МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кваліфікаційна наукова праця
на правах рукопису

КРИВИЙ ЯРОСЛАВ ВЯЧЕСЛАВОВИЧ

УДК 004.75:004.42]: 517.962(043.5)

ДИСЕРТАЦІЯ

ПОДІЄВО-ОРІЄНТОВАНА МІКРОСЕРВІСНА АРХІТЕКТУРА МАСШТАБОВАНОЇ ПЛАТФОРМ СКІНЧЕННО-ЕЛЕМЕНТНОГО АНАЛІЗУ

122 Комп’ютерні науки
12 Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії з **комп’ютерних наук**

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Я. В. Кривий

Науковий керівник:
Лісняк Андрій Олександрович,
кандидат фізико-математичних наук, доцент

Запоріжжя – 2026

АНОТАЦІЯ

Кривий Я. В. Подієво-орієнтована мікросервісна архітектура масштабованої платформи скінченно-елементного аналізу. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки». – Запорізький національний університет, Запоріжжя, 2026.

У дисертаційній роботі розв'язано актуальне науково-прикладне завдання організації повного циклу скінченно-елементного аналізу (FEA) в розподіленому програмному середовищі з можливістю незалежного горизонтального масштабування найбільш ресурсоємної обчислювальної стадії та збереження контрольних чисельних характеристик результатів. Запропоновано подієво-орієнтовану мікросервісну архітектуру FEA-платформи, у якій підготовку даних, чисельне розв'язання, післяобробку та візуалізацію результатів представлено як логічно відокремлені компоненти з визначеними контрактами взаємодії, незалежними життєвими циклами розгортання та різними профілями використання обчислювальних ресурсів.

Актуальність дослідження зумовлена зростанням складності інженерних і наукових задач, що розв'язуються методом скінченних елементів, збільшенням обсягів геометричних, сіткових та чисельних даних, а також потребою в паралельному виконанні значної кількості незалежних обчислювальних задач. Традиційні монолітні FEA-системи поєднують підготовку геометрії, генерацію сітки, побудову й розв'язання систем рівнянь, післяобробку та візуалізацію в межах єдиного програмного середовища. Такий підхід спрощує локальну взаємодію компонентів, проте ускладнює незалежне масштабування окремих стадій, інтеграцію з хмарною інфраструктурою, асинхронне виконання довготривалих задач і оперативне інформування користувача про стан обчислень.

У роботі обґрунтовано, що для платформ скінченно-елементного аналізу доцільним є поєднання мікросервісної декомпозиції, подієво-орієнтованої

взаємодії, спеціалізованого зберігання великих артефактів і контейнерної оркестрації. Такий підхід дозволяє відокремити керування життєвим циклом задачі від передавання та зберігання великих масивів даних, а також змінювати кількість обчислювальних виконавців без масштабування всіх компонентів платформи.

Об’єкт дослідження – процеси побудови та функціонування розподілених програмних платформ скінченно-елементного аналізу.

Предмет дослідження – моделі, методи та архітектурні рішення подієво-орієнтованої мікросервісної організації FEA-конвеєра, механізми незалежного горизонтального масштабування його обчислювальної стадії, а також методи експериментального оцінювання продуктивності, чисельної коректності та відтворюваності результатів.

Мета дослідження полягає у підвищенні масштабованості й керованості програмної платформи скінченно-елементного аналізу шляхом розробки подієво-орієнтованої мікросервісної архітектури, реалізації прототипу повного FEA-конвеєра та експериментального визначення впливу кількості обчислювальних виконавців і складності задачі на продуктивність платформи.

Для досягнення поставленої мети використано методи системного та порівняльного аналізу програмних архітектур, методи моделювання програмних систем у нотаціях UML та IDEF0, принципи мікросервісної й подієво-орієнтованої архітектури, методи проєктування контрактів програмних інтерфейсів і повідомлень, засоби контейнеризації та оркестрації. Для реалізації й перевірки чисельної складової застосовано метод скінченних елементів, лінійні P1-апроксимації на трикутних і тетраедральних сітках, розріджене подання матриць і метод виготовлених розв’язків. Під час обробки експериментальних результатів використано методи описової статистики, аналіз середніх значень, медіани, 95-го перцентиля, вибіркового стандартного відхилення, відносного прискорення та ефективності масштабування.

У першому розділі проведено аналіз теоретичних і прикладних передумов застосування мікросервісної архітектури до систем скінченно-елементного

аналізу. Розглянуто сутність методу скінченних елементів і повного FEA-циклу, що охоплює підготовку геометрії, генерацію сітки, задання математичної моделі, чисельне розв’язання, післяобробку та візуалізацію. Проаналізовано особливості монолітної та мікросервісної архітектур, їхні переваги, обмеження й умови доцільного застосування.

Виконано порівняння існуючих комерційних і відкритих FEA-рішень. Встановлено, що більшість із них забезпечують розвинені засоби чисельного моделювання, проте питання незалежного масштабування окремих стадій, відкритої подієвої координації, відокремлення великих артефактів від керівних повідомлень і відтворюваного експериментального оцінювання масштабованості залишаються недостатньо розкритими. Сформульовано передумови переходу до сервісної декомпозиції та асинхронної взаємодії в FEA-платформах.

У другому розділі розроблено концептуальну модель подієво-орієнтованої мікросервісної FEA-платформи. Сформульовано функціональні та нефункціональні вимоги до системи, визначено ролі користувача й компонентів, побудовано діаграми варіантів використання, компонентів, класів, послідовностей, діяльності та розгортання. Функціональну структуру додатково формалізовано засобами IDEF0.

Запропоновано поділ повного FEA-циклу між компонентами Preprocessor, Processor, Postprocessor, Aggregator і модулем тривимірної візуалізації. Визначено клієнтський і серверний контури, межі відповідальності компонентів, зовнішню точку входу, модель передавання станів і подій, а також принципи зберігання метаданих та обчислювальних артефактів.

Обґрунтовано відокремлення каналів передавання керівних подій, службових даних про стан задачі та великих обчислювальних даних. Керівний контур формують події, статуси, часові мітки та посилання на артефакти. Великі дані – скінченно-елементні сітки, описи постановок, чисельні розв’язки й пакети візуалізації – зберігаються в об’єктному сховищі та передаються між стадіями через стабільні посилання. Це зменшує навантаження на брокер повідомлень і послаблює зв’язність прикладних компонентів.

У третьому розділі реалізовано експериментальний прототип запропонованої архітектури. Клієнтський вебзастосунок побудовано з використанням Angular і Three.js. Єдиною зовнішньою точкою входу є Traefik, автентифікацію організовано за допомогою Keycloak і OpenID Connect, а функції агрегування станів, створення задач і оперативного передавання змін стану задачі реалізовано в BFF на Node.js і Socket.IO.

Подієвий контур побудовано на основі Apache Kafka. Для повідомлень визначено уніфіковану структуру, яка містить тип події, ідентифікатор задачі, часову мітку, версію контракту та прикладні дані. Стан задач і часові характеристики стадій зберігаються в PostgreSQL, а великі проміжні та кінцеві обчислювальні дані – у MinIO. Redis і Celery використовуються для внутрішньої організації черги обчислювальних задач.

Preprocessor реалізовано як окремий Python-сервіс, який формує двовимірні та тривимірні сітки засобами Gmsh і створює опис математичної постановки. Processor розділено на споживач подій і множину незалежно масштабованих Celery виконавців. Кожен виконавець виконує чисельне розв’язання однієї задачі, формує артефакт результату та публікує події прогресу або завершення. Postprocessor перетворює чисельний розв’язок у формат, придатний для браузерної візуалізації.

Компоненти контейнеризовано та розгорнуто в Kubernetes. Кількість реплік processor-worker може змінюватися незалежно від кількості реплік інших сервісів, що створює основу для вибіркового масштабування найбільш ресурсоємної стадії.

У четвертому розділі проведено експериментальне дослідження продуктивності, масштабованості, чисельної коректності та відтворюваності розробленої платформи. Досліджувалися двовимірний сценарій задачі Пуассона на одиничному квадраті та тривимірний сценарій на одиничному кубі з неоднорідними граничними умовами Діріхле. Для обох постановок використано метод виготовлених розв’язків.

Керованою змінною була кількість виконавців обчислювального компонента: $N_w = \{1,2,4\}$.

Для кожного сценарію досліджено два значення параметра сітки, три повтори кожної конфігурації та пакет із 30 незалежних задач. Загалом виконано 36 основних експериментальних серій, що охопили 1080 задач.

Встановлено, що ефект горизонтального масштабування залежить від співвідношення між часом корисного чисельного обчислення та накладними витратами розподіленого конвеєра. Для двовимірних задач перехід від одного до двох виконавців збільшив пропускну здатність приблизно на 8-12 % і зменшив 95-й перцентиль наскрізної затримки приблизно на 17-19 %. Подальше збільшення кількості виконавців до чотирьох не забезпечило стабільного приросту продуктивності.

Для тривимірної задачі з грубішою сіткою використання двох виконавців збільшило пропускну здатність на 15,97 %, тоді як чотири виконавці не дали додаткового стійкого покращення. Для складнішої тривимірної задачі з дрібнішою сіткою пропускна здатність зросла на 29,03 % за двох виконавців і на 35,21 % при чотирьох. Значення p95(E2E) зменшилося відповідно на 22,76 % і 26,12 %.

Водночас встановлено непропорційний характер масштабування. Зі збільшенням кількості паралельних виконавців скорочувався час очікування задач у черзі, але зростав внутрішній час виконання окремої складної FEM-задачі. Такий результат узгоджується з припущенням про конкуренцію компонентів за ресурси одного фізичного хоста та наявність незмінних стадій конвеєра.

Чисельну коректність перевірено за параметрами сітки, статистичними характеристиками вузлового поля, максимальною абсолютною та середньоквадратичною вузловими похибками. Для кожної фіксованої математичної постановки контрольні чисельні характеристики збігалися в усіх задачах, повторах і конфігураціях виконавців. Згущення сітки приводило до закономірного зменшення похибки. Отже, зміна кількості обчислювальних виконавців впливала на часові показники, але не впливала на контрольні характеристики математичного результату.

Наукова новизна отриманих результатів полягає в такому.

Вперше розроблено архітектурну модель подієво-орієнтованої мікросервісної FEA-платформи, у якій повний обчислювальний цикл логічно поділено на окремі мікросервіси, відокремлено канали передавання керівних подій, службових та великих обчислювальних даних, а також визначено правила й формати взаємодії між компонентами системи, що дає змогу незалежно змінювати обчислювальну потужність окремих стадій без перебудови решти платформи.

Вперше для FEA-платформ експериментально встановлено, що розмежування функцій координації подій і виконання ресурсоємних чисельних задач між незалежними компонентами забезпечує відтворюваність чисельних результатів при зміні кількості обчислювальних виконавців, що дає змогу розглядати горизонтальне масштабування як математично нейтральну операцію щодо точності розв’язку.

Отримали подальший розвиток методи оцінювання масштабованості розподілених обчислювальних платформ шляхом включення до складу показників пропускної здатності, наскрізної затримки та її статистичних параметрів, тривалостей окремих стадій конвеєра і часу роботи розв’язувача, що дає змогу комплексно характеризувати ефективність горизонтального масштабування платформи.

Уточнено закономірності впливу обчислювальної складності скінченно-елементної задачі на ефективність горизонтального масштабування розподіленої FEA-платформи: встановлено непропорційний характер масштабування та визначено частку часу чисельного розв’язання в повній наскрізній затримці як визначальний критерій вибору кількості обчислювальних виконавців, що дає змогу обґрунтовано планувати ресурси платформи та уникати надлишкового виділення обчислювальних потужностей.

Теоретичне значення отриманих результатів полягає в розвитку наукових засад побудови та оцінювання розподілених платформ скінченно-елементного аналізу. Розроблена архітектурна модель визначає принципи логічного поділу

повного FEA-циклу між окремими мікросервісами, відокремлення каналів передавання керівних подій, службових і великих обчислювальних даних, а також правила й формати взаємодії між компонентами платформи. Експериментально встановлено, що розмежування функцій координації подій і виконання ресурсоємних чисельних задач забезпечує відтворюваність чисельних результатів при зміні кількості обчислювальних виконавців, що обґрунтовує математичну нейтральність горизонтального масштабування щодо точності розв’язку. Подальшого розвитку набули методи оцінювання масштабованості розподілених обчислювальних платформ шляхом комплексного врахування пропускної здатності, наскрізної затримки та її статистичних параметрів, тривалості окремих стадій конвеєра і часу роботи розв’язувача. Уточнено залежність ефективності горизонтального масштабування від обчислювальної складності FEM-задачі та визначено частку часу чисельного розв’язання в повній наскрізній затримці як критерій вибору кількості обчислювальних виконавців.

Практичне значення отриманих результатів полягає у створенні працездатного прототипу подієво-орієнтованої мікросервісної FEA-платформи, у якому реалізовано повний цикл оброблення задачі: формування сітки, чисельне розв’язання, післяоброблення, зберігання даних, відстеження стану та браузерну тривимірну візуалізацію. Запропонована архітектура дає змогу незалежно змінювати обчислювальну потужність окремих стадій без перебудови інших компонентів платформи. Розмежування координації подій і виконання чисельних задач забезпечує керовану зміну кількості обчислювальних виконавців зі збереженням відтворюваності результатів. Розроблені методи оцінювання можуть бути використані для комплексного визначення ефективності горизонтального масштабування подібних розподілених платформ. Встановлена залежність ефективності масштабування від складності FEM-задачі та визначений критерій вибору кількості виконавців дають змогу обґрунтовано планувати ресурси платформи й уникати надлишкового виділення обчислювальних потужностей. Визначені правила та формати взаємодії між компонентами можуть бути використані під час розширення платформи новими

математичними постановками, чисельними розв’язувачами та засобами візуалізації.

Основні результати дослідження відображено у трьох наукових статтях і матеріалах наукової конференції.

Ключові слова: метод скінченних елементів, скінченно-елементний аналіз, FEA-платформа, чисельні методи, мікросервісна архітектура, подієво-орієнтована архітектура, розподілені системи, декомпозиція, паралельні обчислення, горизонтальне масштабування, брокер повідомлень, Kubernetes, пропускна здатність, відтворюваність, метод виготовлених розв’язків.

ABSTRACT

Kryvyi Ya. V. Event-Driven Microservice Architecture of a Scalable Finite Element Analysis Platform. – Qualification scientific work in the form of a manuscript.

Dissertation for the degree of Doctor of Philosophy in specialty 122 «Computer Science». – Zaporizhzhia National University, Zaporizhzhia, 2026.

The dissertation addresses the relevant scientific and applied problem of organising a complete finite element analysis (FEA) cycle in a distributed software environment while enabling independent horizontal scaling of the most resource-intensive computational stage and preserving the reference numerical characteristics of the results. An event-driven microservice architecture for an FEA platform is proposed, in which data preparation, numerical solution, post-processing, and result visualisation are represented as logically separated components with defined interaction contracts, independent deployment life cycles, and different computational resource utilisation profiles.

The relevance of the research is determined by the increasing complexity of engineering and scientific problems solved using the finite element method, the growing volumes of geometric, mesh, and numerical data, and the need to execute a large number of independent computational tasks in parallel. Traditional monolithic FEA systems combine geometry preparation, mesh generation, system assembly and solution, post-processing, and visualisation within a single software environment. This approach simplifies local interaction between components but complicates independent scaling of individual stages, integration with cloud infrastructure, asynchronous execution of long-running tasks, and timely notification of users about the current state of computations.

The dissertation substantiates that finite element analysis platforms benefit from combining microservice decomposition, event-driven interaction, specialised storage of large computational artefacts, and container orchestration. This approach separates task life-cycle management from the transfer and storage of large data volumes and

enables the number of computational workers to be changed without scaling all platform components.

The object of the research is the processes of designing and operating distributed software platforms for finite element analysis.

The subject of the research is the models, methods, and architectural solutions for the event-driven microservice organisation of an FEA workflow, mechanisms for independently scaling its computational stage, and methods for experimentally evaluating performance, numerical correctness, and reproducibility of results.

The purpose of the research is to improve the scalability and manageability of a finite element analysis software platform by developing an event-driven microservice architecture, implementing a prototype of the complete FEA workflow, and experimentally determining the influence of the number of computational workers and problem complexity on platform performance.

To achieve this purpose, methods of system and comparative analysis of software architectures, software system modelling methods using UML and IDEF0 notations, principles of microservice and event-driven architectures, methods for designing application programming interface and message contracts, and containerisation and orchestration tools were used. The finite element method, linear P1 approximations on triangular and tetrahedral meshes, sparse matrix representations, and the method of manufactured solutions were applied to implement and verify the numerical component. Experimental results were processed using descriptive statistical methods, including analysis of mean values, the median, the 95th percentile, sample standard deviation, relative speedup, and scaling efficiency.

The first chapter analyses the theoretical and applied prerequisites for applying microservice architecture to finite element analysis systems. The essence of the finite element method and the complete FEA cycle, which includes geometry preparation, mesh generation, specification of the mathematical model, numerical solution, post-processing, and visualisation, is considered. The characteristics of monolithic and microservice architectures, their advantages, limitations, and conditions for appropriate application are analysed.

Existing commercial and open-source FEA solutions are compared. It is established that most of them provide advanced numerical modelling capabilities; however, the issues of independent scaling of individual stages, open event-driven coordination, separation of large computational artefacts from control messages, and reproducible experimental evaluation of scalability remain insufficiently addressed. The prerequisites for transitioning to service decomposition and asynchronous interaction in FEA platforms are formulated.

The second chapter develops a conceptual model of an event-driven microservice FEA platform. Functional and non-functional system requirements are formulated, the roles of the user and system components are defined, and use-case, component, class, sequence, activity, and deployment diagrams are developed. The functional structure is additionally formalised using IDEF0.

The complete FEA cycle is divided among the Preprocessor, Processor, Postprocessor, Aggregator, and three-dimensional visualisation module. The client and server layers, component responsibilities, external entry point, model for transmitting states and events, and principles for storing metadata and computational artefacts are defined.

The separation of channels for transmitting control events, service data on task states, and large computational data is substantiated. The control layer consists of events, statuses, timestamps, and references to artefacts. Large data, including finite element meshes, problem specifications, numerical solutions, and visualisation packages, are stored in object storage and transferred between stages through stable references. This reduces the load on the message broker and decreases coupling between application components.

The third chapter implements an experimental prototype of the proposed architecture. The client web application is developed using Angular and Three.js. Traefik serves as the single external entry point, authentication is organised using Keycloak and OpenID Connect, while task creation, state aggregation, and timely delivery of task state changes are implemented in a Node.js BFF using Socket.IO.

The event-driven layer is implemented using Apache Kafka. A unified message structure is defined that contains the event type, task identifier, timestamp, contract

version, and application data. Task states and stage timing characteristics are stored in PostgreSQL, whereas large intermediate and final computational data are stored in MinIO. Redis and Celery are used to organise the internal queue of computational tasks.

The Preprocessor is implemented as a separate Python service that generates two-dimensional and three-dimensional meshes using Gmsh and creates a description of the mathematical problem. The Processor is divided into an event consumer and a set of independently scalable Celery workers. Each worker performs the numerical solution of one task, creates a result artefact, and publishes progress or completion events. The Postprocessor converts the numerical solution into a format suitable for browser-based visualisation.

The components are containerised and deployed in Kubernetes. The number of processor-worker replicas can be changed independently of the number of replicas of other services, thereby providing a basis for selectively scaling the most resource-intensive stage.

The fourth chapter presents an experimental investigation of the performance, scalability, numerical correctness, and reproducibility of the developed platform. A two-dimensional Poisson problem on the unit square and a three-dimensional problem on the unit cube with non-homogeneous Dirichlet boundary conditions are investigated. The method of manufactured solutions is applied to both problem formulations.

The controlled variable is the number of workers of the computational component: $N_w = \{1, 2, 4\}$.

For each scenario, two mesh-size values, three repetitions of each configuration, and a batch of 30 independent tasks are investigated. In total, 36 experimental series comprising 1,080 tasks are completed.

It is established that the effect of horizontal scaling depends on the relationship between useful numerical computation time and the overhead of the distributed workflow. For the two-dimensional problems, increasing the number of workers from one to two increased throughput by approximately 8–12% and reduced the 95th

percentile of end-to-end latency by approximately 17–19%. Increasing the number of workers to four did not provide a stable additional performance improvement.

For the three-dimensional problem with a coarser mesh, using two workers increased throughput by 15.97%, whereas four workers did not provide a further sustainable improvement. For the more complex three-dimensional problem with a finer mesh, throughput increased by 29.03% with two workers and by 35.21% with four workers. The p95 end-to-end latency decreased by 22.76% and 26.12%, respectively.

At the same time, the scaling behaviour was found to be non-proportional. As the number of parallel workers increased, the time tasks spent waiting in the queue decreased, whereas the internal execution time of an individual complex FEM task increased. This result is consistent with the assumption that the components competed for the resources of a single physical host and that several stages of the workflow remained unchanged.

Numerical correctness was evaluated using mesh parameters, statistical characteristics of the nodal field, the maximum absolute nodal error, and the root mean square nodal error. For each fixed mathematical problem, the reference numerical characteristics were identical across all tasks, repetitions, and worker configurations. Mesh refinement resulted in a consistent reduction in numerical error. Therefore, changing the number of computational workers affected the timing characteristics but did not affect the reference characteristics of the mathematical result.

The scientific novelty of the obtained results is as follows.

For the first time, an architectural model of an event-driven microservice FEA platform has been developed in which the complete computational cycle is logically divided into separate microservices, channels for transmitting control events, service data, and large computational data are separated, and the rules and formats of interaction between system components are defined. This makes it possible to independently change the computational capacity of individual stages without restructuring the rest of the platform.

For the first time for FEA platforms, it has been experimentally established that separating the functions of event coordination and resource-intensive numerical task

execution between independent components ensures the reproducibility of numerical results when the number of computational workers is changed. This makes it possible to regard horizontal scaling as a mathematically neutral operation with respect to solution accuracy.

Methods for evaluating the scalability of distributed computing platforms have been further developed by extending the set of indicators to include throughput, end-to-end latency and its statistical characteristics, the durations of individual workflow stages, and solver execution time. This makes it possible to comprehensively characterise the efficiency of horizontal platform scaling.

The patterns governing the influence of finite element problem complexity on the horizontal scaling efficiency of the distributed FEA platform have been refined: the scaling behaviour was found to be non-proportional, and the proportion of numerical solution time within the total end-to-end latency was identified as the determining criterion for selecting the number of computational workers. This makes it possible to plan platform resources in a substantiated manner and avoid excessive allocation of computational capacity.

The theoretical significance of the obtained results lies in the development of the scientific foundations for designing and evaluating distributed finite element analysis platforms. The developed architectural model defines the principles of the logical division of the complete FEA cycle among individual microservices, the separation of channels for transmitting control events, service data, and large computational data, as well as the rules and formats of interaction between platform components. It has been experimentally established that separating event coordination from the execution of resource-intensive numerical tasks ensures the reproducibility of numerical results when the number of computational workers is changed, thereby substantiating the mathematical neutrality of horizontal scaling with respect to solution accuracy. Methods for evaluating the scalability of distributed computing platforms have been further developed through the integrated consideration of throughput, end-to-end latency and its statistical parameters, the duration of individual workflow stages, and solver execution time. The patterns governing the dependence of horizontal scaling

efficiency on FEM problem complexity have been refined, and the proportion of numerical solution time within the total end-to-end latency has been identified as a criterion for selecting the number of computational workers.

The practical significance of the obtained results lies in the development of a functioning prototype of an event-driven microservice FEA platform that implements the complete task-processing cycle, including mesh generation, numerical solution, post-processing, data storage, task state tracking, and browser-based three-dimensional visualisation. The proposed architecture makes it possible to independently change the computational capacity of individual stages without restructuring the remaining platform components. The separation of event coordination from numerical task execution enables controlled changes in the number of computational workers while preserving the reproducibility of results. The developed evaluation methods can be used to comprehensively determine the efficiency of horizontal scaling in similar distributed platforms. The established dependence of scaling efficiency on FEM problem complexity and the defined criterion for selecting the number of workers make it possible to plan platform resources in a substantiated manner and avoid excessive allocation of computational capacity. The defined rules and formats of interaction between components can be used when extending the platform with new mathematical problem formulations, numerical solvers, and visualisation tools.

The main research results are presented in three scientific articles and one conference paper.

Keywords: finite element method, finite element analysis, FEA platform, numerical methods, microservice architecture, event-driven architecture, distributed systems, decomposition, parallel computing, horizontal scaling, message broker, Kubernetes, throughput, reproducibility, method of manufactured solutions.

ПЕРЕЛІК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковано основні наукові результати дисертації

1. Кривий Я. В., Лісняк А. О. Мікросервісна архітектура систем скінченно-елементного аналізу. *Computer Science and Applied Mathematics*. 2024. № 1. С. 75-83. DOI: 10.26661/2786-6254-2024-1-09.

2. Kryvyi Y., Lisnyak A. Event-Driven Microservice Architecture for Scalable Finite Element Analysis Platforms. *Computer Design Systems. Theory and Practice*. 2026. Vol. 8, No. 1. P. 56-69. DOI: 10.23939/cds2026.01.056.

3. Кривий Я. В., Лісняк А. О. Подієво-орієнтована мікросервісна архітектура масштабованої платформи скінченно-елементного аналізу: розробка та експериментальна оцінка. *Computer Science and Applied Mathematics*. 2026. № 1. С. 120-141. DOI: 10.26661/2786-6254-2026-1-13.

Наукові праці, які засвідчують апробацію матеріалів дисертації

4. Кривий Я. В., Лісняк А. О. Експериментальна оцінка масштабування подієво-орієнтованого FEA-конвеєра у 3D сценарії. *Актуальні проблеми математики та інформатики :* збірка тез доповідей Сімнадцятої Всеукраїнської, двадцять четвертої регіональної наукової конференції молодих дослідників (Запоріжжя, 23-24 квітня 2026 р.). Запоріжжя : Запорізький національний університет, 2026. С. 50-54. URL : https://www.znu.edu.ua/faculty/math/confirences/aktual__n___problemi_matematiki_ta___nformatiki_-_2026__tezi_konferents__yi.pdf (дата звернення: 25.06.2026).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

2D	Two-Dimensional, двовимірний
3D	Three-Dimensional, тривимірний
API	Application Programming Interface, інтерфейс програмування застосунків
BFF	Backend for Frontend, серверний компонент, адаптований до потреб клієнтського застосунку
CAD	Computer-Aided Design, автоматизоване проєктування
COO	Coordinate List, координатний формат подання розрідженої матриці
CPU	Central Processing Unit, центральний процесор
CSR	Compressed Sparse Row, формат розрідженої матриці зі стисненням за рядками
CSV	Comma-Separated Values, формат табличних даних зі значеннями, розділеними комами
E2E	End-to-End, наскрізний час або наскрізна затримка обробки задачі
EDA	Event-Driven Architecture, подієво-орієнтована архітектура
FEA	Finite Element Analysis, скінченно-елементний аналіз
FEM	Finite Element Method, метод скінченних елементів
GPU	Graphics Processing Unit, графічний процесор
HTTP	Hypertext Transfer Protocol, протокол передавання гіпертексту
HTTPS	Hypertext Transfer Protocol Secure, захищений протокол передавання гіпертексту
IDEF0	Integration Definition for Function Modeling, нотація функціонального моделювання систем
IdP	Identity Provider, провайдер ідентичності
JSON	JavaScript Object Notation, текстовий формат подання структурованих даних
JWT	JSON Web Token, вебмаркер у форматі JSON для передавання тверджень про автентифікованого користувача

JWKS	JSON Web Key Set, набір відкритих вебключів у форматі JSON
MMS	Method of Manufactured Solutions, метод виготовлених розв’язків
OAuth 2.0	Open Authorization 2.0, протокольний каркас авторизації
OIDC	OpenID Connect, протокол автентифікації, побудований над OAuth2.0
P1	лінійна скінченно-елементна апроксимація першого порядку
PDE	Partial Differential Equation, диференціальне рівняння в частинних похідних
PKCE	Proof Key for Code Exchange, механізм захисту обміну кодом авторизації
PVC	PersistentVolumeClaim, запит на виділення постійного тому зберігання в Kubernetes
RAM	Random Access Memory, оперативна пам’ять
REST	Representational State Transfer, архітектурний стиль побудови мережевих програмних інтерфейсів
RMS	Root Mean Square, середньоквадратичне значення
S3	Simple Storage Service, інтерфейс об’єктного зберігання даних
SDK	Software Development Kit, набір засобів розроблення програмного забезпечення
SQL	Structured Query Language, мова структурованих запитів
UI	User Interface, інтерфейс користувача
ULID	Universally Unique Lexicographically Sortable Identifier, універсальний унікальний лексикографічно впорядковуваний ідентифікатор
UML	Unified Modeling Language, уніфікована мова моделювання
URL	Uniform Resource Locator, уніфікований покажчик ресурсу
WebGL	Web Graphics Library, програмний інтерфейс для відтворення інтерактивної графіки у веббраузері
p95	95-й перцентиль вибірки

ВСТУП

Актуальність. Метод скінченних елементів є одним із основних чисельних методів розв’язання крайових задач математичної фізики та широко застосовується в механіці деформівного твердого тіла, теплопровідності, гідродинаміці, електромагнетизмі та інших галузях інженерного й наукового моделювання [1–5]. Його використання дозволяє досліджувати поведінку складних об’єктів і фізичних процесів шляхом дискретизації розрахункової області та зведення неперервної математичної задачі до системи алгебраїчних рівнянь.

Скінченно-елементний аналіз охоплює не лише безпосереднє чисельне розв’язання, а повний обчислювальний цикл, що включає підготовку геометрії, формування скінченно-елементної сітки, задання фізико-математичної моделі, формування й розв’язання системи рівнянь, післяобробку та візуалізацію отриманих результатів. Кожна з цих стадій має власні алгоритми, програмні залежності, формати даних і вимоги до обчислювальних ресурсів.

Зростання складності математичних моделей, деталізації геометрії та розмірності скінченно-елементних сіток приводить до збільшення тривалості обчислень і обсягів проміжних даних. Додатковим викликом є необхідність одночасного виконання серій незалежних задач, наприклад під час параметричних досліджень, оптимізації, верифікації чисельних моделей або проведення обчислювальних експериментів.

Традиційні програмні системи скінченно-елементного аналізу часто будуються як інтегровані монолітні застосунки, у межах яких підготовка даних, чисельне розв’язання, післяоброблення та візуалізація функціонують як частини єдиного програмного комплексу. Така організація спрощує локальну взаємодію компонентів і є доцільною для настільного використання, проте створює обмеження під час переходу до хмарних і розподілених середовищ.

Монолітна організація ускладнює незалежне масштабування найбільш навантажених компонентів, оскільки збільшення обчислювальної потужності

зазвичай потребує масштабування всього застосунку. Вона також обмежує можливість незалежного оновлення окремих алгоритмів, інтеграції різних технологічних стеків, локалізації відмов і організації асинхронного виконання довготривалих задач.

Альтернативою є застосування мікросервісної архітектури, у якій функціональність системи розподіляється між автономними компонентами з чітко визначеними межами відповідальності та контрактами взаємодії [6–10]. Для Finite Element Analysis (FEA) платформи такий підхід дозволяє окремо реалізувати стадії підготовки даних, чисельного розв’язання, післяобробки, керування станом задачі та візуалізації.

Водночас механічне розділення монолітної системи на сервіси не гарантує підвищення продуктивності.

Розподілена архітектура створює додаткові накладні витрати, пов’язані з мережевою взаємодією, серіалізацією даних, координацією станів, передаванням артефактів, роботою брокера повідомлень і підтримкою інфраструктурних компонентів. Тому архітектура FEA-платформи має враховувати специфіку чисельного конвеєра та співвідношення між тривалістю корисного обчислення й витратами розподіленого середовища [11–14].

Особливе значення має організація обміну даними. Геометричні моделі, сітки, параметри постановки, чисельні розв’язки та пакети візуалізації можуть мати значний обсяг. Передавання таких даних безпосередньо через брокер повідомлень підвищує навантаження на подієвий контур, ускладнює повторне опрацювання повідомлень і посилює залежність між компонентами.

Перспективним рішенням є розділення керівного й обчислювального потоків. Події повинні передавати ідентифікатори задач, статуси, часові мітки та посилання на артефакти, тоді як великі дані доцільно зберігати в спеціалізованому об’єктному сховищі. Такий підхід зменшує обсяг повідомлень, забезпечує відтворюваність обчислень і спрощує повторне використання результатів окремих стадій.

Подієво-орієнтована взаємодія також дозволяє відокремити момент створення задачі від її фактичного виконання. Користувач отримує ідентифікатор

задачі, після чого стадії конвеєра виконуються асинхронно, а зміни стану передаються у вигляді подій. Це особливо важливо для довготривалих чисельних обчислень, які не повинні виконуватися в межах одного синхронного HTTP-запиту.

Окремою проблемою є масштабування обчислювальної стадії. Підготовка даних, чисельне розв'язання та післяоброблення мають різні профілі навантаження. Найбільша потреба в процесорному часі зазвичай виникає під час складання та розв'язання системи алгебраїчних рівнянь. Масштабування всіх компонентів платформи одночасно в такому разі може призводити до нераціонального використання ресурсів.

Тому актуальним є формування архітектури, у якій обчислювальний компонент може масштабуватися незалежно. Практична доцільність такого рішення має підтверджуватися не лише технічною можливістю створення додаткових реплік, а й кількісним оцінюванням пропускної здатності, наскрізної затримки, часу виконання окремих стадій та ефективності масштабування.

Під час оцінювання розподіленої FEA-платформи також необхідно контролювати чисельну коректність. Зміна кількості виконавців, порядку надходження задач і рівня паралельності не повинна впливати на сформовану сітку, математичну постановку та контрольні характеристики розв'язку. Тому дослідження продуктивності доцільно поєднувати з перевіркою відтворюваності результатів і зменшення чисельної похибки при згущенні сітки.

Аналіз існуючих комерційних і відкритих рішень показує, що сучасні FEA-системи забезпечують широкий набір чисельних і візуалізаційних можливостей. Водночас питання відкритої подієво-орієнтованої декомпозиції повного FEA-циклу, розділення подій, метаданих і великих артефактів, незалежного масштабування обчислювальної стадії та спільного оцінювання продуктивності й чисельної коректності потребують подальшого дослідження.

У межах цієї роботи розглядається веборієнтована програмна платформа скінченно-елементного аналізу, доступ до якої здійснюється через браузерний клієнт, а підготовка даних, постановка задач у чергу, чисельне розв'язання,

післяоброблення та зберігання результатів виконуються серверними компонентами розподіленого середовища. Дослідження не спрямоване на заміну настільних CAE/FEA-комплексів або оцінювання їхніх локальних чисельних ядер; предметом архітектурного аналізу є організація повного FEA-циклу як вебзастосунку з асинхронним виконанням довготривалих задач, централізованим доступом до станів і артефактів та можливістю незалежного масштабування обчислювальної стадії.

Проблеми побудови складних програмних систем, вибору архітектурного стилю та забезпечення необхідних властивостей якості досліджено у працях Л. Басса, П. Клементса і Р. Казмана [6]. Розвиток мікросервісного підходу висвітлено у працях С. Ньюмена, Дж. Льюїса, М. Фаулера, П. Джамшіді, К. Паля та інших дослідників [7–9]. У цих роботах мікросервісна архітектура розглядається як підхід до побудови програмних систем з автономних компонентів, що мають визначені межі відповідальності, взаємодіють через стабільні контракти та можуть незалежно розгортатися і масштабуватися. Водночас дослідники відзначають, що такий розподіл ускладнює міжсервісну взаємодію, керування розподіленими даними, спостережуваність і забезпечення надійності системи. Експериментальне порівняння монолітної та мікросервісної архітектур, виконане Г. Бліновським, А. Ойдовською та А. Пшибилеком, підтверджує необхідність оцінювання продуктивності й масштабованості з урахуванням структури навантаження та накладних витрат розподіленої взаємодії, а не лише самого факту структурної декомпозиції системи [10].

Питання організації обміну даними та координації компонентів у розподілених системах розглянуто в працях М. Клеппмана, Г. Хопе, Б. Вулфа, В. Фогелса, Дж. Крепса, Н. Нархеде та Дж. Рао [11–14]. У цих дослідженнях систематизовано принципи побудови надійних і масштабованих систем опрацювання даних, патерни асинхронного обміну повідомленнями, маршрутизації та взаємодії за моделлю «видавець – підписник», підходи до узгодження станів у розподіленому середовищі й засади використання розподілених журналів подій для масштабованої асинхронної взаємодії

компонентів. Такі підходи формують теоретичну основу для побудови систем, у яких довготривалі стадії виконуються незалежно, а їх координація здійснюється через повідомлення про зміну стану.

Практична реалізація розподілених платформ такого класу спирається на засоби контейнеризації та оркестрації, що забезпечують ізоляцію компонентів, відтворюване розгортання та керування кількістю обчислювальних реплік [15, 16].

У сучасних дослідженнях, дотичних до предмета роботи, розглядаються веборієнтовані засоби після оброблення, паралельні алгоритми скінченно-елементних обчислень, чисельне моделювання контактних і нелінійних задач, архітектура програмних бібліотек FEM, мультиплатформні застосунки, моделювання паралельних обчислювальних систем і альтернативні нейромережеві методи розв’язання задач математичної фізики [17–30].

Вагомий внесок у розвиток чисельних методів і програмних засобів скінченно-елементного аналізу зроблено у працях С. І. Гоменюка, С. М. Гребенюка, С. В. Чопорова, Ю. О. Бєлоконя, О. В. Кудіна, Ю. Г. Козуба та І. О. Астіоненко. У цих дослідженнях розглянуто побудову скінченно-елементних матриць і базисів, чисельне моделювання контактних та геометрично нелінійних задач, архітектуру програмного забезпечення для скінченно-елементного моделювання, а також паралельну організацію окремих етапів скінченно-елементних обчислень [18–21, 23, 27–30].

Окремий напрям становлять роботи, присвячені веборієнтованим засобам післяоброблення, мультиплатформним клієнтським застосункам, моделюванню паралельних обчислювальних систем і альтернативним нейромережевим методам розв’язання задач математичної фізики. Відповідні результати представлено у працях С. І. Гоменюка, С. М. Гребенюка, Ю. Г. Козуба та О. В. Кудіна [17, 22, 24–26].

Отже, у науковій літературі достатньо ґрунтовно досліджено загальні принципи програмної архітектури, мікросервісної декомпозиції, асинхронного обміну повідомленнями, подієвої координації та функціонування розподілених

систем. Паралельно розвиваються чисельні алгоритми FEM, програмні бібліотеки, паралельні обчислення, веборієнтовані засоби післяоброблення та альтернативні методи розв’язання задач математичної фізики. Разом із тим у розглянутих працях окремо не розв’язується задача побудови цілісної веборієнтованої FEA-платформи, у якій повний цикл – від підготовки даних до браузерної візуалізації – координується подієво, великі артефакти відокремлено від керівних повідомлень, а обчислювальна стадія масштабується незалежно зі спільною перевіркою продуктивності та чисельної відтворюваності.

Отже, **актуальним** науково-прикладним завданням є розробка подієво-орієнтованої мікросервісної архітектури платформи скінченно-елементного аналізу, яка забезпечує асинхронне проходження повного обчислювального циклу, кероване зберігання великих артефактів, незалежне горизонтальне масштабування чисельного компонента та збереження контрольних характеристик математичного результату.

Зв’язок роботи з науковими програмами, планами, темами. Дисертаційну роботу виконано відповідно до основних напрямів наукових досліджень кафедри програмної інженерії Запорізького національного університету в галузі архітектури програмного забезпечення, розподілених інформаційних систем, хмарних обчислень і методів чисельного моделювання.

Мета дослідження – підвищення масштабованості й керованості програмної платформи скінченно-елементного аналізу шляхом розроблення подієво-орієнтованої мікросервісної архітектури, реалізації прототипу повного FEA-конвеєра та експериментального визначення впливу кількості обчислювальних виконавців і складності задачі на продуктивність платформи.

Завдання дослідження. Для досягнення поставленої мети необхідно розв’язати такі взаємопов’язані завдання:

1) провести аналіз монолітного та мікросервісного підходів до побудови програмних систем, дослідити існуючі комерційні, хмарні й відкриті FEA-рішення та визначити невирішені архітектурні проблеми створення масштабованих платформ скінченно-елементного аналізу;

2) сформулювати функціональні й нефункціональні вимоги та розробити концептуальну подієво-орієнтовану мікросервісну модель повного FEA-конвеєра з визначенням складу компонентів, меж їхньої відповідальності, контрактів взаємодії та способу розділення керівних подій, транзакційних метаданих і великих обчислювальних артефактів;

3) розробити й реалізувати експериментальний прототип FEA-платформи та її обчислювальний контур за моделлю «споживач подій – незалежно масштабовані виконавці», що забезпечує зміну кількості реплік processor-worker без масштабування інших компонентів і зміни зовнішніх контрактів;

4) реалізувати двовимірну та тривимірну тестові задачі Пуассона із застосуванням методу скінченних елементів і методу виготовлених розв’язків для контролю чисельної коректності платформи;

5) розробити методику експериментального оцінювання продуктивності та масштабованості платформи з урахуванням пропускної здатності, наскрізної затримки, тривалості окремих стадій, внутрішнього часу чисельного розв’язувача, прискорення, ефективності масштабування та контрольних чисельних характеристик;

6) провести серії обчислювальних експериментів при різній кількості обчислювальних виконавців і різній складності скінченно-елементних задач, кількісно оцінити ефект горизонтального масштабування, встановити його залежність від обчислювального навантаження та розробити рекомендації щодо вибору кількості виконавців;

7) перевірити збереження контрольних чисельних характеристик при зміні кількості виконавців, оцінити зменшення похибки при згущенні сітки та визначити обмеження і загрози валідності експериментального дослідження.

Об’єкт дослідження – процеси побудови та функціонування розподілених програмних платформ скінченно-елементного аналізу.

Предмет дослідження – моделі, методи та архітектурні рішення подієво-орієнтованої мікросервісної організації FEA-конвеєра, механізми незалежного горизонтального масштабування його обчислювальної стадії, а також методи

експериментального оцінювання продуктивності, чисельної коректності та відтворюваності результатів.

Методи дослідження. Для розв’язання поставлених завдань використано комплекс взаємопов’язаних методів.

Методи системного аналізу застосовано для дослідження повного життєвого циклу FEA-задачі, визначення складу компонентів, інформаційних потоків і архітектурних обмежень.

Порівняльний аналіз використано для зіставлення монолітної та мікросервісної архітектур, а також існуючих комерційних, хмарних і відкритих FEA-рішень.

Методи структурного й об’єктно-орієнтованого моделювання застосовано для формалізації функціональних процесів, взаємодії компонентів, класів і послідовностей виконання. Моделі подано в нотаціях IDEF0 та UML.

Методи проєктування програмних архітектур використано для сервісної декомпозиції FEA-конвеєра, формування REST API, подієвих схем, контрактів артефактів і моделі зберігання даних.

Принципи подієво-орієнтованої архітектури застосовано для асинхронної координації стадій, передавання станів задач і відокремлення моменту створення задачі від її фактичного виконання.

Методи контейнеризації та оркестрації використано для ізоляції компонентів, відтворюваного розгортання й керування кількістю обчислювальних реплік [15, 16].

Метод скінченних елементів застосовано для чисельного розв’язання двовимірної та тривимірної задач Пуассона. Використано лінійні P1-елементи на трикутних і тетраедральних сітках, розріджене подання глобальної матриці та чисельне розв’язання системи лінійних алгебраїчних рівнянь.

Метод виготовлених розв’язків використано для формування тестових правих частин і кількісного оцінювання похибки чисельних результатів.

Експериментальний метод застосовано для дослідження роботи платформи при одному, двох і чотирьох обчислювальних виконавцях та для різних рівнів складності сітки.

Методи описової статистики використано для обчислення середніх значень, медіани, 95-го перцентиля, вибіркового стандартного відхилення, відносного прискорення та ефективності масштабування.

Наукова новизна отриманих результатів. У дисертаційній роботі отримано такі наукові результати:

– вперше розроблено архітектурну модель подієво-орієнтованої мікросервісної FEA-платформи, у якій повний обчислювальний цикл логічно поділено на окремі мікросервіси, відокремлено канали передавання керівних подій, службових та великих обчислювальних даних, а також визначено правила й формати взаємодії між компонентами системи, що дає змогу незалежно змінювати обчислювальну потужність окремих стадій без перебудови решти платформи;

– вперше для FEA-платформ експериментально встановлено, що розмежування функцій координації подій і виконання ресурсоємних чисельних задач між незалежними компонентами забезпечує відтворюваність чисельних результатів при зміні кількості обчислювальних виконавців, що дає змогу розглядати горизонтальне масштабування як математично нейтральну операцію щодо точності розв’язку;

– отримали подальший розвиток методи оцінювання масштабованості розподілених обчислювальних платформ шляхом включення до складу показників пропускної здатності, наскрізної затримки та її статистичних параметрів, тривалостей окремих стадій конвеєра і часу роботи розв’язувача, що дає змогу комплексно характеризувати ефективність горизонтального масштабування платформи;

– уточнено закономірності впливу обчислювальної складності скінченно-елементної задачі на ефективність горизонтального масштабування розподіленої FEA-платформи: встановлено непропорційний характер масштабування і визначено частку часу чисельного розв’язання в повній наскрізній затримці як визначальний критерій вибору кількості

обчислювальних виконавців, що дає змогу обґрунтовано планувати ресурси платформи та уникати надлишкового виділення обчислювальних потужностей.

Теоретичне значення отриманих результатів полягає в розвитку наукових засад побудови та оцінювання розподілених платформ скінченно-елементного аналізу. Розроблена архітектурна модель визначає принципи логічного поділу повного FEA-циклу між окремими мікросервісами, відокремлення каналів передавання керівних подій, службових і великих обчислювальних даних, а також правила й формати взаємодії між компонентами платформи. Експериментально встановлено, що розмежування функцій координації подій і виконання ресурсоємних чисельних задач забезпечує відтворюваність чисельних результатів при зміні кількості обчислювальних виконавців, що обґрунтовує математичну нейтральність горизонтального масштабування щодо точності розв’язку. Подальшого розвитку набули методи оцінювання масштабованості розподілених обчислювальних платформ шляхом комплексного врахування пропускної здатності, наскрізної затримки та її статистичних параметрів, тривалості окремих стадій конвеєра і часу роботи розв’язувача. Уточнено закономірності впливу обчислювальної складності скінченно-елементної задачі на ефективність горизонтального масштабування розподіленої FEA-платформи та визначено частку часу чисельного розв’язання в повній наскрізній затримці як критерій вибору кількості обчислювальних виконавців.

Практичне значення отриманих результатів полягає у створенні працездатного прототипу подієво-орієнтованої мікросервісної FEA-платформи, у якому реалізовано повний цикл оброблення задачі: формування сітки, чисельне розв’язання, післяоброблення, зберігання даних, відстеження стану та браузерну тривимірну візуалізацію. Запропонована архітектура дає змогу незалежно змінювати обчислювальну потужність окремих стадій без перебудови інших компонентів платформи. Розмежування координації подій і виконання чисельних задач забезпечує керовану зміну кількості обчислювальних виконавців зі

збереженням відтворюваності результатів. Розроблені методи оцінювання можуть бути використані для комплексного визначення ефективності горизонтального масштабування подібних розподілених платформ. Встановлена залежність ефективності масштабування від складності FEM-задачі та визначений критерій вибору кількості виконавців дають змогу обґрунтовано планувати ресурси платформи й уникати надлишкового виділення обчислювальних потужностей. Визначені правила та формати взаємодії між компонентами можуть бути використані під час розширення платформи новими математичними постановками, чисельними розв’язувачами та засобами візуалізації

Апробація результатів дослідження. Основні положення та результати дисертаційної роботи доповідалися й обговорювалися на Сімнадцятій Всеукраїнській, двадцять четвертій регіональній науковій конференції молодих дослідників «Актуальні проблеми математики та інформатики» (Запоріжжя, 23-24 квітня 2026 р.).

На конференції представлено результати дослідження подієво-орієнтованого масштабування тривимірного FEA-конвеєра, методику зміни кількості обчислювальних виконавців та результати оцінювання пропускної здатності й p95 наскрізної затримки.

Публікації. За результатами дисертаційного дослідження опубліковано чотири наукові праці, серед яких три статті у наукових фахових виданнях та одні матеріали доповіді на науковій конференції.

У публікаціях висвітлено результати аналізу монолітної та мікросервісної архітектур FEA-систем, концептуальну модель подієво-орієнтованої платформи, її програмну реалізацію та результати експериментального оцінювання горизонтального масштабування.

Особистий внесок здобувача. Основні результати дисертаційної роботи отримано здобувачем особисто.

У працях, опублікованих у співавторстві, здобувачем проведено аналіз предметної області та існуючих FEA-рішень; сформульовано архітектурні

вимоги; розроблено концептуальну модель платформи, UML- та IDEF0-моделі; визначено подієві, API- та артефактні контракти; спроєктовано й реалізовано експериментальний програмний прототип; реалізовано двовимірний і тривимірний FEM-розв’язувач; розгорнуто компоненти у Kubernetes; розроблено методику експериментального дослідження; проведено обчислювальні експерименти та виконано статистичну обробку їх результатів.

Структура та обсяг дисертації. Дисертаційна робота складається з анотації українською й англійською мовами, переліку публікацій здобувача, переліку умовних скорочень, вступу, чотирьох розділів, загальних висновків, списку використаних джерел і шести додатків.

Основний зміст викладено на 180 сторінках друкованого тексту, проілюстровано 45 рисунками та 31 таблицею. Загальний обсяг дисертації становить 226 сторінок.

РОЗДІЛ 1. ТЕОРЕТИКО-АРХІТЕКТУРНІ ЗАСАДИ ПОБУДОВИ МАСШТАБОВАНИХ FEA-СИСТЕМ

1.1 Порівняльний аналіз монолітної та мікросервісної архітектур

Архітектура програмної системи визначає спосіб організації її компонентів, зв’язки між ними, принципи розгортання, масштабування, супроводу та подальшої модернізації [6]. Для систем, що працюють із великими обсягами даних, тривалими обчисленнями та складними сценаріями взаємодії користувача з результатами моделювання, архітектурний підхід має не лише інженерне, а й методологічне значення. Він впливає на здатність системи адаптуватися до збільшення навантаження, підтримувати нові типи задач, ізолювати відмови окремих компонентів і забезпечувати повторюваність обчислювального процесу.

У контексті FEA-платформ це питання є особливо важливим, оскільки такі системи поєднують декілька різнорідних видів діяльності: підготовку геометричної моделі, генерацію сітки, налаштування фізичної постановки задачі, чисельне розв’язання, післяобробку результатів і їх візуалізацію. Кожен з етапів зазнає різного навантаження: одні є більш залежними від процесора та оперативної пам’яті, інші – від операцій введення/виведення, збереження великих артефактів або реактивної взаємодії з користувачем. Тому вибір архітектури безпосередньо впливає на можливість ефективного масштабування FEA-системи [1, 2, 31].

Традиційним підходом до побудови складних прикладних програм є монолітна архітектура. У такій моделі функціональні частини системи реалізуються в межах єдиної кодової бази та, як правило, розгортаються як один застосунок. Моноліт може містити користувацький інтерфейс, бізнес-логіку, модулі обробки даних, доступ до бази даних та допоміжні компоненти в межах єдиного програмного комплексу. Такий підхід має переваги на початкових

етапах: спрощується розроблення першої версії, легше організувати локальне налагодження, зменшується кількість інфраструктурних компонентів, а взаємодія між модулями відбувається без мережевих накладних витрат [6–10].

На рисунку 1.1 наведено схему типової монолітної системи.

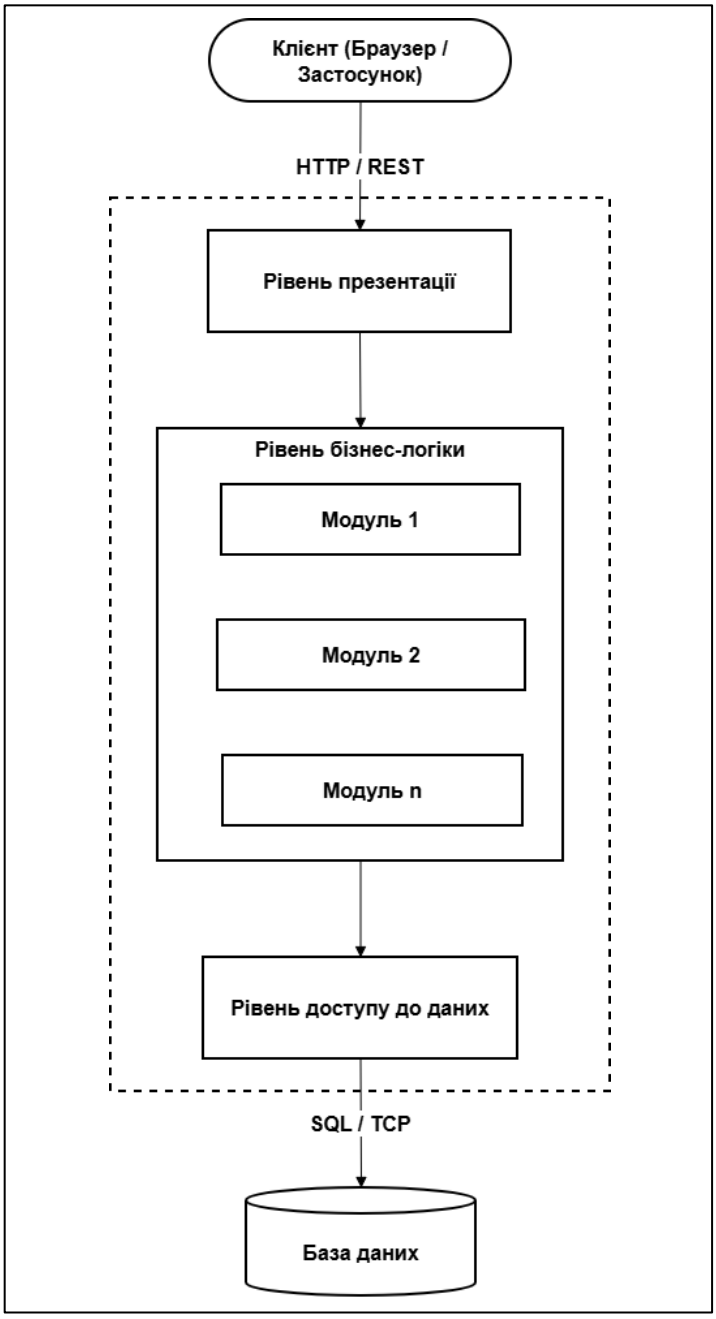


Рисунок 1.1 – Схема типової монолітної системи

Водночас із розростанням системи монолітна архітектура починає створювати обмеження. По-перше, будь-які зміни в окремому модулі можуть потребувати повторного збирання, тестування та розгортання всього застосунку.

По-друге, ускладнюється ізоляція відмов: помилка в одному функціональному блоці потенційно може вплинути на роботу всієї системи. По-третє, масштабування виконується переважно для застосунку в цілому, навіть якщо підвищене навантаження припадає лише на окрему його частину. Для FEA-систем це означає, що при зростанні навантаження на обчислювальний модуль доводиться масштабувати також ті частини, які не є вузьким місцем: інтерфейс, модулі керування задачами або засоби післяобробки [7–10, 31].

Мікросервісна архітектура виникла як відповідь на обмеження великих монолітних систем. Її сутність полягає у поділі програмного забезпечення на набір автономних сервісів, кожен з яких відповідає за окрему бізнес-функцію або технологічний етап і взаємодіє з іншими сервісами через визначені інтерфейси [7–9, 32–34].

З типовою схемою мікросервісної архітектури можна ознайомитися на рисунку 1.2.

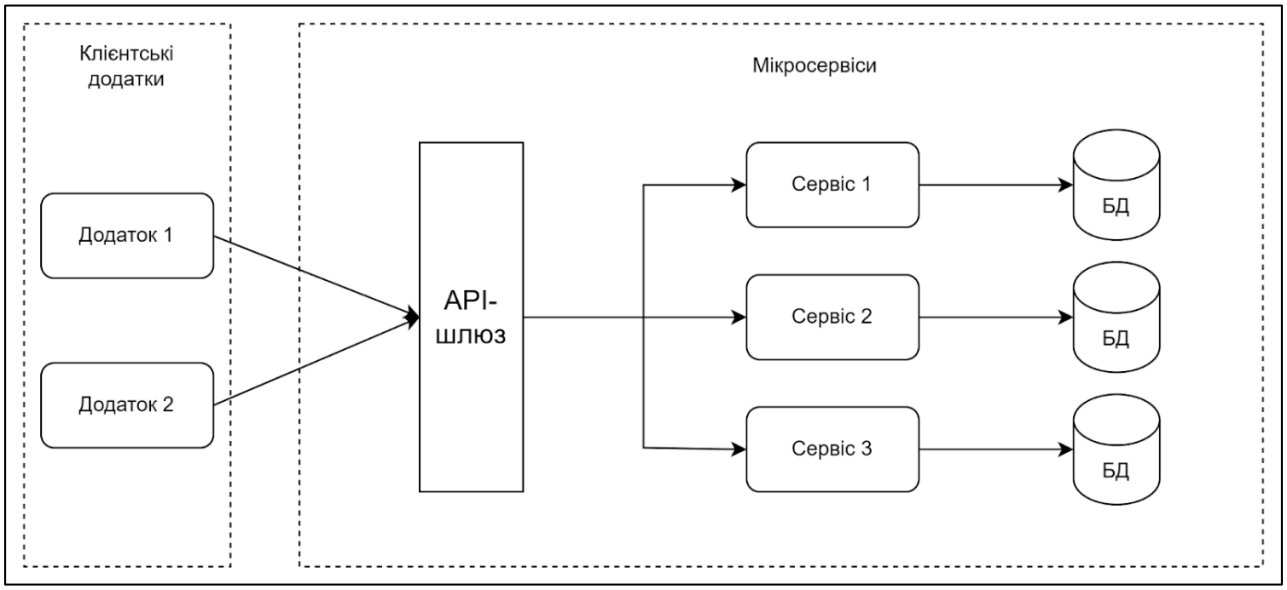


Рисунок 1.2 – Схема типової мікросервісної системи [31]

На відміну від моноліту, мікросервісний підхід дозволяє масштабувати окремі компоненти відповідно до їхнього фактичного навантаження: якщо один сервіс виконує ресурсомісткі обчислення, а інший лише приймає запити або агрегує дані, то кількість екземплярів збільшують саме для обчислювального

сервісу. Для FEA-платформи така властивість є принциповою, оскільки основне навантаження зазвичай концентрується на етапах генерації сітки, складання та розв’язання систем рівнянь, тоді як інші компоненти виконують координаційні функції [1, 2, 31].

Серед переваг мікросервісної архітектури виокремлюються незалежне розгортання, технологічна гнучкість, ізоляцію відмов, можливість горизонтального масштабування окремих сервісів, а також спрощення супроводу через чіткі межі відповідальності компонентів. Однак мікросервісний підхід не є універсальним: він додає операційну складність, потребує налагодження міжсервісної взаємодії, спостережуваності, управління контрактами API, контролю версій повідомлень і забезпечення узгодженості даних у розподіленому середовищі [11–14].

Для обґрунтування архітектурного рішення виконано порівняння монолітного та мікросервісного підходів за ключовими критеріями: одиниця розгортання, масштабування, вплив змін, характер взаємодії, узгодженість даних, відмовостійкість та операційна складність (табл. 1.1).

Таблиця 1.1 – Порівняння монолітної та мікросервісної архітектур

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Одиниця розгортання	Єдиний логічний виконуваний компонент, де зміни часто вимагають перевипуску / перерозгортання всього застосунку	Набір незалежно розгортовуваних сервісів, які працюють у власних процесах і взаємодіють через «легкі» механізми
Масштабування	Зазвичай виконується для всього застосунку. Можливе горизонтальне масштабування кількома екземплярами за балансувальником, але без вибірковості по модулях	Можна масштабувати окремий сервіс, а не весь застосунок
Вплив змін	Зміна частини системи тягне реліз усієї	Незалежне розгортання зменшує ймовірність того, що зміна одного компонента зірве реліз усього застосунку

Продовження таблиці 1.1

Критерій	Монолітна архітектура	Мікросервісна архітектура
Характер взаємодії	Переважно внутрішньопроцесні виклики (функції/класи), менші накладні витрати на комунікацію	Взаємодія через віддалені виклики та обмін повідомленнями
Узгодженість даних	Найчастіше використовується спільне сховище (наприклад, одна реляційна БД для всіх модулів), що полегшує узгодженість у межах транзакцій, але призводить до сильнішого зв’язування модулів через дані	Найпоширеніша модель – децентралізація даних і взаємодія через програмні інтерфейси та події
Відмовостійкість	Збої всередині єдиного процесу потенційно впливають на весь застосунок	Автономність сервісів полегшує локалізацію наслідків збоїв
Операційна складність	Простіший старт експлуатації за рахунок меншої кількості компонентів для підтримки	Необхідна зріла експлуатація, автоматизація, моніторинг і компетенції для керування кластером сервісів

Порівняння показує, що мікросервісна архітектура доцільна за умови непропорційного навантаження на окремі компоненти та вимог до незалежного розгортання, однак потребує зрілої інфраструктури та врахування складності розподілених систем [15, 16, 35].

1.2 Системи скінченно-елементного аналізу

Метод скінченних елементів є одним із базових чисельних методів, що використовується для розв’язання крайових задач математичної фізики, інженерної механіки, теплопровідності, гідродинаміки, електромагнетизму та інших прикладних задач. Його сутність полягає у поділі складної області на скінченну кількість простіших підобластей – елементів, у межах яких шукану

функцію апроксимують за допомогою базисних функцій. У результаті неперервна задача зводиться до дискретної алгебраїчної системи, яку можна розв’язувати чисельно [1–5, 36, 37].

Finite Element Analysis (FEA) є прикладним процесом використання методу скінченних елементів для аналізу поведінки об’єктів, конструкцій, матеріалів або фізичних процесів під дією заданих умов. У технічній та навчальній літературі FEA розглядається не лише як чисельний розв’язувач, а як повний інженерний цикл, що охоплює підготовку геометрії, побудову сітки, задання фізичної моделі, виконання обчислень, аналіз і візуалізацію результатів [1–5]. Отже, програмна FEA-система має забезпечувати повний ланцюг дій від постановки задачі до інтерпретації отриманого результату.

Типова FEA-система включає декілька функціональних частин. На рівні користувацької взаємодії вона містить інтерфейс для створення або імпорту моделі, задання параметрів задачі, запуску розрахунку та перегляду результатів. На рівні підготовки даних виконується побудова або імпорт геометрії, перевірка коректності, генерація сітки та формування вхідного опису задачі [38]. На рівні обчислень здійснюється формування матриць, урахування граничних умов, розв’язання системи рівнянь та отримання чисельного розв’язку [39]. На рівні післяобробки дані перетворюються у форму, придатну для аналізу, побудови графіків, полів значень і тривимірних сцен.

У спрощеному вигляді FEA-процес можна подати як послідовність трьох основних етапів: підготовка даних (pre-processing), чисельне розв’язання (solving) і післяоброблення (post-processing). Етап підготовки даних пов’язаний із підготовкою вхідних даних (геометрія, сітка, матеріали, граничні умови). Етап чисельного розв’язання охоплює чисельне розв’язання дискретизованої задачі. Етап післяоброблення забезпечує інтерпретацію та візуалізацію отриманих результатів. Такий поділ є важливим не лише з математичного, а й з архітектурного погляду, оскільки кожен етап має власний профіль навантаження, залежності та потенційно може розвиватися незалежно [1, 2, 17].

Рисунок 1.3 відображає узагальнений FEA-процес.

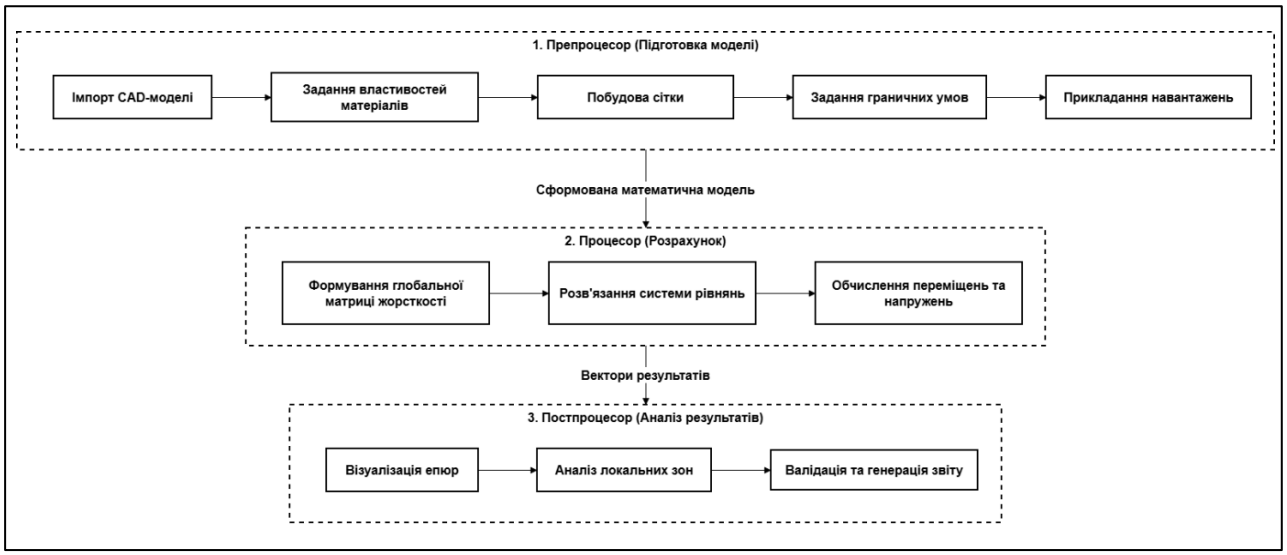


Рисунок 1.3 – Узагальнений FEA-процес

З архітектурної точки зору класичні FEA-системи часто будувалися як монолітні програмні комплекси. Це пояснюється історичною специфікою предметної області: тісною взаємодією геометричного ядра, генератора сітки, чисельного розв’язувача, бібліотек лінійної алгебри та модулів візуалізації. Така інтеграція спрощує роботу з єдиними структурами даних і зменшує витрати на передавання інформації між компонентами. Однак у сучасних умовах, коли зростають обсяги моделей, виникає потреба в хмарному виконанні, паралельній обробці серій задач і віддаленій інтерактивній візуалізації, монолітна модель поступово виявляє обмеження.

Практичний аналіз сучасних FEA-рішень демонструє, що відкриті інструменти забезпечують гнучкість дослідницького використання, тоді як хмарні платформи спрощують доступ до обчислювальних ресурсів і зменшують навантаження на локальну інфраструктуру користувача. Разом із тим для дослідницьких і інженерних задач актуальною залишається архітектурна модель, яка поєднує масштабованість, модульність, керованість життєвого циклу задачі та можливість інтерактивного доступу до результатів.

Важливою архітектурною особливістю FEA-систем є наявність великих за обсягом проміжних і кінцевих даних: геометричних моделей, сіток, матриць, векторів розв’язку, результатів постобробки та даних для візуалізації. Передавання таких даних через універсальний канал повідомлень або включення

їх у кожен міжсервісний виклик створює значні накладні витрати, тому доцільно розділяти керування життєвим циклом задачі (статуси, події, метадані) та передавання великих даних (артефакти) [11, 12].

Отже, FEA-система є неоднорідним обчислювальним конвеєром, окремі стадії якого відрізняються за характером операцій, вимогами до ресурсів і обсягами сформованих даних. Поєднання тривалих чисельних обчислень, великих проміжних даних та необхідності оперативного доступу користувача до стану й результатів задачі визначає потребу в архітектурі, що підтримує розділення відповідальності між компонентами, асинхронну координацію стадій та відокремлене зберігання великих обчислювальних даних. Ці особливості формують основу для подальшого аналізу готових FEA-рішень та визначення архітектурних обмежень їх застосування у веборієнтованому розподіленому середовищі.

1.3 Аналіз та порівняння готових рішень

Для оцінки сучасного стану розвитку програмного забезпечення у сфері комп’ютерного інжинірингу доцільно проаналізувати наявні FEA-рішення. До аналізу було обрано як провідні комерційні хмарні платформи, так і популярні відкриті інструменти, що дозволяє оцінити архітектурні тренди з різних точок зору.

Окрему групу відкритих засобів становлять програмні середовища для формулювання та чисельного розв’язання задач методом скінченних елементів, зокрема FEniCS [40] і SfePy [41]. У межах цього дослідження їх доцільно розглядати передусім як науково-обчислювальні середовища для програмної реалізації FEM-постановок, тоді як предметом подальшого порівняння є також готові настільні та веборієнтовані платформи повного циклу.

Спеціалізована відкрита ліцензована мовна платформа FreeFEM орієнтована на розв’язання диференціальних рівнянь у частинних похідних (PDE) у 2D та 3D просторах [42]. Вона надає користувачеві можливість

написання власних фізичних модулів за допомогою внутрішньої мови. Проте, на відміну від SimScale, вона не має вбудованої автоматично масштабованої хмарної інфраструктури «з коробки».

Elmer є відкритим програмним забезпеченням для розв'язання задач мультифізики [43]. Воно демонструє відмінну масштабованість та підтримку паралельних обчислень, що дозволяє запускати його на широкому спектрі обладнання – від локальних ноутбуків до суперкомп'ютерів. Система легко адаптується під специфічні потреби дослідників завдяки відкритому коду.

ANSYS Cloud є прикладом міграції класичного важкого монолітного ПЗ у хмарне середовище [44]. Він пропонує гнучкі обчислювальні ресурси за керованими моделями та самообслуговуванням, забезпечуючи користувачам доступ до майже необмежених потужностей з будь-якого пристрою. Однак архітектурно він залишається доволі закритим комерційним екосистемним рішенням.

SimScale є представником сучасного хмарного підходу (веборієнтована платформа) [45]. Платформа повністю функціонує в браузері та використовує еластичну обчислювальну інфраструктуру, що автоматично масштабується під потреби мультифізичних розрахунків (CFD, FEA, thermal). Важливою архітектурною перевагою є наявність SimScale API, що дозволяє інтегрувати сторонні розв'язувачі.

Для систематизації результатів аналізу та проведення порівняльної оцінки архітектурно-функціональних можливостей розглянутих систем, їхні ключові характеристики зведено у таблиці 1.2.

Таблиця 1.2 – Огляд готових рішень

Критерій	**ANSYS Cloud**	**SimScale**	**Elmer**	**FreeFEM**
Модель доступу	Хмарні обчислювальні ресурси для симуляцій; наявні керовані рішення та варіанти самообслуговування	Хмарна платформа, повний доступ через вебінтерфейс у браузері	Програмне забезпечення із запуском на різних платформах	Програмна та мовна платформа для розв'язання диференціальних рівнянь у частинних похідних (2D та 3D)

Продовження таблиці 1.2

Критерій	**ANSYS Cloud**	**SimScale**	**Elmer**	**FreeFEM**
Масштабування / інфраструктура	Гнучка, масштабована обчислювальна потужність, майже необмежені ресурси	Еластична обчислювальна інфраструктура, вбудоване керування для автоматичного масштабування	Підтримка паралельних обчислень із відмінною масштабованістю	–
Сфера застосування (фізичні процеси)	–	Інтегровані модулі: гідрогазодинаміка, міцнісний аналіз, теплові процеси, електромагнетизм	Мультифізика: механіка рідин, структурний аналіз, електромагнетизм, теплопередача, акустика	Мультифізична платформа скінченних елементів; розв'язувач диференціальних рівнянь
Спосіб взаємодії	Доступ з будь-якого місця та практично з будь-якого пристрою	Миттєво доступний вебінтерфейс та середовище для спільної роботи	Локальний або кластерний запуск (фокус на застосуванні розв'язувача)	Спеціалізована мова опису моделей на кількох сітках, реалізація власних фізичних модулів
Розширюваність	Обмежена екосистемою розробника	Відкрита інтеграція, інтерфейс програмування SimScale API, підтримка сторонніх розв'язувачів	Легко адаптується та налаштовується під нові потреби користувачів	Можливість реалізації власних модулів фізики за допомогою внутрішньої мови FreeFEM
Ліцензія	Комерційна	Комерційна (модель SaaS)	Відкритий код	Вільна ліцензія LGPL 3.0

Проведений аналіз показує, що сучасний ринок FEA-систем розділений: комерційні платформи (SimScale, ANSYS Cloud) забезпечують відмінну хмарну масштабованість, але є закритими, тоді як відкриті наукові інструменти (Elmer, FreeFEM) надають максимальну гнучкість і розширюваність, але зорієнтовані переважно на традиційну локальну чи кластерну експлуатацію. Таким чином, актуальним науково-практичним завданням є розроблення відкритої архітектурної моделі FEA-системи на базі мікросервісів, яка поєднувала б можливість вибіркового масштабування, модульну розширюваність і кероване проходження повного життєвого циклу задачі. Невирішеною залишається проблема узгодження асинхронних стадій FEA-конвеєра, передавання великих артефактів поза брокером повідомлень і кількісного визначення умов, за яких горизонтальне масштабування обчислювального компонента забезпечує реальний приріст продуктивності.

1.4 Передумови застосування мікросервісної архітектури до FEA-платформ

Надалі під FEA-платформою в роботі розуміється веборієнтована система з браузерним клієнтом і серверним розподіленим контуром; настільні застосунки розглядаються як клас систем для архітектурного порівняння.

Застосування мікросервісної архітектури до FEA-платформ є обґрунтованим насамперед через декомпозицію FEA-процесу. Повний цикл скінченно-елементного аналізу містить відносно самостійні етапи: підготовку моделі, генерацію сітки, чисельне розв’язання, післяобробку та візуалізацію. Кожен з них має окрему логіку, різну тривалість виконання та різні вимоги до обчислювальних ресурсів. Тому поділ системи на автономні компоненти відповідає як загальним принципам архітектури, так і внутрішній структурі предметної області [7–10, 31–34, 46].

Першою передумовою є непропорційність навантаження між етапами. У серійних розрахунках, параметричних дослідженнях або при виконанні великої кількості незалежних задач саме компонент чисельного розв’язувача (solver) стає основним вузьким місцем. Монолітна архітектура ускладнює масштабування лише цього етапу без одночасного масштабування інших компонентів. Натомість мікросервісний підхід дозволяє виділити обчислювальний сервіс та збільшувати кількість його екземплярів незалежно [7–10, 31].

Другою передумовою є незалежний розвиток компонентів. Алгоритми генерації сітки, чисельні методи розв’язання та засоби візуалізації можуть змінюватися з різною швидкістю. Мікросервісна модель, за умови стабільних контрактів, дозволяє оновлювати або замінювати окремі компоненти без повної перебудови системи [7–9, 32–34].

Третьою передумовою є асинхронний характер довготривалих задач. FEA-розрахунки не завжди можуть бути виконані в межах короткого HTTP-запиту: генерація сітки або розв’язання можуть тривати значний час залежно від розміру моделі та параметрів дискретизації. Тому архітектура повинна підтримувати

запуск задачі, фонове виконання, збереження проміжних станів і інформування користувача про прогрес. Це підводить до подієво-орієнтованого підходу, у якому компоненти реагують на зміну стану задачі, а не блокують один одного синхронними викликами [11–14].

Четвертою передумовою є робота з великими артефактами. Сітки, результати розрахунків і дані для 3D-візуалізації можуть мати значний обсяг, тому ефективною є модель, у якій сервіси обмінюються подіями та метаданими, а великі артефакти зберігаються в спеціалізованому сховищі; подія сигналізує про готовність результату та містить посилання на нього, але не передає його повністю. Такий підхід реалізує розглянутий у підрозділі 1.2 принцип розділення керування життєвим циклом задачі та передавання великих даних. Події й метадані утворюють керівний контур платформи, тоді як сітки, результати обчислень і дані візуалізації зберігаються та передаються як окремі артефакти [11, 12, 46].

П’ятою передумовою є реактивна взаємодія з користувачем: користувач має бачити статус задачі, етап виконання, прогрес, помилки та доступність результатів для перегляду. Для веборієнтованих платформ це зумовлює потребу в компоненті, який агрегує стани задачі та передає їх до UI у режимі, близькому до реального часу (push-модель) [46].

На основі зазначених передумов FEA-платформу доцільно концептуально описувати як набір взаємопов’язаних, але автономних компонентів: інтерфейсу користувача, єдиної точки входу (API Gateway), агрегатора/BFF (Backend for Frontend), препроцесора, процесора, постпроцесора, брокерів і сховищ даних [31, 46].

Рисунок 1.4 відображає архітектурну модель веборієнтованої FEA-платформи.

Отже, застосування мікросервісної архітектури є обґрунтованим для FEA-платформ, що характеризуються непропорційним навантаженням на окремі етапи, довготривалими обчисленнями, значними обсягами проміжних артефактів і потребою у вибірковому масштабуванні. Водночас доцільність такого підходу

залежить від складності системи та готовності інфраструктури до підтримки розподіленої взаємодії [15, 16, 35].

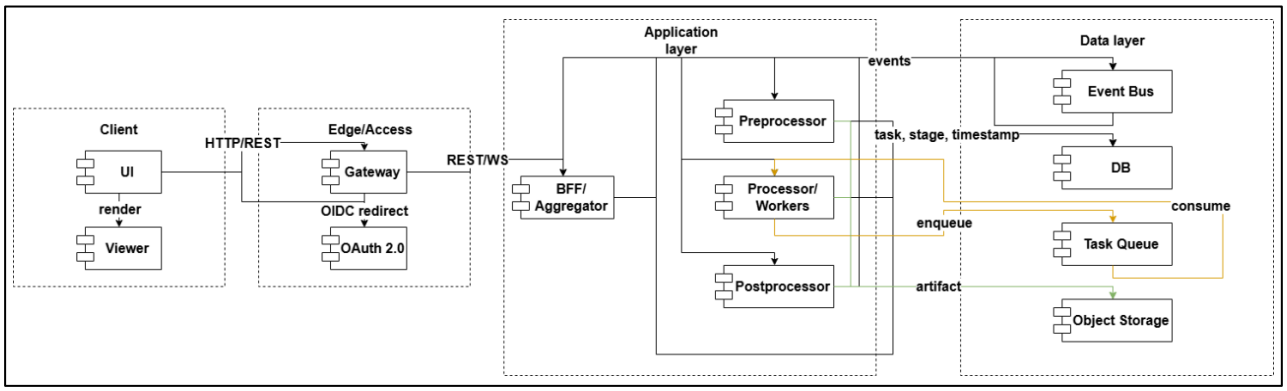


Рисунок 1.4 – Архітектурна модель веборієнтованої FEA-платформи

Розподіл платформи на автономні сервіси створює можливість незалежного масштабування найбільш ресурсомістких етапів, асинхронної обробки довготривалих задач і винесення великих артефактів до спеціалізованого сховища. Практична ефективність такого рішення, зокрема межі масштабування обчислювального компонента, потребує експериментальної перевірки.

1.5 Висновки до розділу 1

У розділі розглянуто архітектурні підходи до побудови програмних систем та обґрунтовано актуальність їх аналізу для FEA-платформ. Встановлено, що монолітна архітектура є придатною для швидкого початку розроблення та спрощує початкову інтеграцію компонентів, проте створює обмеження щодо вибіркового масштабування, ізоляції відмов, незалежного оновлення модулів і підтримки довготривалих обчислювальних процесів. Мікросервісний підхід, своєю чергою, забезпечує незалежність життєвого циклу компонентів і можливість їх окремого масштабування, однак потребує врахування додаткової складності розподіленої взаємодії.

Показано, що FEA-систему доцільно розглядати як повний обчислювальний конвеєр, який охоплює підготовку даних, чисельне розв’язання, післяоброблення та візуалізацію. Кожна з цих стадій має власний профіль використання ресурсів, формує специфічні проміжні або кінцеві дані та може мати різну тривалість виконання. Це визначає специфічні архітектурні вимоги до керування життєвим циклом задачі, координації стадій і зберігання обчислювальних артефактів.

Аналіз готових рішень показав, що сучасні комерційні хмарні платформи забезпечують розвинені засоби доступу до обчислювальних ресурсів, тоді як відкриті наукові інструменти надають ширші можливості модифікації чисельних алгоритмів і постановок. Водночас актуальною залишається потреба у відкритій архітектурній моделі, що поєднує веборієнтований доступ, асинхронне виконання повного FEA-циклу, відокремлене зберігання великих артефактів і незалежне масштабування найбільш ресурсоємних стадій.

Узагальнено основні передумови застосування мікросервісного підходу до FEA-платформ: нерівномірність навантаження між стадіями, необхідність вибіркового масштабування обчислювального компонента, асинхронний характер довготривалих обчислень, робота з великими артефактами та потреба в оперативному інформуванні користувача. Отримані результати становлять теоретико-архітектурну основу для формулювання вимог, визначення меж відповідальності компонентів і побудови концептуальної моделі платформи в розділі 2.

РОЗДІЛ 2. КОНЦЕПТУАЛЬНЕ ПРОЄКТУВАННЯ МІКРОСЕРВІСНОЇ FEA-ПЛАТФОРМИ

2.1 Постановка задачі проєктування FEA-платформи

Сучасні платформи скінченно-елементного аналізу (FEA) доцільно розглядати як конвеєр обробки, що охоплює повний цикл: підготовка даних, чисельне розв’язання, післяоброблення та візуальне представлення результатів. Конвеєрна природа FEA зумовлює потребу у підтримці довготривалих обчислень, керуванні життєвим циклом задачі та роботі з великими артефактами (сітка, чисельний розв’язок, пакет для візуалізації) [11–14, 46]. Відповідно, задача проєктування полягає у побудові такої архітектури, яка забезпечує узгоджене виконання етапів, відтворюваність запусків і керованість навантаження у розподіленому середовищі.

Для реалізації слабкозв’язаної взаємодії етапів конвеєра обрано подієво-орієнтовану модель (EDA), у якій етапи реагують на події життєвого циклу задачі. Це зменшує залежність між компонентами та дозволяє уникнути блокуючих синхронних взаємодій для довготривалих операцій. У такій моделі особливе значення мають: гарантії доставки та порядку повідомлень у брокері подій, ідемпотентність обробників для запобігання повторній обробці при повторній доставці, окреме зберігання артефактів поза повідомленнями [11–14].

На рисунку 2.1 продемонстровано схему узагальненого FEA-конвеєра і артефактів.

Таким чином, задача проєктування полягає не лише у функціональному розділенні повного FEA-циклу, а й у визначенні меж відповідальності компонентів, правил переходу між стадіями, контрактів подій і артефактів, а також способів зберігання стану задачі [47–49]. Сформована постановка визначає необхідність поєднання асинхронної координації, відокремленого зберігання великих даних і незалежного масштабування ресурсоємних компонентів.

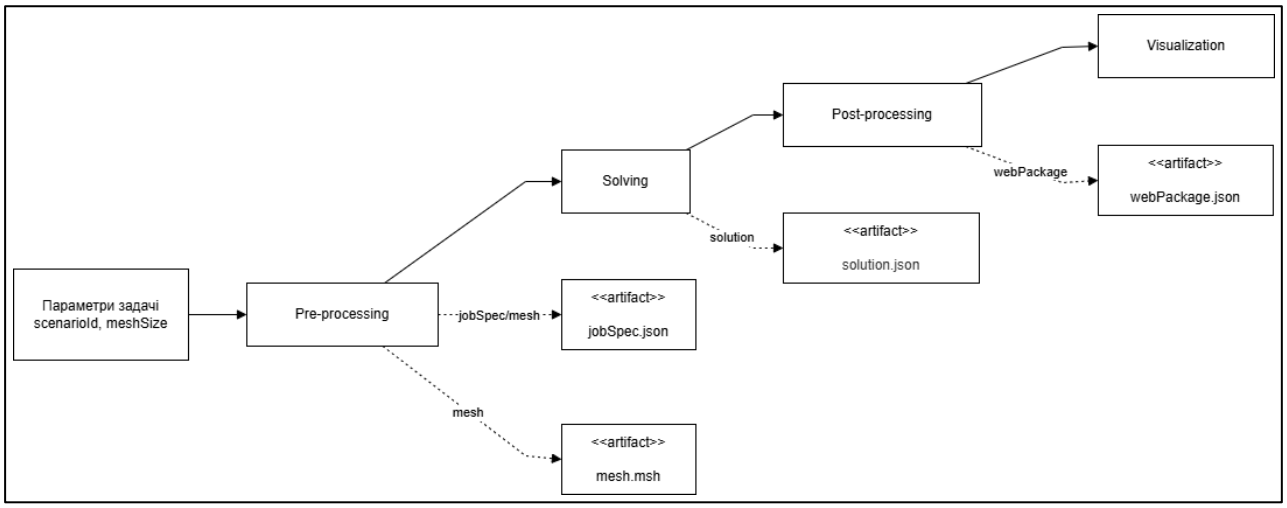


Рисунок 2.1 – Узагальнений FEA-конвеєр і артефакти

Для подальшої формалізації архітектури використовуються уніфікована мова моделювання UML [50], нотація функціонального моделювання IDEF0 [51], принципи REST-взаємодії [52], підхід єдиної точки входу API Gateway [53] та двоспрямований канал WebSocket [54].

2.2 Функціональні вимоги

Функціональні вимоги визначають, які дії має підтримувати платформа з погляду користувача та життєвого циклу задачі [47]. У центрі взаємодії знаходиться користувач, який створює задачу моделювання, спостерігає за її виконанням і переглядає результати. При цьому сама платформа повинна автоматизувати запуск конвеєра обробки, забезпечити прозорі стани/стадії та надати результат у формі, придатній для 3D-візуалізації. Така логіка узгоджується з концепцією повного FEA-циклу та роллю UI як клієнта, що отримує статуси через REST/WebSocket без включення обчислювальної логіки в клієнт.

Для фіксації зовнішньої поведінки системи використано діаграму варіантів використання [50]. На рисунку 2.2 подано набір ключових прецедентів, які охоплюють повний життєвий цикл: створення задачі, моніторинг виконання, а також отримання та візуалізація результатів.

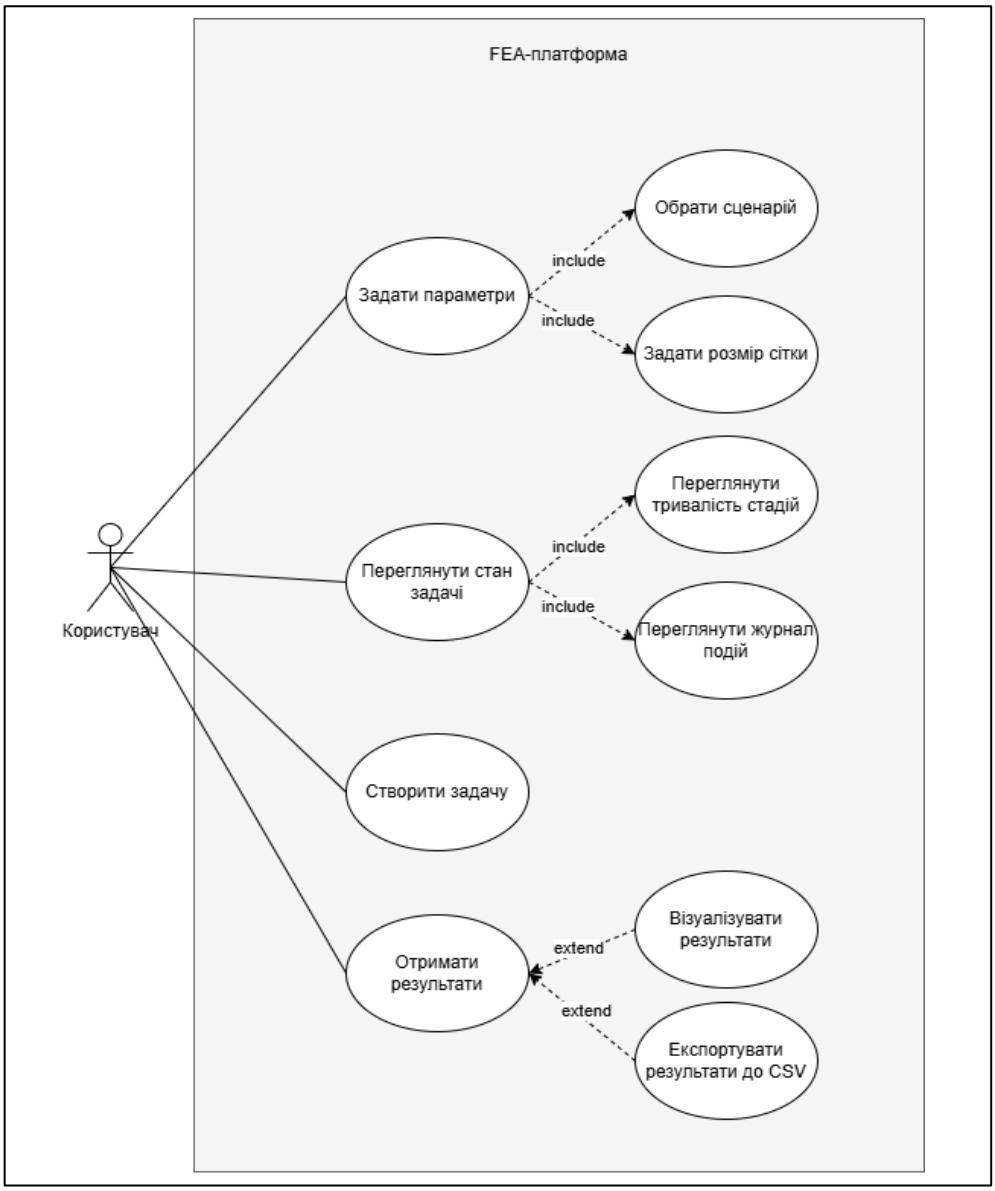


Рисунок 2.2 – Діаграма варіантів використання

Діаграма відображає, що користувач взаємодіє з платформою як з єдиною системою, а внутрішня сервісна взаємодія є реалізаційним механізмом виконання прецедентів. Зокрема, після підтвердження параметрів постановки система створює задачу та ініціює конвеєр обробки підготовка даних/чисельне розв’язання/післяоброблення у асинхронному режимі: користувач не керує етапами вручну, а спостерігає за їхнім станом і прогресом. Такий підхід узгоджується з конвеєрною природою FEA, де підготовка даних, чисельне розв’язання та післяоброблення є різними за профілем ресурсів і доцільно організуються як послідовність автономних стадій, синхронізованих подіями та станами задачі.

З метою формалізації прецедентів, у таблиці 2.1 подано відповідність між вимогами, їхнім результатом та компонентами, що забезпечують реалізацію.

Таблиця 2.1 – Функціональні вимоги FEA-платформи

Прецедент	**Призначення / результат**	**Вхідні дані**	**Вихідні дані**	**Основні компоненти**
Задати параметри	Підготовка постановки задачі на рівні параметрів користувача	scenarioId, meshSize	Валідований набір параметрів для створення задачі	UI
Створити задачу	Створення задачі, присвоєння taskId, автоматичний запуск конвеєра обробки	Параметри	taskId, початковий стан задачі	UI, BFF/Aggregator
Переглянути стан задачі	Відображення прогресу, станів стадій і подій виконання	taskId	Статус/прогрес/стадії/події	UI, BFF/Aggregator
Переглянути тривалість стадій	Часова шкала та тривалості стадій created, preprocess, solve, postprocess	taskId	Мітки часу, тривалості	BFF/Aggregator, сховище метаданих
Переглянути журнал подій	Перегляд останніх подій життєвого циклу задачі	taskId	Перелік подій	BFF/Aggregator
Отримати результати	Отримання результатних артефактів/даних задачі	taskId	Посилання/дані результатів	BFF/Aggregator, об’єктне сховище
Візуалізувати результати	3D-перегляд результатів	taskId + дані результату	Візуальне представлення поля	UI + модуль 3D-візуалізації
Експортувати результати до CSV	Вивантаження параметрів серії, часових і чисельних метрик задач для подальшого статистичного аналізу	Конфігурація серії та зібрані метрики задач	CSV-файл	UI (формує CSV) + BFF/Aggregator (надає дані)

Нижче наведено опис основних прецедентів.

Прецедент «Задати параметри». Робота з платформою починається з формування постановки задачі на рівні параметрів, доступних користувачу. Цей прецедент включає два обов’язкові кроки: вибір сценарію моделювання та задання розміру сітки. У межах прототипу параметри сценарію визначають тип задачі та узгоджений набір внутрішніх налаштувань, тоді як meshSize впливає на дискретизацію і, відповідно, на складність обчислень та обсяг артефактів. Якщо введені значення невалідні (наприклад, від’ємний або нульовий розмір сітки), UI блокує запуск наступного кроку та відображає повідомлення про помилку.

Прецедент «Створити задачу». Після задання параметрів користувач ініціює створення задачі. Система формує унікальний ідентифікатор taskId,

зберігає метадані задачі та запускає виконання конвеєра обробки як послідовність стадій: підготовка даних / чисельне розв’язання / післяоброблення. Виконання конвеєра відбувається асинхронно: користувач одразу отримує taskId і переходить до моніторингу, тоді як зміна станів стадій і поява результатів відбуваються незалежно від клієнтського запиту. У разі помилки створення (наприклад, недоступність сховища метаданих) система відхиляє запит і повертає повідомлення про неможливість створити задачу.

Прецедент «Переглянути стан задачі». Після створення задачі користувач переходить до моніторингу. Даний прецедент охоплює відображення поточного статусу, прогресу та останніх подій. Він включає два підпрецеденти: перегляд тривалості стадій та перегляд журналу подій. Така структура забезпечує прозорість виконання: користувач бачить не лише факт завершення/помилки, а й те, на якій стадії виникла затримка або збій.

Прецедент «Отримати результати». Після завершення конвеєра користувач отримує доступ до результатних даних. Цей прецедент є базовим і може бути розширений двома опційними сценаріями: візуалізацією результатів або експортом даних.

Прецедент «Візуалізувати результати». У випадку потреби інтерпретації результатів користувач активує режим 3D-візуалізації, який відображає дискретне поле значень на сітці з можливістю взаємодії (масштабування, каркасного відображення тощо). Візуалізація є опційною дією після «Отримати результати», оскільки у деяких сценаріях користувачу достатньо числових показників, не вдаючись до графічного аналізу.

Прецедент «Експортувати результати до CSV». Дозволяє експортувати у форматі CSV параметри пакетного запуску, статуси задач, часові характеристики стадій, показники пропускної здатності, параметри сітки та чисельні похибки. Цей прецедент не є обов’язковою частиною повсякденного виконання одиничної FEA-задачі, а використовується під час експериментального оцінювання платформи та підготовки даних для позасистемного статистичного аналізу.

Отже, сформовані функціональні вимоги охоплюють повний користувацький сценарій роботи з платформою: від задання параметрів математичної постановки й створення задачі до моніторингу її виконання, отримання, візуалізації та експорту результатів. На цьому рівні платформа розглядається користувачем як єдина система, тоді як внутрішня декомпозиція та подієва взаємодія компонентів є механізмами реалізації визначених функцій. Для забезпечення прогнозованої роботи цих механізмів необхідно також визначити вимоги до якості системи, що розглядаються в наступному підрозділі.

2.3 Нефункціональні вимоги

Нефункціональні вимоги визначають властивості якості платформи: масштабованість, надійність, продуктивність, безпека, відтворюваність і підтримуваність [48]. Для FEA-платформ вони є особливо критичними, оскільки система працює з довготривалими обчисленнями та великими артефактами, а користувач очікує прогнозованої поведінки та доступності результатів.

Платформа має забезпечити можливість горизонтального масштабування ресурсомістких етапів конвеєра. Це передбачає, що обчислювальні компоненти можуть запускатися у кількох екземплярах, а інфраструктура забезпечує кероване масштабування та відновлення.

Для розподілених конвеєрів характерною є потреба у відновленні після часткових відмов: збій окремого сервісу не повинен призводити до втрати стану всієї задачі. На концептуальному рівні для цього передбачаються самовідновлення контейнерів оркестратором, контроль станів стадій, ідемпотентне опрацювання повторних повідомлень і можливість організації повторних спроб [11–16]. Ступінь реалізації кожного з цих механізмів у прототипі уточнюється в розділі 3.

Передавання великих даних (mesh/results) через брокер повідомлень є неефективним. Натомість артефакти доцільніше зберігати у сховищі об’єктів, а у подіях передаються лише посилання та метадані (формат/хеш/розмір) [11, 46].

У подієвих системах можлива повторна доставка повідомлень, зокрема за семантики at-least-once. Тому архітектура має передбачати перевірку стану стадії перед повторним виконанням ресурсомісткої операції та узгодження станів за принципом eventual consistency. Конкретна семантика повторної доставки залежить від моменту фіксації Kafka offset і способу обробки помилкових повідомлень [11–14].

Для підтримки інтерактивної роботи необхідна push-модель оновлення станів (WebSocket), коли користувач отримує прогрес та статуси без постійного опитування. Це зменшує затримки та покращує UX для довготривалих задач [54].

Платформа має підтримувати автентифікацію користувачів, авторизацію доступу до задач і артефактів, а також захист каналів обміну. Для веборієнтованих рішень типовими є стандартні протоколи на базі OAuth2/OpenID Connect [55–59].

Доступ користувача до серверного контуру доцільно організувати через єдину точку входу (API Gateway). Цей компонент забезпечує приймання зовнішніх HTTP(S)-запитів та базову маршрутизацію до серверної підсистеми [53].

Узагальнення нефункціональних вимог і відповідних архітектурних рішень наведено у таблиці 2.2.

Таблиця 2.2 – Нефункціональні вимоги та архітектурні рішення

Вимога	**Архітектурне рішення**
Масштабованість	незалежне масштабування сервісів/виконавців, оркестрація
Відмовостійкість	повторні спроби, перезапуск компонентів, ідемпотентність
Великі дані	об’єктне сховище для артефактів, посилання та метадані
Узгодженість	ідемпотентне опрацювання та узгодженість у підсумку для станів
Оперативне оновлення інтерфейсу	push-статуси через WebSocket
Відтворюваність	фіксація параметрів задачі, стабільні контракти
Безпека доступу	OAuth 2.0 / OpenID Connect, перевірка JWT, зв’язування задачі з ідентифікатором користувача
Єдина точка входу	API Gateway та BFF/Aggregator для ізоляції внутрішніх сервісів від клієнта
Спостережуваність виконання	збереження станів і часових міток стадій, журнал подій, push-оновлення через WebSocket

Сформовані нефункціональні вимоги визначають архітектурні обмеження, які мають бути враховані під час подальшої декомпозиції платформи. Масштабованість потребує відокремлення ресурсоємних компонентів, надійність – збереження стану задачі та коректного опрацювання повторних повідомлень, продуктивність – розмежування керівних повідомлень і великих даних, а безпека — централізованої автентифікації та контролю доступу до задач і результатів. Сукупність функціональних і нефункціональних вимог є основою для побудови концептуальної моделі системи та визначення взаємозв’язків між її компонентами.

2.4 Концептуальна модель системи

На концептуальному рівні платформа розглядається як цілісна система, що трансформує вхідну постановку (scenarioId, meshSize) у результати моделювання та дані візуалізації. Така модель корисна тим, що відокремлює вхідні впливи, керуючі обмеження, вихідні результати та механізми виконання, не деталізуючи внутрішню реалізацію компонентів [49, 51].

На рисунку 2.3 наведено IDEF0 концептуальної FEA-платформи.

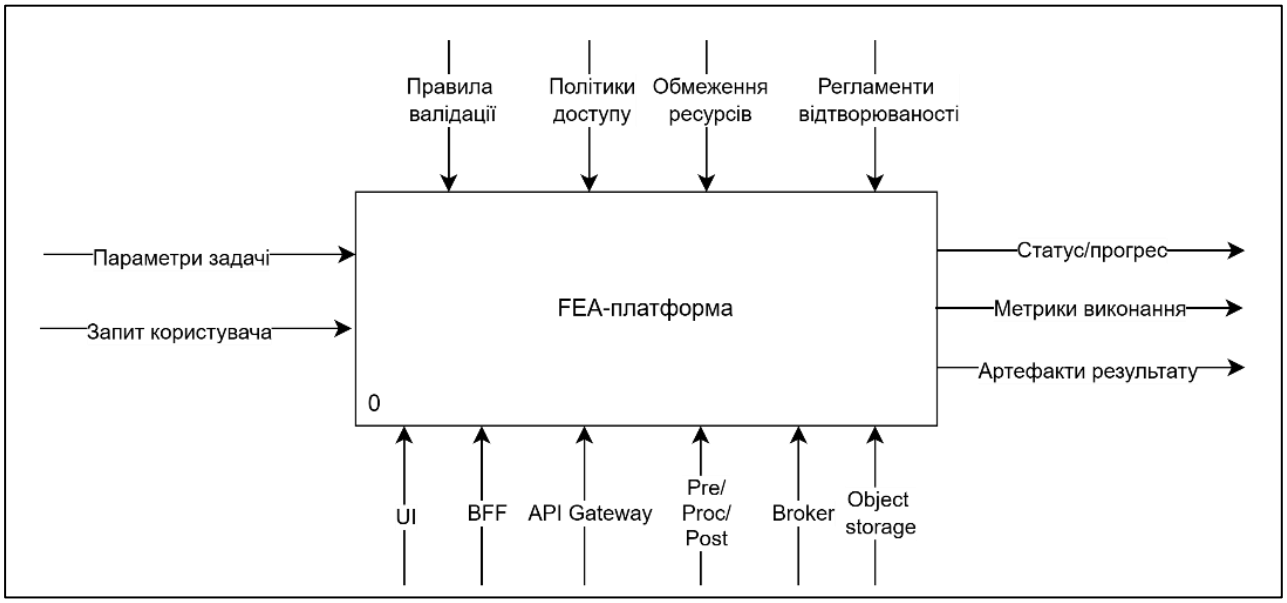


Рисунок 2.3 – IDEF0 концептуальної моделі системи

Модель фіксує дві принципові ідеї, важливі для подальшого проєктування: результат платформи включає не лише чисельний розв’язок, а й артефакти для подальшої візуалізації та аналітики; важкі дані доцільно розглядати як артефакти, керовані механізмами зберігання та доступу.

2.5 Сервісна декомпозиція FEA-циклу

Концептуальна декомпозиція FEA-платформи має відображати конвеєрну природу процесу і водночас забезпечувати вибіркове масштабування ресурсомістких етапів та слабке зв’язування між компонентами. Ключовими вимогами, що визначають декомпозицію, є: асинхронне виконання довготривалих етапів, мінімізація передавання великих артефактів між модулями, узгоджений життєвий цикл задачі у розподіленому середовищі та оперативне інформування користувача. Запропонований підхід ґрунтується на сервісній декомпозиції повного FEA-циклу на п’ять ізольованих компонентів (Preprocessor / Processor / Postprocessor / Aggregator / 3D Viewer) із подієвим оркеструванням етапів та проєктування на основі контрактів (contract-first) [7, 12, 34, 46].

У поточній реалізації 3D Viewer представлено як клієнтський модуль у складі вебінтерфейсу, однак концептуально він виділяється окремо, оскільки має іншу відповідальність (візуалізація результатів), інший профіль ресурсів (GPU/рендеринг) та окремий життєвий цикл розвитку порівняно з обчислювальними етапами конвеєра.

Архітектуру платформи доцільно розглядати як взаємодію двох логічних контурів – клієнтського та серверного, що мають різні цілі, обмеження та життєвий цикл. Клієнтський контур охоплює засоби взаємодії з користувачем: інтерфейс задання параметрів, відображення станів/прогресу, часова шкала стадій і журнал подій, а також модуль 3D-візуалізації результатів у складі UI. Серверний контур реалізує життєвий цикл задачі та виконання FEA-конвеєра:

приймання запитів, асинхронну координацію стадій, формування подій прогресу/готовності, зберігання метаданих та артефактів. Для інженерної платформи важливо, що клієнтський контур не містить обчислювальної логіки та працює через стабільний зовнішній інтерфейс, тоді як обчислювальні компоненти розміщені в серверному контурі й можуть масштабуватися незалежно.

На концептуальному рівні gateway забезпечує приймання зовнішніх HTTP(S)-запитів і маршрутизацію до агрегатора (BFF/Aggregator). Контроль доступу розглядається як функція вхідного контуру (gateway та/або BFF) і деталізується на рівні реалізації у наступних розділах, залежно від обраної схеми автентифікації та перевірки токенів.

Для узагальнення взаємозв'язків контурів та компонентів на рисунку 2.4 наведено компонентну схему платформи. Вона демонструє: вхідний edge-компонент (API Gateway), роль агрегатора для UI, подієву взаємодію між стадіями конвеєра та винесення важких артефактів у сховище об'єктів із передаванням через брокер лише метаданих і посилань. Винесення артефактів у об'єктне сховище та подієва синхронізація стадій розглядаються як ключові архітектурні рішення для зменшення навантаження на комунікаційний шар і підвищення масштабованості.

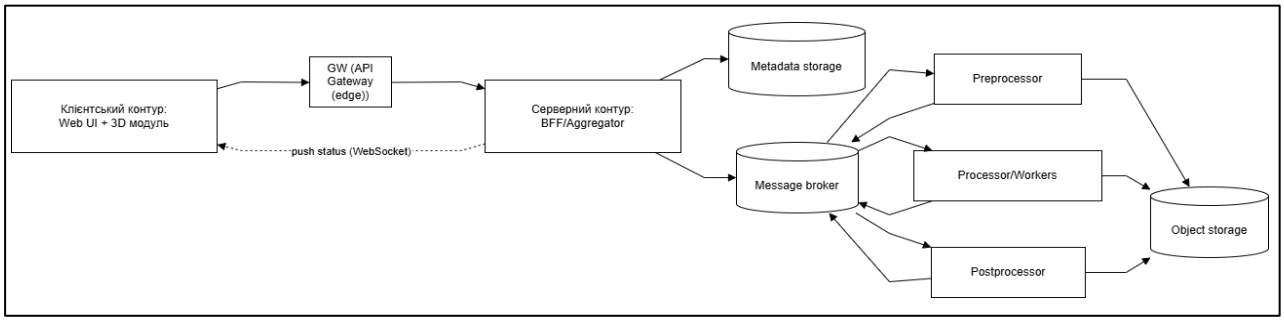


Рисунок 2.4 – Компонентна схема платформи з виділенням клієнтського та серверного контурів

Декомпозиція стадій конвеєра у вигляді окремих компонентів забезпечує можливість незалежного масштабування найресурсомісткіших етапів.

Наприклад, для Preprocessor підкреслюється доцільність асинхронного підходу та можливість масштабувати генерацію сіток окремо, не блокуючи вхідний контур. Аналогічно, Processor розглядається як ядро обчислень, на яке припадає основне навантаження, тому саме цей компонент є первинним кандидатом для горизонтального масштабування [10, 31]. Postprocessor, у свою чергу, функціонує як реактивний підписник, що спрацьовує після готовності результатів і трансформує їх у структури, придатні для аналітики та візуалізації; асинхронність дозволяє оптимізувати використання ресурсів та зменшити затримки.

Для формалізації відповідальностей кожного з ключових компонентів на рівні IDEF0 доцільно використати діаграми, що відображають входи, виходи, механізми та керуючі впливи сервісів [49–51]. Зокрема, для Preprocessor на рисунку 2.5 подано його IDEF0-модель, яка фіксує: вхідні параметри задачі, механізми генерації сітки та сховище артефактів, вихід у вигляді готової сітки та події про завершення.

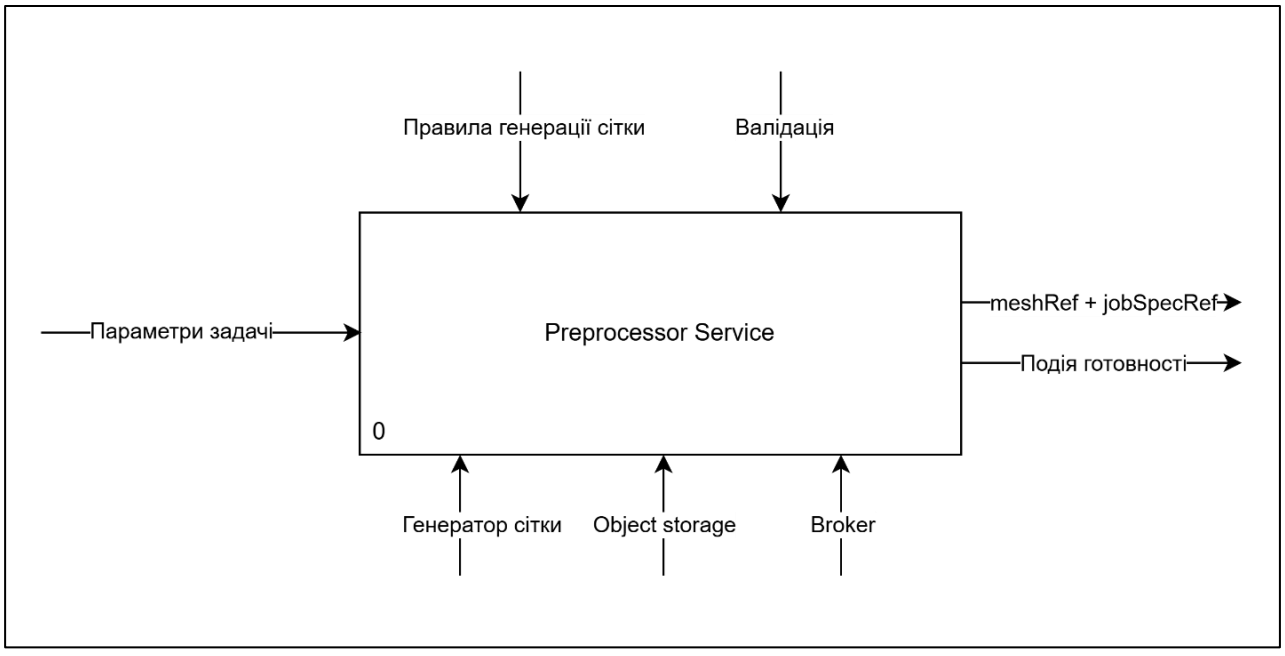


Рисунок 2.5 – IDEF0 Preprocessor Service

Аналогічні IDEF0-моделі для Processor і Postprocessor подано на рисунку 2.6 та рисунку 2.7 відповідно.

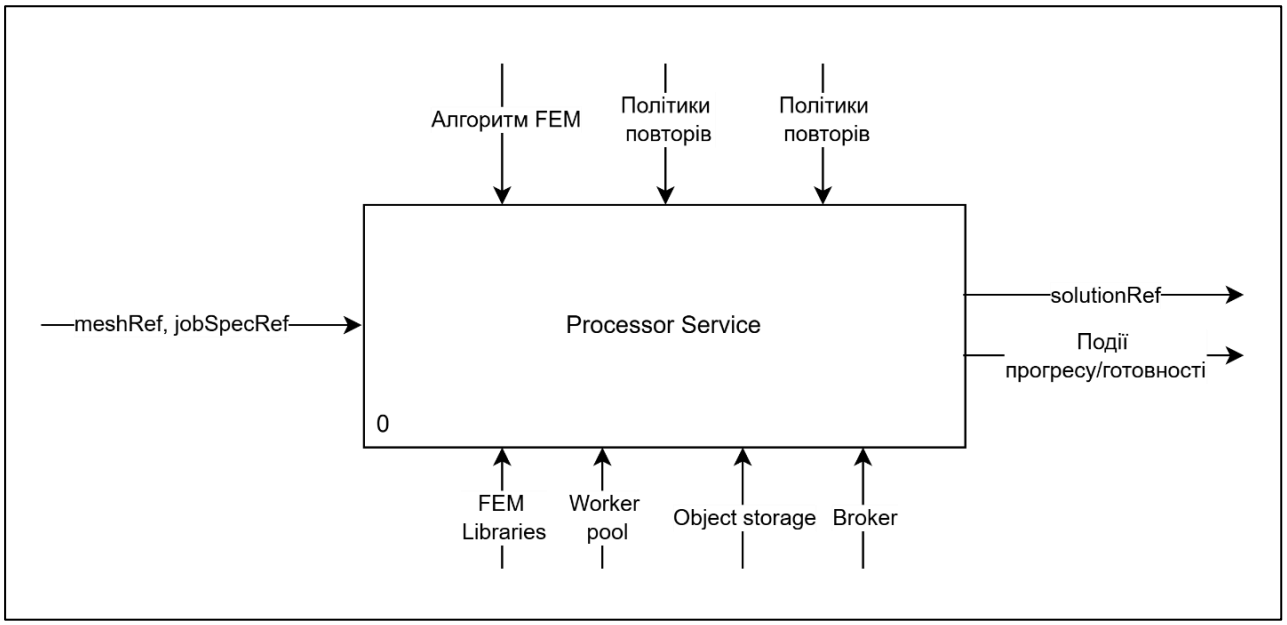


Рисунок 2.6 – IDEF0 Processor Service

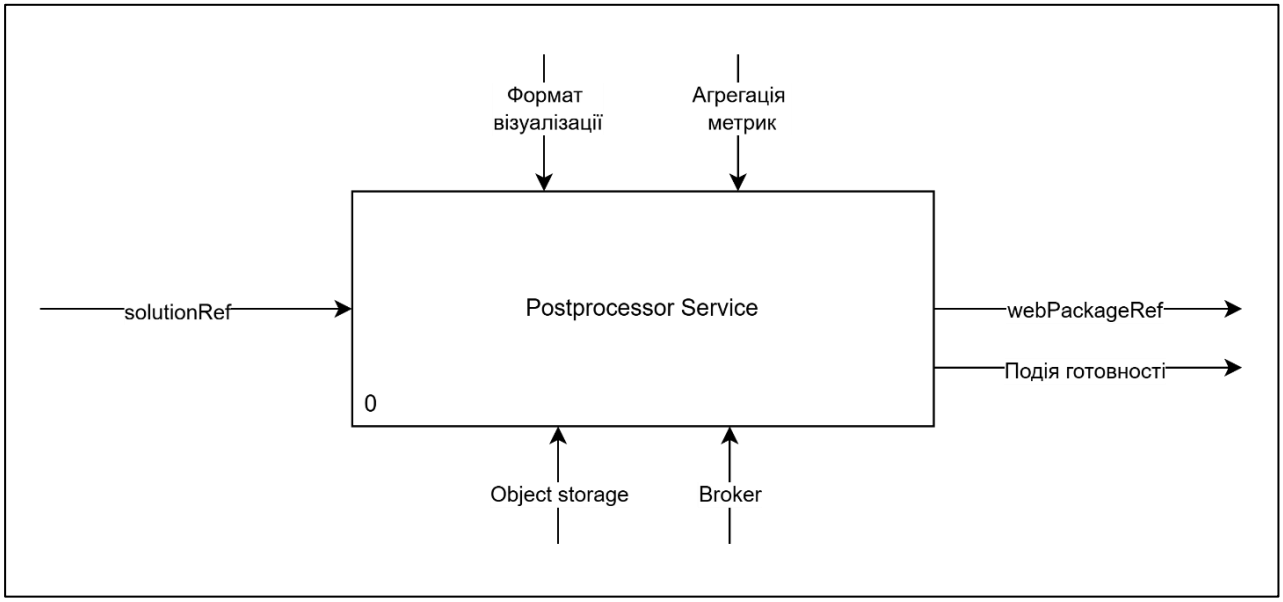


Рисунок 2.7 – IDEF0 Postprocessor Service

Окрему роль у сервісній декомпозиції відіграє Aggregator (BFF), який агрегує статуси та транслює їх до UI у push-режимі через WebSocket, що забезпечує оперативне інформування користувача про прогрес виконання задачі. Така модель взаємодії завершує логічний цикл декомпозиції: обчислювальні стадії залишаються незалежними та асинхронними, тоді як клієнтський контур отримує уніфікований погляд на стан задачі без прямої залежності від внутрішнього устрою конвеєра.

2.6 Концептуальна модель взаємодії компонентів

Оскільки платформа має конвеєрну природу, принципово важливо описати динаміку її роботи: як узгоджуються переходи між стадіями, як у розподіленому середовищі забезпечується керування станом задачі та як користувач отримує оновлення без блокуючих запитів. У подієво-орієнтованому підході стадії конвеєра синхронізуються через повідомлення брокера: кожен компонент реагує лише на релевантні події, що підвищує незалежність модулів і дозволяє уникати блокувань, а підписки на події виконуються на етапі запуску сервісів.

Доступ користувача до серверного контуру здійснюється через API Gateway, а клієнтський інтерфейс виступає виключно клієнтом gateway, отримуючи дані у реактивному форматі через WebSocket або через REST [52, 53]. Агрегація подій життєвого циклу задачі та трансляція станів у клієнтський контур покладається на BFF/Aggregator, який споживає події, формує уніфікований знімок стану і передає його до UI через WebSocket.

Для узагальненого представлення життєвого циклу задачі на рівні платформи використовується діаграма послідовності (див. рис. 2.8). Вона демонструє, що користувацька дія «Створити задачу» запускає асинхронний конвеєр обробки, у якому стадії повідомляють про завершення шляхом публікації подій, а важкі артефакти (mesh, solution, дані для візуалізації) зберігаються в об’єктному сховищі і доступні за посиланням. Такий підхід узгоджується з принципом розділення метаданих задачі та важких артефактів, коли через брокер передаються лише метадані/посилання.

Наведена послідовність підкреслює концептуальні властивості платформи: асинхронність довготривалих стадій, слабке зв’язування компонентів через брокер повідомлень, винесення артефактів у об’єктне сховище, реактивне інформування користувача. Відповідно до концепції агрегатора, останній постійно споживає події, що стосуються стану задачі (початок/завершення стадій, доступність результатів), агрегує їх у структуру стану та транслює в UI через WebSocket [54].

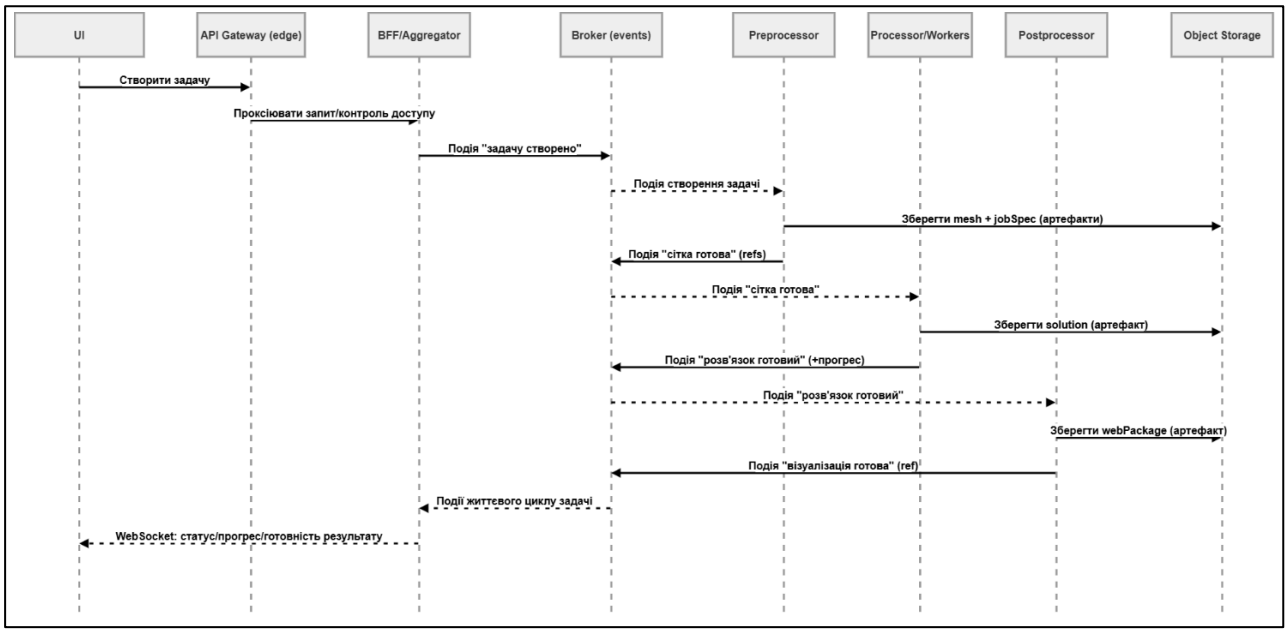


Рисунок 2.8 – Діаграма послідовності життєвого циклу задачі

Діаграма послідовності відображає взаємодію компонентів у часі, однак не деталізує внутрішні кроки кожного сервісу. Для уточнення поведінки стадій конвеєра доцільно застосовувати діаграми діяльності, які описують логіку роботи окремого компонента в контексті подієвого циклу. На рисунку 2.9 представлено обробку задачі на етапі підготовки даних.

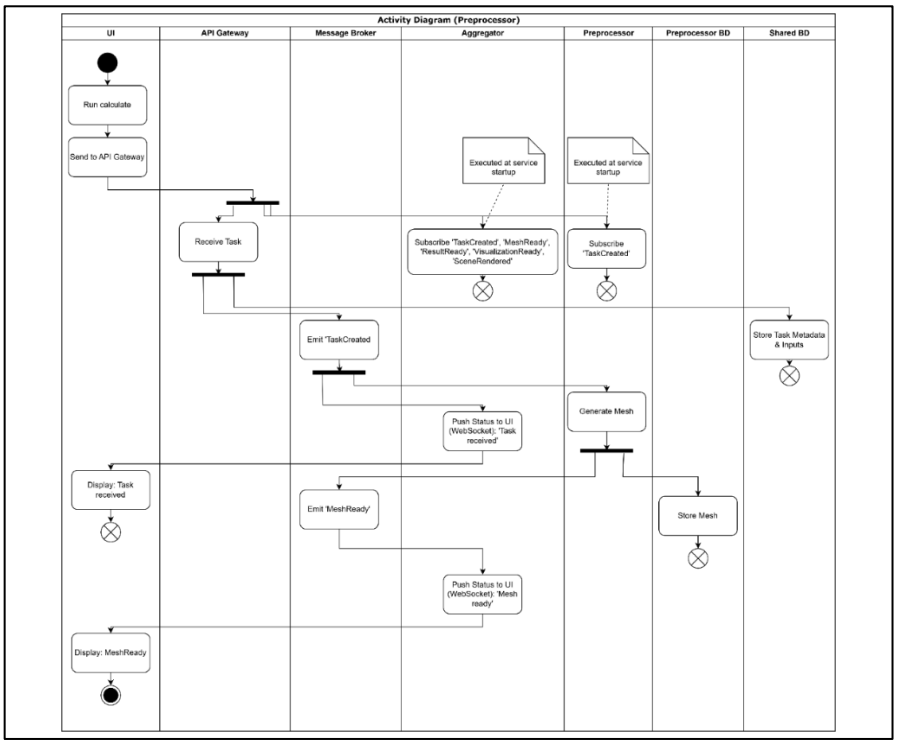


Рисунок 2.9 – Діаграма діяльності Preprocessor Service [46]

Рисунок 2.10 демонструє логіку роботи Processor Service. Рисунок 2.11 відображає етап обробки результатів у Postprocessor Service.

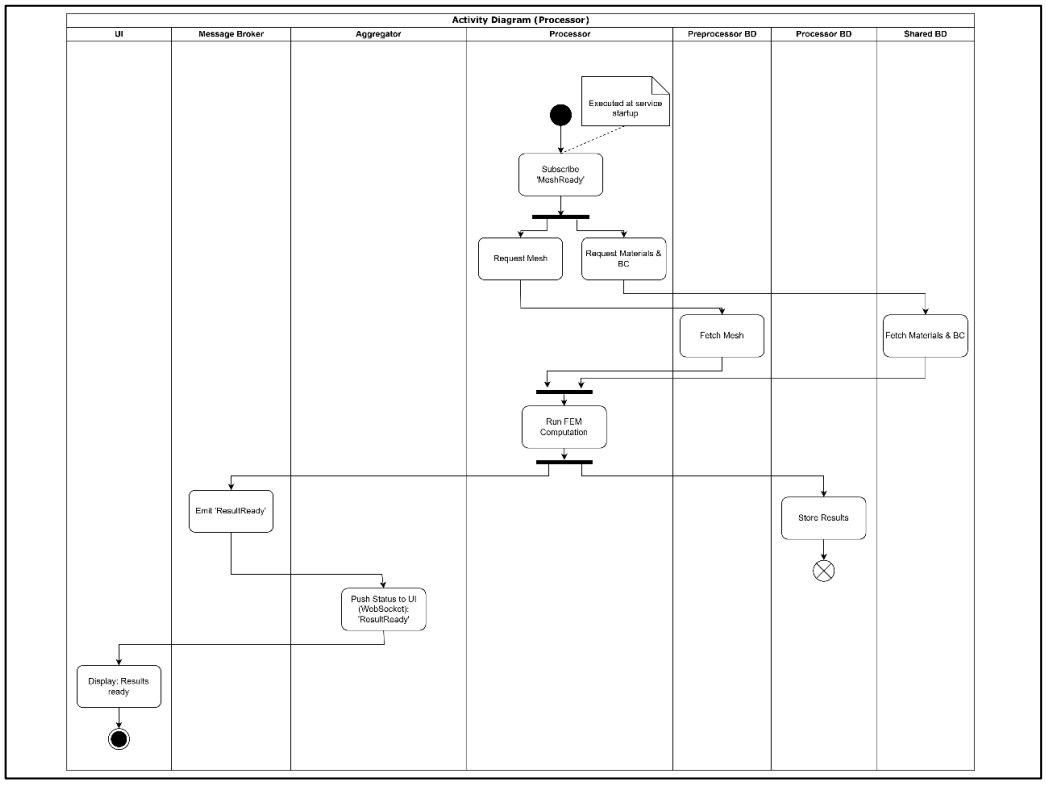


Рисунок 2.10 – Діаграма діяльності Processor Service [46]

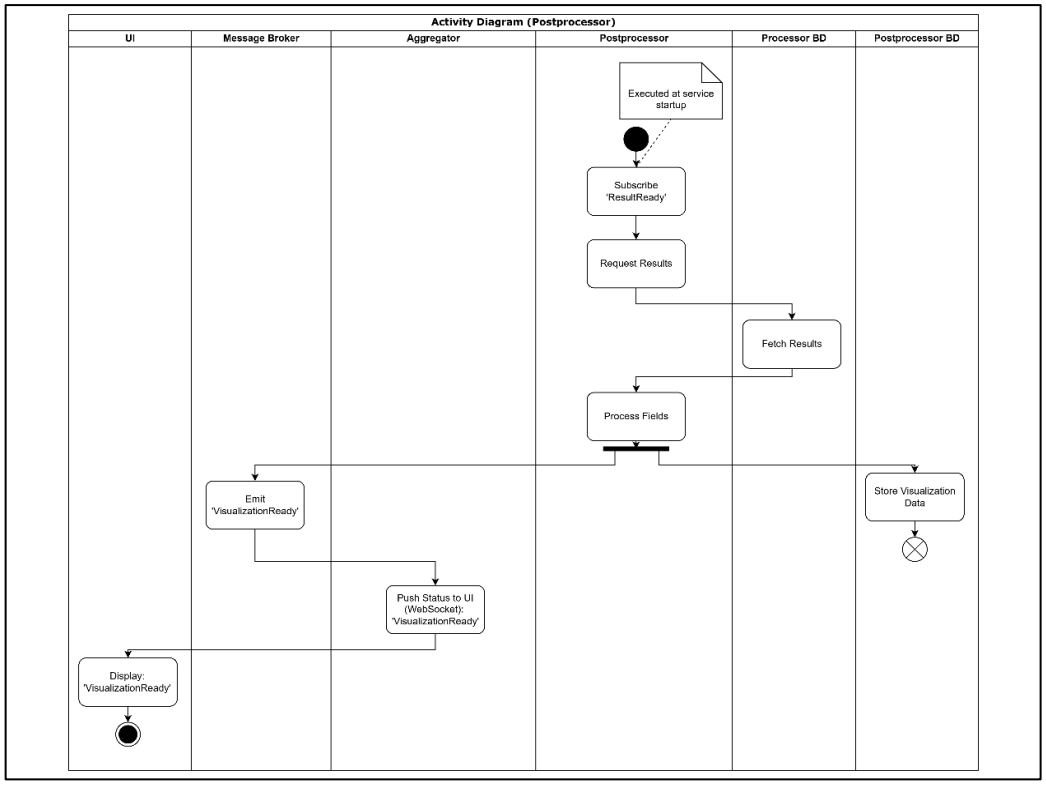


Рисунок 2.11 – Діаграма діяльності Postprocessor Service [46]

Наведені діаграми діяльності підкреслюють ключову концептуальну властивість платформи: кожен сервіс реалізує одну стадію конвеєра, виконує її асинхронно та сигналізує про завершення подією [11–14, 46]. Це забезпечує гнучкість у розподіленому виконанні та створює передумови для вибіркового масштабування найнавантаженіших компонентів без перебудови всього циклу.

2.7 Висновки до розділу 2

У розділі виконано концептуальне проєктування мікросервісної FEA-платформи як керованого конвеєра повного циклу, що функціонує у розподіленому середовищі. Вихідною постановкою визначено ключові архітектурні виклики FEA-платформ: необхідність роботи з великими артефактами (сітки, результати, дані візуалізації), синхронізацію стадій конвеєра при асинхронному виконанні та забезпечення оперативного інформування користувача про стан і готовність результатів. На основі цих викликів сформовано функціональні вимоги у вигляді узгодженої моделі прецедентів, що відображає взаємодію користувача з платформою як єдиною системою.

Нефункціональні вимоги систематизовано через властивості масштабованості, відмовостійкості, керованості затримок, безпеки доступу, відтворюваності та підтримуваності. Концептуально обґрунтовано подієво-орієнтований підхід взаємодії компонентів як механізм слабкого зв’язування стадій конвеєра та підтримки асинхронних довготривалих обчислень. Оскільки в подієвих системах можливе повторне надходження повідомлень, до архітектури включено перевірку станів стадій, кореляцію подій і базові механізми ідемпотентного опрацювання завершених операцій. Узгодження подій, метаданих та артефактів відбувається за принципом eventual consistency.

Для ефективної роботи з великими даними закладено принцип розділення керування станом і передавання артефактів: метадані й події використовуються для сигналізації та координації, тоді як важкі результати зберігаються у спеціалізованому сховищі та передаються між компонентами через посилання.

Платформу формалізовано як «чорну скриньку» (IDEF0 A-0), що визначає її входи, виходи, керуючі впливи та механізми виконання. На основі цього виконано сервісну декомпозицію FEA-циклу та уточнено архітектуру через виділення клієнтського і серверного контурів: клієнтський контур відповідає за подання, моніторинг і 3D-перегляд, тоді як серверний контур реалізує життєвий цикл задачі та асинхронне виконання стадій. Доступ клієнта до серверного контуру концептуально організовано через API Gateway (edge-компонент) як єдину точку входу, що відокремлює клієнт від внутрішньої структури сервісів; роль агрегатора (BFF/Aggregator) визначено як уніфікований інтерфейс для UI та точку агрегації станів і подій.

Динаміку роботи платформи описано через поєднання моделей взаємодії: діаграма послідовності відображає наскрізний життєвий цикл задачі та подієву синхронізацію стадій, а діаграми діяльності деталізують внутрішню логіку Preprocessor, Processor/Workers та Postprocessor як автономних стадій конвеєра, що реагують на події, формують артефакти та публікують повідомлення про готовність.

Сформована концептуальна модель визначає цільову архітектуру платформи та є ширшою за межі експериментального прототипу. На рівні реалізації перевіряється базовий сценарій параметрично заданих двовимірних і тривимірних задач Пуассона, ручне масштабування обчислювальних виконавців, JSON-контракти артефактів і клієнтська візуалізація засобами Three.js. Імпорт довільних CAD-моделей, автоматичне масштабування, повноцінні повторні спроби, канали відхилених повідомлень та розподілене об’єктне сховище залишаються напрямами подальшого розвитку.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПОДІЄВО-ОРІЄНТОВАНОЇ МІКРОСЕРВІСНОЇ FEA-ПЛАТФОРМИ

3.1 Архітектура реалізації та склад компонентів

Перехід від концептуальної моделі, розглянутої в розділі 2, до програмної реалізації потребує конкретизації двох аспектів: складу клієнтського й серверного контурів та розподілу відповідальності між ними, а також способу реалізації подієвої координації конвеєра й обміну великими артефактами. У реалізованій платформі етапи підготовки даних, чисельного розв’язання та післяоброблення представлені як автономні компоненти з незалежним життєвим циклом розгортання і масштабування, а взаємодія між ними організована через брокер подій та уніфіковану структуру повідомлень, що забезпечує слабке зв’язування та підтримку причинно-наслідкового відстеження задач [46].

Реалізація вхідного та інфраструктурного контурів платформи спирається на стандартну семантику HTTP [60], Traefik [61], Keycloak [62], Apache Kafka [63], PostgreSQL [64], MinIO [65], Redis [66], Celery [67], Docker [68] і Kubernetes [69]. Клієнтський та прикладний рівні реалізовано з використанням Angular [70], Node.js [71], Socket.IO [72], Three.js [73], FastAPI [74], Gmsh [75], NumPy [76] і SciPy [77]. Деталі програмної реалізації й експериментального прототипу раніше оприлюднено автором у праці [78].

Архітектурно платформа зберігає принциповий поділ на клієнтський та серверний контури. Клієнтський контур забезпечує введення параметрів, моніторинг стану задач, перегляд часової шкали стадій і журналу подій, а також 3D-візуалізацію результатів як окрему підсистему інтерфейсу. Серверний контур, у свою чергу, реалізує життєвий цикл задачі та виконання конвеєра, включно зі зберіганням метаданих (статуси/часові мітки) і обміном артефактами. Узгодженість між стадіями підтримується подієво-орієнтованою моделлю, а реактивне інформування користувача забезпечується агрегатором через WebSocket (push-модель).

Клієнтський контур реалізовано як окремий односторінковий вебзастосунок, що виконується поза межами Kubernetes-кластера та взаємодіє із серверним контуром через його зовнішню точку входу [70]. У кластері розгорнуто BFF/Aggregator, обчислювальні мікросервіси та інфраструктурні компоненти, тоді як API Gateway/Ingress Controller формує межу доступу до серверного контуру.

Для фіксації складу компонентів на реалізаційному рівні та їхніх інтерфейсів у таблиці 3.1 подано відповідність «компонент → роль → інтерфейс → реалізаційний інструмент». Таблиця відображає лише ті елементи, які є необхідними для відтворення реалізованого конвеєра (FEA-конвеєра) та подальших експериментів масштабування.

Таблиця 3.1 – Склад компонентів реалізованої платформи та їхня роль

Компонент	**Контур**	**Роль у платформі**	**Основні інтерфейси**	**Реалізація у прототипі**
Web UI + модуль 3D-візуалізації	клієнтський	Створення задач, моніторинг, перегляд часової шкали/подій, 3D-візуалізація результатів	HTTP(S) до gateway, WebSocket до BFF, завантаження webPackage	Angular + Three.js, розгортається поза Kubernetes-кластером
API Gateway (edge)	серверний (вхідний контур)	Маршрутизація зовнішніх HTTP(S)- і WebSocket-з'єднань до BFF та інших доступних сервісів кластера	HTTP(S) reverse-proxy	Traefik
Провайдер ідентичності (IdP)	серверний (вхідний контур)	Автентифікація користувачів, видача токенів доступу	OAuth 2.0 / OpenID Connect	OIDC-сумісний IdP
BFF/Aggregator	серверний	REST API для UI, агрегація станів/стадій/подій, push-оновлення	REST, WebSocket, publish у broker, читання метаданих/артефактів	Node.js + Socket.IO
Message Broker	серверний (інфра)	Подієва координація конвеєра, слабке зв'язування стадій	Канали подій, групи споживачів, упорядкування за ключем	Apache Kafka
Preprocessor	серверний	Генерація/підготовка даних (mesh + jobSpec), старт стадії preprocess	Споживач подій, запис артефактів	FastAPI + Gmsh
Processor (споживач)	серверний	Споживання подій готовності preprocess, постановка задач у чергу воркерів	Споживач подій, поставлення завдання в чергу	FastAPI + Celery client
Processor Workers	серверний	Виконання чисельного розв'язання (solve) та публікація прогресу/результату	Виконавець черги, publish подій, запис артефактів	Celery workers (масштабована множина)
Postprocessor	серверний	Трансформація solution -> webPackage, завершення конвеєра	Споживач подій, запис артефактів	FastAPI

Продовження таблиці 3.1

Компонент	Контур	Роль у платформі	Основні інтерфейси	Реалізація у прототипі
Metadata storage	серверний (інфра)	Метадані задачі, стадії, часові мітки для E2E/часових шкал	SQL	PostgreSQL
Object storage	серверний (інфра)	Зберігання важких артефактів (mesh/solution/webPackage)	S3-подібний API	MinIO (S3-compatible)
Queue backend	серверний (інфра)	Транспорт/бекенд для worker-моделі	Протокол Redis	Redis (для Celery)

Надана структура відображає також принцип розділення керівний контур (control plane) та контур даних (data plane): метадані й події формують керування конвеєром (стани/стадії/прогрес), тоді як великі артефакти (сітка, розв’язок, пакет візуалізації) зберігаються в об’єктному сховищі та передаються між стадіями через посилання. Такий підхід зменшує навантаження на брокер подій і робить виконання конвеєра масштабованим у межах кластеру.

Загальну структуру компонентів подієво-орієнтованої моделі FEA-конвеєра відображено на рисунку 3.1.

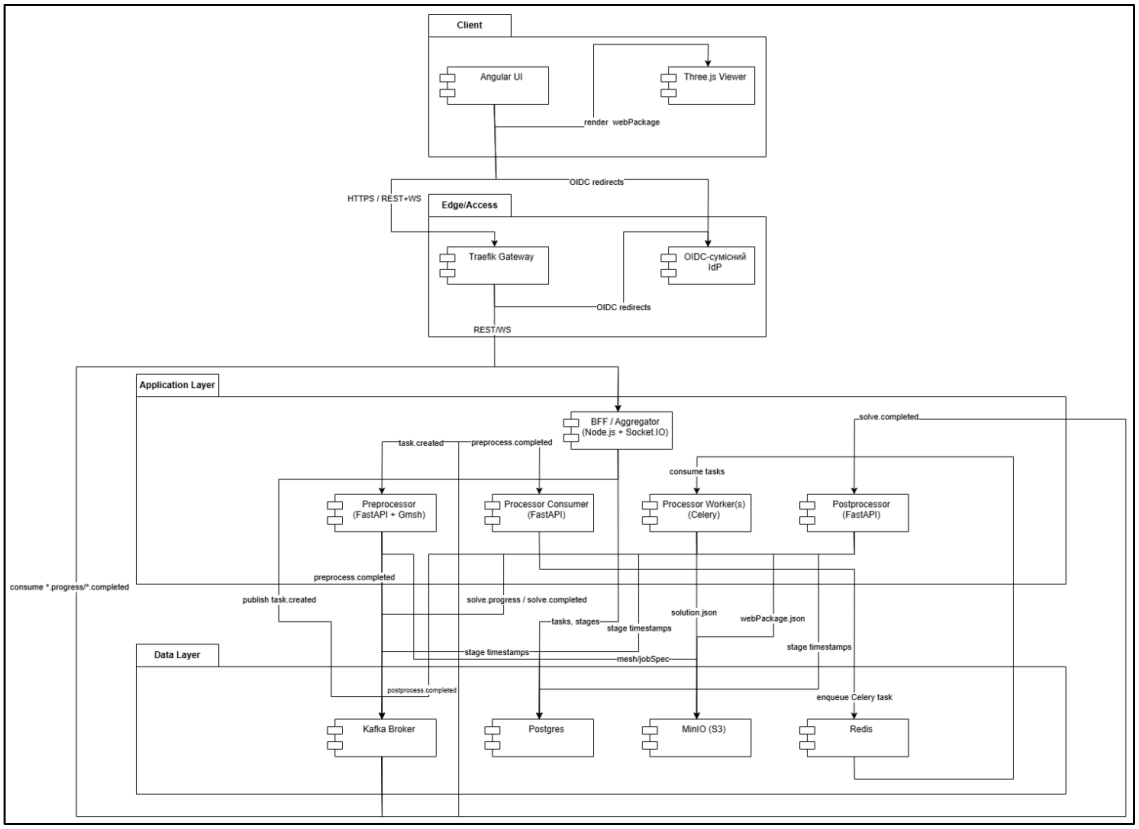


Рисунок 3.1 – Діаграма компонентів подієво-орієнтованої моделі FEA-конвеєра

Порівняно з початковою архітектурною моделлю, у якій як один із можливих патернів розглядалося окреме NoSQL-сховище для кожного сервісу, у прототипі структуру зберігання змінено відповідно до характеру даних. Компактні транзакційні метадані задач і стадій зберігаються у PostgreSQL, великі проміжні та кінцеві артефакти – у MinIO, а Redis використовується як транспорт і backend внутрішньої черги Processor.

3.2 Вхідний контур: автентифікація та маршрутизація запитів

Вхідний контур реалізованої платформи виконує дві взаємопов'язані функції: забезпечує єдину точку входу для клієнтського контуру та маршрутизацію запитів до серверних компонентів; забезпечує ідентифікацію користувача для доступу до задач, стадій та артефактів. У межах обраного підходу автентифікація організована через OAuth 2.0 / OpenID Connect-сумісний провайдер ідентичності, а для односторінкового вебклієнта застосовується схема Authorization Code Flow із PKCE [55–59, 62]. Це дозволяє отримати маркер доступу (access token) без зберігання клієнтського секрету у браузері та забезпечує стандартизовану інтеграцію з серверним API.

Маршрутизація запитів виконується через API Gateway (edge-компонент). На реалізаційному рівні gateway виконує роль reverse-proxy: термінує зовнішні HTTP(S)-запити та переспрямовує їх до BFF/Aggregator як публічного API платформи [53, 61]. Це узгоджується з принципом ізоляції внутрішньої структури мікросервісів від клієнта: UI не взаємодіє напряму з Preprocessor/Processor/Postprocessor, а працює з єдиною точкою доступу, яка надає уніфікований інтерфейс для створення задач, перегляду станів і завантаження результатів.

Контроль доступу на рівні серверного контуру базується на ідентифікаторі користувача з токена доступу. BFF/Aggregator валідує токен та формує контекст користувача для виконання запитів (див. рис. 3.2).

```
async function verifyBearerToken(authHeader) {
   if (!authHeader?.startsWith("Bearer ")) return null;
   const token = authHeader.slice("Bearer ".length).trim();
   const { payload } = await jwtVerify(token, JWKS, { algorithms: ["RS256"] }); return payload;
}
function requireAuth() {
   return async (req, res, next) => {
      try {
         const payload = await verifyBearerToken(req.headers.authorization);
         if (!payload?.sub) return res.status(401).json({ message: "Unauthorized" });
         req.user = payload; next();
      } catch (e) {
         return res.status(401).json({ message: "Unauthorized", error: String(e?.message || e) });
      }
   }; }
```

Рисунок 3.2 – Перевірка JWT у BFF

Надалі доступ до ресурсів платформи (задачі, стадії, артефакти) обмежується перевіркою приналежності taskId відповідному користувачу: це забезпечує ізоляцію даних між користувачами без потреби перевіряти авторизацію в кожному внутрішньому сервісі конвеєра. При цьому внутрішні стадії (підготовка даних, чисельне розв’язання та післяоброблення) працюють з taskId як з ключем життєвого циклу та використовують його для кореляції подій і посилань на артефакти.

Окремим елементом вхідного контуру є реактивний канал інформування користувача. Для задач FEA характерна тривалість виконання, що не вкладається у межі одного синхронного HTTP-запиту. Тому BFF/Aggregator реалізує push-оновлення через WebSocket, транслюючи зміни стану задачі (стадії, прогрес, готовність результатів) у клієнтський контур. Такий підхід відповідає вимогам інформування користувача у реальному часі, які були сформульовані на концептуальному рівні та відображені як механізм взаємодії агрегатора з UI.

3.3 Контракти взаємодії: REST API та моделі даних

Синхронну взаємодію між клієнтським інтерфейсом і серверним контуром реалізовано через REST API компонента BFF/Aggregator [52]. На відміну від

внутрішніх стадій FEA-конвеєра, які обмінюються асинхронними подіями, REST-інтерфейс застосовується для короткотривалих операцій: створення задачі, отримання її метаданих і стану стадій, а також завантаження підготовленого пакета візуалізації. Такий поділ дає змогу не утримувати HTTP-з'єднання протягом виконання чисельного розрахунку: після створення задачі клієнт одразу отримує її ідентифікатор, а подальша обробка відбувається асинхронно.

Семантика методів і кодів відповіді ґрунтується на стандартній моделі HTTP [60]. Метод POST використовується для створення нового ресурсу задачі, тоді як GET – для отримання актуального представлення вже наявного ресурсу. Результат обробки запиту передається за допомогою відповідного коду стану: успішні операції належать до класу 2xx, помилки вхідних даних або доступу – до 4xx, а внутрішні збої серверного контуру – до 5xx [60].

У реалізованому BFF маршрути визначено засобами серверного фреймворку відповідно до HTTP-методу та URI [78]. Обробник маршруту виконує валідацію параметрів, перевіряє доступ користувача, звертається до сховища метаданих або артефактів і формує HTTP-відповідь. Зовнішні шляхи використовують позначення {id} для параметра ідентифікатора задачі, тоді як у програмному коді маршрутизатора цьому параметру відповідає форма :id.

У фінальній реалізації BFF надає п'ять REST-операцій. Три з них безпосередньо підтримують проходження FEA-конвеєра: створення задачі, отримання стадій із часовими мітками та завантаження пакета візуалізації. Додаткові операції надають список останніх задач користувача і детальне представлення вибраної задачі. Узагальнення реалізованих контрактів наведено в таблиці 3.2.

Таблиця 3.2 – REST-контракти BFF/Aggregator

Метод і шлях	**Призначення**	**Вхідні дані**	**Успішна відповідь**	**Основні помилки**
POST /tasks	Створення задачі та ініціювання FEA-конвеєра	JSON: scenarioId, meshSize	201 Created, { "taskId": "..." }	400 – невалідні параметри; 401 – відсутній або невалідний токен

Продовження таблиці 3.2

Метод і шлях	**Призначення**	**Вхідні дані**	**Успішна відповідь**	**Основні помилки**
GET /tasks	Отримання списку останніх задач користувача	–	200 OK, масив задач	401 – помилка автентифікації
GET /tasks/{id}	Отримання детальної інформації про задачу	параметр шляху id	200 OK, об’єкт задачі	404 – задача не знайдена або не належить користувачу
GET /tasks/{id}/stages	Отримання станів і часових міток стадій	параметр шляху id	200 OK, масив записів стадій	404 – задача не знайдена або немає доступу
GET /tasks/{id}/web-package	Завантаження даних для 3D-візуалізації	параметр шляху id	200 OK, JSON-пакет візуалізації	404 – задача відсутня або пакет ще не сформований

Усі операції виконуються в контексті автентифікованого користувача. Ідентифікатор користувача не приймається у тілі запиту, а визначається BFF із поля sub перевіреного маркера доступу. Під час читання задачі або пов’язаних із нею даних BFF виконує запит із двома умовами – ідентифікатором задачі та ідентифікатором користувача (див. рис. 3.3). Такий підхід запобігає доступу до чужих задач через підміну параметра {id}.

```
const userId = req.user.sub;
const taskId = req.params.id
const { rows } = await db.query("SELECT id FROM tasks WHERE  id=$1 AND user_id=$2 LIMIT 1",
[taskId, userId]
);
```

Рисунок 3.3 – Перевірка належності задачі автентифікованому користувачу

Якщо запис із заданою парою taskId і userId відсутній, сервер повертає код 404 Not Found. Однакове представлення ситуацій «ресурс не існує» і «ресурс належить іншому користувачу» не розкриває клієнту інформацію про наявність чужої задачі. Після успішної перевірки BFF виконує відповідну операцію зі сховищем метаданих або об’єктним сховищем.

Основним командним контрактом є POST /tasks. Тіло запиту містить ідентифікатор параметризованої постановки та розмір сітки (див. рис. 3.4). На

рівні BFF перевіряється наявність непорожнього рядкового значення scenarioId, а meshSize перетворюється на число і має бути скінченним додатним значенням. Після валідації генерується унікальний taskId (рис. 3.5), створюється запис у таблиці задач і початковий запис стадії created, після чого публікується подія створення задачі. Таким чином, REST-операція завершується після реєстрації задачі та ініціювання конвеєра, а не після виконання чисельного розрахунку.

```
POST /tasks

{
  "scenarioId": "poisson_cube_mms_vbc",
  "meshSize": 0.2
}
```

Рисунок 3.4 – POST /tasks запит

```
HTTP/1.1 201 Created
{
 "taskId": "01KV5S5NHY1TQZC05ZQE4RKQXR"
}
```

Рисунок 3.5 – POST /tasks відповідь

Отриманий taskId виступає наскрізним ідентифікатором у REST-контрактах, записах бази даних, подіях брокера та ключах артефактів. Завдяки цьому клієнт може одразу підписатися на оновлення задачі, а внутрішні компоненти – корелювати результати окремих стадій без передавання повного стану через синхронний API.

Операція GET /tasks повертає впорядкований за часом створення перелік останніх задач поточного користувача. Вибірка обмежена п’ятдесятьма записами, що запобігає необмеженому зростанню обсягу відповіді. Скорочене представлення задачі містить ідентифікатор, сценарій, параметр сітки, поточний статус і часові мітки. Операція GET /tasks/{id} використовується для отримання детального представлення окремої задачі.

Для моніторингу проходження конвеєра використовується GET /tasks/{id}/stages. Контракт повертає впорядкований масив стадій із їхнім станом,

часом початку, часом завершення та номером спроби. Ці дані використовуються клієнтським інтерфейсом для побудови часової шкали, а в експериментальній частині – для обчислення тривалості окремих стадій і повного часу E2E (рис. 3.6).


```
[
  {
    "stage": "created",
    "status": "COMPLETED",
    "started_at": "2026-06-15T13:57:38.508Z",
    "completed_at": null,
    "attempt": 1
  },
  {
    "stage": "preprocess",
    "status": "COMPLETED",
    "started_at": "2026-06-15T13:57:38.517Z",
    "completed_at": "2026-06-15T13:57:38.646Z",
    "attempt": 1
  },
  {
    "stage": "solve",
    "status": "COMPLETED",
    "started_at": "2026-06-15T13:57:38.647Z",
    "completed_at": "2026-06-15T13:57:38.842Z",
    "attempt": 1
  },
  {
    "stage": "postprocess",
    "status": "COMPLETED",
    "started_at": "2026-06-15T13:57:38.833Z",
    "completed_at": "2026-06-15T13:57:38.849Z",
    "attempt": 1
  }
]
```


Рисунок 3.6 – Відповідь GET /tasks/{id}/stages

Наявність окремого контракту стадій дає змогу не змішувати поточний агрегований статус задачі з детальною історією її проходження. Наприклад, загальний статус може бути представлений одним значенням, тоді як масив стадій зберігає інформацію, необхідну для аналізу затримок і повторних спроб.

Операція GET /tasks/{id}/web-package надає клієнтському модулю візуалізації підготовлений JSON-артефакт. Перед зверненням до об’єктного сховища BFF перевіряє належність задачі користувачу, після чого формує ключ

артефакта за прийнятою схемою. Якщо постобробку ще не завершено і файл відсутній, сервер повертає повідомлення про те, що пакет візуалізації ще не готовий.

REST API виконує роль стабільної зовнішньої межі між клієнтом та асинхронним FEA-конвеєром. Клієнт не отримує доступу до внутрішніх адрес Preprocessor, Processor або Postprocessor і не керує переходами між стадіями безпосередньо. Відповідальність BFF полягає у прийманні команд користувача, перевірці доступу, наданні метаданих і результатів, тоді як внутрішня координація стадій реалізується подієвими контрактами, які розглядаються в наступному підрозділі.

3.4 Подієвий контур: брокер повідомлень, канали подій та уніфікована структура повідомлень

Внутрішню координацію стадій реалізованого FEA-конвеєра побудовано за подієво-орієнтованою моделлю. На відміну від REST API, який формує зовнішню синхронну межу між клієнтським застосунком і BFF/Aggregator, взаємодія між Preprocessor, Processor та Postprocessor не потребує безпосередніх викликів одного сервісу іншим. Кожна стадія отримує подію, що засвідчує готовність необхідних вхідних даних, виконує власну операцію та публікує нову подію про її завершення або помилку.

Як транспорт подій у платформі використано Apache Kafka [11–14, 63]. Вибір брокера зумовлений потребою в асинхронній передачі повідомлень, підтримці груп споживачів, збереженні подій протягом визначеного періоду та можливості незалежного масштабування компонентів-споживачів. Подієва модель зменшує часову зв’язність сервісів: продюсер публікує повідомлення без необхідності очікувати завершення роботи наступної стадії, а споживач обробляє його відповідно до власної доступності та поточного навантаження [11–14].

Події в Kafka організовано за окремими каналами подій (topics), назви яких відображають стадію конвеєра та характер зміни її стану. Для завершення стадії

використовується суфікс completed, для помилки – failed, а для проміжного інформування про виконання обчислювальної стадії – progress. Початковою подією життєвого циклу є task.created, яка публікується BFF після успішної реєстрації задачі в сховищі метаданих (рис. 3.7).

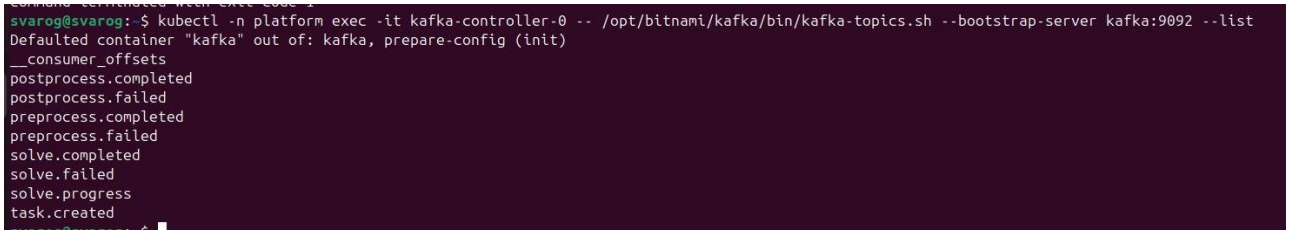


Рисунок 3.7 – Список подій в Kafka

Для подій однієї задачі як ключ Kafka-повідомлення використовується taskId. За умови незмінної конфігурації кількості партицій однаковий ключ спрямовує повідомлення відповідного каналу подій до тієї самої партиції. Це забезпечує збереження порядку повідомлень у межах конкретної партиції, що важливо для послідовної обробки подій одного типу для однієї задачі [14, 63]. Водночас глобальний порядок між різними каналами подій не гарантується. Тому компоненти не повинні покладатися лише на фактичний час надходження повідомлень, а мають перевіряти стан відповідної стадії у сховищі метаданих і коректно обробляти події, що надійшли повторно або із затримкою.

Структуру каналів подій, продюсерів і споживачів подієвого контуру наведено в таблиці 3.3.

Таблиця 3.3 – Структура каналів подій Kafka реалізованої FEA-платформи

Канал подій	**Продюсер**	**Основні споживачі**	**Призначення та ключові дані**
task.created	BFF/Aggregator	Preprocessor	Ініціювання FEA-конвеєра; taskId, scenarioId, meshSize, контекст користувача
preprocess.completed	Preprocessor	Processor consumer	Завершення генерації сітки, meshArtifactRef, jobSpecRef, параметри постановки
preprocess.failed	Preprocessor	BFF/Aggregator	Повідомлення про помилку стадії підготовки даних
solve.progress	Processor/Workers	BFF/Aggregator	Поточний прогрес чисельного розв’язання у відсотках

Продовження таблиці 3.3

Канал подій	**Продюсер**	**Основні споживачі**	**Призначення та ключові дані**
solve.completed	Processor/Workers	Postprocessor, BFF/Aggregator	Завершення розрахунку; resultArtifactRef, jobSpecRef
solve.failed	Processor/Workers	BFF/Aggregator	Повідомлення про помилку чисельного розв'язання
postprocess.completed	Postprocessor	BFF/Aggregator	Завершення постобробки, webPackageRef, підсумкові метрики та статистики
postprocess.failed	Postprocessor	BFF/Aggregator	Повідомлення про помилку підготовки даних для візуалізації

Коли одна подія повинна бути незалежно опрацьована декількома компонентами, відповідні споживачі використовують різні групи. Наприклад, подія solve.completed має бути отримана Postprocessor для запуску наступної стадії та BFF/Aggregator для оновлення стану користувацького інтерфейсу. Використання різних груп забезпечує доставку повідомлення кожному функціональному підписнику, тоді як декілька екземплярів одного сервісу в межах спільної групи розподіляють повідомлення між собою.

Як показано в Додатку А, синхронна взаємодія завершується після реєстрації задачі та повернення клієнту taskId, тоді як подальше проходження стадій координується подіями Kafka. Великі артефакти не передаються через брокер, а зберігаються в MinIO; події містять лише посилання на відповідні об'єкти. Внутрішню чергу Redis/Celery використано виключно в межах Processor для розподілу обчислювальних задач між масштабованими виконавцями. BFF споживає користувацькі події solve.*, postprocess.* і preprocess.failed та транслює їх до авторизованої Socket.IO-кімнати відповідної задачі.

Узгодженість структури повідомлень забезпечується використанням уніфікованої структури повідомлення (event envelope) [78]. Вона відокремлює службові атрибути події від предметних даних конкретної стадії. Завдяки цьому всі сервіси однаково опрацьовують ідентифікацію події, версію контракту, час виникнення, джерело, кореляційні атрибути та ключ ідемпотентності, тоді як поле data має структуру, специфічну для типу повідомлення.

Уніфікована структура повідомлення виконує декілька функцій. Поле eventId однозначно ідентифікує конкретне повідомлення. Поле type визначає його

семантику, наприклад preprocess.completed або solve.failed. Поля version і schemaVersion дозволяють еволюціонувати контракти без одночасного оновлення всіх компонентів: перше характеризує версію типу події, а друге – версію структури її предметного навантаження. Поле occurredAt містить момент формування події, а producer – ідентифікатор сервісу, який її опублікував.

Зв'язок повідомлення з життєвим циклом задачі задається об'єктом subject, що містить taskId та, за наявності, ідентифікатор користувача. Поле correlationId об'єднує всі повідомлення одного запуску в спільний кореляційний ланцюжок і в реалізованому прототипі відповідає taskId. Поле causationId посилається на подію, яка безпосередньо стала причиною формування поточного повідомлення. Наприклад, для solve.completed таким значенням може бути eventId події preprocess.completed, на основі якої було запущено розв'язання.

Склад службових і предметних полів уніфікованої структури повідомлення наведено в таблиці 3.4.

Таблиця 3.4 – Поля уніфікованої envelope-структури події

Поле	**Тип**	**Призначення**
eventId	UUID/string	Унікальний ідентифікатор екземпляра події
type	string	Тип події, наприклад solve.completed
version	integer	Версія контракту події
schemaVersion	integer	Версія структури поля data
occurredAt	ISO 8601	Час формування події
producer	string	Ідентифікатор сервісу-продюсера
subject.taskId	string	Ідентифікатор задачі
subject.userId	string/null	Ідентифікатор користувача
correlationId	string	Ідентифікатор кореляційного ланцюжка
causationId	string/null	Ідентифікатор події-причини
idempotencyKey	string	Ключ запобігання повторному виконанню
trace	object/null	Дані розподіленого трасування
data	object	Payload події
meta.contentType	string	Тип вмісту поля data

Успішний шлях обробки Kafka-повідомлення реалізовано з ручною фіксацією offset після виконання відповідної координаційної операції. Тому в разі

збою між виконанням операції та commit повідомлення може бути доставлене повторно, що створює поведінку, близьку до at-least-once. Водночас у прототипі offset помилкового повідомлення також фіксується для запобігання циклічному опрацюванню некоректних даних. Отже, автоматичні повтори та окремий канал відхілених повідомлень у поточній реалізації не передбачені. Тому сам факт отримання повідомлення не повинен безумовно запускати повторне виконання ресурсомісткої стадії.

Для ідентифікації повторно отриманого повідомлення в envelope передбачено поле idempotencyKey, що формується з ідентифікатора задачі, типу події та версії контракту. У поточному прототипі воно є частиною контракту повідомлення, тоді як фактичне запобігання повторному виконанню завершеної стадії здійснюється шляхом перевірки її стану в PostgreSQL. Перед початком виконання сервіс перевіряє поточний стан відповідної стадії. Якщо вона вже має термінальний статус COMPLETED, повторне повідомлення не призводить до повторної генерації сітки, обчислення розв’язку або формування пакета візуалізації. Якщо стадія не була завершена, сервіс реєструє початок стадії із заданим номером спроби (у поточному сценарії використовується значення attempt = 1, а структура таблиці передбачає можливість подальшого додавання повторних спроб) та продовжує обробку. Таким чином, у прототипі реалізовано базовий механізм ідемпотентного опрацювання вже завершених стадій: повторне повідомлення не запускає операцію, якщо в PostgreSQL наявний відповідний статус COMPLETED. Водночас перевірка стану й реєстрація початку стадії не є атомарною операцією, тому механізм не гарантує взаємного виключення при одночасному надходженні кількох однакових повідомлень.

На рисунку 3.8 наведено приклад події preprocess.completed, отриманої з Apache Kafka.

Важливою властивістю подієвого контуру є відокремлення керування процесом від передавання великих даних. Kafka використовується для передачі легких повідомлень, що описують зміну стану стадії та містять посилання на результати. Сітка, чисельний розв’язок і пакет візуалізації не включаються

безпосередньо до подій, а зберігаються в об’єктному сховищі. Це зменшує обсяг повідомлень, спрощує їх повторне споживання та запобігає використанню брокера як файлового сховища.


```
{
  "eventId": "eafb273b-b903-4296-a1e0-e76cf531bce0",
  "type": "preprocess.completed",
  "version": 1,
  "schemaVersion": 1,
  "occurredAt": "2026-06-16T07:04:58.315222+00:00",
  "producer": "preprocessor-service",
  "subject": {
    "taskId": "01KV7KYRG2RMDNJFJADB9WHJPN",
    "userId": "67cae3c5-7e25-4855-b1d6-73761c11eb29"
  },
  "correlationId": "01KV7KYRG2RMDNJFJADB9WHJPN",
  "causationId": null,
  "idempotencyKey": "01KV7KYRG2RMDNJFJADB9WHJPN:preprocess.completed:1",
  "trace": null,
  "data": {
    "scenarioId": "poisson_cube_mms_vbc",
    "meshSize": 0.1,
    "dimension": 3,
    "meshArtifactRef": {
      "bucket": "fea-artifacts",
      "key": "tasks/01KV7KYRG2RMDNJFJADB9WHJPN/mesh/mesh.msh",
      "etag": "07eebfd22bbed1ee270a8b96bc03a574",
      "sizeBytes": 240683,
      "contentType": "application/octet-stream"
    },
    "jobSpecRef": {
      "bucket": "fea-artifacts",
      "key": "tasks/01KV7KYRG2RMDNJFJADB9WHJPN/job/jobSpec.json",
      "etag": "7f1be54eeed05a2feaf5ccbbe922ba4c",
      "sizeBytes": 279,
      "contentType": "application/json"
    }
  },
  "meta": {
    "contentType": "application/json"
  }
}
```


Рисунок 3.8 – Подія preprocess.completed

Отже, подієвий контур забезпечує асинхронну координацію стадій, слабке зв’язування компонентів, можливість незалежного масштабування споживачів і реактивне інформування клієнтського застосунку. Водночас надійність його

роботи ґрунтується не лише на можливостях брокера, а й на стабільних контрактах, кореляції повідомлень, ідемпотентності обробників та збереженні стану стадій. Оскільки події містять лише посилання на великі результати, наступним кроком є формалізація контрактів артефактів і правил їх зберігання в об’єктному сховищі.

3.5 Контракти артефактів та організація їх зберігання

Подієвий контур, описаний у попередньому підрозділі, передає повідомлення про зміну стану задачі, але не призначений для транспортування значних обсягів чисельних даних. Скінченно-елементна сітка, масиви чисельного розв’язку та дані для тривимірної візуалізації можуть істотно перевищувати розмір службових повідомлень. Їх безпосереднє включення до Kafka-подій збільшувало б мережеве навантаження, ускладнювало повторну доставку повідомлень і перетворювало б брокер на неспеціалізоване файлове сховище.

Тому використано розділення між керуванням виконанням і передаванням даних [11]. Події, стани й часові мітки утворюють керівний контур, тоді як великі проміжні та кінцеві результати формують контур даних. У подіях передаються ідентифікатор задачі, службові метадані та посилання на артефакти, а самі файли зберігаються в S3-сумісному об’єктному сховищі.

Для реалізації сховища артефактів використано MinIO, що підтримує сумісний із Amazon S3 програмний інтерфейс [65]. Об’єктна модель зберігання є придатною для конвеєра, оскільки кожен артефакт ідентифікується комбінацією назви bucket та ключа, а сервіси можуть записувати й отримувати файли без використання спільної файлової системи контейнерів. Це зберігає автономність мікросервісів і дозволяє масштабувати їх незалежно від фізичного розміщення pod у кластері [11, 65].

Усі об’єкти однієї задачі організовуються за фіксованою схемою ключів: tasks/{taskId}/{artifact-group}/{file-name}.

Використання taskId як першої змінної частини ключа формує окремий логічний простір для кожної задачі. Наступний сегмент позначає групу або стадію формування даних, а останній – конкретний файл. Така структура робить розташування артефактів передбачуваним для всіх компонентів конвеєра і не потребує прямого узгодження шляхів між Preprocessor, Processor та Postprocessor.

У межах реалізованого FEA-конвеєра використовуються чотири основні контракти артефактів: скінченно-елементна сітка, опис постановки задачі, чисельний розв’язок і пакет для клієнтської візуалізації [78]. Їх призначення, формати й напрями передавання систематизовано в таблиці 3.5.

Таблиця 3.5 – Контракти артефактів реалізованого FEA-конвеєра

Артефакт	**Ключ у сховищі**	**Формат**	**Компонент-продюсер**	**Основні споживачі**
Скінченно-елементна сітка	tasks/{taskId}/mesh/mesh.msh	Gmsh MSH	Preprocessor	Processor
Опис постановки задачі	tasks/{taskId}/job/jobSpec.json	JSON	Preprocessor	Processor
Чисельний розв’язок	tasks/{taskId}/results/solution.json	JSON	Processor / Workers	Postprocessor
Пакет візуалізації	tasks/{taskId}/web/webPackage.json	JSON	Postprocessor	BFF, клієнтський модуль 3D-візуалізації

Першим артефактом, який формується після створення задачі, є скінченно-елементна сітка mesh.msh. Preprocessor генерує її відповідно до вибраного сценарію та параметра дискретизації meshSize. Формат MSH використовується як машинний контракт між стадією підготовки даних і чисельним розв’язувачем. Сітка не передається в повідомленні preprocess.completed, сама подія містить лише посилання meshArtifactRef, за яким Processor завантажує відповідний об’єкт зі сховища.

Окремо від сітки Preprocessor формує jobSpec.json. Цей артефакт фіксує постановку задачі у відтворюваній формі та пов’язує користувацькі параметри з

даними, необхідними наступним стадіям. До складу опису належать ідентифікатор задачі, ідентифікатор сценарію, параметри дискретизації, тип задачі, характеристика граничних умов і посилання на сформовані вхідні артефакти. Зберігання постановки в окремому незмінному JSON-документі дозволяє відтворити вхідні умови розрахунку незалежно від поточного стану користувацького інтерфейсу.

Вміст артефакта jobSpec.json наведено на рисунку 3.9.


```
{
 "taskId": "01KV7KYRG2RMDNJFJADB9WHJPN",
 "scenarioId": "poisson_cube_mms_vbc",
 "meshSize": 0.1,
 "dimension": 3,
 "meshRef": {
  "bucket": "fea-artifacts",
  "key": "tasks/01KV7KYRG2RMDNJFJADB9WHJPN/mesh/mesh.msh"
 },
 "problemType": "poisson",
 "boundary": {
  "dirichletGroup": "DIRICHLET"
 }
}
```


Рисунок 3.9 – Артефакт jobSpec.json

Після отримання події preprocess.completed Processor завантажує mesh.msh і jobSpec.json, формує дискретну систему та виконує чисельне розв’язання. Результатом стадії є solution.json. У реалізованому прототипі він містить геометричні дані сітки, топологію елементів, масив значень шуканого скалярного поля та підсумкові характеристики чисельного розв’язувача.

До геометричної частини входять координати вузлів points і масив індексів елементів triangles. Поле values містить значення чисельного розв’язку у вузлах. Окремий блок summary містить характеристики розміру задачі, часові метрики складання й розв’язання системи, статистики чисельного поля та, для верифікаційного MMS-сценарію, значення похибок maxAbsError і l2Error.

Таким чином, solution.json виконує дві функції. По-перше, він є предметним результатом обчислювальної стадії, який використовується

Postprocessor. По-друге, він фіксує внутрішні метрики Processor, необхідні для контролю коректності й подальшого аналізу продуктивності.

Скорочений вміст артефакта solution.json наведено на рисунку 3.10.


```
{
  "taskId": "01KV7KYRG2RMDNJFJADB9WHJPN",
  "points": [
    [0.0, 0.0, 1.0],
    [0.0, 0.0, 0.0],
    [0.0, 1.0, 1.0]
  ],
  "triangles": [
    [709, 735, 721], [717, 732, 712]
  ],
  "values": [0.0, 0.0, 0.0024],
  "summary": {
    "dimension": 3,
    "nPoints": 1201,
    "nTetra": 4979,
    "nTriangles": 1470
  }
}
```


Рисунок 3.10 – Артефакт solution.json

Postprocessor отримує подію solve.completed, завантажує solution.json і перетворює його на формат, безпосередньо придатний для клієнтського модуля візуалізації. Вихідним артефактом є webPackage.json. Його структура відокремлює специфіку чисельного розв’язувача від потреб вебклієнта: UI не повинен знати внутрішню організацію Processor або самостійно перетворювати формат чисельного розв’язувача.

Пакет візуалізації містить геометрію, топологію та скалярне поле, необхідні для побудови сцени засобами Three.js. У реалізованому клієнтському модулі використовуються координати вузлів, трикутні елементи та значення поля у вузлах, на основі яких формується кольорове представлення результату. До пакета також включається блок summary, завдяки чому користувацький інтерфейс може одночасно відобразити чисельний результат, параметри сітки, часові характеристики та показники похибки без додаткового звернення до Processor.

Скорочений вміст артефакта webPackage.json наведено на рисунку 3.11.


```
{
 "formatVersion": 1,
 "taskId": "01KV7KYRG2RMDNJFJADB9WHJPN",
 "geometry": {
  "points": [
   [0.0, 0.0, 1.0],
   [0.0, 0.0, 0.0]
  ],
  "triangles": [
   [709, 735, 721],
   [717, 732, 712]
  ]
 },
 "values": [0.1, 0.0, 0.2, 0.1, 0.2,..., 0.3679929371630095],
 "summary": {
  "dimension": 3,
  "nPoints": 1201,
  "nTetra": 4979,
  "assembleMs": 165,
  "solveMs": 3,
  "totalMs": 168,
  "maxAbsError": 0.02912995375023608,
  "l2Error": 0.006768991143176869,
  "nTriangles": 1470,
  "min": 0,
  "max": 1.1283249783952556,
  "avg": 0.2907330554922901
 }
}
```


Рисунок 3.11 – Артефакт webPackage.json

Доступ клієнта до webPackage.json не здійснюється безпосередньо через внутрішню адресу MinIO. Клієнтський застосунок звертається до захищеного REST-контракту GET /tasks/{id}/web-package. BFF перевіряє маркер доступу і належність задачі користувачу, після чого отримує об’єкт зі сховища та повертає його клієнту. Таким чином, внутрішня структура об’єктного сховища залишається прихованою, а правила авторизації зосереджуються на зовнішній межі серверного контуру.

Артефакти записуються до сховища до публікації події про завершення відповідної стадії. Наприклад, Preprocessor спочатку зберігає mesh.msh і jobSpec.json, а лише після успішного завершення операцій публікує

preprocess.completed. Аналогічно Processor публікує solve.completed після запису solution.json, а Postprocessor – postprocess.completed після формування webPackage.json. Такий порядок запобігає ситуації, коли споживач отримує подію готовності, але відповідний об’єкт ще відсутній у сховищі.

Фіксована схема ключів також сприяє ідемпотентності. Повторне опрацювання тієї самої стадії з тим самим taskId звертається до передбачуваного ключа, а не створює неконтрольовану кількість об’єктів із випадковими назвами. Водночас за потреби зберігати результати повторних спроб схема може бути розширена сегментом attempt.

Отже, контракти артефактів забезпечують відтворювану передачу даних між стадіями без прямої залежності мікросервісів один від одного. Preprocessor, Processor і Postprocessor узгоджуються через стабільні формати та передбачувані ключі, а Kafka-події містять лише посилання на результати. Наступним елементом реалізації є сховище метаданих, у якому фіксуються загальний стан задачі, переходи між стадіями, часові мітки й повторні спроби виконання.

3.6 Зберігання метаданих задач і станів стадій

Об’єктне сховище, розглянуте в попередньому підрозділі, призначене для зберігання значних за обсягом проміжних і кінцевих артефактів, однак не забезпечує ефективного керування поточним станом виконання FEA-конвеєра. Для реєстрації задач, контролю переходів між стадіями, перевірки повторного опрацювання подій і накопичення часових міток у реалізованій платформі використано окреме реляційне сховище метаданих на основі PostgreSQL [11, 64].

До реляційного сховища не записуються масиви вузлів та елементів сітки, значення чисельного поля або пакет візуалізації. Воно містить компактні структуровані дані: ідентифікатор задачі, її власника, параметри запуску, агрегований статус конвеєра, записи окремих стадій, номери спроб і часові мітки. Таким чином, PostgreSQL формує частину керівного контуру платформи, тоді як

MinIO використовується як сховище фактичних вхідних, проміжних і вихідних даних.

Модель метаданих реалізовано за допомогою двох пов’язаних таблиць – tasks і task_stages [78]. Таблиця tasks містить параметри запуску та агрегований стан усього конвеєра. Таблиця task_stages зберігає окремі записи про проходження стадій created, preprocess, solve та postprocess. Між ними встановлено зв’язок «один до багатьох»: одна задача може мати декілька записів стадій, а кожний запис стадії належить одній задачі.

Зв’язок реалізовано зовнішнім ключем task_stages.task_id, який посилається на первинний ключ tasks.id. Для зовнішнього ключа встановлено правило ON DELETE CASCADE. Отже, видалення запису задачі супроводжується автоматичним видаленням пов’язаної історії її стадій. Основні поля фактичної схеми наведено в таблиці 3.6.

Таблиця 3.6 – Структура сховища метаданих задач і стадій

Таблиця	**Поле**	**Тип PostgreSQL**	**Призначення**
tasks	id	text	Первинний і наскрізний ідентифікатор задачі taskId
tasks	user_id	text	Ідентифікатор користувача, отриманий із маркера доступу
tasks	scenario_id	text	Ідентифікатор параметризованої постановки
tasks	mesh_size	double precision	Параметр дискретизації
tasks	status	text	Агрегований стан FEA-конвеєра
tasks	created_at	timestamp with time zone	Час реєстрації задачі
tasks	updated_at	timestamp with time zone	Час останньої зміни стану
task_stages	id	bigint	Первинний ідентифікатор запису стадії
task_stages	task_id	text	Зовнішній ключ на tasks.id
task_stages	stage	text	Назва стадії конвеєра
task_stages	status	text	Поточний або термінальний стан стадії
task_stages	started_at	timestamp with time zone	Час початку виконання
task_stages	completed_at	timestamp with time zone	Час успішного або аварійного завершення
task_stages	attempt	integer	Номер спроби виконання стадії

Для часових полів використано тип timestamp with time zone. Це дає змогу зберігати часові моменти узгоджено та перетворювати їх відповідно до

налаштувань часового поясу під час виведення [64]. Оскільки часові мітки формуються серверною функцією now(), їх значення не залежить від годинника клієнтського застосунку.

Структуру зв’язків між сутностями подано на рисунку 3.12.

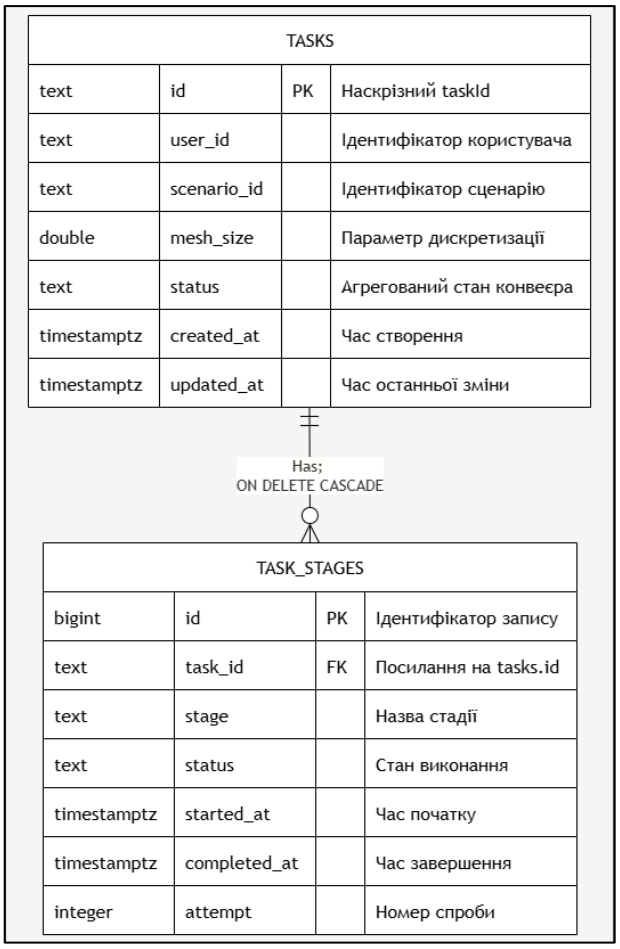


Рисунок 3.12 – Схема зберігання метаданих задач і стадій FEA-конвеєра

Схема відображає, що записи стадій не існують незалежно від задачі. Водночас для однієї задачі може бути створено декілька записів, зокрема для різних етапів або повторних спроб одного етапу. Поле attempt дає змогу логічно розрізняти такі спроби.

Фізичну структуру таблиць, обмежень та індексів наведено в Додатку Б.

Початковим значенням поля tasks.status є CREATED. У процесі проходження конвеєра воно набуває деталізованого значення, що поєднує назву активної стадії та її стан. Наприклад, для стадії розв’язання використовуються значення SOLVE_STARTED, SOLVE_COMPLETED або SOLVE_FAILED. Така

модель дає змогу визначити поточне положення задачі у конвеєрі без виконання окремого агрегувального запиту до таблиці стадій.

У таблиці tasks реалізовано B-tree-індекси для полів created_at і user_id. Перший використовується під час отримання списку останніх задач, а другий – для вибірки задач, що належать поточному користувачу.

Початок виконання стадії реєструється функцією mark_stage_started. Вона створює новий запис у task_stages зі статусом STARTED і одночасно оновлює агрегований статус задачі. Реалізацію цієї операції наведено на рисунку 3.13.

```
def mark_stage_started(engine: Engine,  task_id: str, stage: str, attempt: int = 1):
    with engine.begin() as conn:
        conn.execute(text(
            "INSERT INTO task_stages "
            "(task_id, stage, status, started_at, attempt) "
            "VALUES (:task_id, :stage, 'STARTED', "
            "now(), :attempt)"
        ), {
            "task_id": task_id,
            "stage": stage,
            "attempt": attempt
        })

        conn.execute(text(
            "UPDATE tasks "
            "SET status = :status, updated_at = now() "
            "WHERE id = :task_id"
        ), {
            "status": stage.upper() + "_STARTED",
            "task_id": task_id
        })
```

Рисунок 3.13 – Функція mark_stage_started

Контекст engine.begin() створює транзакцію, яка автоматично фіксується після успішного завершення блоку або відкочується у разі виникнення винятку. Отже, додавання запису стадії та зміна агрегованого статусу задачі виконуються як одна логічна операція. Це запобігає ситуації, коли в таблиці стадій уже зафіксовано початок виконання, а загальний статус задачі залишився незмінним, або навпаки.

У SQL-запитах використовуються іменовані параметри :task_id, :stage, :attempt та :status. Їхні значення передаються окремим словником і не конкатенуються безпосередньо з текстом SQL. Це відокремлює структуру оператора від фактичних даних і забезпечує коректну параметризацію запитів.

Після успішного завершення стадії викликається функція mark_stage_completed. Вона встановлює у відповідному записі task_stages статус COMPLETED, записує поточний час у completed_at та змінює агрегований стан задачі на <STAGE>_COMPLETED. У разі винятку функція mark_stage_failed аналогічно встановлює статус FAILED, фіксує час завершення спроби й переводить задачу до стану <STAGE>_FAILED.

Таким чином, у таблиці стадій використовуються три основні значення стану:

– STARTED – виконання стадії розпочато;
– COMPLETED – стадію успішно завершено;
– FAILED – поточну спробу завершено з помилкою.

Значення STARTED є робочим станом, тоді як COMPLETED і FAILED – термінальними. Поле completed_at заповнюється для обох термінальних станів, тому тривалість можна визначати як для успішних, так і для невдалих спроб.

У поточній реалізації функції роботи з метаданими продубльовано в модулях Preprocessor, Processor і Postprocessor. Кожен сервіс викликає їх через власний шар роботи зі стадіями. Preprocessor керує записами preprocess, Processor – solve, а Postprocessor – postprocess. Такий розподіл відповідає принципу єдиної відповідальності компонентів.

Перед початком ресурсомісткої операції кожний із трьох сервісів викликає функцію is_stage_completed. У Preprocessor ця перевірка виконується перед повторною генерацією сітки, у Processor – перед постановкою задачі чисельного розв’язання, а у Postprocessor – перед повторним формуванням пакета візуалізації. Якщо для відповідної пари taskId і stage уже існує запис зі статусом COMPLETED, повторне повідомлення не призводить до повторного виконання стадії.

Отже, перевірка стану в PostgreSQL реалізує базовий механізм ідемпотентної обробки повідомлень. Він захищає від повторного виконання вже завершеної стадії, що є важливим за семантики доставки повідомлень at-least-once. Водночас перевірка is_stage_completed і створення запису STARTED є окремими операціями. Тому цей механізм не забезпечує повного взаємного

виключення у випадку, коли декілька однакових повідомлень починають оброблятися конкурентно до завершення першої спроби.

Поле attempt дозволяє фіксувати номер спроби виконання. Його значення за замовчуванням дорівнює 1, а функція mark_stage_started приймає номер спроби як окремий параметр. При цьому в поточній схемі відсутнє унікальне обмеження на комбінацію task_id, stage і attempt. Тому контроль послідовності номерів спроб покладено на прикладну логіку сервісів.

Часові мітки з таблиці task_stages використовуються не лише для відображення часової шкали, а й для експериментальної оцінки продуктивності платформи. Тривалість завершеної стадії визначається як

$$T_{\text{stage}} = t_{\text{completed}} - t_{\text{started}}. \tag{3.1}$$

Відповідно, для основних стадій конвеєра обчислюються:

$$\begin{gathered} T_{\text{preprocess}} = t_{\text{preprocess.completed}} - t_{\text{preprocess.started}}, \\ T_{\text{solveStage}} = t_{\text{solve.completed}} - t_{\text{solve.started}}, \\ T_{\text{postprocess}} = t_{\text{postprocess.completed}} - t_{\text{postprocess.started}}. \end{gathered} \tag{3.2}$$

Повна наскрізна затримка (E2E) визначається як інтервал від реєстрації задачі до завершення постобробки:

$$T_{\text{E2E}} = t_{\text{postprocess.completed}} - t_{\text{created.started}}. \tag{3.3}$$

На відміну від внутрішніх метрик чисельного розв’язувача, повна затримка охоплює не лише безпосередню обчислювальну роботу, а й очікування в чергах, доставку Kafka-повідомлень, завантаження та запис артефактів, запуск виконавця і передавання керування між стадіями.

Сукупну величину таких витрат можна оцінити як

$$T_{\text{overhead}} = T_{\text{E2E}} - \left(T_{\text{preprocess}} + T_{\text{solveStage}} + T_{\text{postprocess}}\right). \tag{3.4}$$

Важливим аспектом є розрізнення solveStageMs і внутрішньої метрики solveMs. Перша обчислюється за часовими мітками PostgreSQL і характеризує повне перебування задачі на стадії solve, включно з очікуванням виконавця та операціями введення-виведення. Друга формується всередині чисельного розв'язувача і характеризує лише тривалість розв'язання сформованої системи лінійних алгебраїчних рівнянь. Аналогічно assembleMs стосується лише складання скінченно-елементної системи, а не всієї стадії Processor.

Доступ клієнтського застосунку до метаданих здійснюється через BFF/Aggregator. Операція GET /tasks/{id}/stages виконує вибірку полів stage, status, started_at, completed_at і attempt та повертає впорядковану історію проходження задачі. Клієнт не має прямого доступу до PostgreSQL, а перед читанням метаданих BFF перевіряє належність задачі автентифікованому користувачу.

Отже, реляційне сховище виконує функцію реєстру життєвого циклу задачі. Воно забезпечує збереження параметрів запуску, агрегованого стану, історії стадій, термінальних результатів спроб і часових міток. Перевірка статусу COMPLETED створює основу для ідемпотентного опрацювання повторних подій, а накопичені часові дані використовуються для моніторингу та експериментального оцінювання платформи. Наступним елементом реалізації є механізм передавання обчислювальних завдань від Kafka-споживачів до масштабованої множини Processor-виконавців.

3.7 Асинхронне виконання обчислювальних задач за моделлю «споживач – виконавці»

Чисельне розв'язання є найбільш ресурсоємною стадією FEA-конвеєра. Виконання FEM-алгоритму безпосередньо у процесі, який споживає Kafka-повідомлення, призвело б до блокування приймання наступних подій на весь час складання та розв'язання системи рівнянь. Тому компонент Processor поділено на

дві частини: легкий споживач подій preprocess.completed і горизонтально масштабовану множину Celery-виконавців [11–14, 63, 66, 67, 78].

Processor-споживач реалізовано як FastAPI-застосунок. Під час його запуску створюється з’єднання з PostgreSQL, формується об’єкт SolveConsumer, після чого в окремому фоновому потоці запускається цикл споживання Kafka-повідомлень. Такий поділ дозволяє одночасно підтримувати HTTP-endpoint перевірки працездатності /health та тривалий цикл читання подій.

Kafka-споживач підписується на канал подій preprocess.completed і входить до групи processor-group. Значення повідомлення спочатку декодується у текстовий формат, а потім перетворюється з JSON на об’єкт envelope. Після валідації повідомлення з нього вилучаються taskId, userId, посилання на сітку та опис постановки.

Виконання чисельної операції не відбувається у потоці Kafka-споживача. Після обробки envelope викликається асинхронна Celery-функція run_solver.delay(...), яка розміщує завдання у Redis. Лише після виклику delay() споживач фіксує поточний Kafka offset. Реалізацію цього механізму наведено на рисунку 3.14.

Для успішного шляху ручна фіксація offset після поставлення в чергу допускає повторну доставку в разі збою між постановкою Celery task і commit. Для повідомлень, обробка яких завершується винятком, у прототипі offset також фіксується, щоб уникнути циклічного споживання некоректного повідомлення.

Перед постановкою задачі в Celery компонент SolveConsumer перевіряє структуру предметних даних. Відсутність taskId, meshArtifactRef або jobSpecRef розглядається як помилка контракту. Якщо дані коректні, перевіряється наявність завершеної стадії solve у PostgreSQL. Якщо стадія ще не завершена, функція mark_stage_started створює запис зі статусом STARTED і змінює агрегований статус задачі на SOLVE_STARTED. Отже, початок стадії фіксується Processor-споживачем до передавання задачі Redis, а не безпосередньо Celery-виконавцями.

Логіку обробника наведено на рисунку 3.15.

```
def consume_loop():
    consumer = KafkaConsumer(
        TOPIC_IN,
        bootstrap_servers=KAFKA_BROKERS,
        group_id=os.getenv(
            "KAFKA_GROUP_ID",
            "processor-group"
        ),
        enable_auto_commit=False,
        auto_offset_reset="earliest",
        value_deserializer=lambda v: v.decode("utf-8"),
    )

    for msg in consumer:
        raw = msg.value
        try:
            env = json.loads(raw)

            result = HANDLER.handle_preprocess_completed(env)

            if result is not None:
                task_id, user_id, mesh_ref, job_ref = result
                run_solver.delay(task_id, user_id, mesh_ref, job_ref)

            consumer.commit()
            print(
                f"[processor] queued solve "
                f"for task={task_id}"
            )

        except Exception as e:
            print("[processor] ERROR:", e)
            consumer.commit()
```

Рисунок 3.14 – Споживання preprocess.completed і передавання задачі Celery-виконавців

```
def handle_preprocess_completed(self, envelope: dict) -> tuple[str, Optional[str], dict, dict]:
    task_id = envelope.get(
        "subject", {}
    ).get("taskId")

    user_id = envelope.get(
        "subject", {}
    ).get("userId")

    data = envelope.get("data", {}) or {}
    mesh_ref = data.get("meshArtifactRef")
    job_ref = data.get("jobSpecRef")

    if not task_id or not mesh_ref or not job_ref:
        raise RuntimeError(
            "Invalid preprocess.completed payload"
        )

    if self.stages.is_completed(task_id, "solve"):
        return None

    self.stages.start(task_id, "solve")

    return task_id, user_id, mesh_ref, job_ref
```

Рисунок 3.15 – Обробник handle_preprocess_completed

Для внутрішньої черги Processor використано Celery, а як broker і result backend – Redis. Обидві адреси передаються через змінні середовища. Створення Celery application та команда запуску виконавця наведені на рисунку 3.16.

```
BROKER_URL = os.getenv("CELERY_BROKER_URL")
RESULT_BACKEND = os.getenv(
   "CELERY_RESULT_BACKEND",
   BROKER_URL
)
celery_app = Celery(
   "processor",
   broker=BROKER_URL,
   backend=RESULT_BACKEND
)
containers:
  - name: worker
    image: fea-processor:0.1.0
    workingDir: /app/src
    command: ["celery"]
    args:
      - "-A"
      - "worker.celery_app"
      - "worker"
      - "--loglevel=INFO"
      - "--concurrency=1"
    env:
      - name: CELERY_BROKER_URL
        value: "redis://redis-master.platform.svc.cluster.local:6379/0"
      - name: CELERY_RESULT_BACKEND
        value: "redis://redis-master.platform.svc.cluster.local:6379/0"
```

Рисунок 3.16 – Конфігурація Celery та запуск Processor-виконавця

Параметр --concurrency=1 (рівень паралелізму) означає, що одна репліка processor-worker одночасно виконує одну обчислювальну задачу. Тому для N_w реплік теоретична кількість паралельно виконуваних FEM-задач становить

$$C_{\text{solve}} = N_w. \tag{3.5}$$

Саме кількість реплік processor-worker використовується як змінний параметр експерименту. Processor-споживач при цьому зберігає одну репліку, тому масштабування обчислювальної частини не змінює кількість Kafka-споживачів і не впливає на зовнішні REST-контракти.

Після отримання Celery task виконавець публікує подію solve.progress зі значенням 0 та повідомленням Started. Далі він завантажує jobSpec.json, визначає сценарій, отримує назву граничної групи та завантажує файл сітки mesh.msh. Після розбору сітки публікується прогрес 50 із повідомленням про складання й розв'язання FEM-системи.

Підтримуються двовимірні та тривимірні сценарії. Для двовимірної сітки використовується складання системи на P1-трикутниках, а для тривимірної – на P1-тетраедрах. Вибір алгоритму здійснюється за полем dimension, отриманим під час розбору сітки. Для MMS-сценаріїв обчислюються maxAbsError і l2Error, а часові показники assembleMs, solveMs і totalMs формуються за допомогою монотонного таймера time.perf_counter().

Після розв'язання виконавець формує об'єкт solution.json, що містить:

– ідентифікатор задачі;
– координати вузлів;
– поверхневі трикутники;
– значення чисельного поля;
– розмірність задачі;
– кількість вузлів та елементів;
– часові метрики;
– мінімальне, максимальне й середнє значення поля;
– похибки MMS-сценарію.

Артефакт записується до MinIO за ключем tasks/{taskId}/results/solution.json.

Після успішного запису операції виконуються у заданому порядку:

1) публікується solve.completed;
2) стадія solve переводиться у стан COMPLETED;
3) публікується solve.progress зі значенням 100.

Така послідовність створює коротке вікно eventual consistency: Postprocessor або BFF можуть отримати solve.completed до того, як у PostgreSQL буде зафіксовано SOLVE_COMPLETED. Крім того, у разі успішної публікації

події та помилки оновлення бази даних артефакт і подія вже існуватимуть, але метадані залишаться у стані SOLVE_STARTED.

Послідовність роботи Processor показано на рисунку 3.17.

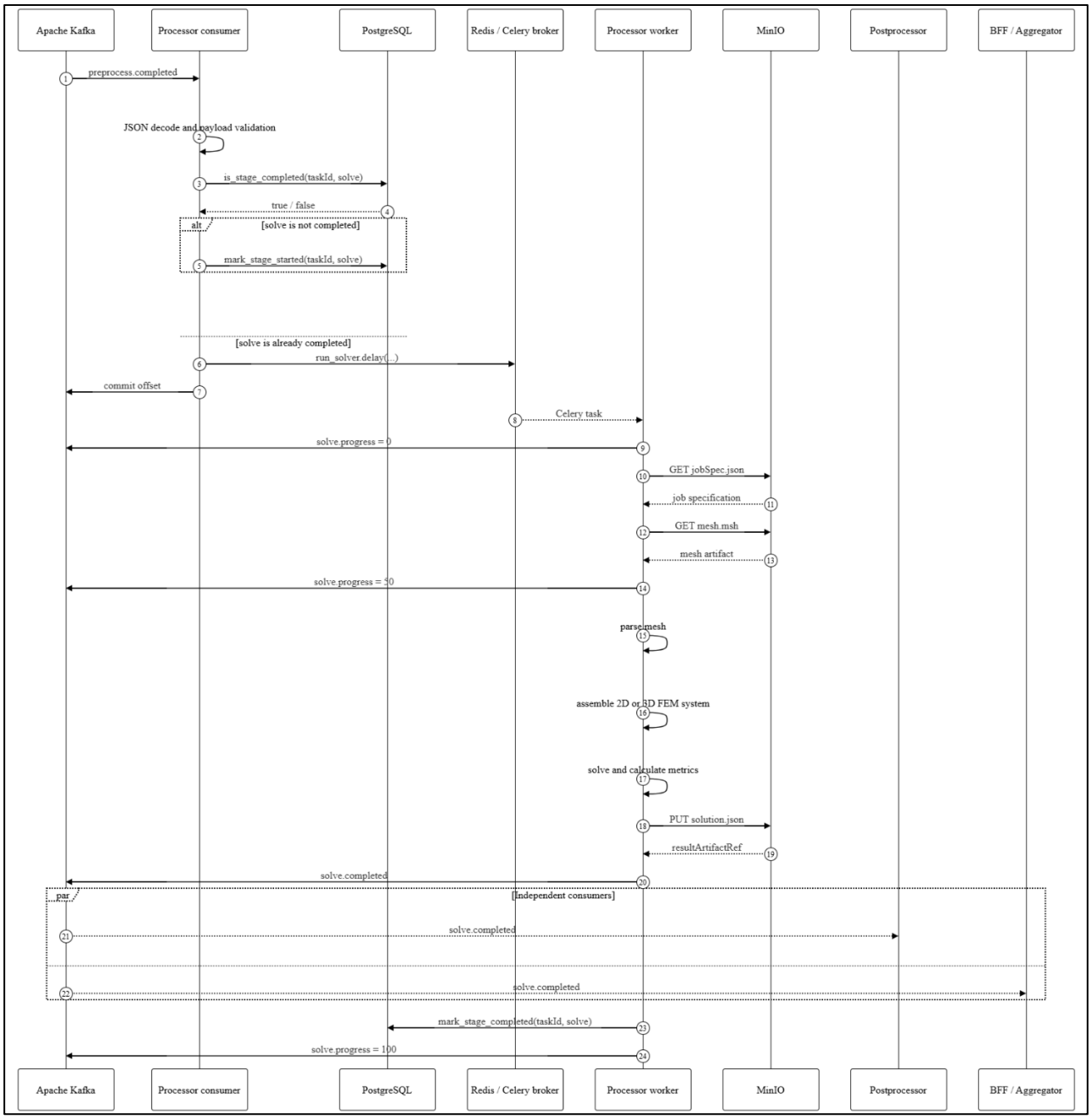


Рисунок 3.17 – Послідовність асинхронного виконання стадії solve

Горизонтальне масштабування здійснюється командою kubectl scale deployment processor-worker -n platform --replicas=<Nw>.

Усі репліки підключаються до однієї Redis-черги та використовують однаковий образ. Зі збільшенням N_w зростає кількість одночасно виконуваних

розрахунків, але лише доти, доки не досягнуто обмежень фізичного процесора, оперативної пам’яті, об’єктного сховища або інших спільних інфраструктурних компонентів.

Отже, модель «споживач – виконавці» відокремлює приймання Kafka-подій від чисельного розв’язання і створює пряму точку горизонтального масштабування.

3.8 Контейнеризація та розгортання платформи в Kubernetes

Прикладні компоненти серверного контуру упаковано в окремі контейнерні образи. Контейнеризація фіксує середовище виконання, версію мови програмування, набір бібліотек і команду запуску, завдяки чому компоненти можуть однаково виконуватися під час локальної перевірки та в Kubernetes-кластері [16, 68].

Для BFF, Preprocessor, Processor і Postprocessor використовуються окремі Dockerfile та образи:

– fea-bff:0.1.0;
– fea-preprocessor:0.1.0;
– fea-processor:0.1.0;
– fea-postprocessor:0.1.0.

Processor-споживач і Processor-виконавець використовують спільний образ fea-processor:0.1.0. Це пояснюється тим, що обидві частини мають спільний програмний код, моделі контрактів, доступ до PostgreSQL, MinIO та Kafka, але запускаються з різними командами. Dockerfile Processor наведено на рисунку 3.18.

Як базове середовище використано мінімальний образ Python 3.11. Спочатку до контейнера копіюється файл залежностей і виконується їх установлення, після чого додається каталог src. Такий порядок дозволяє Docker повторно використовувати шар зі встановленими бібліотеками, якщо програмний код змінився, але requirements.txt залишився незмінним.

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install \
   --no-cache-dir \
   -r requirements.txt

COPY src ./src

WORKDIR /app/src

EXPOSE 8000

CMD [
   "uvicorn",
   "main:app",
   "--host",
   "0.0.0.0",
   "--port",
   "8000"
]
```

Рисунок 3.18 – Dockerfile компонента Processor

Команда CMD запускає FastAPI-застосунок Processor через Uvicorn на порту 8000. Саме ця команда використовується Deployment processor. Для Deployment processor-worker вона перевизначається полями command та args, тому з того самого образу запускається Celery-виконавець.

Оркестрацію серверних та інфраструктурних компонентів реалізовано в Kubernetes [15, 69]. Основні ресурси платформи розміщено в namespace platform. Прикладні stateless-компоненти керуються Deployment, а компоненти, яким потрібна стабільна мережева ідентичність і постійне сховище, переважно розгорнуто через StatefulSet. Склад розгортання наведено в таблиці 3.7.

Таблиця 3.7 – Склад Kubernetes-розгортання

Компонент	**Kubernetes-ресурс**	**Поточна кількість pod**	**Збереження стану**	**Особливості**
BFF/Aggregator	Deployment	1	не потребує локального PVC	зовнішній REST і Socket.IO контур
Preprocessor	Deployment	1	артефакти у MinIO	споживає task.created
Processor consumer	Deployment + Service	1	метадані у PostgreSQL	FastAPI, Kafka consumer, /health
Postprocessor	Deployment	1	артефакти у MinIO	формує webPackage.json

Продовження таблиці 3.7

Компонент	Kubernetes-ресурс	Поточна кількість pod	Збереження стану	Особливості
Processor workerss	окремий Deployment	(N_w), поточно 1	результат у MinIO	concurrency=1, основна точка масштабування
Apache Kafka	StatefulSet	3	три PVC по 8 GiB	kafka-controller-0..2
PostgreSQL платформи	StatefulSet	1	PVC 8 GiB	метадані задач і стадій
Redis	StatefulSet	1	PVC 8 GiB	Celery broker і result backend
MinIO	Deployment	1	PVC 8 GiB	S3-сумісне сховище артефактів
OIDC-провайдер	StatefulSet	1	окрема PostgreSQL	–
База даних IdP	StatefulSet	1	PVC 8 GiB	зберігає дані провайдера ідентичності
Web UI	поза кластером	–	–	Angular і Three.js

Kafka складається з трьох pod kafka-controller-0, kafka-controller-1 і kafka-controller-2. Для кожного pod створено окремий PersistentVolumeClaim місткістю 8 GiB. PostgreSQL платформи, Redis та база даних провайдера ідентичності також використовують StatefulSet і окремі PVC. MinIO розгорнуто як одноекземплярний Deployment, але його дані винесено до PVC minio місткістю 8 GiB.

Усі надані PVC мають режим доступу ReadWriteOnce і використовують StorageClass local-path. Це відповідає локальному кластеру, у якому постійні томи фізично пов’язані з файловою системою вузла. Такий підхід забезпечує збереження даних після перезапуску pod, але не є повноцінним розподіленим сховищем і не гарантує доступність даних після втрати самого вузла [69].

Deployment processor має одну репліку та запускає команду, визначену в Dockerfile. Контейнер відкриває порт 8000, а його працездатність контролюється HTTP liveness probe (див. рис. 3.19).

```
livenessProbe:
  httpGet:
    path: /health
    port: 8000
  initialDelaySeconds: 10
  periodSeconds: 10
```

Рисунок 3.19 – HTTP liveness probe

Наявність liveness probe дозволяє Kubernetes перезапустити контейнер у разі, якщо FastAPI-застосунок перестає відповідати. Статус готовності pod базується переважно на стані контейнера, а не на перевірці доступності Kafka, PostgreSQL чи Redis.

Deployment processor-worker використовує той самий образ, однак перевизначає стандартну команду. Скорочене зіставлення двох Deployment наведено на рисунку 3.20.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: processor
  namespace: platform
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: processor
        role: api
    spec:
      containers:
        - name: processor
          image: fea-processor:0.1.0
          imagePullPolicy: Never
          ports:
            - containerPort: 8000
          livenessProbe:
            httpGet:
              path: /health
              port: 8000
apiVersion: apps/v1
kind: Deployment
metadata:
  name: processor-worker
  namespace: platform
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: processor
        role: worker
    spec:
      containers:
        - name: worker
          image: fea-processor:0.1.0
          imagePullPolicy: Never
          workingDir: /app/src
          command: ["celery"]
          args:
            - "-A"
            - "worker.celery_app"
            - "worker"
            - "--loglevel=INFO"
            - "--concurrency=1"
```

Рисунок 3.20 – Запуск Processor-споживача і Processor-виконавця зі спільного образу

Поле imagePullPolicy: Never означає, що Kubernetes не завантажує прикладний образ із зовнішнього registry, а використовує образ, попередньо доступний у локальному середовищі вузла.

Взаємодія компонентів усередині кластера здійснюється через Kubernetes DNS. У змінних середовища використовуються сервісні адреси:

– kafka.platform.svc.cluster.local:9092;
– redis-master.platform.svc.cluster.local:6379;
– postgres-postgresql.platform.svc.cluster.local:5432;
– minio.platform.svc.cluster.local:9000.

Завдяки цьому прикладний код не залежить від динамічних IP-адрес окремих pod. Заміна або перезапуск pod не потребує зміни адрес у програмному коді.

Параметри конфігурації передаються контейнерам безпосередньо через секцію env. До них належать адреси інфраструктурних сервісів, назви каналів подій Kafka, назва bucket, параметри PostgreSQL і доступу до MinIO.

Загальну структуру розгортання наведено на рисунку 3.21.

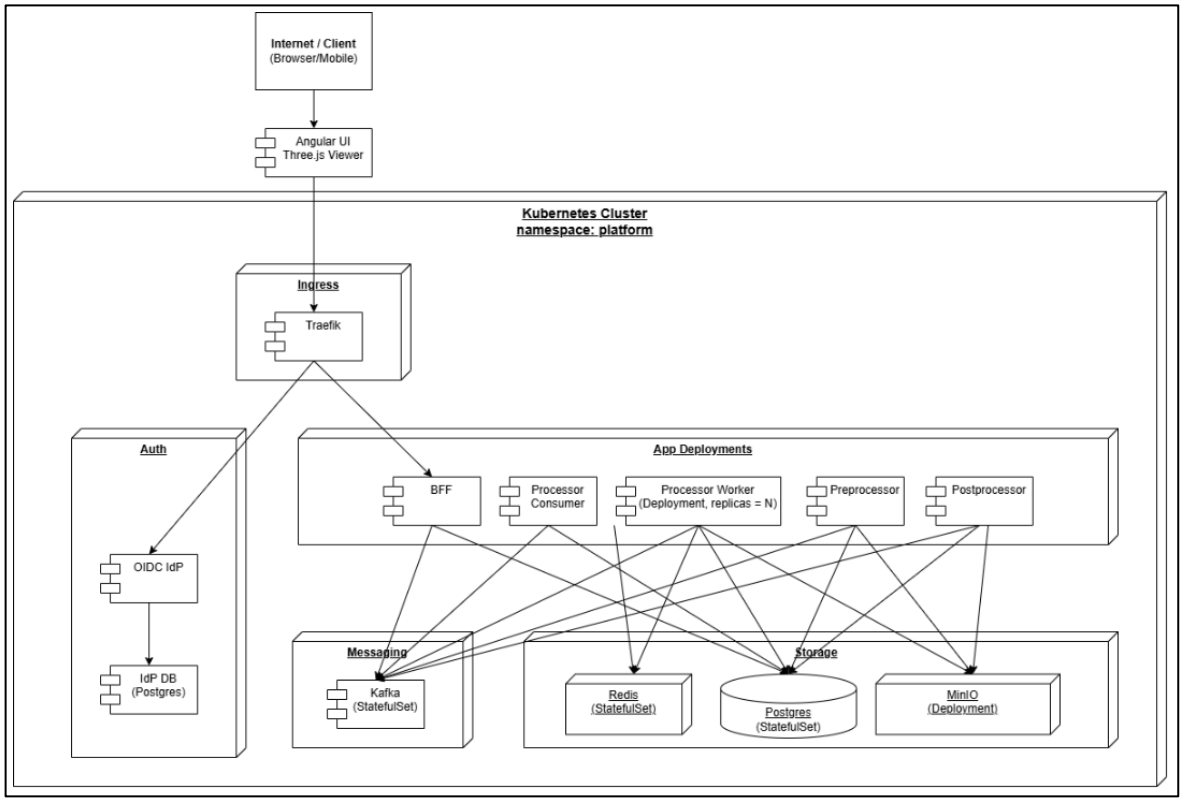


Рисунок 3.21 – Діаграма розгортання FEA-платформи

Клієнтський Angular-застосунок розташований поза кластером. Усередині Kubernetes розгорнуто серверний контур, мікросервіси та інфраструктурні компоненти. Processor-виконавця представлено окремим Deployment, кількість реплік якого змінюється незалежно від Processor-споживачів.

Для масштабування використовується команда kubectl scale deployment processor-worker -n platform --replicas=<Nw>.

Перед запуском експериментальної серії фактичну готовність реплік потрібно перевіряти за допомогою kubectl -n platform get deployment processor-worker та kubectl -n platform get pods -l app=processor,role=worker.

Серія може починатися лише після того, як усі заплановані pod мають статус Running, а кількість READY і AVAILABLE відповідає значенню N_w.

Таким чином, Docker забезпечує однакове програмне середовище компонентів, а Kubernetes – їх декларативне розгортання та керування кількістю реплік. Використання спільного образу для Processor-споживачів і -виконавців спрощує підтримку сумісних версій коду, тоді як окремий worker Deployment утворює керовану точку горизонтального масштабування [78].

3.9 Реалізація ключових прикладних компонентів платформи

У попередніх підрозділах було розглянуто зовнішні та внутрішні контракти платформи, організацію подієвої взаємодії, зберігання артефактів і метаданих, а також контейнеризоване розгортання. На рівні програмної реалізації ці механізми зосереджено у чотирьох основних прикладних компонентах: BFF/Aggregator, Preprocessor, Processor і Postprocessor.

Кожний компонент має окрему предметну відповідальність і взаємодіє з іншими складовими через формалізовані REST-, подієві та артефактні контракти. Внутрішні функції одного мікросервісу не викликаються безпосередньо іншим мікросервісом. Завдяки цьому внутрішню реалізацію окремої стадії можна змінювати за умови збереження структури її вхідних і вихідних даних.

Узагальнений розподіл відповідальності наведено в таблиці 3.8.

Таблиця 3.8 – Відповідальність ключових прикладних компонентів

Компонент	**Вхідний контракт**	**Основні операції**	**Вихідний контракт**
BFF/Aggregator	REST-запити UI, Kafka-події станів	Автентифікація, створення задач, читання метаданих, контроль підписок, передавання оновлень	REST-відповіді, Socket.IO-повідомлення, task.created
Preprocessor	task.created	Валідація параметрів, генерація сітки, формування jobSpec, запис артефактів	mesh.msh, jobSpec.json, preprocess.completed або preprocess.failed
Processor	preprocess.completed, Celery task	Розбір сітки, складання FEM-системи, чисельне розв'язання, розрахунок похибок і метрик	solution.json, solve.progress, solve.completed або solve.failed
Postprocessor	solve.completed	Перетворення результату до стабільного формату вебвізуалізації	webPackage.json, postprocess.completed або postprocess.failed

3.9.1 Реалізація BFF/Aggregator

BFF/Aggregator формує зовнішню програмну межу серверного контуру. Клієнтський застосунок не взаємодіє безпосередньо з Preprocessor, Processor або Postprocessor, а використовує REST API і Socket.IO-інтерфейс BFF. Така організація приховує внутрішню структуру FEA-конвеєра та надає вебклієнту контракти, орієнтовані на його потреби [53, 78].

Компонент реалізовано в середовищі Node.js із використанням Express, KafkaJS і Socket.IO [71, 72]. Його синхронна частина приймає HTTP-запити, перевіряє маркер доступу, визначає поточного користувача та виконує операції з метаданими й артефактами. Основні REST-контракти BFF розглянуто в підрозділі 3.3.

Під час створення задачі BFF перевіряє scenarioId і meshSize, генерує наскрізний taskId, створює записи в таблицях tasks і task_stages, формує уніфіковані структури повідомлення та публікує повідомлення task.created. Подія

надсилається до Kafka з ключем, що дорівнює taskId, після чого клієнту повертається відповідь 201 Created.

Початковий запис стадії створюється зі значеннями:

– stage = created;
– status = COMPLETED;
– started_at = now().

У схемі для цієї стадії поле completed_at не заповнюється. Тому під час розрахунку E2E як початкова точка використовується created.started_at, а сама стадія не інтерпретується як звичайний часовий інтервал.

Окремою функцією BFF є передавання подій із Kafka до клієнтського застосунку. Компонент створює споживача із групою bff-updates і підписується на такі канали подій:

– preprocess.failed;
– solve.progress;
– solve.completed;
– solve.failed;
– postprocess.completed;
– postprocess.failed.

Подія preprocess.completed у BFF не споживається, оскільки вона має внутрішнє призначення та використовується для запуску Processor. Водночас клієнт отримує інформацію про виконання через агрегований стан задачі, події наступних стадій і REST-контракт стадій.

Після отримання Kafka-повідомлення BFF десеріалізує його, визначає subject.taskId і передає скорочене представлення envelope до Socket.IO-кімнати, назва якої відповідає ідентифікатору задачі.

Доступ до такої кімнати контролюється окремо. Під час встановлення Socket.IO-з’єднання клієнт передає JWT у полі handshake.auth.token. BFF перевіряє токен, вилучає поле sub і зберігає його в контексті з’єднання як socket.userId. Після отримання команди joinTask виконується параметризований запит до PostgreSQL, який перевіряє, що задача належить поточному користувачу. Лише після успішної перевірки клієнт приєднується до кімнати.

Механізм авторизованої підписки й передавання подій наведено в Додатку В.

Наведена реалізація поєднує подієву адресацію з перевіркою володіння ресурсом. Знання taskId саме по собі не дозволяє користувачу підписатися на чужу задачу, оскільки приєднання до кімнати підтверджується запитом за парою taskId і userId.

REST і Socket.IO мають різне функціональне призначення. REST використовується для створення задачі та отримання актуального стану, тоді як Socket.IO повідомляє про зміни, що відбуваються після початкового запиту. У разі розриву з'єднання у реальному часі стан задачі не втрачається, оскільки клієнт може повторно отримати його з PostgreSQL через REST API.

3.9.2 Реалізація Preprocessor

Preprocessor реалізує першу предметну стадію FEA-конвеєра – підготовку даних для чисельного розв'язання. Сервіс виконано мовою Python із використанням FastAPI як середовища запуску та Gmsh як генератора скінченно-елементних сіток [74, 75].

Під час запуску FastAPI-застосунку створюються з'єднання з PostgreSQL і MinIO, Kafka producer та предметні об'єкти сервісу. Після цього в окремому фоновому потоці запускається Kafka-споживач, який підписується на task.created у групі preprocessor-group.

HTTP-endpoint /health використовується для контролю працездатності контейнера. Основна предметна логіка не викликається через HTTP, а запускається подією Kafka.

Обробник PreprocessConsumer вилучає з envelope:

– taskId;
– userId;
– scenarioId;
– meshSize.

Після цього перевіряється наявність обов’язкових атрибутів, числовий тип meshSize і його додатність. Якщо стадія preprocess уже має статус COMPLETED, обробка припиняється без повторної генерації артефактів.

Для нової задачі Preprocessor реєструє початок стадії, викликає MeshGenerator, формує передбачувані ключі MinIO та зберігає два артефакти:

– tasks/{taskId}/mesh/mesh.msh;
– tasks/{taskId}/job/jobSpec.json.

Розмірність постановки в поточному прототипі визначається зі значення scenarioId: якщо його нормалізований рядок містить слово cube, встановлюється dimension = 3, інакше – dimension = 2. Такий механізм є достатнім для обмеженого набору заздалегідь визначених сценаріїв, але не є універсальним способом класифікації довільної постановки.

Артефакт jobSpec.json містить:

– ідентифікатор задачі;
– сценарій;
– параметр сітки;
– розмірність;
– посилання на mesh.msh;
– тип задачі poisson;
– назву граничної фізичної групи DIRICHLET.

Основну реалізацію наведено в Додатку Г.

Preprocessor спочатку записує обидва артефакти, а потім публікує preprocess.completed. Це виключає передавання Processor посилання на об’єкт, який ще не створено.

Водночас подія завершення публікується до виклику self.stages.complete(...). Тому існує короткий проміжок, у якому Processor уже може отримати preprocess.completed, але в PostgreSQL стадія ще має статус STARTED. Це відповідає моделі eventual consistency, а також враховується під час моніторингу та повторної обробки.

У разі помилки предметний обробник самостійно встановлює FAILED, публікує preprocess.failed і повторно генерує виняток. Після цього зовнішній цикл

consume_loop() знову намагається позначити стадію як невдалу та опублікувати подію помилки.

3.9.3 Реалізація Processor

Processor є обчислювальним ядром платформи. Його реалізацію поділено на координаційну й обчислювальну частини. Координаційна частина споживає preprocess.completed і передає завдання Celery, а обчислювальна виконується одним із екземплярів processor-worker.

Внутрішню структуру компонента організовано за принципом розподілу відповідальностей (див. табл. 3.9).

Таблиця 3.9 – Основні класи реалізації Processor

Клас	**Відповідальність**
StageRepository	Робота зі станами стадії solve у PostgreSQL
ArtifactStore	Читання та запис артефактів у MinIO
PoissonSolver	Складання і розв’язання двовимірної або тривимірної FEM-системи
EventPublisher	Публікація solve.progress, solve.completed і solve.failed
SolveConsumer	Валідація preprocess.completed і підготовка Celery task
SolveWorker	Координація повного виконання обчислювальної операції

Клас PoissonSolver підтримує двовимірний варіант на P1-трикутних елементах і тривимірний варіант на P1-тетраедрах. Вибір здійснюється за властивістю parsed.dimension, отриманою під час розбору MSH-файлу.

Для MMS-сценаріїв вибирається відома точна функція, яка використовується для визначення значень на границі та обчислення maxAbsError і l2Error [79–81]. Час складання системи, її розв’язання та сумарний внутрішній час фіксується у полях assembleMs, solveMs і totalMs. Ці метрики не слід ототожнювати з тривалістю всієї стадії solve. Вони не враховують очікування в Redis, завантаження jobSpec.json і mesh.msh, розбір сітки, запис solution.json та публікацію Kafka-подій.

Основну координацію виконує SolveWorker.run. На початку він публікує прогрес 0, після завантаження та розбору вхідних даних – прогрес 50. Далі вибирається двовимірний або тривимірний чисельний розв’язувач, формуються статистики чисельного поля й артефакт розв’язку (Додаток Д).

Для тривимірної постановки в solution.json передаються поверхневі трикутники, а не повний масив тетраедрів. Це зменшує обсяг даних, які надалі передаються в модуль візуалізації, оскільки Three.js відображає зовнішню поверхню об’єкта. Кількість тетраедрів при цьому зберігається у блоці summary.

Після запису артефакта Processor публікує solve.completed, переводить стадію до COMPLETED, а потім публікує прогрес 100. Як і в Preprocessor, предметна подія завершення передує фіксації термінального стану в PostgreSQL [78].

3.9.4 Реалізація Postprocessor

Postprocessor реалізує завершальну серверну стадію FEA-конвеєра. Його відповідальністю є перетворення предметного результату Processor у стабільний контракт, орієнтований на клієнтський модуль візуалізації.

Сервіс виконано мовою Python із використанням FastAPI. Під час запуску створюються підключення до PostgreSQL і MinIO, Kafka producer і предметні класи, після чого в фоновому потоці запускається споживач групи postprocessor-group, підписаний на solve.completed.

Обробник вилучає з envelope:

– taskId;
– userId;
– resultArtifactRef.

Якщо стадія postprocess уже має статус COMPLETED, повторне повідомлення припиняється оператором return, тому артефакт не формується повторно.

Для нової задачі Postprocessor реєструє початок стадії, завантажує solution.json і передає його до WebPackageBuilder. Формується стабільна оболонка, що містить:

– formatVersion;
– taskId;
– геометрію з полями points і triangles;
– масив values;
– блок summary.

Така операція відокремлює контракт клієнта від внутрішньої назви полів Processor і створює точку майбутнього розширення постобробки.

Реалізацію наведено в Додатку Е.

Артефакт записується за ключем tasks/{taskId}/web/webPackage.json.

Після цього публікується postprocess.completed, предметна частина якого містить webPackageRef і блок summary. BFF отримує подію, передає її клієнту через Socket.IO, а сам пакет завантажується через захищений REST-контракт GET /tasks/{id}/web-package.

Внутрішній обробник Postprocessor самостійно встановлює FAILED, публікує postprocess.failed і повторно генерує виняток. Зовнішній цикл Kafka-споживача після цього повторно виконує best-effort обробку помилки.

Таким чином, Postprocessor виконує роль адаптера між обчислювальним і презентаційним форматами. Він не здійснює безпосереднього рендерингу: побудова WebGL-сцени виконується зовнішнім Angular/Three.js-модулем після отримання webPackage.json [73].

Реалізація розглянутих компонентів демонструє наскрізне застосування контрактного підходу. BFF формує захищену зовнішню межу системи, Preprocessor перетворює користувацькі параметри на сітку та машинний опис постановки, Processor виконує чисельне розв’язання, а Postprocessor формує стабільний клієнтський пакет.

3.10 Висновки до розділу 3

У розділі розроблено програмну реалізацію подієво-орієнтованої мікросервісної FEA-платформи відповідно до концептуальної моделі та системи вимог, сформованих у Розділі 2. Реалізоване рішення охоплює повний цикл виконання задачі скінченно-елементного аналізу: створення постановки, формування сітки, чисельне розв’язання, підготовку результатів для візуалізації та їх передавання до клієнтського застосунку [78].

Архітектуру реалізації побудовано на розмежуванні клієнтського, вхідного, прикладного й інфраструктурного контурів. Клієнтську частину представлено зовнішнім односторінковим вебзастосунком, який не входить до Kubernetes-кластера. Доступ до серверного контуру здійснюється через API Gateway, а централізовану автентифікацію реалізовано на основі OAuth 2.0 та OpenID Connect. BFF/Aggregator формує захищену межу системи, надає REST API, контролює доступ користувачів до задач і транслює внутрішні події до клієнта через Socket.IO.

Зовнішню синхронну взаємодію реалізовано за допомогою REST-контрактів створення задачі, отримання її метаданих і стадій, а також завантаження пакета візуалізації. Після реєстрації задачі клієнт одразу отримує наскрізний taskId, тоді як подальше виконання відбувається асинхронно. Ідентифікатор користувача не передається у відкритих параметрах запиту, а визначається з перевіреного маркера доступу. Перевірка належності задачі виконується за парою taskId і userId, що забезпечує ізоляцію ресурсів користувачів.

Внутрішню координацію стадій preprocess, solve і postprocess реалізовано через Apache Kafka. Для кожного переходу визначено окремі типи подій, а повідомлення мають уніфіковану envelope-структуру з ідентифікаційними, кореляційними та версійними полями. Використання taskId як ключа Kafka-повідомлення дозволяє маршрутизувати повідомлення однієї задачі до відповідної партиції та спрощує відстеження її життєвого циклу.

Для виключення передавання великих чисельних даних через брокер реалізовано окремий контур даних на основі S3-сумісного сховища MinIO. Між стадіями передаються посилання на артефакти, тоді як самі файли сітки, постановки, розв'язку й пакета візуалізації зберігаються за передбачуваними ключами в межах простору конкретної задачі. Основними контрактами є mesh.msh, jobSpec.json, solution.json і webPackage.json.

Метадані задач і стадій зберігаються у PostgreSQL. Таблиця tasks містить параметри запуску й агрегований стан конвеєра, а таблиця task_stages – окремі записи про початок, успішне завершення або помилку стадії. Часові мітки started_at і completed_at забезпечують побудову часової шкали й формують джерело даних для обчислення тривалостей окремих стадій та повної наскрізної затримки.

Обчислювальну частину Processor реалізовано за моделлю «споживач – виконавці». Окремий Kafka-споживач приймає події preprocess.completed і ставить задачі у внутрішню чергу Celery, транспортом якої є Redis. Безпосереднє FEM-розв'язання виконується множиною processor-worker. Кожна репліка має внутрішню паралельність рівною 1, тому кількість одночасно виконуваних задач визначається кількістю реплік N_w. Це створює ізольовану точку горизонтального масштабування без зміни Preprocessor, Postprocessor, BFF або зовнішніх контрактів платформи.

Реалізований Processor підтримує двовимірні й тривимірні постановки задачі Пуассона. Для двовимірного випадку використовуються P1-трикутні скінченні елементи, а для тривимірного – P1-тетраедральні. У межах MMS-сценаріїв обчислюються показники maxAbsError і l2Error, що дозволяє перевіряти коректність чисельної реалізації незалежно від характеристик розподіленого середовища. Окремо фіксуються внутрішні метрики assembleMs, solveMs і totalMs.

Preprocessor формує сітку засобами Gmsh і створює відтворюваний опис постановки jobSpec.json. Processor завантажує ці артефакти, формує й розв'язує скінченно-елементну систему та зберігає solution.json. Postprocessor перетворює

результат до стабільного клієнтського контракту webPackage.json, який містить геометрію, значення скалярного поля і підсумкові метрики. Безпосереднє відображення результатів виконується клієнтським модулем на основі Three.js.

Контейнеризацію прикладних компонентів виконано засобами Docker, а розгортання серверного та інфраструктурного контурів – у Kubernetes. Processor-споживач і Processor-виконавці запускаються з одного контейнерного образу, але в окремих Deployment із різними командами. Масштабування обчислювальної частини виконується зміною кількості реплік processor-worker, тоді як Kafka, Redis, PostgreSQL, MinIO та інші компоненти можуть зберігати незмінну конфігурацію протягом експериментальної серії.

Таким чином, у розділі реалізовано працездатний експериментальний прототип, що поєднує сервісну декомпозицію FEA-конвеєра, асинхронну подієву взаємодію, розділення метаданих і великих артефактів, централізований контроль доступу, реактивне інформування користувача та вибіркове масштабування обчислювальної стадії. Розроблена реалізація створює основу для експериментальної перевірки двох ключових властивостей платформи: зростання продуктивності при збільшенні кількості Processor-виконавців і збереження коректності чисельних результатів. Методика та результати цієї перевірки розглядаються у Розділі 4.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ МАСШТАБОВАНОСТІ ТА КОРЕКТНОСТІ FEA-ПЛАТФОРМИ

4.1 Програма та гіпотези експериментального дослідження

У попередніх розділах обґрунтовано архітектурні засади побудови масштабованих FEA-систем, сформовано концептуальну модель подієво-орієнтованої платформи та розроблено її програмну реалізацію. Запропонована платформа забезпечує асинхронне проходження задачі через стадії підготовки даних, чисельного розв’язання та післяобробки, а обчислювальна стадія реалізована за моделлю «споживач – виконавці» з можливістю незалежної зміни кількості реплік processor-worker.

Наявність механізму горизонтального масштабування сама по собі не підтверджує його ефективності. Збільшення кількості виконавців може скорочувати очікування задач у черзі та підвищувати пропускну здатність, але водночас створювати додаткову конкуренцію за процесорний час, оперативну пам’ять, сховище артефактів та інфраструктурні компоненти. Тому властивості реалізованої архітектури потребують експериментальної перевірки в контрольованих умовах.

Методичні засади експериментального дослідження ґрунтуються на принципах аналізу продуктивності комп’ютерних систем [82], окремому контролі верхньої частини розподілу затримок [83], класичних обмеженнях масштабування паралельних обчислень [84] та інженерних принципах вимірюваності й надійності розподілених систем [85]. Попередні результати апробації тривимірного сценарію масштабування наведено в роботі [86].

Експериментальне дослідження організовано як послідовність взаємопов’язаних етапів [82]. Сформовано двовимірну та тривимірну тестові постановки із відомими точними розв’язками; зафіксовано конфігурацію експериментального стенда й контрольовані параметри; виконано серії пакетних

запусків для N_w = {1, 2, 4}; зібрано показники пропускної здатності, наскрізної затримки, тривалості окремих стадій і внутрішнього часу чисельного розв'язувача; проведено пілотний двовимірний та основний тривимірний експерименти; обчислено показники прискорення й ефективності масштабування; перевірено відтворюваність контрольних чисельних характеристик і визначено умови насичення ресурсів експериментального стенда [78, 86].

Основним керованим параметром дослідження є кількість реплік обчислювального компонента:

$$N_w \in \{1,2,4\}. \tag{4.1}$$

При цьому кожна репліка processor-worker запускається з параметром concurrency = 1. Отже, за відсутності інших обмежень значення N_w визначає максимальну кількість FEM-задач, які можуть одночасно виконуватися на обчислювальній стадії:

$$C_{\text{solve}} = N_w. \tag{4.2}$$

Інші прикладні компоненти протягом порівнюваної експериментальної серії зберігають незмінну кількість реплік. Це дозволяє відокремити вплив масштабування Processor від змін конфігурації Preprocessor, Postprocessor, BFF/Aggregator, Kafka, Redis, PostgreSQL та MinIO.

Дослідження передбачає два послідовні етапи. На першому етапі використовується двовимірна задача Пуассона, яка була застосована під час початкової апробації реалізованої архітектури. Вона дає змогу перевірити проходження повного конвеєра, правильність збору часових міток і вплив кількості обчислювальних виконавців на виконання серії відносно легких задач.

На другому етапі використовується тривимірна задача Пуассона на тетраедральній сітці. Зміна параметра дискретизації дозволяє сформувати

щонайменше два рівні обчислювальної складності. Це необхідно для перевірки припущення, що ефект горизонтального масштабування залежить не лише від кількості обчислювальних виконавців, а й від співвідношення між тривалістю корисного FEM-обчислення та накладними витратами розподіленого конвеєра.

Для експериментального дослідження сформульовано такі гіпотези.

Гіпотеза H1. Збільшення кількості реплік processor-worker підвищує пропускну здатність платформи під час виконання серії незалежних FEA-задач.

Гіпотеза H2. Збільшення кількості обчислювальних виконавців зменшує середню та перцентильну наскрізну затримку до моменту досягнення насичення доступних ресурсів.

Гіпотеза H3. Ефект горизонтального масштабування є більш вираженим для задач із більшим обчислювальним навантаженням, оскільки для них частка FEM-обчислень у повній затримці перевищує частку накладних витрат подієвого конвеєра.

Гіпотеза H4. Після певного значення N_w подальше збільшення кількості обчислювальних виконавців не забезпечує пропорційного приросту продуктивності через конкуренцію за спільні апаратні та інфраструктурні ресурси.

Гіпотеза H5. Зміна кількості реплік processor-worker не впливає на математичну постановку та чисельний результат окремої задачі. За однакових значень сценарію і параметрів дискретизації характеристики сітки, значення поля та показники похибки мають залишатися узгодженими.

Гіпотези H1-H4 стосуються продуктивності та масштабованості реалізованої архітектури. Для їх перевірки використовуються пропускна здатність, середня, медіанна та p95 наскрізна затримка, часові характеристики стадій, а також показники прискорення й ефективності масштабування.

Гіпотеза H5 стосується функціональної та чисельної коректності. Її перевірка ґрунтується на порівнянні параметрів сітки, статистичних характеристик чисельного поля та похибок відносно відомого точного розв’язку, сформованого за методом виготовлених розв’язків.

Загальну послідовність експериментального дослідження наведено на рисунку 4.1.

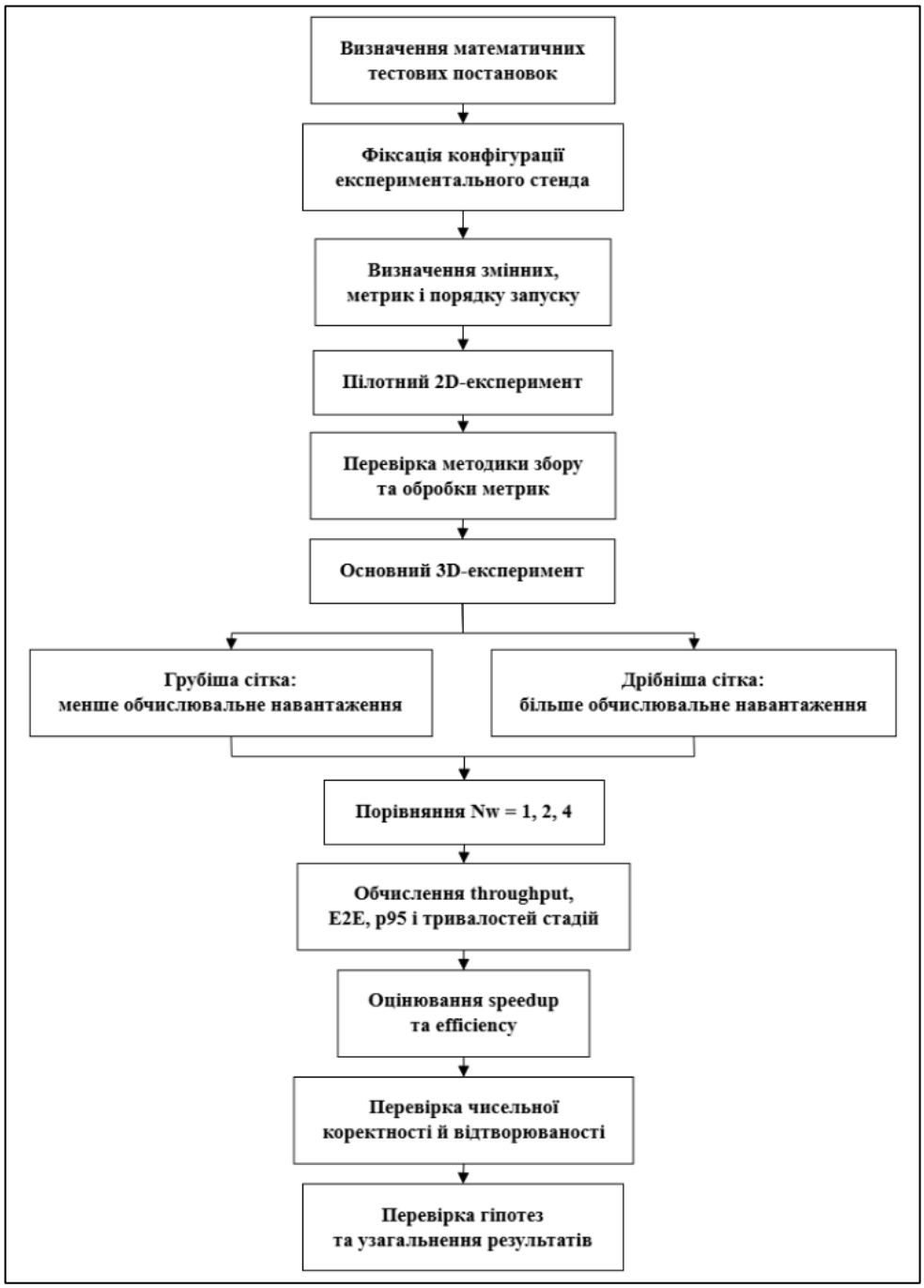


Рисунок 4.1 – Загальна схема експериментального дослідження FEA-платформи

Експериментальна логіка передбачає, що спочатку перевіряється правильність математичної постановки та чисельного розв’язувача, після чого оцінюється поведінка повного розподіленого конвеєра під навантаженням. Такий порядок дозволяє відокремити похибки чисельної реалізації від змін

продуктивності, спричинених чергами, подієвою взаємодією, операціями зі сховищами та конкуренцією між обчислювальними виконавцями.

Отже, експериментальне дослідження спрямоване не лише на підтвердження факту запуску декількох реплік Processor, а на кількісне визначення ефекту такого масштабування, його залежності від складності задачі та меж ефективності в умовах використаного стенда. Паралельно перевіряється, що зміна конфігурації виконання не порушує коректність і відтворюваність чисельних результатів.

4.2 Математична постановка тестових задач та критерії чисельної коректності

Для експериментального дослідження платформи використано модельну крайову задачу Пуассона [1–5, 36, 37]. Її вибір зумовлений тим, що вона має відносно просте математичне формулювання, але містить усі основні операції скінченно-елементного розв'язання: побудову сітки, формування локальних матриць, складання глобальної розрідженої системи, задання граничних умов і розв'язання системи лінійних алгебраїчних рівнянь.

Загальна математична постановка має вигляд

$$\begin{aligned} -\Delta u &= f \quad \text{в області } \Omega, \\ u &= g \quad \text{на границі } \partial\Omega, \end{aligned} \tag{4.3}$$

де u – шукане скалярне поле, f – задана права частина, а g – значення функції на границі області.

Для перевірки чисельної коректності застосовано метод виготовлених розв'язків [79–81]. Спочатку задається аналітично відома функція u_{exact}, після чого права частина обчислюється за співвідношенням

$$f = -\Delta u_{\text{exact}}. \tag{4.4}$$

Значення u_{exact} на границі використовуються як умови Діріхле. Після чисельного розв'язання значення u_h, отримані у вузлах скінченно-елементної сітки, порівнюються з точним розв'язком у тих самих точках.

У дослідженні використовуються дві постановки: двовимірна на одиничному квадраті та тривимірна на одиничному кубі. Перша застосовується для пілотної перевірки експериментальної методики, друга – для основного дослідження масштабованості платформи.

4.2.1 Двовимірна тестова постановка

Двовимірний сценарій має ідентифікатор poisson_square_mms. Областю розв'язання є одиничний квадрат

$$\Omega_{2D} = (0,1) \times (0,1). \tag{4.5}$$

Точний manufactured solution визначено функцією

$$u_{\text{exact}}(x, y) = \sin(\pi x) \sin(\pi y). \tag{4.6}$$

Для неї

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} &= -\pi^2 \, sin(\pi x) \, sin(\pi y), \\ \frac{\partial^2 u}{\partial y^2} &= -\pi^2 \sin(\pi x) \sin(\pi y). \end{aligned} \tag{4.7}$$

Отже,

$$-\Delta u_{\text{exact}} = 2\pi^2 \sin(\pi x) \sin(\pi y), \tag{4.8}$$

а права частина задачі має вигляд

$$f(x, y) = 2\pi^2 \sin(\pi x) \sin(\pi y). \quad (4.9)$$

На всій границі одиничного квадрата принаймні одна з координат набуває значення (0) або (1). Тому точний розв'язок на границі дорівнює нулю:

$$u_{\text{exact}}(x, y) = 0, \qquad (x, y) \in \partial\Omega_{2D}. \quad (4.10)$$

Таким чином, для двовимірного сценарію застосовуються однорідні умови Діріхле:

$$u = 0 \quad \text{на } \partial\Omega_{2D}. \quad (4.11)$$

Геометрію області формує Preprocessor засобами Gmsh як прямокутник із розмірами 1×1. Усі граничні криві об'єднуються у фізичну групу DIRICHLET, а поверхня області – у групу DOMAIN. Для дискретизації використовуються лінійні трикутні скінченні елементи першого порядку [1, 2, 5, 79].

4.2.2 Тривимірна тестова постановка

Для основного експерименту використано тривимірний сценарій poisson_cube_mms_vbc. Областю розв'язання є одиничний куб

$$\Omega_{3D} = (0,1) \times (0,1) \times (0,1). \quad (4.12)$$

Базовий тригонометричний точний розв'язок визначається як

$$u_{\sin}(x, y, z) = \sin(\pi x) \sin(\pi y) \sin(\pi z). \quad (4.13)$$

У сценарії зі змінними граничними умовами до нього додається лінійна складова:

$$u_{\text{exact}}(x, y, z) = \sin(\pi x)\sin(\pi y)\sin(\pi z) + 0{,}1(x + y + z). \qquad (4.14)$$

Лапласіан лінійної функції дорівнює нулю:

$$\Delta\big(0{,}1(x + y + z)\big) = 0. \qquad (4.15)$$

Тому права частина має вигляд:

$$f(x, y, z) = 3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z). \qquad (4.16)$$

На границі одиничного куба принаймні одна з координат дорівнює (0) або (1), тому тригонометричний добуток на границі дорівнює нулю. Відповідно, для сценарію poisson_cube_mms_vbc гранична функція має вигляд

$$g(x, y, z) = 0{,}1(x + y + z), \qquad (x, y, z) \in \partial\Omega_{3D}. \qquad (4.17)$$

На відміну від двовимірної постановки, у тривимірному сценарії використовуються неоднорідні змінні умови Діріхле. Це додатково перевіряє правильність урахування ненульових граничних значень під час формування скороченої системи рівнянь [1, 2, 5, 79].

Геометрію області Preprocessor формує як куб із розмірами $1 \times 1 \times 1$. Усі граничні поверхні об’єднуються у фізичну групу DIRICHLET, а об’єм – у групу DOMAIN.

Область дискретизується лінійними тетраедральними скінченними елементами першого порядку. Трикутники, що утворюють зовнішню поверхню тетраедральної сітки, окремо зберігаються для подальшої візуалізації результатів.

Параметр meshSize передається до Gmsh як мінімальне та максимальне характерне значення розміру елементів [75]:

$$h_{\min} = h_{\max} = \text{meshSize}. \tag{4.18}$$

Зменшення meshSize приводить до збільшення кількості вузлів і тетраедрів, а отже – до збільшення розміру глобальної системи та обчислювального навантаження.

Основні характеристики двох математичних постановок узагальнено в таблиці 4.1.

Таблиця 4.1 – Параметри тестових математичних постановок

Характеристика	**Двовимірна постановка**	**Тривимірна постановка**
Область	$(0,1)^2$	$(0,1)^3$
Геометрична модель	одиничний квадрат	одиничний куб
Скінченні елементи	P1-трикутники	P1-тетраедри
Умови Діріхле	однорідні, $g = 0$	неоднорідні, $g = 0{,}1(x + y + z)$
Гранична група Gmsh	Physical Curve("DIRICHLET")	Physical Surface("DIRICHLET")
Група області	Physical Surface("DOMAIN")	Physical Volume("DOMAIN")
Призначення	апробація	основний експеримент

4.2.3 Скінченно-елементна дискретизація

Слабке формулювання задачі Пуассона полягає у знаходженні функції (u), яка задовольняє задані умови Діріхле та співвідношення [1, 5, 36, 37]

$$\int_{\Omega} \nabla u \cdot \nabla v \, d\Omega = \int_{\Omega} f \, v \, d\Omega \tag{4.19}$$

для всіх допустимих тестових функцій v, що дорівнюють нулю на границі Діріхле.

Чисельний розв’язок апроксимується у просторі лінійних скінченно-елементних функцій [1, 5, 36, 37]:

$$u_h(\mathbf{x}) = \sum_{i=1}^{N} U_i \, \varphi_i(\mathbf{x}), \tag{4.20}$$

де N – кількість вузлів сітки, φ_i – локально-лінійні базисні функції, а U_i – невідомі вузлові значення.

У результаті формується глобальна система

$$KU = b, \tag{4.21}$$

де K – розріджена матриця жорсткості, U – вектор вузлових значень, b – вектор правої частини.

Для двовимірного трикутного елемента площею A_eлокальні елементи матриці обчислюються за формулою [1, 5, 36, 37]

$$K_{mn}^{(e)} = \frac{b_m b_n + c_m c_n}{4A_e}. \tag{4.22}$$

Права частина інтегрується одновузловою квадратурною формулою в центроїді трикутника: [1, 5, 36, 37]

$$b_m^{(e)} \approx f(x_c, y_c)\frac{A_e}{3}, \qquad m = 1,2,3. \tag{4.23}$$

Для тривимірного тетраедрального елемента об’ємом V_e локальна матриця має вигляд

$$K_{mn}^{(e)} = V_e \nabla\varphi_m \cdot \nabla\varphi_n. \tag{4.24}$$

Градієнти базисних функцій є сталими в межах лінійного тетраедра. Права частина також обчислюється за одновузловою квадратурою в центроїді [1, 5, 36, 37]:

$$b_m^{(e)} \approx f(x_c, y_c, z_c)\frac{V_e}{4}, \qquad m = 1,2,3,4. \tag{4.25}$$

Локальні матриці та вектори накопичуються у глобальній системі. Матриця формується у розрідженому форматі COO з подальшим перетворенням у CSR [39].

Умови Діріхле враховуються шляхом виключення фіксованих ступенів вільності [1, 5, 36, 37]. Після поділу вузлів на вільні f та граничні d розв'язується система

$$K_{ff}U_f = b_f - K_{fd}U_d, \tag{4.26}$$

де U_d містить значення точного розв'язку на граничних вузлах. Для сценарію poisson_square_mms ці значення дорівнюють нулю, а для poisson_cube_mms_vbc визначаються функцією

$$U_d = 0{,}1(x + y + z). \tag{4.27}$$

Розв'язання сформованої розрідженої системи здійснюється функцією spsolve бібліотеки SciPy [77].

4.2.4 Критерії чисельної коректності

Для кожної задачі Processor обчислює точний розв'язок у всіх вузлах сітки:

$$u_i^{\text{exact}} = u_{\text{exact}}(\mathbf{x}_i). \tag{4.28}$$

Похибка у вузлі визначається як

$$e_i = u_i^h - u_i^{\text{exact}}. \tag{4.29}$$

Першою метрикою є максимальна абсолютна вузлова похибка:

$$e_{\max} = \max_{1 \le i \le N} \left| u_i^h - u_i^{\text{exact}} \right|. \tag{4.30}$$

У програмному контракті це значення зберігається в полі maxAbsError. Друга метрика обчислюється за формулою [79–81]

$$e_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(u_i^h - u_i^{\text{exact}} \right)^2}. \tag{4.31}$$

У реалізації ця величина зберігається в полі l2Error. Водночас за математичним змістом вона є дискретною середньоквадратичною похибкою у вузлах, а не інтегральною нормою

$$\left(\int_{\Omega} |u_h - u_{\text{exact}}|^2 \, d\Omega \right)^{1/2}. \tag{4.32}$$

Тому надалі позначення l2Error використовується як назва програмного поля, тоді як у математичному аналізі величина інтерпретується як вузлова RMS-похибка [79–81].

Окрім показників похибки, для перевірки незмінності чисельної постановки між різними конфігураціями масштабування порівнюються:

– кількість вузлів nPoints;

– кількість двовимірних елементів nTriangles;

– кількість тривимірних елементів nTetra;

– кількість поверхневих трикутників;
– мінімальне значення поля;
– максимальне значення поля;
– середнє вузлове значення;
– maxAbsError;
– l2Error.

Для однакових scenarioId і meshSize кількість реплік processor-worker не змінює геометрію, сітку, праву частину, граничні умови або алгоритм розв’язання окремої задачі. Тому характеристики сітки та чисельного поля мають залишатися однаковими незалежно від N_w.

Для порівняння числових показників використовується критерій наближеної рівності:

$$|a - b| \leq \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}}|b|, \tag{4.33}$$

де

$$\varepsilon_{\text{abs}} = 10^{-12}, \qquad \varepsilon_{\text{rel}} = 10^{-9}. \tag{4.34}$$

Такий критерій враховує можливі відмінності, пов’язані з представленням чисел подвійної точності та серіалізацією результатів. Якщо значення збігаються у всіх збережених розрядах, критерій виконується автоматично.

Гіпотеза про збереження чисельної коректності вважається підтвердженою, якщо для кожної фіксованої математичної постановки:

1) усі задачі завершено зі статусом COMPLETED;
2) значення nPoints, nTriangles, nTetra та параметрів постановки не залежать від N_w;
3) значення maxAbsError і l2Error узгоджуються між конфігураціями в межах заданого критерію;
4) мінімальні, максимальні та середні значення поля не змінюються при масштабуванні;

5) конвеєр завершено успішно та сформовано артефакти solution.json і webPackage.json, необхідні для подальшої перевірки й візуалізації.

Таким чином, метод виготовлених розв’язків дозволяє відокремити оцінювання архітектурної продуктивності від перевірки математичної правильності. Кількість обчислювальних виконавців може впливати на час очікування, швидкість завершення серії та порядок виконання задач, але не повинна впливати на чисельний результат кожної окремої постановки.

4.3 Експериментальний стенд

Експериментальне дослідження проведено в локальному контейнеризованому середовищі на одному фізичному комп’ютері. Для розгортання серверного контуру використано Kubernetes-кластер на основі k3d і k3s. Кластер містить один керівний і два робочі логічні вузли, однак усі вони функціонують як контейнери Docker на одному фізичному хості та спільно використовують його процесор, оперативну пам’ять і накопичувач.

Така конфігурація дає змогу відтворити механізми Kubernetes-розгортання, планування pod і горизонтального масштабування, але не є фізично розподіленим багатовузловим кластером. Тому збільшення кількості processor-worker підвищує паралельність виконання задач лише в межах ресурсів одного комп’ютера.

4.3.1 Апаратна конфігурація

Фізичним хостом експериментального стенда є ноутбук ASUS TUF Gaming F15 FX506HF. Основні характеристики апаратного середовища наведено в таблиці 4.2.

Чисельне розв’язання виконується лише на центральному процесорі. Реалізовані FEM-операції використовують NumPy, SciPy та розріджений чисельний розв’язувач spsolve, спеціалізовані GPU-обчислення в Processor не

застосовуються. Наявність дискретного графічного адаптера не впливає на результати експерименту, оскільки Three.js використовується лише для клієнтської візуалізації після завершення серверного конвеєра.

Таблиця 4.2 – Апаратна конфігурація експериментального стенда

Характеристика	**Значення**
Модель комп’ютера	ASUS TUF Gaming F15 FX506HF
Процесор	Intel Core i5-11400H, 11-те покоління
Фізичні ядра	6
Логічні процесори	12
Потоків на ядро	2
Базова тактова частота	2,70 ГГц
Максимальна тактова частота	до 4,50 ГГц
Кеш L3	12 МіБ
Оперативна пам’ять	15 ГіБ доступного системі обсягу
Swap	4 ГіБ
Накопичувач	Samsung MZVLQ512HBLU-00B00
Тип накопичувача	NVMe SSD
Доступний обсяг накопичувача	476,9 ГіБ
NUMA-вузли	1
Архітектура	x86-64

Усі компоненти, включно з Kafka, Redis, PostgreSQL, MinIO, BFF і обчислювальними виконавцями, використовують спільні ресурси фізичного хоста. Тому при збільшенні кількості процесів обчислювальних виконавців вони можуть конкурувати як між собою, так і з інфраструктурними компонентами за процесорний час, оперативну пам’ять та операції введення-виведення.

4.3.2 Системне та контейнерне середовище

На фізичному комп’ютері встановлено операційну систему Ubuntu 24.04.1 LTS з ядром Linux 6.17.0-29-generic. Контейнерне середовище побудовано на Docker Engine 27.5.1 [68].

Kubernetes-кластер створено засобами k3d 5.8.3. Серверною реалізацією Kubernetes є k3s 1.31.5 [69]. Контейнери вузлів використовують containerd 1.7.23-k3s2.

Логічну структуру кластера наведено в таблиці 4.3.

Таблиця 4.3 – Логічні вузли Kubernetes-кластера

Вузол	**Роль**	**Версія**	**Внутрішня адреса**
k3d-fea-server-0	control plane, master	k3s 1.31.5	172.19.0.2
k3d-fea-agent-0	worker node	k3s 1.31.5	172.19.0.3
k3d-fea-agent-1	worker node	k3s 1.31.5	172.19.0.4

Наявність трьох логічних вузлів дозволяє Kubernetes розподіляти pod між різними k3d-контейнерами. Проте такий розподіл не надає кожному вузлу окремих фізичних процесорів або оперативної пам’яті: усі вузли працюють на одному ноутбуці.

4.3.3 Версії прикладного програмного забезпечення

Версії основних засобів, використаних у серверних компонентах, наведено в таблиці 4.4.

Для прикладних компонентів використовуються локальні контейнерні образи версії 0.1.0:

– fea-bff:0.1.0;
– fea-preprocessor:0.1.0;
– fea-processor:0.1.0;
– fea-postprocessor:0.1.0.

Processor-споживач і всі репліки Processor-виконавця використовують однаковий образ fea-processor:0.1.0. Відмінність між ними визначається командою запуску: споживач запускає FastAPI-застосунок, а виконавець – Celery.

Таблиця 4.4 – Версії основного програмного забезпечення

Компонент або бібліотека	**Версія**
Python	3.11.15
Node.js [71]	20.20.2
Gmsh [75]	4.13.1
NumPy [76]	1.26.4
SciPy [77]	1.11.4
FastAPI [74]	0.115.0
Celery [67]	5.4.0
kafka-python	2.0.2
MinIO Python SDK	7.2.8
meshio	5.3.5
Docker Engine [68]	27.5.1
k3d	5.8.3
k3s / Kubernetes server	1.31.5
kubectl client	1.35.5

Основні інфраструктурні компоненти розгорнуто на базі таких контейнерних образів:

– Apache Kafka 4.0.0 [63];
– Keycloak 26.3.3 [62];
– MinIO (RELEASE.2025-07-23T15-54-02Z) [65];
– PostgreSQL 17.6 для сховища провайдера ідентичності [64];
– PostgreSQL для метаданих платформи [64];
– Redis для Celery [66, 67].

Для PostgreSQL платформи та Redis у маніфесті використано контейнерні теги latest. Тому під час відтворення експерименту необхідно або використовувати ті самі локально наявні образи, або додатково зафіксувати їхні digest чи точні версії.

4.3.4 Конфігурація Kubernetes-компонентів

Прикладні та інфраструктурні компоненти розміщено в namespace platform. Базову конфігурацію, яка має залишатися сталою протягом порівнюваних експериментальних серій, наведено в таблиці 4.5.

Таблиця 4.5 – Контрольована конфігурація компонентів платформи

Компонент	**Kubernetes-ресурс**	**Кількість реплік**
BFF/Aggregator	Deployment	1
Preprocessor	Deployment	1
Processor consumer	Deployment	1
Processor worker	Deployment	$N_w \in \{1,2,4\}$
Postprocessor	Deployment	1
Kafka controller	StatefulSet	3
PostgreSQL платформи	StatefulSet	1
Redis	StatefulSet	1
MinIO	Deployment	1
Keycloak	StatefulSet	1
PostgreSQL Keycloak	StatefulSet	1

Єдиним компонентом, кількість реплік якого змінюється між порівнюваними конфігураціями, є processor-worker. Кількість реплік BFF, Preprocessor, Processor consumer, Postprocessor та інфраструктурних компонентів залишається сталою.

Processor-виконавець запускається командою celery -A worker.celery_app worker --loglevel=INFO --concurrency=1.

Тому одна репліка обчислювального виконавця одночасно виконує одну FEM-задачу. Теоретичний рівень паралельності обчислювальної стадії становить

$$C_{\text{solve}} = N_w. \tag{4.35}$$

Для Processor-виконавця також використовується одна репліка. Отже, зміна N_w не змінює кількість Kafka-виконавців групи processor-group, а впливає лише на кількість виконавців, які отримують завдання з Redis.

Для контейнерів processor і processor-worker не визначено resources.requests та resources.limits. Таким чином, Kubernetes не резервує для кожної репліки окрему частку CPU або оперативної пам’яті та не встановлює жорстке верхнє обмеження їх використання. Всі процеси обчислювальних виконавців й інфраструктурні компоненти конкурують за ресурси фізичного хоста відповідно до фактичного навантаження.

Ця конфігурація є важливою для інтерпретації результатів. Перехід від одного до двох виконавців може використовувати додаткові доступні ядра процесора, тоді як подальше збільшення кількості реплік здатне привести до зниження ефективності через конкуренцію за CPU, пам’ять, Redis, PostgreSQL, Kafka та MinIO.

4.3.5 Організація постійного зберігання

Для збереження стану інфраструктурних компонентів використовуються PersistentVolumeClaim зі StorageClass local-path. Постійні томи мають режим доступу ReadWriteOnce і місткість 8 ГіБ.

Окремі PVC створено для:

– трьох pod Kafka;
– PostgreSQL платформи;
– PostgreSQL провайдера ідентичності;
– Redis;
– MinIO.

Оскільки використовується local-path, дані фізично зберігаються на тому самому комп’ютері, де працює k3d-кластер. Операції Kafka, PostgreSQL, Redis і MinIO відповідно конкурують за пропускну здатність одного NVMe-накопичувача.

4.3.6 Схема експериментального стенда

Загальну структуру експериментального середовища наведено на рисунку 4.2.

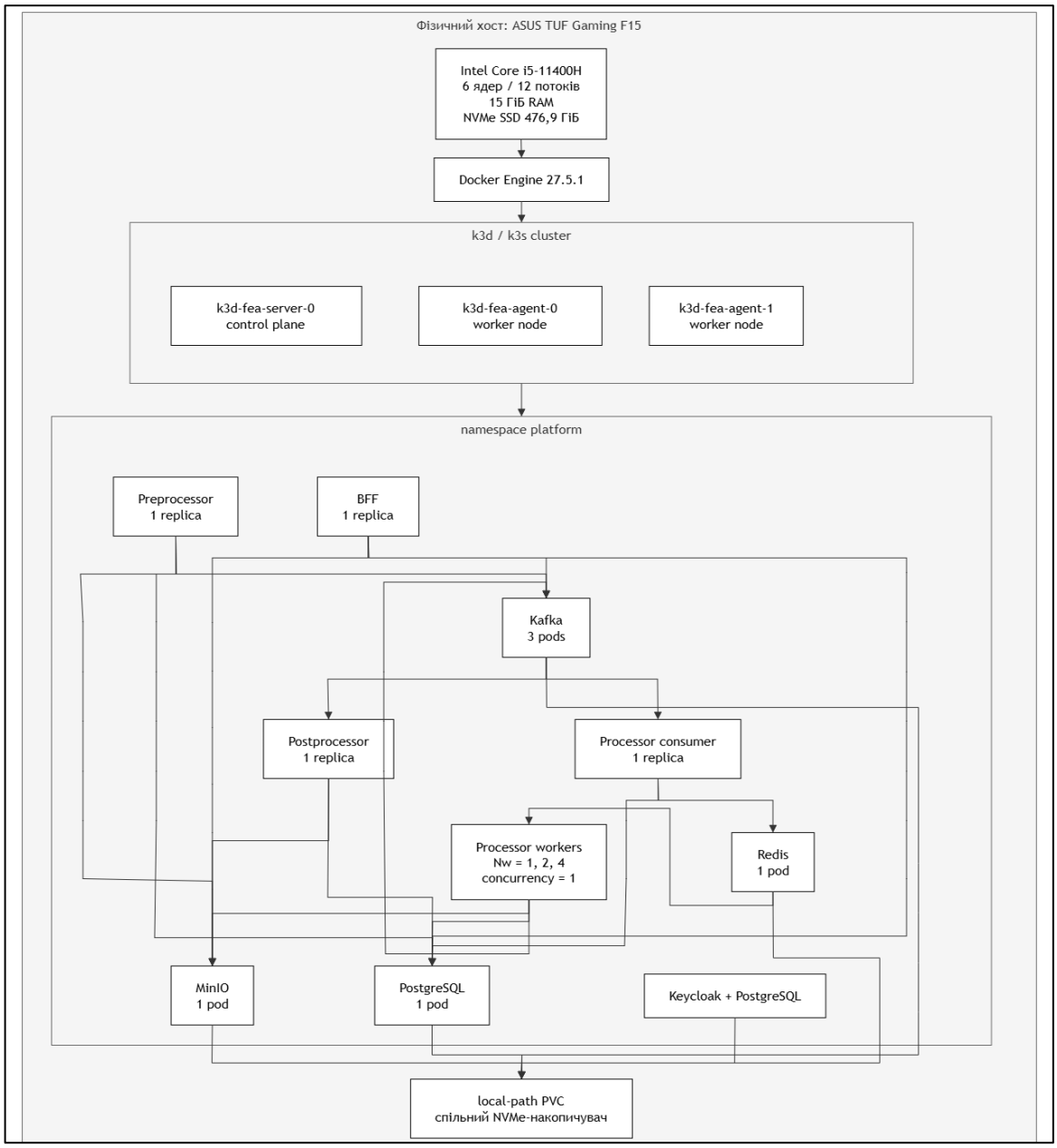


Рисунок 4.2 – Схема експериментального стенда

Перед початком кожного експериментального прогону перевіряється, що всі необхідні pod перебувають у стані Running, а кількість готових і доступних реплік processor-worker відповідає запланованому значенню N_w. Запуск серії до завершення масштабування або відновлення компонентів не допускається.

Для зменшення впливу зовнішніх факторів основні експериментальні серії виконувалися без паралельного запуску інших ресурсомістких програм. Ноутбук був під’єднаний до мережі живлення, а режим продуктивності системи залишався незмінним. Отже, експериментальний стенд відтворює тривузлову Kubernetes-конфігурацію на одному фізичному хості. Такий підхід дозволяє перевірити архітектурну можливість незалежного масштабування Processor-виконавців і дослідити поведінку системи в умовах спільного використання обмежених локальних ресурсів. Отримані результати характеризують саме цю конфігурацію і не повинні без додаткової перевірки переноситися на фізично розподілений багатовузловий кластер.

4.4 Методика проведення експериментів

Експериментальне дослідження організовано як серію контрольованих пакетних запусків задач скінченно-елементного аналізу [82]. У межах кожної серії математична постановка, параметр дискретизації, кількість задач і спосіб їх подання залишаються незмінними, а керованою змінною є кількість реплік обчислювального компонента processor-worker.

Пакетні запуски виконуються через спеціалізований експериментальний модуль клієнтського застосунку Angular. Модуль забезпечує створення заданої кількості задач, приєднання до їхніх Socket.IO-кімнат, моніторинг подій виконання, отримання часових характеристик стадій, збирання чисельних метрик і формування CSV-файлу з первинними результатами [78, 86].

4.4.1 Змінні експерименту

Основною незалежною змінною є кількість реплік processor-worker:

$$N_w \in \{1,2,4\}. \tag{4.36}$$

Кожна репліка запускається з параметром --concurrency=1 і одночасно виконує одну FEM-задачу. Тому теоретична кількість паралельних обчислень на стадії solve дорівнює кількості активних реплік обчислювальних виконавців:

$$C_{\text{solve}} = N_w. \tag{4.37}$$

Другою незалежною змінною є параметр характерного розміру елементів сітки:

$$\text{meshSize} \in \{0{,}2;\, 0{,}08\}. \tag{4.38}$$

Зменшення meshSize приводить до збільшення кількості вузлів і скінченних елементів, а отже – до збільшення обсягу операцій складання матриці та розв’язання системи лінійних алгебраїчних рівнянь.

Дослідження охоплює два математичні сценарії:

- poisson_square_mms – двовимірна задача на одиничному квадраті;
- poisson_cube_mms_vbc – тривимірна задача на одиничному кубі зі змінними граничними умовами Діріхле.

У кожній серії створюється $N = 30$ незалежних задач з однаковими значеннями scenarioId і meshSize.

Кількість паралельних клієнтських процедур подання задач установлено рівною

$$C_{\text{submission}} = 3. \tag{4.39}$$

При цьому зазначений параметр визначає не кількість одночасно виконуваних FEM-розрахунків, а кількість асинхронних клієнтських процедур, які надсилають запити POST /tasks. Після отримання відповіді з taskId процедура переходить до створення наступної задачі. Оскільки BFF реєструє задачу та повертає відповідь до завершення її обробки конвеєром, кількість активних задач у системі може перевищувати значення рівеня паралелізму.

Залежними змінними експерименту є:

– загальна тривалість серії;
– пропускна здатність;
– наскрізна затримка окремої задачі;
– тривалість стадій preprocess, solve і postprocess;
– внутрішні часові метрики FEM-розв’язувача;
– кількість успішно завершених і невдалих задач;
– характеристики сформованої сітки;
– статистичні характеристики чисельного поля;
– максимальна абсолютна та середньоквадратична вузлові похибки.

Контрольованими параметрами є:

– кількість задач у серії;
– рівень паралелізму;
– кількість реплік усіх компонентів, крім processor-worker;
– конфігурація Kafka, Redis, PostgreSQL і MinIO;
– версії контейнерних образів;
– апаратне середовище;
– математична постановка в межах порівнюваної групи;
– параметр meshSize у межах порівнюваної групи;
– спосіб збирання та обчислення метрик.

Основні параметри експерименту наведено в таблиці 4.6.

Таблиця 4.6 – Параметри експериментальних серій

Параметр	**Значення**
2D-сценарій	poisson_square_mms
3D-сценарій	poisson_cube_mms_vbc
meshSize	0,2 / 0,08
Кількість задач у серії	30
Рівень паралелізму	3
Кількість реплік виконавців	1 / 2 / 4
Кількість повторів	3
Рівень паралелізму однієї репліки Celery	1

4.4.2 Формування пакета задач

На початку серії експериментальний модуль зберігає її конфігурацію та фіксує локальну часову мітку seriesStartedAt. Після цього формується черга з $N = 30$ елементів, кожен із яких відповідає одній задачі.

Для обробки черги створюються три асинхронні клієнтські процедури. Кожна процедура послідовно:

1) вилучає один елемент із черги;
2) надсилає до BFF запит POST /tasks;
3) передає scenarioId і meshSize;
4) отримує у відповіді унікальний taskId;
5) реєструє задачу в локальному стані експерименту;
6) приєднується до Socket.IO-кімнати цієї задачі;
7) переходить до наступного елемента черги.

Процедури працюють паралельно до вичерпання черги. Така схема обмежує кількість одночасних HTTP-запитів, але не очікує завершення повного FEA-конвеєра перед поданням наступної задачі.

Усі задачі однієї серії мають однакові параметри математичної постановки, проте кожна отримує новий ULID. Це ізолює артефакти різних запусків у MinIO та записи задач у PostgreSQL.

Попередні задачі, записи метаданих і артефакти перед новою серією не видаляються. Черга Redis/Celery також примусово не очищається. Нова серія запускається лише після завершення попередньої, тому за нормального проходження експерименту в черзі не повинно залишатися задач попереднього запуску.

4.4.3 Зміна кількості обчислювальних виконавців

Кількість реплік processor-worker змінюється засобами Kubernetes командою kubectl -n platform scale deployment/processor-worker --replicas=<N_w>.

Після масштабування перевіряється завершення розгортання: kubectl -n platform rollout status deployment/processor-worker --timeout=180s.

Значення --timeout=180s у цій команді використовується лише як максимальний час очікування готовності Kubernetes Deployment і не є тайм-аутом FEM-задачі або експериментальної серії.

Після підтвердження готовності необхідної кількості pod витримується стабілізаційна пауза тривалістю 30 секунд. Вона використовується для завершення запуску контейнерів, установлення з'єднань із Redis, PostgreSQL, Kafka та MinIO, а також стабілізації фонового навантаження.

Лише після завершення 30-секундної паузи розпочинається подання пакета задач.

Фактична кількість готових реплік перевіряється командами: kubectl -n platform get deployment processor-worker та kubectl -n platform get pods -l app=processor,role=worker -o wide.

Серія не запускається, якщо кількість готових реплік не відповідає запланованому значенню N_w.

4.4.4 Моніторинг виконання задач

Після створення задачі клієнтський застосунок приєднується до Socket.IO-кімнати, ідентифікатором якої є taskId. Подальший стан задачі оновлюється на основі повідомлень, які BFF транслює з Kafka.

Подія solve.progress використовується для оновлення поточного відсотка виконання. Подія postprocess.completed сигналізує про завершення обчислювального конвеєра та містить підсумкові чисельні характеристики:

– розмірність задачі;
– кількість вузлів;
– кількість трикутників;
– кількість тетраедрів для 3D;

– assembleMs;
– solveMs;
– totalMs;
– мінімальне, максимальне й середнє значення поля;
– maxAbsError;
– l2Error.

Події з суфіксом .failed переводять задачу у стан FAILED і зберігають отриманий опис помилки.

Часові характеристики стадій не обчислюються за локальним часом надходження Socket.IO-повідомлень. Після завершальної події клієнт звертається до REST-методу GET /tasks/{taskId}/stages і отримує часові мітки, зафіксовані в PostgreSQL.

Оскільки подія postprocess.completed може надійти раніше, ніж завершальна часова мітка буде доступною в базі даних, клієнт повторює отримання стадій до появи запису postprocess зі статусом COMPLETED і повного набору часових показників.

Таким чином, статус COMPLETED у локальному експериментальному наборі фіксується лише після отримання:

– preprocessMs;
– solveStageMs;
– postprocessMs;
– e2eMs;
– чисельного summary.

Тривалість стадії визначається як різниця між її часовими мітками завершення та початку:

$$T_{\text{stage}} = t_{\text{completed}} - t_{\text{started}}. \tag{4.40}$$

Наскрізна затримка окремої задачі визначається за часовими мітками PostgreSQL:

$$T_{\mathrm{E2E},i} = t_{\mathrm{postprocess.completed},i} - t_{\mathrm{created.started},i}. \quad (4.41)$$

Отже, E2E охоплює повний шлях задачі від реєстрації в BFF до завершення післяобробки, включно з:

– очікуванням у Kafka;
– підготовкою даних;
– операціями MinIO;
– очікуванням у Redis/Celery;
– FEM-обчисленням;
– післяобробкою;
– міжсервісними затримками.

4.4.5 Критерій завершення серії

Експериментальна серія вважається завершеною, коли кожна з 30 задач перебуває в одному з термінальних станів:

– COMPLETED;
– FAILED.

Для завершених задач додатково перевіряється наявність часових метрик стадій і чисельного підсумку. Це запобігає формуванню CSV до синхронізації завершальних записів у PostgreSQL.

Після виконання зазначеної умови клієнт фіксує часову мітку seriesFinishedAt. Загальна тривалість серії обчислюється як

$$T_{\mathrm{series}} = t_{\mathrm{seriesFinishedAt}} - t_{\mathrm{seriesStartedAt}}. \quad (4.42)$$

Початок серії фіксується перед формуванням і поданням задач, а завершення – після переходу всіх задач у термінальний стан та отримання необхідних метрик.

Тайм-аут для окремої задачі або серії не встановлюється. Тому експериментальний модуль очікує фактичного завершення всіх задач. Якщо одна з них залишається у нетермінальному стані, серія автоматично не завершується і CSV не експортується до усунення причини або ручного припинення запуску.

Якщо створення задачі завершується помилкою HTTP, у наборі формується окремий рядок зі статусом FAILED. Якщо помилка виникає на одній зі стадій конвеєра, вона також реєструється у відповідному рядку. Кількість успішних і невдалих задач зберігається окремо, а статистичні часові показники обчислюються лише для успішно завершених задач.

4.4.6 Порядок проведення серій

Для експерименту конфігурації виконувалися послідовними блоками за кількістю обчислювальних виконавців. Спочатку було встановлено $N_w = 1$, після чого виконано по три серії для meshSize=0,2 і meshSize=0,08. Далі аналогічну послідовність повторено для $N_w = 2$ та $N_w = 4$. Після кожної зміни кількості реплік processor-worker, підтвердження готовності Deployment і pod витримувалася стабілізаційна пауза тривалістю 30 секунд. Між повторними серіями за незмінної кількості обчислювальних виконавців окрема фіксована пауза не встановлювалася.

Матрицю експериментів наведено в таблиці 4.7.

Таблиця 4.7 – Матриця експериментальних запусків

Сценарій	**meshSize**	**Повтори**	**N_w**	**Кількість серій**
poisson_square_mms	0,2	3	1 / 2 / 4	9
poisson_square_mms	0,08	3	1 / 2 / 4	9
poisson_cube_mms_vbc	0,2	3	1 / 2 / 4	9
poisson_cube_mms_vbc	0,08	3	1 / 2 / 4	9
Разом	–	–	–	**36**

Загальна кількість задач становить

$$N_{\text{total}} = 2 \cdot 2 \cdot 3 \cdot 3 \cdot 30 = 1080. \quad (4.43)$$

Тут перший множник відповідає кількості математичних сценаріїв, другий – кількості значень meshSize, третій – кількості повторів, четвертий – кількості конфігурацій N_w, а останній – кількості задач у серії.

4.4.7 Збереження результатів

Після завершення кожної серії формується окремий CSV-файл. Його назва містить:

– сценарій;
– значення meshSize;
– кількість обчислювальних виконавців;
– номер повтору;
– часову мітку експорту.

Приклад назви: experiment_poisson_cube_mms_vbc_mesh-0.08_workers-2_repeat-3_<timestamp>.csv.

У кожному рядку CSV повторюються параметри серії:

– label;
– repeat;
– scenarioId;
– meshSize;
– workerReplicas;
– concurrency;
– n;
– seriesStartedAt;
– seriesFinishedAt;

– seriesDurationMs;
– throughputPerMin.

Крім того, для кожної задачі зберігаються:

– taskId;
– статус;
– прогрес;
– тривалість кожної стадії;
– E2E;
– характеристики сітки;
– внутрішні часові метрики чисельного розв’язувача;
– статистичні характеристики поля;
– чисельні похибки;
– повідомлення про помилку, якщо вона виникла.

Повторення метаданих серії в кожному рядку дозволяє об’єднати всі CSV-файли в єдиний набір даних без залежності від назв файлів.

Запропонована методика забезпечує порівнюваність серій завдяки фіксації параметрів математичної постановки, однаковій кількості задач, сталому рівню паралелізму, незмінній конфігурації інших компонентів і єдиному механізму збирання метрик. Зміна лише кількості processor-worker дає змогу оцінити вплив горизонтального масштабування обчислювальної стадії на продуктивність повного FEA-конвеєра.

4.5 Метрики та методика обробки результатів

Оцінювання продуктивності та масштабованості реалізованої FEA-платформи ґрунтується на поєднанні серійних, поетапних і внутрішніх обчислювальних метрик [82, 85]. Серійні показники характеризують виконання всього пакета задач, поетапні – проходження окремої задачі через мікросервісний конвеєр, а внутрішні метрики Processor відображають тривалість безпосередніх операцій скінченно-елементного розв’язувача.

Використання декількох груп метрик необхідне тому, що прискорення повної серії не завжди супроводжується зменшенням часу окремого FEM-обчислення. Зі збільшенням кількості обчислювальних виконавців може скорочуватися очікування задач у черзі, але водночас зростати конкуренція за процесорний час і спільні інфраструктурні ресурси. Тому результати аналізуються як на рівні повного конвеєра, так і на рівні його окремих складових.

4.5.1 Джерела експериментальних даних

Для формування експериментального набору використовуються три джерела даних:

1) локальні часові мітки експериментального модуля Angular;
2) часові мітки стадій, збережені в PostgreSQL;
3) підсумкові чисельні та часові характеристики Processor, передані в події postprocess.completed.

Локальний клієнтський час використовується лише для визначення загальної тривалості пакетної серії. Перед початком створення задач фіксується seriesStartedAt, а після переходу всіх задач у термінальний стан і отримання повного набору метрик – seriesFinishedAt.

Часові характеристики окремої задачі визначаються не за моментом отримання Socket.IO-повідомлень, а за серверними часовими мітками таблиці task_stages. Це усуває з метрик вплив затримки доставки повідомлення від BFF до браузера.

Внутрішні характеристики розв’язувача формуються безпосередньо в Processor за допомогою монотонного таймера time.perf_counter() і зберігаються в мілісекундах.

Поєднання трьох джерел даних дає змогу розмежувати різні рівні спостереження за виконанням платформи. Клієнтські часові мітки характеризують повну тривалість пакетного запуску з погляду користувача, серверні часові мітки PostgreSQL відображають проходження задачі через окремі

стадії конвеєра, а внутрішні метрики Processor характеризують безпосередню вартість чисельних операцій. Такий розподіл дозволяє окремо аналізувати загальний ефект масштабування, тривалість стадій і зміну вартості виконання однієї FEM-задачі.

Основні метрики та їхні джерела наведено в таблиці 4.8.

Таблиця 4.8 – Метрики експериментального дослідження

Метрика	**Позначення**	**Джерело**
Тривалість серії	T_{series}	Angular Experiment Runner
Пропускна здатність	Q	T_{series}, кількість завершених задач
Наскрізна затримка	T_{E2E}	PostgreSQL task_stages
Тривалість підготовки даних	T_{pre}	PostgreSQL task_stages
Тривалість стадії розв'язання	$T_{solveStage}$	PostgreSQL task_stages
Тривалість післяоброблення	T_{post}	PostgreSQL task_stages
Час складання системи	$T_{assemble}$	Processor
Час розв'язання системи	$T_{linearSolve}$	Processor
Загальний час FEM-частини	T_{FEM}	Processor
Максимальна вузлова похибка	e_{max}	Processor
Вузлова RMS-похибка	e_{RMS}	Processor
Параметри сітки	N_p, N_Δ, N_{tet}	Processor
Мінімум, максимум і середнє поля	u_{min}, u_{max}, $\bar{u}$	Processor

4.5.2 Загальна тривалість серії

Загальна тривалість експериментальної серії визначається як

$$T_{\text{series}} = t_{\text{finish}} - t_{\text{start}}, \tag{4.44}$$

де t_{start} – локальний час перед початком подання першої задачі, а t_{finish} – час після переходу всіх задач у стани COMPLETED або FAILED та отримання необхідних метрик.

Таким чином, T_{series} включає:

– створення задач через BFF;

– публікацію та обробку подій Kafka;

– підготовку сіток;

– операції з MinIO;

– очікування в Redis/Celery;

– FEM-обчислення;

– післяоброблення;

– отримання клієнтом завершальних станів;

– коротку затримку опитування стану експериментальним модулем.

Тривалість серії є фактичним клієнтським часом. Вона характеризує час, протягом якого користувач очікує завершення всього пакета задач.

4.5.3 Пропускна здатність

Пропускна здатність визначається як кількість успішно завершених задач за одиницю часу [82]:

$$Q = \frac{N_{\text{completed}}}{T_{\text{series}}/60000}, \tag{4.45}$$

де T_{series} задається в мілісекундах, а Q вимірюється в задачах за хвилину.

У разі успішного завершення всіх задач:

$$N_{\text{completed}} = 30. \tag{4.46}$$

Якщо частина задач завершується помилкою, у чисельнику враховуються лише задачі зі статусом COMPLETED. Кількість невдалих задач подається окремо.

Пропускна здатність є основною метрикою для перевірки гіпотези про ефективність горизонтального масштабування. На відміну від середнього часу внутрішнього розв’язання однієї задачі, вона враховує паралельне виконання декількох незалежних задач.

4.5.4 Наскрізна затримка

Для окремої задачі наскрізна затримка визначається як

$$T_{\mathrm{E2E},i} = t_{\mathrm{post},i}^{\mathrm{completed}} - t_{\mathrm{created},i}^{\mathrm{started}}. \tag{4.47}$$

Початковою точкою є реєстрація задачі в BFF, а кінцевою – завершення стадії postprocess.

Показник охоплює як активне опрацювання задачі, так і очікування між стадіями [82, 85]. Тому загалом

$$T_{\mathrm{E2E}} \neq T_{\mathrm{pre}} + T_{\mathrm{solveStage}} + T_{\mathrm{post}}. \tag{4.48}$$

Різниця містить міжстадійні інтервали, зокрема:

– очікування події в Kafka;
– затримку між публікацією та споживанням повідомлення;
– очікування задачі в Redis/Celery;
– операції маршрутизації;
– інші інфраструктурні затримки.

Наскрізна метрика є основною характеристикою часу обслуговування окремої задачі з погляду користувача.

4.5.5 Тривалість окремих стадій

Для кожної стадії тривалість обчислюється за часовими мітками PostgreSQL:

$$T_{\text{stage},i} = t_{\text{stage},i}^{\text{completed}} - t_{\text{stage},i}^{\text{started}}. \tag{4.49}$$

Аналізуються три прикладні стадії:

$$T_{\text{pre},i}, \qquad T_{\text{solveStage},i}, \qquad T_{\text{post},i}. \tag{4.50}$$

Показник solveStageMs потрібно відрізняти від внутрішнього solveMs. Перший характеризує повну тривалість серверної стадії solve відповідно до її записів у PostgreSQL. Другий відображає лише час виклику розрідженого лінійного чисельного розв’язувача у Processor.

Отже,

$$T_{\text{solveStage}} \geq T_{\text{FEM}} \geq T_{\text{linearSolve}}, \tag{4.51}$$

для більшості задач, оскільки стадія додатково включає отримання артефактів, розбір сітки, формування результату, запис до MinIO та супровідні операції.

4.5.6 Внутрішні метрики FEM-розв’язувача

Processor формує три часові показники:

$$T_{\text{assemble}}, \qquad T_{\text{linearSolve}}, \qquad T_{\text{FEM}}. \tag{4.52}$$

AssembleMs охоплює формування глобальної матриці жорсткості та вектора правої частини:

$$T_{\text{assemble}} = t_1 - t_0. \tag{4.53}$$

SolveMs охоплює підготовку граничних значень і розв’язання скороченої розрідженої системи:

$$T_{\text{linearSolve}} = t_2 - t_1. \tag{4.54}$$

TotalMs визначається як

$$T_{\text{FEM}} = t_2 - t_0. \tag{4.55}$$

Усі три значення перетворюються до цілих мілісекунд. Через відкидання дробової частини для дуже швидких операцій можливе значення

$$T_{\text{linearSolve}} = 0 \text{ мс}. \tag{4.56}$$

Таке значення не означає відсутність операції, а свідчить, що її тривалість була меншою за одну мілісекунду.

Через окреме округлення інтервалів також не вимагається точна рівність

$$T_{\text{FEM}} = T_{\text{assemble}} + T_{\text{linearSolve}}. \tag{4.57}$$

Можлива різниця в одну мілісекунду.

Внутрішні метрики FEM-розв’язувача використовуються спільно з тривалістю серверної стадії чисельного розв’язування та наскрізною затримкою. Їх зіставлення дає змогу відокремити час корисного чисельного обчислення від очікування у черзі, операцій завантаження й збереження даних, розбору сітки та

інших накладних витрат розподіленого конвеєра. Це є важливим для інтерпретації масштабування, оскільки зростання пропускної здатності платформи може відбуватися одночасно зі збільшенням часу виконання окремої FEM-задачі.

4.5.7 Статистичне узагальнення задач однієї серії

Для кожної часової метрики успішно завершених задач формується впорядкована вибірка

$$x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(m)}, \tag{4.58}$$

де $m = N_{\text{completed}}$.

Середнє арифметичне визначається як

$$\overline{x} = \frac{1}{m}\sum_{i=1}^{m} x_i. \tag{4.59}$$

Для непарного m медіана дорівнює центральному елементу:

$$\text{med}(x) = x_{\left(\frac{m+1}{2}\right)}. \tag{4.60}$$

Для парного m:

$$\text{med}(x) = \frac{x_{\left(\frac{m}{2}\right)} + x_{\left(\frac{m}{2}+1\right)}}{2}. \tag{4.61}$$

У експериментальному модулі 95-й перцентиль визначається без інтерполяції.

Індекс елемента обчислюється як

$$k = \lfloor 0{,}95(m-1) \rfloor + 1, \tag{4.62}$$

після чого

$$p95(x) = x_{(k)}. \tag{4.63}$$

Для серії з 30 успішно завершених задач:

$$k = \lfloor 0{,}95 \cdot 29 \rfloor + 1. \tag{4.64}$$

Отже, p95 відповідає 28-му елементу впорядкованої вибірки.

Для кожної серії визначаються середнє, медіана та p95 таких показників: $T_{\mathrm{E2E}}; T_{\mathrm{pre}}; T_{\mathrm{solveStage}}; T_{\mathrm{post}}; T_{\mathrm{assemble}}; T_{\mathrm{linearSolve}}; T_{\mathrm{FEM}}$.

Середнє характеризує загальний рівень витрат, медіана – типову задачу, а p95 – верхню частину розподілу затримок [83].

4.5.8 Узагальнення повторних серій

Для кожного поєднання $(scenarioId, \mathrm{meshSize}, N_w)$ виконується три незалежні серії. Основною одиницею порівняння під час аналізу масштабування є окрема серія, а не окрема задача.

Для показника y, отриманого в трьох повторах, обчислюється середнє:

$$\overline{y} = \frac{1}{R}\sum_{r=1}^{R} y_r\,, \qquad R = 3. \tag{4.65}$$

Вибіркове стандартне відхилення визначається як

$$s_y = \sqrt{\frac{\sum_{r=1}^{R}(y_r - \overline{y})^2}{R-1}}. \quad (4.66)$$

Результат подається у вигляді

$$\overline{y} \pm s_y. \quad (4.67)$$

За повторами узагальнюються:

– пропускна здатність (throughput);
– середня E2E-затримка;
– медіана E2E;
– p95(E2E);
– середні тривалості стадій;
– внутрішні метрики чисельного розв`язувача [82].

Такий підхід зберігає відомості про варіативність окремих запусків і не підмінює три незалежні серії однією об’єднаною вибіркою зі 90 задач.

Об’єднані дані окремих задач можуть додатково використовуватися для побудови розподілів і діаграм, але висновки щодо масштабування формуються насамперед за серійними показниками.

4.5.9 Показники масштабування

Відносне прискорення за пропускною здатністю та ефективність масштабування визначаються відповідно до загальних принципів аналізу продуктивності й паралельного прискорення [82, 84].

$$S_Q(N_w) = \frac{Q(N_w)}{Q(1)}. \quad (4.68)$$

Для одного виконавця:

$$S_Q(1) = 1. \tag{4.69}$$

Ефективність масштабування обчислюється як

$$E_Q(N_w) = \frac{S_Q(N_w)}{N_w}. \tag{4.70}$$

Ідеальне лінійне масштабування відповідало б

$$S_Q(N_w) = N_w, \qquad E_Q(N_w) = 1. \tag{4.71}$$

У неекспериментальній системі ефективність очікувано буде меншою за одиницю через послідовні стадії, накладні витрати, конкуренцію за ресурси та незмінну кількість реплік інших компонентів [84].

Для p95-затримки використовується коефіцієнт відносного прискорення:

$$S_{p95}(N_w) = \frac{p95_{\mathrm{E2E}}(1)}{p95_{\mathrm{E2E}}(N_w)}. \tag{4.72}$$

Значення

$$S_{p95}(N_w) > 1 \tag{4.73}$$

означає зменшення затримки порівняно з конфігурацією одного виконавця. Відносна зміна показника обчислюється як

$$\Delta y(N_w) = \frac{y(N_w) - y(1)}{y(1)} \cdot 100\%. \tag{4.74}$$

Для пропускної здатності додатне значення означає покращення, тоді як для часових метрик покращенню відповідає від'ємне значення.

4.5.10 Метрики чисельної коректності

Для кожної задачі зберігаються: nPoints; nTriangles; nTetra; min; max; avg; maxAbsError; l2Error.

Використання зазначених показників похибки для перевірки чисельної коректності узгоджується з методологією верифікації чисельних моделей і методом виготовлених розв'язків [79–81]. У двовимірному сценарії nTriangles визначає кількість трикутних елементів, а nTetra не використовується.

У тривимірному сценарії:

– nTetra визначає кількість тетраедральних елементів об'єму;
– nTriangles визначає кількість трикутників зовнішньої поверхні, переданих для візуалізації.

Показники $u_{\min}$, $u_{\max}$, $\overline{u}$ характеризують вузлові значення отриманого поля. Чисельна коректність перевіряється за незмінністю параметрів сітки та результатів для однакових scenarioId і meshSize при різній кількості виконавців.

Цілочисельні параметри сітки мають збігатися точно:

$$\begin{aligned} N_p^{(1)} &= N_p^{(2)} = N_p^{(4)}, \\ N_\triangle^{(1)} &= N_\triangle^{(2)} = N_\triangle^{(4)}, \\ N_{tet}^{(1)} &= N_{tet}^{(2)} = N_{\text{tet}}^{(4)}. \end{aligned} \tag{4.75}$$

Числові характеристики порівнюються за критерієм

$$|a - b| \le \varepsilon_{\text{abs}} + \varepsilon_{\text{rel}}|b|, \tag{4.76}$$

де $\varepsilon_{\text{abs}} = 10^{-12}$, $\varepsilon_{\text{rel}} = 10^{-9}$. Перевірка охоплює: $u_{\min}$, $u_{\max}$, $\overline{u}$, $e_{\max}$, e_{RMS}.

4.5.11 Перевірка якості експериментальних даних

Перед включенням серії до основного аналізу перевіряється:

1) наявність 30 рядків даних;
2) відповідність scenarioId, meshSize, workerReplicas, concurrency, n і repeat запланованій конфігурації;
3) однаковість серійних метаданих у всіх рядках;
4) унікальність taskId;
5) наявність термінального статусу для кожної задачі;
6) заповнення часових і чисельних метрик для завершених задач;
7) додатне значення seriesDurationMs;
8) відповідність пропускної здатності формулі;
9) наявність nTetra для 3D і його відсутність для 2D;
10) відсутність нечислових значень у числових полях;
11) узгодженість параметрів сітки та чисельних результатів усередині серії.

Основний аналіз продуктивності виконується для серій, у яких усі 30 задач завершено успішно. Серії з помилками не вилучаються без пояснення: кількість і характер помилок фіксуються окремо, а відповідна конфігурація запускається повторно після встановлення причини.

Таким чином, методика обробки поєднує аналіз повної тривалості пакетного виконання, затримок окремих задач, тривалості стадій, внутрішніх FEM-операцій і чисельної коректності [82, 85]. Це дозволяє відокремити ефект скорочення черги від зміни вартості окремого обчислення та встановити межі ефективного горизонтального масштабування.

4.6 Результати пілотного двовимірного експерименту

Пілотний експеримент проведено для двовимірного сценарію poisson_square_mms на одиничному квадраті. Його основним призначенням була

перевірка працездатності запропонованої методики, оцінювання впливу кількості processor-worker на виконання пакетів незалежних задач і встановлення співвідношення між часом безпосереднього FEM-обчислення та накладними витратами повного мікросервісного конвеєра.

Для кожного значення параметра дискретизації виконано серії з кількістю обчислювальних виконавців $N_w \in \{1,2,4\}$.

Кожна конфігурація повторювалася тричі, у кожній серії створювалося 30 задач, а рівень паралелізму дорівнював трьом. Загалом у межах двовимірного етапу виконано $2 \cdot 3 \cdot 3 \cdot 30 = 540$ задач. Усі задачі завершено успішно; помилок створення, підготовки даних, чисельного розв'язання або післяоброблення не було зафіксовано.

4.6.1 Загальні результати пакетного виконання

У таблиці 4.9 наведено середні значення показників за трьома повторними серіями та вибіркове стандартне відхилення.

Таблиця 4.9 – Результати пакетного виконання двовимірних задач

meshSize	N_w	T_{series}, с	Throughput, задач/хв	E2E avg, мс	E2E median, мс	E2E p95, мс
0,2	1	5,92 +- 0,49	305,51 +- 24,43	2826,0 +- 183,1	2597,7 +- 180,0	4505,7 +- 228,9
0,2	2	5,30 +- 0,55	342,08 +- 33,66	2285,0 +- 262,0	2173,5 +- 315,2	3672,3 +- 265,0
0,2	4	5,41 +- 0,15	332,77 +- 9,47	2317,5 +- 128,6	2348,0 +- 79,8	3558,7 +- 81,1
0,08	1	5,73 +- 0,05	313,89 +- 2,60	2738,1 +- 221,1	2544,2 +- 345,5	4598,0 +- 260,0
0,08	2	5,30 +- 0,34	340,44 +- 21,93	2407,2 +- 81,2	2493,3 +- 242,3	3813,0 +- 92,2
0,08	4	5,56 +- 0,26	324,01 +- 15,84	2388,1 +- 218,1	2137,8 +- 228,5	3857,3 +- 363,0

Для обох рівнів дискретизації найбільше середнє значення пропускної здатності отримано при двох виконавцях. Для meshSize=0,2 воно становило 342,08 задач/хв, а для meshSize=0,08 – 340,44 задач/хв.

Перехід від одного до двох виконавців привів до одночасного збільшення пропускної здатності і зменшення E2E-затримок. Подальше збільшення кількості

виконавців до чотирьох не забезпечило стабільного додаткового приросту пропускної здатності. Це свідчить про досягнення зони насичення вже після двох обчислювальних виконавців для досліджених двовимірних задач [78, 84, 85].

Залежність пропускної здатності від кількості виконавців наведено на рисунку 4.3. Зміна p95(E2E) наведена на рисунку 4.4.

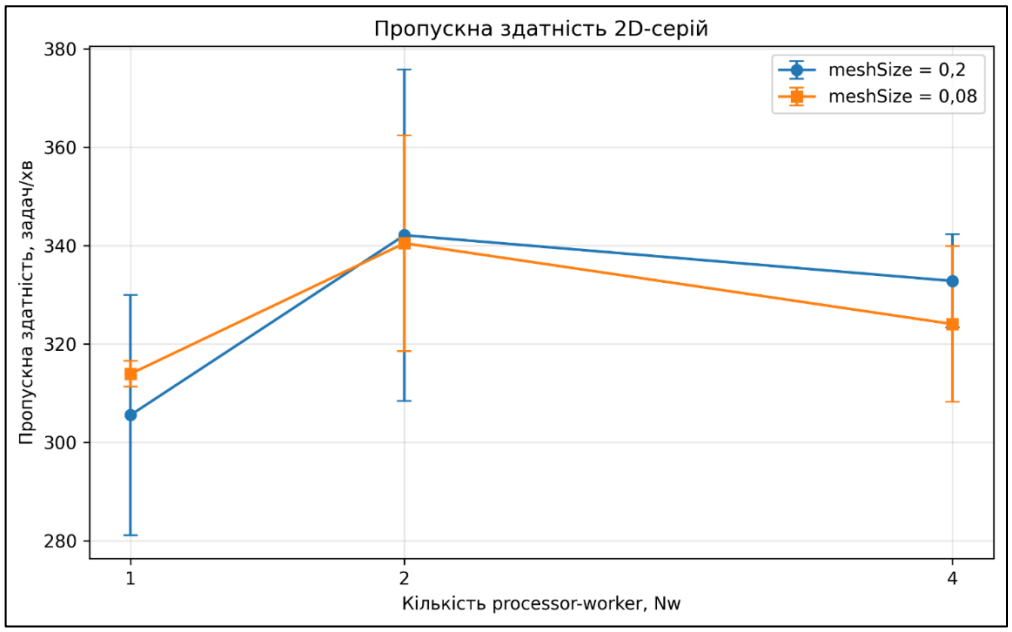


Рисунок 4.3 – Залежність пропускної здатності двовимірних серій від кількості processor-worker

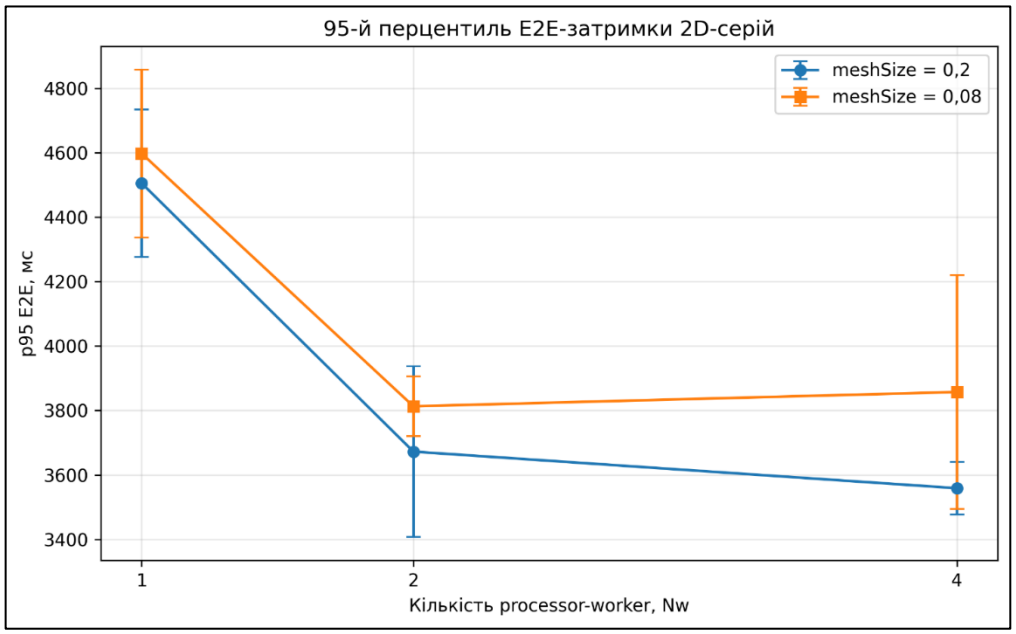


Рисунок 4.4 – Залежність 95-го перцентиля E2E-затримки двовимірних серій від кількості processor-worker

Отже, загальні результати пілотного етапу показують, що для обох досліджених двовимірних постановок основний позитивний ефект масштабування досягається при переході від одного до двох обчислювальних виконавців.

Подальше збільшення їх кількості не забезпечує стійкого приросту пропускної здатності, хоча в окремих конфігураціях може додатково впливати на верхню частину розподілу наскрізних затримок. Для детальнішого аналізу цієї поведінки далі окремо розглянуто результати для кожного значення параметра meshSize.

4.6.2 Результати для meshSize=0,2

Для грубішої сітки при одному виконавці середня пропускна здатність становила

$$Q(1) = 305{,}51 \text{ задач/хв.} \tag{4.77}$$

При переході до двох виконавців вона зросла до

$$Q(2) = 342{,}08 \text{ задач/хв,} \tag{4.78}$$

що відповідає приросту

$$\Delta Q(2) = \frac{342{,}08 - 305{,}51}{305{,}51} \cdot 100\% = 11{,}97\%. \tag{4.79}$$

Одночасно середня E2E-затримка зменшилася з 2826,0 до 2285,0 мс, тобто приблизно на 19,1%. Значення p95 скоротилося з 4505,7 до 3672,3 мс, або на 18,5%.

При чотирьох виконавцях пропускна здатність становила 332,77 задач/хв. Це на 8,92% більше, ніж при одному виконавці, але на 2,72% менше, ніж при двох. Середня E2E-затримка також виявилася дещо більшою, ніж для $N_w = 2$: 2317,5 проти 2285,0 мс.

Водночас p95 при чотирьох виконавцях зменшився до 3558,7 мс, що на 21,0% менше за базову конфігурацію. Отже, чотири виконавці дещо покращили верхню частину розподілу затримок, але не дали найбільшої пропускної здатності.

4.6.3 Результати для meshSize=0,08

Зменшення параметра сітки до 0,08 збільшило кількість вузлів із 44 до 232, а кількість трикутних елементів – із 66 до 410. Отже, кількість вузлів зросла приблизно у 5,27 раза, а кількість елементів – у 6,21 раза.

При одному виконавці середня пропускна здатність становила 313,89 задач/хв. При двох виконавцях вона збільшилася до 340,44 задач/хв, тобто на 8,46%.

Середня E2E-затримка зменшилася з 2738,1 до 2407,2 мс, або на 12,1%, а p95 – із 4598,0 до 3813,0 мс, або на 17,1%.

При переході від двох до чотирьох виконавців пропускна здатність зменшилася до 324,01 задач/хв. Вона залишалася на 3,22% вищою за конфігурацію одного виконавця, але була на 4,83% нижчою за результат двох виконавців.

Середня E2E-затримка при чотирьох виконавцях дорівнювала 2388,1 мс і майже не відрізнялася від результату $N_w = 2$. Значення p95, навпаки, дещо збільшилося з 3813,0 до 3857,3 мс. З урахуванням стандартного відхилення ця зміна не свідчить про стійке покращення або погіршення, а характеризує плато продуктивності.

4.6.4 Тривалість стадій і внутрішніх FEM-операцій

Для встановлення джерела отриманого прискорення проаналізовано серверні стадії та внутрішні метрики Processor (табл. 4.10).

Таблиця 4.10 – Середня тривалість стадій та FEM-операцій

meshSize	N_w	T_{pre}, мс	$T_{solveStage}$, мс	T_{post}, мс	assembleMs	solveMs	totalMs
0,2	1	108,6 +- 2,4	695,1 +- 76,1	16,3 +- 7,4	1,19 +- 0,10	0,00 +- 0,00	1,50 +- 0,00
0,2	2	107,7 +- 6,6	199,3 +- 33,9	22,7 +- 6,6	1,13 +- 0,09	0,03 +- 0,06	1,53 +- 0,20
0,2	4	112,2 +- 3,1	184,4 +- 5,9	19,9 +- 7,3	1,13 +- 0,09	0,00 +- 0,00	1,60 +- 0,19
0,08	1	116,7 +- 7,1	640,6 +- 90,9	14,8 +- 4,3	6,51 +- 0,25	0,11 +- 0,02	7,34 +- 0,28
0,08	2	120,3 +- 0,3	205,6 +- 6,9	18,6 +- 1,5	6,74 +- 0,05	0,17 +- 0,03	7,59 +- 0,04
0,08	4	112,3 +- 3,6	209,3 +- 40,7	13,7 +- 2,1	6,66 +- 0,16	0,10 +- 0,06	7,47 +- 0,17

Тривалості preprocess і postprocess не демонструють систематичної залежності від кількості Processor-виконавцях. Це очікувано, оскільки кількість реплік відповідних сервісів у всіх серіях залишалася сталою.

Найбільша зміна спостерігається для повної стадії solve. Для meshSize=0,2 її середня тривалість зменшилася з 695,1 мс при одному виконавці до 199,3 мс при двох і 184,4 мс при чотирьох. Для meshSize=0,08 відповідні значення становили 640,6, 205,6 і 209,3 мс.

Водночас внутрішня тривалість FEM-обчислення майже не залежить від N_w. Для грубішої сітки totalMs знаходиться в межах 1,50-1,60 мс, а для дрібнішої – 7,34-7,59 мс.

Отже, скорочення solveStageMs зумовлене не прискоренням розв’язання окремої задачі, а зменшенням часу її очікування та супровідних операцій у стадії solve. Це підтверджує, що горизонтальне масштабування в дослідженій конфігурації насамперед зменшує чергу незалежних задач.

Зменшення meshSize збільшило внутрішній FEM-час приблизно у 4,7-5,0 раза. Проте навіть для дрібнішої двовимірної сітки безпосереднє обчислення тривало близько 7,5 мс і становило лише невелику частку повної E2E-затримки.

Тому двовимірна постановка залишається недостатньо обчислювально складною для наближеного до лінійного масштабування.

4.6.5 Показники прискорення та ефективності

Розраховані показники масштабування наведено в таблиці 4.11.

Таблиця 4.11 – Показники масштабування двовимірних серій

meshSize	**N_w**	**S_Q**	**E_Q**	**Зміна throughput відносно N_w=1**	**Зміна p95 відносно N_w=1**
0,2	1	1,000	1,000	0,00 %	0,00 %
0,2	2	1,120	0,560	+11,97 %	-18,50 %
0,2	4	1,089	0,272	+8,92 %	-21,02 %
0,08	1	1,000	1,000	0,00 %	0,00 %
0,08	2	1,085	0,542	+8,46 %	-17,07 %
0,08	4	1,032	0,258	+3,22 %	-16,11 %

Значення ефективності при двох виконавцях становить приблизно 0,54-0,56, а при чотирьох зменшується до 0,26-0,27. Це істотно нижче за ідеальне лінійне масштабування.

Отриманий результат пояснюється тим, що обчислювальна частина окремої 2D-задачі є дуже короткою порівняно з повною тривалістю конвеєра. Збільшення кількості виконавців впливає лише на одну зі стадій, тоді як BFF, Preprocessor, Postprocessor, Kafka, Redis, PostgreSQL і MinIO залишаються спільними.

4.6.6 Перевірка чисельної коректності

Параметри сітки та чисельні результати були однаковими для всіх 270 задач кожного значення meshSize, незалежно від кількості виконавців і номера повтору (див. табл. 4.12).

Таблиця 4.12 – Чисельні характеристики двовимірних розв’язків

meshSize	**nPoints**	**nTriangles**	**u_{min}**	**u_{max}**	**$\bar{u}$**	**maxAbsError**	**l2Error**
0,2	44	66	0	0,9907381615	0,2865757238	0,0245990328	0,0086906123
0,08	232	410	0	0,9921343250	0,3510085952	0,0039957900	0,0012528699

При зменшенні meshSize максимальна абсолютна похибка зменшилася приблизно на 83,8%, а вузлова RMS-похибка – на 85,6%. Це відповідає очікуваному підвищенню точності при згущенні сітки.

Для кожного фіксованого meshSize значення nPoints, nTriangles, мінімуму, максимуму, середнього значення поля, maxAbsError та l2Error збігалися в усіх серіях. Таким чином, зміна кількості виконавців не вплинула на математичну постановку або результат розв’язання [79–81].

4.6.7 Узагальнення результатів пілотного етапу

Пілотне двовимірне дослідження показало, що збільшення кількості processor-worker з одного до двох:

– збільшує пропускну здатність приблизно на 8-12 %;
– зменшує середню E2E-затримку приблизно на 12-19 %;
– зменшує p95(E2E) приблизно на 17-19 %;
– суттєво скорочує повну тривалість стадії solve.

Перехід до чотирьох виконавців не дав додаткового стабільного приросту пропускної здатності.

Для обох значень meshSize максимальна пропускна здатність спостерігалася при $N_w = 2$. Це підтверджує наявність насичення та зменшення ефективності після використання двох виконавців.

Внутрішня тривалість FEM-обчислення не змінювалася зі збільшенням N_w. Отже, отримане покращення пов’язане переважно зі скороченням очікування задач у черзі, а не з прискоренням окремого чисельного розв’язання.

За результатами пілотного етапу гіпотези H1 і H2 підтверджено для переходу від одного до двох виконавців.

Гіпотезу H4 про появу насичення також підтверджено, оскільки чотири виконавці не забезпечили пропорційного приросту продуктивності.

Гіпотезу H5 підтверджено повністю: масштабування не вплинуло на чисельні результати.

Гіпотеза H3 про більш виражений ефект масштабування для складнішої задачі в межах двовимірного експерименту не отримала достатнього підтвердження. Хоча згущення сітки збільшило внутрішній FEM-час приблизно у п’ять разів, він залишався незначним порівняно з E2E-затримкою. Тому остаточна перевірка цієї гіпотези потребує використання тривимірної тетраедральної постановки з більшим обчислювальним навантаженням.

4.7 Результати основного тривимірного експерименту

Основне експериментальне дослідження проведено для сценарію poisson_cube_mms_vbc, що описує розв’язання тривимірної задачі Пуассона на одиничному кубі зі змінними неоднорідними умовами Діріхле. На відміну від пілотної двовимірної постановки, у цьому сценарії область дискретизується лінійними тетраедральними елементами, а розмір глобальної системи істотно зростає при зменшенні параметра meshSize [78, 86].

Досліджувалися два рівні дискретизації:

$$\text{meshSize} \in \{0{,}2;\, 0{,}08\}. \tag{4.80}$$

Для кожного рівня виконано серії з кількістю реплік $N_w \in \{1,2,4\}$.

Кожна конфігурація повторювалася тричі. В одній серії створювалося 30 незалежних задач, а рівень паралелізму дорівнював трьом. Загалом у межах тривимірного етапу виконано $2 \cdot 3 \cdot 3 \cdot 30 = 540$ задач.

Усі задачі успішно пройшли стадії підготовки даних, чисельного розв’язання та після оброблення. Помилок створення задач або виконання мікросервісного конвеєра у збережених експериментальних наборах не зафіксовано.

4.7.1 Загальні результати пакетного виконання

У таблиці 4.13 наведено середні показники за трьома повторними серіями та відповідні вибіркові стандартні відхилення.

Таблиця 4.13 – Результати пакетного виконання тривимірних задач

meshSize	N_w	T_{series}, с	Throughput, задач/хв	E2E avg, мс	E2E median, мс	E2E p95, мс
0,2	1	6,82 +- 0,12	264,17 +- 4,53	3192,8 +- 140,1	3158,2 +- 436,7	5168,0 +- 169,5
0,2	2	5,88 +- 0,20	306,37 +- 10,26	2757,7 +- 148,0	2794,3 +- 105,8	4423,0 +- 217,0
0,2	4	6,15 +- 0,36	293,35 +- 17,73	2763,2 +- 198,0	2519,7 +- 256,5	4595,0 +- 593,3
0,08	1	19,86 +- 0,25	90,66 +- 1,15	10012,3 +- 159,3	10036,8 +- 43,1	17640,7 +- 403,1
0,08	2	15,46 +- 1,29	116,98 +- 9,40	7690,5 +- 528,8	7226,8 +- 145,5	13626,3 +- 1450,6
0,08	4	14,68 +- 0,05	122,58 +- 0,41	7419,3 +- 250,3	7359,3 +- 522,4	13033,3 +- 242,9

Отримані результати демонструють різну поведінку двох рівнів дискретизації.

Для грубішої сітки meshSize=0,2 найбільша пропускна здатність спостерігалася при двох виконавцях. Збільшення їх кількості до чотирьох не привело до подальшого покращення.

Для дрібнішої сітки meshSize=0,08 пропускна здатнвсть послідовно зростала при переходах $1 \rightarrow 2 \rightarrow 4$, хоча приріст між двома і чотирма виконавцями був значно меншим, ніж між одним і двома.

Залежність пропускної здатності тривимірних серій від кількості обчислювальних виконавців наведено на рисунку 4.5.

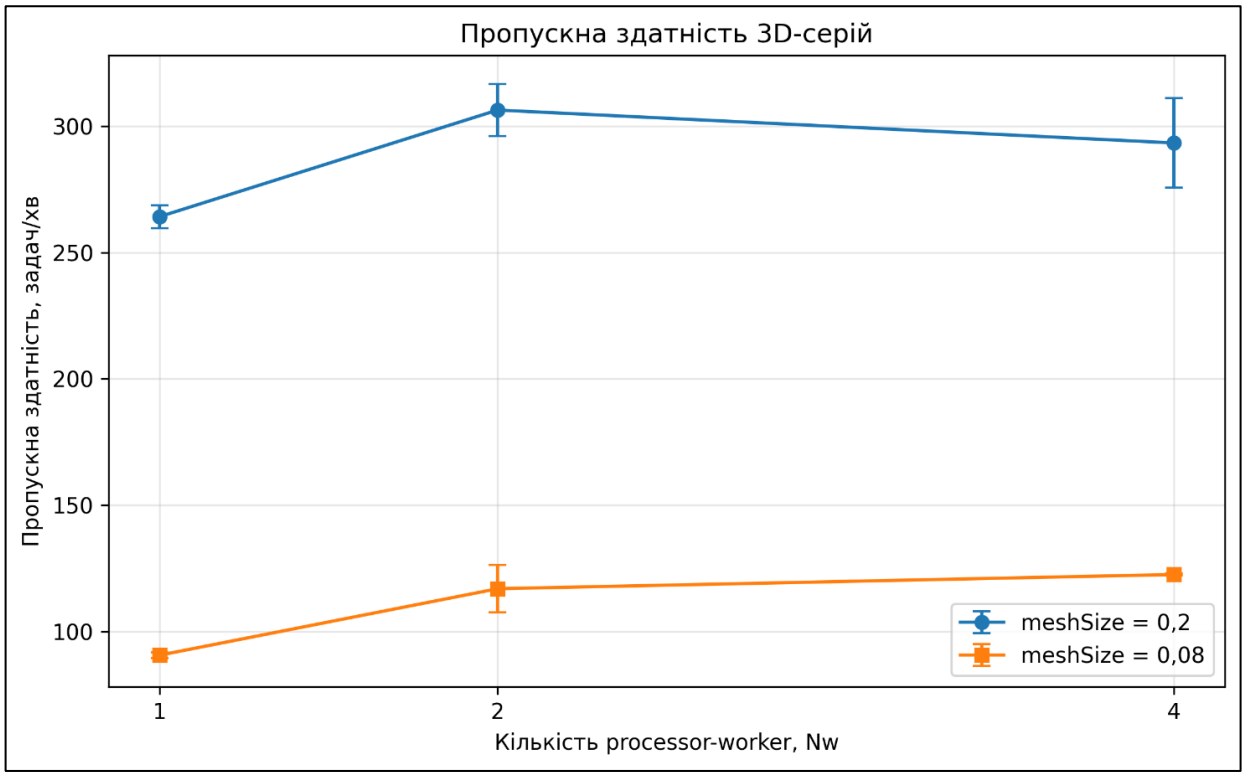


Рисунок 4.5 – Залежність пропускної здатності 3D-серій від кількості processor-worker

Зміну 95-го перцентиля E2E-затримки наведено на рисунку 4.6.

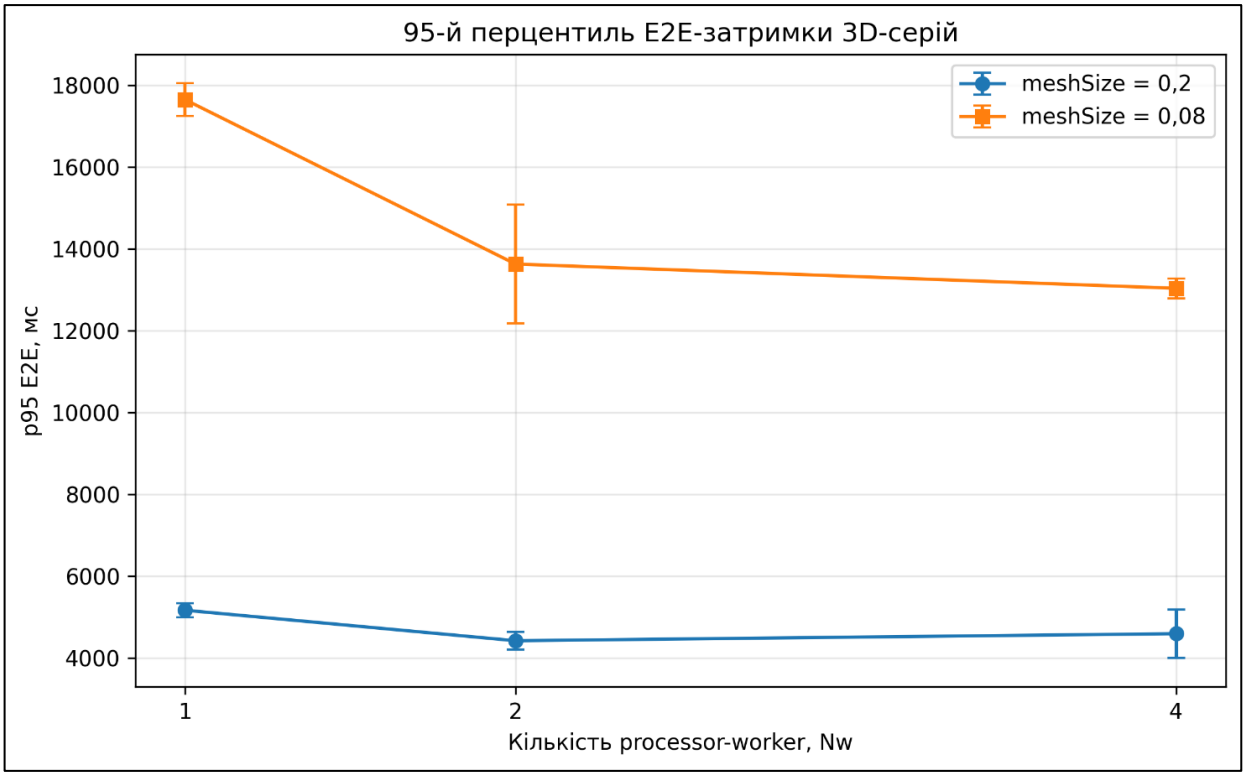


Рисунок 4.6 – Залежність p95(E2E) 3D-серій від кількості processor-worker

Отже, тривимірний експеримент демонструє різний характер масштабування залежно від рівня дискретизації. Для грубішої сітки зона насичення проявляється вже після двох обчислювальних виконавців, тоді як для дрібнішої сітки використання чотирьох виконавців забезпечує подальше покращення абсолютних показників. Водночас менший додатковий приріст при переході від двох до чотирьох виконавців свідчить про непропорційний характер масштабування та необхідність окремого аналізу тривалості стадій і внутрішніх FEM-операцій.

4.7.2 Результати для meshSize=0,2

Для грубішої тривимірної сітки при одному worker середня пропускна здатність становила

$$Q(1) = 264{,}17 \text{ задач/хв.} \tag{4.81}$$

При переході до двох виконавців вона збільшилася до

$$Q(2) = 306{,}37 \text{ задач/хв.} \tag{4.82}$$

Відносний приріст становив

$$\Delta Q(2) = \frac{306{,}37 - 264{,}17}{264{,}17} \cdot 100\% = 15{,}97\%. \tag{4.83}$$

Середня E2E-затримка зменшилася з 3192,8 до 2757,7 мс, тобто на 13,6%. Значення p95 зменшилося з 5168,0 до 4423,0 мс, або на 14,4%.

При чотирьох виконавцях пропускна здатність становила

$$Q(4) = 293{,}35 \text{ задач/хв.} \tag{4.84}$$

Це на 11,04% більше за конфігурацію одного виконавця, але на 4,25% менше за результат двох виконавцях.

Середня E2E-затримка при $N_w = 4$ становила 2763,2 мс і практично збігалася з результатом двох виконавців. Значення p95 збільшилося з 4423,0 до 4595,0 мс. З урахуванням стандартних відхилень отримані значення характеризують плато продуктивності, а не стійке покращення при переході до чотирьох виконавців.

Таким чином, для грубішої 3D-сітки найбільш доцільною з досліджених конфігурацій є $N_w = 2$. Подальше збільшення кількості виконавців не компенсує додаткову конкуренцію за ресурси фізичного хоста.

4.7.3 Результати для meshSize=0,08

Для дрібнішої сітки при одному виконавці середня тривалість серії становила 19,86 с, а пропускна здатність –

$$Q(1) = 90{,}66 \text{ задач/хв.} \tag{4.85}$$

При переході до двох виконавців тривалість серії скоротилася до 15,46 с, а пропускна здатність зросла до

$$Q(2) = 116{,}98 \text{ задач/хв.} \tag{4.86}$$

Відносний приріст становив

$$\Delta Q(2) = \frac{116{,}98 - 90{,}66}{90{,}66} \cdot 100\% = 29{,}03\%. \tag{4.87}$$

Середня E2E-затримка зменшилася з 10012,3 до 7690,5 мс, тобто на 23,2%. Значення p95 скоротилося з 17640,7 до 13626,3 мс, або на 22,8%.

При чотирьох виконавцях середня пропускна здатність збільшилася до

$$Q(4) = 122{,}58 \text{ задач/хв.} \tag{4.88}$$

Порівняно з одним виконавцем приріст становив 35,21%. Середня E2E-затримка зменшилася до 7419,3 мс, тобто на 25,9%, а p95 – до 13033,3 мс, або на 26,1%.

Водночас перехід від двох до чотирьох виконавців дав лише

$$\frac{122{,}58 - 116{,}98}{116{,}98} \cdot 100\% = 4{,}79\% \tag{4.89}$$

додаткової пропускної здатності.

Значення p95 при цьому зменшилося лише на 4,35%. Отже, збільшення кількості виконавців удвічі не забезпечило пропорційного приросту продуктивності. Це свідчить про наближення до насичення, хоча для дрібнішої сітки чотири виконавці усе ще забезпечують найкращий абсолютний результат.

4.7.4 Тривалість стадій та внутрішніх FEM-операцій

Для пояснення отриманих залежностей проаналізовано тривалості серверних стадій і внутрішніх операцій розв’язувача (табл. 4.14).

Таблиця 4.14 – Середня тривалість стадій і FEM-операцій для 3D-задач

meshSize	N_w	T_{pre}, мс	$T_{solveStage}$, мс	T_{post}, мс	assembleMs	solveMs	totalMs
0,2	1	124,7 +- 4,8	825,8 +- 120,7	17,4 +- 6,2	28,61 +- 0,37	0,02 +- 0,04	29,26 +- 0,43
0,2	2	132,4 +- 4,6	221,2 +- 8,3	16,9 +- 4,8	32,01 +- 0,49	0,03 +- 0,03	32,64 +- 0,59
0,2	4	135,4 +- 4,6	235,1 +- 23,5	22,0 +- 8,0	32,48 +- 2,39	0,09 +- 0,13	33,23 +- 2,54
0,08	1	280,5 +- 10,7	5588,1 +- 111,2	43,5 +- 1,3	376,10 +- 9,00	22,23 +- 1,20	398,78 +- 9,81
0,08	2	348,8 +- 20,8	1908,7 +- 188,6	56,8 +- 4,5	578,01 +- 13,68	32,36 +- 3,49	610,89 +- 15,35
0,08	4	393,1 +- 7,0	1293,6 +- 52,5	54,3 +- 5,8	733,02 +- 56,60	49,73 +- 13,42	783,22 +- 69,33

Для meshSize=0,2 середня тривалість повної стадії solve зменшилася з 825,8 мс при одному виконавці до 221,2 мс при двох. При чотирьох виконавцях вона дещо збільшилася до 235,1 мс, що узгоджується з відсутністю додаткової приросту пропускної здатності.

Внутрішній час FEM-частини для цієї сітки залишався малим і становив приблизно 29-33 мс. Тому навіть тривимірна задача з meshSize=0,2 значною мірою визначається накладними витратами конвеєра.

Для meshSize=0,08 стадія solve скоротилася з 5588,1 мс при одному виконавці до 1908,7 мс при двох і 1293,6 мс при чотирьох. Це пояснює суттєве зменшення E2E-затримок і зростання пропускної здатності.

Водночас час безпосереднього FEM-обчислення однієї задачі збільшувався зі зростанням кількості виконавців: 398,8 мс → 610,9 мс → 783,2 мс.

Збільшення totalMs при незмінній математичній постановці узгоджується з припущенням про конкуренцію паралельних процесів обчислювальних виконавців за процесорний час, оперативну пам’ять та інші спільні ресурси фізичного хоста. Оскільки під час експериментальних серій окремі системні показники використання процесора, оперативної пам’яті, диска та мережі не збиралися, це пояснення розглядається як обґрунтована інтерпретація часових результатів, а не як безпосередньо виміряний причинний зв’язок [85].

Залежність внутрішнього часу totalMs від N_w наведено на рисунку 4.7.

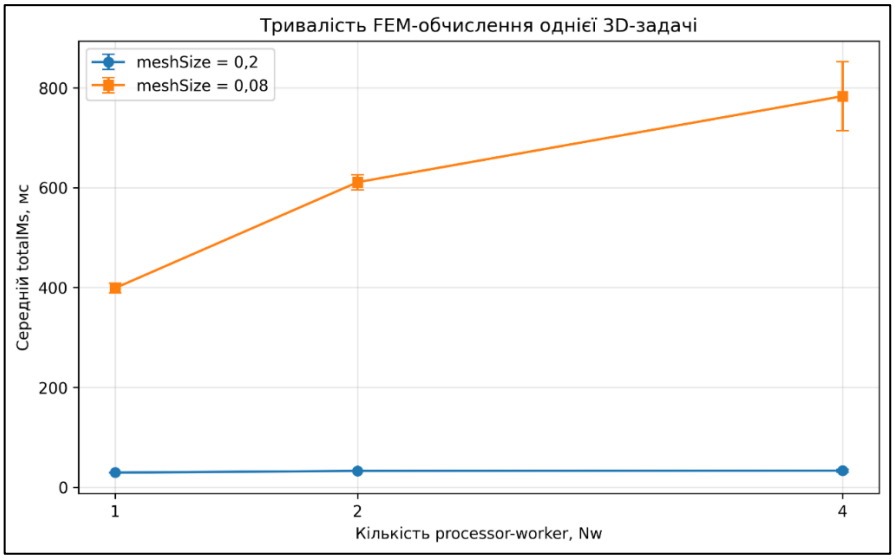


Рисунок 4.7 – Середня тривалість FEM-обчислення однієї 3D-задачі

Отже, горизонтальне масштабування скорочує час очікування задач перед обчислювальною стадією, але одночасно може збільшувати тривалість виконання кожного окремого FEM-розрахунку через конкуренцію за спільні ресурси.

Для дослідженої складної тривимірної постановки до чотирьох виконавців ефект скорочення черги залишається сильнішим, тому загальна пропускна здатність продовжує зростати. Водночас зменшення додаткового приросту між двома і чотирма виконавцями свідчить про наближення до межі ефективності експериментального стенда.

4.7.5 Показники прискорення та ефективності

Показники масштабування, обчислені відносно конфігурації одного виконавця, наведено в таблиці 4.15.

Таблиця 4.15 – Прискорення та ефективність масштабування 3D-серій

meshSize	**N_w**	**S_Q**	**E_Q**	**Зміна throughput**	**Зміна p95(E2E)**
0,2	1	1,000	1,000	0,00 %	0,00 %
0,2	2	1,160	0,580	+15,97 %	-14,42 %
0,2	4	1,110	0,278	+11,04 %	-11,09 %
0,08	1	1,000	1,000	0,00 %	0,00 %
0,08	2	1,290	0,645	+29,03 %	-22,76 %
0,08	4	1,352	0,338	+35,21 %	-26,12 %

Для грубішої сітки максимальне прискорення за пропускною здатністю становило

$$S_Q(2) = 1{,}160. \tag{4.90}$$

При чотирьох виконавцях воно зменшилося до 1,110, а ефективність – до 0,278.

Для дрібнішої сітки прискорення становило

$$S_Q(2) = 1{,}290,$$
$$S_Q(4) = 1{,}352. \quad (4.91)$$

Ефективність при чотирьох виконавцях дорівнювала лише 0,338. Тобто чотири виконавці забезпечили приблизно 35% приріст пропускної здатності, а не чотириразове прискорення.

Отримані значення підтверджують, що масштабування є непропорційний [84, 85]. Причинами цього є:

– незмінна кількість реплік Preprocessor і Postprocessor;
– спільне використання Kafka, Redis, PostgreSQL і MinIO;
– обмеження одного фізичного процесора;
– зростання часу окремого FEM-обчислення через конкуренцію виконавців;
– накладні витрати обміну подіями й артефактами.

Залежність прискорення та ефективності від кількості виконавців наведено на рисунку 4.8.

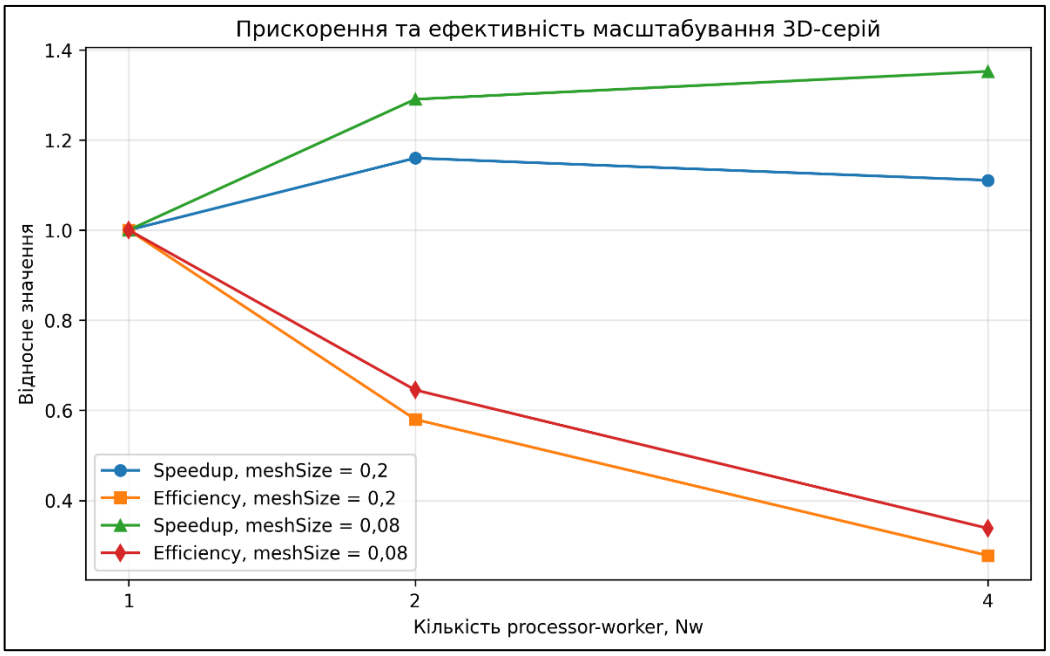


Рисунок 4.8 – Прискорення та ефективність масштабування 3D-серій

Отже, збільшення кількості обчислювальних виконавців забезпечує позитивний абсолютний ефект для складнішої тривимірної постановки, але не приводить до пропорційного прискорення. Зниження ефективності зі збільшенням кількості виконавців підтверджує, що результат масштабування визначається не лише паралельністю Processor, а й незмінними стадіями конвеєра, накладними витратами взаємодії та конкуренцією за спільні ресурси. Тому для подальшої інтерпретації необхідно зіставити ефект масштабування з обчислювальною складністю задачі.

4.7.6 Вплив складності задачі на масштабування

Порівняння двох тривимірних сіток підтверджує, що ефект масштабування залежить від обчислювальної складності задачі.

Для meshSize=0,2 сітка містила:

$$N_p = 235,$$
$$N_{tet} = 734,$$
$$N_{\triangle} = 396. \quad (4.92)$$

Для meshSize=0,08:

$$N_p = 2319,$$
$$N_{tet} = 10427,$$
$$N_{\triangle} = 2424. \quad (4.93)$$

При згущенні сітки кількість вузлів збільшилася приблизно у 9,87 раза, кількість поверхневих трикутників – у 6,12 раза, а кількість тетраедрів – у 14,21 раза.

При одному виконавці внутрішній FEM-час збільшився з 29,26 до 398,78 мс, тобто приблизно у 13,63 раза.

Для грубішої сітки перехід від одного до чотирьох виконавців забезпечив приріст пропускної здатності лише на 11,04%. Для дрібнішої сітки відповідний приріст становив 35,21%.

Аналогічно p95(E2E) зменшився:

– на 11,09% для meshSize=0,2;

– на 26,12% для meshSize=0,08.

Отже, складніша задача краще використовує додаткову паралельність, оскільки більша частина повного часу припадає на обчислювальну стадію [84]. Для легших задач накладні витрати інших компонентів обмежують ефект масштабування значно раніше.

Порівняння з пілотним двовимірним експериментом також підтверджує цю закономірність. У 2D-сценарії максимальна пропускна здатність для обох значень meshSize спостерігалася вже при двох виконавцях, тоді як для складнішої 3D-сітки meshSize=0,08 чотири виконавці дали додаткове, хоча й невелике, покращення.

4.7.7 Чисельні характеристики розв’язків

Параметри сітки та чисельні результати були однаковими для всіх 270 задач кожного рівня дискретизації незалежно від N_w і номера повтору (табл. 4.16).

Таблиця 4.16 – Чисельні характеристики тривимірних розв’язків

meshSize	nPoints	nTriangles	nTetra	u_{min}	u_{max}	$\bar{u}$	maxAbsError	l2Error
0,2	235	396	734	0	1,0657219709	0,2162046720	0,0842780291	0,0160528922
0,08	2319	2424	10427	0	1,1411978506	0,3113450216	0,0152364502	0,0042363857

При зменшенні meshSize максимальна абсолютна похибка скоротилася приблизно на 81,92%, а вузлова RMS-похибка – приблизно на 73,61%.

Це підтверджує підвищення точності чисельного розв’язку при згущенні тетраедральної сітки.

Для кожного фіксованого значення meshSize усі параметри сітки, статистичні характеристики поля та похибки збігалися в усіх серіях. Отже, зміна кількості виконавців вплинула лише на часові характеристики виконання, але не змінила математичний результат.

4.7.8 Узагальнення результатів основного експерименту

Основний тривимірний експеримент показав, що ефективність горизонтального масштабування визначається складністю задачі.

Для грубішої сітки meshSize=0,2:

– перехід від одного до двох виконавців збільшив пропускну здатність на 15,97%;
– p95(E2E) зменшився на 14,42%;
– чотири виконавці не дали додаткового покращення порівняно з двома;
– найкращою конфігурацією є $N_w = 2$.

Для дрібнішої сітки meshSize=0,08:

– два виконавці збільшили пропускну здатність на 29,03%;
– чотири виконавці – на 35,21%;
– p95(E2E) зменшився відповідно на 22,76% і 26,12%;
– перехід від двох до чотирьох виконавців дав лише 4,79 % додаткової пропускної здатності.

Результати підтверджують гіпотезу H1 про збільшення пропускної здатності при масштабуванні, однак лише до межі, що залежить від складності задачі.

Гіпотезу H2 про зменшення E2E-затримки підтверджено для переходу від одного до двох виконавців і для складнішої постановки при переході до чотирьох.

Гіпотезу H3 підтверджено: дрібніша тривимірна сітка продемонструвала істотно більший приріст пропускної здатності і більше зменшення p95, ніж грубіша 3D-сітка та двовимірні постановки.

Гіпотезу H4 також підтверджено. Подвоєння кількості виконавців із двох до чотирьох не дало пропорційного прискорення, а для грубішої сітки навіть супроводжувалося невеликим зниженням пропускної здатності. Для дрібнішої сітки внутрішній час виконання однієї задачі зростав через конкуренцію за ресурси.

Гіпотезу H5 попередньо підтверджено: у всіх 540 задачах чисельні характеристики залишалися незмінними для фіксованих параметрів постановки. Детальне узагальнення чисельної коректності та відтворюваності двовимірного і тривимірного етапів наведено в наступному підрозділі.

4.8 Перевірка чисельної коректності та відтворюваності результатів

Експериментальне дослідження масштабованості має підтверджувати не лише зміну часових характеристик платформи, а й збереження математичного результату при зміні конфігурації виконання. Збільшення кількості реплік processor-worker змінює порядок і паралельність опрацювання задач, але не повинно впливати на геометрію області, сформовану сітку, граничні умови, глобальну систему рівнянь або отриманий чисельний розв’язок [1, 5, 36, 79–81].

Перевірку проведено за результатами 36 основних експериментальних серій: 2 сценарії · 2 значення meshSize · 3 значення N_w · 3 повтори $= 36$ серій.

Кожна серія містила 30 задач, тому загальна кількість перевірених результатів становила $36 \cdot 30 = 1080$.

Із них 540 задач належали до двовимірного сценарію poisson_square_mms, а 540 – до тривимірного сценарію poisson_cube_mms_vbc.

4.8.1 Перевірка повноти експериментальних даних

Перед порівнянням чисельних результатів перевірено цілісність сформованого набору даних.

У кожній із 36 серій наявні 30 рядків, що відповідають 30 створеним задачам. Усі 1080 значень taskId є унікальними.

Усі задачі завершено зі статусом COMPLETED. Помилок створення задач, формування сітки, чисельного розв'язання або післяоброблення в основних експериментальних серіях не виявлено.

Для кожної завершеної задачі були заповнені:

– часові характеристики стадій;
– наскрізна затримка;
– кількість вузлів і елементів;
– внутрішні часові характеристики FEM-розв'язувача;
– мінімальне, максимальне й середнє вузлові значення;
– maxAbsError;
– l2Error.

Для тривимірних задач додатково було заповнено поле nTetra. Для двовимірних задач це поле закономірно залишалося порожнім.

Отже, всі основні серії придатні як для аналізу продуктивності, так і для перевірки чисельної коректності.

4.8.2 Незалежність результату від кількості виконавців

Для кожної фіксованої пари (scenarioId, meshSize) було отримано 270 чисельних результатів:

$$3 \text{ значення } N_w \cdot 3 \text{ повтори} \cdot 30 \text{ задач} = 270. \tag{4.94}$$

У межах кожної такої групи порівнювалися:

– nPoints;
– nTriangles;
– nTetra;
– min;

– max;
– avg;
– maxAbsError;
– l2Error.

Результати перевірки наведено в таблиці 4.17.

Таблиця 4.17 – Чисельні характеристики тестових постановок

Сценарій	meshSize	nPoints	nTriangles	nTetra	min	max	avg	maxAbsError	l2Error
2D	0,2	44	66	-	0	0,9907381615	0,2865757238	0,0245990328	0,0086906123
2D	0,08	232	410	-	0	0,9921343250	0,3510085952	0,0039957900	0,0012528699
3D	0,2	235	396	734	0	1,0657219709	0,2162046720	0,0842780291	0,0160528922
3D	0,08	2319	2424	10427	0	1,1411978506	0,3113450216	0,0152364502	0,0042363857

Для кожного рядка табл. 4.17 наведено єдине значення, оскільки всі контрольні характеристики, включені до експериментального CSV, збігалися у 270 задачах відповідної конфігурації математичної постановки.

Зокрема, при переході між

$$N_w = 1, \quad N_w = 2, \quad N_w = 4 \tag{4.95}$$

не змінювалися ані параметри сітки, ані статистичні характеристики чисельного поля, ані показники похибки. Максимальний розмах кожного порівнюваного числового показника всередині відповідної групи дорівнював нулю:

$$\max(x) - \min(x) = 0. \tag{4.96}$$

Отже, результати не лише задовольнили раніше визначений критерій наближеної рівності

$$|a - b| \leq 10^{-12} + 10^{-9}|b|, \tag{4.97}$$

а й повністю збіглися у значеннях, записаних до CSV.

Це означає, що в межах використаного програмного й апаратного середовища не виявлено впливу паралельного виконання на контрольні параметри сітки, статистичні характеристики поля та показники похибки, збережені в експериментальному наборі.

У тривимірному сценарії nTriangles характеризує кількість трикутників зовнішньої поверхні, що використовуються для візуалізації, тоді як nTetra визначає кількість об'ємних скінченних елементів, за якими виконується розв'язання.

Поле avg є звичайним арифметичним середнім вузлових значень:

$$\overline{u} = \frac{1}{N_p} \sum_{i=1}^{N_p} u_i. \tag{4.98}$$

Воно не є інтегральним середнім поля за областю. Тому середнє значення може змінюватися при згущенні та перебудові неструктурованої сітки. Для перевірки відтворюваності воно порівнюється лише між задачами з однаковими scenarioId і meshSize.

4.8.3 Зменшення похибки у двовимірній постановці

Для двовимірної задачі зменшення meshSize з 0,2 до 0,08 супроводжувалося збільшенням кількості вузлів із 44 до 232, а кількості трикутників — із 66 до 410.

Максимальна абсолютна вузлова похибка зменшилася з

$$e_{\max}(0{,}2) = 0{,}0245990328 \tag{4.99}$$

до

$$e_{\max}(0{,}08) = 0{,}0039957900. \tag{4.100}$$

Відносне зменшення становило

$$\left(1 - \frac{0{,}0039957900}{0{,}0245990328}\right) \cdot 100\% \approx 83{,}76\%. \tag{4.101}$$

Вузлова RMS-похибка зменшилася з

$$e_{\mathrm{RMS}}(0{,}2) = 0{,}0086906123 \tag{4.102}$$

до

$$e_{\mathrm{RMS}}(0{,}08) = 0{,}0012528699, \tag{4.103}$$

що відповідає зменшенню на 85,58%. Відношення похибок дорівнює:

$$\frac{e_{max}(0{,}2)}{e_{max}(0{,}08)} \approx 6{,}156,$$
$$\frac{e_{RMS}(0{,}2)}{e_{RMS}(0{,}08)} \approx 6{,}937. \tag{4.104}$$

Таким чином, згущення двовимірної сітки істотно підвищило точність чисельного розв’язку.

4.8.4 Зменшення похибки у тривимірній постановці

У тривимірній постановці зменшення meshSize з 0,2 до 0,08 збільшило кількість вузлів із 235 до 2319, а кількість тетраедрів – із 734 до 10427.

Максимальна абсолютна похибка зменшилася з

$$e_{\max}(0{,}2) = 0{,}0842780291 \tag{4.105}$$

до

$$e_{\max}(0{,}08) = 0{,}0152364502. \tag{4.106}$$

Її відносне зменшення становило 81,92%. Вузлова RMS-похибка зменшилася з

$$e_{\mathrm{RMS}}(0{,}2) = 0{,}0160528922 \tag{4.107}$$

до

$$e_{\mathrm{RMS}}(0{,}08) = 0{,}0042363857, \tag{4.108}$$

тобто на 73,61%.

Відношення похибок становило:

$$\frac{e_{\max}(0{,}2)}{e_{\max}(0{,}08)} \approx 5{,}531,$$
$$\frac{e_{RMS}(0{,}2)}{e_{RMS}(0{,}08)} \approx 3{,}789. \tag{4.109}$$

Отже, згущення тетраедральної сітки також забезпечило суттєве підвищення точності.

Порівняння похибок для обох сценаріїв наведено на рисунку 4.9.

Таким чином, для обох тестових постановок згущення сітки супроводжується узгодженим зменшенням максимальної абсолютної та вузлової середньоквадратичної похибок. Одночасно для кожного фіксованого поєднання scenarioId і meshSize значення цих показників залишаються незмінними між повторними серіями та конфігураціями з різною кількістю обчислювальних виконавців.

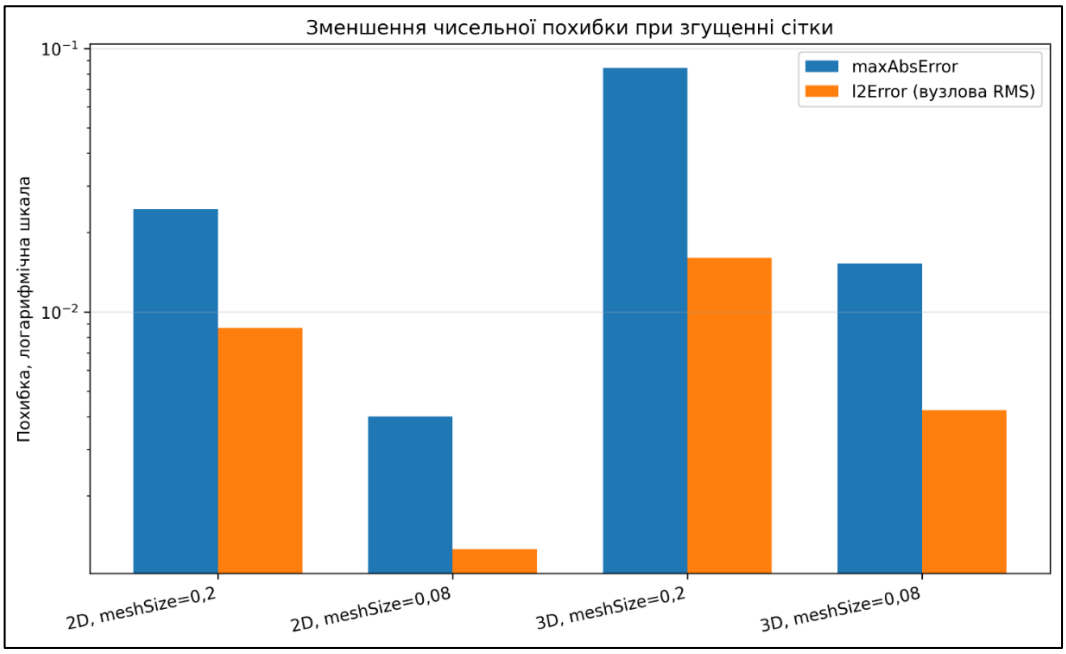


Рисунок 4.9 – Зменшення чисельної похибки при згущенні двовимірної та тривимірної сіток

Це підтверджує, що зміна конфігурації горизонтального масштабування впливає на часові характеристики виконання, але не змінює контрольні характеристики чисельного результату [79–81].

4.8.5 Орієнтовна оцінка порядку зменшення похибки

За двома рівнями дискретизації можна обчислити орієнтовний емпіричний показник [1, 5, 36, 37]:

$$p = \frac{\ln\left(e_{0,2}/e_{0,08}\right)}{\ln(0{,}2/0{,}08)}. \tag{4.110}$$

Для двовимірної задачі отримано:

$$p_{\max}^{2D} \approx 1{,}984,$$
$$p_{\mathrm{RMS}}^{2D} \approx 2{,}114. \tag{4.111}$$

Обидва значення близькі до другого порядку зменшення похибки. Для тривимірної задачі:

$$p_{\max}^{3D} \approx 1{,}867,$$
$$p_{RMS}^{3D} \approx 1{,}454. \quad (4.112)$$

Отримані значення підтверджують систематичне зменшення похибки при згущенні сітки, але не повинні трактуватися як повноцінне дослідження асимптотичної збіжності.

Це пояснюється тим, що:

– використано лише два значення meshSize;
– meshSize є характерним параметром генератора Gmsh, а не точним максимальним діаметром кожного елемента;
– сітки є неструктурованими та не вкладеними;
– l2Error є вузловою RMS-похибкою, а не інтегральною $L_2(\Omega)$-нормою;
– права частина інтегрується одновузловою квадратурою в центроїдах елементів.

Тому наведені значення p використовуються лише як додаткова характеристика тенденції, а основним критерієм коректності є стійке зменшення обох показників похибки при згущенні сітки.

4.8.6 Відтворюваність чисельних результатів

Відтворюваність перевірялася на трьох рівнях.

На першому рівні порівнювалися 30 задач усередині однієї серії. Усі задачі з однаковими scenarioId і meshSize мали однакові характеристики сітки та чисельні показники.

На другому рівні порівнювалися три повторні серії для однакової кількості виконавців. Результати також повністю збігалися.

На третьому рівні порівнювалися конфігурації з одним, двома та чотирма виконавцями. Зміна кількості паралельних виконавців не спричинила жодної зміни збережених чисельних показників.

Отже, для кожної фіксованої постановки виконується рівність векторів контрольних характеристик:

$$\mathrm{m}^{(1)} = \mathrm{m}^{(2)} = \mathrm{m}^{(4)}, \tag{4.113}$$

де $\mathrm{m}^{(N_w)}$ – вектор чисельних характеристик, отриманих при відповідній кількості виконавців:

$$\mathrm{m} = \left[N_p, N_{\triangle}, N_{\mathrm{tet}}, u_{\min}, u_{\max}, \overline{u}, e_{\max}, e_{\mathrm{RMS}}\right]. \tag{4.114}$$

Повне збігання результатів пояснюється тим, що окремі задачі є незалежними. Кожна з них використовує власні артефакти, власний taskId і окремо сформовану систему рівнянь. Виконавці не виконують паралельних операцій над однією глобальною матрицею або спільним вектором розв’язку. Зміна N_w впливає на те, який виконавець і в який момент виконує задачу, але не змінює послідовний алгоритм складання та розв’язання всередині однієї задачі.

Отримана відтворюваність стосується зафіксованого програмного, контейнерного й апаратного середовища. Вона не доводить побітову однаковість результатів між іншими процесорними архітектурами, версіями SciPy, реалізаціями BLAS або іншими операційними системами. Водночас для досліджуваного стенда результати є повністю детермінованими.

4.8.7 Узагальнення перевірки коректності

Перевірка 1080 чисельних результатів показала, що:

1) усі задачі завершено успішно;

2) для кожної фіксованої постановки характеристики сітки були однаковими в усіх серіях;
3) кількість виконавців не впливала на мінімальне, максимальне та середнє вузлові значення;
4) maxAbsError і l2Error не залежали від N_w;
5) повторні серії відтворювали однаковий чисельний результат;
6) згущення 2D- і 3D-сіток приводило до суттєвого зменшення похибки;
7) зміна продуктивності при масштабуванні не була пов’язана зі зміною математичного результату.

Таким чином, гіпотезу H5 підтверджено в межах використаних критеріїв. Горизонтальне масштабування компонента processor-worker змінює часові характеристики і порядок виконання задач, але не впливає на контрольні характеристики сітки, чисельного поля та похибки, зафіксовані в експериментальному наборі.

4.9 Обговорення результатів та загрози валідності

Проведене дослідження дало змогу оцінити поведінку подієво-орієнтованої FEA-платформи при горизонтальному масштабуванні обчислювального компонента та встановити залежність ефекту масштабування від складності задачі. Отримані результати показують, що збільшення кількості processor-worker не забезпечує пропорційного прискорення в усіх конфігураціях. Найбільший ефект спостерігається тоді, коли обчислювальна стадія становить істотну частину повного часу проходження задачі [11, 84, 85].

4.9.1 Інтерпретація результатів масштабування

Для двовимірних задач перехід від одного до двох виконавців збільшив пропускну здатність приблизно на 8-12 % і зменшив p95(E2E) приблизно на 17-

19 %. Подальше збільшення кількості виконавців до чотирьох не забезпечило додаткового стабільного приросту пропускної здатності.

Внутрішня FEM-частина двовимірних задач тривала від приблизно 1,5 до 7,6 мс. Це значно менше за повну E2E-затримку, яка вимірювалася тисячами мілісекунд. Отже, для таких постановок продуктивність визначається переважно не швидкістю чисельного розв’язання, а накладними витратами всього розподіленого конвеєра:

– реєстрацією задач у BFF;
– передаванням подій через Kafka;
– побудовою сітки;
– операціями з MinIO;
– передаванням задач через Redis і Celery;
– оновленням записів PostgreSQL;
– післяобробленням;
– доставкою завершальних подій до клієнта.

Збільшення кількості виконавців впливає лише на обчислювальну частину цього шляху. Тому після скорочення черги на стадії Processor інші компоненти починають обмежувати загальну продуктивність.

Подібну поведінку продемонструвала тривимірна задача з meshSize=0,2. Перехід від одного до двох виконавців збільшило пропускну здатність приблизно на 16 %, але використання чотирьох виконавців не дало подальшого покращення. Для цієї постановки середній внутрішній FEM-час становив приблизно 29-33 мс, що також є відносно малим порівняно з повною E2E-затримкою.

Інша залежність спостерігалася для тривимірної задачі з meshSize=0,08. При одному виконавці середній FEM-час становив близько 399 мс, а повна стадія solve – близько 5,6 с. Перехід до двох виконавців збільшив пропускну здатність приблизно на 29 %, а до чотирьох – на 35 % порівняно з базовою конфігурацією.

Таким чином, складніша задача краще використовувала додаткову паралельність. Водночас перехід від двох до чотирьох виконавців дав лише близько 4,8 % додаткової пропускної здатності. Це свідчить, що навіть для

найбільш обчислювально навантаженої постановки досліджуваний стенд наближався до насичення.

4.9.2 Конкуренція за ресурси фізичного хоста

Важливим результатом є зростання внутрішнього часу виконання однієї складної FEM-задачі зі збільшенням кількості виконавців. Для тривимірної сітки з meshSize=0,08 значення totalMs змінювалося приблизно так:

$$399 \text{ мс} \rightarrow 611 \text{ мс} \rightarrow 783 \text{ мс}. \tag{4.115}$$

Отже, окремий чисельний розрахунок виконувався повільніше при збільшенні кількості паралельних виконавців.

Цей результат узгоджується з конфігурацією стенда. Хоча Kubernetes-кластер містив три логічні вузли, усі вони були контейнерами k3d на одному фізичному ноутбуці. Відповідно, всі компоненти платформи спільно використовували:

- шість фізичних ядер і дванадцять логічних потоків процесора;
- оперативну пам’ять;
- один NVMe-накопичувач;
- мережевий стек Docker;
- ресурси операційної системи.

Для контейнерів processor-worker не були встановлені resources.requests і resources.limits. Тому Kubernetes не резервував для кожного виконавця окрему кількість процесорних ресурсів, а паралельні процеси конкурували між собою та з Kafka, Redis, PostgreSQL, MinIO, Preprocessor, Postprocessor і BFF [69, 85].

Таким чином, збільшення N_w мало два протилежні ефекти:

1) зменшення часу очікування незалежних задач перед обчисленням;
2) збільшення тривалості обчислення окремої задачі через конкуренцію за CPU, пам’ять та операції введення-виведення.

Для складної 3D-постановки до чотирьох виконавців перший ефект переважав другий, тому загальна пропускна здатність зростала. Водночас зниження ефективності масштабування свідчить, що при подальшому збільшенні кількості виконавців приріст міг би повністю зникнути або змінитися погіршенням продуктивності.

Оскільки окремі показники завантаження CPU, RAM, диска та мережі під час серій не збиралися, ресурсна конкуренція є обґрунтованою інтерпретацією часових результатів, але не була безпосередньо підтверджена системними метриками.

4.9.3 Узагальнення перевірки гіпотез

Гіпотезу H1 про підвищення пропускної здатності при збільшенні кількості processor-worker підтверджено частково.

Для всіх досліджених постановок перехід від одного до двох виконавців підвищувало пропускну здатність. Водночас перехід від двох до чотирьох давав додатковий позитивний ефект лише для найбільш складної тривимірної задачі. Для легших постановок пропускна здатність при чотирьох виконавцях була нижчою, ніж при двох.

Отже, залежність має не монотонно лінійний, а насичуваний характер: $Q(N_w) \uparrow$ до певного рівня, після чого приріст зменшується.

Гіпотезу H2 про зменшення середньої та p95 E2E-затримки також підтверджено переважно для переходу від одного до двох виконавців. Для складної 3D-задачі з meshSize=0,08 додаткове зменшення затримки спостерігалося і при чотирьох виконавцях. Для легших задач значення при $N_w = 2$ і $N_w = 4$ були близькими або частково перекривалися з урахуванням стандартного відхилення.

Гіпотезу H3 про більший ефект масштабування для складніших задач підтверджено. Найбільший приріст пропускної здатності і найбільше зменшення

p95(E2E) отримано для тривимірної сітки з найбільшою кількістю вузлів і тетраедрів.

Гіпотезу H4 про зменшення віддачі та появу насичення підтверджено. Подвоєння кількості виконавців із двох до чотирьох не забезпечило пропорційного прискорення в жодній конфігурації.

Гіпотезу H5 про незалежність контрольних чисельних характеристик від кількості виконавців підтверджено для всіх досліджених постановок і конфігурацій.

4.9.4 Внутрішня валідність експерименту

Внутрішня валідність характеризує, наскільки зафіксовані зміни продуктивності можна пов’язати саме зі зміною кількості виконавців, а не з іншими факторами.

Для зменшення впливу сторонніх чинників під час основних серій:

- ноутбук був під’єднаний до мережі живлення;
- використовувався однаковий режим продуктивності;
- паралельно не запускалися інші ресурсомісткі програми;
- математична постановка та параметри серії залишалися незмінними в межах порівнюваних конфігурацій;
- кількість реплік інших прикладних та інфраструктурних компонентів не змінювалася;
- перед запуском серії перевірялася готовність processor-worker;
- після зміни масштабу витримувалася 30-секундна стабілізаційна пауза.

Під час масштабування окремі pod processor-worker завершували роботу або створювалися повторно. Така поведінка відбувалася до початку вимірюваної серії. Серія запускалася лише після перевірки стану Deployment і готовності необхідної кількості pod, тому час їх створення та відновлення не включався до seriesDurationMs.

Водночас залишаються чинники, які могли впливати на результати. По-перше, всі конфігурації запускалися у фіксованому порядку:

$$N_w = 1 \rightarrow N_w = 2 \rightarrow N_w = 4. \quad (4.116)$$

Порядок не рандомізувався між повторами. Тому не можна повністю виключити вплив:

– прогрівання файлового кешу;
– повторного використання мережевих з’єднань;
– прогрівання контейнерів і бібліотек;
– зміни температурного режиму ноутбука;
– можливої зміни частоти процесора протягом тривалого навантаження.

Робота від мережі та незмінний режим продуктивності зменшують цей вплив, але не усувають його повністю.

По-друге, Redis/Celery-черга, PostgreSQL і MinIO не очищалися перед кожною серією. Попередні задачі завершувалися до початку нової серії, а нові задачі отримували унікальні taskId, тому прямого змішування результатів не відбувалося. Проте обсяг збережених записів та артефактів поступово збільшувався.

По-третє, клієнтський Angular-застосунок і браузер працювали на тому самому ноутбуці, що й Kubernetes-кластер. Тому рендеринг інтерфейсу, обробка Socket.IO-подій, виконання JavaScript і формування CSV також використовували частину ресурсів фізичного хоста.

З огляду на те, що інтерфейс і процедура експерименту були однаковими для всіх конфігурацій, цей вплив є систематичним. Проте для точнішого серверного тестування клієнтський генератор навантаження доцільно буде розмістити на окремому комп’ютері.

4.9.5 Валідність використаних метрик

Пропускна здатність визначалася за клієнтським фактичним часом від початку подання задач до отримання повного набору завершальних метрик. Тому вона характеризує фактичний час очікування користувача, а не лише час роботи серверної частини.

Значення seriesFinishedAt фіксувалося клієнтом після виявлення завершення всіх задач. Оновлення стану експерименту виконувалося періодично, тому до тривалості серії могла входити невелика затримка клієнтського опитування. Цей механізм був однаковим у всіх серіях, але його відносний вплив є більш помітним для коротких запусків легких задач.

Часові характеристики окремих стадій і E2E обчислювалися за серверними часовими мітками PostgreSQL, тому не залежали безпосередньо від затримки Socket.IO або швидкості роботи браузера.

Використаний p95 визначався як елемент впорядкованої вибірки без інтерполяції. Для 30 успішних задач це був 28-й елемент. Тому отримані значення можуть дещо відрізнятися від p95, обчисленого статистичними пакетами з іншими методами інтерполяції. Водночас єдиний алгоритм застосовувався до всіх серій, що забезпечує коректність порівняння між конфігураціями.

Поле l2Error у реалізації є середньоквадратичною вузловою похибкою, а не інтегральною $L_2(\Omega)$-нормою. Висновки щодо чисельної коректності відповідно формулюються для вузлових значень.

4.9.6 Статистична валідність

Для кожної конфігурації виконано три повторні серії. Це дозволило обчислити середнє та вибіркове стандартне відхилення і виявити загальну тенденцію масштабування.

Водночас кількість повторів $R = 3$ є обмеженою для проведення

повноцінних статистичних тестів і побудови вузьких довірчих інтервалів. Тому результати інтерпретуються переважно описово [82].

Особливо уважно слід трактувати невеликі відмінності між двома і чотирма виконавцями, коли діапазони стандартного відхилення перекриваються. Наприклад, незначне зменшення або збільшення p95 у таких випадках не розглядається як доказ стійкої переваги однієї конфігурації.

Більш надійна статистична перевірка потребувала б:

– більшої кількості повторів;
– рандомізації порядку конфігурацій;
– окремого аналізу викидів;
– довірчих інтервалів;
– перевірки статистичної значущості відмінностей.

4.9.7 Зовнішня валідність

Отримані результати безпосередньо характеризують реалізований прототип, вибрані математичні сценарії та локальний експериментальний стенд.

Дослідження було обмежене:

– задачею Пуассона;
– одиничним квадратом та одиничним кубом;
– лінійними P1-елементами;
– двома значеннями meshSize;
– пакетами з 30 незалежних однотипних задач;
– кількістю виконавців 1, 2 і 4;
– одним фізичним хостом;
– локальним сховищем local-path;
– фіксованим рівень паралелізму, що дорівнював трьом.

Тому результати не можна безпосередньо переносити на:

– багатовузлові фізично розподілені Kubernetes-кластери;
– промислові CAD-геометрії;

– задачі нелінійної механіки;
– нестаціонарні процеси;
– багатофізичні моделі;
– елементи вищого порядку;
– значно більші сітки;
– інші лінійні чисельні розв’язувача;
– GPU-прискорення;
– неоднорідні змішані потоки задач.

У багатовузловому кластері виконавці можуть отримати окремі фізичні ресурси, що здатне покращити масштабування. Водночас мережеве передавання артефактів між вузлами може збільшити затримки. Тому поведінка фізично розподіленої конфігурації потребує окремого дослідження [82, 85].

4.9.8 Напрями подальших експериментів

Для розширення отриманих результатів доцільно провести такі дослідження:

1) розгорнути платформу на декількох фізичних вузлах;
2) установити та варіювати CPU- і memory-обмеження контейнерів;
3) збирати часові ряди використання CPU, RAM, диска й мережі;
4) перевірити більші значення N_w;
5) збільшити кількість повторів і рандомізувати порядок конфігурацій;
6) винести генератор навантаження на окремий комп’ютер;
7) дослідити різні значення batch size і concurrency;
8) додати задачі з більшими сітками;
9) перевірити неоднорідний потік 2D- і 3D-задач;
10) порівняти статичне масштабування з автоматичним;
11) дослідити інші фізичні моделі.

Реалізація зазначених напрямів дасть змогу послідовно розширити зовнішню валідність отриманих результатів. Перехід до фізично багатовузлового

середовища дозволить відокремити ефект реального розподілу ресурсів від конкуренції процесів на одному хості, а системний моніторинг забезпечить безпосередню перевірку причин спостережуваного насичення. Розширення набору математичних постановок, розмірів сіток і профілів навантаження дасть змогу визначити, наскільки встановлені закономірності масштабування зберігаються для інших класів FEA-задач.

4.9.9 Загальне обговорення

Результати підтверджують доцільність незалежного масштабування обчислювального компонента, але також показують, що сама можливість створення додаткових реплік не гарантує лінійного прискорення.

Для легких задач найбільш ефективною в дослідженому середовищі була конфігурація з двома виконавцями. Для складнішої тривимірної задачі чотири виконавці забезпечили найкращі абсолютні показники, однак додатковий приріст порівняно з двома був невеликим.

Практичний висновок полягає в тому, що кількість виконавців потрібно добирати відповідно до обчислювальної складності задач і доступних ресурсів. Надмірна кількість виконавців може збільшити конкуренцію та не забезпечити очікуваного прискорення.

Водночас у всіх конфігураціях зберігалася повна відтворюваність чисельного результату. Отже, горизонтальне масштабування реалізованої платформи впливає на часові характеристики, але не порушує математичну коректність розв’язання.

4.10 Рекомендації щодо вибору кількості обчислювальних виконавців

Результати експериментального дослідження дають змогу сформулювати рекомендації щодо вибору кількості обчислювальних виконавців для подієво-

орієнтованої мікросервісної FEA-платформи. Основним чинником такого вибору є співвідношення між тривалістю чисельного розв’язання та загальною наскрізною затримкою виконання задачі. Якщо час роботи розв’язувача становить незначну частину повного часу оброблення, збільшення кількості виконавців не забезпечує пропорційного приросту продуктивності, оскільки результат обмежується тривалістю незмінних стадій конвеєра, передаванням подій, операціями зберігання даних і координацією компонентів.

Вибір кількості виконавців доцільно починати з початкової серії запусків з одним processor-worker. Для цієї серії визначають середню та p95 наскрізну затримку, пропускну здатність, тривалість окремих стадій і внутрішній час чисельного розв’язувача. Після цього аналогічну серію виконують із двома виконавцями. Збільшення їх кількості вважається доцільним, якщо приріст пропускної здатності та зменшення p95 наскрізної затримки є стійкими в повторних серіях і перевищують природний розкид отриманих значень [20, 21, 26].

Подальше збільшення кількості виконавців слід здійснювати лише за умови збереження помітного додаткового приросту. Якщо перехід від двох до чотирьох виконавців не приводить до стійкого збільшення пропускної здатності або зменшення наскрізної затримки, додаткове виділення обчислювальних ресурсів є недоцільним. При цьому після кожної зміни кількості виконавців необхідно перевіряти незмінність параметрів сітки, характеристик чисельного поля та показників похибки.

Для досліджених двовимірних задач доцільним є використання двох обчислювальних виконавців. Перехід від одного до двох виконавців забезпечив збільшення пропускної здатності приблизно на 8-12 % і зменшення p95 наскрізної затримки приблизно на 17-19 %. Збільшення їх кількості до чотирьох не дало стійкого додаткового приросту. Це пояснюється малою тривалістю чисельного розв’язання порівняно із загальним часом проходження задачі через розподілений конвеєр.

Для тривимірної задачі з грубішою сіткою доцільною також є конфігурація з двома виконавцями. У такому разі пропускна здатність зросла на 15,97 %, тоді

як перехід до чотирьох виконавців не забезпечив подальшого стійкого покращення. Отже, для задач середньої обчислювальної складності подвоєння кількості виконавців може бути виправданим, однак подальше масштабування потребує окремої перевірки.

Для складної тривимірної задачі з дрібнішою сіткою позитивний результат отримано як для двох, так і для чотирьох виконавців. При використанні двох виконавців пропускна здатність зросла на 29,03 %, а p95 наскрізної затримки зменшилася на 22,76 %. Використання чотирьох виконавців забезпечило приріст пропускної здатності на 35,21 % і зменшення p95 наскрізної затримки на 26,12 % порівняно з одним виконавцем. Водночас додатковий приріст при переході від двох до чотирьох виконавців був значно меншим, ніж при переході від одного до двох.

Таким чином, для складної тривимірної задачі два виконавці забезпечують кращий баланс між використаними ресурсами та отриманим приростом продуктивності. Чотири виконавці доцільно застосовувати тоді, коли пріоритетом є досягнення найбільшої пропускної здатності, а додаткові витрати ресурсів є прийнятними.

Узагальнені рекомендації для досліджених постановок наведено в таблиці 4.18.

Таблиця 4.18 – Рекомендована кількість обчислювальних виконавців

Тип задачі	**Рекомендована кількість виконавців**	**Обґрунтування**
Двовимірна задача	2	Подальше збільшення кількості виконавців не забезпечує стійкого приросту
Тривимірна задача з грубішою сіткою	2	Отримано приріст пропускної здатності на 15,97 %, тоді як чотири виконавці не покращують результат
Тривимірна задача з дрібнішою сіткою	2	Забезпечується основна частина можливого приросту при помірному використанні ресурсів
Тривимірна задача з дрібнішою сіткою за вимоги найбільшої пропускної здатності	4	Досягнуто приросту пропускної здатності на 35,21 % і зменшення p95 наскрізної затримки на 26,12 %

Практичний вибір слід здійснювати за принципом найменшої достатньої кількості виконавців. Оптимальною є найменша кількість, яка забезпечує встановлені вимоги до пропускної здатності та наскрізної затримки. Додавання наступних виконавців є обґрунтованим лише тоді, коли отриманий приріст перевищує розкид між повторними серіями та не супроводжується зміною контрольних чисельних характеристик.

Внутрішній час чисельного розв'язувача не слід розглядати окремо від інших часових характеристик. Визначальне значення має його частка в загальній наскрізній затримці. Чим більшою є ця частка, тим вищою є ймовірність отримання позитивного результату від горизонтального масштабування. Якщо ж основну частину часу становлять підготовка даних, передавання повідомлень, очікування в чергах, звернення до сховища або післяоброблення, збільшення кількості виконавців чисельної стадії матиме обмежений вплив.

Отже, вибір кількості обчислювальних виконавців має ґрунтуватися не на максимально можливій кількості реплік, а на стійкості приросту пропускної здатності, зменшенні p95 наскрізної затримки, частці часу чисельного розв'язання та незмінності контрольних чисельних характеристик. Такий підхід дає змогу обґрунтовано розподіляти ресурси платформи й уникати їх надлишкового виділення.

4.11 Висновки до розділу 4

У четвертому розділі проведено експериментальне дослідження продуктивності, масштабованості, чисельної коректності та відтворюваності розробленої подієво-орієнтованої FEA-платформи. Основною керованою змінною була кількість реплік обчислювального компонента processor-worker, яка набувала значень $N_w \in \{1,2,4\}$.

Для відокремлення впливу масштабування від інших чинників кількість задач у серії, рівеня паралелізму, конфігурація інших сервісів, версії програмного

забезпечення та параметри математичної постановки в межах порівнюваної групи залишалися незмінними.

Експериментальне дослідження виконано у два етапи. На пілотному етапі використовувалася двовимірна задача Пуассона на трикутній сітці, а на основному – тривимірна задача на тетраедральній сітці з неоднорідними граничними умовами Діріхле. Для кожного сценарію досліджено два значення параметра meshSize: 0,2 і 0,08.

Усього проведено 36 основних експериментальних серій по 30 задач у кожній, тобто виконано $N_{\text{total}} = 1080$ задач скінченно-елементного аналізу. Усі задачі завершено успішно. Для кожної з них зібрано наскрізну затримку, тривалості окремих стадій, внутрішні часові характеристики FEM-розв'язувача, параметри сітки, статистичні характеристики поля та показники чисельної похибки.

Результати пілотного двовимірного експерименту показали, що перехід від одного до двох виконавців збільшував пропускну здатність приблизно на 8-12 % і зменшував p95(E2E) приблизно на 17-19 %. Подальше збільшення кількості виконавців до чотирьох не забезпечило додаткового стійкого приросту пропускної здатності. Для обох двовимірних сіток найбільшу пропускну здатність отримано при $N_w = 2$.

Внутрішня тривалість FEM-обчислення двовимірної задачі становила лише приблизно 1,5-7,6 мс, тому основну частку повної затримки формували накладні витрати розподіленого конвеєра.

Основний тривимірний експеримент підтвердив залежність ефекту масштабування від складності задачі. Для грубішої сітки з meshSize=0,2 перехід від одного до двох виконавців збільшило пропускну здатність на 15,97 %, однак використання чотирьох виконавців не дало додаткового покращення. Найкращою для цієї постановки також була конфігурація з двома виконавцями.

Для дрібнішої тривимірної сітки з meshSize=0,08 збільшення кількості виконавців дало більш виражений результат. При двох виконавцях пропускна здатність зросла на 29,03 %, а при чотирьох – на 35,21 % порівняно з базовою конфігурацією. Значення p95(E2E) зменшилося відповідно на 22,76 % і 26,12 %.

Водночас перехід від двох до чотирьох виконавців забезпечив лише 4,79 % додаткової пропускної здатності. Це свідчить про наближення системи до насичення ресурсів навіть для найбільш складної з досліджених постановок.

Аналіз внутрішніх метрик Processor показав, що зі збільшенням кількості паралельних виконавців скорочувався час очікування задач на стадії solve, але зростав час виконання окремого FEM-розрахунку. Для тривимірної сітки з meshSize=0,08 середнє значення totalMs збільшилося приблизно з 399 мс при одному worker до 611 мс при двох і 783 мс при чотирьох.

Отриманий результат пояснюється конкуренцією процесів обчислювальних виконавців за спільні ресурси одного фізичного хоста. Незважаючи на наявність трьох логічних вузлів k3d, усі вони працювали на одному ноутбуці та використовували спільні CPU, оперативну пам’ять, NVMe-накопичувач і інфраструктурні компоненти.

Встановлено, що ефект горизонтального масштабування має непропорційний характер. Додаткові виконавці насамперед зменшують час очікування незалежних задач у черзі, але не прискорюють виконання окремої FEM-задачі. Після досягнення певного рівня паралельності приріст пропускної здатності обмежується незмінною кількістю реплік Preprocessor, Postprocessor, BFF та спільним використанням Kafka, Redis, PostgreSQL і MinIO.

Чисельну коректність перевірено за методом виготовлених розв’язків (MMS). Для кожної фіксованої пари scenarioId і meshSize параметри сітки, мінімальне, максимальне й середнє вузлові значення, maxAbsError і l2Error повністю збігалися в усіх задачах, повторах та конфігураціях

$$N_w = 1, \quad N_w = 2, \quad N_w = 4. \tag{4.117}$$

Отже, горизонтальне масштабування впливало на часові характеристики виконання, але не змінювало параметри сітки, статистичні характеристики поля та показники похибки, використані для контролю математичного результату.

Згущення сітки приводило до закономірного зменшення чисельної похибки. У двовимірній постановці при переході від meshSize=0,2 до

meshSize=0,08 максимальна абсолютна похибка зменшилася на 83,76 %, а вузлова RMS-похибка – на 85,58 %. У тривимірній постановці відповідне зменшення становило 81,92 % і 73,61 %.

За результатами експериментального дослідження сформульовані гіпотези оцінено таким чином:

1) гіпотезу H1 про зростання пропускної здатності при збільшенні кількості виконавців підтверджено частково, оскільки позитивний ефект спостерігався до досягнення насичення;
2) гіпотезу H2 про зменшення E2E-затримки підтверджено переважно для переходу від одного до двох виконавців і для складнішої тривимірної постановки;
3) гіпотезу H3 про більшу ефективність масштабування для складніших задач підтверджено;
4) гіпотезу H4 про зменшення віддачі та конкуренцію за ресурси підтверджено;
5) гіпотезу H5 про збереження чисельної коректності та відтворюваності підтверджено повністю.

Таким чином, експериментально підтверджено працездатність запропонованого механізму незалежного горизонтального масштабування обчислювальної стадії FEA-платформи. Його ефективність залежить від співвідношення між тривалістю корисного чисельного обчислення та накладними витратами розподіленого конвеєра.

Для легких задач доцільним є використання невеликої кількості виконавців, оскільки подальше масштабування не компенсує додаткову конкуренцію за ресурси. Для складніших задач збільшення кількості виконавців забезпечує відчутніше скорочення затримки та підвищення пропускної здатності, однак також має межу ефективності.

Отримані результати підтверджують, що запропонована архітектура дозволяє масштабувати найбільш ресурсоємний компонент незалежно від інших сервісів, зберігаючи чисельну коректність, цілісність артефактів і відтворюваність результатів [78, 86].

ВИСНОВКИ

У дисертаційній роботі розв’язано актуальне науково-прикладне завдання підвищення масштабованості та керованості програмної платформи скінченно-елементного аналізу шляхом розроблення подієво-орієнтованої мікросервісної архітектури, реалізації експериментального прототипу повного FEA-конвеєра та визначення умов, за яких горизонтальне масштабування обчислювального компонента забезпечує приріст продуктивності без зміни контрольних чисельних характеристик результату.

Основні результати дослідження полягають у такому:

1. Проведено порівняльний аналіз монолітного та мікросервісного підходів, а також комерційних, хмарних і відкритих FEA-рішень. Встановлено, що неоднорідність стадій скінченно-елементного аналізу потребує їх незалежного масштабування, а наявні рішення недостатньо висвітлюють подієвий поділ повного FEA-конвеєра, відокремлення різних видів даних і спільне оцінювання продуктивності та чисельної коректності.

2. Сформульовано функціональні й нефункціональні вимоги та розроблено архітектурну модель подієво-орієнтованої мікросервісної FEA-платформи. Повний обчислювальний цикл поділено між компонентами підготовки даних, чисельного розв’язання, післяоброблення, керування станом і візуалізації; визначено правила та формати їхньої взаємодії, відокремлено канали передавання керівних подій, службових даних про стан задачі та великих обчислювальних даних.

3. Розроблено й реалізовано експериментальний прототип повного FEA-конвеєра, який забезпечує створення задач, асинхронне виконання стадій, формування сітки, чисельне розв’язання, післяоброблення, зберігання та відображення результатів. Відокремлення приймання Kafka-подій від виконання чисельних задач дало змогу змінювати кількість реплік processor-worker без масштабування інших компонентів і зміни правил їхньої взаємодії.

4. Реалізовано двовимірну й тривимірну задачі Пуассона з використанням лінійних P1-елементів на трикутних і тетраедральних сітках. Чисельний контур охоплює складання розрідженої системи рівнянь, урахування граничних умов Діріхле та її розв’язання; правильність реалізації перевірено методом виготовлених розв’язків за максимальною абсолютною та середньоквадратичною вузловою похибками.

5. Розроблено методику оцінювання продуктивності, масштабованості та чисельної коректності платформи. Вона спільно враховує пропускну здатність, середню, медіанну та p95 наскрізну затримку, тривалість окремих стадій, внутрішній час чисельного розв’язувача, прискорення, ефективність масштабування, параметри сітки та показники похибки.

6. Проведено 36 експериментальних серій, що охопили 1080 успішно завершених задач для двовимірних і тривимірних постановок та конфігурацій з одним, двома й чотирма обчислювальними виконавцями. Для двовимірних задач перехід до двох виконавців збільшив пропускну здатність на 8-12 % і зменшив p95 наскрізної затримки на 17-19 %. Для тривимірної задачі з грубішою сіткою пропускна здатність зросла на 15,97 %, а для складнішої постановки – на 29,03 % при двох і на 35,21 % при чотирьох виконавцях; p95 затримки зменшилася відповідно на 22,76 % і 26,12 %. На основі отриманих результатів розроблено рекомендації щодо вибору найменшої достатньої кількості виконавців залежно від частки часу чисельного розв’язання та стійкості приросту продуктивності.

7. Підтверджено збереження контрольних чисельних характеристик при зміні кількості виконавців: для кожної постановки збіглися параметри сітки, характеристики вузлового поля та показники похибки. При згущенні сітки максимальна абсолютна й середньоквадратична похибки зменшилися відповідно на 83,76 % і 85,58 % для двовимірної задачі та на 81,92 % і 73,61 % для тривимірної. Встановлено непропорційний характер масштабування, зумовлений незмінними стадіями конвеєра та, за результатами інтерпретації вимірювань, спільним використанням ресурсів одного фізичного середовища.

Таким чином, розроблена подієво-орієнтована мікросервісна архітектура забезпечує технічну можливість незалежного масштабування обчислювальної

стадії скінченно-елементного аналізу, асинхронне проходження повного життєвого циклу задачі, розділене зберігання метаданих і великих артефактів та збереження контрольних чисельних характеристик. Експериментально встановлено, що практична ефективність масштабування визначається співвідношенням між тривалістю корисного чисельного обчислення, накладними витратами розподіленого конвеєра та доступними ресурсами фізичного середовища.

Перспективами подальших досліджень є реалізація автоматичного масштабування на основі довжини черги та ресурсних метрик, проведення експериментів у багатовузловому кластері, дослідження великих тривимірних моделей, упровадження повторних спроб і каналів відхілених повідомлень, атомарних механізмів ідемпотентності, розподіленого об’єктного сховища, систем трасування й моніторингу, а також розширення платформи новими класами крайових задач і чисельними розв’язувачами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Zienkiewicz O. C., Taylor R. L., Zhu J. Z. The Finite Element Method: Its Basis and Fundamentals : 7th ed. Oxford : Butterworth-Heinemann, 2013. 756 p.

2. Cook R. D., Malkus D. S., Plesha M. E., Witt R. J. Concepts and Applications of Finite Element Analysis : 4th ed. Hoboken : John Wiley & Sons, 2002. 736 p.

3. Hughes T. J. R. The Finite Element Method: Linear Static and Dynamic Finite Element Analysis. Mineola : Dover Publications, 2000. 704 p.

4. Bathe K. J. Finite Element Procedures. Upper Saddle River : Prentice Hall, 1996. 1043 p.

5. Larson M. G., Bengzon F. The Finite Element Method: Theory, Implementation, and Applications. Berlin ; Heidelberg : Springer, 2013. 402 p. DOI: 10.1007/978-3-642-33287-6.

6. Bass L., Clements P., Kazman R. Software Architecture in Practice : 4th ed. Boston : Addison-Wesley Professional, 2021. 464 p.

7. Newman S. Building Microservices: Designing Fine-Grained Systems : 2nd ed. Sebastopol : O'Reilly Media, 2021. 612 p.

8. Lewis J., Fowler M. Microservices: a definition of this new architectural term. 2014. URL : https://martinfowler.com/articles/microservices.html (дата звернення: 25.06.2026).

9. Jamshidi P., Pahl C., Mendonça N. C., Lewis J., Tilkov S. Microservices: The Journey So Far and Challenges Ahead. IEEE Software. 2018. Vol. 35, No. 3. P. 24–35. DOI: 10.1109/MS.2018.2141039.

10. Blinowski G., Ojdowska A., Przybyłek A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*. 2022. Vol. 10. P. 20357–20374. DOI: 10.1109/ACCESS.2022.3152803.

11. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 670 p.

12. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston : Addison-Wesley, 2003. 736 p.

13. Vogels W. Eventually consistent. Communications of the ACM. 2009. Vol. 52, No. 1. P. 40–44. DOI: 10.1145/1435417.1435432.

14. Kreps J., Narkhede N., Rao J. Kafka: a distributed messaging system for log processing. Proceedings of the 6th International Workshop on Networking Meets Databases. Athens, 2011. P. 1–7.

15. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega, and Kubernetes. Communications of the ACM. 2016. Vol. 59, No. 5. P. 50–57. DOI: 10.1145/2890784.

16. Merkel D. Docker: Lightweight Linux containers for consistent development and deployment. Linux Journal. 2014. No. 239. Article 2.

17. Гоменюк С. І., Гребенюк С. М., Панасенко Є. В. Вебреалізація постпроцесора з відкритим початковим кодом. *Computer Science and Applied Mathematics.* 2025. № 1. С. 104–110. DOI: 10.26661/2786-6254-2025-1-12.

18. Shtanko H. I., Grebenyuk S. M. Features of the construction of the stiffness matrix of the spatial hexagonal finite element for composite material with discrete inclusions based on the moment scheme. *Computer Science and Applied Mathematics*. 2023. № 2. С. 75–85. DOI: 10.26661/2786-6254-2023-2-10.

19. Гоменюк С. І., Гребенюк С. М., Манько Н. І.-В., Спиця О. Г. Чисельне моделювання контактної взаємодії штампів та гумовокордної смуги. Прикладні питання математичного моделювання. 2021. Т. 4, № 2.2. С. 64–73. DOI: 10.32782/KNTU2618-0340/2021.4.2.2.6.

20. Гоменюк С. І., Козуб В. Ю. Алгоритм паралельних обчислень у методі скінченних елементів. Computer Science and Applied Mathematics. 2022. № 2. С. 66–71. DOI: 10.26661/2786-6254-2022-2-08.

21. Гоменюк С. І., Козуб В. Ю. Паралельний алгоритм формування матриці жорсткості скінченного елемента. Вчені записки ТНУ імені В. І. Вернадського. Серія: Технічні науки. 2023. Т. 34(73), № 1. С. 82–87. DOI: 10.32782/2663-5941/2023.1/12.

22. Ярош А. О., Кудін О. В. Нейромережеві методи розв’язання задач пружності. Вісник Херсонського національного технічного університету. 2024. № 1(88). С. 295–305. DOI: 10.35546/kntu2078-4481.2024.1.41.

23. Hniezdovskyi O., Kudin O., Belokon Y., Kruglyak D., Ilin S. Designing an object-oriented architecture for the finite element simulation of structural elements. *Eastern-European Journal of Enterprise Technologies*. 2022. Vol. 6(2 (120). P. 78–84. DOI: 10.15587/1729-4061.2022.268018.

24. Ярош А. О., Кудін О. В. Нейроеволюційний метод колокації розв’язання диференціальних рівнянь. Таврійський науковий вісник. Серія: Технічні науки. 2024. № 6. С. 72–81. DOI: 10.32782/tnv-tech.2023.6.9.

25. Козуб Г. О., Козуб Ю. Г. Декларативний підхід при створенні мультиплатформних додатків. Вісник Східноукраїнського національного університету імені Володимира Даля. 2022. № 5(275). С. 10–15. DOI: 10.33216/1998-7927-2022-275-5-10-15.

26. Padmanaban H., Arkabaev N., Rusho M. A., Kozub V., Kozub Y. Using Java to create and analyze models of parallel computing systems. Parallel Computing. 2025. Vol. 125. Article 103146. DOI: 10.1016/j.parco.2025.103146.

27. Solodei I., Kozub Y., Stryhun R., Shovkivska V. Algorithms analysis for solving geometrically nonlinear mechanics problems in the scheme of the semi-analytical finite element method. Strength of Materials and Theory of Structures. 2022. No. 109. P. 109–119. DOI: 10.32347/2410-2547.2022.109.109-119.

28. Астіоненко І., Гучек П., Дудченко О., Литвиненко О. Геометричний метод побудови базису дискретного елемента H32. Сучасні проблеми моделювання. 2025. Вип. 28. С. 3–11. DOI: 10.33842/2313-125X-2025-30-3-11.

29. Guchek P., Litvinenko O., Astionenko I., Dudchenko O., Khomchenko A. Investigating the properties of mixed finite elements. Procedia Computer Science. 2024. Vol. 237. P. 354–362. DOI: 10.1016/j.procs.2024.05.115.

30. Guchek P., Litvinenko O., Astionenko I., Dudchenko O. Multiparametric basis functions on the finite element Q12. Procedia Computer Science. 2025. Vol. 263. P. 608–618. DOI: 10.1016/j.procs.2025.07.073.

31. Кривий Я. В., Лісняк А. О. Мікросервісна архітектура систем скінченно-елементного аналізу. Computer Science and Applied Mathematics. 2024. № 1. С. 75–83. DOI: 10.26661/2786-6254-2024-1-09.

32. Velepucha V., Flores P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. IEEE Access. 2023. Vol. 11. P. 88339–88358. DOI: 10.1109/ACCESS.2023.3305687.

33. Di Francesco P., Lago P., Malavolta I. Architecting with microservices: A systematic mapping study. Journal of Systems and Software. 2019. Vol. 150. P. 77–97. DOI: 10.1016/j.jss.2019.01.001.

34. Taibi D., Lenarduzzi V., Pahl C. Architectural Patterns for Microservices: A Systematic Mapping Study. Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018). 2018. P. 221–232. DOI: 10.5220/0006798302210232.

35. Fehling C., Leymann F., Retter R., Schupeck W., Arbitter P. Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Vienna : Springer, 2014. DOI: 10.1007/978-3-7091-1568-8.

36. Brenner S. C., Scott L. R. The Mathematical Theory of Finite Element Methods : 3rd ed. New York : Springer, 2008. DOI: 10.1007/978-0-387-75934-0.

37. Ern A., Guermond J.-L. Theory and Practice of Finite Elements. New York : Springer, 2004. DOI: 10.1007/978-1-4757-4355-5.

38. Frey P. J., George P.-L. Mesh Generation: Application to Finite Elements : 2nd ed. London : ISTE ; Hoboken : John Wiley & Sons, 2008. 848 p. DOI: 10.1002/9780470611166.

39. Davis T. A. Direct Methods for Sparse Linear Systems. Philadelphia : Society for Industrial and Applied Mathematics, 2006. DOI: 10.1137/1.9780898718881.

40. Alnæs M. S. et al. The FEniCS Project Version 1.5. Archive of Numerical Software. 2015. Vol. 3, No. 100. DOI: 10.11588/ans.2015.100.20553.

41. Cimrman R., Kolman P., Lukeš V. SfePy – write your own finite element code in Python. Acta Polytechnica. 2014. Vol. 54, No. 1. P. 50–56. DOI: 10.14311/AP.2014.54.0050.

42. Hecht F. New development in FreeFem++. Journal of Numerical Mathematics. 2012. Vol. 20, No. 3–4. P. 251–266. DOI: 10.1515/jnum-2012-0013.

43. CSC – IT Center for Science. Elmer finite element software. URL : https://www.csc.fi/web/elmer (дата звернення: 25.06.2026).

44. ANSYS. Ansys Cloud. URL : https://www.ansys.com/products/cloud (дата звернення: 25.06.2026).

45. SimScale. Cloud-native engineering simulation platform. URL : https://www.simscale.com/ (дата звернення: 25.06.2026).

46. Kryvyi Y., Lisnyak A. Event-Driven Microservice Architecture for Scalable Finite Element Analysis Platforms. *Computer Design Systems. Theory and Practice*. 2026. Vol. 8, No. 1. P. 56–69. DOI: 10.23939/cds2026.01.056.

47. ISO/IEC/IEEE 29148:2018. Systems and software engineering – Life cycle processes – Requirements engineering. Geneva : International Organization for Standardization, 2018. 92 p.

48. ISO/IEC 25010:2023. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model : 2nd ed. Geneva : International Organization for Standardization, 2023.

49. ISO/IEC/IEEE 42010:2022. Software, systems and enterprise – Architecture description : 2nd ed. Geneva : International Organization for Standardization, 2022.

50. Object Management Group. OMG Unified Modeling Language, Version 2.5.1. 2017. URL : https://www.omg.org/spec/UML/2.5.1 (дата звернення: 25.06.2026).

51. National Institute of Standards and Technology. Integration Definition for Function Modeling (IDEF0). FIPS PUB 183. Gaithersburg, 1993. URL : https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub183.pdf (дата звернення: 25.06.2026).

52. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral dissertation. Irvine : University of California, 2000. URL : https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 25.06.2026).

53. Zuo X., Su Y., Wang Q., Xie Y. An API gateway design strategy optimized for persistence and coupling. Advances in Engineering Software. 2020. Vol. 148. Article 102878. DOI: 10.1016/j.advengsoft.2020.102878.

54. Fette I., Melnikov A. The WebSocket Protocol. RFC 6455. Internet Engineering Task Force, 2011. DOI: 10.17487/RFC6455.

55. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force, 2012. DOI: 10.17487/RFC6749.

56. Sakimura N., Bradley J., Jones M., de Medeiros B., Mortimore C. OpenID Connect Core 1.0 incorporating errata set 2. OpenID Foundation, 2014. URL : https://openid.net/specs/openid-connect-core-1_0.html (дата звернення: 25.06.2026).

57. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force, 2015. DOI: 10.17487/RFC7519.

58. Jones M. JSON Web Key (JWK). RFC 7517. Internet Engineering Task Force, 2015. DOI: 10.17487/RFC7517.

59. Sakimura N., Bradley J., Agarwal N. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636. Internet Engineering Task Force, 2015. DOI: 10.17487/RFC7636.

60. Fielding R., Nottingham M., Reschke J. HTTP Semantics. RFC 9110. Internet Engineering Task Force, 2022. DOI: 10.17487/RFC9110.

61. Traefik Labs. Traefik Kubernetes Ingress documentation. URL : https://doc.traefik.io/traefik/reference/install-configuration/providers/kubernetes/kubernetes-ingress/ (дата звернення: 25.06.2026).

62. Keycloak. Keycloak documentation. URL : https://www.keycloak.org/documentation (дата звернення: 25.06.2026).

63. Apache Software Foundation. Apache Kafka documentation. URL : https://kafka.apache.org/documentation/ (дата звернення: 25.06.2026).

64. PostgreSQL Global Development Group. PostgreSQL documentation. URL : https://www.postgresql.org/docs/current/ (дата звернення: 25.06.2026).

65. MinIO. S3 API compatibility. URL : https://docs.min.io/enterprise/aistor-object-store/developers/s3-api-compatibility/ (дата звернення: 25.06.2026).

66. Redis. Redis documentation. URL : https://redis.io/docs/latest/ (дата звернення: 25.06.2026).

67. Celery Project. Celery documentation. URL : https://docs.celeryq.dev/en/stable/ (дата звернення: 25.06.2026).

68. Docker Inc. Docker documentation. URL : https://docs.docker.com/ (дата звернення: 25.06.2026).

69. The Kubernetes Authors. Kubernetes documentation. URL : https://kubernetes.io/docs/home/ (дата звернення: 25.06.2026).

70. Angular. Angular documentation: Overview. URL : https://angular.dev/overview (дата звернення: 25.06.2026).

71. OpenJS Foundation. Node.js API documentation. URL : https://nodejs.org/docs/latest/api/ (дата звернення: 25.06.2026).

72. Socket.IO. Socket.IO documentation. Version 4. URL : https://socket.io/docs/v4/ (дата звернення: 25.06.2026).

73. Three.js. Three.js documentation. URL : https://threejs.org/docs/ (дата звернення: 25.06.2026).

74. FastAPI. FastAPI documentation. URL : https://fastapi.tiangolo.com/ (дата звернення: 25.06.2026).

75. Geuzaine C., Remacle J.-F. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering. 2009. Vol. 79, No. 11. P. 1309–1331. DOI: 10.1002/nme.2579.

76. Harris C. R. et al. Array programming with NumPy. Nature. 2020. Vol. 585. P. 357–362. DOI: 10.1038/s41586-020-2649-2.

77. Virtanen P. et al. SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods. 2020. Vol. 17. P. 261–272. DOI: 10.1038/s41592-019-0686-2.

78. Кривий Я. В., Лісняк А. О. Подієво-орієнтована мікросервісна архітектура масштабованої платформи скінченно-елементного аналізу: розробка та експериментальна оцінка. Computer Science and Applied Mathematics. 2026. № 1. С. 120–141. DOI: 10.26661/2786-6254-2026-1-13.

79. Salari K., Knupp P. Code Verification by the Method of Manufactured Solutions. Albuquerque : Sandia National Laboratories, 2000. DOI: 10.2172/759450.

80. Roache P. J. Verification and Validation in Computational Science and Engineering. Albuquerque : Hermosa Publishers, 1998. 446 p.

81. Oberkampf W. L., Roy C. J. Verification and Validation in Scientific Computing. Cambridge : Cambridge University Press, 2010. 767 p.

82. Jain R. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York : John Wiley & Sons, 1991. 720 p.

83. Dean J., Barroso L. A. The Tail at Scale. Communications of the ACM. 2013. Vol. 56, No. 2. P. 74–80. DOI: 10.1145/2408776.2408794.

84. Amdahl G. M. Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the AFIPS Spring Joint Computer Conference. Atlantic City, 1967. Vol. 30. P. 483–485. DOI: 10.1145/1465482.1465560.

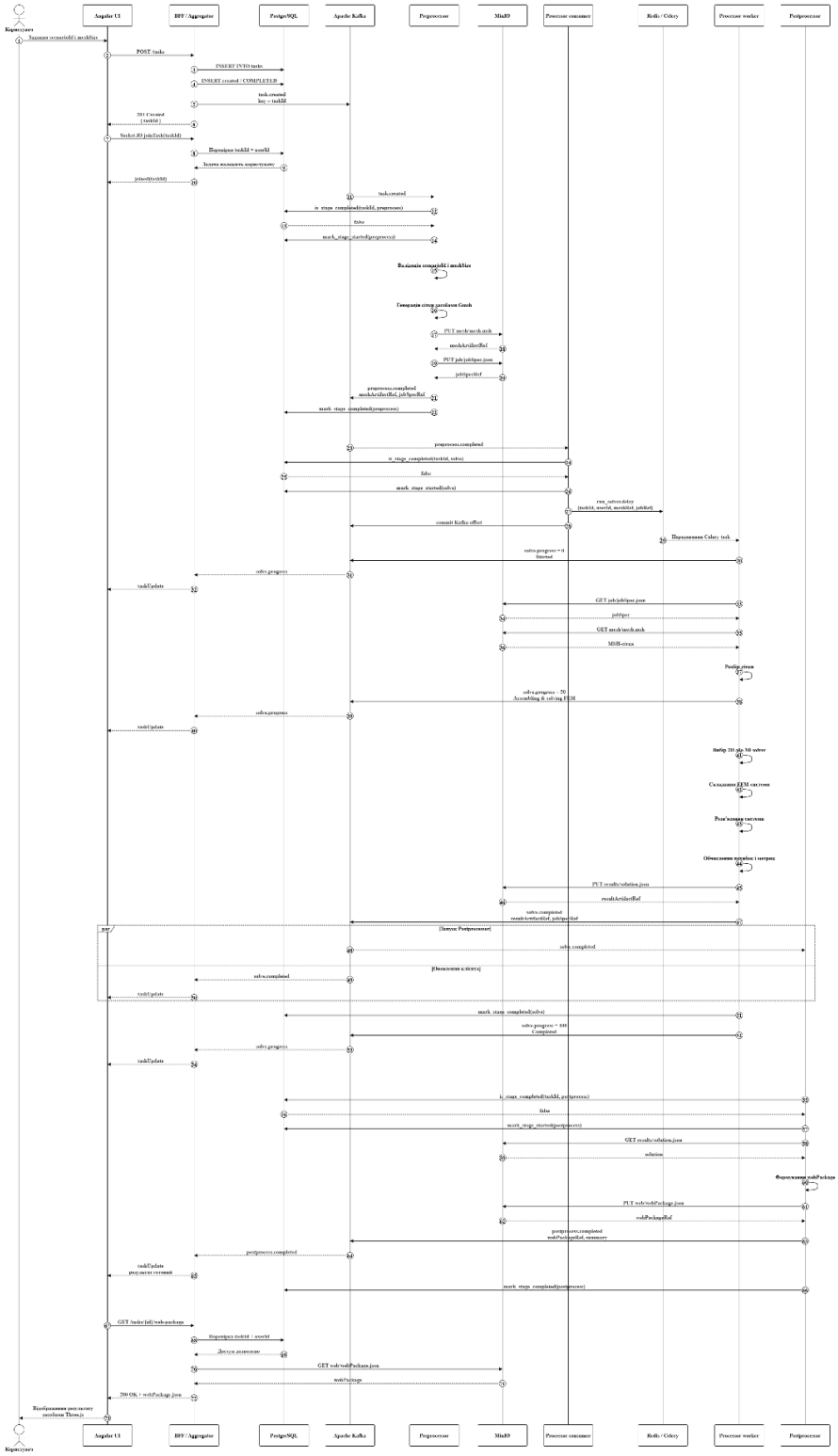
85. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol : O'Reilly Media, 2016. 552 p.

86. Кривий Я. В., Лісняк А. О. Подієво-орієнтоване масштабування 3D FEA-конвеєра у мікросервісній архітектурі. Актуальні проблеми математики та інформатики : збірка тез доповідей Сімнадцятої Всеукраїнської, двадцять четвертої регіональної наукової конференції молодих дослідників (Запоріжжя, 23–24 квітня 2026 р.). Запоріжжя : ЗНУ, 2026. С. 50–54. URL : https://www.znu.edu.ua/faculty/math/confirences/aktual__n___problemi_matematiki_ta___nformatiki_-_2026__tezi_konferents__yi.pdf (дата звернення: 25.06.2026).

ДОДАТКИ

ДОДАТОК А

Діаграма послідовності подієвої взаємодії компонентів FEA-конвеєра



ДОДАТОК Б

Фізична структура таблиць та індексів

```
CREATE TABLE public.tasks (
    id text NOT NULL,
    user_id text NOT NULL,
    scenario_id text NOT NULL,
    mesh_size double precision NOT NULL,
    status text DEFAULT 'CREATED'::text NOT NULL,
    created_at timestamp with time zone DEFAULT now() NOT NULL,
    updated_at timestamp with time zone DEFAULT now() NOT NULL
);

CREATE TABLE public.task_stages (
    id bigint NOT NULL,
    task_id text NOT NULL,
    stage text NOT NULL,
    status text NOT NULL,
    started_at timestamp with time zone,
    completed_at timestamp with time zone,
    attempt integer DEFAULT 1 NOT NULL
);

CREATE SEQUENCE public.task_stages_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;

ALTER SEQUENCE public.task_stages_id_seq
    OWNED BY public.task_stages.id;

ALTER TABLE ONLY public.task_stages
    ALTER COLUMN id
    SET DEFAULT nextval(
        'public.task_stages_id_seq'::regclass
    );
```

```
ALTER TABLE ONLY public.tasks
    ADD CONSTRAINT tasks_pkey
    PRIMARY KEY (id);

ALTER TABLE ONLY public.task_stages
    ADD CONSTRAINT task_stages_pkey
    PRIMARY KEY (id);

ALTER TABLE ONLY public.task_stages
    ADD CONSTRAINT task_stages_task_id_fkey
    FOREIGN KEY (task_id)
    REFERENCES public.tasks(id)
    ON DELETE CASCADE;

CREATE INDEX idx_tasks_created_at
    ON public.tasks USING btree (created_at);

CREATE INDEX idx_tasks_user_id
    ON public.tasks USING btree (user_id);
```

ДОДАТОК В

Механізм авторизованої підписки й передавання подій

```
async function startKafkaConsumer(io) {
  const brokers = (
    process.env.KAFKA_BROKERS || ""
  )
    .split(",")
    .map(s => s.trim())
    .filter(Boolean);

  const kafka = new Kafka({
    clientId: "bff-consumer",
    brokers
  });

  const consumer = kafka.consumer({
    groupId: "bff-updates"
  });

  const topics = [
    process.env.TOPIC_SOLVE_PROGRESS
      || "solve.progress",
    process.env.TOPIC_SOLVE_COMPLETED
      || "solve.completed",
    process.env.TOPIC_SOLVE_FAILED
      || "solve.failed",
    process.env.TOPIC_POSTPROCESS_COMPLETED
      || "postprocess.completed",
    process.env.TOPIC_POSTPROCESS_FAILED
      || "postprocess.failed",
    process.env.TOPIC_PREPROCESS_FAILED
      || "preprocess.failed"
  ];

  await consumer.connect();

  for (const topic of topics) {
    await consumer.subscribe({
```

```
            topic,
            fromBeginning: false
        });
    }

    await consumer.run({
        eachMessage: async ({ topic, message }) => {
            const value = message.value
                ?.toString("utf-8");

            if (!value) return;

            const env = JSON.parse(value);
            const taskId = env?.subject?.taskId;

            if (!taskId) return;

            io.to(taskId).emit("taskUpdate", {
                topic,
                type: env.type,
                occurredAt: env.occurredAt,
                data: env.data,
                subject: env.subject
            });
        }
    });
}

io.use(async (socket, next) => {
    try {
        const token =
            socket.handshake.auth?.token;

        if (!token) {
            return next(
                new Error("Missing token")
            );
        }

        const payload =
            await verifyBearerToken(
                `Bearer ${token}`
```

```
            );

        if (!payload?.sub) {
            return next(
                new Error("Unauthorized")
            );
        }

        socket.userId = payload.sub;
        next();
    } catch {
        next(new Error("Unauthorized"));
    }
});

io.on("connection", socket => {
    socket.on("joinTask", async taskId => {
        if (
            typeof taskId !== "string"
            || taskId.length < 5
        ) {
            return;
        }
        const { rows } = await db.query(
            `SELECT id
              FROM tasks
             WHERE id = $1
               AND user_id = $2
             LIMIT 1`,
            [taskId, socket.userId]
        );
        if (!rows.length) {
            socket.emit("joinDenied", {
                taskId
            });
            return;
        }
        socket.join(taskId);
        socket.emit("joined", {
            room: taskId
        });
    });
});
```

ДОДАТОК Г

Обробка події task.created у Preprocessor

```
def handle_task_created(
        self,
        envelope: dict
):
    task_id = envelope.get(
        "subject", {}
    ).get("taskId")

    user_id = envelope.get(
        "subject", {}
    ).get("userId")

    data = envelope.get("data", {}) or {}

    scenario_id = data.get("scenarioId")
    mesh_size_raw = data.get("meshSize")

    if not task_id or not scenario_id:
        raise RuntimeError(
            "Invalid event: "
            "missing taskId/scenarioId"
        )

    try:
        mesh_size = float(mesh_size_raw)
    except Exception:
        raise RuntimeError(
            "Invalid event: "
            "meshSize must be numeric"
        )

    if mesh_size <= 0:
        raise RuntimeError(
            "Invalid event: "
            "meshSize must be positive"
        )
```

```
if self.stages.is_completed(
    task_id,
    "preprocess"
):
    return

self.stages.start(
    task_id,
    "preprocess"
)

try:
    scenario_str = str(scenario_id)

    msh_bytes = self.mesh_gen.generate(
        scenario_str,
        mesh_size
    )

    mesh_key = (
        f"tasks/{task_id}/mesh/mesh.msh"
    )
    job_key = (
        f"tasks/{task_id}/job/"
        f"jobSpec.json"
    )

    mesh_ref = self.store.put_bytes(
        mesh_key,
        msh_bytes,
        "application/octet-stream"
    )

    dimension = (
        3
        if "cube" in
           scenario_str.strip().lower()
        else 2
    )
    job_spec = {
        "taskId": task_id,
```

```
                    "scenarioId": scenario_str,
                    "meshSize": mesh_size,
                    "dimension": dimension,
                    "meshRef": {
                        "bucket": self.store.bucket,
                        "key": mesh_key
                    },
                    "problemType": "poisson",
                    "boundary": {
                        "dirichletGroup":
                            "DIRICHLET"
                    }
                }
                job_ref = self.store.put_json(
                    job_key,
                    job_spec
                )
                self.publisher.publish_completed(
                    task_id,
                    user_id,
                    {
                        "scenarioId": scenario_str,
                        "meshSize": mesh_size,
                        "dimension": dimension,
                        "meshArtifactRef": mesh_ref,
                        "jobSpecRef": job_ref
                    }
                )
                self.stages.complete(
                    task_id,
                    "preprocess"
                )
            except Exception as e:
                self.stages.fail(
                    task_id,
                    "preprocess"
                )
                self.publisher.publish_failed(
                    task_id,
                    user_id,
                    str(e)
                )
                raise
```

ДОДАТОК Д

Вибір FEM-алгоритму та формування solution.json

```
if parsed.dimension == 2:
    values, fem_summary = (
        self.solver.solve_2d(
            parsed.points,
            parsed.elements,
            parsed.dirichlet_nodes,
            scenario_id=scenario_id
        )
    )

    triangles_for_output = (
        parsed.surface_triangles
    )
else:
    values, fem_summary = (
        self.solver.solve_3d(
            parsed.points,
            parsed.elements,
            parsed.dirichlet_nodes,
            scenario_id=scenario_id
        )
    )

    triangles_for_output = (
        parsed.surface_triangles
    )

    fem_summary["nTriangles"] = int(
        len(triangles_for_output)
    )

vmin = (
    float(min(values))
    if values else 0.0
)
```

```
        vmax = (
            float(max(values))
            if values else 0.0
        )

        vavg = (
            float(sum(values) / len(values))
            if values else 0.0
        )

        solution = {
            "taskId": task_id,
            "points": parsed.points,
            "triangles": triangles_for_output,
            "values": values,
            "summary": {
                **fem_summary,
                "min": vmin,
                "max": vmax,
                "avg": vavg
            }
        }

        solution_key = (
            f"tasks/{task_id}/results/"
            f"solution.json"
        )

        solution_ref = self.store.put_json(
            solution_key,
            solution
        )
```

ДОДАТОК Е

Формування і збереження webPackage.json

```
class WebPackageBuilder:
    def build(
            self,
            task_id: str,
            solution: Dict[str, Any]
    ) -> Dict[str, Any]:
        return {
            "formatVersion": 1,
            "taskId": task_id,
            "geometry": {
                "points": solution.get(
                    "points", []
                ),
                "triangles": solution.get(
                    "triangles", []
                )
            },
            "values": solution.get(
                "values", []
            ),
            "summary": solution.get(
                "summary", {}
            )
        }


def handle_solve_completed(
        self,
        envelope: dict
):
    task_id = envelope.get(
        "subject", {}
    ).get("taskId")

    user_id = envelope.get(
        "subject", {}
    ).get("userId")
```

```
        data = envelope.get("data", {}) or {}

        result_ref = data.get(
            "resultArtifactRef"
        )

        if not task_id or not result_ref:
            raise RuntimeError(
                "Invalid solve.completed payload"
            )

        if self.stages.is_completed(
            task_id,
            "postprocess"
        ):
            return

        self.stages.start(
            task_id,
            "postprocess"
        )

        try:
            solution = self.store.read_json(
                result_ref["bucket"],
                result_ref["key"]
            )

            web_package = self.builder.build(
                task_id,
                solution
            )

            web_key = (
                f"tasks/{task_id}/web/"
                f"webPackage.json"
            )

            web_ref = self.store.put_json(
                web_key,
                web_package
```

```
            )

            self.publisher.completed(
                task_id,
                user_id,
                {
                    "webPackageRef": web_ref,
                    "summary": web_package.get(
                        "summary", {}
                    )
                }
            )

            self.stages.complete(
                task_id,
                "postprocess"
            )

        except Exception as e:
            self.stages.fail(
                task_id,
                "postprocess"
            )

            self.publisher.failed(
                task_id,
                user_id,
                str(e)
            )

            raise
```