DESIGN AND IMPLEMENTATION OF A RO E-INVOICE PLATFORM INTEGRATED WITH SPV ANAF

Orlando Sebastian BUHAIU, Raul MALUTAN
Communications Department, Technical University of Cluj-Napoca, Cluj-Napoca, Romania
Orlando.Buhaiu@campus.utcluj.ro, Raul.Malutan@com.utcluj.ro

Abstract: This paper proposes and evaluates an architecture and prototype implementation of a Romanian e-Invoice platform integrated with the ANAF SPV. It describes the B2B obligations in 2024, the B2C obligations in 2025, the legal five-day submission deadline, and the role of the receipt. The architecture uses Blazor and ASP.NET Core and Azure Blob Storage, with idempotency, rate limiting, and retry with backoff. The paper details the OAuth 2.0 integration from authorization to token exchange and token rotation, the data model from CSV or JSON to UBL XML, and the validation pipeline. It analyzes work queues and scheduling, including formulas for throughput and latency. The evaluation reports P50 and P95 latency, success rate, UBL error rate, throughput, and cost per one thousand invoices. It also discusses limitations and future work.

Keywords: *Electronic invoice, RO e-Invoice, UBL 2.1, OAuth2, SPV ANAF.*

I. INTRODUCTION

In the context of fiscal digitization, Romania operates the national electronic invoicing system RO e-Factura, aligned with the European standard EN 16931 (semantic model of essential invoice elements) and the **UBL 2.1** syntax (ISO/IEC 19845), with the national specifications **RO_CIUS**. In practice, any invoice transmitted through the system is an XML file that complies with both EN 16931 and RO_CIUS rules [1], [8], [11], [10]. MF Order No. 1366/2021 establishes the compliance identifiers (CustomizationID) for UBL 2.1/RO_CIUS, ensuring interoperability and automatic processing at national and European level [10]. EN 16931 defines the semantic model and usage rules, and UBL 2.1 (published as ISO/IEC 19845:2015) provides the XML messages for invoice and credit memo used in compliant implementations [1], [8], [11].

The legal regime was introduced in stages. In B2G (Business to Government) relations, the use of RO e-Factura is mandatory starting with 1 July 2022, pursuant to **GEO 120/2021** approved by Law 139/2022 and official ANAF communications [3]. In B2B (Business to Business) relations, reporting via RO e-Factura has become mandatory since 1 January 2024, according to announcements by the Ministry of Finance/ANAF [9]. The extension to B2C (Business to Consumer) was regulated by GEO 69/2024, with mandatory implementation from 1 January 2025 and a specific transitional regime; the Ministry of Finance has published clarifications dedicated to this extension [3], [4]. Failure to comply with these obligations will result in administrative penalties, communicated publicly by ANAF [3].

A key constraint is the transmission deadline: issued invoices must be uploaded to the system within a maximum of 5 days from the date of issue. From **1 July 2024**, the authorities have expressly clarified that the deadline is 5 calendar days, calculated from the day following the date of issue, regardless of whether the 5th day is a working day or a non-working day. This relatively short deadline requires robust error tolerance mechanisms (retry with backoff), sending rate control, and continuous monitoring of processing status [3].

The operational flow in SPV includes the generation of a unique identifier at the time of upload, called the **"upload index**". This is subsequently used to verify the status of the file, indicating whether validation was successful, whether there are errors, or whether processing is in progress. According to the official documentation, valid responses and files can be downloaded for 60 days, after which they are archived and can be reissued upon request. In practice, this identifier acts as a receipt in the ANAF system and allows for subsequent status tracking [3].

Third-party applications accessing RO e-Factura services exposed through SPV use **OAuth 2.0** with digital certificate-based authentication. The ANAF documentation describes the **Authorization Code** authorization flow, identifies the access points for production at logincert.anaf.ro and specifies the parameters required to obtain tokens. The validity is 90 days for the JWT access token and 365 days for the refresh token, and rotation is performed through the standard exchange defined by OAuth [5], [6].

In practice, there are already several ways to work with RO e-Factura. Some ERP systems come with dedicated modules that directly issue XML UBL 2.1 in accordance with EN 16931 and RO_CIUS rules and send documents via connectors to SPV. Other solutions are SaaS services that receive data in CSV or JSON, convert it to UBL, and then track the status on behalf of the taxpayer. In the technical area, tool chains based on UBL

Manuscript received October 31, 2025; revised December 12, 2025

libraries and Schematron rules are also used, which are useful for validation and conversions in customized flows. Existing approaches documented in the literature on electronic invoicing and in national guidelines focus mainly on standardization and integration models. European and national documents describe reference architectures in which ERP systems or service providers act as intermediaries between taxpayers and the central platform, relying on EN 16931, UBL 2.1, and national CIUS rules to ensure interoperability. Commercial solutions typically expose modules for XML document generation and connectivity to the SPV but treat the internal delivery pipeline as a black box. Issues such as idempotent uploads, explicit retry policies, or quantitative assessment of processing capacity and latency are rarely discussed in detail. This paper complements these approaches by detailing a concrete architecture that includes an internal work queue layer, strict idempotency, and request rate limiting, accompanied by an experimental evaluation based on P50/P95 latency, success rate, and cost per thousand invoices.

The contribution of this paper is both practical and methodological. On the technical side, an architectural model is proposed for integration with the national RO e-Factura and SPV systems, structured on three layers: a web interface for users, an application-services layer, and an integration layer with the ANAF infrastructure. Within this model, the idempotence of operations, request-rate limiting, and retry mechanisms are treated explicitly as first-class design requirements. On the methodological side, a set of measurable indicators is defined for this type of platform.

These indicators include end-to-end latency expressed as median and 95th percentile, success rate without human intervention, UBL validation error rate, and cost per thousand invoices processed. A structured test procedure is applied in an environment that approximates production conditions. The experimental evaluation shows how these mechanisms influence system performance and reliability and can serve as a benchmark for the design of other electronic tax-reporting solutions, including a quantitative comparison with sequential baseline implementation.

This paper presents and evaluates the proposed architecture and technical solutions for a platform that issues, validates and transmits UBL 2.1/RO_CIUS electronic invoices in RO e-Factura, with SPV integration via OAuth 2.0, idempotency mechanisms, rate limiting and retry with backoff, as well as real-time feedback to the user [1], [5], [8], [10]. Section II describes the system model and its integration with SPV, including diagrams and code excerpts. Section III presents the methodology and experimental results. Section IV summarizes the conclusions and outlines future directions. Section V lists the references.

II. IMPLEMENTATION

A. System model and architecture

The platform is built on a layered architecture, common in business web applications.

From an architectural perspective, the proposed solution is organized on three main levels, illustrated in Figure 1. The first level consists of the interface layer, implemented in Blazor WebAssembly and connected to a SignalR hub for real-time notifications. The second level comprises the application services layer, implemented in ASP.NET Core Web API, which exposes invoice-specific operations, manages the processing queue, and performs UBL/RO CIUS validations.

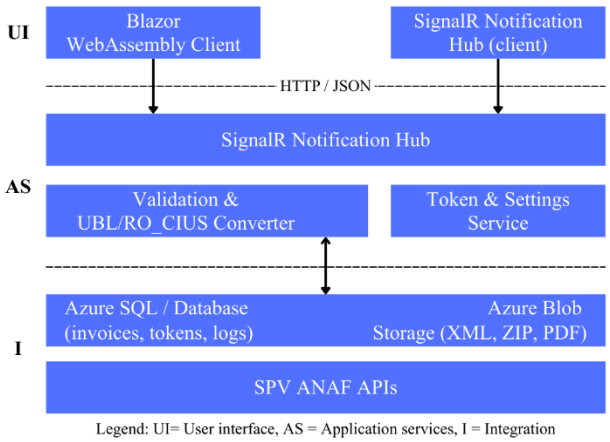

Figure 1. Layered architecture of the e-Invoice web platform

The third level includes the integration and persistence layer, which ensures integration with SPV ANAF through OAuth 2.0, file storage in Azure Blob Storage, and the persistence of metadata, tokens, and audit logs in the database. Communication is unidirectional from the interface to the API and then to external services, allowing independent scaling of components and clear isolation of performance aspects.

The interface is developed in Blazor, so that the presentation logic remains in C# and the components can be easily reused. **Blazor WebAssembly** was chosen for browser execution. SignalR, a WebSocket channel through which the server immediately sends status changes, is used for real time notifications to the user. The server side runs on **ASP.NET Core** as a Web API and exposes REST operations for uploading and viewing invoices. Integration with ANAF systems is also handled here, including OAuth2 authentication and calls to RO e-Factura services. XML files and receipts persisted in Azure Blob Storage, which offers inexpensive storage and immediate availability.

The three-layer structure described above is not tied to the technologies used in the prototype (Blazor, ASP.NET Core, or Azure). It can be viewed as an architectural pattern for applications that must submit fiscal documents to external services under strict time constraints. The web interface layer manages user interaction and real-time notifications, the application-services layer implements business rules and orchestrates asynchronous work queues, and the integration layer isolates SPV and RO e-Factura specifics, including authentication mechanisms, XML formats, and rate limits imposed by the external APIs.

Figure 2. describes the path of an invoiced on the platform. Initially, it is "**Pending**", meaning it only exists in the internal system. Upon submission, the application generates XML UBL, runs local validations, and publishes the document via SPV. The ANAF server responds with a receipt, and the invoice enters "**Processing**". The result is obtained through periodic

queries: "**Accepted**" when all checks pass, or "**Error**" when problems arise. If errors occur, the invoice is corrected and resubmitted, and the cycle begins again.

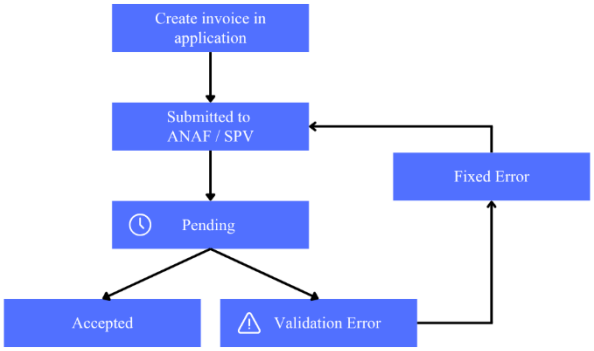

Figure 2. Invoice flow

In terms of components, the Blazor interface and SignalR hub keep the user up to date with the status of each invoice. The ASP.NET Core backend receives requests, runs business rules, manages the queue for submissions, and communicates with ANAF services. A dedicated SPV integration service handles tokens, uploads, and status checks.

Azure storage keeps files and receipts. There is also a logging module with audit, errors, and response times, useful for support and compliance. An internal work queue is used for volume and reliability. Invoices enter the queue and are processed asynchronously by a worker. Scheduling is round robin or, if necessary, prioritized. For example, documents close to the legal deadline can be moved to the front of the queue. Basic queue operations, such as adding and extracting, have *O(1)* complexity. Processing an invoice involves XML generation and local validations, with *O(n)* cost depending on the number of lines, but in practice this time is much less than the network latency and processing time in ANAF.

Idempotence is handled explicitly to avoid duplicates. Each upload has a unique identifier derived from the invoice data. When writing to **Azure Blob**, the conditional **If-None-Match header with the value "*"** is used. The result is simple: if the file does not exist, it is created. If it already exists, Azure responds with 412 Precondition Failed and the operation stops. For updates, the If-Match condition with the current version's ETag prevents accidental overwriting [7]. In code, these responses translate into clear branches: duplicate detected, or optimistic concurrency failed. This approach guarantees that the request is sent only once, even in the case of retries or repeated clicks.

Rate limiting is also applied to protect both the platform and ANAF services. The backend keeps a counter and allows only a certain number of requests per second, configurable depending on the runtime environment. Status queries can be grouped into batches so that multiple receipts can be checked in a single call. This aggregation reduces the total number of requests, decreases average latency, and increases the overall throughput of the system.

B. SPV protocol and OAuth2 integration

Integration with the ANAF system is achieved through the **SPV (Virtual Private Space)** portal, which provides secure web services that allow invoices to be uploaded and receipts to be obtained. Access to these services is protected by a standard **OAuth 2.0** mechanism: third-party developers must register an application on the ANAF portal, obtaining a *client_id* and a secret. Then the end user (usually the legal representative of the company or an authorized representative with a digital certificate) must authorize the application to access their SPV account through an **OAuth2 Authorization Code** flow [5], [6]. Specifically, the application redirects the user to the ANAF authentication page, where they log in with their digital certificate and grant access to the client application. At the end of this process, the application receives an authorization code that can exchange for an **access token** (JWT) and a **refresh token** using the **ANAF OAuth2** endpoint.

To implement this flow, the implementation uses a dedicated controller, *AnafController*, which manages the OAuth2 steps (see Figure 3.). The code snippet below illustrates the essential parts: building the authorization URL and handling the callback from ANAF to obtain the tokens.

Goal: connect the application to ANAF and obtain tokens**.**
When the user clicks "Connect to ANAF":
1. Build the sign-in link using:
client id (from config)
redirect url (from config)
scope "efi:factura"
2.Send the browser to that link.

When the application receives the callback with a "code":
3.POST to ANAF /oauth2/token with:
grant type = authorization_code
the code we just received
client id and client secret (from config)
the same redirect url
4.If tokens are returned:
save access token, refresh token, and expiry
send the user to the Dashboard
Else:
show an error and let the user try again

Figure 3. Pseudocode "AnafController" OAuth2 Tokens

In implementation, after obtaining the access token, it is used in all subsequent calls to the e-Factura APIs, adding it to the HTTP Authorization. The refresh token (a secret with a longer lifetime, in the order of tens of days) is stored in the application database, associated with the user account. This refresh token allows us to obtain a new access token without user intervention when the old one expires, through an automatic flow (grant type = refresh token).

Token management is a sensitive area. The application keeps the credentials received from the SPV safe and never exposes them to the user interface. Refresh tokens are stored encrypted in the database and remain accessible only to backend components. The access token has a short

duration, approximately 30 minutes, and the refresh token is valid for a much longer period of 90 days. As long as the user's digital certificate remains valid, authorization can be renewed automatically. When the access token expires, the backend uses the refresh token and obtains a new one, without any action on the part of the user. If the provider issues a new refresh token along with the access token, the application immediately saves it and invalidates the old one.

This maintains rotation and limits the impact of a potential security incident. From a security perspective, the main vulnerabilities identified are the interception of the authorization code on the redirect channel, the theft of the refresh token from local storage, and the unauthorized use of client credentials stored on the server. In the developed prototype, all OAuth2 exchanges are performed via TLS, redirect URIs are restricted to a pre-approved list, and no tokens are stored in the browser. Refresh tokens and client secrets are stored exclusively on the server, encrypted in the database and accessible only through restrictive application roles. In addition, the identity of the application used for calls to the SPV is separate from end-user accounts, and all token-related operations are recorded in audit logs. These measures reduce the impact of a potential database compromise or credential leak, keeping the attack surface in line with the recommendations in the OAuth 2.0 and PKCE specifications. As future directions for strengthening security, secrets can be migrated to a dedicated key management service, automatic periodic rotation can be implemented, and in infrastructures that allow it, mutual TLS based on certificates can be enabled.

C. Data model and invoice validation

The platform accepts invoices entered manually into the interface or imported from CSV and JSON files exported from other systems. Regardless of the source, the data arrives in a unified invoice model in C#, with objects for header, supplier, customer, lines, taxes, and totals. The UBL converter starts with this model and produces XML according to **UBL 2.1** and **RO_CIUS** rules. Serialization is used for fields that match directly, and where special codes or conditional attributes are required, the XML nodes are explicitly filled in. Mappings for VAT, units of measure, and other official nomenclatures are applied before generation, so that the document is already consistent when it goes for validation.

The verification begins with the **XSD (XML Schema Definition**) schema. Each XML is validated against UBL 2.1 extended with CIUS Romania, so that the correct structure and presence of mandatory elements are confirmed. If schema errors occur, the invoice cannot be finalized. The system marks the document as invalid and immediately displays the explanation in the interface, with reference to the element that caused the problem.

After passing the **XSD**, the business rules follow. **RO_CIUS** introduces requirements that cannot be verified by the schema alone. For invoices to public institutions, the existence of **CPV** codes is verified. For certain types of transactions, additional fields become mandatory, such as the buyer's address or the reverse charge mention. The arithmetic consistency of the amounts is also checked so that the totals and rounding are within the accepted limits. The value of the supplier's and customer's CUI is checked at the format and check digit level. Many minor non-conformities can be corrected automatically. Diacritics or invalid characters are replaced with valid equivalents in XML. Country codes and units of measurement are normalized to the standards used by CIUS. For major discrepancies, such as incorrect totals, the application stops the flow and asks the user to correct the information [1], [8], [10], [11].

Official validation takes place on ANAF servers after the document has been sent. The results can be viewed based on the receipt. If ANAF rejects the invoice, the returned message is taken as is and clearly displayed in the application. The error may be serious, in which case the document is not registered, or it may be just a warning. When the status remains in processing for a long time or temporary errors occur during the query, the system continues to check at regular intervals without involving the user.

D. Forwarding, throttling, and retry/backoff mechanisms

Fault tolerance is not only about infrastructure, but also about how the code is written. The goal is simple: an invoice should be sent only once, so that external services are not overloaded when volumes increase, and there should be retries when the network or external service has problems. An invoice receives a unique identifier and, once it has received a loading index (receipt from ANAF), it is marked as transmitted. At the HTTP level, the transmission was exposed as an idempotent operation, so that the same repeated request no longer produces any effects. ETag is used for storage, so that two parallel processes attempting to save the same XML file do not create duplicates [7]. If a call accidentally reaches the backend twice, the response indicates the existing status and does not start a new upload.

Limiting invoice transmission is necessary to avoid problems with ANAF services, but also to protect the application when many companies send simultaneously. In practice, a configurable limit of requests per second and a maximum number of concurrent calls have been set. In the code, the request “waits in line” on an asynchronous traffic light for a few milliseconds when the system is below the limit, and during traffic peaks it may wait a little longer.

Automatic retries only occur when external problems arise. The strategy uses exponential backoff with jitter to avoid multiple retries lining up at the same time. The first pause is short, with subsequent pauses gradually increasing to a reasonable limit. If the access token expires, the refresh token is used transparently, and the initial operation is automatically resumed after refreshment. When the status remains “processing” for a longer period, the query is rescheduled with a dynamic interval. When a result appears, the loop stops.

E. Real time notifications and front-end integration

The purpose of notification modules is to allow users to immediately see what is happening with their invoices, without having to manually refresh or check the SPV. Every time a new receipt is available or when ANAF changes the status of an invoice, the backend directly notifies the interface. In practice, communication is done

via SignalR. The server application exposes a WebSocket hub, and the Blazor client connects to this hub and listens to relevant events.

On the server, the services that process the upload and status check send a signal to the hub immediately after saving the upload index, receipt, or new status from ANAF to the database. The message is directed to the appropriate user based on their authentication identifier. When the event leaves the backend, the already connected client receives a short command to refresh the tax events "inbox", so that the bell badge and message list are updated in sync with the change in the system.

In the interface, the notification logic lives right in the layout (*FrontLayout*). Upon initialization, the component builds the SignalR connection to the hub exposed at "/UserNotifications" and subscribes to the "ReceiveRefreshInvoicePage" event. When the server emits this event, the layout calls the method that reloads the ANAF message list (*LoadInvoiceAnafMessages*), then performs a StateHasChanged, so that the UI redraws without losing the context of the page the user is on. In the same step, the badge on the bell icon shows the number of unread messages, and the notification panel can be opened with a click to see details.

Updating the list does not download the data but calls the application services to bring only what is needed. First, the ANAF messages filtered for the selected user and company are requested (*GetInvoiceAnafMessages*). Then, for each message, the links to the associated files are completed: PDF for preview and ZIP for the official package downloaded from SPV. If a message does not yet have locally saved files, the user can modify the download and attachment via a button. The action calls the backend method that downloads and persists the content, after which the panel refreshes and provides the download links.

The flow from server to client remains simple and readable. When the server completes a relevant operation (e.g., received "ok" on a receipt or recorded a "not ok" error), it publishes a signal to that user, and the front-end reacts by reloading only the notifications section. From the user's point of view, the effect is immediate: the "Accepted" status or the error message with explanations received from ANAF appears on the screen, and if available, the buttons for **PDF** and **ZIP** package appear.

Below (Figure 4.) is a code snippet that shows the mechanism used in the application.

Purpose: keep a live notifications connection for the current user and refresh the invoice list when a message arrives.

Procedure RefreshUserNotificationHubConnection

1) If there is an existing hub connection: try to stop it (ignore errors).

2) If notifications are enabled in settings AND a user is logged in:

a) Create a new SignalR connection to "/UserNotifications".
b) Subscribe to the event "ReceiveRefreshInvoicePage(applicationUserID)":
If applicationUserID equals the current user's Id:
run on the UI thread:
- reload invoice messages
- refresh the screen (StateHasChanged)
c) Start the hub connection.
d) Put the current user's Id into the search model.
e) Load invoice messages once immediately.

3) Else (no messages or no user):
optionally clear any local notification state.

End Procedure

Figure 4. User Notification Hub Connection

The *RefreshUserNotificationHubConnection* function in the Blazor interface reinitializes the SignalR connection to the /UserNotifications route and subscribes to the **ReceiveRefreshInvoicePage** event. If a connection already exists, it is stopped, and a new one is started only when messages are enabled and there is an authenticated user. Upon receiving an event addressed to the current user, the component safely runs on the UI thread (InvokeAsync) the loading of ANAF messages (**LoadInvoiceAnafMessages**) and immediately updates the screen (StateHasChanged).

F. User interface and main functionality

Figure 5. shows the main screen used to create an invoice in the proposed solution.

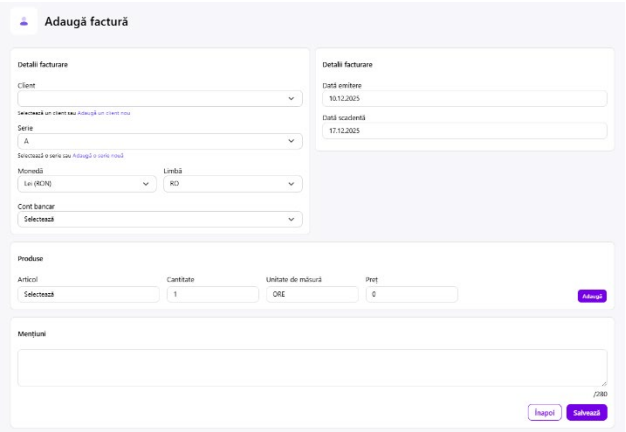

Figure 5. Invoice creation screen (add invoice form)

The screen is organized into logical panels. The upper-left panel groups the general billing details: selection of

the customer from the master data, choice of series, currency and language, and the bank account that will be used for collection. The upper-right panel contains the document dates, specifically the issue date and due date, aligned with the accounting rules enforced in the back-end services. The “Products" section allows the composition of invoice lines by selecting items from the catalogue and specifying quantity, unit of measure and unit price. At the bottom, the “Notes" area stores free-text remarks that are later propagated into the UBL document. When the Save button is pressed, the client sends a structured request to the ASP.NET Core API, which applies to the validation and mapping pipeline described in Section II-C and generates the corresponding UBL 2.1/RO_CIUS XML representation.

Figure 6. presents the invoice list view, which consolidates the main operational states of the platform on a single table.

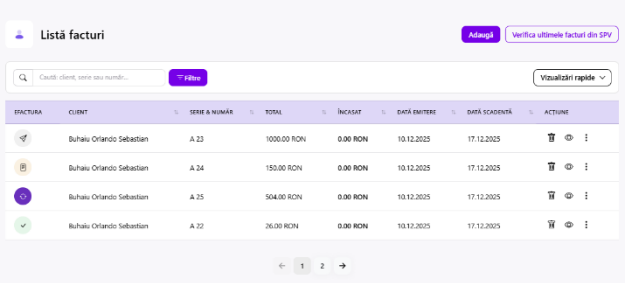

Figure 6. Invoice list and status overview

Each row displays the client’s name, series and number, total amount, collected amount, issue date and due date. The rightmost column exposes the available actions, including delete, view, and a contextual menu. The row labelled “A 23” represents an invoice currently being submitted to SPV ANAF. The associated icon indicates an ongoing upload operation according to the workflow presented in Figure 1. The row “A 24” corresponds to a pro-forma invoice that exists only in the local system and is not reported to SPV. The entry “A 25” denotes an automatically generated recurring invoice, created from a predefined schedule but not necessarily transmitted yet. Finally, row “A 22” shows an invoice that has already been validated by ANAF, for which the accepted status and the related receipt have been retrieved and stored by the back-end services.

At the top of the list, the “Check latest invoices from SPV” button triggers an on-demand synchronization with the SPV ANAF APIs, complementing the periodic background polling described in Section II-D. Together with the real-time notification channel based on SignalR, presented in Section II-E, this interface allows observation of the complete life cycle of an invoice in a single location, from local creation through submission and processing to final acceptance or error notification, while keeping the domain logic and communication with ANAF entirely within the application and integration layers.

III. EXPERIMENTAL RESULTS

This section presents test data from a production like environment, which briefly answers three simple questions: how fast the platform processes an invoice from start to finish, how often does it succeed without human intervention, and how much does it cost per volume. The focus is on the end-to-end behavior of the platform, so that the results reflect the real experience of the user who sends invoices, waits for the receipt, and tracks the final status.

A. Test methodology

To evaluate the platform in a realistic manner, a dedicated environment was used in Azure, with an App Service instance and a **Blob storage** account. The connection to ANAF services was made on the test endpoints. For scenarios where there is no stable public sandbox on B2B, a simulation service was introduced that responds in the same way as the official API, including delays and intermittent errors. The idea was to observe end-to-end behavior, not just local code execution.

The tasks were designed to resemble a typical workday in a finance department. Invoices with realistic content were generated, with around ten items per document, with different combinations of VAT rates, discount lines, and products with or without VAT. The volume ranged from a few dozen documents, used for quick checks, to several thousand, used to see how the processing queue behaves when running constantly for several hours. Two scenarios were tested: a surge scenario, in which many invoices are sent in a short period of time to observe the system's response to a traffic peak, and a conveyor belt scenario, in which documents arrive constantly at fixed intervals, as in a normal workflow.

The code recorded the time for each important step, from generating the XML file to receiving the receipt and the moment when the invoice reaches its final status. At the same time, errors and retrieves were tracked, and processor and memory usage were monitored on the server. For costs, consumption values were taken from Azure and reported by volume so that a cost per 1000 invoices could be estimated on a comparable basis.

The indicators analyzed were those that matter in everyday use. End-to-end latency was viewed by median and 95th percentile, measured from the moment of sending until the appearance of the “accepted” or “error” message in the interface. The transmission success rate showed how many invoices reached “ok” without manual intervention, even if there were automatic retries in the background. The UBL error rate separated content issues from network or unavailability issues to assess how well the local pipeline filters documents before they reach the ANAF server.

B. Results and analysis

In terms of processing times, the platform performed as expected. Under normal conditions, with a flow of approximately ten invoices per minute, the median time from sending to final status was around 2.8 seconds, and the 95th percentile was around 5.2 seconds. Under **high-load** conditions, at approximately fifty invoices per minute, the times increased moderately: the median rose

to four seconds, and the 95th percentile reached approximately eight seconds. These values also include the waiting time for the receipt. In most cases, the receipt was received quickly, in less than a second, and the difference to the final state was generated by the actual processing on the ANAF side, which varied between one and four seconds.

The distribution curves (**CDFs of latency**) for the "nominal" and "peak" scenarios indicate a slightly longer tail at the top: approximately five percent of invoices exceeded ten seconds, and the slowest ones occasionally reached fifteen seconds. Log analysis showed that these isolated cases involved at least one retry, usually caused by a reconnection. Even in high-load scenarios, 95% of invoices are processed in less than eight seconds, which is a reasonable level for current usage.

The maximum flow rate obtained in stress tests was approximately *120 invoices* per minute, equivalent to two invoices per second. After this threshold, a deliberate limitation was imposed to avoid overloading external services. The values observed are consistent with what was noted for **OAuth2 tokens**: for each token, services can support around two to three requests per second. Horizontal scaling, with multiple instances and distinct tokens for each taxpayer, can increase flow almost linearly. For the target audience, consisting of individual companies or ERP integrations, a rate of over one hundred invoices per minute comfortably covers current needs.

To obtain a clear benchmark, the current platform was compared to a baseline variant in which invoices are sent regularly, without concurrent queues and without retribution.

The summarized results for a batch of one thousand invoices are presented below in Table 1.

Config.	TP (inv/min)	P50 (s)	P95 (s)	Succ. (%)	$/1k inv.
Baseline (seq.)	30	5.0	12.4	98.7	0.45
Proposed (par.)	120	2.8	5.2	99.5	0.50
No retry	120	2.5	4.8	92.0	0.50
ANAF fault sim.	120 (0 off)	~2.8	7.5	97.0	0.52

Table 1. System performance (analysis per 1000 invoices)

The basic version remained limited by the individual processing times for each invoice. The flow did not exceed approximately thirty invoices per minute, and the 95th percentile increased significantly due to queue accumulation. The success rate was easily below that of the current platform because, without the retry mechanism, some network errors were not corrected. In the parallel version, the success rate approached 99.5%. Out of a batch of a thousand invoices, only a few required manual interventions, mainly for content reasons.

In the scenario without retry, the success rate dropped to 92%. Basically, some of the temporary errors that would normally have been resolved by retries became definitive failures. The results confirm the important role of the controlled backoff mechanism and explain the performance difference between the full platform and the basic variant.

Compared to typical ERP extension modules or SaaS connectors used in practice, which often rely on synchronous calls to SPV APIs and manual recovery in case of errors, the proposed architecture emphasizes asynchronous processing and explicit observability.

The queue-based delivery model decouples invoice creation from their transmission to ANAF, while idempotent upload semantics avoid duplicates even when client applications or network links are unstable. In addition, the exposure of end-to-end indicators such as P50/P95 latency, success rate without human intervention, and cost per thousand invoices allows operators to plan capacity and define clear service level objectives. These aspects are usually absent from vendor documentation and national guidelines.

Viewed from a broader perspective, these results suggest several design guidelines that may be useful to other developers of tax reporting platforms. First, the almost eight-percentage-point difference in success rate between the variant with retry (≈99.5%) and the one without retry (≈92%) shows that transient errors in external services cannot be ignored; an explicit retry policy with controlled backoff is essential to move towards an almost 100% delivery rate. Second, the fact that throughput stabilizes around 120 invoices per minute even when the number of worker threads is increased beyond four to five indicates that the bottleneck is imposed by the external service (ANAF), not by the application. In such contexts, adding more workers brings little benefit, and horizontal scaling with multiple instances and separate tokens per taxpayer becomes the recommended strategy

Figure 7 shows the cumulative distribution of end-to-end times for the two load scenarios: nominal and peak.

The curve for the nominal scenario rises more steeply (P50 ≈ 2.8 s, P95 ≈ 5.2 s). Under peak load, the curve shifts right (P50 ≈ 4 s, P95 ≈ 8 s) and shows a thin tail: ~1–2% of invoices exceed 10 s, with rare outliers approaching 15 s. These slow cases typically correspond to retries after reconnection. Even so, 95% of invoices finish under 8 s.

The graph in Figure 8 also shows how throughput evolves depending on the number of execution threads in internal scaling tests.

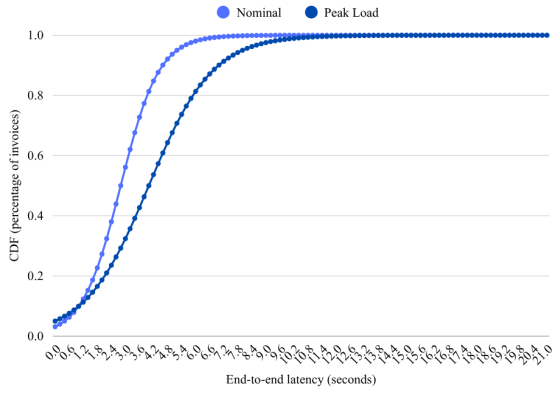

Figure 7. CDF Data

The throughput increases almost linearly up to about four to five workers, after which it stabilizes at around 120 invoices per minute. The limitation is imposed by downstream services, not by the application. From a practical point of view, beyond five threads, additional parallelism no longer brings significant benefits. For higher throughput, horizontal scaling is required, through additional instances and separate tokens for each taxpayer, in accordance with the observations and values in the table.

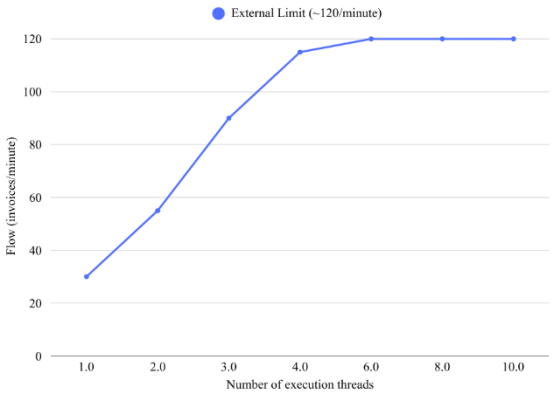

Figure 8. Scaling Data

IV. CONCLUSIONS AND FUTURE WORK

The proposed solution for RO e-Factura, integrated with SPV, aligns with EN 16931, UBL 2.1, and RO_CIUS and is implemented with ASP.NET Core, Blazor, and Azure services. In practice, this means legally compliant XML invoices, transmission to ANAF, real time feedback for the user, and processing times that remain constant even under heavy loads.

From an architectural perspective, the paper demonstrates that a combination of an internal queue layer, a strict idempotence mechanism, and explicit rate limits is sufficient to simultaneously satisfy legal and performance constraints. In addition, designing all critical operations as idempotent actions greatly simplifies the handling of network errors and allows the retry logic to remain transparent from the user's perspective.

Compared to a simple, sequential variant, the results are obvious: almost four times more throughput, times almost halved, and a success rate of around 99.5%. In the basic variant, without retries and without a concurrent queue, losses occur due to transient errors and delays accumulated. In the proposed variant, these situations are absorbed by correctly calibrated retry policies, and manual intervention is only necessary when the data is incorrect.

Beyond e-Invoicing, the same base can also support integration with e-Transport, using the same authentication and monitoring mechanisms. A reporting module would bring visibility to operations: how many invoices have been validated, how long the medians (P50/P95) are for each customer, where recurring errors occur, and what the actual cost per 1000 invoices is. In the medium term, alignment with the **European ViDA** (VAT in the Digital Age) initiatives is also worth pursuing [2]. For implementations that already comply with EN 16931, adaptation will only mean format and rule updates.

From a user experience perspective, the current evaluation has focused on technical indicators and has not yet quantified the impact of real-time notifications on daily activities. A natural direction for applied research is to instrument the interface and conduct controlled studies measuring user satisfaction, perceived response speed, and error reduction when using live status updates compared to periodic manual refreshes. Such measurements would complement the latency and processing capacity results presented in Section III and provide empirical evidence on how the design of notifications influences operators' workload and the risk of omitting or delaying invoice processing.

In conclusion, the solution is ready for production in typical scenarios. It offers good times, a high success rate, and predictable user experience. The next steps are scaling and operation: more automation in the face of outages, more advanced observability tools, and some security optimizations around authentication.

REFERENCES

[1] CEN, "EN 16931-1:2017 — Electronic invoicing — Semantic data model of the core elements of an electronic invoice," 2017. [Online]. Available: https://standards.cencenelec.eu (Accessed: Oct. 13, 2025).
[2] European Commission, "Navigating the eInvoicing standard — documentation," n.d. [Online]. Available: https://ec.europa.eu/digital-building-blocks (Accessed: Oct. 13, 2025).
[3] European Commission, "eInvoicing Country Sheet — Romania," Aug. 14, 2025. [Online]. Available: https://ec.europa.eu/digital-building-blocks (Accessed: Oct. 13, 2025).
[4] Government of Romania, "Emergency Ordinance No. 69/2024 on amendments concerning the RO e-Factura system," 2024. [Online]. Available: https://legislatie.just.ro (Accessed: Oct. 13, 2025).
[5] Internet Engineering Task Force (IETF), "RFC 6749: The OAuth 2.0 Authorization Framework," Oct. 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6749 (Accessed: Oct. 13, 2025).
[6] Internet Engineering Task Force (IETF), "RFC 7636: Proof Key for Code Exchange (PKCE) by OAuth Public Clients," Sept. 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7636 (Accessed: Oct. 13, 2025).
[7] Microsoft, "Azure Storage — Specify conditional headers for Blob service operations (ETag/If-Match/If-None-Match)," 2025. [Online]. Available: https://learn.microsoft.com/azure/storage (Accessed: Oct. 13, 2025).
[8] OASIS, "Universal Business Language Version 2.1 (UBL 2.1)," Nov. 2013. [Online]. Available: https://docs.oasis-open.org/ubl/UBL-2.1.html (Accessed: Oct. 13, 2025).
[9] Parliament of Romania, "Law No. 296/2023 on certain fiscal-budgetary measures to ensure Romania's long-term financial sustainability," Official Gazette No. 977, Oct. 27, 2023. [Online]. Available: https://static.anaf.ro (Accessed: Oct. 13, 2025).
[10] Romanian Ministry of Finance, "Order No. 1366/2021 approving the national RO_CIUS specification (Romania's Core Invoice Usage Specification)," 2021. [Online]. Available: https://static.anaf.ro (Accessed: Oct. 13, 2025).
[11] ISO/IEC, "ISO/IEC 19845:2015 — Information technology — Universal Business Language (UBL)," 2015. [Online]. Available: https://www.iso.org/standard/66370.html (Accessed: Oct. 13, 2025).