МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет будівництва і архітектури

Архітектура розподілених програмних систем

Методичні вказівки до виконання лабораторних робіт 1-9
для підготовки здобувачів освітньо-кваліфікаційного рівня «магістр»
спеціальності F2 «Інженерія програмного забезпечення»

Київ 2025

УДК 004.6

Укладач: Р.В. МАЗУРЕНКО д-р філософії, ас. кафедри інформаційних технологій

Рецензент О.О. Терентьєв, д-р техн. наук, професор, Київський національний університет будівництва і архітектури

Відповідальна за випуск Т.А. Гончаренко, д-р техн. наук, доцент

Затверджено на засіданні кафедри інформаційних технологій, протокол № 8 від 21 жовтня 2025 року.

В авторській редакції.

П **Архітектура розподілених програмних систем**: методичні вказівки до виконання лабораторних/ уклад.: Мазуренко Р.В. – Київ: КНУБА, 2025. – 56 с.

Містять зміст, порядок оформлення і вказівки до виконання лабораторних робіт.

Призначено для здобувачів спеціальностей 122 «Комп'ютерні науки» та F2 «Інженерія програмного забезпечення»

ЗМІСТ

Загальні положення	4
ЛАБОРАТОРНА РОБОТА № 1	6
ЛАБОРАТОРНА РОБОТА № 2	9
ЛАБОРАТОРНА РОБОТА № 3	12
ЛАБОРАТОРНА РОБОТА № 4	16
ЛАБОРАТОРНА РОБОТА № 5	20
ЛАБОРАТОРНА РОБОТА № 6	24
ЛАБОРАТОРНА РОБОТА № 7	28
ЛАБОРАТОРНА РОБОТА № 8	32
ЛАБОРАТОРНА РОБОТА № 9	36
Список літератури	40

Загальні положення

Мета вивчення освітньої компоненти «Архітектура розподілених програмних систем» полягає у формуванні в здобувачів другого (магістерського) рівня освіти знань і практичних навичок у галузі проєктування, побудови та підтримки масштабованих, надійних і відмовостійких розподілених програмних систем, що функціонують у хмарних або гібридних середовищах.

Освітня компонента охоплює сучасні підходи до створення архітектур, які забезпечують ефективну взаємодію незалежних програмних модулів — мікросервісів, контейнеризованих застосунків, потокових систем обробки даних, IoT-пристроїв та аналітичних платформ.

Основна ідея полягає у розподілі функцій між окремими компонентами, що взаємодіють через стандартизовані інтерфейси, для досягнення високої гнучкості, масштабованості й стійкості системи до збоїв.

Дисципліна спрямована на розвиток у здобувачів здатності:

- аналізувати вимоги до складних розподілених систем;
- обґрунтовано вибирати архітектурні підходи (клієнт-сервер, мікросервіси, подієво-орієнтована архітектура, serverless);
- розробляти архітектурні моделі, які враховують вимоги до надійності, безпеки, продуктивності й масштабування;
- оцінювати компроміси між централізацією й розподілом функцій.

Засвоєння освітньої компоненти передбачає опанування технологій, що забезпечують практичну реалізацію архітектурних рішень:

контейнеризація (Docker, Podman), оркестрація (Kubernetes, Docker Compose), системи обміну повідомленнями (Kafka, RabbitMQ), моніторинг (Prometheus, Grafana), логування (ELK Stack, OpenTelemetry) та інструменти CI/CD (GitHub Actions, Jenkins).

У процесі навчання студенти виконують лабораторні роботи, які охоплюють повний цикл створення розподіленої системи — від аналізу вимог та проєктування архітектури до її контейнеризації, оркестрації, моніторингу й масштабування.

Виконання практичних завдань орієнтоване на реальні галузеві приклади, зокрема — системи управління та моніторингу будівельних процесів, де розподілені сервіси взаємодіють із IoT-сенсорами, аналітичними платформами й веб-інтерфейсами для інженерів.

Компонента також сприяє розвитку комунікаційних і командних навичок. Здобувачі вчаться документувати архітектурні рішення, спілкуватися з іншими фахівцями (аналітиками, DevOps-інженерами, тестувальниками) та узгоджувати технічні рішення у межах спільного проєкту.

Завдання дисципліни полягає не лише у засвоєнні теоретичних основ побудови розподілених систем, але й у набутті здатності застосовувати ці знання на практиці — у процесі розробки, тестування, оптимізації та підтримки складних архітектур.

Основні програмні результати навчання:

- розуміння принципів побудови розподілених і мікросервісних архітектур;
- володіння методами масштабування, балансування навантаження і забезпечення відмовостійкості;
- здатність використовувати сучасні інструменти контейнеризації та оркестрації;
- знання підходів до моніторингу, логування та трасування подій у розподіленому середовищі;
- уміння створювати архітектурну документацію з використанням UML, C4-моделей, ArchiMate;
- розуміння тенденцій і технологій розвитку галузі: Cloud Native, Edge Computing, IoT, Big Data, Serverless.

Таким чином, освітня компонента «Архітектура розподілених програмних систем» формує у здобувачів комплексне бачення сучасних архітектур програмного забезпечення, необхідне для роботи над інноваційними проєктами в IT та суміжних галузях — зокрема, у сфері інформаційного супроводу будівельних процесів, управління об’єктами та інфраструктурними системами.

ЛАБОРАТОРНА РОБОТА № 1

Тема: Аналіз вимог до розподіленої системи

Мета: навчитися визначати функціональні та нефункціональні вимоги до РПС

Теоретичні відомості

Розподілена система (distributed system) — це сукупність незалежних обчислювальних вузлів (серверів, контейнерів, пристроїв IoT), які взаємодіють між собою через мережу для досягнення спільної мети.

Кожен вузол може виконувати окрему частину загальної задачі — наприклад, зберігати дані, обробляти запити, проводити розрахунки чи забезпечувати візуалізацію.

Приклад у сфері будівництва:

- система моніторингу будівельного майданчика може складатися з:
- сервісу збору даних з датчиків температури й вібрацій (IoT-шлюз),
- серверу аналітики, що оцінює стабільність конструкцій,
- веб-додатку для інженерів, який показує результати аналізу у реальному часі.

Ці компоненти функціонують незалежно, але разом формують цілісну розподілену систему.

Основні властивості, які потрібно враховувати при аналізі вимог:

Масштабованість (Scalability) — можливість додавати нові вузли без зупинки системи (додати новий сервер для обробки відеопотоку з камер на будівництві).

Надійність (Reliability) — здатність працювати при відмові окремих вузлів (якщо один датчик не передає дані — система продовжує роботу).

Доступність (Availability) — відсоток часу, коли система доступна користувачам (99.9% uptime у системі планування техніки).

Узгодженість (Consistency) — однаковий стан даних у різних вузлах (зміна статусу замовлення має бути однакова у вебі й мобільному).

Безпека (Security) — захист даних від несанкціонованого доступу (аутентифікація інженерів через корпоративний SSO).

Затримки (Latency) — час відгуку системи (у відеомоніторингу — важливо, щоб затримка не перевищувала 1 секунди).

Першим кроком при створенні будь-якої РПС є визначення користувачів і сторін, що взаємодіють із системою.

Їх часто групують у три рівні:

- Зовнішні користувачі — кінцеві користувачі, клієнти, замовники.

- Внутрішні користувачі — адміністратори, технічний персонал, менеджери.
- Інтегровані системи — сторонні сервіси або пристрої, що обмінюються даними (API, сенсори, CRM).

Приклад для системи управління будівельними проєктами:

- Замовник — контролює бюджет і строки.
- Інженер — додає звіти про виконані роботи.
- Архітектор — переглядає проєктну документацію.
- IoT-система — передає показники навантаження на конструкції.

Функціональні вимоги описують, що система повинна робити:

- користувач може створювати будівельні об’єкти;
- система відображає дані з IoT-сенсорів;
- адміністратор має змогу експортувати звіт у PDF.

Нефункціональні вимоги описують як система повинна працювати:

- затримка відповіді ≤ 500 мс;
- підтримка до 10 000 одночасних з’єднань;
- щоденне резервне копіювання бази даних;
- доступність 99.9%;
- можливість розгортання у Docker.

Context Diagram — це візуальна схема, яка показує, як система взаємодіє із зовнішнім середовищем.

На ній відображаються:

- система як “чорна скринька”;
- зовнішні актори (користувачі, інші системи);
- потоки даних між ними.

Use Case Diagram допомагає деталізувати функціональні вимоги — показати, які дії виконують користувачі системи.

Приклад для системи планування техніки:

Користувач: “Менеджер об’єкта”

→ “Додає будівельну техніку”, “Планує розклад”, “Переглядає зайнятість техніки”.

Система: “Виконує оптимізацію графіка”, “Надсилає сповіщення”.

Приклад короткого технічного завдання (ТЗ)

Назва: Розподілена система моніторингу стану конструкцій будівель.

Мета: Забезпечити безперервний збір і аналіз даних із сенсорів.

Основні функції:

- збір даних із IoT-пристроїв;
- аналітика у режимі реального часу;
- зберігання історичних даних;
- панель адміністратора;
- сповіщення про критичні значення.

Нефункціональні вимоги:

- середня затримка обробки ≤ 1 сек;
- доступність 99.95%;
- можливість масштабування до 1000 сенсорів.

Аналіз вимог — це ключовий етап, що визначає успішність усього проєкту.

Саме тут формується архітектурна логіка майбутньої системи, і помилки на цьому етапі призводять до значних витрат у розробці.

Чітке визначення вимог, контексту та ролей користувачів забезпечує основу для подальшого проєктування розподіленої архітектури.

Завдання

1. Обрати предметну область (наприклад, система моніторингу будівельних робіт).
2. Визначити користувачів та точки взаємодії (веб, мобільний, IoT).
3. Описати нефункціональні вимоги: масштабованість, надійність, безпека.
4. Побудувати **Use Case Diagram** та **Context Diagram** (опціонально).

Контрольні запитання

1. Що таке розподілена система?
2. Які типи архітектур використовуються у розподілених системах?
3. Що таке функціональні та нефункціональні вимоги?
4. Як визначають користувачів і зацікавлені сторони системи?
5. Що таке контекстна діаграма і для чого вона потрібна?
6. Які типові ризики при проєктуванні розподілених систем?
7. Як вимоги впливають на вибір архітектурного стилю?

ЛАБОРАТОРНА РОБОТА № 2

Тема: Проєктування мікросервісної архітектури

Мета: Навчитися декомпозувати систему на мікросервіси

Теоретичні відомості

Мікросервісна архітектура (microservice architecture) — це підхід до проєктування програмних систем, у якому застосунок розділяється на низку незалежних сервісів, кожен з яких виконує одну чітку бізнес-функцію та взаємодіє з іншими через добре визначений інтерфейс (зазвичай REST або gRPC API). Кожен мікросервіс має власний життєвий цикл, базу даних і може розгортатися незалежно від інших компонентів системи.

На відміну від монолітної архітектури, де всі функції системи зосереджені в одному великому додатку, мікросервіси дозволяють розробляти, масштабувати й оновлювати частини системи окремо. Це особливо важливо для розподілених систем, де різні підсистеми можуть мати різні вимоги до продуктивності, безпеки чи обсягів даних.

Приклад у сфері будівництва:

- система управління будівельними об’єктами може бути розділена на кілька мікросервісів:
- сервіс «Проєкти» — зберігає дані про об’єкти будівництва, строки та відповідальних;
- сервіс «Працівники» — керує даними про персонал і бригади;
- сервіс «Техніка» — відповідає за облік технічних засобів та їх розклад;
- сервіс «Звіти» — формує фінансові й технічні звіти;
- сервіс «Сповіщення» — надсилає повідомлення менеджерам про зміни або затримки.

Усі ці сервіси взаємодіють через API-шлюз (API Gateway), який виступає єдиною точкою входу для клієнтів (веб-додатку, мобільного додатку, інтеграцій з іншими системами). Це дозволяє спростити маршрутизацію запитів і централізовано керувати безпекою, аутентифікацією та моніторингом.

Ключовими принципами побудови мікросервісної архітектури є незалежність, слабке зв’язування, автономність даних та можливість самостійного розгортання. Сервіс не повинен напряму залежати від внутрішніх структур іншого сервісу. Для обміну даними між сервісами використовуються або синхронні HTTP-запити (REST/gRPC), або асинхронна взаємодія через брокер повідомлень (RabbitMQ, Kafka).

Дуже важливим є поняття межі сервісу (bounded context) — кожен сервіс повинен мати чітко визначену відповідальність, не дублювати логіку інших компонентів і працювати з власними даними. Наприклад, сервіс «Техніка» не має напряму змінювати інформацію про працівників — натомість він може отримати її через API сервісу «Працівники».

Щоб система залишалася стабільною і керованою, кожен мікросервіс має зберігати свої дані у власній базі, що дозволяє уникнути конфліктів при оновленнях або міграціях. Це називається принципом “database per service”. Якщо ж потрібно об’єднати інформацію з різних сервісів (наприклад, для створення загального звіту про ефективність будівництва), використовується сервіс-агрегатор або шар бізнес-аналітики.

Мікросервісна архітектура має низку переваг:

- можливість паралельної розробки різними командами;
- масштабування лише тих компонентів, що потребують навантаження;
- зручність оновлень без зупинки всієї системи;
- висока стійкість до відмов (збій одного сервісу не призводить до зупинки всієї системи);
- гнучкість у виборі технологій для різних сервісів.

Разом із перевагами існують і складнощі: необхідність правильної організації взаємодії між сервісами, централізованого моніторингу, безпечної автентифікації, а також підвищені вимоги до DevOps-інфраструктури (CI/CD, контейнеризація, оркестрація).

Приклад: якщо компанія зводить кілька об’єктів одночасно, і кількість користувачів зростає у кілька разів, то достатньо збільшити кількість копій лише сервісу «Аналітика», який відповідає за формування

звітів, — решта сервісів залишаються без змін. Таким чином досягається гнучке горизонтальне масштабування.

Для документування структури мікросервісної архітектури часто використовують нотації **C4 Model (Container/Component Level)** або **Deployment Diagram** з UML, де описується, які сервіси взаємодіють між собою, які технології використовуються, які бази даних підключено та через які інтерфейси здійснюється зв’язок.

Проєктування мікросервісної архітектури є основою побудови будь-якої сучасної розподіленої програмної системи. Від правильного визначення меж сервісів, способів взаємодії між ними та вибору технологічного стеку залежить продуктивність, масштабованість і надійність всієї системи.

Завдання

1. Визначити 2–3 основних мікросервісів (наприклад: управління працівниками, обладнанням, календарем робіт, оплатами, аналітика).
2. Для кожного мікросервісу визначити базу даних, API, черги повідомлень.
3. Побудувати Component Diagram або C4 Level 2.

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Що таке мікросервісна архітектура?
2. У чому відмінність моноліту від мікросервісів?
3. Які принципи проєктування мікросервісів ви знаєте?
4. Як визначити межі сервісів (bounded context)?
5. Що таке API Gateway і навіщо він потрібен?
6. Як забезпечується узгодженість даних між мікросервісами?
7. Які переваги та недоліки має мікросервісний підхід?

ЛАБОРАТОРНА РОБОТА № 3

Тема: Моделювання взаємодії сервісів

Мета: Змоделювати потоки даних у системі

Теоретичні відомості

У розподілених програмних системах взаємодія між компонентами — це основа їхньої роботи. Мікросервіси або окремі модулі системи повинні обмінюватися даними, повідомленнями чи запитами так, щоб загальний бізнес-процес виконувався узгоджено. Правильне моделювання взаємодії сервісів дозволяє уникнути помилок при реалізації, підвищує стабільність системи та спрощує масштабування.

Взаємодія сервісів може бути **синхронною** або **асинхронною**.

Синхронна взаємодія означає, що сервіс викликає інший сервіс і чекає на відповідь. Такий підхід простіше реалізувати, але він створює залежності між компонентами та може призвести до затримок у випадку, якщо один із сервісів недоступний.

Асинхронна взаємодія, навпаки, базується на передачі повідомлень або подій через черги (message queues). У цьому випадку відправник не очікує відповіді — він просто публікує повідомлення, а інші сервіси реагують на нього у свій час.

Приклад у будівельній сфері: система контролю постачання матеріалів може працювати асинхронно. Сервіс «Замовлення» надсилає подію “матеріал замовлено”, сервіс «Склад» реагує на цю подію, зменшуючи кількість залишків, а сервіс «Логістика» отримує власну подію “доставка запланована”. Кожен з них працює незалежно, але разом формують узгоджений процес.

Для реалізації асинхронної взаємодії часто використовуються **брокери повідомлень** — спеціальні системи, що приймають, зберігають і доставляють повідомлення між сервісами. Найпопулярніші рішення: **RabbitMQ, Apache Kafka, NATS, Redis Streams**. Вони дозволяють будувати подійно-орієнтовані архітектури (event-driven architecture), у яких логіка роботи визначається не запитами, а послідовністю подій у системі.

Синхронна взаємодія зазвичай реалізується через REST або gRPC API. REST використовує HTTP-протокол і підходить для інтеграції з веб-додатками чи сторонніми сервісами. gRPC — це більш ефективний протокол на основі HTTP/2, який забезпечує меншу затримку і підходить для взаємодії між внутрішніми сервісами в мікросервісній архітектурі.

Наприклад, сервіс «Звіти» може через REST API запитати у сервісу «Працівники» інформацію про кількість відпрацьованих годин, а сервіс «Планування» — через gRPC викликати розрахунок навантаження на техніку.

Щоб описати, як саме сервіси взаємодіють, використовуються **діаграми послідовності (Sequence Diagram)**. Вони показують порядок викликів між компонентами системи, час відправлення і отримання повідомлень.

У прикладі з моніторингом будівельного майданчика Sequence Diagram може виглядати так:

1. IoT-сенсор надсилає показники до сервісу збору даних.
2. Сервіс збору передає дані у брокер повідомлень (Kafka).
3. Сервіс аналітики зчитує подію і обчислює середні показники.
4. Сервіс «Сповіщення» отримує результат аналітики й надсилає попередження у мобільний застосунок інженера.

Також використовуються **C4 Model (Container Level або Component Level)** — більш високорівневі діаграми, що описують логічні взаємозв’язки між сервісами, протоколи обміну, бази даних та інтерфейси.

Важливим аспектом у моделюванні є забезпечення **ідемпотентності** сервісів — властивості, за якої повторне виконання однієї й тієї ж операції не змінює результату. Це критично для систем, де повідомлення можуть надходити кілька разів через затримки або повторні відправлення. Наприклад, якщо сервіс «Склад» двічі отримує подію “матеріал отримано”, він не повинен двічі збільшити кількість одиниць товару.

Ще один важливий принцип — **fault tolerance (стійкість до збоїв)**. У розподілених системах сервіси не завжди доступні. Щоб уникнути збоїв, використовують механізми **retry (повторна спроба)**, **circuit breaker (автоматичне відключення недоступного сервісу)** та **dead letter queue**

(черга для невдалих повідомлень). Наприклад, якщо сервіс аналітики тимчасово не працює, події накопичуються в черзі RabbitMQ і будуть оброблені після відновлення.

У складних системах застосовують також **саги (saga pattern)** — спосіб узгодження довготривалих транзакцій між кількома сервісами. Наприклад, у системі управління замовленнями:

1. Сервіс «Замовлення» створює новий запис;
2. Сервіс «Оплата» списує кошти;
3. Сервіс «Склад» резервує матеріали.

Якщо один із кроків не вдався (наприклад, немає матеріалів на складі), інші сервіси виконують компенсаційні дії — повернення коштів і скасування замовлення.

У системах для будівництва саги часто застосовуються для управління проєктами — наприклад, коли потрібно узгодити декілька взаємопов’язаних дій: замовлення матеріалів, бронювання техніки та найм працівників. Усі вони мають бути виконані або скасовані узгоджено.

Моделювання взаємодії сервісів дозволяє проєктувати систему так, щоб вона залишалася стійкою, гнучкою і зрозумілою навіть у масштабах сотень вузлів. Воно забезпечує основу для подальшої реалізації мікросервісної архітектури, де кожен сервіс виконує свою роль, а разом вони формують цілісну інтелектуальну систему.

Завдання

1. Вибрати сценарій (наприклад, подання звіту з будівельного майданчика).
2. Описати послідовність взаємодії сервісів у вигляді **Sequence Diagram**.
3. Вказати тип взаємодії (HTTP/gRPC/Message Queue).
4. Побудувати **Deployment Diagram** із зазначенням контейнерів.

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту

4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Які є види міжсервісної взаємодії?
2. У чому різниця між синхронною та асинхронною взаємодією?
3. Що таке брокер повідомлень?
4. Які переваги має подійно-орієнтована архітектура?
5. Для чого використовують діаграму послідовності?
6. Що означає ідемпотентність сервісу?
7. Як забезпечити надійну доставку повідомлень у розподіленій системі?

ЛАБОРАТОРНА РОБОТА № 4

Тема: Технологічна архітектура

Мета: Розробити схему технологічного забезпечення РПС.

Теоретичні відомості

Технологічна архітектура розподіленої системи визначає, на яких платформах і в яких середовищах виконуються її компоненти. Вона описує інфраструктуру, технологічний стек, способи розгортання, мережеву взаємодію, безпеку, а також взаємозв'язок між апаратними і програмними ресурсами. Іншими словами, це "каркас" усієї системи, який забезпечує працездатність мікросервісів і взаємодію між ними.

У сучасних розподілених системах технологічна архітектура зазвичай базується на **контейнеризації** та **хмарних технологіях**. Контейнери (Docker, Podman) дозволяють упаковувати застосунок разом із усіма залежностями, щоб він міг стабільно працювати в будь-якому середовищі. Це значно спрощує розгортання, оновлення і масштабування. Оркестратори, такі як **Kubernetes**, забезпечують автоматичне керування контейнерами, балансування навантаження, моніторинг стану й відновлення при збоях.

Приклад у будівельній сфері: компанія розробляє систему управління будівельними майданчиками, яка складається з кількох мікросервісів — «Працівники», «Техніка», «Моніторинг», «Аналітика», «Сповіщення». Кожен із них контейнеризовано за допомогою Docker і розгорнуто у кластері Kubernetes. Оркестратор слідкує, щоб кожен сервіс мав потрібну кількість копій (реплік), перезапускає його при збої, а також розподіляє навантаження між вузлами. Це дозволяє системі залишатися доступною навіть при виході з ладу частини серверів.

Типова технологічна архітектура включає кілька шарів:

1. **Інфраструктурний рівень** — фізичні або віртуальні сервери, мережеве обладнання, системи зберігання даних.
2. **Платформний рівень** — середовище виконання (Docker, Kubernetes, OpenShift), бази даних, брокери повідомлень, інструменти моніторингу.

3. **Рівень застосунків** — мікросервіси, API-шлюзи, служби авторизації, системи кешування.
4. **Рівень доступу користувачів** — веб-інтерфейси, мобільні додатки, API для інтеграцій.

Сучасні розподілені системи дедалі частіше розгортаються у **хмарних середовищах** (AWS, Google Cloud, Azure, DigitalOcean), що дає змогу гнучко керувати ресурсами — автоматично збільшувати кількість вузлів при навантаженні й зменшувати при простої. Це знижує витрати і підвищує стабільність.
Наприклад, система для управління будівельними об’єктами може використовувати AWS EC2 для обчислень, S3 для зберігання креслень і фото, RDS для бази даних, а SNS/SQS для обміну повідомленнями між сервісами.

Важливою складовою технологічної архітектури є **балансувальники навантаження (load balancers)**. Вони рівномірно розподіляють запити між сервісами, щоб уникнути перевантаження одного вузла. Для цього часто використовуються **NGINX, HAProxy, Traefik**, або вбудовані рішення у Kubernetes — **Ingress Controller**. Наприклад, якщо кілька інженерів одночасно працюють у веб-додатку, балансувальник автоматично спрямовує запити до вільних копій сервісу, забезпечуючи стабільну роботу системи.

Ще один важливий елемент — **системи кешування** (Redis, Memcached). Вони дозволяють зберігати тимчасові дані, щоб зменшити навантаження на бази даних. У будівельному застосунку це може бути кешування інформації про поточні проєкти або статистику за останню добу, щоб пришвидшити завантаження сторінок.

З точки зору зберігання даних технологічна архітектура може включати **реляційні бази даних** (PostgreSQL, MySQL) для структурованих даних і **NoSQL-рішення** (MongoDB, Cassandra, InfluxDB) для неструктурованих або часових рядів. Наприклад, показники з IoT-сенсорів про температуру чи вібрації доцільно зберігати в InfluxDB, а дані про користувачів і техніку — у PostgreSQL.

Безпека в технологічній архітектурі забезпечується через кілька рівнів захисту:
— **аутентифікація та авторизація** (OAuth 2.0, JWT, Keycloak);
— **шифрування трафіку** (TLS/HTTPS);
— **ізоляція контейнерів** (namespace, cgroups);
— **обмеження доступу до ресурсів** через ролі (Role-Based Access Control у Kubernetes).

У великих розподілених системах важливу роль відіграє **спостережуваність (observability)** — здатність відстежувати стан усіх компонентів у реальному часі. Для цього використовують **Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana)**. Вони збирають метрики (затримки, використання CPU, кількість запитів), що дозволяє швидко виявляти проблеми.

Приклад: якщо у системі моніторингу будівель зростає час обробки запитів від сенсорів, Prometheus фіксує аномалію, Grafana відображає графік з піком затримки, а Kubernetes автоматично додає ще одну копію сервісу, розвантажуючи систему.

Технологічна архітектура також визначає стратегію розгортання (deployment strategy). Існують різні підходи — **rolling update** (поступове оновлення без простою), **blue-green deployment** (одночасне існування старої й нової версії), **canary release** (оновлення для частини користувачів). Наприклад, при впровадженні нового сервісу для управління технікою можна спочатку активувати його лише для одного об’єкта будівництва, щоб перевірити стабільність роботи, і лише потім розгорнути для всіх.

Таким чином, технологічна архітектура є критичним елементом розподіленої програмної системи. Вона визначає, наскільки ефективно і стабільно система працює, як швидко її можна масштабувати, оновлювати і відновлювати після збоїв. Правильно спроєктована технологічна архітектура — це основа для безперервної, безпечної та гнучкої роботи всієї інформаційної інфраструктури.

Завдання

1. Обрати стек технологій (наприклад, Node.js + RabbitMQ + PostgreSQL + Docker).

2. Побудувати **топологію розгортання** у хмарі (AWS/GCP/Azure).
3. Описати способи забезпечення стійкості: load balancer, replica sets, autoscaling.

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Що таке технологічна архітектура?
2. Які рівні архітектури розрізняють у РПС?
3. Які типові компоненти включає інфраструктура розподіленої системи?
4. Що таке контейнер і для чого його використовують?
5. Як забезпечити масштабування у хмарному середовищі?
6. Яку роль відіграє балансувальник навантаження?
7. У чому переваги Kubernetes перед Docker Compose?

ЛАБОРАТОРНА РОБОТА № 5

Тема: Розробка прототипу сервісу

Мета: Створити мікросервіс із REST або gRPC API

Теоретичні відомості

Розробка прототипу сервісу — це етап, на якому архітектурна модель системи переходить у практичну площину. Прототип дозволяє перевірити правильність вибору технологій, структуру API, взаємодію компонентів і загальну життєздатність архітектурних рішень. У розподілених системах найчастіше використовуються два підходи для створення інтерфейсів взаємодії між сервісами — **REST** та **gRPC**.

REST (Representational State Transfer) — це архітектурний стиль, що базується на використанні стандартних HTTP-методів:

- GET — отримання ресурсів,
- POST — створення,
- PUT/PATCH — оновлення,
- DELETE — видалення.

REST-сервіси оперують поняттям “ресурсів”, які ідентифікуються URL-адресами. Кожен запит супроводжується кодом відповіді HTTP (200, 404, 500 тощо) і, за потреби, передає дані у форматі JSON або XML. Такий підхід добре підходить для веб-систем, інтеграцій із зовнішніми клієнтами, мобільних застосунків і публічних API.

Приклад у сфері будівництва: компанія створює систему управління технікою. Сервіс “Техніка” може мати REST-інтерфейс:

`GET /equipment` — повертає список усієї техніки;

`POST /equipment` — додає нову одиницю;

`PUT /equipment/{id}` — оновлює інформацію;

`DELETE /equipment/{id}` — видаляє запис.

Клієнтський застосунок (веб-панель інженера) може надсилати ці запити через звичайний HTTP і отримувати JSON-відповіді з поточним станом техніки.

REST-підхід є простим у реалізації, підтримується будь-якою мовою програмування і легко тестується за допомогою інструментів на кшталт **Postman**, **curl** або **Swagger UI**. Однак при інтенсивній внутрішній взаємодії між мікросервісами REST може виявитися неефективним — великі JSON-повідомлення, накладні HTTP-заголовки та відсутність контрактного контролю знижують продуктивність. У таких випадках доцільно використовувати **gRPC**.

gRPC (Google Remote Procedure Call) — це фреймворк, який реалізує концепцію виклику віддалених процедур поверх протоколу HTTP/2. Він використовує бінарний формат серіалізації даних **Protocol Buffers (protobuf)**, що робить обмін набагато швидшим і компактнішим.

На відміну від REST, де виклики описуються URL-шляхами, у gRPC визначаються сервіси та методи у файлі `.proto`. На основі цього файлу автоматично генеруються клієнтські та серверні класи для різних мов програмування. Це забезпечує суворий контракт між сторонами — клієнт не може викликати неіснуючий метод або передати некоректні поля.

Приклад: у системі моніторингу будівництва сервіс “Сенсори” може визначати методи:

```
service SensorData {
    rpc SendReading (SensorRequest) returns (SensorResponse);
    rpc GetAverage (AverageRequest) returns (AverageResponse);
}
```

Сервіс “Аналітика” викликає метод `SendReading`, передаючи показники з датчиків. Сервер обробляє їх і повертає відповідь. Завдяки використанню HTTP/2 можна передавати багато запитів у одному з’єднанні, що мінімізує затримки — це критично для систем реального часу.

Порівняно з REST, gRPC краще підходить для **внутрішніх сервісів** у мікросервісній архітектурі — коли потрібна висока швидкість і контроль над структурою повідомлень. REST натомість більш зручний для зовнішніх

API або мобільних клієнтів. Часто обидва підходи комбінуються: зовнішні користувачі працюють через REST-шлюз, а внутрішні сервіси обмінюються даними через gRPC або брокери повідомлень.

Під час розробки прототипу важливо враховувати **структуру сервісу**. Зазвичай він складається з:
— шару представлення (API), що приймає запити користувача або інших сервісів;
— шару бізнес-логіки, який обробляє дані відповідно до правил предметної області;
— шару доступу до даних (repository або DAO), який взаємодіє з базою;
— конфігураційного шару, де визначаються параметри середовища, змінні оточення, підключення до брокерів тощо.

Для зручності розробки використовують фреймворки: **FastAPI**, **Spring Boot**, **Go Fiber**, **ASP.NET Core**, **gRPC-Go** або **gRPC-Java**. У середовищах із кількома сервісами варто налаштувати **Dockerfile**, щоб кожен сервіс можна було запускати у контейнері.

Приклад практичного сценарію: сервіс “Моніторинг” приймає дані з IoT-сенсорів у gRPC-форматі, обробляє їх і зберігає у базі даних InfluxDB. Сервіс “Веб-панель” через REST API запитує агреговані дані для відображення на дашборді. Таким чином, обидва підходи доповнюють один одного — gRPC забезпечує швидку передачу даних усередині системи, а REST — зручний доступ для користувачів.

Під час створення прототипу обов’язково потрібно реалізувати обробку помилок і валідацію даних. Для REST це коди HTTP-помилок (400 — неправильний запит, 404 — не знайдено, 500 — внутрішня помилка сервера), для gRPC — статуси (`OK`, `INVALID_ARGUMENT`, `UNAVAILABLE` тощо). Це дозволяє сервісам коректно реагувати на непередбачені ситуації і зберігати стабільність системи.

Розробка прототипу сервісу — це перевірка архітектурної концепції на практиці. Навіть мінімальна реалізація одного ендпоінта або RPC-виклику дозволяє оцінити швидкодію, обсяг даних, зручність контракту й потенційні вузькі місця. Надалі прототип може бути розширений до повноцінного мікросервісу, інтегрованого в систему.

Завдання

1. Реалізувати CRUD-операції для однієї сутності (наприклад, «Об’єкт будівництва»).
2. Зберігати дані у базі (PostgreSQL - https://www.postgresql.org).
3. Налаштувати контейнеризацію за допомогою Dockerfile (https://docs.docker.com/reference/dockerfile).

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Що таке REST і які основні принципи RESTful API?
2. Які методи HTTP використовуються у REST API?
3. Чим gRPC відрізняється від REST?
4. Як описуються дані у gRPC?
5. Що таке статус-коди HTTP?
6. Які інструменти використовують для тестування API?
7. Як забезпечити безпеку REST API?

ЛАБОРАТОРНА РОБОТА № 6

Тема: Реалізація міжсервісної взаємодії

Мета: реалізувати зв’язок між двома сервісами

Теоретичні відомості

Міжсервісна взаємодія є ключовим аспектом побудови розподілених програмних систем. Навіть найкраще спроєктовані мікросервіси втрачають ефективність без правильно організованого обміну даними. Взаємодія між сервісами може бути реалізована за двома основними підходами — **синхронним** (через прямі запити до API) або **асинхронним** (через обмін повідомленнями або подіями).

У синхронній взаємодії один сервіс викликає інший безпосередньо, очікуючи відповіді. Найчастіше це REST або gRPC-виклики. Такий підхід простіший у реалізації, але створює **жорсткі залежності** між сервісами — якщо один із них не працює, інші можуть також зупинитися. Наприклад, у системі керування будівництвом сервіс “Звіти” може запитувати через REST дані у сервісу “Техніка”. Якщо той недоступний, звіт не формується. Для невеликих систем це прийнятно, але у великих — такий тип зв’язків знижує надійність і масштабованість.

Асинхронна взаємодія базується на принципі **event-driven architecture (архітектури, керованої подіями)**. У цьому підході сервіси не викликають один одного безпосередньо, а спілкуються через **черги повідомлень** або **брокери подій**. Кожен сервіс може публікувати події (“material_ordered”, “sensor_alert”), на які інші сервіси підписані. Це забезпечує слабке зв’язування: відправник не знає, хто отримає подію, і не залежить від нього у часі.

Для реалізації асинхронної взаємодії використовуються брокери повідомлень — **RabbitMQ, Apache Kafka, NATS, Redis Streams** тощо. RabbitMQ базується на протоколі AMQP і добре підходить для класичних черг: коли один відправник передає повідомлення одному або кільком споживачам. Kafka, натомість, створена для потокової обробки подій і здатна обробляти сотні тисяч повідомлень за секунду. Вона зберігає історію подій, що дозволяє сервісам повторно їх зчитувати — це зручно для аналітики або побудови історичних звітів.

Приклад у сфері будівництва: компанія використовує розподілену систему моніторингу проєктів. Коли сервіс “Постачання” фіксує прибуття матеріалів, він публікує подію “materials_delivered” у брокер RabbitMQ. Сервіс “Аналітика” отримує цю подію і оновлює статистику по об’єкту, а сервіс “Сповіщення” надсилає повідомлення менеджеру. Якщо “Сповіщення” тимчасово не працює, повідомлення залишиться у черзі, поки сервіс не відновиться — це гарантує надійність доставки.

Під час реалізації міжсервісної взаємодії важливо враховувати кілька технічних аспектів:

1. **Ідемпотентність операцій** — повторне отримання того самого повідомлення не повинно призводити до дублювання дій. Наприклад, якщо подія “замовлення виконано” надійшла двічі, система не має двічі списати ресурси.
2. **Гарантії доставки**. Існують три основні рівні:
   — *at most once* — повідомлення може загубитися, але ніколи не буде продубльоване;
   — *at least once* — повідомлення може бути доставлене повторно, але не загубиться;
   — *exactly once* — повідомлення доставляється рівно один раз (зазвичай досягається складними транзакційними механізмами або ідемпотентністю обробки).
3. **Обробка помилок і повторні спроби (retry)**. Якщо повідомлення не вдалося обробити, воно може бути повторно відправлене після певної затримки або переміщене у спеціальну чергу — **dead letter queue**, щоб адміністратор міг перевірити проблему.
4. **Моніторинг черг** — необхідно контролювати швидкість обробки повідомлень, кількість непрочитаних подій, час затримки. Для цього використовують панелі адміністрування RabbitMQ або Kafka, а також інтеграції з Prometheus і Grafana.

У розподілених системах із великою кількістю сервісів часто застосовують **патерн “Сага” (Saga pattern)**. Він допомагає узгоджувати транзакції між сервісами без єдиної спільної бази даних. Наприклад, у системі управління будівельними процесами:

1. Сервіс “Замовлення” створює нове замовлення матеріалів.

2. Сервіс “Постачання” бронює їх на складі.
3. Сервіс “Логістика” планує доставку.
   Якщо на будь-якому етапі сталася помилка (наприклад, немає потрібного матеріалу), інші сервіси виконують компенсаційні дії — скасовують бронювання і повідомляють менеджера. Так досягається логічна узгодженість без централізованої транзакції.

Ще один популярний підхід — **choreography** (взаємодія через події) і **orchestration** (взаємодія через координуючий сервіс). У першому випадку сервіси “спілкуються” між собою безпосередньо через події, у другому — окремий “оркестратор” визначає порядок виконання дій. Наприклад, у будівельному проєкті, коли потрібно створити об’єкт, призначити бригаду та запустити закупівлі — оркестратор координує виклики сервісів у правильній послідовності.

Для тестування міжсервісної взаємодії використовують утиліти на кшталт **Postman**, **Kafka CLI**, **RabbitMQ Management UI**, а також автоматизовані тести на основі **Contract Testing (Pact)** — щоб перевірити сумісність API між сервісами.

У практичному сенсі впровадження міжсервісної взаємодії у розподіленій архітектурі дозволяє системі залишатися стабільною навіть при зростанні навантаження або тимчасових збоях. Наприклад, якщо у системі моніторингу будівництва сервіс “Сенсори” надсилає тисячі подій за хвилину, брокер Kafka може зберігати їх без перевантаження аналітичного сервісу. Той обробляє події у своєму темпі, зберігаючи плавність роботи всієї системи.

Таким чином, правильно спроєктована міжсервісна взаємодія забезпечує **надійність, масштабованість і гнучкість** розподіленої системи. Вона дозволяє сервісам бути незалежними, взаємодіяти ефективно й підтримувати цілісність бізнес-процесів навіть у складних сценаріях.

Завдання

1. Налаштувати чергу RabbitMQ або Kafka (https://kafka.apache.org).
2. Реалізувати обмін подіями між двома мікросервісами.
3. Продемонструвати обробку події у Postman або логах контейнера.

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Що таке асинхронна взаємодія?
2. Які системи повідомлень ви знаєте?
3. Як працює RabbitMQ?
4. Що таке “publisher” і “subscriber”?
5. У чому різниця між чергою та топіком?
6. Як обробляти помилки у подійно-орієнтованих системах?
7. Як тестувати міжсервісну взаємодію?

ЛАБОРАТОРНА РОБОТА № 7

Тема: Оркестрація мікросервісів
Мета: Навчитися керувати кількома контейнерами

Теоретичні відомості

Оркестрація мікросервісів — це процес автоматизованого керування розгортанням, масштабуванням, оновленням і життєвим циклом контейнеризованих застосунків. Вона дозволяє координувати роботу великої кількості контейнерів, які утворюють розподілену програмну систему. Без оркестрації адміністрування десятків або сотень сервісів було б практично неможливим.

Контейнеризація (Docker, Podman) дала змогу ізолювати кожен мікросервіс у власне середовище з усіма залежностями. Проте навіть у невеликому проєкті кількість контейнерів може швидко зрости: окремо для бази даних, брокера повідомлень, API-шлюзу, моніторингу, аналітики тощо. Оркестрація вирішує завдання їх централізованого керування, моніторингу стану та автоматичного відновлення при збоях.

Найпоширенішими інструментами оркестрації є **Docker Compose** та **Kubernetes**.

Docker Compose — це зручний інструмент для локального розгортання багатоконтейнерних середовищ. У файлі `docker-compose.yml` описується, які сервіси потрібно запустити, які образи використовуються, які порти відкриті, які змінні оточення задаються, і які томи (volumes) монтуються для збереження даних.
Це дозволяє створити повноцінну інфраструктуру системи однією командою:
`docker compose up -d`.

Приклад у будівельній сфері: під час розробки системи моніторингу будівельних об’єктів студент може підняти таке середовище:

- сервіс `sensor-service` — приймає дані з сенсорів через gRPC;

- сервіс `analytics-service` — обробляє показники та виявляє відхилення;
- сервіс `web-dashboard` — відображає інформацію у браузері;
- база даних `influxdb`;
- брокер повідомлень `rabbitmq`.

  Усі ці контейнери описуються в одному YAML-файлі, після чого система автоматично створює мережу, налаштовує залежності й запускає їх у правильному порядку.

Kubernetes — це більш потужна платформа для оркестрації контейнерів, що використовується у промислових масштабах. Вона автоматично розподіляє контейнери по вузлах кластера, слідкує за їхнім станом, перезапускає у разі збоїв, виконує масштабування і оновлення без простоїв. Основні об'єкти Kubernetes:

- **Pod** — найменша одиниця розгортання, що містить один або кілька контейнерів;
- **Deployment** — описує, як має бути розгорнуто й оновлено Pod-и (наприклад, скільки копій сервісу потрібно);
- **Service** — забезпечує доступ до Pod-ів через стабільну IP-адресу або DNS-ім'я;
- **Ingress** — керує зовнішнім доступом (HTTP/HTTPS) до сервісів;
- **ConfigMap** і **Secret** — зберігають параметри конфігурацій та чутливі дані.

У практиці будівельних IT-рішень Kubernetes часто використовується для підтримки систем моніторингу або аналітики. Наприклад, у компанії, що керує декількома великими будівництвами, кожен об'єкт може мати власний набір мікросервісів (аналітика, датчики, відеоспостереження). Kubernetes дозволяє автоматично масштабувати ті частини системи, які отримують більше навантаження — наприклад, додати більше копій сервісу "Сенсори" у години пік.

Механізми масштабування — одна з ключових переваг Kubernetes. Система може збільшувати або зменшувати кількість Pod-ів на основі поточного навантаження (Horizontal Pod Autoscaler). Це забезпечує оптимальне використання ресурсів і високу доступність без ручного втручання.

Ще однією важливою властивістю є **fault tolerance (стійкість до збоїв)**. Якщо один вузол кластера виходить з ладу, Kubernetes автоматично переносить Pod-и на інші вузли. Для зберігання стану використовуються **persistent volumes**, які не залежать від конкретного контейнера, тому дані залишаються навіть після його перезапуску.

Для безпеки застосовуються політики доступу (RBAC — Role-Based Access Control), що визначають, які користувачі чи сервіси мають право створювати, оновлювати або видаляти ресурси. З'єднання між компонентами шифруються, а параметри середовища зберігаються у Secret-об'єктах.

Оновлення застосунків у Kubernetes здійснюється за допомогою стратегій **Rolling Update** або **Blue-Green Deployment**. Перша поступово замінює старі версії Pod-ів на нові без зупинки системи, а друга дозволяє мати дві версії сервісу одночасно й переключати трафік між ними.
Наприклад, якщо в системі для контролю будівництва потрібно оновити сервіс "Аналітика", Kubernetes спочатку створює нову версію поряд із чинною, перевіряє її працездатність і лише потім поступово перенаправляє трафік.

У навчальних і невеликих промислових проєктах Docker Compose зазвичай використовується для локальної розробки, а Kubernetes — для продакшн-середовищ або тестових стендів. Обидва інструменти можуть співіснувати: розробник тестує мікросервіс у Compose, а DevOps-фахівець розгортає його у Kubernetes.

Приклад практичного сценарію: система моніторингу бетонування має три сервіси — "Сенсори", "Аналітика" і "Сповіщення". Розробник локально тестує їх у Compose. Після перевірки стабільності CI/CD-пайплайн автоматично публікує нові Docker-образи, Kubernetes оновлює Deployment-и, додає додаткові Pod-и для навантаження й відкриває доступ через Ingress Controller. Система залишається доступною навіть під час оновлення.

Отже, оркестрація — це невід'ємна частина архітектури розподілених програмних систем. Вона дозволяє ефективно керувати складними багатокомпонентними застосунками, гарантує стабільність,

масштабованість і безперервність роботи навіть у великих інфраструктурах. Без оркестрації сучасна мікросервісна система не може вважатися повноцінною.

Завдання

1. Розробити `docker-compose.yml` для запуску кількох сервісів.
2. Налаштувати мережеві зв'язки між контейнерами.
3. Перевірити роботу системи через REST API.

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв'язання задач
5. Висновки

Контрольні запитання

1. Що таке оркестрація контейнерів?
2. Які основні елементи Docker Compose?
3. Які основні компоненти Kubernetes (Pod, Deployment, Service)?
4. Як забезпечується масштабування у Kubernetes?
5. Що таке Helm і для чого його використовують?
6. Як реалізується стійкість до відмов у Kubernetes?
7. Як організувати спільну мережу для кількох контейнерів?

ЛАБОРАТОРНА РОБОТА № 8

Тема: Моніторинг і логування

Мета: Впровадити систему моніторингу та централізованого логування

Теоретичні відомості

Моніторинг і логування є критично важливими складовими будь-якої розподіленої програмної системи. Вони забезпечують спостережуваність (observability) — здатність розуміти, що відбувається всередині системи, як вона працює, чому виникають збої та як на них реагувати. У складних мікросервісних архітектурах, де одночасно функціонують десятки сервісів, без централізованого моніторингу неможливо гарантувати стабільність і безпеку.

Моніторинг — це процес збору, аналізу та візуалізації метрик, що характеризують стан системи. Метрики можуть відображати використання ресурсів (CPU, RAM, мережевий трафік), кількість запитів до сервісу, середній час відповіді, кількість помилок тощо. На основі цих показників можна визначати відхилення від норми й вчасно реагувати.

Логування (logging) — це запис подій, що відбуваються у системі: дії користувачів, помилки, транзакції, повідомлення про стан процесів. Логи допомагають відтворити послідовність подій під час розслідування інцидентів і є незамінним джерелом інформації для діагностики проблем.

У розподіленій системі важливо, щоб усі сервіси передавали свої логи у централізовану систему збору. Якщо логи залишаються всередині контейнера або на окремому сервері, знайти потрібну інформацію буде надзвичайно складно. Саме тому застосовуються спеціальні інструменти для централізованого збору, зберігання та пошуку логів — такі як **ELK Stack (Elasticsearch, Logstash, Kibana)** або **EFK Stack (Elasticsearch, Fluentd, Kibana)**.

- **Logstash** або **Fluentd** відповідають за збір логів із різних джерел (Docker-контейнери, файли, системні журнали).
- **Elasticsearch** — це високопродуктивна база для пошуку та аналітики текстових даних.

- **Kibana** — інструмент візуалізації, який дозволяє переглядати логи, будувати графіки, створювати інформаційні панелі (дашборди).

Приклад у будівельній тематиці: у системі моніторингу технічного стану будівель кожен сервіс — “Сенсори”, “Аналітика”, “Сповіщення” — генерує власні логи. Fluentd збирає їх із контейнерів і передає до Elasticsearch, а Kibana відображає аналітичні дашборди: скільки було отримано подій, скільки з них містили аномальні значення, коли востаннє виникала помилка зв’язку. Інженери можуть у реальному часі бачити, де саме виникла проблема — у сервісі, у мережі або в сенсорі.

Моніторинг метрик зазвичай реалізується через систему **Prometheus** у поєднанні з **Grafana**.

Prometheus збирає числові показники (метрики) з мікросервісів за HTTP-протоколом (формат /metrics) і зберігає їх у часовій базі даних. Grafana підключається до Prometheus і відображає ці метрики у вигляді графіків, таблиць та індикаторів.

Кожен мікросервіс може публікувати власні метрики — наприклад, кількість оброблених подій, середню затримку, кількість помилок 5xx або розмір черги повідомлень.

Приклад: у системі управління будівельними майданчиками Prometheus може збирати такі показники:

- сервіс “Аналітика”: середній час обробки даних із сенсорів (latency_seconds);
- сервіс “Сенсори”: кількість отриманих показників за хвилину (sensor_events_total);
- сервіс “Сповіщення”: кількість успішно відправлених повідомлень (notifications_sent_total).

Grafana відображає все це у вигляді дашборду, де видно загальну динаміку навантаження системи та ефективність обробки подій.

Важливою частиною моніторингу є **алертинг (оповіщення)**. Prometheus може автоматично відправляти сповіщення у Telegram, Slack або електронну пошту при перевищенні критичних порогів. Наприклад, якщо час відповіді сервісу перевищує 1 секунду або кількість помилок 5xx перевищує 10 за хвилину, система негайно повідомить адміністратора.

У сучасних розподілених системах часто застосовується концепція **“три стовпи спостережуваності” (three pillars of observability)**:

1. **Логи** — текстові повідомлення про події в системі.
2. **Метрики** — числові показники, що відображають стан системи.
3. **Трейси (traces)** — відстеження ланцюжка викликів між сервісами.

Розподілене трасування (distributed tracing) дозволяє бачити, як запит проходить через усю систему — від користувача до бази даних. Для цього використовують інструменти **Jaeger**, **Zipkin** або **OpenTelemetry**.

Наприклад, у будівельній системі, якщо інженер скаржиться, що звіт завантажується повільно, трасування допоможе побачити, де саме відбувається затримка — у сервісі аналітики, у базі даних або у REST-запиті до сервісу техніки.

Для збору телеметрії все частіше використовується **OpenTelemetry** — відкритий стандарт, що дозволяє об’єднати метрики, логи та трейси в єдину систему. Це спрощує інтеграцію з Grafana, Prometheus, Jaeger і хмарними платформами.

У будівельних компаніях, що використовують IoT-сенсори, моніторинг і логування мають практичне значення. Якщо, наприклад, у системі збору даних з’являється затримка або втрачено з’єднання з окремими пристроями, моніторинг одразу це виявляє, а логування допомагає зрозуміти причину — мережевий збій, помилку сенсора або перевантаження бази даних.

Ефективний моніторинг і централізоване логування дають можливість:

— вчасно реагувати на відмови та аномалії;
— знаходити “вузькі місця” системи;
— прогнозувати навантаження і планувати масштабування;
— підвищувати рівень безпеки і якості обслуговування.

Отже, моніторинг і логування — це не просто допоміжні інструменти, а повноцінна частина архітектури розподілених систем. Вони забезпечують прозорість, контроль і стабільність роботи всієї інфраструктури, що є особливо важливим для критичних галузей, таких як

будівництво, де навіть короткий збій може вплинути на безпеку або строки проєкту.

Завдання

1. Інтегрувати Prometheus (https://prometheus.io) + Grafana (https://grafana.com/) або Elastic Stack.
2. Налаштувати збір метрик з мікросервісів.
3. Побудувати дашборд з показниками роботи системи.

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Що таке метрика?
2. Які метрики найчастіше використовують у розподілених системах?
3. Як працює Prometheus?
4. Для чого потрібен Grafana?
5. Що таке централізоване логування?
6. Які переваги ELK Stack?
7. Як організувати моніторинг мікросервісів у Kubernetes?

ЛАБОРАТОРНА РОБОТА № 9

Тема: Тестування і масштабування

Мета: Перевірити працездатність системи під навантаженням

Теоретичні відомості

Тестування і масштабування є завершальними, але надзвичайно важливими етапами розробки розподіленої програмної системи. Вони дозволяють переконатися, що система працює стабільно, витримує навантаження, правильно реагує на збої і здатна зростати разом із кількістю користувачів або обсягом даних. Без цих етапів навіть добре спроєктована архітектура може виявитися нестійкою у реальному середовищі.

Тестування розподілених систем охоплює кілька рівнів і типів перевірок:

1. **Функціональне тестування** — перевіряє, чи правильно сервіси виконують свої задачі згідно з вимогами. Наприклад, чи система управління технікою правильно розподіляє ресурси між будівельними об’єктами.
2. **Інтеграційне тестування** — перевіряє взаємодію між сервісами. Наприклад, чи сервіс “Постачання” правильно отримує події від “Замовлень” через RabbitMQ, і чи коректно обробляє їх “Аналітика”.
3. **Навантажувальне тестування (Load Testing)** — визначає, як система поводиться під нормальним і піковим навантаженням. Дозволяє оцінити продуктивність, час відгуку, кількість одночасних користувачів і стабільність при зростанні трафіку.
4. **Стрес-тестування (Stress Testing)** — імітує ситуації, коли система перевищує свої ресурси. Мета — визначити, при якому навантаженні система “ламається” і як вона відновлюється після збою.
5. **Тестування на стійкість (Chaos Testing)** — перевіряє, як система реагує на відмови компонентів. Наприклад, штучно “вимикають” один мікросервіс, базу даних або вузол кластера, щоб перевірити, чи інші частини системи продовжують працювати. Такий підхід використовується у Netflix під назвою **Chaos Monkey**.

Для проведення навантажувального тестування застосовують інструменти **k6, Apache JMeter, Locust, Gatling**. Вони дозволяють

створювати сценарії, які імітують роботу сотень або тисяч користувачів, і вимірювати ключові метрики:
— середній час відповіді (response time),
— кількість запитів на секунду (RPS),
— відсоток помилок,
— використання процесора, пам’яті, мережі.

Приклад у будівельній сфері: компанія створює систему онлайн-моніторингу для кількох об’єктів. Під час тестування імітують надходження тисяч запитів від IoT-сенсорів кожну хвилину, паралельно з роботою інженерів у веб-панелі. Завдяки цьому визначають, скільки подій система здатна обробити одночасно, і які сервіси потребують оптимізації або масштабування.

Масштабування — це процес збільшення обчислювальних ресурсів для підтримки більшого навантаження. Існує два основних типи масштабування:

- **Вертикальне масштабування (Vertical Scaling)** — збільшення ресурсів одного вузла (додавання CPU, оперативної пам’яті, швидшого диска). Це простий, але обмежений підхід: ресурси окремого сервера не безкінечні.
- **Горизонтальне масштабування (Horizontal Scaling)** — додавання нових вузлів або копій сервісів. Це основний підхід у мікросервісній архітектурі. Kubernetes або Docker Swarm дозволяють автоматично створювати нові екземпляри сервісу, розподіляючи навантаження між ними через балансувальник.

Приклад: у системі аналітики будівельних проєктів сервіс “Звіти” стає “вузьким місцем” під час пікових годин. Замість купівлі потужнішого сервера (вертикальне масштабування) Kubernetes створює три додаткові копії цього сервісу (горизонтальне масштабування). Балансувальник трафіку рівномірно розподіляє запити між копіями, і затримка відповіді зменшується у кілька разів.

Ще один важливий аспект — **автоматичне масштабування (autoscaling)**. Сучасні системи можуть самостійно збільшувати кількість екземплярів сервісу при зростанні навантаження і зменшувати при простої.

Для цього використовуються метрики з Prometheus або Kubernetes Horizontal Pod Autoscaler (HPA). Наприклад, якщо CPU використовується понад 80% протягом хвилини, система додає новий Pod.

Окрім тестів продуктивності, важливим є **тестування відмовостійкості (fault tolerance testing)**. У цьому випадку перевіряють, як система реагує на збої: втрату мережі, відключення бази даних, перевищення таймаутів або падіння одного з сервісів. У розподіленій архітектурі це робиться за допомогою retry-політик, circuit breaker-патернів або резервного зберігання даних (replication, failover).

У системах, що працюють у будівельній галузі, тестування і масштабування мають практичне значення. Наприклад, коли компанія відкриває новий об’єкт або підключає десятки нових сенсорів, система повинна безперебійно приймати дані і не втрачати повідомлення. Завдяки регулярному навантажувальному тестуванню можна заздалегідь передбачити, коли знадобиться збільшення ресурсів або зміна конфігурації.

Результати тестів зазвичай документуються у вигляді **звіту з продуктивності**, де зазначається:
— середній час відповіді при різних рівнях навантаження;
— максимальна кількість одночасних з’єднань;
— поведінка системи при відмовах;
— використання ресурсів;
— рекомендації щодо оптимізації.

Отже, тестування і масштабування є завершальним етапом циклу створення розподіленої системи. Вони не лише підтверджують її працездатність, а й забезпечують основу для подальшого розвитку — адже система, яка пройшла тестування під навантаженням і здатна масштабуватись, готова до промислової експлуатації. Для галузей, де надійність і безперервність процесів критичні (як у будівництві), ці етапи мають першочергове значення.

Завдання

1. Провести навантажувальне тестування (k6, JMeter).
2. Проаналізувати результати та запропонувати оптимізацію.

3. Продемонструвати масштабування через Kubernetes (https://kubernetes.io).

Зміст звіту

1. Тема та мета роботи
2. Короткі теоретичні відомості
3. Завдання до роботи згідно варіанту
4. Протоколи розв’язання задач
5. Висновки

Контрольні запитання

1. Що таке навантажувальне тестування?
2. Які типи тестів застосовують для РПС?
3. У чому різниця між stress test і load test?
4. Які інструменти використовують для тестування продуктивності?
5. Що таке масштабування і які його види?
6. Як визначити “вузьке місце” у системі?
7. Як перевірити стійкість систем*и до відмов?*

Список літератури

1. Таненбаум, Е., Ван Стеен, М. **Розподілені системи: принципи та парадигми** / пер. з англ. – Київ : Видавнича група BHV, 2003. – 880 с.
2. Кук, Дж. **Архітектура мікросервісів. Патерни розподілених систем** / Дж. Кук. – Київ : Діалектика, 2021. – 432 с.
3. Краснов, М., Чуприна, К., Білошицький, А. **Архітектура розподілених програмних систем : навчальний посібник** / М. Краснов, К. Чуприна, А. Білошицький. – Київ : НАУ, 2022. – 156 с.
4. Марс, Д., Андерсон, Д. **Docker і Kubernetes для DevOps** / Д. Марс, Д. Андерсон. – Київ : Фактор, 2022. – 384 с.
5. Hightower, K., Burns, B., Beda, J. **Kubernetes: Up and Running**. 3rd ed. – Sebastopol : O'Reilly Media, 2023. – 350 p.
6. Newman, S. **Building Microservices**. 2nd ed. – Sebastopol : O'Reilly Media, 2021. – 520 p.

Додаткова література

7. Biloshchytskyi, A., Omirbayev, S., Mukhatayev, A., Biloshchytska, S., Toxanov, S., Faizullin, A. *The concept of the Internet of Things in the development of information and analytical systems based on the method of constructing a scalar assessment of the results of research activities of scientists.* // *Procedia Computer Science.* – 2024. – Vol. 231. – P. 684–690. DOI: https://doi.org/10.1016/j.procs.2023.12.161
8. Chupryna, K., Ivakhnenko, I., Biloshchytska, S., Mykhaylo, C., Ryzhakov, D., Sobol, D. *Formalized Management of Changes at the Enterprise by Means of Fuzzy Logic.* // *IEEE International Conference on Smart Information Systems and Technologies (SIST)*. – Astana, Kazakhstan, 2023. – P. 490–494. DOI: https://doi.org/10.1109/SIST58284.2023.10223567
9. Fowler, M. **Patterns of Enterprise Application Architecture**. – Boston : Addison-Wesley, 2020. – 560 p.
10. Richards, M., Ford, N. **Fundamentals of Software Architecture: An Engineering Approach**. – Sebastopol : O'Reilly Media, 2020. – 432 p.

Електронні ресурси

11. Kubernetes Documentation [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs/
12. Docker Documentation [Електронний ресурс]. – Режим доступу: https://docs.docker.com/
13. Postman API Platform [Електронний ресурс]. – Режим доступу: https://www.postman.com/
14. Swagger UI [Електронний ресурс]. – Режим доступу: https://swagger.io/tools/swagger-ui/
15. GitHub Kubernetes Examples [Електронний ресурс]. – Режим доступу: https://github.com/kubernetes/examples