Microservices Architecture for Building a Crypto Freelance Exchange

Svitlana Popereshnyak[1,*,†], Maksym Bielikov[1,†] and Anton Bur[1,†]

[1] *National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 37, Prospect Beresteiskyi, Kyiv, 03056, Ukraine*

Abstract

The growth of freelancing exposed several drawbacks in the platforms that are now in use, including high service costs, fraud risks, and late payments [1]. A well-designed system architecture is necessary to develop a freelance platform to guarantee scalability, security, and effectiveness. A modular architecture was chosen to give the required flexibility, scalability, and ease of maintenance to overcome these issues. This paper presents the design of a microservices-based architecture for a crypto freelance exchange platform, which uses Domain-Driven Design principles [2]. The architecture is built to support decentralized transactions, smart contract integration, and secure user authentication. It also ensures high availability and fault tolerance. The system uses a multi-layered architecture incorporating PostgreSQL, MongoDB, Redis, and Web3.js. The main components are a web application, KrakenD API Gateway, auth microservice, files microservice, and main microservice for managing transactions, orders, and payments. They are designed to meet critical non-functional requirements such as scalability, security, and maintainability. Each service can be independently deployed, updated, and scaled according to transaction volumes. With an emphasis on security, the platform uses JWT token techniques and multi-factor authentication to authenticate users. Also, the integration of blockchain technology enhances transparency, enabling freelancers and clients to have a trusted record of all transactions and reducing the risk of fraud. The architecture is visualized through UML and C4 model diagrams showing component interactions and service orchestration. This paper also discusses the rationale behind the chosen technologies, security mechanisms, and the benefits of using a modular microservices approach for building a crypto freelance platform.

Keywords

web application, software architecture, microservices, freelance, Internet of Everything, blockchain

1. Introduction

In recent years, freelance work has emerged as one of the most dynamic and rapidly growing sectors of the global labor market. As of 2025, an estimated 1.57 billion individuals worldwide engage in freelance activities, representing nearly half of the global workforce [3]. This shift is driven by several advantages: access to diverse projects, flexible working hours, and reduced operational costs for employers. However, existing freelance platforms continue to face numerous challenges that hinder user trust, efficiency, and scalability [1]. These include high intermediary fees, a lack of transparency in contract execution, fraud risks, and delayed payments.

Blockchain technology has been widely recognized as a promising solution to these systemic issues. As a decentralized data infrastructure, it enables transparency, immutability, and automated transaction execution through smart contracts [4]. Platforms like Ethereum provide the tools

Workshop "Intelligent information technologies" UkrProg-IIT`2025 co-located with 15th International Scientific and Practical Programming Conference UkrPROG'2025, May 13-14, 2025, Kyiv, Ukraine

[*] Corresponding author.

[†] These authors contributed equally.

spopereshnyak@gmail.com (S. Popereshnyak); bielikov.maksym@lll.kpi.ua (M. Bielikov); bur.anton@lll.kpi.ua (A. Bur)

0000-0002-0531-9809 (S. Popereshnyak);0009-0004-1204-0349 (M. Bielikov); 0009-0004-9320-7445 (A. Bur)

necessary for designing such trustless, decentralized interactions. However, leveraging blockchain alone is insufficient to ensure a resilient and scalable freelance system. The complexity and evolving nature of such platforms requires a robust, modular software architecture that supports high availability, system evolution, and integration with external services and technologies.

This paper focuses not on the business implementation of a freelance exchange but on the design of its microservices-based architecture, which underpins its functionality. The goal is to examine how the principles of software modularity, service isolation, and domain-driven design (DDD) [2] can be applied to architect a modern, crypto-enabled freelance platform that integrates blockchain payments, secure user authentication, distributed data management, and scalable service orchestration.

A multi-layered system architecture is proposed that is composed of independently deployable services, each responsible for a specific domain (e.g., authentication, file management, blockchain interaction). Communication between services is handled via gRPC, while HTTPS and SMTP protocols are used for external communication. Additionally, the architecture leverages technologies such as PostgreSQL, MongoDB, Redis, and monitoring tools like Jaeger and Grafana to meet critical non-functional requirements including performance, security, and fault tolerance.

The architectural design is visualized using C4 model diagrams and UML, providing a detailed understanding of component interactions across abstraction levels. This paper presents the rationale behind the selection of technologies and architectural patterns and highlights how this microservices approach enables the modular, scalable, and secure operation of a decentralized freelance platform.

2. Related work

The integration of blockchain technology, particularly through smart contracts, has gained significant attention for digital platforms like freelance marketplaces. In [4] authors explain that smart contracts are best suited for simple transactions, such as transferring funds once certain conditions are met. However, the complexity of real-world contracts, such as subjective legal criteria, remains a challenge for smart contracts, and they are far from being a complete replacement for traditional legal processes.

In [5], a decentralized, non-profitable freelancing platform is proposed. It utilizes a public blockchain and smart contracts to enhance transparency and trust compared to centralized platforms.

The B. Pallam and M. Gore paper proposes Boomerang, a decentralized freelance administration system built on Hyperledger Fabric, aiming to address security, credibility, and flexibility issues in centralized freelancing platforms through blockchain-based solutions and fine-grained access control [6].

In [7], the paper presents HireChain, a decentralized freelancing system built on the Ethereum blockchain, designed to address issues of trust, reliability, and payment delays between employers and freelancers through smart contracts.

The need for secure cryptographic operations [8] and efficient cloud-based data processing [9] further highlights the technological challenges addressed by the proposed platform, ensuring both transaction security and scalable management of freelance service operations.

3. System architecture

The proposed system architecture for a crypto freelance exchange platform is based on a microservices architecture utilizing principles of Domain-Driven Design (DDD) [2]. The primary architectural pattern is a layered architecture, which ensures flexibility, scalability, and ease of maintenance. To visualize the system, the C4 model is used across three abstraction levels, illustrating components, containers, and internal structures of the services.

At the highest abstraction (C4 Level 1, see Figure 1), the system is represented as a unified solution interacting with external actors and services such as Ethereum blockchain, LLM-based text generation services (used for automating job description enhancements, contract summarization, and intelligent proposal suggestions), and external email providers. The “User” is generalized to include administrators, clients, freelancers, and guests – all interacting with the system through a unified entry point.

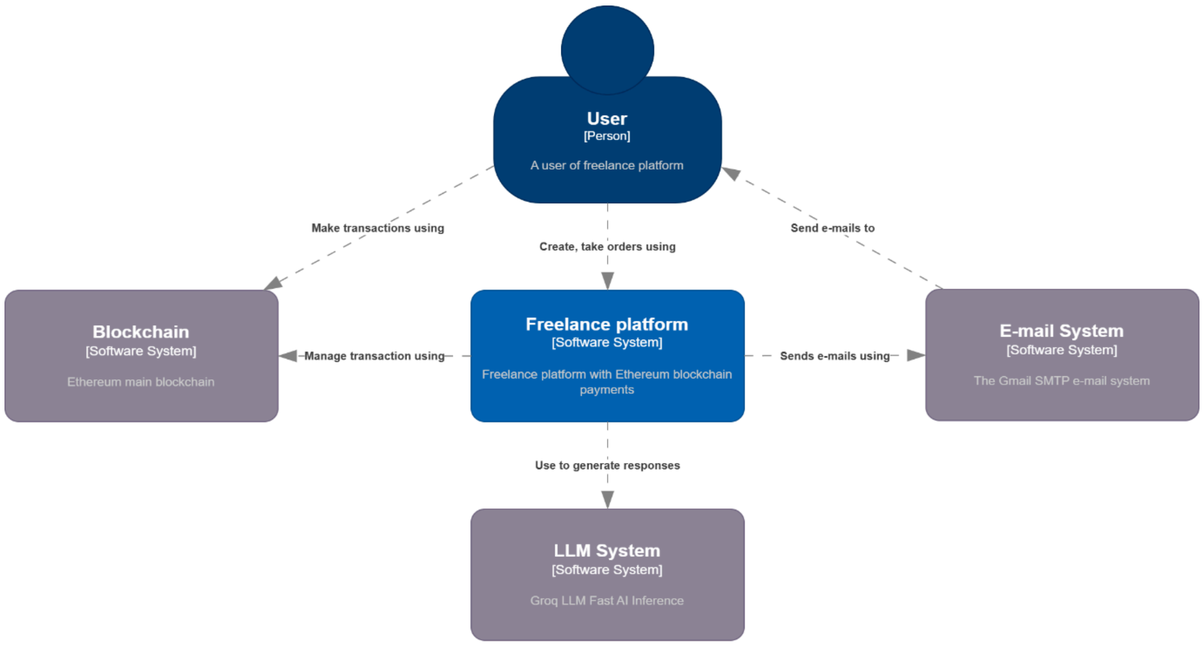

Figure 1: C4 Level 1 – Context Diagram of the Crypto Freelance Exchange

3.1. Container view

At C4 Level 2, the architecture decomposes into the following core containers:

1. Web Application – A frontend interface built with Next.js, delivering a responsive UI for all platform users.
2. API Gateway – Central entry point for routing HTTP/gRPC requests to backend services, handling authentication, throttling, and observability.
3. Auth Service – Handles user authentication and authorization, including login, registration, multi-factor authentication (MFA), account bans, and JWT issuance. It integrates with PostgreSQL (persistent storage), Redis (token caching), and Gmail SMTP for email verification and MFA.
4. Main Service – Responsible for core business logic, including project management, user roles, dispute handling, smart contract interaction, and AI validation. It uses MongoDB for flexible and scalable data persistence, and communicates with Ethereum smart contracts via Web3.js and Groq’s LLM API for prompt-based assistance.
5. Files Service – Manages upload, storage (via Blob Storage), and secure access to user uploaded files.

Communication between services is performed over gRPC for internal calls, and HTTPS/JSON for external APIs (see Figure 2).

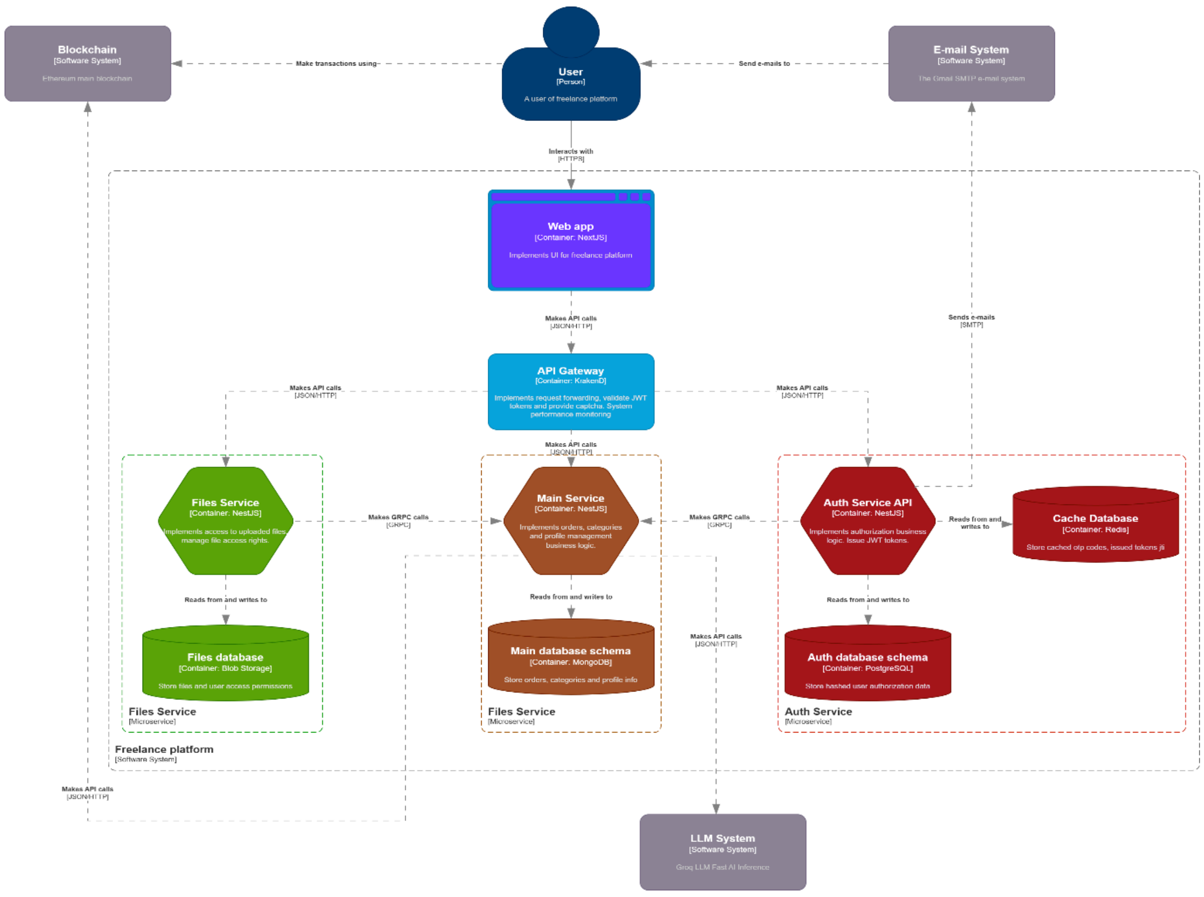

Figure 2: C4 Level 2 – Container Diagram of Core Microservices

3.2. Component view

At C4 Level 3 (see Figure 3), each container is further decomposed into components, emphasizing modularity and separation of concerns. Each microservice contains:

1. Controllers for handling incoming API/gRPC requests.
2. Services that encapsulate business logic.
3. Repositories or data access layers for communicating with databases.
4. Clients or adapters for external services (SMTP, Ethereum, LLMs).

gRPC is chosen over REST for inter-service communication due to its low latency and efficient binary protocol (Protocol Buffers). For external APIs, HTTPS ensures encrypted communication and server authentication.

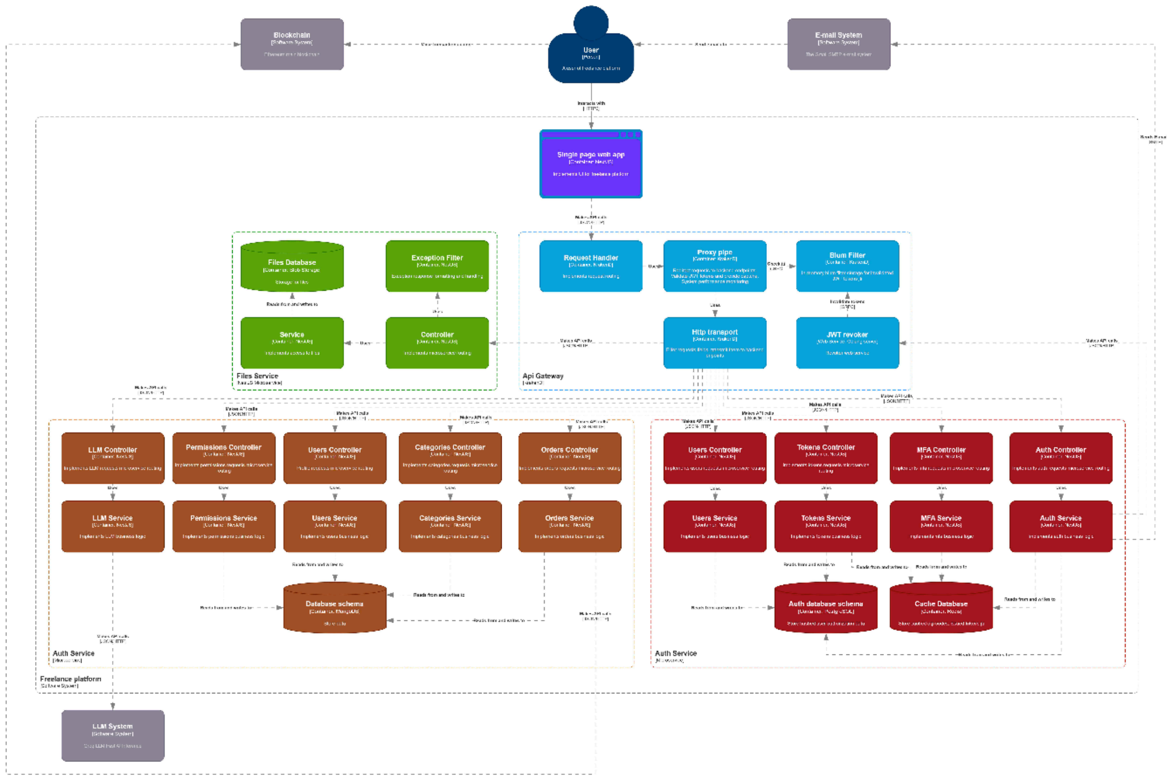

Figure 3: C4 Level 3 – Internal Architecture of All Microservices

The secure gRPC protocol is used to transfer data between microservices because it has many advantages. First, gRPC allows for high data exchange rates by using the HTTP/2 protocol, which supports multithreading and efficient connection management. This is especially important in a microservice architecture, where each microservice must connect to others to perform complex operations. In addition, gRPC uses the Buffers protocol serialization, which allows for a smaller amount of data to be transferred and a higher efficiency in processing requests. This reduces the latency between requests and responses, which improves overall system performance.

To interact with the API, users use the HTTPS protocol, which provides a high level of security during data transfer between clients and the server. HTTPS uses the TLS protocol to encrypt data, which guarantees protection against eavesdropping and man-in-the-middle attacks. HTTPS also provides server authentication, which helps prevent request forgery and promotes user trust in the API. Additionally, HTTPS is a standard for interaction through web browsers, ensuring compatibility with a wide range of browsers and ensuring that the connection remains secure when users change networks or move to new devices.

3.3. UML package and sequence diagrams

To illustrate logical dependencies and relationships, a UML package diagram was developed for auth microservice (see Figure 4). It integrates multiple functionalities like JWT tokens, two-factor authentication using TOTP and email, and API tokens.

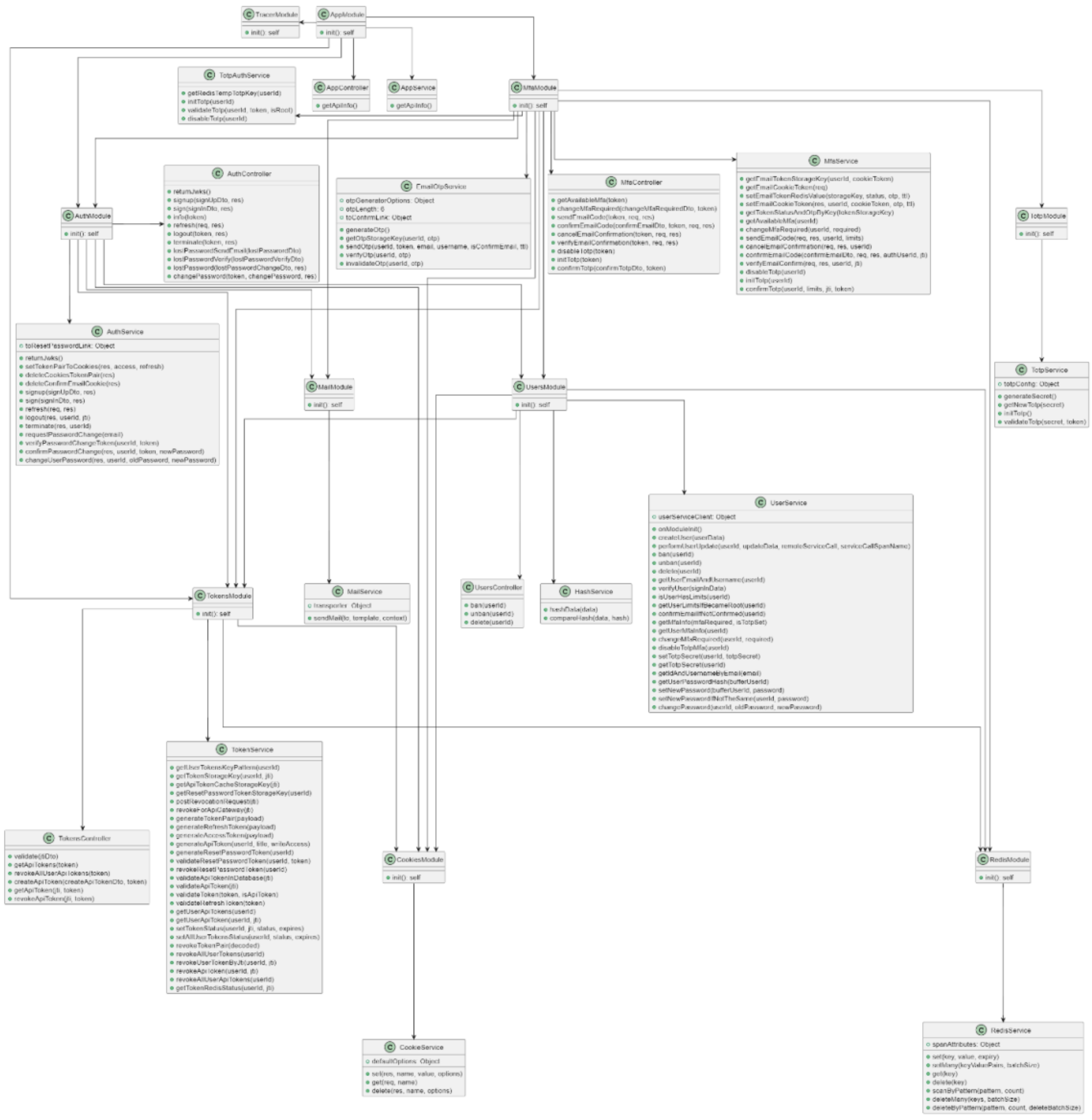
TracerModule
init(): self
AppModule
init(): self
TotpAuthService
getRedisTempTotpKey(userId)
initTotp(userId)
validateTotp(userId, token, isRoot)
disableTotp(userId)
AppController
getApiInfo()
AppService
getApiInfo()
MfaModule
init(): self
AuthController
returnJwks()
signup(signUpDto, res)
sign(signInDto, res)
info(token)
refresh(req, res)
logout(token, res)
terminate(token, res)
lostPasswordSendEmail(lostPasswordDto)
lostPasswordVerify(lostPasswordVerifyDto)
lostPassword(lostPasswordChangeDto, res)
changePassword(token, changePassword, res)
EmailOtpService
otpGeneratorOptions: Object
otpLength: 6
toConfirmLink: Object
generateOtp()
getOtpStorageKey(userId, otp)
sendOtp(userId, token, email, username, isConfirmEmail, ttl)
verifyOtp(userId, otp)
invalidateOtp(userId, otp)
MfaController
getAvailableMfa(token)
changeMfaRequired(changeMfaRequiredDto, token)
sendEmailCode(token, req, res)
confirmEmailCode(confirmEmailDto, token, req, res)
cancelEmailConfirmation(token, req, res)
verifyEmailConfirmation(token, req, res)
disableTotp(token)
initTotp(token)
confirmTotp(confirmTotpDto, token)
MfaService
getEmailTokenStorageKey(userId, cookieToken)
getEmailCookieToken(req)
setEmailTokenRedisValue(storageKey, status, otp, ttl)
setEmailCookieToken(res, userId, cookieToken, otp, ttl)
getTokenStatusAndOtpByKey(tokenStorageKey)
getAvailableMfa(userId)
changeMfaRequired(userId, required)
sendEmailCode(req, res, userId, limits)
cancelEmailConfirmation(req, res, userId)
confirmEmailCode(confirmEmailDto, req, res, authUserId, jti)
verifyEmailConfirm(req, res, userId, jti)
disableTotp(userId)
initTotp(userId)
confirmTotp(userId, limits, jti, token)
AuthModule
init(): self
TotpModule
init(): self
AuthService
toResetPasswordLink: Object
returnJwks()
setTokenPairToCookies(res, access, refresh)
deleteCookiesTokenPair(res)
deleteConfirmEmailCookie(res)
signup(signUpDto, res)
sign(signInDto, res)
refresh(req, res)
logout(res, userId, jti)
terminate(res, userId)
requestPasswordChange(email)
verifyPasswordChangeToken(userId, token)
confirmPasswordChange(res, userId, token, newPassword)
changeUserPassword(res, userId, oldPassword, newPassword)
MailModule
init(): self
UsersModule
init(): self
TotpService
totpConfig: Object
generateSecret()
getNewTotp(secret)
initTotp()
validateTotp(secret, token)
UserService
userServiceClient: Object
onModuleInit()
createUser(userData)
performUserUpdate(userId, updateData, remoteServiceCall, serviceCallSpanName)
ban(userId)
unban(userId)
delete(userId)
getUserEmailAndUsername(userId)
verifyUser(signInData)
isUserHasLimits(userId)
getUserLimitsIfBecameRoot(userId)
confirmEmailIfNotConfirmed(userId)
getMfaInfo(mfaRequired, isTotpSet)
getUserMfaInfo(userId)
changeMfaRequired(userId, required)
disableTotpMfa(userId)
setTotpSecret(userId, totpSecret)
getTotpSecret(userId)
getIdAndUsernameByEmail(email)
getUserPasswordHash(bufferUserId)
setNewPassword(bufferUserId, password)
setNewPasswordIfNotTheSame(userId, password)
changePassword(userId, oldPassword, newPassword)
TokensModule
init(): self
MailService
transporter: Object
sendMail(to, template, context)
UsersController
ban(userId)
unban(userId)
delete(userId)
HashService
hashData(data)
compareHash(data, hash)
TokenService
getUserTokensKeyPattern(userId)
getTokenStorageKey(userId, jti)
getApiTokenCacheStorageKey(jti)
getResetPasswordTokenStorageKey(userId)
postRevocationRequest(jti)
revokeForApiGateway(jti)
generateTokenPair(payload)
generateRefreshToken(payload)
generateAccessToken(payload)
generateApiToken(userId, title, writeAccess)
generateResetPasswordToken(userId)
validateResetPasswordToken(userId, token)
revokeResetPasswordToken(userId)
validateApiTokenInDatabase(jti)
validateApiToken(jti)
validateToken(token, isApiToken)
validateRefreshToken(token)
getUserApiTokens(userId)
getUserApiToken(userId, jti)
setTokenStatus(userId, jti, status, expires)
setAllUserTokensStatus(userId, status, expires)
revokeTokenPair(decoded)
revokeAllUserTokens(userId)
revokeUserTokenByJti(userId, jti)
revokeApiToken(userId, jti)
revokeAllUserApiTokens(userId)
getTokenRedisStatus(userId, jti)
TokensController
validate(jtiDto)
getApiTokens(token)
revokeAllUserApiTokens(token)
createApiToken(createApiTokenDto, token)
getApiToken(jti, token)
revokeApiToken(jti, token)
CookiesModule
init(): self
RedisModule
init(): self
CookieService
defaultOptions: Object
set(res, name, value, options)
get(req, name)
delete(res, name, options)
RedisService
spanAttributes: Object
set(key, value, expiry)
setMany(keyValuePairs, batchSize)
get(key)
delete(key)
scanByPattern(pattern, count)
deleteMany(keys, batchSize)
deleteByPattern(pattern, count, deleteBatchSize)

Figure 4: UML Package Diagram of the Auth Microservice

Additionally, a module dependency diagram was built for the authorization microservice. It depicts the relationships between modules, their services, and controllers within the application architecture.

Additionally, a sequence diagram was created to illustrate the user authentication process (see Figure 6). It traces the flow from login form submission to JWT issuance, including MFA via email and caching with Redis.

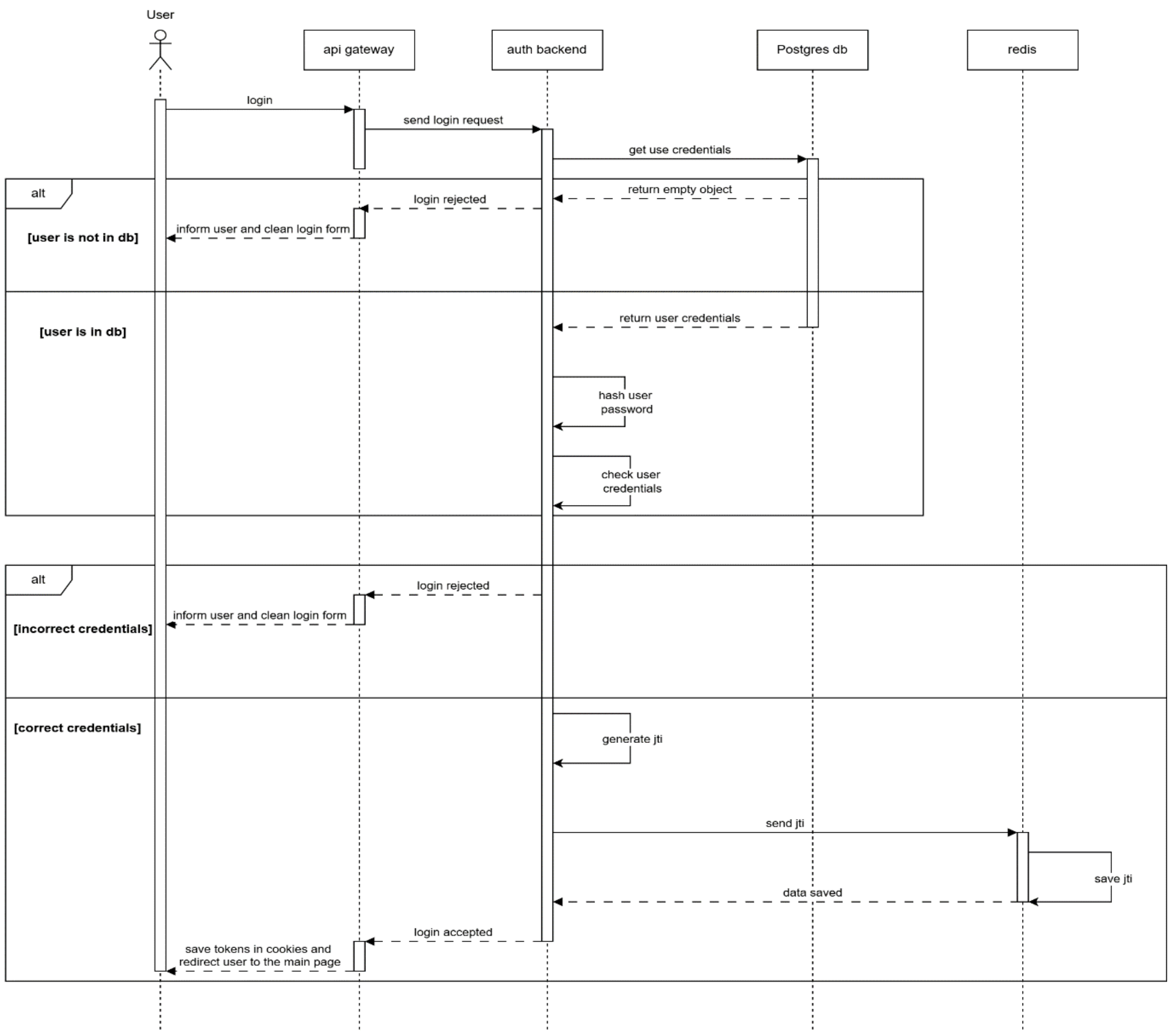

Figure 5: Sequence Diagram of User Authentication Flow

It was decided to demonstrate this type of diagram to visually represent the architecture of modules written on the NestJS framework. This is useful for understanding the code structure, simplifying debugging, and optimizing the architecture. It helps developers navigate complex systems and maintain modularity.

3.4. Data security analysis

Data security analysis in the Web application of the freelance platform with payment via the Ethereum blockchain takes into account potential risks and provides protection against common threats by integrating modern security technologies.

Possible attacks that can be directed at the system include SQL injections, cross-site scripting (XSS) attacks, brute force attacks (DDoS), phishing, exploitation of vulnerabilities in third-party software, attacks on blockchain wallets, and data interception via Man-in-the-Middle.

To protect against SQL injections, the system uses parameterized queries and ORM tools that minimize the risk of executing malicious commands in the database. XSS vulnerabilities are neutralized by thoroughly checking and cleaning input data, as well as encrypting sensitive information before it is displayed on the frontend.

To combat brute force attacks, a limit on the number of login attempts, captchas, and two-factor authentication are used at the registration and authorization stages. Phishing risks are minimized by encrypting data via SSL TLS and creating an interface that includes direct user warnings about possible threats. Current software updates reduce the risk of exploiting vulnerabilities in third-party libraries and frameworks.

To protect integration with blockchain wallets, transaction signing algorithms, private key encryption, and interaction only through official API interfaces have been implemented. Protection against Man-in-the-Middle attacks is provided by using HTTPS.

Data integrity and confidentiality are maintained by encrypting the storage of confidential information (passwords, tokens) using AES-256 and bcrypt algorithms.

3.5. Database architecture

In the developed system architecture, a combination of relational and non-relational databases is employed to ensure both data integrity and operational flexibility. PostgreSQL serves as the primary storage solution for structured data, providing strong consistency guarantees, support for complex queries, and robust transaction management. It is specifically utilized by the authentication microservice (see Figure 6), where reliability, strict data schema enforcement, and security are critical for managing user credentials and session information.

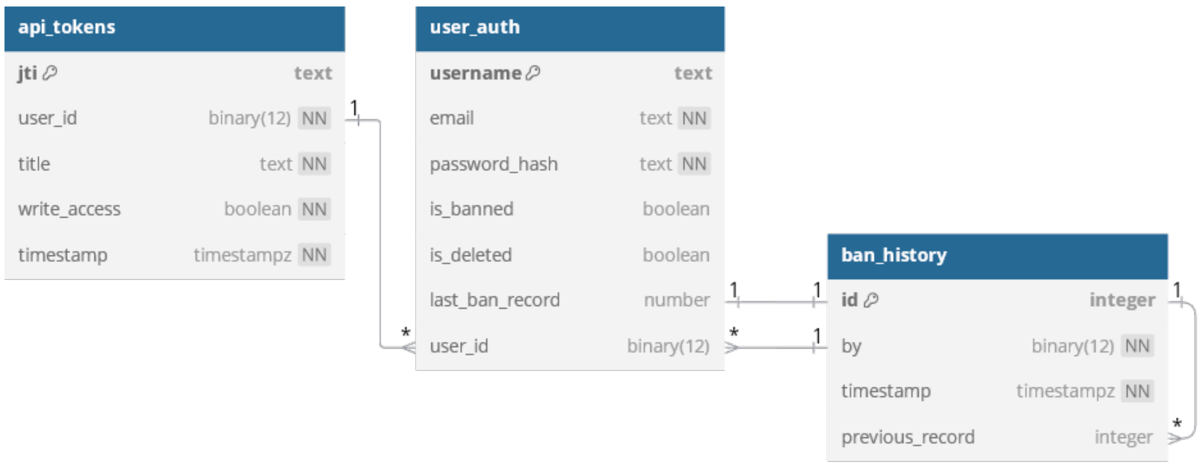

Figure 6: PostreSQL ER-diagram of auth microservice

On the other hand, MongoDB is used within the main microservice (see Figure 7) to efficiently handle dynamic and semi-structured data. Its document-oriented model allows flexible schema designs, enabling the system to easily adapt to changes in data structure without requiring complex migrations. This choice is particularly important for managing variable and evolving datasets, such as real-time parking space information, vehicle detection results, and user interaction logs.

Additionally, Redis is integrated as an in-memory data store to cache temporary and frequently accessed data. By storing session states, intermediate computation results, and real-time status updates in Redis, the system significantly reduces database query loads and accelerates response times, thus improving overall performance and scalability.

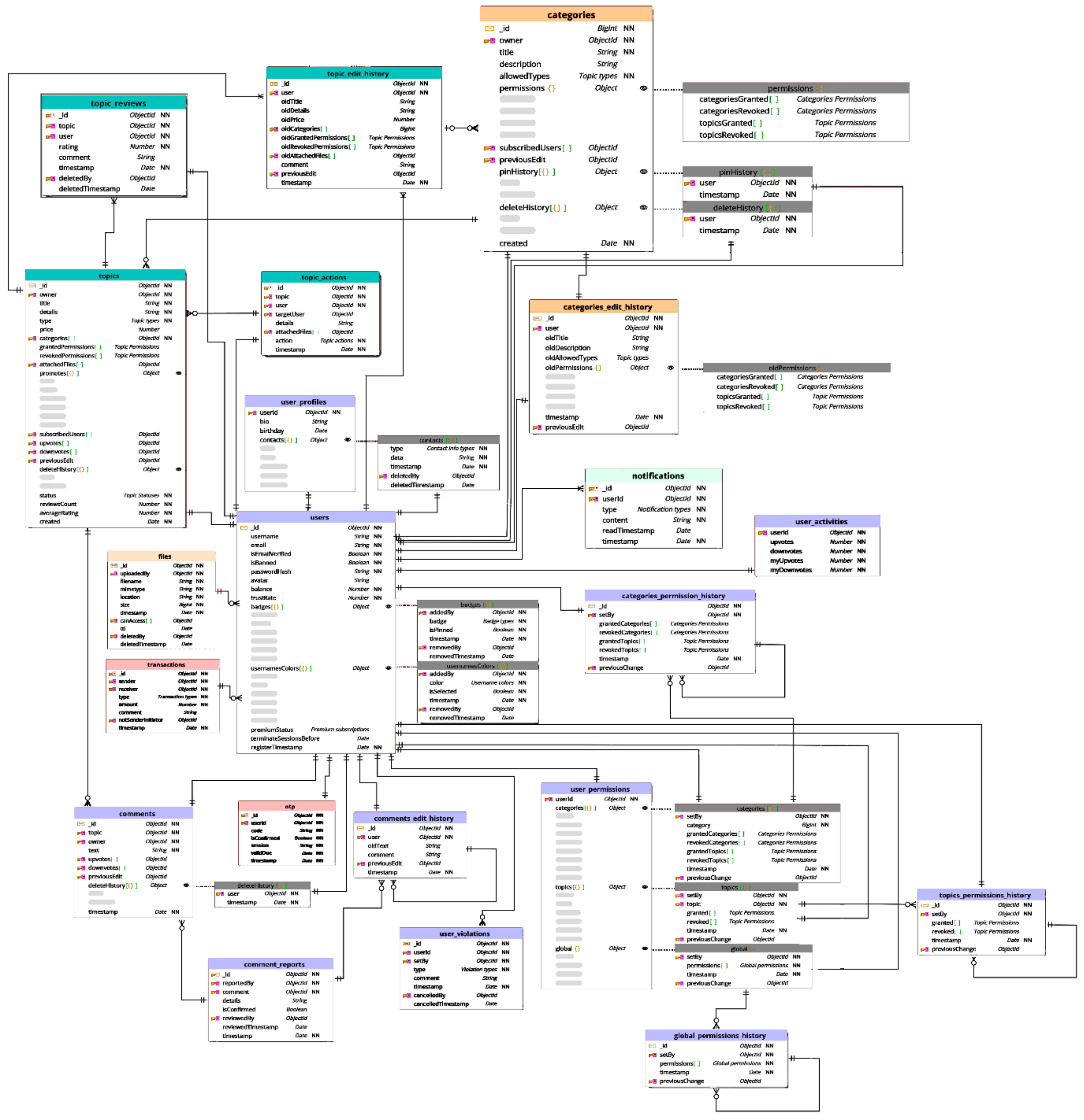
Figure 7: MongoDB ER-diagram of main microservice

The use of two specialized databases—PostgreSQL for the authentication subsystem and MongoDB for the core functional services—ensures that the system maintains a balance between strong data consistency where necessary and high flexibility and performance where demanded by operational dynamics.

4. Workflow of an order with smart contracts

The complete order workflow is shown in Figure 8.

The customer creates a task on the platform website, and a smart contract is generated to his wallet address, which blocks the amount of money he has set (the budget for execution). The customer can cancel the order. This can be done both on the website of the web application and by calling the cancellation function independently on the blockchain via a crypto wallet. In this case, the money will be unlocked, and the smart contract will be considered completed.

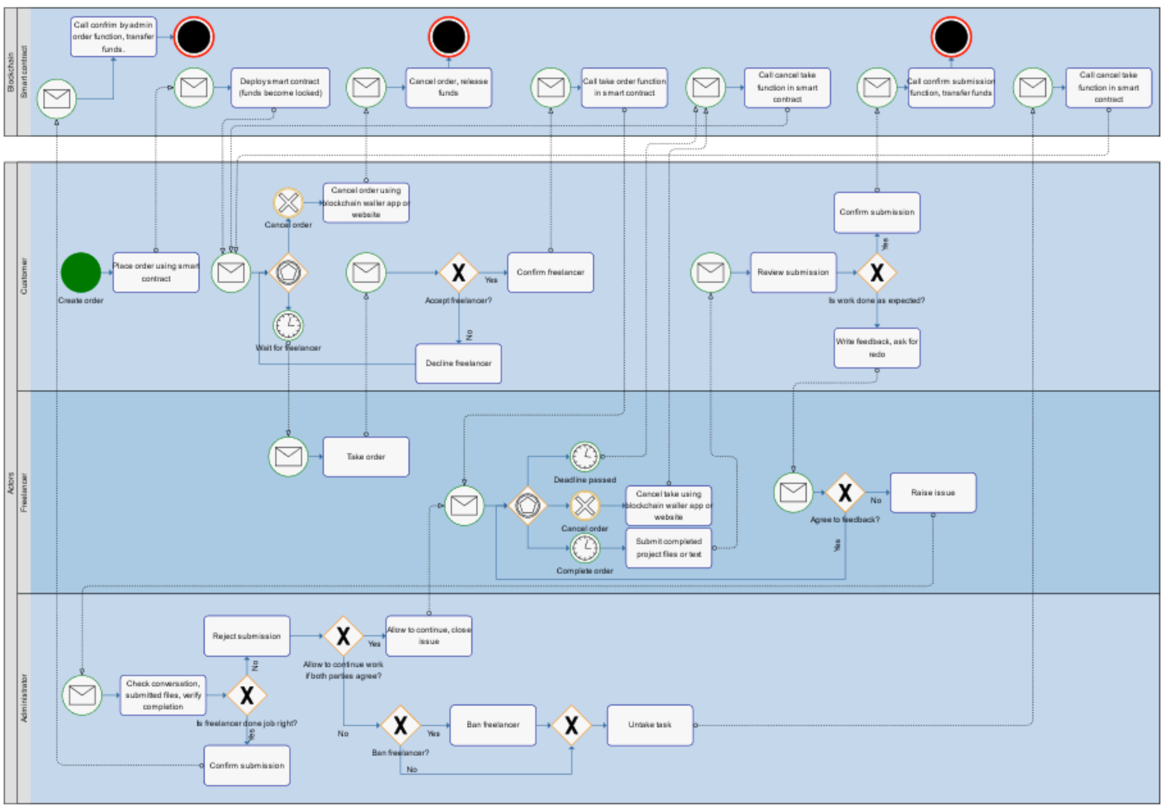
Blockchain
Smart contract
Call confirm by admin order function, transfer funds.
Deploy smart contract (funds become locked)
Cancel order, release funds
Call take order function in smart contract
Call cancel take function in smart contract
Call confirm submission function, transfer funds
Call cancel take function in smart contract
Actors
Customer
Create order
Place order using smart contract
Cancel order
Cancel order using blockchain waller app or website
Wait for freelancer
Accept freelancer?
Yes
No
Confirm freelancer
Decline freelancer
Review submission
Is work done as expected?
Yes
Confirm submission
Write feedback, ask for redo
Freelancer
Take order
Deadline passed
Cancel order
Cancel take using blockchain waller app or website
Complete order
Submit completed project files or text
Agree to feedback?
No
Yes
Raise issue
Administrator
Check conversation, submitted files, verify completion
Is freelancer done job right?
No
Yes
Reject submission
Confirm submission
Allow to continue work if both parties agree?
Yes
No
Allow to continue, close issue
Ban freelancer?
Yes
No
Ban freelancer
Untake task

Figure 8: BPMN Diagram of the Freelance Order Execution Process

Otherwise, the customer will wait until one of the freelancers takes the order on our platform. And when this happens, he will receive a corresponding message. The customer must reject or accept the freelancer who wants to perform the task. When the performer has been selected, a record is created in the blockchain. The security of the election is guaranteed because the "take order" function can only be called from the wallet that was used to initiate the smart contract.

The freelancer, in turn, can refuse to perform the task by calling the "cancel order" function through a wallet connected to the platform, but this will worsen his rating. Then, the customer will can wait and choose another freelancer or cancel his task. Automatic cancellation will work when the deadline set by the customer is passed.

The freelancer must attach the files of the completed task. The customer will check them, and if they both agree, the money will be transferred to the performer. If a conflict arises, the platform administrator gets involved, who must check the chat history and decide whether the freelancer has completed the task assigned to him. If so, the money is sent to the performer, and the smart contract is completed. If not, the administrator communicates with the parties to the conflict about whether to allow the current freelancer to make changes to the task and continue working. In case of agreement, the performer must attach new files before the deadline or refuse to perform it.

If the parties to the conflict do not reach an agreement, the administrator decides whether to allow the freelancer to take new orders on the platform and has the right to block him. The task performer is canceled from the smart contract. The customer can wait for a new freelancer. However, if the task is no longer relevant, he can cancel it. The money will be unlocked and the smart contract will be considered completed.

5. Security considerations

All communication within the platform, including internal microservice interactions and external communications, is encrypted using HTTPS. This protects against man-in-the-middle (MITM) attacks.

Smart contracts deployed on the Ethereum blockchain are immutable once they are executed. This guarantees that they cannot be altered or tampered with by any party, ensuring fairness and trust in the transaction.

JWT tokens are stored in the client's cookies with HttpOnly and Secure flags to prevent session hijacking and cross-site scripting (XSS) attacks. They have a defined expiration time, so in case of suspicious activity, tokens can be invalidated via API, ensuring that stale or compromised sessions do not remain active.

Blockchain-based smart contracts provide an escrow-like mechanism, where funds are only released when predefined conditions (e.g., task completion) are met. This minimizes the risk of fraud or non-completion of tasks, ensuring fair payments for freelancers and clients.

Passwords are stored as bcrypt hash, preventing the risk of credential theft. To further strengthen the authentication process, two-factor authentication via TOTP and email can be enabled. This ensures that even if login credentials are compromised, unauthorized access is mitigated.

6. Conclusion

The architecture presented in this paper demonstrates how a microservices-based approach can effectively address the challenges faced by current freelance platforms. The platform achieves high scalability, maintainability, and fault tolerance by leveraging a modular design according to Domain-Driven Design principles [2]. Blockchain technology integration for transactions ensures reliable service delivery and robust security.

Each system component is carefully designed to operate independently while maintaining seamless interoperability. This separation of concerns simplifies future system enhancements and facilitates integration with new external services. Smart contracts add an extra layer of trust and transparency, protecting the interests of all parties involved. This mechanism ensures that funds are securely held and only released upon successful task completion, thus mitigating fraud risks.

Adopting a microservices architecture paired with blockchain technology positions the proposed freelance platform as a forward-thinking solution for the evolving gig economy. The platform outlined in this paper meets current industry demands. It lays a solid foundation for continuous improvement and adaptation to emerging technologies, ensuring long-term operational success and user satisfaction.

Declaration on Generative AI

During the preparation of this work, the authors used AI program Chat GPT 4.0 for correction of text grammar. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A. Batool, Y. Byun, Reduction of Online Fraudulent Activities in Freelancing Sites Using Blockchain and Biometric (2022). doi:10.3390/electronics11050789.

[2] S. K. Jaiswal, R. Agrawal, Domain-Driven Design (DDD) — Bridging the Gap between Business Requirements and Object-Oriented Modeling, International Journal of Innovative Research in Engineering and Management (IJIREM), 11(2) (2024): 79-83, doi:10.55524/ijirem. 2024.11.2.16.

[3] N. Kumar, Latest Freelance Statistics 2025 — Industry Size & Trends, 2025. URL: https://www.demandsage.com/freelance-statistics/.

[4] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin and F. -Y. Wang, An Overview of Smart Contract: Architecture, Applications, and Future Trends, in: Proceedings of the IEEE Intelligent Vehicles Symposium (IV), Changshu, China, 2018, pp. 108-113. doi: 10.1109/ IVS.2018.8500488.

[5] P. Deshmukh, S. Kalwaghe, A. Appa, A. Pawar, Decentralised Freelancing using Ethereum Blockchain, in: Proceedings of the International Conference on Communication and Signal Processing (ICCSP), Chennai, India, 2020, pp. 881-883. doi: 10.1109/ICCSP 48568.2020.9182127.

[6] B. Pallam, M. M. Gore, Boomerang: Blockchain-based Freelance Paradigm on Hyperledger, in: Proceedings of the 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Kanpur, India, 2019, pp. 1-6. doi: 10.1109/ICCCNT 45670.2019.8944572.

[7] M. Gandhi et al. Decentralized Freelancing System-Trust and Transparency. International Research Journal of Engineering and Technology (IRJET) 6.09 (2019).

[8] S. Popereshnyak, Y. Novikov, Y. Zhdanova Cryptographic system security approaches by monitoring the random numbers generation. (2024) CEUR Workshop Proceedings, 3826, pp. 301-309. URL: https://ceur-ws.org/Vol-3826/short21.pdf

[9] M. Reshetniak, S. Popereshnyak Method for accessing and processing multimedia content in a cloud environment, in: Proceedings of the IEEE International Scientific-Practical Conference: Problems of Infocommunications Science and Technology, 9061463, pp. 71-76. doi: 10.1109/PICST47496.2019. 9061463