International Science Group
ISG-KONF.COM

TECHNICAL SCIENCES AND COMPUTER SYSTEMS ENGINEERING: PROMISING DEVELOPMENTS, MODERN METHODS AND NEW TECHNOLOGIES

Collective monograph

ISBN 979-8-90214-596-7
DOI 10.46299/ISG.2026.MONO.TECH.2
BOSTON(USA)-2026

ISBN – 979-8-90214-596-7
DOI – 10.46299/ISG.2026.MONO.TECH.2

Technical sciences and computer systems engineering: promising developments, modern methods and new technologies

Collective monograph

Boston 2026

SECTION 3. CYBERSECURITY AND INFORMATION PROTECTION

DOI: 10.46299/ISG.2026.MONO.TECH.2.3.1

3.1 Features of token-based authentication and authorization in a web-oriented microservice architecture

Introduction

In the context of the ongoing digital transformation of the economy, web-oriented information systems have become a foundational infrastructure for e-commerce, financial services, government e-services, and corporate information resources. The widespread adoption of distributed architectures, particularly microservices, along with the use of cloud and containerized platforms, increases the need for reliable authentication of both users and services while maintaining acceptable performance. Under these conditions, authentication mechanisms are no longer merely a local security component; they substantially affect scalability, fault tolerance, and the efficiency of computing resource utilization.

One of the de facto standards for token-based authentication is the JSON Web Token (JWT), which is widely used across the OAuth 2.0 and OpenID Connect (OIDC) ecosystems, as well as in proprietary solutions for securing RESTful APIs and microservice interactions. Standards and normative documents (e.g., RFC 7519, RFC 7518, RFC 8032) define the token format and the permissible signing algorithms. At the same time, in practical web-application design, the choice of a specific cryptographic algorithm and authentication scheme is often made empirically, without relying on formalized models or quantitative criteria to assess the impact of this choice on latency, throughput, and system resource consumption. As a result, designers may adopt either overly “heavy” algorithms that significantly increase latency under high load, or overly “lightweight” mechanisms that do not provide an adequate cryptographic level of protection—an issue that is particularly critical for microservice architectures, where tokens may be verified repeatedly across a chain of services.

An analysis of scientific publications and standards reveals that much of the research focuses on the security of authorization and authentication protocols, the

stability of access schemes, and the construction of architectural templates for using JSON Web Tokens (JWT) in web services. However, the issue of comprehensively quantifying the impact of cryptographic algorithm choices for JWT token signing and specific authentication schemes on the performance of web-oriented applications within various architectural approaches (e.g., monolith, SOA, and microservices) has not been sufficiently researched. This issue is usually considered only for specific technology stacks or load scenarios. Thus, there is a need to develop models, methods, and software tools that enable a comparative analysis of these mechanisms using uniform criteria and performance metrics.

1. Authentication processes in web applications

Authentication is a fundamental process of identifying a system entity (user, device, or service) in compliance with security requirements. In the context of information systems, implementing authentication mechanisms is a prerequisite for preventing unauthorized access to protected resources and services.

Functionally, authentication and authorization solve different security lifecycle tasks:

authentication answers the question “Who are you?” by verifying the claimed identity of the entity.

authorization determines the answer to the question “What are you allowed to do?” by establishing the level of access rights for the authenticated subject.

implementing these mechanisms requires creating a secure infrastructure to verify and authenticate the relevant entities.

In web application architecture, the need for authentication arises when the client-side accesses the API provided by the server-side of the system. Below are the main authentication mechanisms that are widely used in modern web solutions.

1.1. Basic HTTP authentication

The Internet Engineering Task Force (IETF) standardizes a number of authentication and authorization mechanisms suitable for use by web clients. The most fundamental form of authentication is the transmission of user credentials (username and password) via the HTTP Authorization header. This mechanism follows a

“challenge–response” model in which the server initiates the authentication procedure when a protected endpoint is accessed, and the client provides the required credentials.

The standard defines two primary schemes for implementing this approach. Basic Authentication involves the client sending the username and password in Base64-encoded form, having first concatenated them into a string of the format "username:password" [21].

Digest Authentication is a more secure scheme in which the client constructs a response to the server’s challenge as a hash value derived from the username, password, and a unique nonce provided by the server [22]. Historically, the MD5 algorithm was used for hashing; however, modern versions of the specification provide for the use of a cryptographically secure algorithm such as SHA-256. Despite improved security characteristics compared to Basic Authentication, the Digest scheme has not gained widespread adoption in web development practice.

It is important to note that this mechanism, by itself, does not ensure the confidentiality of the transmitted data. Base64 encoding of credentials is sometimes mistakenly perceived as equivalent to encryption; in practice, however, such data can be easily decoded back into plaintext. Therefore, the security of these schemes depends entirely on the security of the transport channel. The de facto standard for securing data transmission is the use of HTTPS, which encrypts traffic using the SSL/TLS cryptographic protocols.

1.2. API keys

API keys are cryptographically random sequences that function as a shared secret between a client and an API provider. The authenticating subjects may be either individual users or entire projects/applications integrated with the API. When API keys are used for user authentication, the API server generates a unique secret key for each account. In this context, the API server (i.e., the back-end server) is responsible for key generation, secure storage (typically in a protected database), and validation of the key on every incoming request.

To prevent predictability, API keys must be generated using cryptographically secure algorithms. A practical approach is to use Universally Unique Identifiers

(UUIDs), which are random strings generated for each registered user. For additional protection against forgery, a digital signature mechanism may be applied using a shared secret key.

The key distribution process typically involves delivering the key to the user via an HTTP redirect after successful initial authentication or registration. Depending on the implementation, keys may be provided to the client in the form of:

cookies stored in the browser for the corresponding domain;

URL redirect parameters.

When the cookie mechanism is used, each subsequent request to the API server automatically includes the cookie, enabling session identification.

Despite the simplicity of implementing client authentication and session support, API keys have significant security limitations. Because they constitute a shared secret, compromising a key enables an attacker to impersonate a legitimate user. As an alternative approach, API keys are commonly recommended for identifying projects/applications rather than individual users.

In architectures where a front-end service acts as a proxy between clients and the API server, a higher level of security can be achieved. Users do not have direct access to the front-end server, which allows the API key to be stored securely. This service appends the API key to each proxied request, forwards the response to the client, and implements its own mechanisms for authenticating individual users.

API key provisioning typically requires developers to register their applications to obtain a unique key, often accompanied by a warning that it will be shown only once. A common anti-pattern is embedding API keys directly in front-end source code, which can lead to compromise when the code is published in public repositories.

The recommended methodology is to use environment variables to store confidential data, including API keys. Modern Platform-as-a-Service (PaaS) cloud platforms provide built-in tools for configuring environment variables without requiring code modifications. Embedding API keys in client-side JavaScript or otherwise exposing them in a web browser is an especially critical error [23, 24].

1.3. Token-based authentication

Token-based authentication is another approach that has gained popularity in recent years. The authentication mechanisms discussed earlier follow a stateful model, which leads to scalability constraints and architectural heterogeneity. Token-based authentication enables a stateless authentication flow in which the server is not required to maintain any client-related state.

In token-based authentication, a front-end application interacts with a dedicated Identity Provider (IdP) by redirecting the user to the IdP's interface. The IdP performs user authentication, typically using a username–password pair, with optional additional verification factors (e.g., one-time passwords). Upon successful authentication, the IdP issues and returns to the client:

an ID token, which confirms the user's identity;

an access token, which grants access to protected APIs.

The front-end application stores the access token for subsequent use when requesting protected resources.

OpenID Connect (OIDC) defines a standardized protocol for token-based authentication [25], built on top of the OAuth 2.0 specifications. A key element of OIDC is the scope parameter with the value OpenID, which must be included in the authentication request [26, 27]. The authorization server returns an ID token in JWT format that contains identification information about the authenticated user as claims. OIDC specifies a clear set of rules for validating the ID token before relying on its contents. It should also be noted that the claims from the ID token are available via the authorization server's /userinfo endpoint [25], access to which requires presentation of a valid access token.

ID tokens contain a standard set of parameters that characterize the user session state:

issuer identifier;

subject identifier (user);

audience (intended token consumers);

expiration time;

authentication time.

A critical parameter of the authentication request is response_type, which must include the value id_token in order to obtain an ID token [28].

OIDC defines different authentication flows depending on the application type. For architectures with predominantly server-side logic processing (traditional web applications or SPAs with server-side middleware), OIDC specifies the Authorization Code Flow. In this case, the OIDC client receives an authorization code from the IdP and then exchanges it for an ID token and an access token via the token endpoint [26, 27]. According to the OAuth 2.0 specification, the authorization server always returns an access token; OIDC extends this requirement by also mandating the return of an ID token [29].

The need for an access token depends on the application architecture. In scenarios where only confirmation of the user's identity is required, the claims contained in the ID token are sufficient. In such cases, the access token may be used solely to retrieve additional user information via the /userinfo endpoint.

A sequence diagram of the Authorization Code Flow is shown in Fig. 1.

An important stage in the token-based authentication procedure is client identification when calling the token endpoint. The OIDC specification defines a set of mandatory client authentication methods that the Identity Provider (IdP) must support. Each of these methods uses different mechanisms to verify the client's authenticity via the *client_secret* parameter.

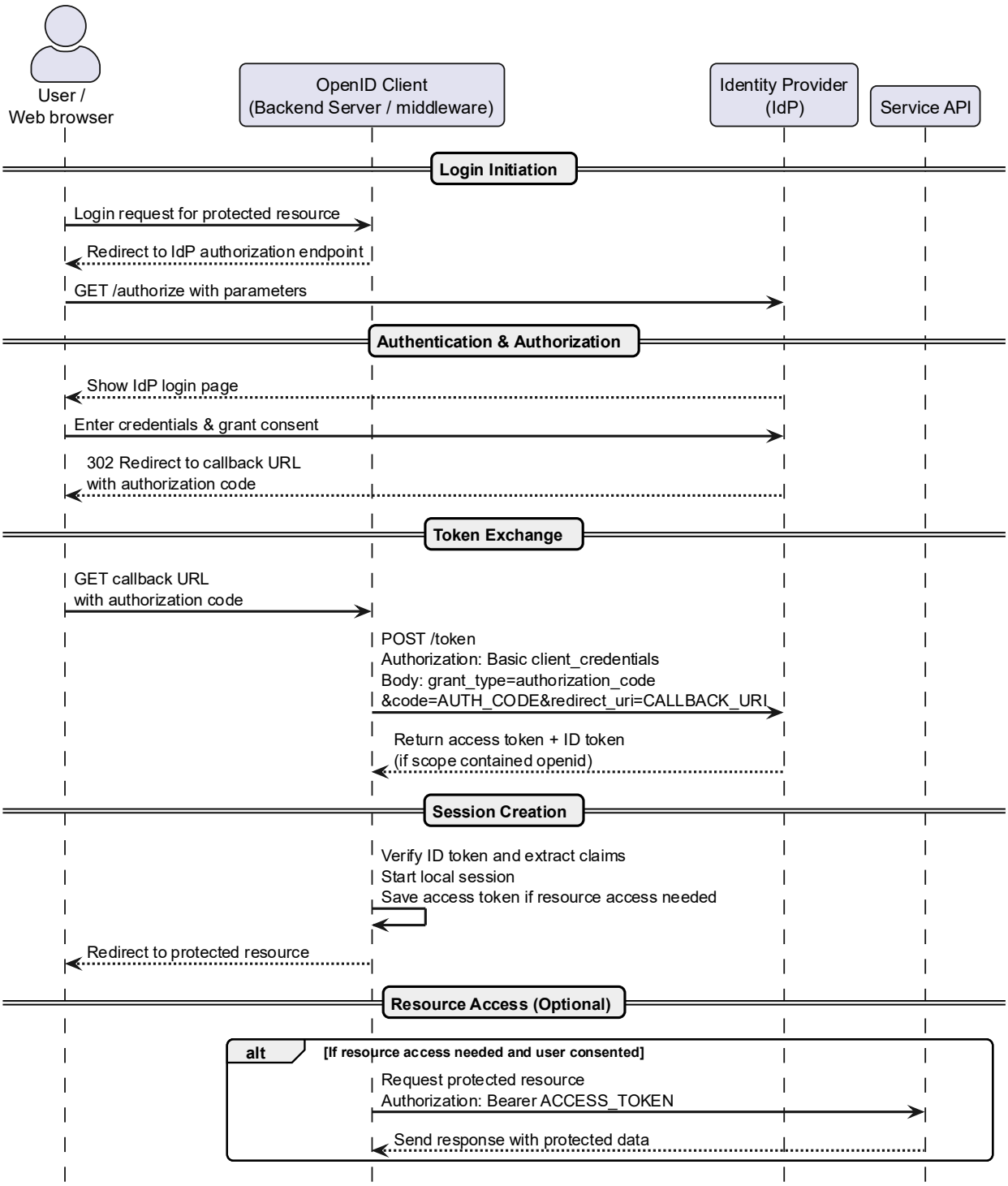

Figure 1. Token-based Authorization Code Flow

Source of the figure: original

1.4. Session Management

ID tokens and access tokens issued by the Identity Provider (IdP) typically have a limited validity period in order to enhance security. This necessitates periodic token

renewal by the client once the current tokens expire. An OIDC client is obliged to manage sessions for both token types, which may have different lifetimes.

Functionally, tokens are classified as follows:

identity token is intended for the OIDC client and contains information about the user's identity and authentication state, presented as claims in the payload. It is used to build the user profile.

access token is directed at the resource server and provides authorized access to protected resources on behalf of the user.

According to the OIDC specification, a user session is the period during which the OIDC client maintains a valid authenticated status with the IdP. The session management process involves monitoring this status. Session termination leads to invalidation of all active tokens, even if their formal expiration time has not yet been reached. Subsequent acquisition of new tokens requires user re-authentication.

This approach provides an additional layer of security because it enables immediate access revocation in the event of detected credential compromise or other security threats, regardless of the configured lifetimes of individual tokens. To maintain application continuity, it is recommended to implement automatic token renewal mechanisms using refresh tokens, which allow new access tokens to be obtained without user involvement (see Fig. 2).

To support continuity of authentication sessions in OIDC, two primary approaches are used, one of which is silent re-authentication [25].

This mechanism assumes that the web application initiates a standard authentication procedure using the same parameters as in the initial request to the IdP authorization endpoint. The key difference is that the client periodically contacts the OpenID Provider to check the end user's session status without the user's participation.

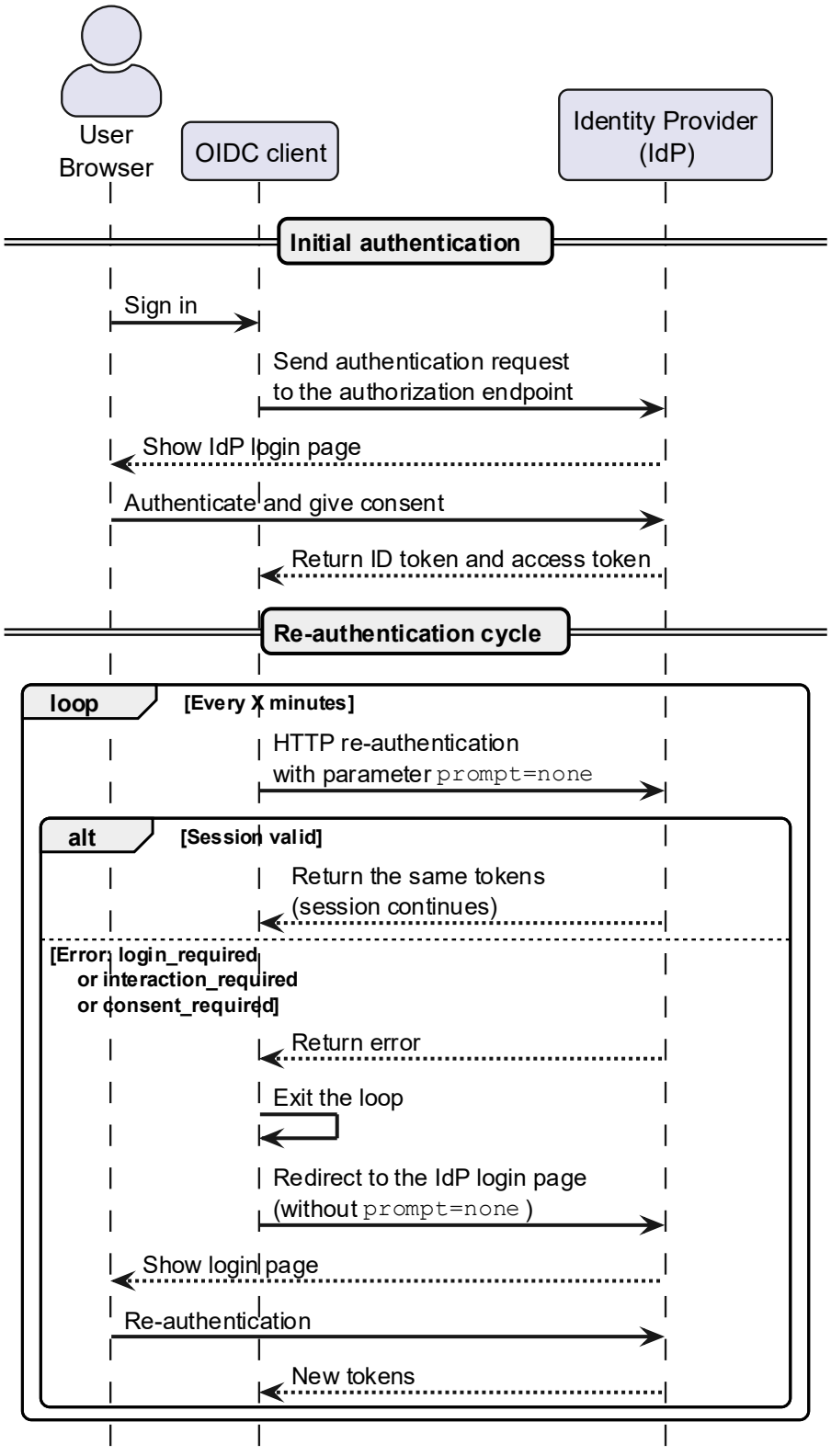

Figure 2. Session management using silent authentication

Source of the figure: original

A key technical parameter of this process is prompt=none, which instructs the Identity Provider to avoid interactive engagement with the user. Provided that an active session exists on the IdP side, the provider automatically returns the requested tokens.

The second method for maintaining sessions involves implementing a mechanism based on hidden iframes integrated with the Identity Provider (IdP). This approach is based on the principle of regularly polling the session status by the web client at predefined intervals [30].

1.5. Single Sign-On Integration

The Single Sign-On (SSO) system enables a user to utilize a single set of credentials to access multiple applications within an organizational environment. Implementing this architecture involves centralized storage and management of credentials at the organization's Identity Provider (IdP).

Principles of SSO Operation:

trusted infrastructure: each organizational application establishes a trust relationship with the IdP.

unified authentication point: the user is redirected to the IdP’s single sign-in page for all applications.

efficient status propagation: after successful authentication, the IdP conveys the confirmed identity status to the application.

In the context of OIDC, authentication status is propagated via an ID token, which contains verified information about the user [31]. This mechanism provides a standardized way for trusted parties to exchange authentication information, eliminating the need for separate authentication for each application.

The advantages of this approach include not only improved convenience for the end user but also enhanced security through centralized account management and the ability to enforce a unified security policy across the organization’s application ecosystem.

1.6. JWT Tokens

JWT is the de facto standard for implementing ID tokens in modern authentication systems [32]. The payload of an ID token contains a set of claims that

represent identification information about the user. Client applications use these data to personalize the user interface and adapt functionality.

A JWT is a JSON-based token format exchanged between parties. It is easy to transfer and compactly encodes information about a user or a session. A JWT (Fig. 3) typically consists of:

Header – the first part of the JWT is a JSON object encoded using Base64URL. It specifies metadata such as the signing algorithm (e.g., RS256).

Payload – contains the data conveyed by the token, including the user identifier, user information, session details, permissions, expiration time, and related attributes.

Signature – the Base64URL-encoded header and payload are concatenated and signed using a private key (for asymmetric algorithms) or a shared secret (for symmetric algorithms). The resulting signature is then Base64URL-encoded.

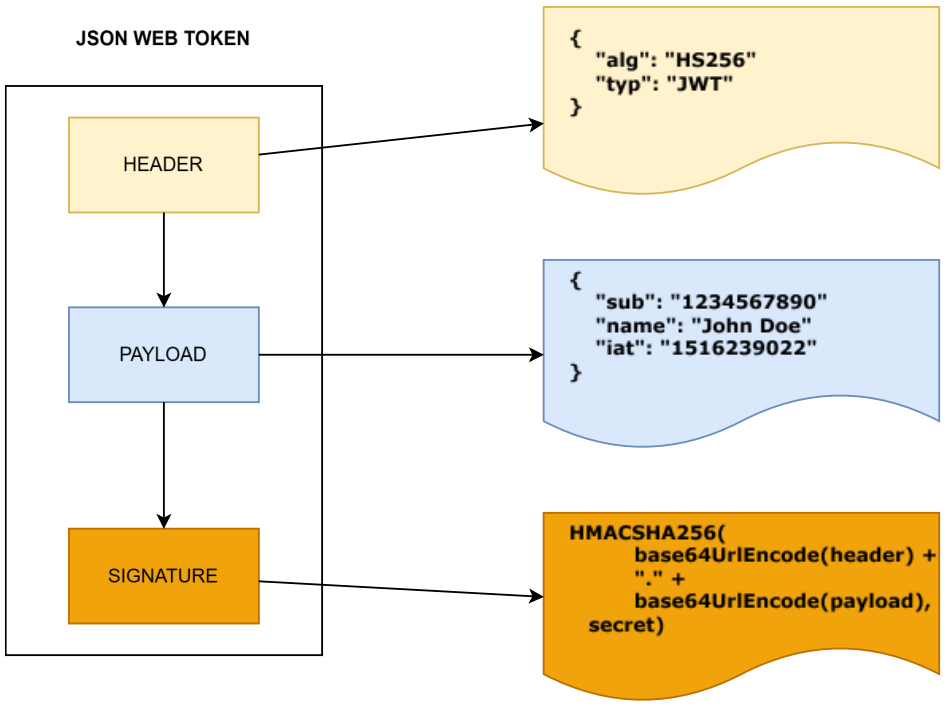

Figure 3. Structure of a JWT Token

Source of the figure: https://www.miniorange.com/blog/what-is-jwt-json-web-token-how-does-jwt-authentication-work/

JWT validation and processing:

token decoding – the front-end application first decodes the JWT from the Base64URL format defined by the standard [32];

digital signature verification – a mandatory step is verifying the JWT's cryptographic signature to confirm data integrity and authenticity;

payload analysis – after successful verification, the application gains access to the structured data in the form of a JSON object.

Base64URL encoding enables safe transmission of the token via HTTP headers by avoiding issues with special characters inherent in standard Base64. However, without prior signature verification, payload data must not be trusted, since it can be easily inspected after simple Base64URL decoding.

This mechanism allows applications to obtain up-to-date user information without additional server requests, which significantly improves performance and user experience.

Previous studies have evaluated token-based authentication using JWT, noting that JWT-based approaches may outperform alternative methods in terms of authentication mechanisms, access control mechanisms, compact and self-contained token structure, support for multi-factor authentication/SSO, and scalability of the access control system [31]. Other comparative evaluations analyzed JWT-based systems using both single and multiple tokens in both single-server and multi-server deployments, indicating that multi-token designs may exhibit lower performance than systems relying on a single token [33]. Common algorithms for JWT include HMAC-SHA-256, RSASSA-PKCS1-v1_5 with SHA-256, and ECDSA P-256 with SHA-256 [34]. In addition, RFC 8032 [35] defines newer algorithms, such as Ed25519 and Ed448; however, performance test results for these modern algorithms have not yet been widely published.

2. Schemes for securing microservices using JWT

JWT addresses two main challenges in microservice security design: protecting communication between services and propagating end-user context across microservices (Fig. 4). All microservices within a deployment trust a Security Token

Service (STS). The API Gateway exchanges JWTs received from client applications for new JWTs issued by the STS. In most cases, JWT is used together with mTLS

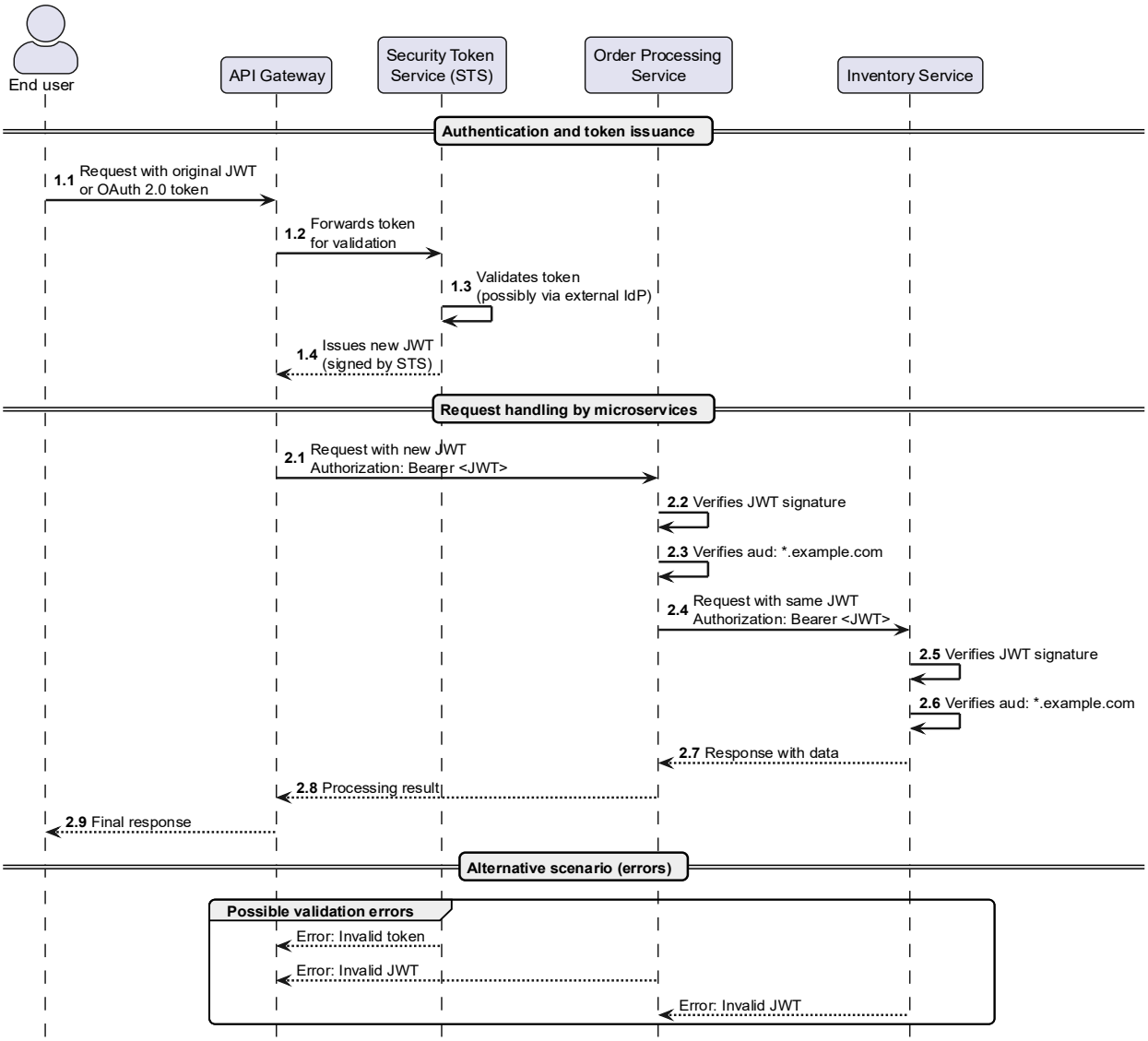

Figure 4. Propagation of end-user identity data via JWT among microservices

Source of the figure: original

2.1. Scheme for Sharing End-User Context Between Microservices Using a Shared JWT

When microservice-to-microservice identification is not critical but end-user identification (human or system) is required, using JWT should be considered. In this case, the services do not authenticate each other directly. Each request must carry the identity of the end user who initiates the message flow; otherwise, the receiving

microservice rejects the request. The flow for propagating end-user identity data in a JWT among microservices, as shown in Fig. 3, is as follows:

1.1. The end user initiates the request flow. This end user may be a human or a system.

1.2. The API Gateway authenticates the end user. The gateway intercepts the request, extracts the token (which may be an OAuth 2.0 reference token or a self-contained token), and interacts with the connected Security Token Service (STS) to validate it. The token provided by the end user may not have been issued by this STS; it can originate from any other identity provider trusted by the STS. Therefore, the STS must be able to validate the token presented at this step.

1.3. After validation, the STS issues a new JWT signed by the STS. This JWT contains end-user data copied from the original token (Step 2). When the API gateway forwards the new JWT to upstream microservices, those microservices only need to trust this STS to treat the token as valid. Typically, all microservices within a single trust domain rely on the same STS.

2.1. The API gateway forwards the STS-issued JWT to the Order Processing microservice over TLS, using the HTTP Authorization header with the Bearer scheme. The Order Processing microservice verifies the JWT signature to ensure that it was issued by a trusted STS. In addition to signature verification, it validates the `aud` (audience) claim. For the pattern described in this section to work, all microservices in the same trust domain (i.e., trusting the same STS) must accept JWTs with a wildcard `aud` value, such as **.shop.com*.

2.2. When the Order Processing microservice calls the Inventory microservice, it propagates the same JWT received from the API gateway. The Inventory microservice verifies the JWT signature to confirm that it was issued by the trusted STS and checks that the `aud` claim equals **.shop.com*.

This approach allows JWT to achieve two objectives. First, it enables propagation of end-user context across microservices in a way that is difficult to forge: because the JWT claim set is signed by the STS, no microservice can modify it without invalidating the signature. Second, it helps secure service-to-service communication: a

microservice can call another microservice only if it presents a valid JWT issued by a trusted STS. Any receiving microservice rejects requests that do not include a valid JWT.

2.2. User Context Sharing Scheme with a Session Service JWT for Each Inter-Service Interaction

The user-context sharing scheme based on issuing a session service JWT for each inter-service interaction is essentially a modification of the shared-JWT scheme. The objective remains limited to establishing the identity of the end user, whereas microservice identification is not critical. Instead of propagating the same JWT across all microservices with a single audience value accepted by each service, a new JWT is generated for each distinct service interaction. This approach provides a higher level of security compared to using a shared (distributed) JWT.

However, it should be emphasized that absolute security is unattainable: the appropriateness of a given approach depends on specific usage scenarios and the level of trust in the microservices deployment infrastructure.

The principle of operation of the circuit is shown in Fig. 5. The interaction flow is similar to that described in the previous section, except for Steps 2.1 and 2.2.

In Step 2.1, before the Order Processing microservice initiates interaction with the Inventory microservice, it contacts the STS and performs a token exchange operation. It submits the JWT received from the API gateway (issued with `aud`: op.shop.com) and requests a new JWT to access the Inventory microservice (see Fig. 4).

In Step 2.2, the STS issues a new JWT with `aud`: *iv.shop.com*. The processing flow then continues according to the scenario outlined in the previous subsection.

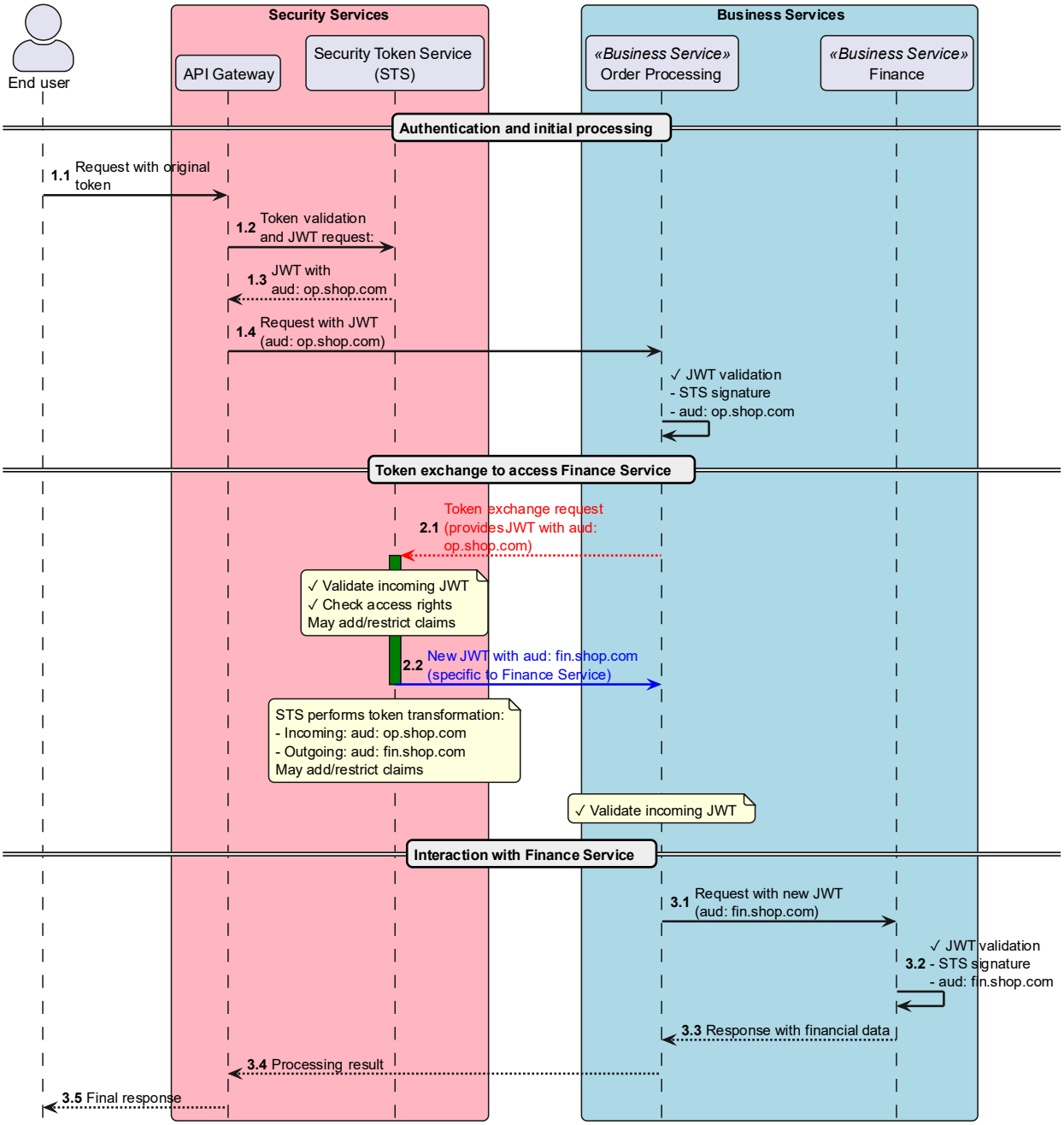

Figure 5. Propagation of end-user identity data in JWTs across microservices using token exchange

Source of the figure: original

The need to issue a new JWT with a new `aud` value during the interaction between the Order Processing microservice and the Inventory microservice is driven by the requirements of trust isolation between services and attack-surface minimization. This approach is more secure than allowing all microservices within a

deployment to share the same JWT received from the API gateway under a single audience value. At least two significant reasons can be highlighted:

*explicit mapping between a microservice and its audience (*aud*) value*. If each microservice in the deployment is assigned a unique audience value and the STS issues JWTs with that value, then for any token it is unambiguous which microservice is the intended recipient. For example, in Step 1.4 of Fig. 2.2, when a request arrives at the Order Processing microservice from the API gateway, the microservice can strictly verify that the token is intended for it rather than for another microservice. If the JWT is mistakenly or maliciously routed to a different microservice, it will be rejected due to an aud mismatch. This prevents incorrect or unauthorized use of the token outside its intended audience.

prevention of unauthorized token reuse across microservices. If the Order Processing microservice attempts to reuse the received JWT "as is" to access another service (e.g., a Finance microservice, which ideally it should not be permitted to access), the request will be rejected because the aud value in the original JWT does not match the audience expected by the Finance microservice. Consequently, the only way for the Order Processing microservice to interact with the Finance microservice is to submit its current JWT to the STS and perform a token exchange to obtain a new JWT with an audience value acceptable to the Finance microservice. In this model, access-control decisions are centralized at the STS: the STS determines whether the Order Processing microservice is allowed to obtain a token for accessing the Finance microservice. This enables finer-grained and more manageable control over inter-service interactions and implements the principle of least privilege.

2.3. Scheme for Sharing User Context Between Microservices in Different Trust Domains

The use case in this section extends the token-exchange use case discussed in Section 2.1.2. As shown in Fig. 6, most steps are straightforward. There are no changes relative to the previous section up to Step 3.2. (Steps 3.1 and 3.2 in Fig. 6 are equivalent

to Steps 2.1 and 2.2 in Fig. 5.

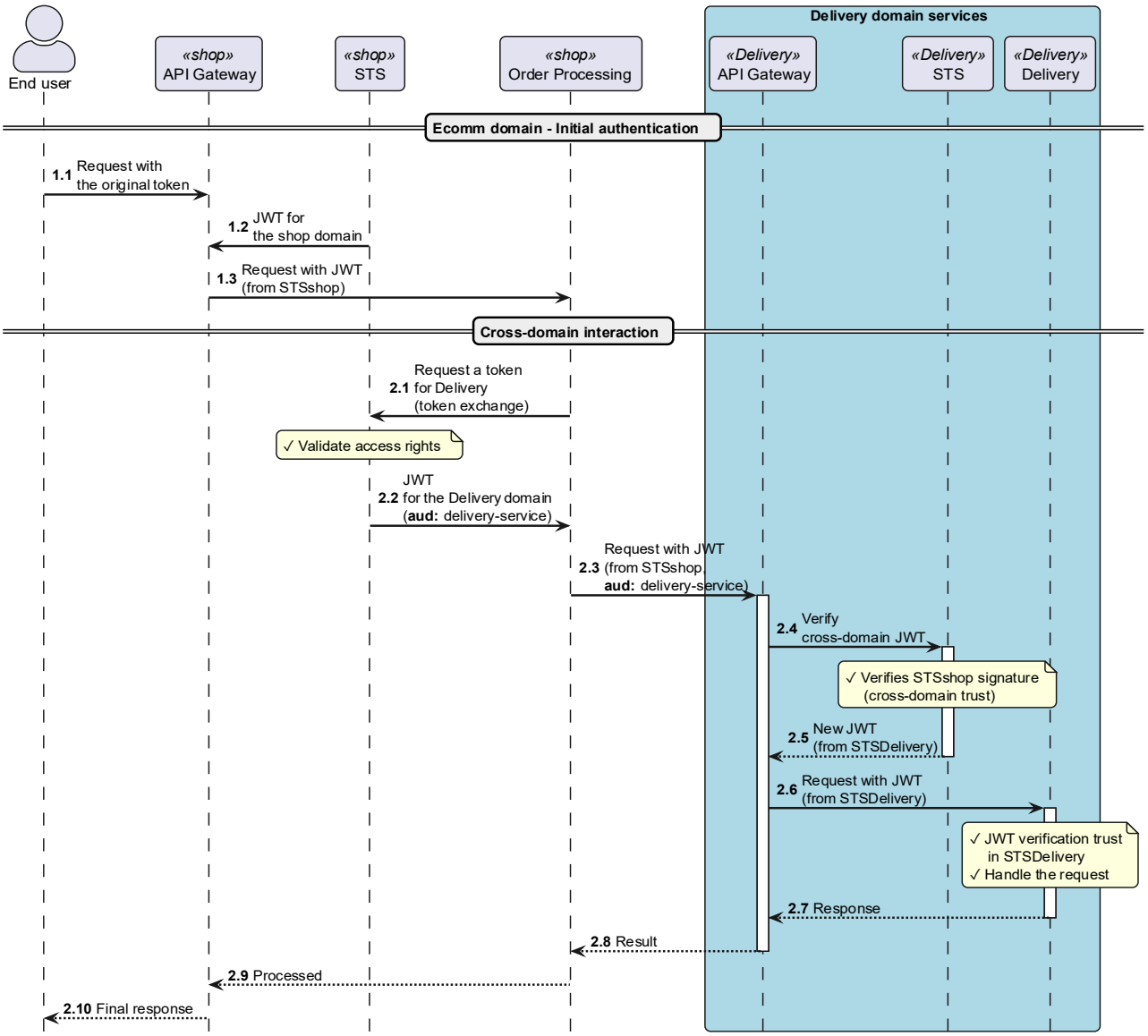

Figure 6. Cross-domain authentication and sharing of user context across multiple trust domains

Source of the figure: original

In Step 2.3, the Order Processing microservice in the shop domain attempts to access the Delivery microservice in the delivery domain via the delivery API gateway. The JWT included in this request (Step 2.3) is issued by the STS in the shop domain and has an audience value corresponding to the Delivery microservice.

In Step 2.4, the API gateway in the delivery domain interacts with its own STS to validate the JWT. Validation succeeds only if the delivery STS trusts the shop STS.

In other words, the public key required to verify signatures on JWTs issued by the shop STS must be known to the delivery STS.

If this condition is satisfied, then in Step 2.5 the delivery STS issues its own JWT and forwards it to the Delivery microservice via the API gateway. All microservices within the delivery domain trust only the STS of their own domain.

2.4. Self-Issued JWTs

In the previous use cases, the identity of microservices as separate entities was not critical. The system relied on a JWT issued by a trusted STS that contained the end-user identity. In contrast, in the self-issued JWT model (see Fig. 7), identifying microservices during service-to-service interaction becomes essential, similarly to the mTLS model.

As with mTLS, in this model each microservice must possess its own cryptographic public–private key pair. The initiating microservice constructs a JWT, signs it with its private key, and sends it along with the request to the receiving microservice via the HTTP Authorization header using the Bearer scheme over a secure TLS channel. Because the JWT in this case is a bearer token, TLS is effectively mandatory to protect against interception and subsequent unauthorized use. The receiving microservice verifies the JWT signature using the initiating microservice's public key, thereby unambiguously identifying it.

A self-issued JWT should include the standard JWT claims (see Table 1), as well as any additional claims (see Table 2) required by the application, including the key claims that are commonly used.

Table 1

1) Standard JWT Claims

Claim	Name	Description
1	2	3
`iss`	Issuer	Identifier of the service that issued the token. In the case of a self-signed (self-issued) JWT, this is the identifier of the microservice that created it (e.g., "order.shop.com").
`sub`	Subject	Identifier of the subject (often matches `iss`). For a microservice, this may be the identifier of the microservice itself or the user on whose behalf it acts. However, for self-issued JWTs, `sub` frequently contains the same identifier as `iss`.

Continuation of Table 1.

1	2	3
`aud`	Audience	Target service(s) for which the token is intended. This may be one or multiple services. For example, "inventory.shop.com".
`iat`	Issued At	Token issuance time in Unix timestamp format.
`exp`	Expiration	Token expiration time.
`jti`	JWT ID	Unique token identifier used to prevent replay attacks.

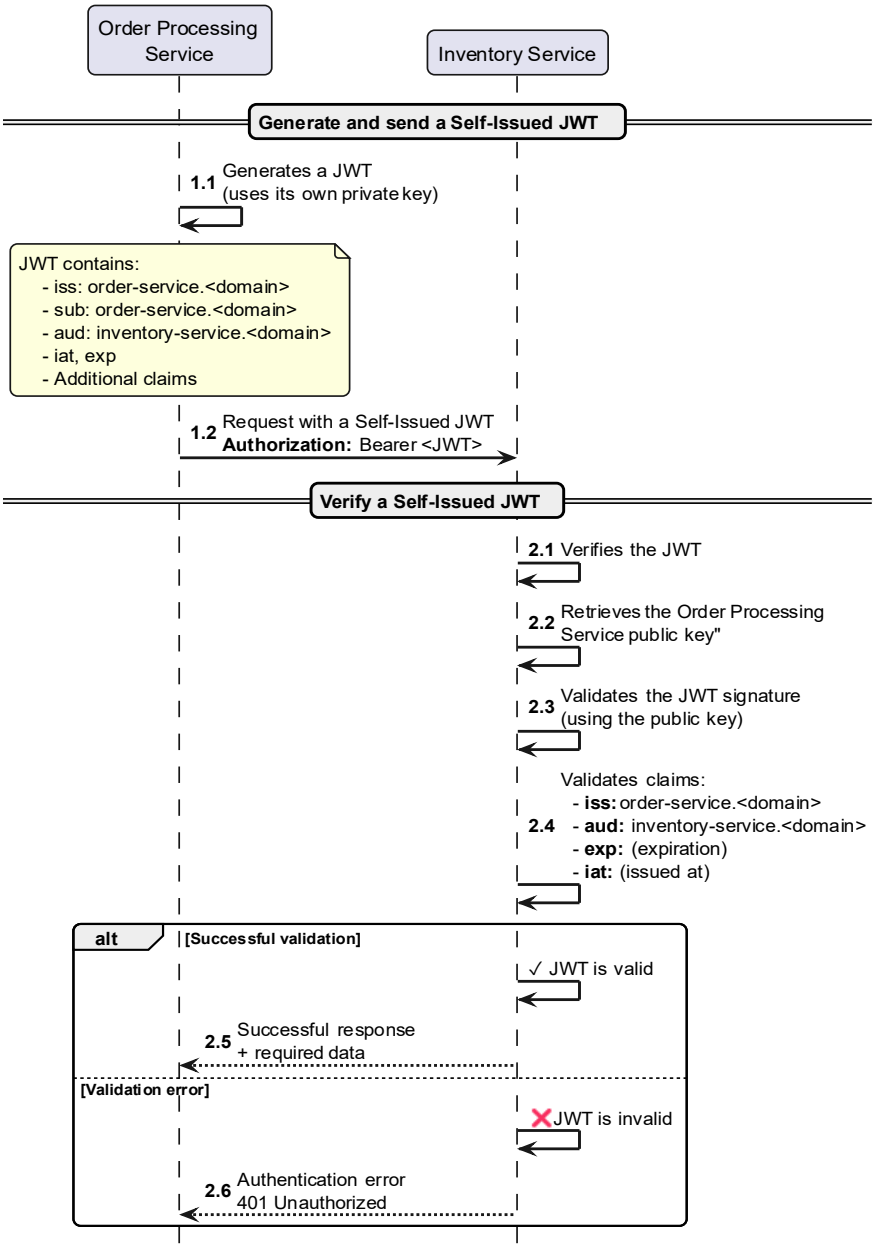

Figure 7. Self-issued JWT. The JWT is signed using the private key of the order-processing microservice <JWT>

Source of the figure: original

Table 2

2) Recommended JWT Claims

Claim	Name	Description
nbf	Not Before	The time from which the token becomes valid.
scope	Scope	A list of permissions granted by the token (e.g., "inventory:read inventory:write"). This is not a standard JWT claim, but it is commonly used in OAuth 2.0.
roles	Roles	The service's roles within the system.
service_id	Service ID	The identifier of the service used to discover/resolve it within the environment (Docker, Kubernetes). The ID should be a URI (domain name) or a set of aliases that clearly identify a specific resource or a group of resources.
service_version	Service Version	The service version used for tracing/observability.
environment	Environment	The execution environment (production/staging/development).

Custom claims may also be added when additional information must be conveyed (e.g., roles, access rights, etc.; see Fig. 8).

```
{
  // Main claims
  "iss": "order-service.example.com",
  "sub": "order-service.example.com",
  "aud": "inventory-service.example.com ",
  "iat": 1620000000,
  "exp": 1620003600,
  "jti": "550e8400-e29b-41d4-a716-446655440000",

  // Additional safety claims
  "nbf": 1620000000,

  // Authorization requests
  "scope": "inventory:read inventory:write",
  "roles": ["order_processor", "data_reader"],

  // Service context statements
  "service_id": "order-service-v1",
  "service_version": "1.2.3",
  "environment": "production",

  // Business context (optional)
  "tenant_id": "tenant-123",
  "permissions": ["read_inventory", "update_stock"]
}
```

Figure 8. Example of a JWT token payload

Source of the figure: original

2.5. Security Considerations for Bearer Tokens

A JWT used as a bearer token is inherently similar to cash: anyone who possesses the token can use it until it expires, without providing additional proof of ownership. If an attacker intercepts a JWT, they can issue requests on behalf of the legitimate token holder for as long as the token remains valid.

Therefore, when JWT is used for authentication between microservices (i.e., for mutual service-to-service authentication), it is necessary to:

secure the communication channel using TLS to minimize the risk of token interception;

keep the JWT lifetime as short as possible (short-lived tokens) to reduce the potential impact of a compromised token.

When these requirements are met, the risk of abuse of a stolen token is significantly reduced, and the self-issued JWT model becomes a powerful mechanism for implementing secure inter-service communication while preserving context and supporting non-repudiation properties.

2.6. Nested JWTs

The usage variant considered in this subsection is an extension of the scenario described in Section 2.1.3. A nested JWT is a token in which one JWT encapsulates another (see Fig. 9). A typical use case is to protect communication between two microservices using a self-issued JWT that contains an inner JWT issued by a trusted Security Token Service (STS) and carrying the end-user context. The resulting nested JWT therefore combines two layers of trust: trust in the STS and trust in the calling microservice.

On the recipient side, the microservice must:

1) verify the signature of the outer (enclosing) JWT using the public key of the calling microservice;

2) extract the inner JWT and verify its signature using the corresponding public key of the trusted STS.

Thus, the nested JWT simultaneously contains the end-user identity data and the calling microservice’s data, while the integrity and authenticity of these data are

cryptographically protected: forging such a token without compromising the corresponding private keys is practically impossible.

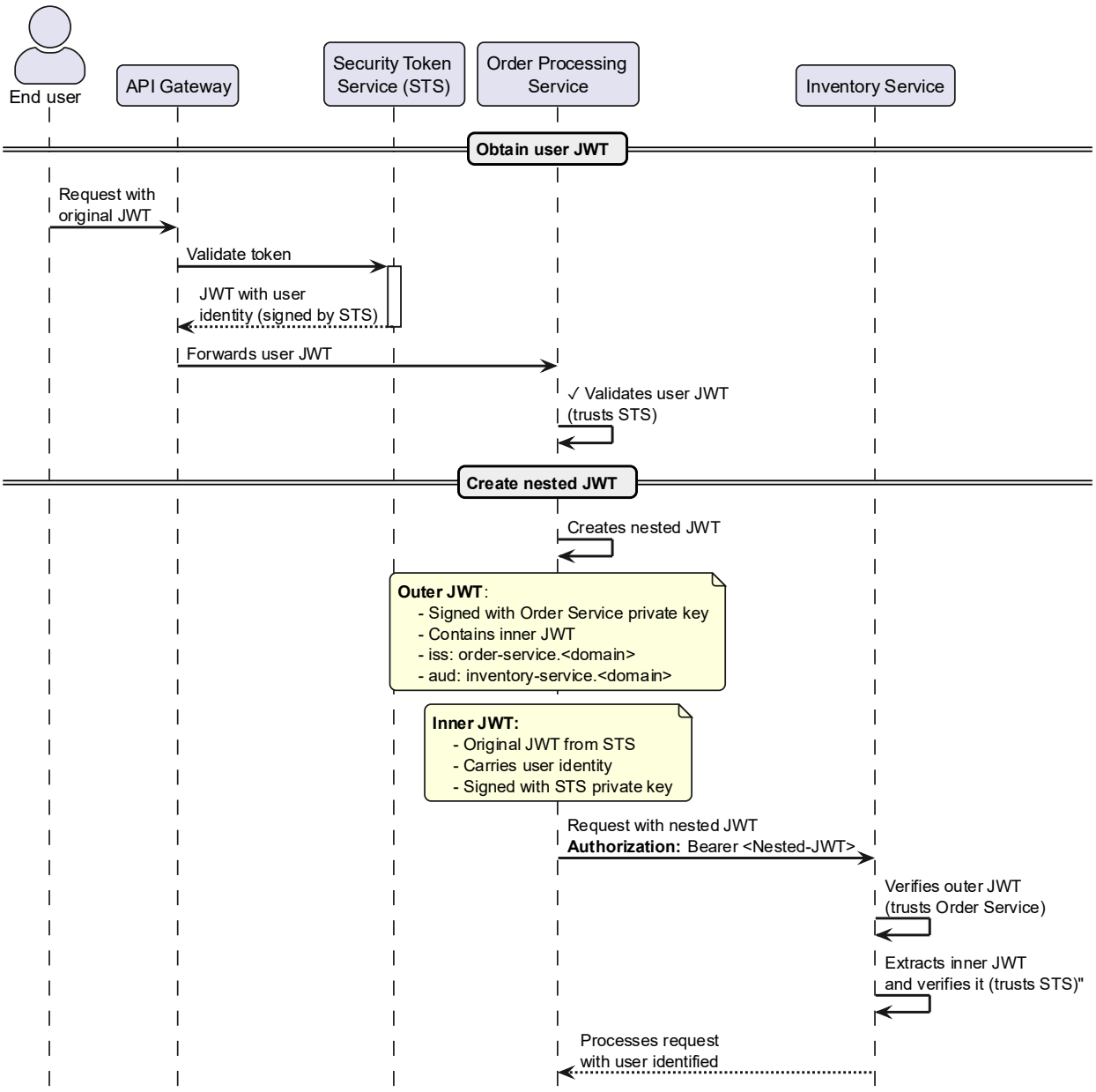

Figure 9. Nested JWT: The order-processing microservice creates its own JWT and embeds within it the JWT received from the next microservice (or the API Gateway)

Source of the figure: original

3. Model for Implementing JWT-Based Authentication in the Web-Oriented Microservice E-Commerce System “Shop”

A prototype model of a JWT-based authentication service is developed for the web-oriented e-commerce system “Shop,” built using a microservice architecture. The application domain is an online store where users register, browse the product catalog, manage a shopping cart, place orders, and handle their financial accounts. Such systems are characterized by a high volume of concurrent requests, the need for horizontal scaling, and heightened security requirements for operations involving money and personal data. Therefore, the objective is to build a prototype that demonstrates how JWT can be used as a basic mechanism for authentication and for propagating user identity across independent microservices.

The overall structure of the Shop model is shown in Fig. 10. At the top level, infrastructure services are distinguished from functional services of the e-commerce domain. The infrastructure layer includes the User Authentication service (User AuthN) and the Security Token Service (STS). The STS is responsible for issuing cryptographically protected JWTs, while User AuthN interacts with the user, validates credentials, and generates token-issuance requests. Another key infrastructure element is the API Gateway, which serves as the single external entry point to the microservice ecosystem of the Shop.com model. It receives all HTTP requests from clients, performs preliminary checks and routing, and is responsible for transmitting JWTs between the user and internal services.

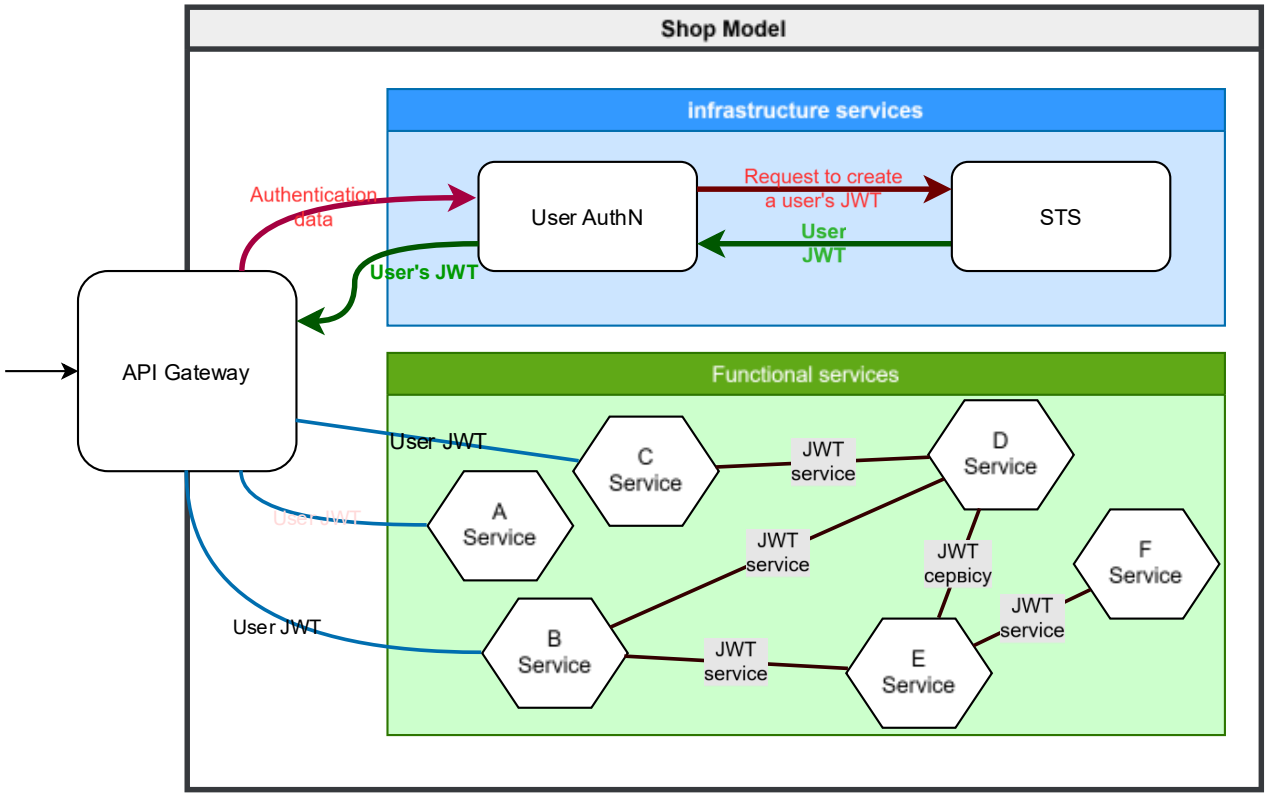

Figure 10. Shop model. Primary propagation via a JWT token

Source of the figure: original

The functional layer consists of a set of business services that implement the core capabilities of the online store. These include the User Service, Account Service, Product Service, and Basket Service, which are conventionally labeled in the diagram as Services A, B, C, D, E, F. Each of these services is autonomous: it has its own database, implements a limited set of well-defined operations, and scales independently. All inter-service calls are performed via HTTP interfaces and are accompanied by JWT transmission, ensuring end-to-end propagation of user identity and enforcement of access policies at the level of each individual microservice.

The STS is the core of the security model. Its primary function is to issue JWTs for user sessions and, when required, service tokens for inter-service communication. Upon successful authentication, the STS generates a token that includes the user identifier (UserID), a role or set of roles, additional attributes (e.g., a list of permitted operations or identifiers of associated accounts), as well as the issuance time and

expiration time. The token is signed with the STS private key using one of the selected algorithms from the JWT family (e.g., HS256, RS256, EdDSA/Ed25519) and is then returned through User AuthN to the API Gateway. From that point onward, this token becomes the primary carrier of information about the user's authenticated identity for the entire system.

The API Gateway acts as a facade for clients and as a central router for internal services. All initial requests arrive at the gateway directly from a browser or a mobile application. If the user is not yet authenticated, the request is routed to User AuthN together with credentials (username and password, and other authentication attributes). After successful verification, User AuthN submits a request to the STS to issue a user JWT. The token received from the STS is returned to the API Gateway and then to the client, typically in a response header or as an HTTP cookie. Subsequently, the client attaches this JWT to every request (e.g., in the Authorization: Bearer header), and the gateway verifies its signature, validity, and lifetime. Only after successful validation is the request forwarded to the target functional service together with the identity token.

The User Service is responsible for the lifecycle of user accounts. It provides operations for creating and deleting users, changing passwords, updating profiles, and initiating login. Upon registration, a unique UserID is created and subsequently used as the primary identifier across all other services. During login, the User Service does not issue a token itself, but delegates this function to the STS via User AuthN. This centralizes security policies and allows signing algorithms or token formats to be changed without modifying business logic. The obtained JWT is stored on the client side.

The Account Service models the financial subsystem of the online store. Each user has one or more accounts linked to their UserID. The service supports opening and closing accounts, retrieving a list of accounts for a specific user, and changing account balances (debiting funds for purchases, deposits, refunds, etc.). All requests to the Account Service must contain a valid JWT. The service extracts the user identifier from the token and checks whether the user is authorized to perform the requested operation on the specific account. Thus, even if a request reaches the service directly

without passing through the gateway (e.g., as a result of an internal call from another service), authorization checks are still performed based on the same token.

The Product Service implements the product catalog. It stores product descriptions, prices, stock levels, and other attributes in its own database. The service provides operations for adding new products, updating characteristics and prices, retrieving product lists or detailed information for a specific productID, and modifying inventory when orders are placed. For most read operations, a valid user token is sufficient, whereas administrative actions (creating and editing products) require a specific role in the JWT (e.g., role = admin). This keeps authorization logic straightforward: the Product Service relies on the already validated token and does not consult a centralized session store.

The Basket Service is responsible for forming and managing orders. Each user has a separate basket identified by the UserID contained in the JWT. The service supports adding items to the basket, removing items, changing quantities, clearing the basket, and initiating checkout. When the user proceeds to payment, the Basket Service interacts with the Product Service (to verify availability) and the Account Service (to verify and reserve the required balance). All such internal interactions are accompanied by token propagation: in the simplest case, the same user JWT is used; however, the model also allows issuing service JWTs ("service JWTs") with restricted permissions that duplicate or refine the user context for inter-service calls.

A key characteristic of the model is that both external user access and internal calls between services follow the same token-based authentication principle. In Fig. 2.8 this is represented by two types of markers: the "User JWT" passed from the API Gateway to services A, B, C, etc., and the "Service JWT" used during inter-service interactions (e.g., when Service B calls Service E). This approach simplifies end-to-end security policy enforcement, enables new services to be added without changes to central authentication logic, and supports more advanced schemes, including nested tokens and delegated authorization.

For correct operation of the STS in the shop.com prototype, three key assumptions are formulated. First, the time synchronization problem is considered

solved: the clock of the node issuing tokens and the clocks of all nodes validating them are synchronized. This ensures correct interpretation of issuance time (iat), not-before time (nbf), and expiration time (exp) embedded in JWTs, and minimizes the risk of premature rejection or acceptance of expired tokens. Second, all channels used to exchange tokens are assumed to be protected by TLS or an equivalent mechanism. If the channel is not secure, a token may be intercepted and reused (replay attack) within its validity period. To mitigate this risk, token lifetimes in the prototype are limited to relatively short intervals. Third, the STS private key is stored in a secure environment and is inaccessible to attackers; otherwise, attackers could sign their own JWTs and present them as legitimate.

The token-based authentication mechanism selected for the Shop model is typical for modern distributed systems. The token acts as a compact cryptographic object that encodes all information required to confirm user identity and verify access rights. Token issuance is performed on the server side, while the client is responsible only for correctly transmitting tokens in subsequent requests. Due to token self-containment, microservices do not require a shared session store and do not maintain user state in memory; for each request, the context is reconstructed from the JWT. This substantially improves system scalability and simplifies deployment of new service instances.

The model accounts for the fact that token-based authentication can be implemented using different authorization approaches: role-based, attribute-based, and fine-grained access control. The STS prototype supports embedding both roles (e.g., USER, ADMIN, SUPPORT) and specific permissions into JWTs, which are interpreted directly by business services. For example, the Account Service may rely only on the user role, the Product Service may check an attribute permitting product creation, and the Basket Service may consider a maximum order amount limit encoded in the token. Thus, the prototype does not constrain authorization policy, but provides a flexible mechanism for implementing it.

The request-processing flow in the prototype can be summarized as follows:

1. An incoming HTTP request from the client arrives at the API Gateway. If the request already contains a JWT, the gateway immediately validates it (signature validity, non-expiration, absence of revoked identifiers, and other rules).

2. If the token is absent or validation fails, the user is redirected to User AuthN to perform login.

3. Based on data from the User Service, User AuthN submits a request to the STS to issue a new token.

4. The STS issues an access token, which is returned to the gateway and, depending on the scenario, to the client.

5. The gateway forwards the request to the appropriate functional service, attaching the JWT in the request headers.

6. The functional service re-validates the token (if required by policy) and executes the business operation.

7. If the operation requires calls to other services, the current service either forwards the user JWT or requests a service token from the STS and uses it for inter-service interactions.

The prototype implementation emphasizes that security cannot be evaluated solely from a performance standpoint, yet its impact on microservice performance is critical. Each JWT validation operation incurs additional computations: cryptographic signature verification, processing of header and payload fields, and analysis of authorization attributes. In a microservice architecture, where a single user request often triggers a cascade of calls across multiple services, the token may be validated repeatedly. For this reason, the Shop.com prototype model is subsequently used as a test environment to study the trade-off between security level and cryptographic overhead under different algorithms and token propagation strategies.

Therefore, the proposed JWT-based authentication service prototype for the Shop model demonstrates a comprehensive approach to security in a microservice architecture. It combines centralized token issuance in the STS, a single entry point via the API Gateway, autonomous business services (User, Account, Product, Basket, and

others), and end-to-end propagation of user identity using cryptographic tokens. The model clearly separates authentication and authorization concerns from business logic, ensures scalability and flexibility, and provides a foundation for further experiments measuring how different JWT implementation variants affect the performance of e-commerce microservices.

3.1. Model for Testing JWT Token Performance

The JSON Web Algorithms (JWA) specification is a key element of the JOSE standards family [34, 35], because it formally defines the permissible cryptographic algorithms for producing digital signatures and Message Authentication Codes (MACs) used in JWT and JSON Web Signature (JWS). In the JWT context, the alg header parameter indicates the selected signing or MAC algorithm, while JWA defines the allowed values for this parameter, usage rules, and implementation requirements. This design ensures interoperability between independent implementations and enables clients and servers to unambiguously interpret the cryptographic protection applied to a particular token.

In JWA, algorithms are grouped by cryptographic primitives. Symmetric algorithms include HMAC based on SHA-2 hash functions (HS256, HS384, HS512), which use a shared secret key and are typically applied in configurations where issuance and verification occur within a single trust domain. Asymmetric algorithms include RSA-based signature schemes with PKCS#1 v1.5 padding (RS256, RS384, RS512) and RSASSA-PSS schemes (PS256, PS384, PS512), which use a public/private key pair. A separate group comprises ECDSA algorithms (ES256, ES384, ES512), based on the P-256, P-384, and P-521 elliptic curves, combining compact parameters with a high level of cryptographic strength. The special value none corresponds to the absence of a digital signature or MAC and is intended only for strictly controlled scenarios.

Further evolution of the standards extended the JOSE algorithm set by introducing Edwards-curve signatures. Based on RFC 8037 [36] and RFC 8032 [35], the use of EdDSA with Ed25519 and Ed448 is defined. For EdDSA, the alg header value is EdDSA, while the specific curve (Ed25519 or Ed448) is specified in the JSON

Web Key (JWK) representation (via the key parameters). Table 3 summarizes the supported algorithms and their implementation requirements. In this work, this set of algorithms constitutes the experimental object for performance evaluation during JWT generation and validation.

Table 3

3) Signature and MAC Algorithms from RFC 7518 and RFC 8032

"alg" value	Digital signature / MAC algorithm	Implementation requirement
HS256	HMAC using SHA-256	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS1-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS1-v1_5 using SHA-384	Optional
RS512	RSASSA-PKCS1-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended
ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional
none	No digital signature or MAC	Optional
EdDSA	EdDSA (Ed25519 or Ed448, as specified in JWK) [34]	Recommended

Based on JWA and the RFCs defining Ed25519/Ed448, a generalized model for benchmarking JWT generation and validation is formulated. The model is intentionally implementation-agnostic: it defines an abstract set of entities, parameters, and measurement procedures that can be implemented in any software environment supporting the algorithms in Table 3. It specifies which token and cryptographic

properties must be recorded, in which modes experiments are conducted, and which metrics are collected for subsequent analysis.

The model is grounded in a typical JWT usage scenario in the web-oriented e-commerce system “Shop.” As shown in Figs. 11 – 13, the token structure includes a header with alg and typ, a payload containing a set of claims (user identifier, issuer, audience, issuance time, expiration time, scope, etc.), and a cryptographic signature produced according to the selected algorithm. The model assumes the same logical JWT structure across all experiments; differences are limited to the choice of alg, the key type, and the payload size.

Figure 11. Three main elements of a JWT token

Source of the figure: https://medium.com/@ilham76c/jwt-explained-a-complete-guide-to-its-structure-flow-and-security-considerations-2c9b379308f3

$$\mathrm{WT} = [header].[payload].[signature],$$

where $header = \mathrm{base64url}_{\mathrm{encode}}(′\{"\mathrm{alg}": "\mathrm{HS256}", "\mathrm{typ}": "\mathrm{JWT}"\}′)$

$payload = \mathrm{base64url}_{\mathrm{encode}}(′\{\mathrm{sub}: 131175321, \mathrm{name}: \mathrm{John}\}′)$

$$signature = base64url_{encode}\left(hmac_{sha256}(header + "." + payload, key_{secret})\right)$$

Description of experiment input parameters. The model fixes:

– the set of algorithms under test (alg values from Table 3, including EdDSA with Ed25519/Ed448 keys);

– operation mode: token generation (encode), token validation (verify), or a combined mode (encode→verify);

– key material characteristics: key type (symmetric, RSA, ECDSA, EdDSA) and key length or curve parameters;

– payload parameters: a baseline payload of fixed size and, if required, a set of payloads of different sizes to model “lightweight” and “heavy” tokens;

– the number of iterations per measurement and the number of repeated runs per algorithm/mode.

Description of the test environment. For each test series, the model requires recording platform characteristics: CPU model and clock frequency, number of cores/threads, RAM size, operating system version, runtime/interpreter version (e.g., JVM or Python), and versions of cryptographic libraries implementing JWA and EdDSA. These parameters are not part of the experimental procedure itself, but they are mandatory metadata to ensure comparability and reproducibility.

```
Header: Algorithm & Token Type
{
  "alg": "ES256",
  "typ": "JWT"
}

Payload: Data
{
  "sub": "131175321",
.."iss": "order-service.shop.com",
  "name": "Fuul Name",
  "aud": "gw.shop.com"
  "iat": 1760352633,
  "exp": 1760356233,
.."scope": "gw:auth"
}

Sign JWT: Private Key
{
  "kid": "3d570486-d154-4b15-bffd-5a1560fe5f22",
  "kty": "EC",
  "alg": "ES256"
  "use": "sig",
  "crv": "P-256",
  "x": "2KRDaXgtNJ1AJotZ8qpo_ghUJxIdyU99HhBzBpRPhPo",
  "y": "2ZcTq-CfwuJi1P_Gc2hB2cSazatPG0U71emHwOqvKFA"
  "d": "Bpg9RGkiUHyo5F9HnG9dm2c1fuPZYAcKgX2WjZRWTow",
}
```

Figure 12. Example of a JWT token and ES256 private key (JWK)

eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJzdWIiOiIxMzExNzUzMjEiLCJuYW1lIjoiRGFuaWVsIEt1ZHJpYSIsImFkbWluIjp0cnVlLCJpYXQiOjE3NjAzNTI2MzN9
.
Q923hWL19zStZFYONL_h-EhY9oxxZSuenH3t1WR-6dzLRz3bfMpHcBRZuFCJiKZTxzx_PSM8FWe6xr-WhuOtoQ

Figure 13. Example of a signed and Base64URL-encoded JWT

Procedure for generating test tokens. For each algorithm in Table 3, key material is pre-generated according to the relevant standard: a random secret key for HS256/HS384/HS512; a public/private key pair for RSASSA-PKCS1-v1_5 and RSASSA-PSS; EC keys on P-256/P-384/P-521 for ECDSA; and Ed25519/Ed448 key pairs for EdDSA. For each “algorithm–key type” combination, a baseline JWT header is formed with the corresponding alg value and typ="JWT", together with one or more representative payload variants that reflect real scenarios (user authentication and authorization to access services such as order-service, payment-service, gw.shop.com, etc.).

Execution time measurement scheme. The model assumes high-resolution monotonic timers to record the start and end of an operation series. In encode mode, a new JWT is generated iteratively using the fixed header and payload; fields such as iat and exp may be updated to approximate realistic token issuance behavior. In verify mode, one or more valid tokens are pre-generated and stored; the benchmark loop performs only signature and integrity verification without generating new tokens.

Performance metrics. Each experiment must report:

total execution time for the iteration series for the selected algorithm and mode;

number of operations (iterations), enabling throughput calculation (operations per second);

average time per operation (e.g., in microseconds);

token configuration parameters (algorithm, key type, payload size).

If required, the model allows additional metrics (e.g., memory consumption or CPU utilization), but time-based metrics are treated as baseline.

Result storage structure. All measurements are stored in tabular form (e.g., CSV), where each row corresponds to a single experiment or a single iteration series. Mandatory fields include: algorithm identifier (alg), test mode (encode, verify, encode→verify), iteration count, total execution time, derived metrics (ops/s, average time/op), key characteristics, and a concise description of the test environment. This format aligns measurements with Table 3 and binds each record to a specific JWA/EdDSA algorithm.

Thus, the proposed performance testing model for JWT generation and validation relies on the formal requirements of JWA and related RFCs and defines a unified framework for experiments comparing the algorithms in Table 3. It specifies the algorithm set, operating modes, token parameters, environment metadata, timing procedures, and result storage format without being tied to any particular implementation.

3.2. Justification for Choosing the Flask Web Framework for Modeling a Microservice Architecture

To implement the software model of the microservice system within the scope of the master's thesis, the Flask microframework (Python) was selected. Flask is a lightweight WSGI (Web Server Gateway Interface) web framework that follows a philosophy of minimalism and extensibility. Its architectural core is based on the Werkzeug[1] library (routing and WSGI request handling) and the Jinja2 templating engine. Unlike full-stack frameworks, Flask does not impose a rigid project structure or prescribe specific tools for database access, which makes it particularly suitable for academic modeling where flexibility and component isolation are critical. Table 4 summarizes the advantages and disadvantages of Flask for use in microservice-based systems.

[1] https://werkzeug.palletsprojects.com/en/stable/

Table 4

Characteristics of the Flask Framework in the Context of Microservices

Advantages	Disadvantages
Architectural flexibility. Flask allows the developer to choose design patterns independently. This is critical for modeling microservices because each service may require a distinct set of libraries (e.g., one service uses SQL, another uses NoSQL, and a third is purely computational).	Limited native asynchrony. Although Flask 2.0+ introduced support for async view functions, its asynchronous capabilities are generally less efficient under high I/O workloads than those of frameworks designed around an async runtime (e.g., FastAPI).
Lightweight nature. Minimal startup overhead makes it possible to deploy many microservice models on limited hardware resources (e.g., on a developer's local machine when testing service interactions).	"Empty box" problem. Basic concerns (ORM, data validation, authorization) must be configured manually, which may increase initial setup time if reusable templates are not available.
Low entry barrier. The simplicity of syntax allows focusing on business logic and service interaction algorithms rather than on extensive framework configuration.	
Mature ecosystem. A wide range of extensions (Flask-RESTful, Flask-SQLAlchemy, Flask-Migrate) enables rapid integration of required functionality.	

A comparison with the most common alternatives in the Python ecosystem—Django and FastAPI—is provided in Table 5.

Table 5

4) Comparison of Web Frameworks in the Python Ecosystem

Characteristic	Flask	Django	FastAPI
1	2	3	4
Type	Microframework	Full-stack (monolithic)	Microframework (async)
Architecture	Modular, extensible	"Batteries included"	Asynchronous, based on Starlette

Continuation of Table 5

1	2	3	4
MVP development speed	High (for simple services)	High (for standard CMS/CRUD)	Very high (automatic documentation generation)
Performance	Medium	Lower (higher overhead)	High
Suitability for microservices	High (logic isolation)	Low (excessive coupling)	High

Django is primarily designed for monolithic applications. Using it for a microservice where only an API endpoint is required is often excessive, as it introduces a substantial set of additional components (e.g., an admin interface and a heavyweight ORM), which conflicts with the lightweight principle typical of microservices.

FastAPI is a strong competitor and a leader in performance. However, for modeling purposes, Flask offers advantages in stability and ease of debugging. FastAPI also encourages strict typing (via Pydantic) and assumes familiarity with asynchronous programming, which can complicate rapid prototyping when the researcher is focused on interaction logic rather than high-load optimization.

Given the requirements for flexibility, rapid prototype development, and the need to demonstrate component interaction principles, Flask is an appropriate choice. It provides a balanced combination of functionality and simplicity, allowing the research to focus on architectural modeling rather than on the complexity of the tooling.

3.3. Analysis of Database Technologies for Microservice Architecture and Justification for Selecting the Modeling Toolset

In modern software engineering, microservice architecture entails decentralized data management. According to the Database-per-Service pattern, each microservice must own its own data store, which is accessed exclusively through its API. This ensures loose coupling between components and enables selecting the database technology that best fits the nature of a particular service’s data.

In production environments, this approach is most commonly implemented using robust client–server database management systems (DBMSs), such as:

PostgreSQL and MySQL as a standard choice for transactional data requiring ACID compliance;

MongoDB or Cassandra for unstructured data and high availability in distributed clusters;

Redis for caching and fast access to key–value data.

However, when designing a research environment and modeling microservice interactions, the use of full-scale server-based DBMSs often introduces excessive complexity that is not justified by the research objectives. Deploying a separate PostgreSQL instance or a MongoDB cluster for each modeled service requires substantial computational resources (CPU, RAM), network configuration, containerization, and administration. This shifts the focus from studying algorithms and architectural patterns to addressing infrastructure (DevOps) concerns.

Given these constraints, SQLite was selected as the baseline data management system for this study. The rationale for choosing SQLite for modeling is as follows:

architectural compliance with the isolation principle: SQLite is an embedded, serverless library that stores the database as a regular file on disk. This enables physical enforcement of data isolation for each microservice: each service operates on its own .db file. This directly emulates the Database-per-Service pattern without the need to maintain dozens of network connections [37];

lightweight nature and performance: SQLite does not require a separate server process. Read and write operations are performed via direct library calls, minimizing overhead from inter-process communication (IPC) and eliminating network latency. This allows complex multi-service interaction scenarios to be executed and tested even on a researcher’s local machine with limited resources;

full-featured sql support: Despite its compactness, SQLite supports most of the SQL-92 standard, including ACID transactions, triggers, complex queries, and indexes. This enables development of business logic that remains syntactically and logically compatible with “larger” DBMSs (e.g., PostgreSQL). Code written against SQLite via an ORM (e.g., SQLAlchemy) can later be migrated to a production DBMS without architectural changes;

simplified experiment reproducibility: Because the entire database is contained in a single file, creating backups, resetting the system state, or transferring test datasets for reproducing experimental results reduces to copying the file. This is essential for research, where reproducibility is a key requirement;

support for modern capabilities: Recent SQLite versions, particularly with Write-Ahead Logging (WAL), provide improved read concurrency, which is sufficient for load simulation in a modeling setting. In addition, extensions (e.g., sqlite-vec) enable experimentation with vector search for AI-related components.

Thus, selecting SQLite represents a strategically justified compromise: it enables a "clean" microservice architecture with strict data isolation while avoiding infrastructure overhead. This allows the research to focus on service interaction logic, data consistency, and process orchestration, which are central to this master's thesis.

3.4. Selection of a JWT Token Support Library for the Python Platform

To implement authentication and authorization in the microservice system, the PyJWT library (jpadilla/pyjwt) was selected as a widely adopted JWT implementation for Python. This choice is driven by the need for rapid prototyping and microservice-architecture modeling within the master's thesis, where ease of integration, the availability of documentation and examples, and compliance with RFC 7519 [32] are critical factors.

PyJWT is an open-source library that supports encoding and decoding JWTs in accordance with RFC 7519. It provides practical support for the main JOSE/JWA signing families used with JWT in real systems: symmetric HMAC (HS256/HS384/HS512), RSA signatures (RS* and PS*), elliptic-curve signatures (ES*), and EdDSA (with Ed25519/Ed448 key types, depending on the cryptographic backend). This breadth allows selecting a security–performance trade-off appropriate to the experimental objectives.

From a maintenance and adoption perspective, PyJWT is actively released on PyPI (e.g., 2.10.1 published in late November 2024, and 2.11.0 published on January 30, 2026) and has a large public user base (deps.dev reports ~22k public dependents, which is a reasonable proxy for ecosystem adoption). (PyPI) On GitHub, the repository

is also among the most starred JWT libraries in the Python ecosystem (≈5.6k stars in GitHub) The minimalist API (jwt.encode(), jwt.decode()) and straightforward installation via pip further support rapid prototyping [38].

A comparative analysis with alternative Python libraries – python-jose, jwcrypto, and Authlib – is provided in Table 6. The star counts below reflect GitHub repository metadata as of February 2026.

Table 6

5) Characteristics of JWT Libraries on the Python Platform

Feature	jpadilla/pyjwt[2]	mpdavis/python-jose[3]	latchset/jwcrypto[4]	lepture/authlib[5]
GitHub stars	~5.6k	~1.7k	~473	~5.2k
Sign	+	+	+	+
Verify	+	+	+	+
`iss`	+	+	+	+
`sub` (built-in validation)	limited / app-level	+	+	+
`aud`	+	+	+	+
`exp`	+	+	+	+
`nbf`	+	+	+	+
`iat`	+	+	+	+
`jti` (replay semantics)	app-level	+	+	+
typ	implementation-dependent	implementation-dependent	+	implementation-dependent
HS256/384/512	+	+	+	+
RS256/384/512	+	+	+	+
ES256/256K/384/512	+	+	+	+
PS256/384/512	+	limited	+	+
EdDSA	+	unclear/limited	+	+

[2] https://github.com/jpadilla/pyjwt/
[3] https://github.com/mpdavis/python-jose/
[4] https://github.com/latchset/jwcrypto/
[5] https://github.com/lepture/authlib

Interpretation of the comparison. python-jose provides broad functionality and aims at a more complete JOSE scope (JWS/JWE/JWK/JWA), which can be excessive when the research goal is specifically JWT issuance and verification rather than full JOSE coverage. jwcrypto is a cryptography-focused JOSE implementation (including JWE), but its design emphasis on completeness can increase complexity for a JWT-centric prototype. Authlib is primarily an OAuth 2.0 / OIDC framework; while it includes JOSE/JWT features, its lifecycle considerations matter: Authlib documentation explicitly states that authlib.jose is being deprecated in favor of joserfc.

Overall, PyJWT offers the most favorable functionality-to-complexity ratio for the thesis objectives: it is simple to integrate, sufficiently feature-complete for JWT-centric microservice experiments, actively released, and widely adopted. (GitHub) Potential gaps in strict claim semantics (e.g., project-specific `sub` constraints or jti replay protection) are typically handled at the application layer and are not critical for modeling baseline JWT authentication scenarios.

3.5. Containerized Deployment of Microservices. Docker

A production environment must provide four key capabilities:

service management interface – enables developers to create, update, and configure services. Ideally, this interface is exposed as a REST API invoked via command-line tools and GUI deployment tools;

runtime service management – ensures continuous operation of the required number of service instances. If a service instance fails or cannot handle requests, the production environment must restart it. If a host machine fails, the environment must restart the affected service instances on another machine;

monitoring – provides developers with visibility into service behavior, including logs and metrics. If issues occur, the production environment must notify developers.

request routing – routes user requests to the appropriate services.

Based on [39], the choice of Docker as the baseline containerization technology for implementing a microservice architecture is justified primarily by the fact that microservices increase the number of independent components and, consequently, the number of environments, dependencies, and integration points that must be managed.

In team development, this complexity grows rapidly: differences in operating systems (Windows/macOS/Linux), language and library versions, and configurations across local machines, test servers, and production environments create the classic “works locally, but fails elsewhere” risk. Docker mitigates this risk by encapsulating each microservice in a container with fixed dependencies and configuration, ensuring runtime reproducibility and reducing conflicts between development and server environments. For microservices, the key benefits are OS-level isolation, standardized runtime conditions, advantages over traditional virtualization, and support for subsequent orchestration, which is a baseline requirement for operating distributed systems.

Docker Compose provides a flexible interface for managing multi-container applications through a declarative approach to describing and orchestrating distributed services. It offers a comprehensive set of commands for full lifecycle management of containers.

Main Docker Compose commands include:

docker-compose up – creates and starts all services defined in the configuration file;

docker-compose down – stops and removes containers, networks, and volumes;

docker-compose start / docker-compose stop – manages the state of services without deleting resources;

docker-compose ps – displays the status of all containers in the project;

docker-compose logs – provides access to aggregated logs of all services;

docker-compose exec – executes commands in running containers;

docker-compose build – rebuilds service images;

docker-compose pull – downloads the latest image versions from the registry;

docker-compose config – validates and checks the configuration file.

The commands support modular options for flexible configuration, such as:

-f – select an alternative configuration file;

--env-file – load environment variables;

--scale – dynamically scale specific services.

This interface integrates automation mechanisms for sequential deployment of services while accounting for inter-service dependencies, which significantly simplifies managing complex microservice architectures during development and testing.

4. Modeling of the theoretical research results achieved.

4.1. Implementation of a JWT Token-Based Authentication Model for the Web-Oriented Microservice E-Commerce System “Shop”

The Basket Service interacts with the Inventory Service and the Account Service. To place an order, it first retrieves detailed product information and then checks the available balance by contacting the Account Service. The service APIs are presented in Tables 6 – 11.

Table 6

6) Account Service REST API

HTTP Method	Endpoint	Input Data	Function
GET	/	–	Returns a list of all service endpoints.
GET	/accounts	userID (optional query parameter)	Retrieves all accounts or accounts for a specific user.
POST	/accounts	{"userID": int}	Creates a new account for a user with an initial balance.
GET	/accounts/<acc_num>	acc_num (path parameter)	Retrieves an account by number.
DELETE	/accounts/<acc_num>	acc_num (path parameter)	Closes (deletes) an account.
POST	/accounts/<acc_num>	{"amount": int}	Changes the account balance by the specified amount.
POST	/accounts/user/<user_id>/update_balance	{"amount": int}	Updates the user’s account balance by userID.
GET	/accounts/user/<user_id>	user_id (path parameter)	Retrieves the user’s account balance.

Table 7

7) User Service REST API

HTTP Method	Endpoint	Input Data	Function
GET	/	–	Returns a list of all service endpoints.
GET	/users	username (optional query parameter)	Retrieves all users or a specific user by username.
POST	/users	{"username": str, "pwd": str}	Registers a new user.
DELETE	/users/userID/<userID>	userID (path parameter)	Deletes a user by ID.
POST	/users/login	{"username": str, "pwd": str}	Authenticates a user (returns a token).

Table 8

8) Product Service REST API

HTTP Method	Endpoint	Input Data	Function
GET	/	–	Returns a list of all service endpoints.
GET	/products	–	Retrieves all products.
POST	/products	{"name": str, "price": float, "stock": int}	Adds a new product.
GET	/products/<product_id>	product_id (path parameter)	Retrieves product information.
PUT	/products/<product_id>/updatestock	{"stock": int}	Updates product stock quantity.

Table 9

9) Basket Service REST API

HTTP Method	Endpoint	Input Data	Function
GET	/	–	Returns a list of all service endpoints.
GET	/basket	–	Retrieves all baskets for all users.
POST	/basket/<user_id>/<product_id>	{"quantity": int}	Adds a product to the user's basket.
DELETE	/basket/<user_id>/<product_id>	–	Removes a product from the user's basket.
GET	/basket/<user_id>	–	Retrieves the user's basket.
POST	/basket/checkout/<user_id>	–	Places an order (debits balance, updates inventory).

Table 10

10) Gateway Service REST API

HTTP Method	Endpoint	Input Data	Function
GET	/	–	Returns a list of all service endpoints.
GET	/users	username (optional query parameter)	Retrieves user information.
POST	/users	{"username": str, "pwd": str}	Registers a user.
GET	/users/<userID>/accounts	–	Retrieves user accounts.
POST	/users/<userID>/accounts	–	Opens a new account for the user.

Continuation of table 10

GET	/users/<userID>/accounts/<accNum>/transactions	–	Retrieves account transactions.
POST	/accounts/<userID>	–	Retrieves the user's account balance.
POST	/cart/checkout/{user_id}	{"cart": object}	Places an order via the basket.
POST	/login	{"username": str, "pwd": str}	Authenticates a user.
POST	/logout	–	Logs out a user (not implemented).

Table 11

11) STS (Security Token Service) REST API

HTTP Method	Endpoint	Input Data	Function
GET	/	–	Returns a list of all service endpoints.
POST	/login	{"username": str}	Issues a JWT for a user (without password verification).

The system is implemented in Python 3.13. All HTTP methods are exposed via REST APIs implemented using Flask. The number of worker processes per service can be configured. The Account and User services maintain separate SQLite databases, which keeps services loosely coupled. To simplify the experimental setup, each microservice is deployed as a single instance and can run in isolation within a container.

The experiments were conducted on a Windows workstation using Docker Desktop (Linux-based containers). The host system configuration was: Windows 10; Intel Core i7-6700 CPU @ 3.40 GHz (4 cores, 8 logical processors); 16 GB RAM.

The C4 component diagram (see Fig. 14) illustrates the deployment and communication protocols among the Shop services.

The implementation supports multiple users operating either concurrently or sequentially. When the test client increases the request rate, the application uses multi-process execution. Each service uses its own database and resources. The Shop e-commerce system is implemented as a set of microservices and follows a composite architectural model.

The test client registers two users and opens accounts for them. Initial balances are then credited sequentially. After that, load testing is performed for 10, 100, 500, 1,000, 2,500, and 10,000 requests. These requests are sent to both users either concurrently (in parallel) or sequentially to execute checkout operations. For each user, a basket containing a predefined number of items is created, after which a series of checkout operations is performed.

For each load configuration, the experiment is conducted in two modes: (1) without security mechanisms enabled and (2) with JWT-based authorization. To evaluate system performance on the client side, the average execution time of checkout requests is calculated. In addition, total response time and throughput of checkout operations across all requests are recorded. The checkout operation involves interaction among four microservices (see Fig. 3.1). The duration of a single operation is affected by network latency and processing time inside the microservices, including database access time. In the specified configuration, successful execution is possible only if the user’s account has sufficient funds; otherwise, the operation is not performed.

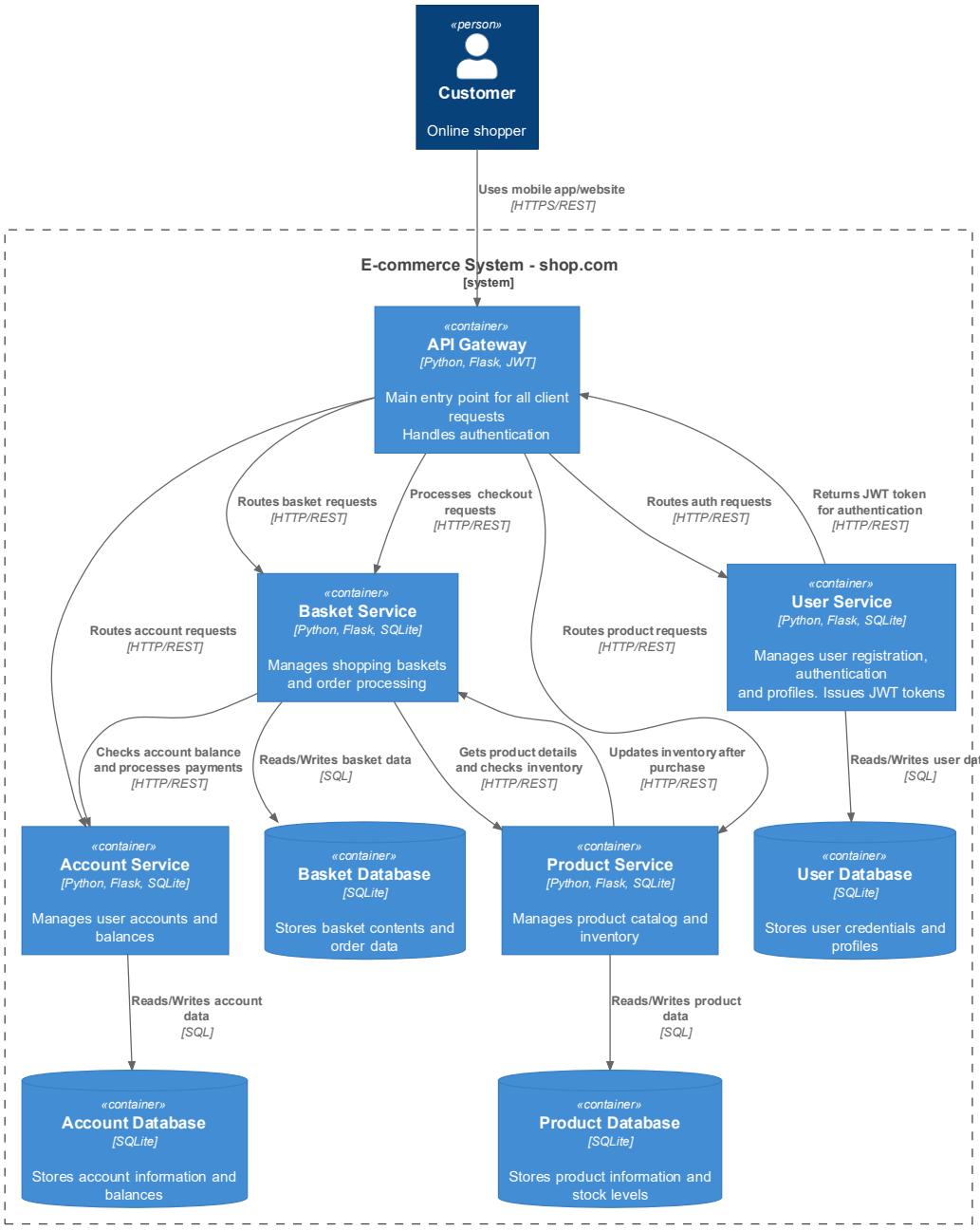

Figure 14. Component diagram of service deployment and communication in the Shop model

Source of the figure: original

4.2. Implementation of an Environment Model for Testing JWT Performance Across Cryptographic Algorithms

An environment model was implemented to evaluate the performance of JWT token generation (encode) and validation (verify) operations for a set of cryptographic algorithms from the HMAC, RSA, RSASSA-PSS, ECDSA, and EdDSA families. A

simulation–statistical approach was adopted: for each algorithm, a series of identical operations with a fixed number of iterations is executed, and aggregate indicators are obtained by averaging results across repeated runs.

The baseline assumptions are that testing is performed in an isolated containerized environment with fixed resource constraints, background system load is negligible, and the impact of network latency and input/output operations is minimized. A limitation of the model is that it reproduces a typical, but not exhaustive, range of industrial web-application deployment scenarios.

he experimental platform is based on Docker Desktop running on Windows 10 on a workstation equipped with an Intel® Core™ i7-6700 CPU @ 3.40 GHz (4 physical cores, 8 logical threads) and 16 GB of RAM. To enhance reproducibility, the Docker container running the load-testing module was strictly constrained to 1 GB RAM and 1 virtual CPU. This configuration simulates the deployment of an individual microservice under resource constraints.

Input data preparation includes generating key material (secrets for HMAC; public/private key pairs for RSA/RSASSA-PSS and ECDSA; and key pairs for EdDSA), and forming a unified JWT header and payload that emulate typical authentication and authorization claims. For each “algorithm–mode (encode/verify)” combination, the number of iterations, payload size, and key parameters are specified.

The model is implemented in Python using the PyJWT library with a cryptographic backend based on the cryptography library. The module employs a high-resolution monotonic timer to measure total execution time, calculates average per-operation latency and throughput (operations per second), and automatically stores results in CSV/JSON formats.

4.3. Description of modeling results.

4.4. Load Modeling on the Authentication Subsystem with JWT Tokens

The experimental study aimed to assess the impact of JWT tokens on the operation of the authentication service in a web-oriented system. For comparison, two resource-access modes were considered:

- sequential requests without security mechanisms enabled;

– sequential requests with JWT token validation enabled.

The load was generated by two users; however, the implementation supports scaling the number of clients. Before the experiments, baseline system parameters were recorded: CPU utilization was approximately 0.03%, and each microservice operated in a Docker container limited to 1024 MB of RAM and one CPU core. This made it possible to study system behavior under controlled resource conditions.

Tables 12 and 13 present the measured total response time for different numbers of sequential requests, as well as the calculated throughput in transactions per second (TPS). In the mode without security functions enabled, 10 requests resulted in a total response time of 0.04 s with a throughput of 250 TPS; as the number of requests increased to 10,000, the total response time rose to 27.23 s and throughput to 367 TPS. In the mode with JWT validation enabled, 10 requests produced 0.043 s and 233 TPS, while 10,000 requests yielded 30.27 s and 330 TPS, respectively. Thus, across all experimental sets, the security-enabled configuration demonstrates a slightly higher total processing time and a slightly lower throughput.

The measurement results for the mode without safety functions enabled are shown in Table 12.

Table 12

Sequential requests without security mechanisms enabled

Total number of requests	Total response time (s)	Throughput (TPS)
10	0.04	250
50	0.156	321
100	0.34	294
500	1.73	289
1,000	2.61	383
2,500	6.63	377
10,000	27.23	367

Similar results for the mode with the JWT security feature enabled are shown in Table 13.

Table 13.

12) Sequential requests with JWT validation enabled

Total number of requests	Total response time (s)	Throughput (TPS)
10	0.043	233
50	0.196	255
100	0.383	261
500	1.92	260
1,000	3.078	325
2,500	7.89	317
10,000	30.27	330

The relationship between the total response time and the number of consecutive requests in both modes is shown in Fig. 15.

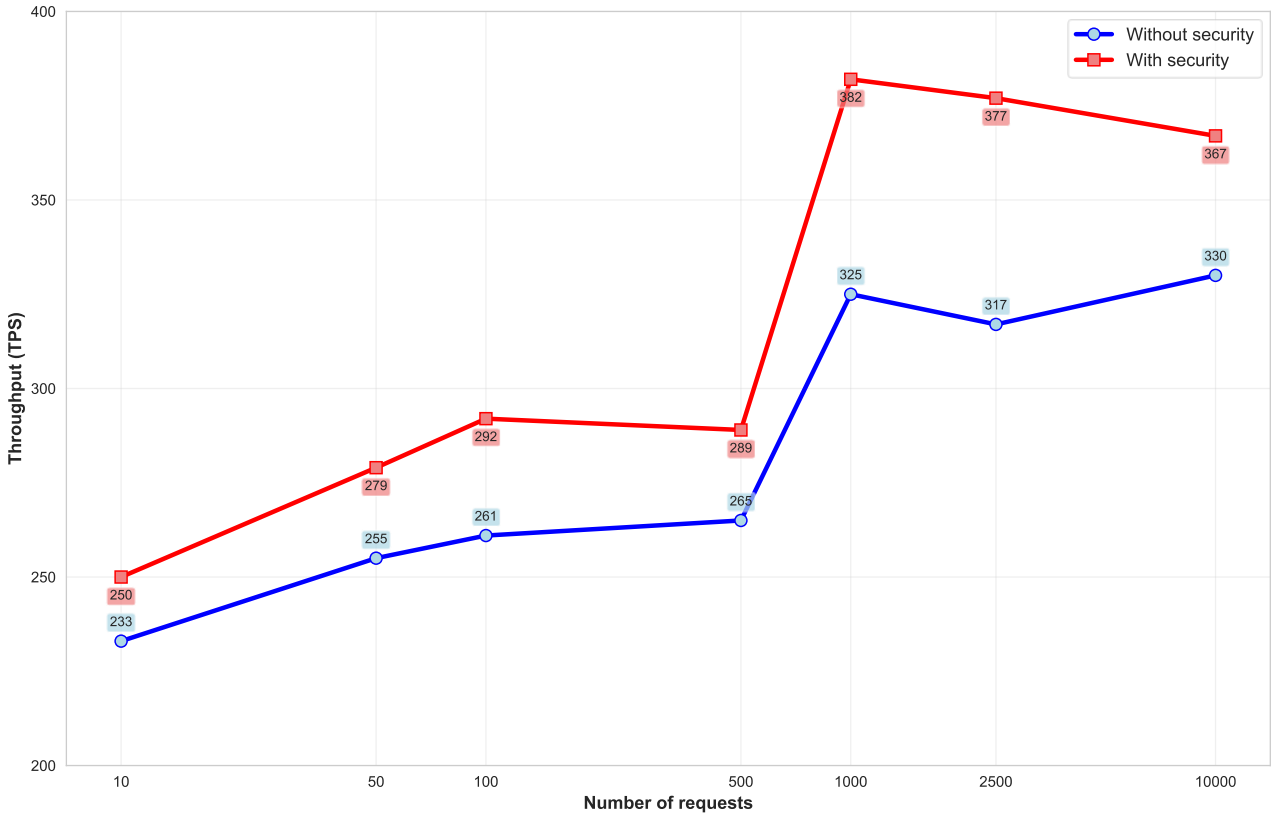

Figure 15. Total response time versus the number of sequential requests (with and without JWT validation)

Source of the figure: original

Both curves exhibit a pronounced nonlinear pattern. Under low load (10–100 requests), response time increases slowly, remaining below one second, and the difference between the secured and unsecured modes is negligible. Starting from 500 requests, growth becomes steeper: for 1,000 requests, the time with JWT validation exceeds three seconds, whereas without security it is approximately 2.6 s. At 10,000 requests, the delay is the largest, and the difference between modes reaches several seconds. The data indicate additional overhead from JWT validation, averaging approximately 10–20% of the total request-processing time. At the same time, the absolute magnitude of this difference remains moderate even under maximum load.

The comparative throughput (TPS) for identical experimental sets is shown in Fig. 16.

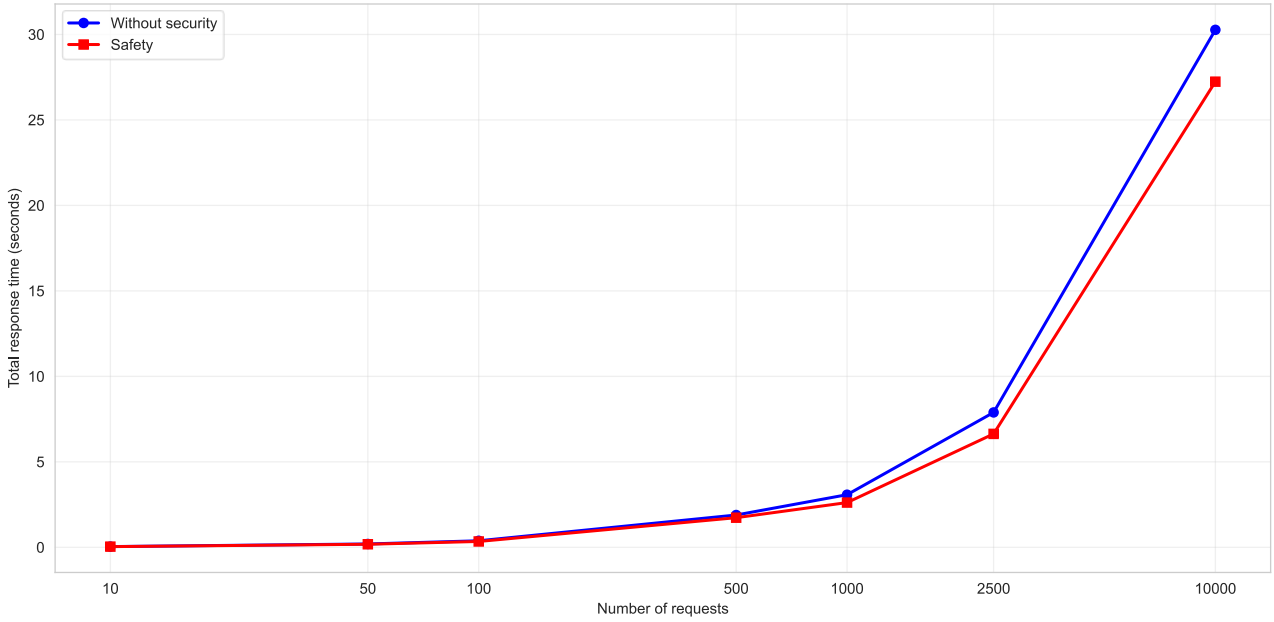

Figure 16. Throughput (TPS) comparison (with and without JWT validation)

Source of the figure: original

The curves for the secured and unsecured modes are close, confirming the limited impact of JWT validation on peak processing speed. In the initial range (10–100 requests), TPS increases moderately in both modes due to more efficient CPU utilization as the number of transactions grows. For 500–1,000 requests, throughput

reaches its maximum: above 380 TPS without security and above 320 TPS with JWT. Increasing the number of requests to 2,500 and 10,000 does not cause a substantial drop in TPS, indicating the absence of obvious bottlenecks in the implemented architecture. The relative throughput decrease in the JWT-enabled mode compared to unrestricted access does not exceed approximately 10–15%.

In parallel with response time and throughput, CPU load and memory usage were measured. As the number of requests increased, CPU utilization rose as expected, approaching upper bounds at 2,500–10,000 requests. However, values remained within a range that ensured stable service operation, and switching from the unsecured mode to JWT validation did not produce a multiplicative increase in CPU load. Memory consumption changed only slightly in both configurations, as request/response payload sizes were fixed and token lifetimes were relatively short; this enabled experiments to be conducted in containers with strict memory limits.

In summary, using JWT tokens to secure sequential requests leads to a moderate increase in response time and a slight reduction in throughput, but does not impose critical CPU or memory load even for tens of thousands of requests. The observed overhead is acceptable for most web applications where authentication and data integrity are prioritized. Therefore, JWT-based access control can be considered a balanced solution between system performance and security requirements.

4.5. Results of Modeling JWT Token Performance Across Cryptographic Algorithms

This section presents the results of performance modeling for JWT token generation (encode) and validation (verify) operations for the selected cryptographic algorithms HS256, ES256, EdDSA (Ed25519), RS256, and PS256. The experiment was conducted as a series of 50 trials (runs) for each algorithm; within each trial, 100 iterations of token creation and signature verification were performed, enabling statistically stable averaged indicators. In parallel with timing characteristics, CPU utilization and RAM consumption of the Docker container hosting the performance testing module were recorded; during the tests, the allocated virtual CPU core was fully utilized.

Testing was performed under a workload of 50 trials × 100 iterations per trial for the main algorithms ES256, EdDSA, HS256, PS256, and RS256. Under these conditions, container memory consumption and CPU usage approached their maximum values. The results are presented in Tables 14 – 17, and their visualizations are shown in Figs. 17 – 19.

Table 14

13) Average JWT encoding time for signing algorithms HS256, ES256, EdDSA, PS256, and RS256 (50 experiments, 100 tests each)

№	ES256	EdDSA	HS256	PS256	RS256	№	ES256	EdDSA	HS256	PS256	RS256
1	2	3	4	5	6	1	2	3	4	5	6
1	0.06025	0.10396	0.05337	0.64549	0.57444	26	0.06179	0.06947	0.03144	0.66356	0.60384
2	0.06926	0.08088	0.04444	0.62325	0.61790	27	0.05479	0.06068	0.03282	0.70024	0.60813
3	0.07084	0.09522	0.04027	0.62870	0.63756	28	0.06254	0.05646	0.03179	0.63741	0.63457
4	0.06798	0.10038	0.03895	0.62532	0.55204	29	0.06851	0.05643	0.03086	0.56239	0.61937
5	0.07661	0.08069	0.03640	0.56110	0.55926	30	0.07124	0.06792	0.03167	0.59093	0.56166
6	0.06464	0.07918	0.03676	0.59891	0.56766	31	0.07266	0.06835	0.03329	0.70063	0.55917
7	0.05910	0.09964	0.03400	0.61242	0.62927	32	0.06468	0.06986	0.03341	0.64476	0.86553
8	0.06448	0.08379	0.03313	0.56697	0.66046	33	0.05852	0.08392	0.03276	0.65107	1.19382
9	0.06591	0.08028	0.03378	0.61122	0.57786	34	0.07085	0.07161	0.03162	0.58054	0.83847
10	0.06248	0.07494	0.03162	0.59768	0.59678	35	0.06747	0.06834	0.03266	0.57925	0.93222
11	0.07958	0.07471	0.03257	0.62826	0.60186	36	0.07036	0.08095	0.03212	0.67512	1.21455
12	0.06423	0.07977	0.03322	0.68066	0.60077	37	0.06083	0.06684	0.03269	0.62844	0.92167
13	0.05776	0.07547	0.03087	0.56481	0.57232	38	0.06622	0.08865	0.03227	0.60600	0.70097
14	0.05829	0.07032	0.03276	0.65926	0.56853	39	0.06192	0.10305	0.03201	0.65298	0.64792
15	0.05864	0.07049	0.03052	0.57739	0.55452	40	0.06585	0.06544	0.03233	0.69768	0.63142
16	0.05747	0.07433	0.02862	0.58888	0.61042	41	0.06497	0.07059	0.03145	0.62680	0.60631
17	0.05928	0.06971	0.03628	0.57806	0.63676	42	0.06603	0.07100	0.03163	0.62369	0.58624
18	0.08611	0.06861	0.03312	0.58610	0.59658	43	0.06320	0.06162	0.02880	0.57408	0.66438
19	0.06263	0.06792	0.03456	0.60972	0.71413	44	0.06334	0.05956	0.03069	0.56715	0.57567
20	0.07182	0.06401	0.03025	0.57973	0.62363	45	0.06928	0.07008	0.03037	0.65034	0.79267
21	0.06666	0.06433	0.02934	0.61542	0.74352	46	0.07535	0.06761	0.03134	0.60824	0.57069
22	0.05320	0.06512	0.03070	0.64058	0.60483	47	0.07390	0.09140	0.03127	0.56374	0.56861
23	0.06391	0.06588	0.03179	0.62099	0.64152	48	0.07328	0.06387	0.02966	0.59938	0.60291
24	0.05202	0.06708	0.03229	0.66363	0.62917	49	0.06752	0.06184	0.03132	0.62383	0.55654
25	0.05731	0.06730	0.03225	0.64811	0.68206	50	0.06881	0.06370	0.03145	0.59300	0.54738

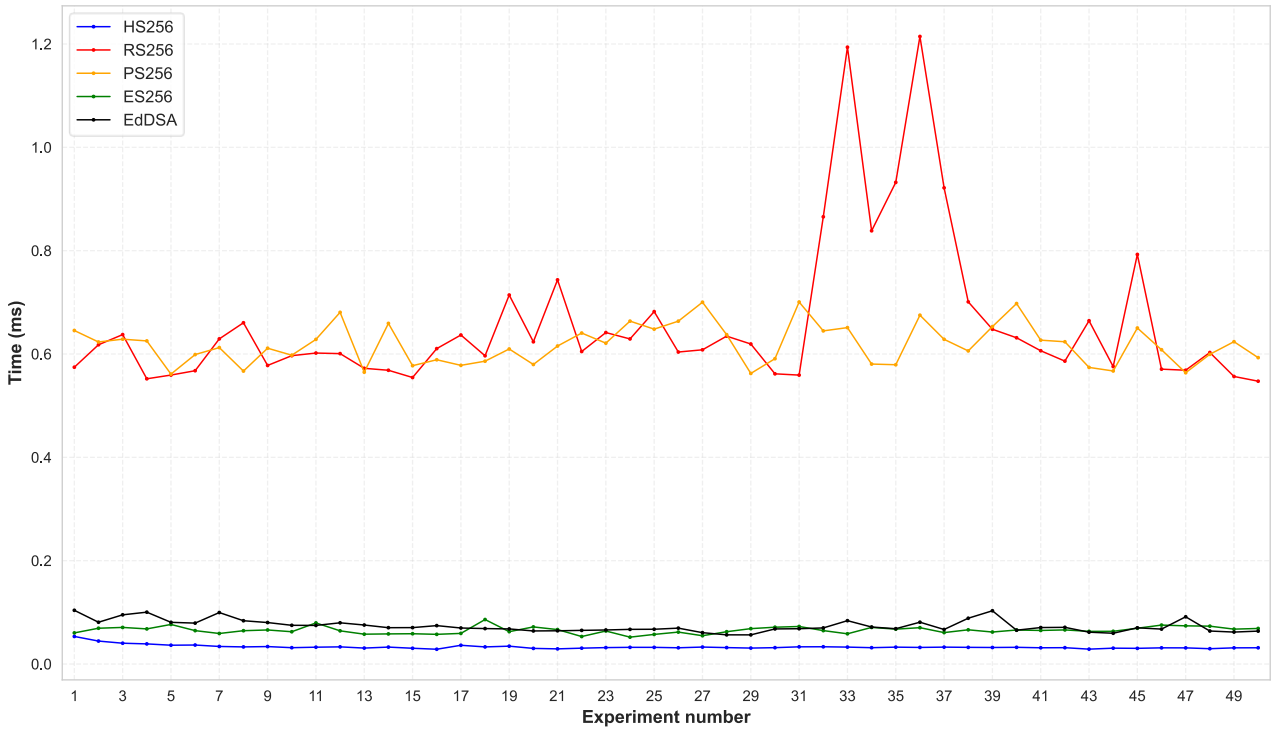

Figure 17. Comparison of JWT signing algorithms by encoding time (encode)

Source of the figure: original

Within the experiment, the execution time of the JWT encoding (signing) operation was measured for five algorithms: HS256, ES256, EdDSA (Ed25519), and the RSA-based algorithms RS256 and PS256. For each mode, a series of 50 runs was conducted with a fixed payload size and identical hardware parameters. The table reports the average time per operation (microseconds), and the corresponding plot is provided in milliseconds to enable a visual comparison of the selected algorithms.

The results demonstrate a clear performance hierarchy. The symmetric algorithm HS256 exhibits the shortest encoding time: average values are approximately 0.03 – 0.04 ms per operation, with only minor variation across runs. Elliptic-curve algorithms ES256 and EdDSA are moderately slower: typical values for ES256 are around 0.06 – 0.07 ms, and for EdDSA around 0.07 – 0.09 ms per operation. Thus, ES256 and EdDSA are roughly 2 – 3× slower than HS256, but remain in the sub-millisecond range, which is acceptable for high-load systems.

The lowest encoding performance is observed for RS256 and PS256. Their average encoding time typically falls within 0.55 – 0.70 ms, and in some runs exceeds 0.8 – 1.0 ms, i.e., approximately 10 – 20× slower than HS256 and several times slower than ES256 and EdDSA. PS256 is generally slightly slower than RS256 due to the higher computational complexity of the PSS padding scheme.

Overall, the plot reflects a stable efficiency ordering: HS256 is the fastest, ES256 and EdDSA form an intermediate group, while RS256 and PS256 lag substantially and form the highest-latency group. Therefore, when latency and throughput are critical, HS256 is preferable (provided a symmetric key model is acceptable) or ES256/EdDSA can be used as a compromise between performance and cryptographic strength. RS256/PS256 are more appropriate primarily for compatibility requirements rather than for maximum speed.

Table 15

14) Results of measuring JWT encoding throughput for algorithms HS256, ES256, EdDSA, RS256, and PS256 (50 experiments, 100 tests each)

№	ES256	EdDSA	HS256	PS256	RS256	№	ES256	EdDSA	HS256	PS256	RS256
1	16598.25	9618.84	18736.06	1549.21	1740.81	26	16184.15	14393.81	31805.97	1507.03	1656.06
2	14437.98	12364.55	22500.35	1604.49	1618.38	27	18250.37	16480.16	30473.84	1428.08	1644.39
3	14115.93	10502.48	24834.01	1590.59	1568.47	28	15989.08	17711.44	31456.40	1568.86	1575.88
4	14709.38	9962.27	25677.10	1599.17	1811.46	29	14596.46	17721.63	32407.98	1778.13	1614.54
5	13052.72	12392.78	27471.70	1782.20	1788.06	30	14036.26	14722.33	31578.95	1692.26	1780.42
6	15469.10	12630.20	27201.18	1669.69	1761.61	31	13762.65	14631.37	30035.91	1427.29	1788.35
7	16921.58	10035.94	29410.09	1632.87	1589.14	32	15461.15	14314.60	29932.65	1550.95	1155.37
8	15508.28	11934.49	30184.24	1763.77	1514.10	33	17088.93	11915.46	30523.45	1535.92	837.65
9	15172.00	12456.70	29601.61	1636.07	1730.53	34	14114.08	13963.76	31626.50	1722.53	1192.65
10	16004.37	13343.87	31624.40	1673.14	1675.67	35	14820.97	14633.53	30622.69	1726.38	1072.71
11	12566.74	13385.07	30699.64	1591.69	1661.53	36	14211.85	12353.19	31137.92	1481.22	823.35
12	15568.26	12536.55	30099.86	1469.16	1664.53	37	16440.54	14961.21	30586.63	1591.25	1084.98
13	17313.26	13251.03	32391.64	1770.50	1747.29	38	15101.34	11280.31	30992.08	1650.16	1426.59
14	17155.40	14221.31	30525.21	1516.86	1758.92	39	16148.60	9703.62	31242.42	1531.43	1543.40
15	17054.25	14186.59	32762.79	1731.93	1803.38	40	15185.88	15281.41	30935.75	1433.32	1583.74
16	17400.73	13453.16	34941.08	1698.15	1638.23	41	15392.35	14167.25	31796.05	1595.41	1649.32

Continuation of table 15

17	16869.70	14345.31	27560.77	1729.93	1570.46	42	15145.44	14084.62	31616.80	1603.35	1705.77
18	11613.52	14574.48	30191.19	1706.20	1676.23	43	15822.76	16227.85	34726.86	1741.91	1505.16
19	15966.22	14723.49	28932.70	1640.09	1400.30	44	15788.80	16788.79	32588.71	1763.19	1737.11
20	13924.08	15623.03	33062.53	1724.94	1603.53	45	14435.02	14270.41	32922.36	1537.66	1261.56
21	15001.18	15544.12	34087.02	1624.92	1344.95	46	13272.21	14791.79	31911.90	1644.10	1752.28
22	18796.15	15357.43	32570.37	1561.09	1653.37	47	13531.76	10941.44	31978.12	1773.87	1758.68
23	15646.32	15178.04	31453.35	1610.34	1558.81	48	13645.45	15656.51	33711.43	1668.38	1658.62
24	19224.75	14907.47	30969.85	1506.87	1589.40	49	14811.26	16170.77	31930.62	1602.99	1796.83
25	17448.32	14859.44	31009.31	1542.94	1466.16	50	14531.72	15697.87	31799.63	1686.35	1826.87

The experimentally measured throughput during JWT encoding (signature) for five algorithms: HS256, ES256, EdDSA (Ed25519), RS256, and PS256 is shown in Fig. 17. The x-axis represents the algorithm, and the y-axis represents the average number of encoding operations per second. Because the payload sizes and hardware configuration were fixed across runs, the figure primarily reflects differences in cryptographic overhead.

The highest throughput is achieved by HS256, providing approximately 30,000 encoding operations per second (roughly 19,000 – 35,000 ops/s). ES256 and EdDSA form an intermediate group: ES256 achieves about 15,000 ops/s, and EdDSA about 13,000 – 14,000 ops/s, i.e., 2–2.5× lower than HS256, but substantially higher than RSA-based schemes.

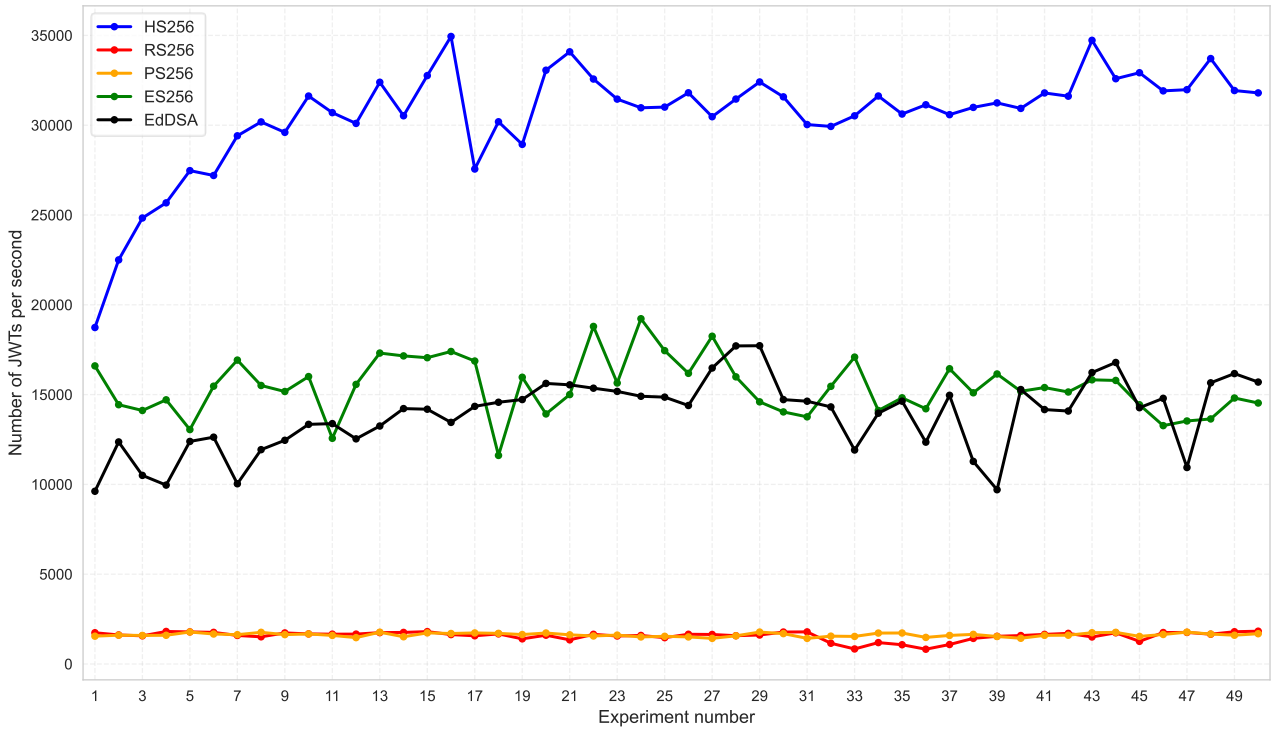

Figure 18. Comparison of the throughput of JWT signature algorithms by the number of encoding operations per second

Source of the figure: original

The lowest throughput is observed for RS256 and PS256 (RSA-2048): average values are approximately 1,500 – 1,600 ops/s, and in some runs RS256 drops to about 800 ops/s. Thus, switching from HS256 to RS256/PS256 reduces signing capacity by roughly 15–20×. Consequently, HS256 is the clear leader by the “operations per second” criterion; ES256 and EdDSA provide a practical compromise between performance and security; RS256 and PS256 are justified mainly when compatibility with existing infrastructure is prioritized over maximum throughput.

Table 16

15) Results of measuring JWT signature verification time for algorithms HS256, ES256, EdDSA, PS256, and RS256 (50 experiments, 100 tests each)

	ES256	EdDSA	HS256	PS256	RS256		ES256	EdDSA	HS256	PS256	RS256
1	0.10847	0.17793	0.03768	0.05268	0.05235	26	0.09436	0.11637	0.02863	0.05914	0.04622
2	0.1188	0.1903	0.04081	0.05715	0.04954	27	0.09913	0.1153	0.02804	0.05679	0.05407
3	0.11796	0.19853	0.03545	0.06113	0.04472	28	0.11395	0.12467	0.02879	0.04839	0.05011
4	0.12618	0.18734	0.03483	0.05552	0.04788	29	0.11939	0.13305	0.03173	0.05062	0.04176
5	0.10865	0.19698	0.03164	0.04649	0.05002	30	0.11716	0.13554	0.02961	0.05178	0.04413
6	0.10841	0.2313	0.03327	0.05897	0.04359	31	0.12224	0.14309	0.0302	0.0589	0.04768
7	0.10852	0.19308	0.03017	0.0546	0.05278	32	0.1102	0.14546	0.03168	0.05634	0.06928
8	0.11293	0.16764	0.02905	0.04879	0.04877	33	0.11917	0.16892	0.02911	0.05032	0.08817
9	0.1023	0.15777	0.02921	0.05937	0.05019	34	0.12382	0.13952	0.03004	0.05269	0.07161
10	0.10593	0.15435	0.02858	0.04828	0.05023	35	0.11371	0.13907	0.02819	0.04881	0.08834
11	0.12004	0.1629	0.03137	0.05893	0.0483	36	0.10752	0.18628	0.02942	0.06188	0.15708
12	0.10694	0.15978	0.02864	0.04954	0.05056	37	0.114	0.13898	0.03	0.05924	0.07771
13	0.09994	0.15037	0.02834	0.05413	0.04284	38	0.09955	0.1868	0.02807	0.05694	0.07574
14	0.10299	0.14928	0.02801	0.06174	0.04349	39	0.12272	0.1605	0.02902	0.06158	0.04898
15	0.09837	0.14728	0.0272	0.05039	0.04787	40	0.10792	0.12931	0.03073	0.0575	0.05024
16	0.10485	0.14272	0.03609	0.05148	0.05356	41	0.10708	0.14285	0.02956	0.05536	0.04906
17	0.10244	0.13722	0.02996	0.05308	0.05358	42	0.10977	0.14437	0.02589	0.05264	0.07708
18	0.11681	0.13845	0.03054	0.05031	0.04664	43	0.10989	0.11676	0.02573	0.0505	0.04673
19	0.11351	0.13164	0.02927	0.0478	0.04712	44	0.11175	0.12372	0.02771	0.06244	0.04261
20	0.11724	0.13482	0.02642	0.04791	0.0784	45	0.12686	0.14038	0.02726	0.06816	0.04598
21	0.10805	0.13583	0.02681	0.0718	0.04977	46	0.1212	0.13772	0.02802	0.05135	0.04544
22	0.10008	0.13784	0.02795	0.05143	0.04784	47	0.13334	0.15176	0.02662	0.05175	0.04883
23	0.11338	0.1358	0.02858	0.05717	0.04774	48	0.11722	0.12917	0.0276	0.05038	0.0456
24	0.09395	0.13997	0.02976	0.05361	0.05187	49	0.11303	0.13333	0.02714	0.04981	0.04439
25	0.10011	0.1371	0.02835	0.05886	0.0615	50	0.11681	0.12731	0.02751	0.05177	0.04385

The data show that HS256 consistently has the shortest verification time: the average is approximately 0.030 ms, with variation roughly within 0.026–0.041 ms.

RSA-based algorithms (PS256 and RS256) form an intermediate group, with an average verification time of about 0.055 ms, i.e., roughly 1.8–2× slower than HS256. The largest verification costs are observed for ES256 and EdDSA, with average values of approximately 0.11 ms and 0.15 ms, respectively, corresponding to a 3.5–5× slowdown compared to HS256. Thus, HS256 is the most efficient for verification, RSA-based schemes occupy the middle position, and ES256/EdDSA are the slowest modes in verify operations.

The graph (Fig. 3.6) in the form of a bar chart shows the average values of the JWT signature verification time, calculated based on the results of 50 runs for each algorithm. The X-axis shows the signature algorithms, and the Y-axis shows the average verification time in milliseconds.

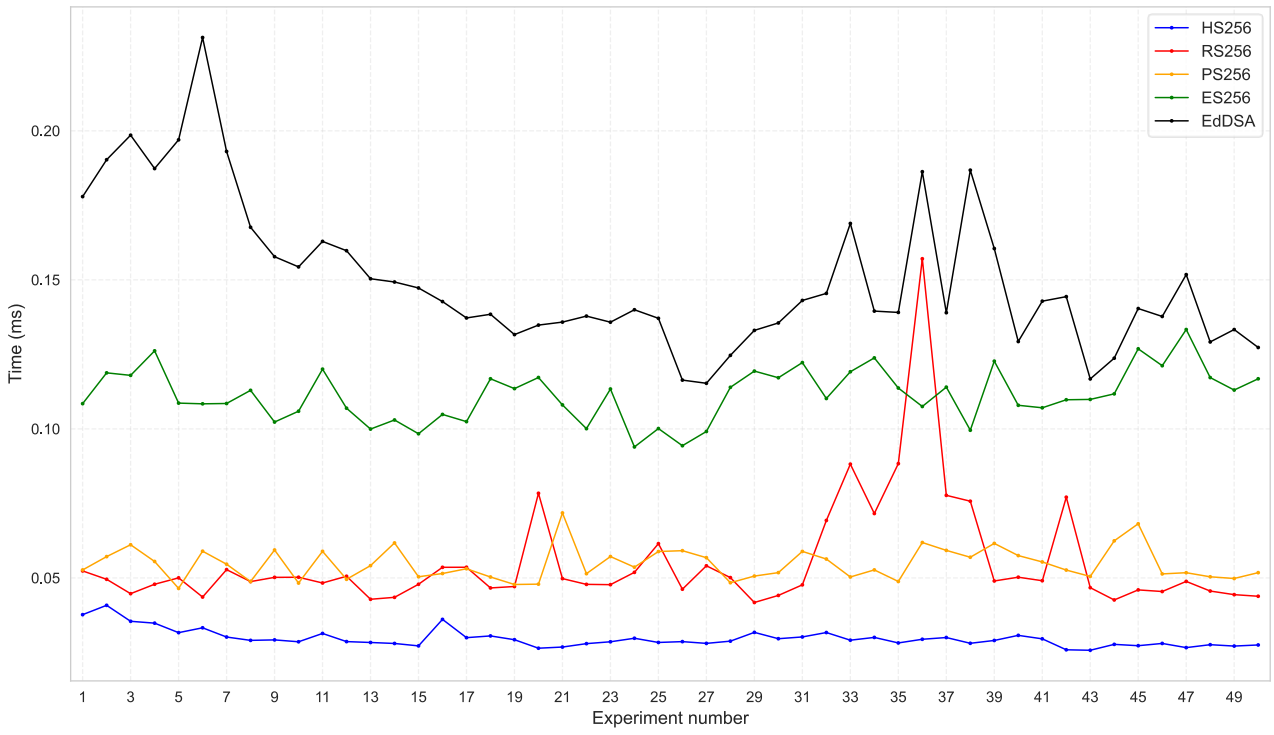

Figure 19. Comparison of average JWT signature verification time for different algorithms (HS256, ES256, EdDSA, PS256, RS256)

Source of the figure: original

A visual analysis of the data reveals that the HS256 column exhibits minimal delays in verify operations, suggesting that it is the lowest among the columns under consideration. The PS256 and RS256 columns are taller, but remain significantly lower than ES256 and especially EdDSA, which form the "slowest" group with the maximum signature verification time. The graph provides clear evidence that HS256 is the optimal choice in terms of signature verification efficiency. RSA algorithms offer an acceptable compromise, while ES256 and EdDSA are suitable when the priority is not speed but other requirements, such as cryptographic strength or standardization.

The table displays the throughput values during JWT signature verification (verify) for five cryptographic algorithms: ES256, EdDSA (Ed25519), HS256, PS256, and RS256. Each row in the table corresponds to a distinct experiment, and the numerical values represent the number of verification operations that can be performed per second (ops/s).

Table 17

16) Results of measuring JWT signature verification throughput for algorithms HS256, ES256, EdDSA, PS256, and RS256 (ops/s)

	ES256	EdDSA	HS256	PS256	RS256		ES256	EdDSA	HS256	PS256	RS256
1	2	3	4	5	6	7	8	9	10	11	12
1	9219.22	5620.3	26540.4	18981.34	19103.38	26	10597.24	8593.64	34929.71	16909	21636.67
2	8417.74	5254.75	24505.91	17498.91	20184.91	27	10087.4	8672.87	35661.29	17607.39	18493.35
3	8477.59	5037.01	28208.43	16358.73	22358.92	28	8775.73	8021.07	34738.94	20663.79	19954.85
4	7925.47	5337.77	28706.97	18012.78	20886.01	29	8376.22	7515.93	31519.26	19754.25	23946.53
5	9203.74	5076.57	31610.29	21511.18	19992.63	30	8535.06	7377.87	33774.21	19312.68	22660.2
6	9224.08	4323.31	30060.76	16958.41	22939.44	31	8180.57	6988.48	33110.04	16979.06	20973.43
7	9215.3	5179.19	33150.1	18315.9	18946.81	32	9074.35	6874.87	31570.08	17747.81	14434.13
8	8855.14	5965.33	34428.6	20495.08	20504.35	33	8391.55	5919.82	34346.9	19872.66	11342.31
9	9775.31	6338.51	34238.04	16843.51	19924.94	34	8076.2	7167.23	33292.54	18978.47	13965.51
10	9440.43	6478.58	34993.1	20712.84	19907.75	35	8794.47	7190.5	35471.95	20487.97	11320.07
11	8330.35	6138.77	31878.98	16968.58	20704.59	36	9300.3	5368.24	33990.1	16159.28	6366.34
12	9350.81	6258.59	34911.3	20187.51	19779.76	37	8772.01	7195.32	33329.36	16880.48	12868.06
13	10005.65	6650.41	35284.96	18474.37	23342.76	38	10044.87	5353.34	35622.52	17563.88	13202.79

Continuation of table 17

14	9710.08	6698.76	35700.08	16197.86	22994.01	39	8148.63	6230.43	34464.79	16239.86	20414.45
15	10166.14	6789.89	36766.19	19845.19	20890.27	40	9265.87	7733.38	32545.47	17390.96	19904.37
16	9537.87	7006.84	27709.64	19423.65	18671.51	41	9338.74	7000.33	33834.52	18062.89	20384.35
17	9761.68	7287.76	33376.23	18838.19	18663.54	42	9109.59	6926.59	38621.01	18995.7	12974.25
18	8560.66	7222.69	32743.18	19876.12	21441.63	43	9099.94	8564.68	38865.15	19802.69	21400.58
19	8809.73	7596.49	34165.35	20920.43	21220.54	44	8948.6	8082.69	36089.93	16016.47	23470.38
20	8529.56	7417.17	37853.95	20871.14	12754.59	45	7882.43	7123.55	36688.27	14671.79	21748.22
21	9254.64	7362.12	37301.02	13927.54	20092.89	46	8250.72	7261.22	35690.04	19472.53	22009.43
22	9992.09	7254.67	35783.27	19443.84	20902.18	47	7499.45	6589.21	37568.17	19324.12	20477.44
23	8819.87	7363.59	34983.76	17491.43	20948.74	48	8530.61	7741.79	36226.9	19848.83	21932.05
24	10643.79	7144.42	33605.67	18653.28	19280.64	49	8847.26	7499.95	36849.42	20075.96	22526.28
25	9988.66	7293.72	35272.02	16989.97	16260.37	50	8560.83	7855.1	36345.67	19315.02	22802.9

Across 50 runs under identical conditions, the highest verification throughput is consistently provided by HS256, with an average of approximately 35,000 ops/s. RS256 and PS256 form an intermediate group at approximately 18,000 – 20,000 ops/s, while RS256 shows higher variability (some runs yield lower values around 6,000 – 11,000 ops/s). The lowest verification throughput is observed for ES256 and EdDSA: ES256 typically achieves about 9,000 – 10,000 ops/s, whereas EdDSA achieves about 7,000 – 8,000 ops/s. Therefore, a clear hierarchy is observed: HS256 is the most efficient by verifications per second, RSA algorithms are intermediate, and ES256/EdDSA are the slowest in terms of verify throughput.

The highest graph (blue) describes HS256, reflecting its undisputed leadership in performance. RS256 (red) and PS256 (orange) demonstrate that RSA-based algorithms provide approximately half the throughput of HS256, yet still significantly outperform elliptic schemes. ES256 (green) exhibits a conspicuously lower bar height, and the EdDSA bar (black) demonstrates the lowest number of signature checks per second for this mode, thereby confirming the observations. The visual comparison offers a clear illustration of the trade-off between security and performance. In scenarios with high service loads, symmetric HS256 is optimal. RSA algorithms are

suitable as a balance between performance and compatibility requirements. ES256 and EdDSA are best used in scenarios where cryptographic strength and compliance with modern standards are prioritized over maximum throughput.

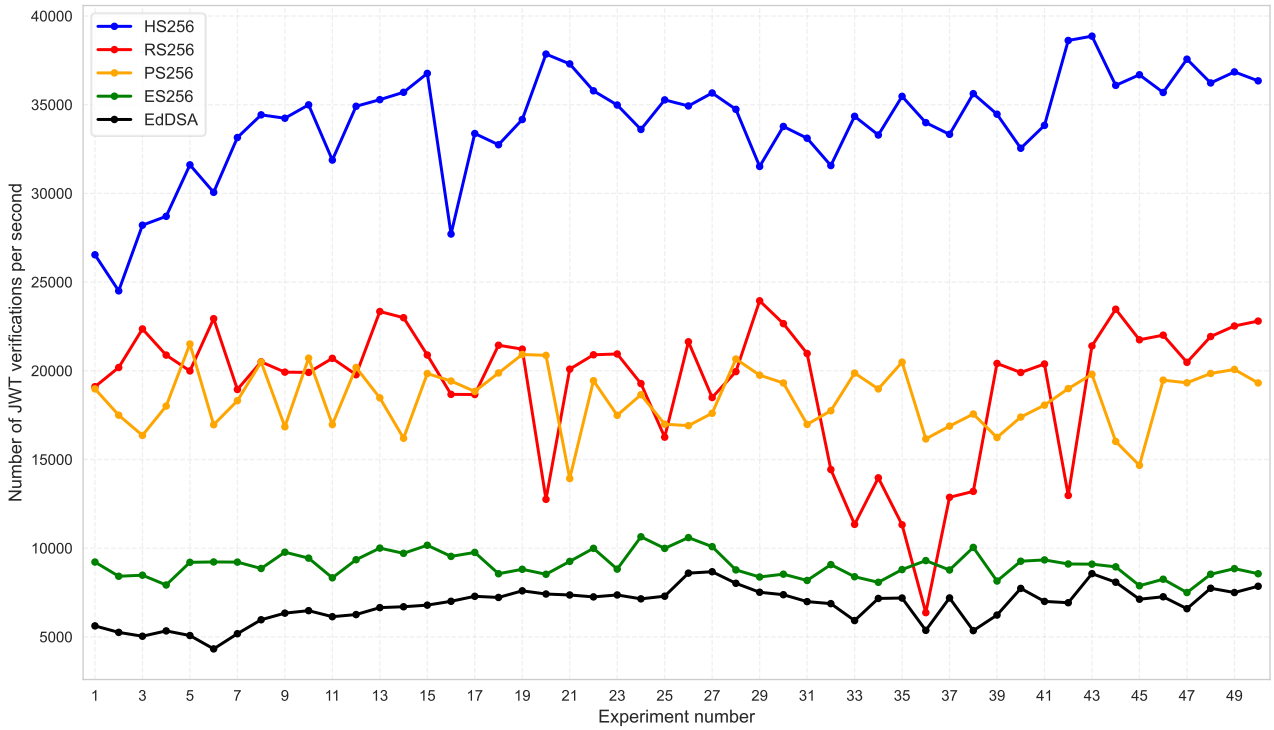

Figure 20. Comparison of JWT verification throughput by the number of verify operations per second

Source of the figure: original

Finally, Table 18 summarizes the averaged encode/verify results for HS256, ES256, EdDSA, PS256, and RS256 (averaged over 50 runs).

Table 18

17) Performance comparison of JWT encode/verify operations for different signing algorithms

Algorithm	Encode: average time, ms	Encode: throughput, thousand ops/s	Verify: average time, ms	Verify: throughput capacity, thousand ops/s
HS256	0,033	30,24	0,030	33,98
ES256	0,065	15,27	0,111	9,03
EdDSA	0,074	13,57	0,150	6,82
PS256	0,618	1,62	0,055	18,44
RS256	0,659	1,52	0,055	19,28

4.6. Assessment of the adequacy of experimental research into the method, model, algorithm.

The study examined a single security feature of web-oriented applications, namely, JWT tokens. However, it is noteworthy that a considerable number of critical mechanisms were not included in the scope of the experiment. The test client can be extended to evaluate multiple security features concurrently; however, the framework employed is intentionally simplified and straightforward to implement. The implemented model is deficient in key rotation and revocation mechanisms, which curtails its applicability in industrial scenarios. Furthermore, the study's emphasis on authentication overshadows the unaddressed matter of authorization. In the context of token-based authentication, users may be granted disparate access rights to a curated subset of resources or discrete microservices, a scenario that necessitates supplementary modeling and analysis.

The experiments were conducted using the Python programming language; however, the same approach to load testing can be implemented for other languages and technology stacks. This would allow for a comparison of the impact of platform features on performance. A notable constraint pertains to the absence of database replication, which entails the storage of data in a single copy. This configuration results in the establishment of a "single point of failure," thereby augmenting the system's

susceptibility to failures and security breaches. In order to enhance reliability and fault tolerance, it is imperative that such infrastructure be deployed in a cloud environment, employing replication, load balancing, and geographic resource distribution mechanisms.

Conclusion

During the experiments, it was observed that as the number of requests increases, network interactions between microservices introduce millisecond-scale delays; consequently, the contribution of the core security functions to the overall request processing time becomes relatively insignificant. To obtain a more realistic picture, it is advisable to perform extended load testing in which the CPU operates under sustained high utilization and system behavior is analyzed under computationally intensive workloads. To reduce response time, caching mechanisms can be implemented at the API Gateway level or within the User Authentication Service, while ensuring that expired tokens are removed via an appropriate cache eviction policy once their validity period has elapsed.

It is possible to replace JWT validation in every microservice with validation performed only in a limited set of specialized services, thereby reducing overall load. If the request rate increases substantially, deploying multiple internal API gateways should be considered. Adding a unique request identifier to the JWT would enable finer-grained correlation of user activity logs and facilitate analysis of which specific functions were executed. Another optimization direction is selecting JWT signature algorithms with better performance characteristics; in particular, Edwards-curve schemes such as Ed25519 and Ed448 demonstrated high performance in the conducted measurements. To limit request volume and mitigate DoS attacks originating from a single user, device, IP address, or set of credentials, rate limiting and load balancing mechanisms can be applied at the API Gateway level.

Summary Table 19 provides recommendations for selecting JWT signature algorithms based on the experimental analysis of encode and verify modes and typical requirements of information systems. For each algorithm (HS256, ES256, EdDSA,

PS256, RS256), the table summarizes key performance and security characteristics, common usage scenarios, and a generalized recommendation.

Table 19

18) Recommendations for Using Algorithms in JWT

Algorithm	Key Features (Performance/Security)	Recommended Use Cases	General Recommendation
HS256	Fastest in both encode and verify; very high throughput; requires secure storage of a shared secret key across all services.	High-load API gateways, microservices with a large volume of authentication requests, internal systems where a shared-secret model is acceptable.	Use as the primary option when the threat model allows symmetric keys and maximum performance is required.
ES256	Moderate performance (slower than HS256, but faster than EdDSA in verify); asymmetric keys; compact EC key material.	External services, B2B integrations where an asymmetric signature is required; modern mobile and web applications emphasizing standardized EC algorithms.	Recommended as a balance between performance, modern standards, and asymmetric cryptography.
EdDSA (Ed25519)	High cryptographic strength and modern design; encode/verify slower than HS256 and ES256; lowest throughput among the evaluated verify modes.	Systems with increased security requirements, long-lived tokens, services where modern cryptographic primitives (Ed25519) are prioritized and load is moderate.	Use where security and modern primitives are more important than maximum performance.
PS256	Very slow encode (tens of times slower than HS256), but verify is noticeably faster and comparable to RS256; asymmetric RSA signature (RSASSA-PSS).	Systems where signing is infrequent (e.g., performed by a dedicated service) but verification is frequent; integrations that require RSASSA-PSS due to security policies.	Apply when policies/standards mandate PSS; not recommended for mass token signing under high load.
RS256	Very slow encode; fast verify (second most efficient after HS256); widely supported across libraries and infrastructure.	Legacy systems, enterprise platforms, and identity federation (OIDC, SSO) where RSA is historically used and maximum compatibility is required.	Recommended for compatibility with existing infrastructure; for new high-load systems, consider HS256 or ES256 instead.

The table indicates that HS256 should be treated as the baseline choice for high-load services due to minimal encoding and verification time and maximum throughput, provided that a symmetric key model is acceptable. ES256 is positioned as a compromise between performance and modern asymmetric-cryptography requirements, making it suitable for external integrations and modern web and mobile applications. EdDSA (Ed25519) is recommended for systems with higher cryptographic-strength requirements where reduced throughput is acceptable in exchange for the use of modern primitives.

RSA-based algorithms are considered separately. PS256 and RS256 have significantly slower encode performance, but provide acceptable verification speed and remain relevant due to widespread deployment and security-policy requirements in existing infrastructure. For these algorithms, the table recommends use primarily in scenarios where compatibility with established solutions (SSO, OIDC, enterprise platforms) is more critical than maximum throughput. Overall, the table serves as a practical checklist for engineers and architects, enabling a rapid mapping of algorithm properties to project requirements and supporting an informed choice of JWT signing mode.