RESEARCH ARTICLE OPEN ACCESS

The Auth Shim: A Lightweight Architectural Pattern for Integrating Enterprise SSO with Standalone Open-Source Applications

Yuvraj Agrawal
Adobe Inc.

ABSTRACT
Open-source software (OSS) is widely adopted in enterprise settings, but standalone tools often lack native sup- port for protocols like SAML or OIDC, creating a critical security integration gap. This paper introduces and formalizes the "Auth Shim," a lightweight architectural pattern designed to solve this problem. The Auth Shim is a minimal, external proxy service that acts as a compatibility layer, translating requests from an enterprise Identity Provider (IdP) into the native session management mechanism of a target application. A key prerequisite for this pattern is that the target application must expose a programmatic, secure administrative API. We present a case study of the pattern's implementation at Adobe to integrate a popular OSS BI tool with Okta SAML, which enabled automated Role-Based Access Control (RBAC) via IAM group mapping and eliminated manual user provisioning. By defining its components, interactions, and production deployment considerations, this paper provides a reusable, secure, and cost- effective blueprint for integrating any standalone OSS tool into an enterprise SSO ecosystem, thereby enabling organizations to embrace open-source innovation without compromising on security governance.

Keywords — Architectural Pattern, Auth Shim, RBAC, IAP, IAM, Zero Trust, SSO, Open Source Software, SAML, Nginx, Docker.

I. Introduction

The adoption of open-source software (OSS) is a cornerstone of modern software engineering strategy [1]. Enterprises leverage a vast ecosystem of standalone OSS tools, but a recurring challenge hinders their secure deployment: these tools frequently lack native support for enterprise authentication protocols like SAML or OIDC. This forces organizations into a dilemma: purchase expensive enterprise licenses solely for SSO, accept the security risks of manual account management, or abandon the tool altogether.

This paper argues for a fourth option by formalizing a reusable architectural pattern: the **Auth Shim**. The term *shim* is used deliberately to denote a minimal component that provides a compatibility layer between an application's internal session management and a standardized external authentication system. The Auth Shim is a specific, lightweight implementation of the broader **Identity-Aware Proxy (IAP)** pattern, tailored for integrating a single application with minimal operational overhead.

Our central thesis is that the Auth Shim pattern provides a secure and efficient solution to this common integration problem. We make the following contributions:

1) We formally define the Auth Shim pattern and present a comprehensive architecture diagram.
2) We present a detailed case study of the pattern's implementation at Adobe to integrate a popular open-source BI tool with Okta SAML.
3) We provide a detailed comparative analysis against alternatives, including full IAPs and open-source proxies, evaluating features, complexity, and failure recovery behavior.
4) We provide a blueprint for a production-grade deployment, including a formal threat analysis and a research roadmap for a reusable implementation.

II. Background and Related Work

The Auth Shim pattern builds upon established security principles and relates to a body of existing work in identity management and secure software architecture.

A. Zero Trust, IAPs, and Modern Enterprise SSO

The zero-trust model, first articulated by Kindervag [2], mandates that no user or device is trusted by default. This model was operationalized at scale by Google's **BeyondCorp** [3], which introduced the IAP as a core component. An IAP functions as a central gateway, enforcing access policies at the application edge. This aligns with the modern security trend of treating **identity as the new perimeter**. As corporate data and services are distributed across cloud and on-premise environments, the traditional network-based security model is no longer sufficient. Instead, access is granted based on user identity and device context, verified at every request. Recent academic work in venues like IEEE S&P and USENIX Security continues to explore microservice perimeter patterns and access governance models, with particular focus on the challenges of enforcing dynamic policies in distributed systems.

B. Identity Federation and Proxy Patterns

The core function of the Auth Shim—translating between security domains—is a form of identity federation [4]. Architecturally, it is an application of the classic Proxy and Adapter design patterns [5], adapting requests from a modern authentication provider to a legacy or standalone application's expected interface. Existing work has explored similar concepts for legacy systems [6] and API gateways [7], but has not formalized a lightweight pattern specifically for the modern OSS-in-the-enterprise context where custom authorization is a key driver [8]. Recent studies on the security of SSO protocols have highlighted the need for careful implementation at the integration point, reinforcing the need for well-defined patterns like the Auth Shim [9].

C. Novelty of the Auth Shim Pattern

The novelty of the Auth Shim does not lie in the invention of a new proxy technology. Rather, its contribution is the **formalization, synthesis, and specific application** of these existing concepts to address a common and underserved problem. Its novelty arises from:

- **Addressing a Niche:** The pattern is purpose-built for scenarios where a full-featured IAP is too complex and a commercial plugin is too inflexible or costly. It fills this pragmatic gap.
- **Formalization as a Reusable Blueprint:** By naming the pattern and defining its participants, interactions, and design rationale, this paper transforms a common ad-hoc fix into a documented, reusable, and secure architectural solution.
- **Emphasis on Just-in-Time Authorization:** A defining characteristic is the tight integration with the target application's API to perform just-in-time RBAC synchronization. This pattern should not be confused with a simple authenticating reverse proxy. While a basic *auth_request* module can verify a user's identity, it lacks the core capability of the Auth Shim: the *write path* integration with the target application to perform user provisioning and role synchronization, which is essential for seamless operation and security.

D. Comparison with Open-Source Authentication Proxies

Several popular open-source projects, such as *oauth2-proxy* or *keycloak-gatekeeper*, provide authentication proxies for web applications. These tools are excellent at enforcing authentication—they can integrate with an IdP and ensure that only valid users with specific roles or groups can access a downstream service. A reverse proxy using a basic *auth_request* module (like in Nginx) achieves a similar outcome.

The Auth Shim's novelty is its focus on the deeper integration required for authorization and user lifecycle management. While an *auth_request* can verify a user's identity (the 'read' path), it is agnostic to the application's internal state. It cannot provision users, synchronize granular permissions, or deactivate accounts within the application. The Auth Shim, in contrast, is fundamentally about this *write path* integration. It uses the identity established during authentication to actively manage the user's lifecycle and permissions inside the target application via its API. This is the key capability that eliminates manual account management and ensures permissions are always consistent with the central IdP, a gap that simple authenticating proxies do not address.

III. The Auth Shim Architectural Pattern

The Auth Shim pattern is defined by its intent, structure, and participants. Its core purpose is to provide an external authentication and authorization layer for a standalone application that lacks native enterprise SSO support.

A. Formalization

To frame our solution in established software engineering terms, we define it using the Gang of Four (GoF) style.

- **Pattern Name:** Auth Shim
- **Intent:** Provide a secure, external authentication and authorization layer for a standalone application that lacks native enterprise SSO support.
- **Applicability:** Use the Auth Shim pattern when an application must be integrated with an enterprise SSO system and a full IAP is considered overkill or a commercial plugin is infeasible. The key prerequisite is a programmatic interface on the target application to manage users and sessions.
- **Structure:** As illustrated in the comprehensive architec-

ture diagram in Fig. 1.

- **Participants:** *Reverse Proxy*, *Auth Shim Service*, *Target Application*, *Enterprise IdP*.
- **Consequences:** Decouples authentication logic from the application. It centralizes authorization logic, but introduces a new component that must be maintained and deployed with high availability.

B. *Core Components and Responsibilities*

The Auth Shim Service is not a monolith; it is a composite of several logical components, each with a distinct responsibility. This modular design enhances maintainability and clarifies the service's internal workings.

- **SAML Handler:** Manages the SAML 2.0 protocol flow. Its duties include initiating authentication requests, processing and validating signed SAML responses from the IdP, and securely extracting user attributes (e.g., email, group memberships) from the assertion.
- **User Manager:** Handles the lifecycle of users within the target application. It automates provisioning by creating new user accounts for first-time logins and ensures user information is kept current.
- **RBAC Engine:** Implements the core authorization logic. It translates group memberships received from the IdP into specific roles or permissions within the target application, based on a configurable mapping. It is responsible for adding and revoking permissions to enforce Just-in-Time access.
- **Session Bridge:** Acts as the final link to the applica- tion. After successful authentication and authorization, it communicates with the target application's API to create a valid user session, receiving a session token or cookie in return.
- **API Client:** A dedicated client responsible for all communication with the target application's administrative API. It uses a secure, pre-configured token to perform privileged actions like creating users, managing group memberships, and initiating sessions.

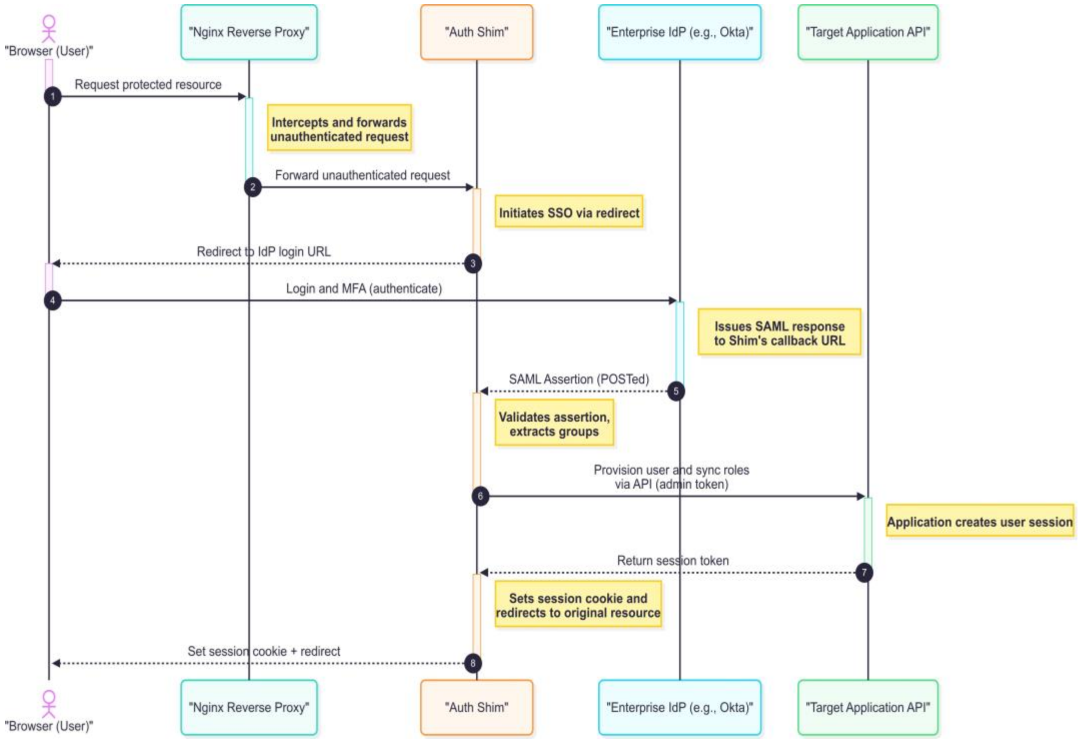

"Browser (User)"
"Nginx Reverse Proxy"
"Auth Shim"
"Enterprise IdP (e.g., Okta)"
"Target Application API"
1
Request protected resource
Intercepts and forwards unauthenticated request
2
Forward unauthenticated request
Initiates SSO via redirect
3
Redirect to IdP login URL
4
Login and MFA (authenticate)
Issues SAML response to Shim's callback URL
5
SAML Assertion (POSTed)
Validates assertion, extracts groups
6
Provision user and sync roles via API (admin token)
Application creates user session
7
Return session token
Sets session cookie and redirects to original resource
8
Set session cookie + redirect
"Browser (User)"
"Nginx Reverse Proxy"
"Auth Shim"
"Enterprise IdP (e.g., Okta)"
"Target Application API"

Fig. 1. Comprehensive Architecture and Request Flow of the Auth Shim Pattern. This diagram illustrates the end-to-end authentication process: (1) An unauthenticated user requests a resource from the application. (2) The Nginx reverse proxy intercepts the request and routes it to the Auth Shim. (3) The Auth Shim initiatezs an SSO flow, redirecting the user to the Enterprise IdP. (4) The user authenticates with the IdP. (5) The IdP issues a signed SAML assertion and sends it to the Auth Shim's callback URL. (6) The Shim validates the assertion, extracts user attributes (like group memzberships), and uses an admin token to communicate with the Target Application's API. (7) The Shim provisions the user and synchronizes their roles via the API. (8) The application creates a session and returns a session token. (9) The Shim sets the session token in the user's browser and redirects them to the originally requested resource.

IV. A Taxonomy and Comparison of SSO Integration Patterns

To position the Auth Shim correctly, we propose a decision taxonomy for selecting an SSO integration pattern, shown in Fig. 2. The Auth Shim is the logical choice when native support is absent and custom logic for authorization is a primary driver, making a commercial plugin unsuitable and a full IAP too complex.

To position the Auth Shim correctly, we first provide a high-level comparison of common SSO integration approaches (Table I), followed by a detailed analysis grounded in operational, architectural, and maintainability considerations.

TABLE I
High-Level Comparison of SSO Integration Approaches

Criterion	Auth Shim (Chosen)	Full IAP (e.g., Pomerium)	Commercial Plugin
Dev Effort	~150 LoC + Config	0 LoC + ~40 lines YAML	0 LoC + UI Config
Custom Logic	High Flexibility	Medium (plugins)	Low to None
Audit Surface	Small (focused)	Large (framework)	Vendor-dependent
Maintainability	High (known stack)	Medium (new dependency)	Low (vendor-managed)

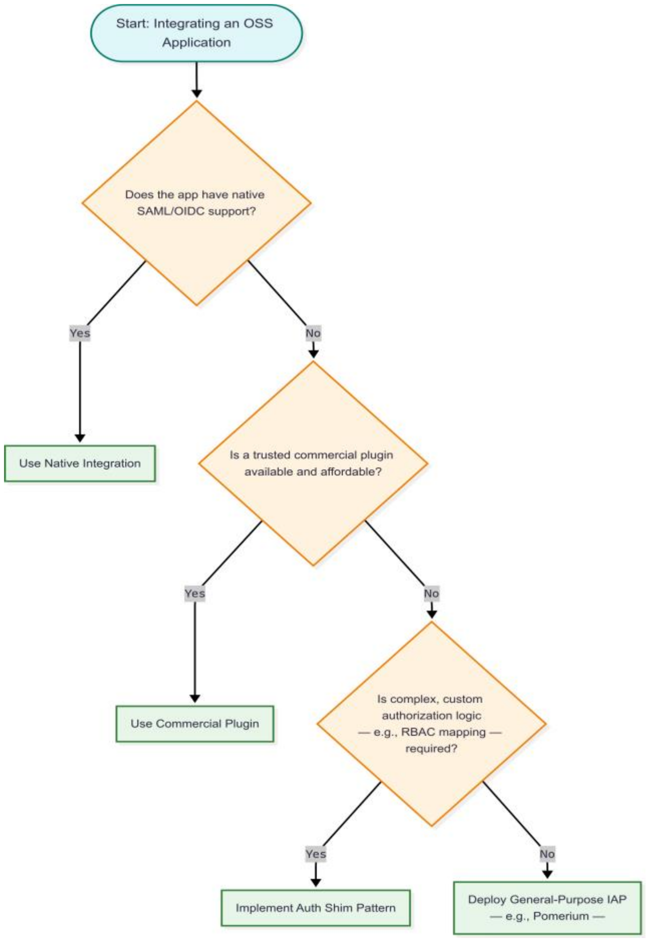

Start: Integrating an OSS Application
Does the app have native SAML/OIDC support?
Yes
No
Use Native Integration
Is a trusted commercial plugin available and affordable?
Yes
No
Use Commercial Plugin
Is complex, custom authorization logic — e.g., RBAC mapping — required?
Yes
No
Implement Auth Shim Pattern
Deploy General-Purpose IAP — e.g., Pomerium —

Fig. 2. A Decision Tree for SSO Integration Approaches.

A. Detailed Comparative Analysis

While Table I provides a quick overview, a deeper qualitative comparison is necessary to understand the trade-offs between approaches.

Full Identity-Aware Proxies (IAPs), such as Pomerium [10] or Ory Oathkeeper [11], offer robust enterprise-grade features: context-aware policies, integration with modern IdPs, TLS enforcement, and protocol support for OAuth2, SAML, and OIDC. However, they are often over-engineered for scenarios where only a single standalone application needs SSO. These solutions introduce additional dependencies—such as policy engines, sidecars, or Redis backed session stores—that increase deployment complexity and the system's attack surface. While maintainable in large-scale microservices environments, they may represent unnecessary overhead for smaller teams or use cases.

Commercial Plugins, offered by proprietary platforms (e.g., Tableau Server, Grafana Enterprise, etc.), are often easy to configure and vendor-supported, but they lack flexibility and visibility. Custom workflows such as just-in-time (JIT) user provisioning, attribute-based access control, or non-standard SAML assertions are rarely supported. Additionally, their behavior is opaque and recovery paths are dependent on vendor patches, creating risk during upgrades or protocol changes [12].

The Auth Shim pattern occupies a pragmatic middle ground. It provides the extensibility of custom development with the simplicity of deployment. Unlike an IAP, the shim directly integrates with the application's administrative APIs, enabling fine-grained control over user provisioning, role mapping, and session management. It remains lightweight—typically under 200 lines of code—and deploys using standard infrastructure components (e.g., Nginx, Docker, Python). This makes it accessible to DevOps teams without requiring knowledge of complex policy DSLs or maintaining external policy stores.

B. Failure and Recovery Behavior

Beyond feature comparisons, it is important to assess how each solution behaves under failure conditions. A detailed comparison of typical failure modes and their implications on reliability and recovery is provided in the appendix (Table V). The Auth Shim's statelessness and narrow operational footprint enhance both fault isolation and resilience. Its simplicity reduces dependencies, allowing teams to focus on maintaining the target application and IdP, without introducing new points of failure.

V. Case Study: Implementation and Deployment

We implemented the pattern at Adobe to integrate a leading open-source BI tool with Okta. While the Auth Shim pattern is applicable to both SAML and OIDC protocols, this paper's case study focuses on a SAML-based integration. The high-level orchestration logic is conceptually shown in Fig. 3.

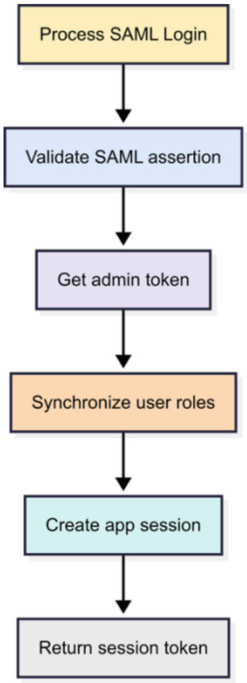


Fig. 3. Conceptual flowchart of the Shim's core logic, executed upon receiving a SAML response.

A. Containerized Deployment

The entire stack is containerized using Docker [14]. The core logic is implemented in a service codenamed *auth-shim*, which acts as the 'Auth Shim Service' described in the pattern's formalization. An abridged *docker-compose.yaml* file, which defines the services and their dependencies, is available in the appendix (Fig. 6).

B. Hardened Reverse Proxy Configuration

Security is enforced at the edge by a hardened Nginx configuration. Key features include HTTPS enforcement, strong TLS ciphers, security headers, and DoS mitigation. The use of *auth_request* is central to the pattern, delegating session validation for every incoming request to the Auth Shim service. A production-grade Nginx configuration file is provided in the appendix (Fig. 4).

C. Key Design Rationale

1) Stateless Service Design: A deliberate decision was made to design the Auth Shim as a **completely stateless service**. This allows the shim to be scaled horizontally without requiring a shared session store (like Redis), simplifying the architecture.

2) API-Based Interaction: We consciously chose to interact with the target application via its official REST API. This creates a clean, decoupled architecture that is resilient to upgrades.

D. Role-Based Access Control (RBAC) via IAM Groups

The IdP (Okta) is configured to send a 'groups' attribute in the SAML assertion. The Auth Shim then performs a full synchronization on each login, ensuring a user's permissions are always an exact reflection of their status in the central IdP. The step-by-step logic for this synchronization is visualized in the appendix (Fig. 10).

To enhance maintainability, the mapping logic is externalized into a configuration file, decoupling authorization rules from business logic. Illustrative Python code and an example YAML configuration can be found in the appendix (Figs. 7 and 8).

E. Authentication Flow Orchestration

The full end-to-end authentication process is detailed in the sequence diagram in the appendix (Fig. 5). The main orchestration logic is shown in the Python code snippet in the appendix (Fig. 9).

F. Detailed User Journeys

To better illustrate the pattern's behavior, sequence diagrams for a first-time user and a returning user are presented in the appendix (Fig. 11 and Fig. 12 respectively), highlighting both just-in-time provisioning and fast-path validation.

VI. Evaluation

A. Transformation of Operational Overhead

Prior to the Auth Shim, integrating the BI tool was characterized by significant operational friction, including manual, ticket-based user provisioning, which violated the Principle of Least Privilege and created a heavy burden for compliance audits.

The pattern's impact was measured by comparing the system pre- and post-integration, as shown in Table II. The transformation eliminated manual toil, saving an estimated **10 hours of engineering work per week**. This translates to an estimated annual saving of over **$35,000** in operational costs for a single application integration,[1] while drastically improving the organization's security posture.

TABLE II
PRE- VS. POST-SHIM IMPACT METRICS

Metric	Pre-Shim	Post-Shim	Delta
Weekly Maintenance	~10 hrs	<1 hr	-90%
User Onboarding	Manual, 1 day	Instant	Automated
Role Management	Manual Tickets	Automated	100% via IAM
MFA Support	No	Yes (via IdP)	Compliant
Audit Coverage	Low (App only)	Full (IdP + App)	Governance Gain

B. Performance, Scalability, and Resource Consumption

The Auth Shim is designed to be lightweight. Its performance impact must be considered in two scenarios. For **authenticated requests**, the Nginx proxy adds a negligible pass-through latency of **<5ms (p95)**. For the **initial login**, a one-time latency of **~850ms (p95)** is introduced. These benchmarks were conducted in a staging environment representative of our production setup (e.g., AWS c5.large instances) under simulated user load. The p95 latency figures represent the 95th percentile from a sample of 10,000 login requests.

The entire stack has a minimal resource footprint. The Nginx container consumes ˜10-50MB of RAM, while the stateless Auth Shim service (e.g., in Python) typically requires ˜50-100MB of RAM. The entire pattern can operate comfortably with less than 200MB of RAM and a fraction of a single CPU core, making it highly efficient.

TABLE III
PERFORMANCE BENCHMARK RESULTS

Metric	Result
Pass-through Latency (p95)	<5ms
Initial Login Latency (p95)	~850ms (incl. IdP & API calls)
Max Throughput (Logins)	200 logins/sec (bottlenecked by app API)

VII. DISCUSSION

A. Security Considerations

The security of the system is multi-layered. It relies on a hardened infrastructure, a robust protocol flow, and the secure implementation of the shim service itself, which must perform tasks like XML parsing and signature validation using well-vetted libraries to prevent vulnerabilities like XML External Entity (XXE) attacks. Beyond this, the pattern aligns with zero-trust principles by enabling the **Principle of Least Privilege** through just-in-time RBAC synchronization.

1) Token Management: The administrative API token (*APP_ADMIN_TOKEN*) is a highly sensitive secret. In production, it is managed via a secure vaulting system (e.g., HashiCorp Vault) and injected into the container at runtime.

2) SAML Security: The protocol-level security relies on strict validation of the SAML assertion. This includes mandatory signature verification to prevent tampering, certificate validation to ensure the assertion originates from the trusted IdP, and timestamp checks to mitigate replay attacks.

3) Session and Network Security: At the transport layer, all communication is secured using TLS 1.2+. Session cookies are flagged as *HttpOnly* and *Secure* to prevent client-side script access and ensure they are only transmitted over HTTPS. The reverse proxy provides an additional layer of defense through rate limiting and security headers.

4) Threat Analysis: We conducted a threat analysis using the STRIDE model [15], summarized in Table IV.

TABLE IV
STRIDE THREAT ANALYSIS OF THE AUTH SHIM PATTERN

Category	Threat Example	Mitigation
Spoofing	Forged SAML assertion	Mandatory IdP signature validation
Tampering	Modified role claims	SAML assertion signature covers attributes
Repudiation	Disputed login event	Centralized IdP and Shim logging
Info. Disclosure	Leaked admin token	Secure vault storage, network isolation
Denial of Service	Login endpoint flood	Nginx rate limiting on auth endpoints
Elev. of Privilege	False group claim	Shim is source of truth for role mapping

5) Potential Failure Scenarios:

- **IdP Group Claim Desynchronization:** If the 'groups' attribute is misconfigured or removed from the IdP's SAML assertion, the shim might interpret this as a user belonging to no groups, incorrectly revoking their permissions. Mitigation involves defensive code in the shim to validate the presence of the claim and fail the login if it is missing, alongside monitoring to detect such anomalies.
- **Target Application API Downtime:** If the target application's API becomes unavailable during a login attempt, the shim cannot provision the user or create a session, causing the login to fail. Mitigation includes robust health checks (as detailed in the appendix, Fig. 6), designing the stack for high availability, and providing clear, user- friendly error messages that distinguish a system outage from an authentication failure.

6) Authorization and Role Security: Beyond authentication, the authorization logic itself must be secure. The system prevents unauthorized privilege escalation by ensuring that role synchronization is a one-way flow from the central IdP to the application. The application's administrative token is used only

[1] This calculation uses an illustrative, conservative, fully-loaded rate of $75/hour for a DevOps engineer. While this rate varies by region and orga- nization, it demonstrates the significant order-of-magnitude savings achieved by automating manual, high-frequency tasks.

to enact the changes dictated by the IdP's SAML assertion; the application itself cannot grant permissions that are not present in the assertion. All role changes are implicitly logged by the IdP and can be audited centrally.

B. Generalizability and Limitations

A preliminary applicability analysis suggests that the Auth Shim pattern is highly suitable for many mature OSS tools. A 'High' fit indicates that the tool exposes a documented, stable administrative REST API for user and session management and has a distinct need for granular, group-based role mapping. However, its limitations must be acknowledged:

- **Requires a Quality Programmatic API:** The pattern's effectiveness is entirely dependent on the target application offering a stable, secure API for user and session management. An ideal API is not only available but also idempotent, well-documented, and not subject to overly aggressive rate-limiting.
- **Stateful Session Complexity:** The shim is most effective with applications that support stateless session tokens (e.g., JWTs) or have a simple API call to create a session.
- **Versatile Deployment Topologies:** While this paper focuses on a reverse proxy implementation, the core Auth Shim service is flexible. It can be adapted to other deployment topologies, such as a sidecar container in a Kubernetes pod or as middleware in an API Gateway, extending its applicability to microservices environments.
- **Not a Universal IAP:** The Auth Shim is deliberately lightweight. It lacks advanced features like device posture checks or contextual access policies. For enterprise-wide zero trust, a general-purpose IAP is more appropriate.
- **Introduces a Managed Component:** Though minimal, the shim is a critical component in the authentication flow. It is crucial to understand that while the shim itself is highly scalable, it cannot fix scalability limitations in the target application. Its reliability is paramount, and it must be deployed, monitored, and maintained with high availability in mind.
- **Dependency on Target Application Performance:** While the shim itself is highly scalable, it cannot fix scalability limitations in the target application. Its reliability is paramount.

VIII. Future Work: A Research Roadmap

Our future work focuses on lowering the barrier to adoption by implementing the concepts presented in this paper as a reusable, open-source tool. This roadmap is divided into three phases.

A. The Auth Shim Scaffold

The first step is to refactor our implementation into a generic **'Auth Shim Scaffold.'** The goal is to create a template where a developer only needs to implement a well-defined '*ApplicationConnector*' interface with methods like '*createUser*', '*createSession*', and '*syncRoles*'. The scaffold would consist of a generic core service to handle the SAML/OIDC protocol flow and a defined *ApplicationConnector* interface. A developer's workflow would be reduced to implementing methods like createUser, createSession, and syncRoles with the specific API calls for their target application. Future iterations could extend this interface to support Attribute Based Access Control (ABAC), where the connector could make more dynamic authorization decisions based on rich user attributes (e.g., project codes, geographic location) passed in the SAML assertion, not just group membership. The primary challenge in developing this scaffold will be creating a flexible *ApplicationConnector* that can accommodate the diverse session management mechanisms of different OSS tools (e.g., cookie-based sessions vs. returning a JWT) and abstracting their unique API conventions into a standardized interface.

B. Extensibility and API Adaptation

The scaffold will be designed with an explicit extensibility model. The 'ApplicationConnector' interface will be designed to handle variations in target application behavior. For instance, it will include optional methods to handle asynchronous session creation or strategies for gracefully backing off when faced with API rate limits, allowing the shim to adapt to both simple and complex application APIs.

C. Validation Against Diverse Applications

The scaffold's effectiveness will be validated by implementing connectors for 2-3 popular open-source applications. This will test the plug-in model's flexibility and provide the community with concrete, working examples, demonstrating its utility beyond the single case study presented here. Future iterations could also extend this interface to support Attribute-Based Access Control (ABAC), making more dynamic authorization decisions based on rich user attributes passed in the SAML assertion.

IX. Conclusion

This paper introduced and formalized the **Auth Shim**, a lightweight architectural pattern that brings enterprise-grade SSO and automated RBAC to standalone OSS tools. Our case study integrating a BI tool with Okta SAML validated its effectiveness in drastically reducing engineering effort and improving security governance. Through a detailed comparative analysis, we have shown that it provides a pragmatic middle ground between expensive enterprise licenses, overlysimplistic authentication proxies, and overly-complex general-purpose IAPs. Ultimately, the Auth Shim offers a pattern for enterprises to systematically reduce security gaps in their software portfolio, one application at a time.

REFERENCES

[1] OpenLogic by Perforce. *2023 State of Open Source Report*. 2023. Available at: https://www.openlogic.com/resources/2023-state-of-open-source-report (Accessed: 2023-11-15).

[2] John Kindervag. No More Chewy Centers: Introducing The Zero Trust Model Of Information Security. *Forrester Research*, Sep 2010.

[3] Rory Ward and Betsy Beyer. BeyondCorp: A New Approach to Enterprise Security. *login: The USENIX Magazine*, 39(6), Dec 2014.

[4] P. A. Karger and J. S. Park. Identity Federation: Issues and Architectures. In *2006 IEEE Security and Privacy Workshops*, pp. 11–11, 2006. doi:10.1109/SPW.2006.19.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[6] Christoph Arndt. Secure and Scalable Legacy IAM Integration. *IEEE Software*, 39(1):89–94, 2022. doi:10.1109/MS.2021.3119102.

[7] C. Pahl, A. Jamshidi, and O. Zimmermann. Architectural Patterns for Secure and Trustworthy API Gateways. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 123–12307, 2018. doi:10.1109/ICSA.2018.00021.

[8] S. Hassan and G. Russello. A Micro-proxy for Enforcing Attribute- Based Access Control in Microservices. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, pp. 191– 200, 2008. doi:10.1145/1377836.1377865.

[9] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and G. Pellegrino. On the Security of Single Sign-On Protocols in the Wild. In *2015 IEEE Symposium on Security and Privacy (SP)*, pp. 611–628, 2015. doi:10.1109/SP.2015.43.

[10] Pomerium. *Pomerium Documentation*. 2024. Available at: https://www. pomerium.com/docs/ (Accessed July 2025).

[11] ORY. *ORY Oathkeeper Docs*. 2024. Available at: https://www.ory.sh/ oathkeeper/ (Accessed July 2025).

[12] OpenLogic. Challenges with Commercial SSO Plugins. 2023. Avail- able at: https://www.openlogic.com/blog/sso-plugin-limitations (Ac- cessed July 2025).

[13] OAuth2 Proxy. *OAuth2 Proxy Documentation*. 2024. Available at: https://oauth2-proxy.github.io/oauth2-proxy/ (Accessed July 2025).

[14] Docker, Inc. Docker Documentation. 2023. Available at: https://docs. docker.com/ (Accessed: 2023-11-15).

[15] Microsoft. The STRIDE Threat Model. 2023. Available at: https://learn.microsoft.com/en-us/azure/security/develop/ threat-modeling-tool-threats (Accessed: 2023-11-16).

APPENDIX

A. Detailed Failure and Recovery Comparison

This appendix provides supplementary materials referenced in the main body of the paper, including detailed comparison tables, implementation artifacts, and process diagrams.

B. Implementation Artifacts (Configurations and Code)

C. Process and User Journey Diagrams

TABLE V
IN-DEPTH COMPARISON OF FAILURE AND RECOVERY CHARACTERISTICS[3]

		Solution	**Failure Scenario** **Observed Behavior** **Recovery Characteristics**
Auth Shim	Target application's API is unavailable (e.g., restart or timeout).	Login fails gracefully. No session is created; no state is corrupted. Shim returns error to user.	Stateless recovery — shim resumes instantly once the app API is back. Health checks ensure traffic gating.
	SAML assertion missing group attribute.	User appears to have no RBAC mapping. Access is denied.	Defensive logic rejects login; alerts can be triggered via logging. No side effects on user store.
	Shim container crashes. restarted	Incoming requests timeout or fail at proxy.	Container can be auto- (e.g., Docker/Podman). No persistent state is lost.
Full IAP (e.g., Pomerium)	Policy backend (e.g., Redis or OPA) unavailable.	All authorization checks fail. Logins are blocked, sessions revoked.	Recovery requires backend service restoration and potential cache sync. Failure cascades across all apps.
	Configuration error in access policy.	Users locked out or improperly granted access across multiple services.	Manual rollback or policy redeploy needed. Risk of systemic misconfiguration.
	TLS misconfiguration or expired certs.	IAP rejects inbound or outbound requests. Entire flow blocked.	Complex recovery — certificate regeneration, redeploy required. Centralized fault domain.
Commercial Plugin	Plugin fails after application version upgrade.	Login screen may break, or bypass SSO entirely. Behavior is unpredictable.	Recovery gated on vendor patch or rollback. Limited user visibility or control.
	IdP metadata changes (e.g., new certificate).	SAML validation fails silently or partially.	Requires manual intervention. Logs often hidden behind vendor abstraction.
Auth Proxy (e.g., oauth2-proxy)	Network issue between proxy and IdP.	New logins fail. Existing sessions continue (if cookies valid).	Stateless proxy auto-recovers once network is restored. Limited visibility into login errors.
	Session cookie store lost or invalid.	All sessions become unauthenticated; re-authentication loop may occur.	Depends on browser/client. Limited application-side debugging.

[3] These scenarios are based on real-world behavior observed during internal deployments and informed by public documentation of open-source identity-aware proxies [10], [13].

Nginx Configuration for Security, Routing, and Session Validation

```
# Rate limiting to mitigate DoS attacks
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;

# HTTP to HTTPS redirect
server {
    listen  80;
    server_name your-domain.com;
    return  301  https://$server_name$request_uri;
}

# HTTPS server with auth validation
server {
    listen 443 ssl http2;
    server_name your-domain.com;

    ssl_certificate /etc/nginx/ssl/cert.pem;
    ssl_certificate_key /etc/nginx/ssl/key.pem;
    ssl_protocols TLSv1.2 TLSv1.3;
    add_header Strict-Transport-Security "max-age=31536000";

    # Internal endpoint for the Auth Shim to validate the session cookie
    location = /auth/validate {
        internal;
        proxy_pass  http://auth-shim:8080/validate-session;
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";
        proxy_set_header X-Original-Cookie $http_cookie;
    }

    # All incoming traffic is subject to session validation
    location / {
        limit_req zone=mylimit burst=20;

        auth_request /auth/validate;
        error_page 401 = @redirect_to_login; # If session is invalid, redirect

        # If validation is successful, proxy to the target application
        proxy_pass  http://target-app:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # Named location to handle the redirect to the Auth Shim's login endpoint
    location @redirect_to_login {
        return 302 http://auth-shim:8080/;
    }
}
```

Fig. 4. A production-grade Nginx configuration demonstrating HTTPS enforcement, rate limiting, and the critical auth_request directive, which delegates session validation to the Auth Shim service for every request.

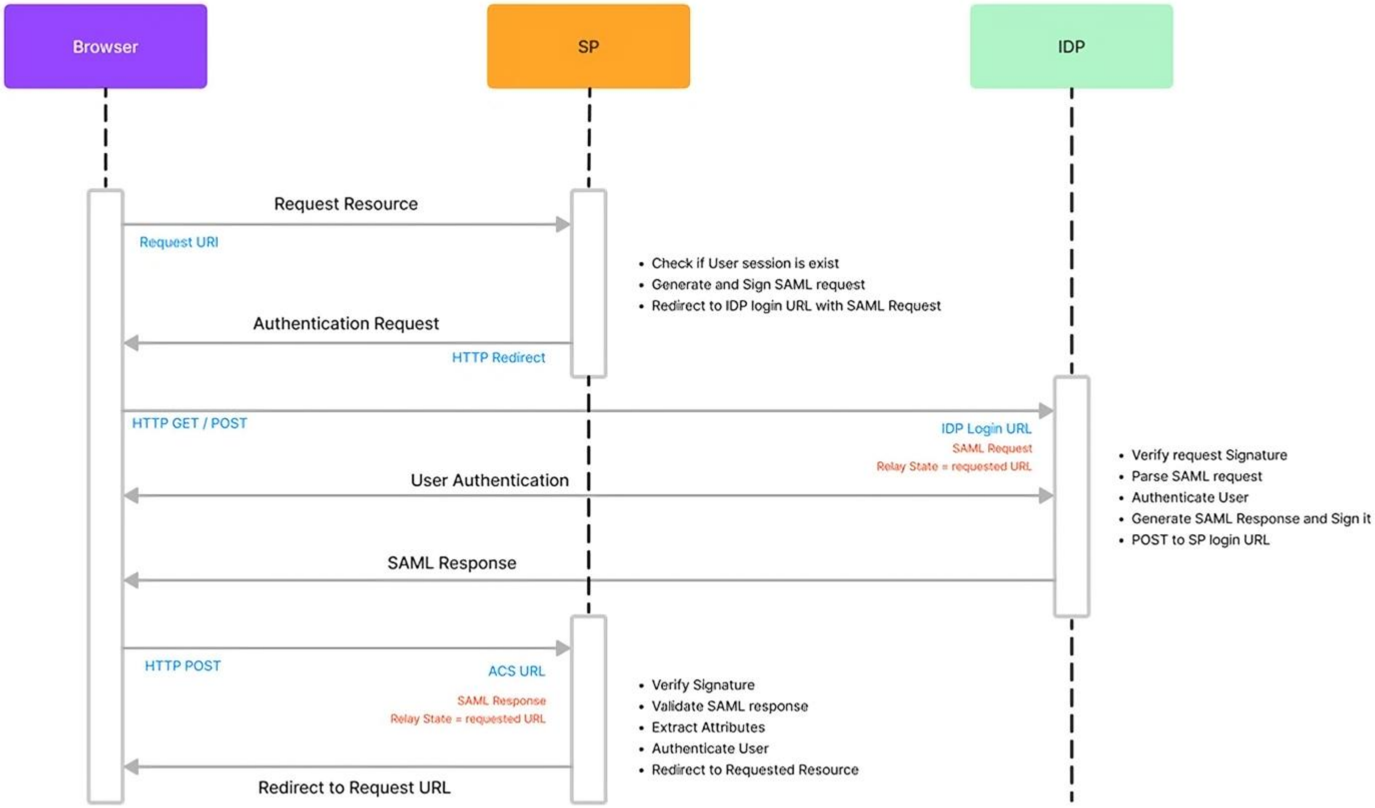
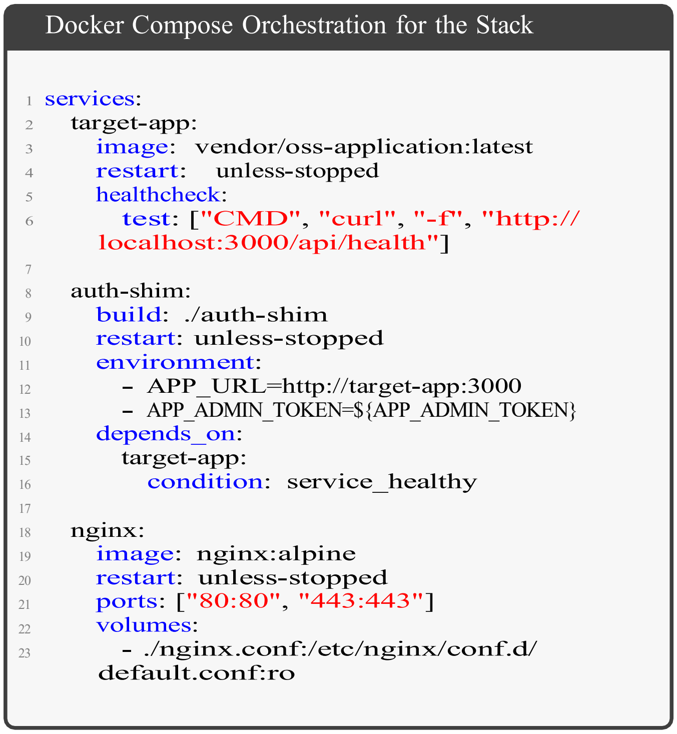


Fig. 5. Detailed SAML sequence diagram illustrating the complete authentication flow brokered by the Auth Shim.



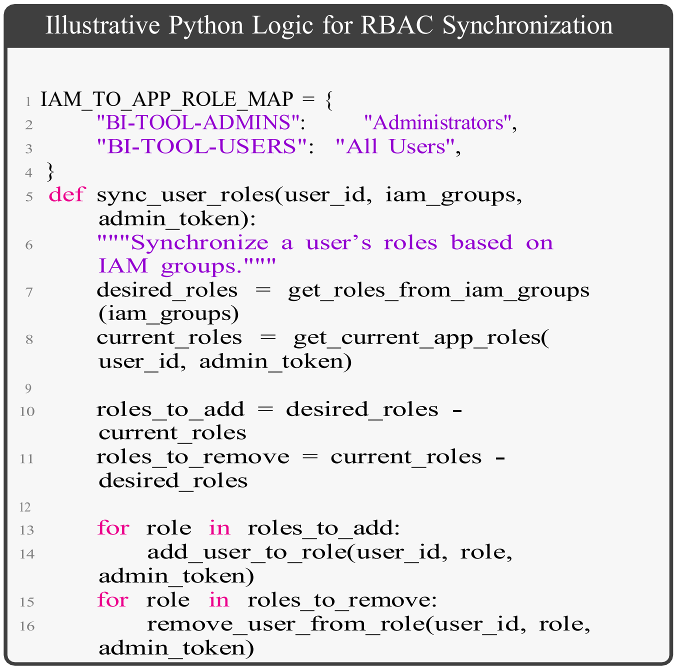
Docker Compose Orchestration for the Stack

```yaml
services:
  target-app:
    image:  vendor/oss-application:latest
    restart:   unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/api/health"]

  auth-shim:
    build:  ./auth-shim
    restart: unless-stopped
    environment:
      - APP_URL=http://target-app:3000
      - APP_ADMIN_TOKEN=${APP_ADMIN_TOKEN}
    depends_on:
      target-app:
        condition:  service_healthy

  nginx:
    image:  nginx:alpine
    restart:  unless-stopped
    ports:  ["80:80", "443:443"]
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro
```

Fig. 6. Abridged docker-compose.yaml defining the three-service stack. The depends_on clause with a service_healthy condition ensures that the shim service only starts after the target application is fully available, preventing race conditions.



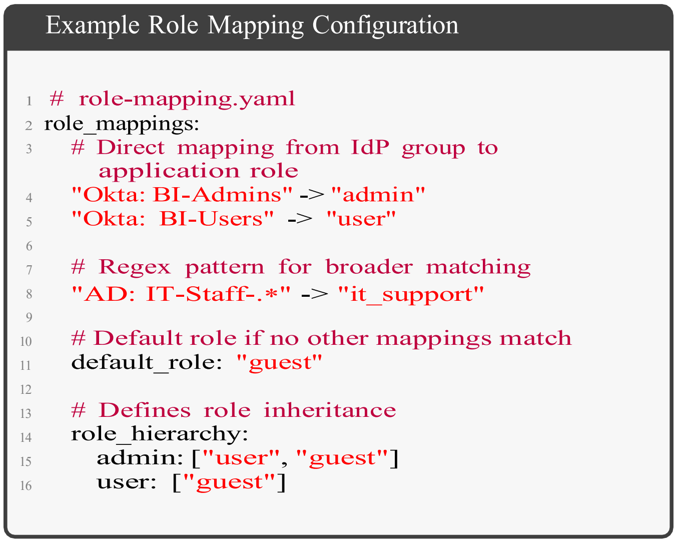
Illustrative Python Logic for RBAC Synchronization

```python
IAM_TO_APP_ROLE_MAP = {
      "BI-TOOL-ADMINS":       "Administrators",
      "BI-TOOL-USERS":  "All Users",
}
def  sync_user_roles(user_id,  iam_groups, admin_token):
      """Synchronize  a  user's  roles  based  on IAM  groups."""
      desired_roles  =  get_roles_from_iam_groups(iam_groups)
      current_roles  =  get_current_app_roles(user_id,  admin_token)

      roles_to_add  =  desired_roles  - current_roles
      roles_to_remove  =  current_roles  - desired_roles

      for  role  in  roles_to_add:
            add_user_to_role(user_id,  role, admin_token)
      for  role  in  roles_to_remove:
            remove_user_from_role(user_id,  role, admin_token)
```

Fig. 7. Illustrative code for synchronizing application roles with IdP group claims.



Example Role Mapping Configuration

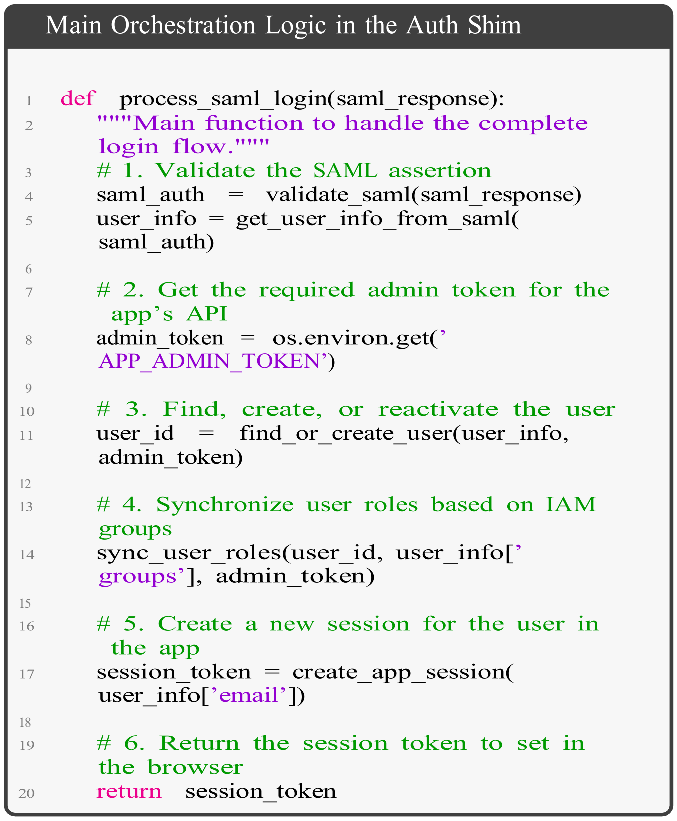
```yaml
# role-mapping.yaml
role_mappings:
  # Direct mapping from IdP group to application role
  "Okta: BI-Admins" -> "admin"
  "Okta: BI-Users" -> "user"

  # Regex pattern for broader matching
  "AD: IT-Staff-.*" -> "it_support"

  # Default role if no other mappings match
  default_role: "guest"

  # Defines role inheritance
  role_hierarchy:
    admin: ["user", "guest"]
    user: ["guest"]
```

Fig. 8. An example of a decoupled role mapping configuration, demonstrating direct, pattern-based, and default role assignments, which provides greater flexibility than hardcoded logic.



Main Orchestration Logic in the Auth Shim

```python
def  process_saml_login(saml_response):
    """Main function to handle the complete login flow."""
    # 1. Validate the SAML assertion
    saml_auth  =  validate_saml(saml_response)
    user_info = get_user_info_from_saml(saml_auth)

    # 2. Get the required admin token for the app's API
    admin_token  =  os.environ.get('APP_ADMIN_TOKEN')

    # 3. Find, create, or reactivate the user
    user_id  =  find_or_create_user(user_info, admin_token)

    # 4. Synchronize user roles based on IAM groups
    sync_user_roles(user_id, user_info['groups'], admin_token)

    # 5. Create a new session for the user in the app
    session_token = create_app_session(user_info['email'])

    # 6. Return the session token to set in the browser
    return  session_token
```

Fig. 9. High-level orchestration function showing the step-by-step logic executed by the Auth Shim upon receiving a valid SAML response.

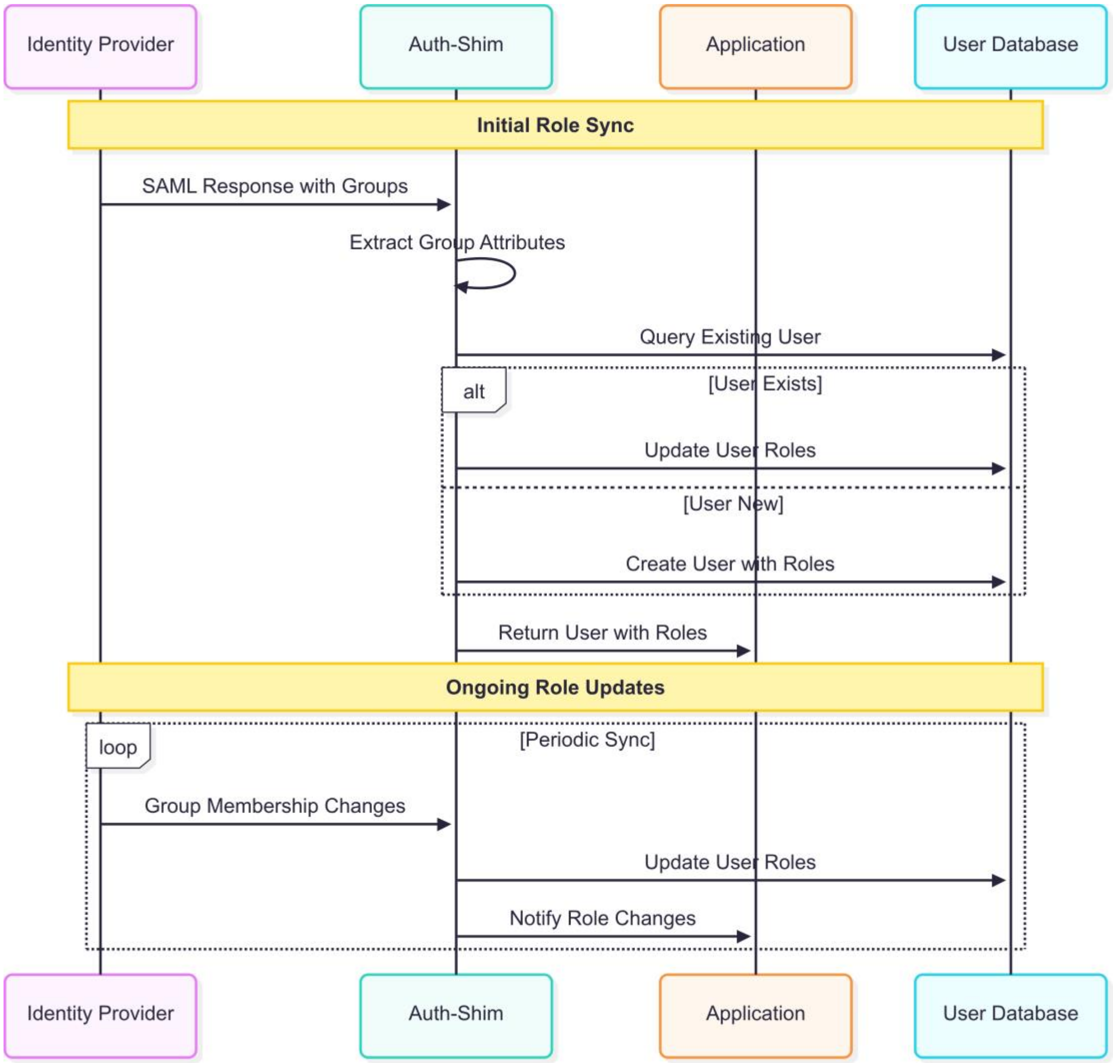


Fig. 10. The step-by-step logic of the Just-in-Time role synchronization process, from SAML attribute extraction to final session generation.

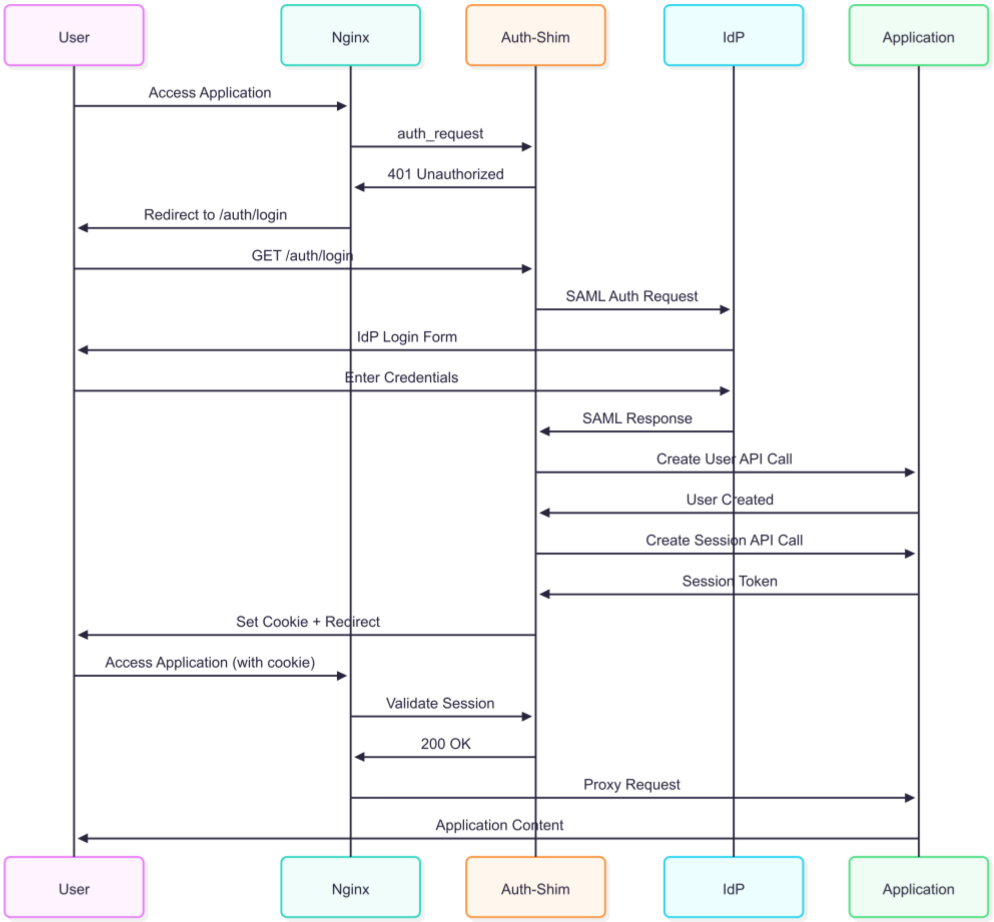


Fig. 11. Sequence diagram for a first-time user login, demonstrating automated user provisioning.

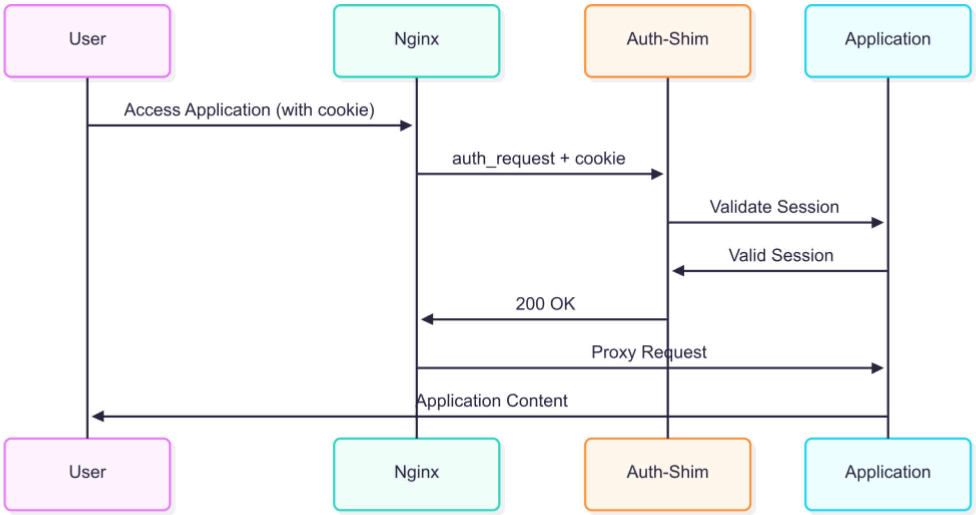


Fig. 12. Sequence diagram for a returning user with a valid session, showing the fast-path authentication check.