic is realized.

Under this definition, the traditional implementation yields $SI = 0$, the `aspectlib` implementation yields $SI = 1$, and the *AspectCompliance* implementation yields $SI = 2$.

(2) Functional Equivalence: Let f_T, f_L, and f_A denote the traditional, `aspectlib`, and *AspectCompliance* implementations, respectively. Functional equivalence holds if:

$$\forall (cpf, role) \in I,\ f_T(cpf, role) \equiv f_L(cpf, role) \equiv f_A(cpf, role)$$

where I is the set of valid input pairs considered in the experiment.

(3) Runtime Overhead: Let t_T, t_L, and t_A denote the mean execution times of the traditional, `aspectlib`, and *AspectCompliance* implementations, respectively, measured over repeated invocations under the same input conditions. Runtime overhead is compared to assess whether gains in modularity are accompanied by a measurable execution-time cost.

4.4. Structural Characterization of Compliance Modularity

Since the goal of this work is to characterize an enforcement architecture rather than merely describe an implementation, we formalize its structural properties through a set of architectural invariants. We model a software system under regulatory constraints as a triple: $\mathcal{S} = (D, C, W)$ where D denotes the set of domain functions (business logic), C the set of compliance rules, and W the composition mechanism that binds C to D.

Traditional Architecture In the traditional implementation, compliance logic is embedded directly within domain functions. Formally, this can be represented as:

$$D' = \{d_i \oplus C_i \mid d_i \in D,\ C_i \subseteq C\}$$

where $\oplus$ denotes the structural embedding of compliance rules inside domain code. In this configuration, W is implicit and distributed across D'.

Aspect-Oriented Architectures In the aspect-oriented variants, domain logic and compliance logic remain structurally distinct. The composition mechanism $W : D \times C \to D^*$ binds both layers dynamically, where D^* denotes the runtime-composed system behavior. In this configuration, D remains unchanged, C is modularized, and W performs the binding between them. The difference between the two aspect-oriented variants lies in the nature of W. In the procedural AOP variant, W is expressed through explicit local attachment of aspects to target functions. In the declarative AOP variant, W is specified through pointcut expressions and runtime weaving, making the binding relation itself explicit at the architectural level.

Scattering Property Let $occ(c)$ denote the number of occurrences of a compliance rule c across domain functions. The *Scattering Degree* is defined as $SD : C \to \mathbb{N}$, where

$SD(c) = occ(c)$. A fully centralized enforcement architecture satisfies $\forall c \in C,\ SD(c) = 1$.

This property highlights the governance advantages of modular compliance enforcement. The formal characterization above establishes the structural basis of compliance modularity as an architectural property. Together, these invariants and evaluation metrics provide the analytical basis for the empirical comparison reported in Section 5.

5. Results and Comparative Analysis

This section presents empirical results evaluating structural modularity and discusses the governance implications of centralized LGPD enforcement in public-sector systems.

5.1. Functional Equivalence

The three implementations were executed under identical input conditions using the experimental script described in Section 4. The evaluated roles (`admin`, `auditor`, and `guest`) yielded equivalent observable outcomes across the traditional, `aspectlib`, and *AspectCompliance* configurations. For the authorized roles, all implementations produced the same results: `admin` received the original record, `auditor` received masked and anonymized data, and `guest` was denied access. This confirms the functional equivalence property defined in Section 4:

$$\forall (cpf, role) \in I,\ f_T(cpf, role) \equiv f_L(cpf, role) \equiv f_A(cpf, role)$$

where f_T, f_L, and f_A denote the traditional, `aspectlib`, and *AspectCompliance* implementations, respectively.

The architectural separation introduced by the two aspect-oriented variants does not alter the functional semantics of the system. The observed differences are therefore structural and organizational rather than behavioral.

5.2. Structural Metrics Comparison

The experimental artifact makes it possible to compare structural properties related to compliance modularization. In the traditional implementation, business and compliance logic remain intertwined in the same service body. In the two aspect-oriented implementations, compliance behavior is externalized into a separate policy layer, leaving only a small business core in the service function. Table 1 summarizes the structural measurements obtained in the experiment.

Criterion	**Traditional**	**`aspectlib`**	**AspectCompliance**
Policy Location	Inside service	Externalized aspect	Centralized declarative aspect
Scattering Risk	High	Low	Low
Policy Evolution	Edit many services	Edit aspect / attach explicitly	Edit one aspect / pointcut
Audit Consistency	Duplication-prone	Centralized	Centralized
Business LOC ($\approx$)	12	2	4
Policy LOC ($\approx$)	12	15	15
Binding/Weaving LOC ($\approx$)	–	1	2
Separation Index (SI)	0	1	2

Table 1. Comparative structural properties of the three implementations.

The results show a progressive increase in architectural separation. In the traditional implementation, compliance rules are embedded directly in the service body, yielding the lowest separation level. In the `aspectlib` implementation, compliance logic is externalized, but its attachment remains procedural and local to the target function. In *AspectCompliance*, compliance logic is likewise externalized, but the connection between policy and target operation is made explicit through declarative pointcut specification and runtime weaving. The Separation Index reflects these architectural differences. The traditional implementation yields $SI = 0$, since compliance logic is embedded in service methods. The `aspectlib` implementation yields $SI = 1$, since compliance logic is externalized but attached procedurally. *AspectCompliance* yields $SI = 2$, since compliance logic is not only externalized but also declaratively bound through an independent weaving mechanism. The LOC measurements reinforce this interpretation. In the traditional implementation, business and compliance logic occupy the same method, so the reported values should be read as an analytical decomposition of the service body rather than as physically disjoint blocks. By contrast, the `aspectlib` and *AspectCompliance* variants leave only a small business core in the service function while moving the policy logic into a separate aspect. The additional Binding/Weaving LOC captures the explicit cost of connecting policy logic to business logic: one line in the procedural attachment of `aspectlib`, and two lines in the declarative composition of *AspectCompliance*. This difference is small in absolute terms but architecturally meaningful, since in the declarative case, the binding relation becomes visible and analyzable as a first-class design element. From the standpoint of policy evolution, the traditional implementation still requires changes to be propagated across service methods, whereas both AOP variants centralize updates within an aspect. However, only *AspectCompliance* centralizes the architectural binding mechanism.

5.3. Performance Comparison

To assess whether gains in modularity introduce measurable runtime costs, we benchmarked the three implementations under repeated invocation for the `admin` and `auditor` roles. Measurements were obtained with `time.perf_counter()` over 5 repetitions of 5000 calls per condition, using in-memory audit accumulation to avoid console I/O interference. Table 2 summarizes the mean execution times observed in the experiment.

Metric	Traditional	`aspectlib`	AspectCompliance
Admin mean total (s)	0.015357	0.022181	0.018375
Admin mean/call (μs)	3.071	4.436	3.675
Auditor mean total (s)	0.021655	0.029160	0.025304
Auditor mean/call (μs)	4.331	5.832	5.061

Table 2. Comparative runtime measurements under repeated invocation.

The traditional implementation exhibited the lowest execution time in both benchmark scenarios, which is expected given that no interception or dynamic weaving mechanism is introduced beyond the service body itself. Both aspect-oriented approaches incurred a measurable runtime overhead, but the observed differences remained small in absolute terms, on the order of microseconds per call. Among the two AOP variants, `aspectlib` presented the highest overhead in both benchmark scenarios, whereas *AspectCompliance* remained consistently closer to the traditional baseline. For the `admin`

role, the mean execution time increased from 3.071 μs per call in the traditional implementation to 4.436 μs in `aspectlib` and 3.675 μs in *AspectCompliance*. For the `auditor` role, the corresponding values were 4.331 μs, 5.832 μs, and 5.061 μs, respectively. These results indicate that the modularization gains obtained through aspect-oriented enforcement are accompanied by a limited and predictable runtime cost rather than a prohibitive performance penalty. They also suggest that the declarative weaving strategy adopted in *AspectCompliance* does not imply higher overhead than the procedural alternative used in `aspectlib`; in this experiment, it remained systematically below the `aspectlib` measurements. The performance results indicate that the proposed framework prioritizes governance modularity and centralized compliance management while maintaining acceptable runtime overhead.

5.4. Governance Implications

The results also have important governance implications for public-sector systems. Centralizing LGPD enforcement reduces the risk of inconsistent policy application across services and makes regulatory control points more explicit. In traditional implementations, audit and enforcement responsibilities remain distributed across service bodies, making regulatory evolution and policy maintenance harder to manage. In both *AOP* variants, audit logging, access control, and masking rules are centralized, improving traceability and consistency in policy enforcement. *AspectCompliance* extends this advantage by making the policy-binding mechanism explicit and declarative, thereby improving architectural visibility and maintainability. Taken together, these results reinforce the role of compliance modularity as a governance-oriented architectural property in digital government systems.

6. Conclusion and Future Work

This paper presented *AspectCompliance*, a lightweight aspect-oriented framework designed to modularize LGPD compliance enforcement through declarative runtime weaving. By separating domain logic from regulatory policies, the proposed approach reduces structural scattering, localizes compliance logic in a distinct architectural unit, and strengthens governance capabilities. The comparative evaluation demonstrated that declarative enforcement preserves functional equivalence while improving structural modularity. The formal characterization $\mathcal{S} = (D, C, W)$ provided a compositional interpretation of compliance enforcement as an architectural property rather than a dispersed implementation artifact. The empirical findings also indicate improved structural separation, more explicit policy binding, and centralized policy evolution, reinforcing the governance advantages of modular compliance control.

This study has some limitations. The evaluation was conducted in a controlled and small-scale experimental environment, focusing primarily on architectural and governance aspects rather than production-scale deployment. In addition, the framework was not validated in an operational governmental information system. Future validation efforts include evaluating the framework in more realistic operational scenarios and public-sector environments. The source code, experimental scripts, and reproducibility artifacts are publicly available[1].

[1]Project repository: `https://github.com/fernandoantoniodantas/AspectPy`

Future work includes extending the framework to support automated compliance verification mechanisms capable of detecting policy inconsistencies across distributed services. We also intend to investigate explicit mappings between LGPD legal provisions and executable enforcement patterns within the declarative DSL, thereby strengthening the connection between normative interpretation and system-level enforcement. Future research also includes broader performance evaluations involving latency, throughput, and memory consumption in more realistic operational and public-sector environments. Additionally, the framework is being prepared for public release through the Python Package Index (PyPI), enabling installation via the standard `pip install` mechanism and facilitating broader adoption in digital government development environments. By positioning compliance enforcement as a measurable structural property, this work contributes to advancing architectural governance models aligned with evolving data protection regulations in public administration.

References

Cavoukian, A. (2009). Privacy by design: The 7 foundational principles. *Information and privacy commissioner of Ontario, Canada*, 5(2009):12.

Cristóvam, J. S. d. S., Bergamini, J. C. L., and Hahn, T. M. (2021). Governança de dados no setor público brasileiro: uma análise a partir da lei geral de proteção de dados (lgpd). *Interesse Público*, 23(129):75–101.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of ECOOP'97*. Springer. LNCS 1241.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-oriented Programming*. In Action Series. Manning.

Maries, I. C. (2026). aspectlib documentation. `https://python-aspectlib.readthedocs.io/`. Accessed in 2026.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers. Pragmatic Bookshelf, 2nd edition.

Peixoto, M., Ferreira, D., Cavalcanti, M., Silva, C., Vilela, J., ao Araújo, J., and Gorschek, T. (2023). The perspective of brazilian software developers on data privacy. *Journal of Systems and Software*, 195:111523.

Presidência da República do Brasil (2018). Lei no. 13.709, de 14 de agosto de 2018 (lei geral de proteção de dados pessoais). Planalto, official publication.

Rashid, A. (2004). *Aspect-Oriented Programming (AOP)*, pages 21–38. Springer Berlin Heidelberg, Berlin, Heidelberg.

The AspectJ Team (2003). The aspectj programming guide. Eclipse Foundation documentation. Online documentation (released guide).

Thulnoon, A. A., Lempereur, B., Shi, Q., and Al-Jumeily, D. (2017). Using aspect oriented programming to enforce privacy preserving communication in distributed systems. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ICC '17, New York, NY, USA. Association for Computing Machinery.