

Redefining Global Payroll Architecture: From Monolithic SAP HR Cores to Event-Driven Orchestration

Lakshmi Srinivas Gogula

Fortune 500 Retail Company IT, USA

Received: 09.02.2026

Accepted: 15.02.2026

Abstract

SAP standard HR payroll systems are monolithic, ABAP-based systems which contain payroll calculation logic, compliance rules, and integration workflows tailor-made for those systems, and tightly coupled to the infrastructure. They provide limited extensibility, pose a higher risk of regression during updates, lack transparency for auditing, and are difficult to adapt to jurisdictional rules in multi-country implementations. In this reference architecture, payroll execution boundaries are redefined to use the SAP Business Technology Platform as a side-by-side, event-driven orchestration layer, and payroll validation and compliance enforcement are decoupled from the HR system of record by using API-first enabled services and asynchronous event propagation as well as evaluation of externally governed rules. This pattern provides a baseline to understand how to address these patterns at scale, improving audit traceability, reducing operational risk, and enabling progressive evolution of payroll logic.

Keywords: Event-Driven Architecture, SAP BTP Orchestration, Externalized Compliance Rules, API-First Integration, Temporal Audit Management

Introduction

Global payroll solutions built on SAP HR application architecture are monolithic ABAP-based execution platforms in the SAP core and include tightly coupled payroll calculation logic, compliance logic, and integration functionality. Despite these functional benefits, the architecture had limitations in multinational implementations such as limited extensibility, the risk of core function regression with each system upgrade, limited auditability, and limited responsiveness to compliance change within each jurisdiction. According to one study, 44% of reimplementations overestimated their required work, compared to 16% of upgrades. This may be due to the greater uncertainty involved in replacing a critical large system, and it has been found that 31% of reimplementations and almost 50% of upgrades are underestimated. These findings are further corroborated by empirical work in monolithic enterprise resource planning (ERP) environments, which report upgrade cycles of six to nine months for organizations. Together, these results show that, given the structural fragility of monolithic systems, tightly coupled customization complicates upgrade planning and exacerbates challenges in remediation [1].

Many of these boundaries become more strict and complicated when payroll processing is not just a single enterprise application but part of regulated financial infrastructure. This paper addresses these issues by redefining the payroll execution boundaries through a reference architecture, with SAP Business Technology Platform (BTP) as an event-driven side-by-side orchestration layer. An API-first approach to payroll validation, compliance and cross-system interaction leveraging asynchronous event-firing and

externally governed payroll business rules [2] could help reduce the re-occurrence of SAP payroll failures at scale, improve auditability, and lower operational risk. It could also enable the managed evolution of payroll business rules, reducing the burden on core HR systems.

Architectural Limitations of Monolithic SAP HR Payroll Systems

Structural Coupling and Extensibility Constraints

In customary SAP HR payroll systems, localization and other compliance-related content was bundled with the core application built in the ABAP programming language, which made extensibility particularly challenging in multinational environments, where these requirements differ by country and change frequently. Once the localizations become technical debt inside the core application, they become difficult to refactor, complicate the architecture and structure, and make the application more difficult to maintain in the long run. Decoupled approaches using externalized payroll services, an API-based orchestration layer, payroll tax calculation and compliance engines, and modular payroll engines allow enterprises to realize a much more modernized payroll operation without the risk and cost of a disruptive replacement [3].

Specifically, monolithic architectures had a slight edge of about 6% higher throughput than microservices architectures in concurrency testing scenarios, and no statistically important performance difference when load testing to find out whether adopting an architectural style would affect performance. Despite this small throughput benefit, applications using monolithic architectures require vastly different approaches to testing and validation. For example, if a change is made to enable a country-specific payroll feature, all integrated payroll components would need to be regression tested. Empirical studies of enterprise system testing cycles have found that the monolithic architecture entails more e2e test cases than modular architectures, that regression test suite execution time grows non-linearly with architecture size, outweighing the modest performance gains, and that tight coupling complicates scalability and maintainability of payroll systems. The resulting testing burden to accommodate deployments to organizations subject to multiple regulatory jurisdictions and changes that cannot safely be made in isolation has been cited as a barrier to operational responsiveness. The technology is such that jurisdiction-specific changes typically take several weeks to deploy [4].

Upgrade Fragility and Technical Debt Accumulation

System upgrades are complex and high-risk in monolithic SAP HR environments because the embedded customizations are in the system itself. The standard SAP upgrade processes do not integrate the local customizations, which must be remediated in their entirety. These factors require analysis, testing, and eventual refactoring of embedded custom logic at every major system upgrade, which creates project scope and defers adoption of new platform capabilities. In another study of enterprise system modernization, organizations using highly customized systems (i.e. greater than 20% modified objects) experienced client errors at a 14% higher rate than those using a low customization density system. The defect count associated with an upgrade was considerably higher for highly customized systems than for out-of-the-box systems due to exponentially higher risk from architectural complexity and wide-ranging code changes. Team experience is another reducing factor: systems designed and maintained by more experienced developers had lower rates of defects caused by client-side failure (38% lower) or vendor-side failure (12% lower). However, effects for team experience are often offset by the increasing size of the codebase. The results imply that each additional 1000 lines of code in the system under management increases the risk that the system is unable to handle client errors by 42%. Vendor related errors increase by 45%. The findings in this study support the concepts that high degrees of customization in upgrade

methodology are perilous even with highly skilled operations teams, and that architectural restraint or modularization of enterprise systems is helpful [5].

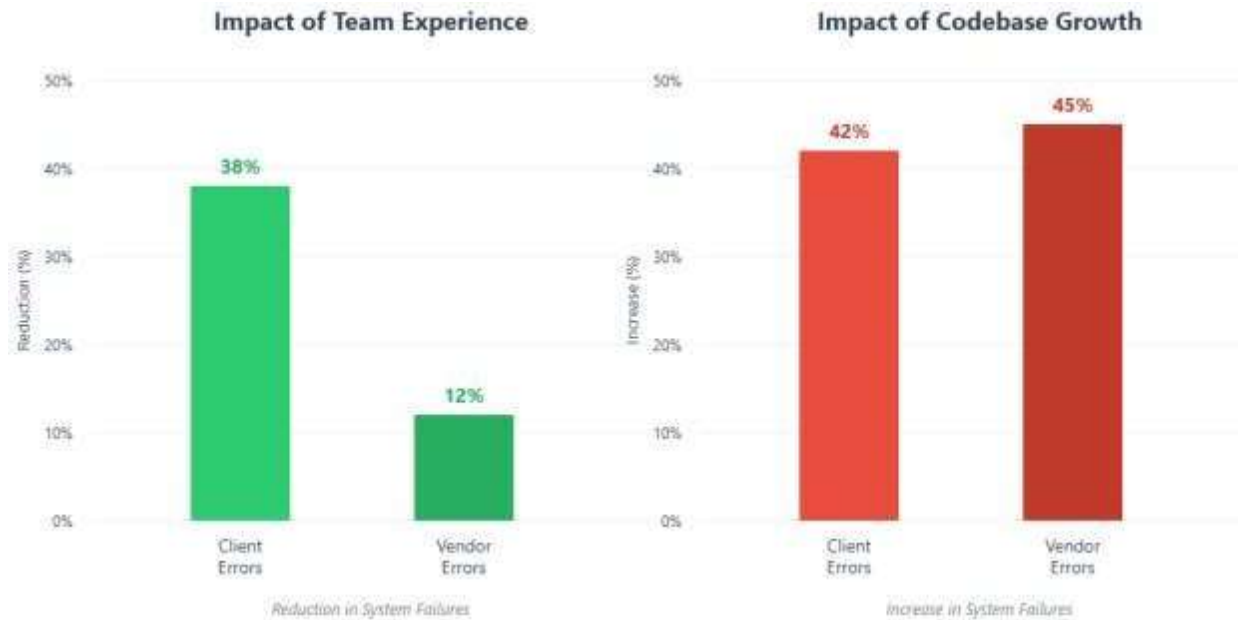


Figure 1: Impact of Team Experience and Codebase Growth on System Failures [5]

Design shortcuts and customized features frequently get introduced during design, causing further complications to a system's upgrade potential. In software development, this is known as technical debt. It is a commitment to pay later for future software maintenance. As custom logic and system integration continue to build up and technical divergence increases between the customized implementation and the vendor-supported configuration of the enterprise system, organizations may defer upgrades due to the perception that they are more risky and time-consuming to execute compared to the status quo. It has been found that the proportion of the effects of architectural maintenance on reducing system failure is about 20%, due to the effect of targeted client-level architectural maintenance activities on reducing technical debt. Targeted architectural maintenance can therefore be viewed as a partial offset to technical debt. Where these activities are delayed, technical debt amasses and gets worse: avoiding an upgrade leads to more structural decay and more expensive long-term maintenance to catch up, as well as further constraints on future upgrades [5].

Audit Transparency and Regulatory Compliance Challenges

Monolithic architectures can pose difficulties with auditability, regulatory compliance, and system evolution because business logic, compliance rules and data processing are executed as a single workflow, which makes it challenging to separate decision points and trace rule evaluations for auditing purposes. These limitations exist, and are magnified, as systems scale and become more complex or must operate in heterogeneous environments. Other research suggests systematic migration from a monolithic application architecture to a service-oriented architecture may yield benefits like 40% faster scaling events, a threefold increase in deployment frequency, and 50% reduction in mean time to recovery due to better fault isolation. Such gains have been attributed to disciplined decomposition using Domain-Driven

10.48047/jocaaa.2026.35.02.25

Design, API-first practices, and automated continuous integration/continuous delivery (CI/CD), which increase traceability, operational resilience, and preparedness for future compliance [3].

The embedded nature of compliance logic makes version control and change management complicated. This includes knowing which rules apply to which payroll runs, knowing when these rules have changed over time and providing documentation for regulatory compliance. The lack of separation between the core payroll calculation and the validation of regulatory compliance make it difficult to create a transparent and auditable payroll environment that is compliant with modern regulatory requirements. Monoliths also tend to outperform microservices in benchmarks with small differences under low to medium thread concurrency, with smaller numbers of concurrent threads leading to higher throughput and more successfully handled requests. In empirical concurrency and load testing studies, monolithic systems outperform microservices by an average of 0.87%, and by 6% at heavy concurrency. For several thousand threads, microservices and monoliths have similar performance, and microservices are even better at satisfying requests. Response time does not considerably differ, and both grow comparably, from 1790 milliseconds for 100 threads of very low concurrency, to 15000 milliseconds for 1000 threads of high concurrency. The performance advantage of the monolithic architecture, however, is workload-dependent and diminishes at scale [4].

Event-Driven Orchestration Layer: Architectural Foundation

Separation of Concerns Through Side-by-Side Architecture

The proposed architecture for side-by-side orchestration is based on SAP BTP and defines clear responsibilities between the system of record (SOR) functions and the payroll execution. The core SAP HR system is the SOR for employee master data, organization structures, and transactional data. SAP BTP would handle orchestration of payroll events, compliance checks, and interfacing between systems, with the payroll logic evolving independently of the underlying HR application. Studies on microservices and modularity show that well-defined separation of concerns results in fewer dependencies between components, making it possible to release them independently and reducing the need for coordination between modules [6]. A microservice architecture improves fault isolation as the failure is limited to one service. This also means it is easier to identify failures and recover from them, because the boundaries for data ownership, processing and integration between different pieces of the architecture are clear. In a side-by-side architecture, core HR systems generate payroll transactions via a well-defined interface, and an external payroll orchestration layer manages lifecycle, compliance checking, and execution of payroll processes. By decoupling payroll from the system of record, the core system is not burdened by payroll's functions, making it easier to maintain. Empirical studies show that effective decomposition leads to microservices that are 100 to 1000 lines of code. These microservices also align with the preferences of architects and developers as reported by 43% of them. This results in resilience and scale optimization benefits. Explicitly separated microservices also allow parallelized development and organizations widely report positive speed-up in development when cross-functional teams work on separate architectural components [6].

API-First Integration Patterns

API-first integration patterns define and enforce versioned APIs that specify well-defined interfaces between a core HR system and a suite of customer-facing or orchestration components. The APIs define the contracts between the loosely-coupled components for communication, event notification, and commands. The API layer conceals the underlying complexity. It exposes a consistent interface that can remain unchanged, even as forms of interaction change or as the core system or orchestration layer is

10.48047/jocaaa.2026.35.02.25

updated independently. Further research shows that the implementation of API-driven architectures can reduce average latency by 44%, failure rates by 89%, and malicious access attempts by 93%. This is dependent upon the proper governance and security of the API such as enforcing encryption in transit with TLS 1.3, and protecting the API with rate limiting, throttling and other protective measures from abuse. Protecting APIs by using a Web Application Firewall (WAF) and/or an API firewall provide perimeter-based protection. Schema validation is an example of contract-based enforcement. AI-based threat detection identifies anomalous behavior. Together, these controls help to achieve fault tolerance, observability, and data integrity in distributed systems, and scalable secure service integration [7].

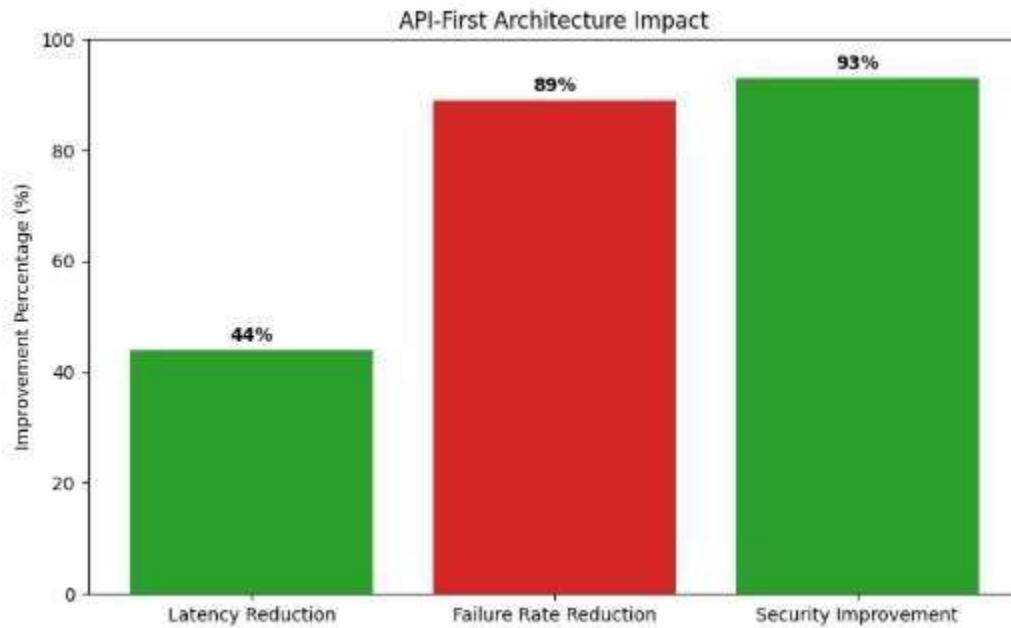


Figure 2: API-First Architecture Impact Features [7]

Asynchronous Event Propagation and Processing

The orchestration layer introduces asynchronous communication between events and users, decoupling event producers from event consumers, building scalable and fault tolerant payroll processing flows. Core HR systems produce payroll-related events - employee data changes, time events or retro payroll changes - on the event infrastructure without triggering execution or acknowledgement of payroll processing. The orchestration layer subscribes to the relevant event and coordinates processing with independently scalable services. Event-driven architectures provide near-linear increases in system throughput as more processing nodes are added to an asynchronous processing cluster [6]. The message queue-based event propagation mechanism can easily absorb bursts of processing load, while greatly smoothing out the peak load provided to upstream systems, all while maintaining sub-second event publication latency even under normal load levels [6].

This asynchronous processing model provides some architectural advantages. As publishing and processing the events are decoupled in time, the HR system is not directly affected by any processing spikes as a result of events. The failure of asynchronous processing operations can also be retried automatically, without reducing the overall reliability of the publishing system due to the eventual consistency rates and idempotent processing guarantees offered by these retry strategies [7]. The event-

10.48047/jocaaa.2026.35.02.25

processing model allows for complex payroll workflows that need to involve multiple systems, jurisdictions, or processing stages to be implemented while providing an auditable log of processing operations [7].

Externalized Compliance and Rule Evaluation Framework

Jurisdiction-Specific Rule Externalization

The proposed architecture offloads jurisdictional payroll rules and compliance logic from the main HR system and delegates processing to independent rule evaluation services. The independent rule evaluation services encapsulate country-specific tax rules, social insurance rules, and statutory compliance rules as independent rule engines that can be tested, modified, and deployed independent of the core HR system. Separating rule engines is also scalable from an empirical perspective: the number of rule evaluations was multiplied by 200, resulting in a 52.8% and 66.9% increase in query execution time (for a single user or five users concurrently). In addition, the overhead to the database's connection to the rule engines is minimal. One rule evaluation engine with 5,000 rules had a response time of approximately 5.21 seconds for one user. This fact makes externalized rule services scalable to regulatory complexity. These observations confirm the decoupling of compliance logic from core HR systems for higher adaptability at acceptable cost [8].

All externalized rules are versioned according to the payroll period and legislative context relative to the calculations being performed, and stored for audit and retroactive calculation (complying with the relevant rules in effect at a point in time, allowing traceability of compliance logic over time). For versioned rule systems, traceability of previous versions is useful in reducing auditing time, as organizations can recreate the compliance evaluations from any point in time accurately [8]. Separating the compliance rules from the payroll calculation rules makes the validation and certification processes faster, as jurisdiction-specific rules are validated in isolation and do not depend on the whole payroll system. Organizations that externalize rules also report reduced effort to certify compliance, and reduced certification costs across jurisdictional cycles due to reduced scope and reduced system complexity [8].

Component	Responsibilities	Versioning Model	Testing Scope	Deployment Independence
Core HR System	Employee master data; organizational structures; transactional records	Standard SAP versioning	Full system regression	Dependent on SAP release cycles
Rule Evaluation Services	Tax calculations; social insurance requirements; statutory compliance logic	Effective-dated versions with historical retention	Jurisdiction-specific validation only	Independent per jurisdiction
Orchestration Layer (BTP)	Event coordination; workflow management; validation aggregation	API version management	Integration contract testing	Independent microservice deployment
Audit Repository	Immutable logs; processing lineage; temporal metadata	Append-only with cryptographic verification	No functional testing required	Independent storage scaling
Compliance Workflows	Rule composition; exception handling; escalation procedures	Workflow definition versioning	Scenario-based validation	Independent workflow updates

Table 1: Rule Externalization Framework - Component Responsibilities [8]

Dynamic Rule Composition and Evaluation

The rule evaluation framework allows compliance checks to be dynamically composed during runtime based on employee profile, business context and payroll scenario. It allows fine-grained rule definition instead of monolithic compliance-checking routines, and invocation of these rules based on the context at runtime. The composability of rules provides possibilities for advanced compliance use cases, especially if multiple rules need to be satisfied at the same time but can be separated into different rule definitions. An analysis of rule composition patterns showed composable rule architectures with fine-grained rule components are more reusable than monolithic rule architectures as rule components can be reused across different compliance scenarios [9]. Reduction rules, previously shown to address the verification complexity of process models, can be used for the verification or engineering of such models. In both cases, the reduction rules are used to simplify process models while guaranteeing that the models are correct. Given that, reduction rules can effectively be used to avoid state space explosion in the model-checking phase for complex systems. However, instead of uniformly abstracting the underlying control graph, the principle of the approach presented here is that only the specific set of activities implicated by the rule under verification is respected. After all, the resulting state space can vary tremendously depending on the reduction rule employed for verification, which may in turn depend on the properties being verified [9].

The evaluation engine also provides standard rule execution, aggregation and exception handling for different execution contexts to ensure jurisdiction-specific rules are not mixed unintentionally. The diagnostic reports generated during the rule evaluation provide detailed data on rule applications, violations, and remediation to enable better transparency and auditability of compliance [9].

Compliance Validation as Orchestrated Workflows

The architecture implements compliance validation via orchestrated workflows that fire upon payroll events. The workflows evaluate applicable suites of compliance rules, consolidate their results, detect violations, and initiate remedial actions. These workflows run in parallel to the core payroll calculations, allowing compliance checks to be performed without the need to rerun calculations. Workflow orchestration ensures visibility into these processes via detailed execution histories and state. Each validation generates a trail of audit records of rule evaluation and the decision made and its result. This provides material for regulatory reporting and for audit and continuing enhancements to the compliance process. The workflow approach allows for exception and escalation steps and human review when the automated check encounters borderline or potentially non-compliant scenarios [9].

Retroactive Adjustment Handling and Audit Traceability

Event-Driven Retroactive Calculation Triggers

In support of retroactive payroll actions, the system processes the fewest number of payroll records (and only if necessary) by listening to events that are triggered when retroactive payroll calculations need to be processed, and identifying the employees and payroll periods that need to be processed. This selective processing of events reduces both the amount of processing required on live data streams and the time taken to produce the final result of retroactive calculation. Searching the literature on event-driven architectures, it is seen that recalculating events is less computationally intensive than reprocessing a batch population of data [10].

The metadata included in the retroactive adjustment triggers informs the payroll system of what data and time periods are affected by the retroactive adjustment, and lets payroll determine what needs to be recalculated first, group recalculations based on dependencies that span payroll periods, and prevent cascading recalculation cycles. To allow for the creation of large graph structures, it must be possible to manage these relationships in a way that is both purpose-oriented and scalable, as the target graphs may expand to thousands of nodes and edges. This requires a subsystem that allows for the manipulation of nodes, recalculation of dependencies, and history retention [11]. With an event-driven approach, retroactive changes can also be accommodated within specified time limits, thereby reducing disruption to normal payroll runs [10].

Processing Approach	Scope Determination	Impact on Regular Cycles	Cascading Prevention	Use Case
Batch Full Population	Recalculate entire employee population	High interference risk; resource contention	No automatic prevention; manual analysis required	Legacy monolithic systems
Event-Driven Targeted	Analyze change events; identify affected employees	Minimal interference; scheduled processing windows	Intelligent dependency management	Proposed architecture
Hybrid Selective Batch	Manual scope definition; subset processing	Moderate interference; requires coordination	Partial prevention through manual oversight	Transitional implementations

Table 2: Retroactive Adjustment Processing - Comparative Analysis [10, 11]

Immutable Audit Logs and Processing Lineage

The design of large heterogeneous information infrastructures and their constituent data components should be guided by an architectural vision that accommodates evolving dependencies and continual change, records rich metadata, maintains an accurate graph of dependencies, and translates incoming changes into structured assessments and suggestions for action. This is taken care of by a modular architecture with extensible storage, configurable control flows, and a lightweight metadata layer that grows incrementally, ensuring that the dependency graph that eventually emerges accurately reflects the behavior of the system, validated and extracted automatically [11]. Conversely, customary documentation-centric approaches cannot effectively model and manage a system's complex dependencies and downstream impacts. A unified graph model for requirements and their structural, semantic, behavioral and temporal interdependencies enables automated and scalable impact analysis and risk classification, as well as improved operational decision-making under the actual dynamics of evolving enterprise environments [11].

Audit logs maintain full context about the conditions of any given processing, such as the versions of rules used, system configuration states, environment variables and other information, to allow payroll calculations to be audited and discrepancies or disputes resolved. Contextualized audit trails with wide-ranging metadata considerably reduce dispute resolution effort and time compared to systems with minimal audit context. Audit logs are immutable and can provide strong evidence of compliance with regulations. These are also used during forensic investigations when there is a reason to suspect processing anomalies or errors. Organizations using immutable audit logging capabilities report having to spend less time remediating audit findings, and finding that auditors are more willing to accept proof as provided or generated by the system, without human review or supplemental evidence. Further, a complete audit trail allows root cause analysis using log parsing and log correlation, thus decreasing the time to detect processing defects [12].

Temporal Data Management and Historical Reconstruction

The orchestration layer also provides self-dating data management capabilities that fully preserve the history of payroll-relevant data in both levels of the system, appropriately dating the employee records, organizational structures, payroll business rules, and system configurations to allow for reconstruction of processing context over time. This also supports regulatory record-keeping requirements, as well as the operational requirements of retroactive payroll processing. Research into temporal database systems has shown that full versioning makes point-in-time reproduction highly accurate, and gives a consistent and reliable way to verify payroll calculations. Temporal data management employs delta-based versioning for effective storage and rapid query of state when viewing historical data [12].

Historical reconstruction reporter capabilities permit each payroll calculation to be reconstructed at any point in time using the data, rules and settings that were valid at that time. This capability is typically used for auditing, dispute resolution and retroactive validation. Automated data reconstruction avoids some of the effort needed to prepare for auditable recordings and to assemble complete data packages to be used for audits. Temporal replay also supports the analysis of the impact of changes by checking proposed changes to rules or business logic against past scenarios. This anticipatory verification allows for the avoidance of unintended effects in the production environment and the post-test repair burden by catching logic errors or edge cases earlier in the testing process [12].

Conclusion

Moving from monolithic SAP HR payroll architectures to event-driven orchestration-based architectures represents a model shift in the design, implementation and operating models for global payroll systems. By deploying a side-by-side orchestration layer using SAP BTP, payroll execution logic, compliance and cross-system orchestration for SAP ERP and SAP SuccessFactors can be decoupled from core HR systems and managed with data integrity and compliance. This pattern solves many of the use case limitations of customary implementations: limited extension points, the fragility of upgrades, and lack of transparently auditable evidence. Externalizing jurisdiction-specific compliance rules allows rapid adaptation to changing regulations. Finally, event-driven processing patterns provide the necessary scalability and resilience for enterprise-scale payroll processing. Strong audit logging and temporal data management capabilities can result in better regulatory and operational compliance, and can enable new payroll capabilities to be developed independently and at their own pace, while reducing technical debt and technical maintenance costs.

References

- [1] Adam Domagała et al., "Post-implementation ERP software development: Upgrade or reimplement," *Applied Sciences*, 2021. Available: <https://www.mdpi.com/2076-3417/11/11/4937>
- [2] Salman A. Baset, "Cloud SLAs: Present and Future," *ACM SIGOPS Operating Systems Review*, 2012. Available: <https://dl.acm.org/doi/pdf/10.1145/2331576.2331586>
- [3] Kiran Kumar Pappula, "Architectural Evolution: Transitioning from Monoliths to Service-Oriented Systems," *International Journal of Emerging Research in Engineering and Technology*, 2022. Available: <https://ijeret.org/index.php/ijeret/article/view/262>
- [4] Omar Al-Debagy and Peter Martinek, "A comparative review of microservices and monolithic architectures," In 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics, 2018. Available: <https://arxiv.org/pdf/1905.07997>
- [5] Narayan Ramasubbu and Chris F. Kemerer, "Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis," *Management Science*, 2016. Available: <https://pubsonline.informs.org/doi/abs/10.1287/mnsc.2015.2196>
- [6] Genc Mazlami, "Algorithmic extraction of microservices from monolithic code bases," Department of Informatics, 2017. Available: https://capuana.ifi.uzh.ch/publications/PDFs/16556_Masterarbeit.pdf
- [7] Jayant Bhat and Sundar Dilliraja, "Building a Secure API-Driven Enterprise: A Blueprint for Modern Integrations in Higher Education," *International Journal of Emerging Research in Engineering and Technology*, 2022. Available: <https://ijeret.org/index.php/ijeret/article/view/381>
- [8] Edward A. Stohr and Wayne Huang, "Business Rules Management: Implementation and Evaluation." Howe School of Technology Management Stevens Institute of Technology Working Paper, 2012. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2070881
- [9] Ahmed Awad et al., "Efficient Compliance Checking Using BPMN-Q and Temporal Logic," in *Business Process Management*, M. Dumas, M. Reichert, and M. C. Shan, Eds. Berlin: Springer, 2008. Available: https://link.springer.com/chapter/10.1007/978-3-540-85758-7_24
- [10] John Grigsby, Erik Hurst and Ahu Yildirmaz, "Aggregate nominal wage adjustments: New evidence from administrative payroll data." *American Economic Review*, 2021. Available: <https://www.aeaweb.org/articles?id=10.1257/aer.20190318>
- [11] Srujana Parepalli, "Mapping Critical Data Relationships to Enable Automated Evaluation of Operational Impact." *Journal of Artificial Intelligence, Machine Learning and Data Science*, 2021. Available: <https://doi.org/10.51219/JAIMLD/srujana-parepalli/646>

10.48047/jocaaa.2026.35.02.25

[12] Yimin Zhu et al. "Application of metadata modeling to dispute review report management," Journal of Civil Engineering and Management, 2010. Available: <https://www.tandfonline.com/doi/abs/10.3846/jcem.2010.55>