

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»

Ф.А. Смілянець, О.А. Зарічковий

КОМПОНЕНТИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ — АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний
посібник для здобувачів ступеня бакалавра за спеціальностями F2 Інженерія
програмного забезпечення

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2026

Автори: *Смілянець Федір Андрійович, PhD, асистент кафедри інформатики та програмної інженерії, факультет інформатики та обчислювальної техніки*
Зарічковий Олександр Анатолійович, PhD, асистент кафедри інформатики та програмної інженерії, факультет інформатики та обчислювальної техніки

Рецензент: *Попенко Володимир Дмитрович, к.т.н., доцент, доцент кафедри інформаційних систем та технологій, факультет інформатики та обчислювальної техніки*

Відповідальний редактор: *Стеценко Інна Вячеславівна, д.т.н., професор, професор кафедри інформатики та програмної інженерії, факультет інформатики та обчислювальної техніки*

*Гриф надано Методичною радою КПП ім. Ігоря Сікорського
(протокол No 6 від 03.04.2026)
за поданням вченої ради Факультету інформатики та обчислювальної техніки
(протокол No 8 від 30.03.2026)*

Компоненти програмної інженерії — архітектура програмного забезпечення. [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інженерія програмного забезпечення інформаційних систем» спец. F2 Інженерія програмного забезпечення / Ф.А. Смілянець, О.А. Зарічковий ; КПП ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПП ім. Ігоря Сікорського, 2026. – 113 с.

У посібнику систематизовано викладено матеріали з дисципліни «Компоненти програмної інженерії. Частина 3. Архітектура програмного забезпечення». Посібник містить матеріали щодо лекцій, спрямованих на надбання студентами знань студентів з проектування та документування архітектури програмного забезпечення. Навчальний посібник призначений для здобувачів ступеня бакалавра зі спеціальності F2 – «Інженерія програмного забезпечення», буде також корисним для студентів інших технічних спеціальностей.

Реєстр. № НП 25/26-298. Обсяг 5,7 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
<https://kpi.ua>

Свідцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Вступ	3
1 Документування архітектури	4
1.1 Документування вимог	5
1.2 Архітектурне представлення	33
1.3 Дизайн-документ	36
2 Тактики та патерни	39
2.1 Архітектурні тактики	39
2.2 Архітектурні патерни	42
3 Архітектурні стилі	45
3.1 Монолітна архітектура	46
3.2 Сервіс-орієнтована архітектура	48
3.3 Мікросервісна архітектура	52
3.4 Подійно-орієнтована архітектура	56
3.5 Безсерверна архітектура	60
3.6 Простір-орієнтована архітектура	62
4 Реалізація архітектури	65
4.1 Масштабування	65
4.2 Відмовостійкість	68
4.3 Безпека	71
4.4 Випуск програмного забезпечення	86
4.5 Сховища даних	97
4.6 Кластерні системи. Теорема CAP.	105

Вступ

Архітектура програмного забезпечення є одним з фундаментальних понять у дисципліні розробки складних програмних систем. Вона визначає структуру системи, її ключові елементи, способи взаємодії між ними та властивості, які система повинна підтримувати. Іншими словами, архітектура є сукупністю рішень, що дозволяють розробникам та іншим зацікавленим сторонам міркувати про систему як про цілісність.

Під архітектурою зазвичай розуміють набір структур, що включає модулі, компоненти та механізми їхньої взаємодії. При цьому поведінка кожного елемента розглядається настільки, наскільки вона важлива для розуміння того, як система функціонує у цілому. Таким чином, архітектура охоплює як статичні аспекти (структуру та організацію), так і динамічні (поведінку та взаємодію).

Важливо розрізняти поняття *системної архітектури* та *архітектури програмного забезпечення*. Якщо перша охоплює повний контекст системи — включно з апаратними компонентами, програмними модулями та людською взаємодією — то друга зосереджується саме на програмних елементах, їхніх відносинах і характеристиках. Системна архітектура створює мапу відповідності між функціональністю, програмними компонентами і апаратним середовищем, тоді як програмна архітектура відповідає за структурування та інтеграцію програмних рішень.

Архітектура програмного забезпечення також включає набір рішень, що стосуються архітектурних характеристик, або так званих “якостей системи” — масштабованості, надійності, безпеки, доступності та інших властивостей, які визначають життєздатність і ефективність продукту. Ці характеристики формуються через свідомий вибір архітектурних стилів, патернів, принципів та технологічних рішень.

Третьою частиною архітектури програмного забезпечення є принципи проектування, що є керівними правилами децентралізованого прийняття менш впливових архітектурних рішень: вимоги до інтерфейсів між компонентами, конвенції найменування, підходи до обробки помилок тощо. Такі принципи зберігають узгодженість та передбачуваність поведінки системи при розробці.

Архітектура не є чимось статичним — вона формується і розвивається протягом усього життєвого циклу програмного продукту. На початкових етапах вона постає у вигляді моделей предметної області, концептуальних діаграм і планів реалізації. Далі, у процесі розробки, архітектура уточнюється, деталізується і тестується. Наприклад, у великих проектах архітектура визначає не тільки структуру коду, а й організацію командної роботи, вибір технологій та стратегій інтеграції.

Архітектура — це одночасно і результат (набір моделей, діаграм і описів), і процес (постійна діяльність із прийняття рішень, аналізу та вдосконалення).

Розділ 1

Документування архітектури

Документування архітектури є ключовим для забезпечення розуміння, узгодження та послідовності у розробці програмного забезпечення. Фіксування вимог та прийнятих на їх основі архітектурних рішень у чітко визначеному наборі документів дозволяє уникнути:

- ізолювання знань про окремі частини архітектури ПЗ між окремими командами та розробниками;
- розпорошення знань між різними джерелами, вербальною та текстовою комунікацією (“Email-driven architecture”);

що спричиняють наступні практичні проблеми:

Комунікаційні завади Різні учасники обговорення формують різні уявлення про вимоги, будову та інші аспекти програмного забезпечення, що перешкоджає ефективній дискусії при організації розробки, утворює багаторазове обговорення одних і тих самих питань, тощо.

Завади при плануванні Непрозорість ПЗ та прихованість архітектурних рішень ускладнюють визначення конкретних змін, що мусять бути втілені у ПЗ для реалізації тих чи інших нових функціональностей, виправлення вад, адаптацію ПЗ до нових вимог з швидкодії тощо.

Відсутність напрямку Нестача якісної документації призводить до прийняття хибних у різних проявах, конфліктних та іншим чином непродуманих рішень, що може призводити до каскадних змін у ПЗ, повторної переробки окремих частин тощо.

Неприйняті рішення Деякі аспекти ПЗ можуть залишатись невизначеними навіть при загальному виконанні вимог. Чим далі відкладається неприйняття рішення, тим складніше буде його прийняти і впровадити.

Необґрунтовані рішення Архітектурні рішення, що були прийняті без консультації з рештою команди чи іншими командами, не були належним чином обґрунтовані вимогами до ПЗ тощо. Такі рішення втрачають свою цінність у відсутності контексту, а також часто стають неприємним “сюрпризом” при втіленні змін у ПЗ, що не враховували існування цього рішення.

Забуті рішення Архітектурні рішення, що були обґрунтовані, обговорені та прийняті, однак не були зафіксовані документально. Без належного запису рішення забуваються в історії проєкту, і їхня цінність зменшується. Відсутність зафіксованої історії рішень також може призвести до повторних обговорень або прийняття суперечливих рішень, або стати неприємною несподіванкою аналогічно до попереднього пункту.

Натомість, фіксація рішень, їхнього контексту та обґрунтування значно скорочує усі етапи життєвого циклу програмного забезпечення — від погодження до впровадження.

Важливо зауважити, що протягом життєвого циклу ПЗ архітектурні рішення можуть прийматись і скасовуватись. Тому архітектурна документація для збереження користі має бути не одноразово виконаним документом, а живим і постійно оновлюваним.

В ідеальних обставинах, зміни в ПЗ плануються та разом з ними оновлюється документація. Однак, реальність може часто розходитися з ідеальним варіантом через лінь, поспіх чи інші причини. У такому разі, необхідним є ретроспективне документування архітектури програмного забезпечення для синхронізації стану ПЗ та інформації про нього.

1.1 Документування вимог

Якісне документування архітектури починається з *архітектурно значущих вимог* (рис 1.1).

Архітектурно значущі вимоги є підмножиною вимог до програмного забезпечення, що мають істотний вплив на його архітектуру.

Межа між вимогами, що можуть впливати на архітектуру, та архітектурно значущими вимогами є доволі розмитою. Однак, є ряд критеріїв [1], які допомагають відмежувати явно значущі вимоги від інших:

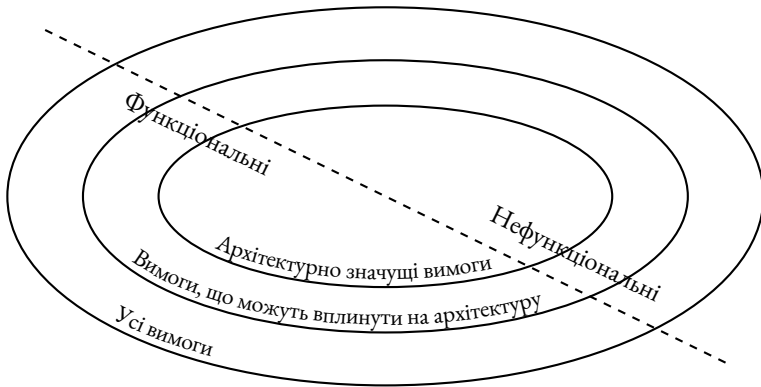


Рис. 1.1. Ієрархія вимог до ПЗ з точки зору архітектури

1. напряду асоціюється з високою бізнес-цінністю або ризиком;
2. є критичною важливою стороною — інвестора, аудитора або державного регулятора;
3. включає в себе атрибути якості надання послуг (англ. Quality-of-Service, QoS), такі як вимоги до швидкодії чи пропускну здатності, для архітектури що змінюється — такі вимоги мають бути суттєво відмінними від тих, що вже задовольняються;
4. залучає нові або включає існуючі залежності, яким може бути властива непередбачувана або ненадійна поведінка, від бібліотек до зовнішніх API чи фізичних залежностей (наприклад, доставка поштою);
5. є наскрізною, і таким чином зачіпляє декілька частин системи та їх взаємодії, може мати короткостроковий чи навіть довгостроковий вплив на усю систему (наприклад, безпекові вимоги);
6. є принципово новою для організації, команда не має досвіду реалізації підсистем чи компонент, що реалізують дану функціональність;
7. вже висувалась раніше в схожому контексті в попередніх проектах та викликала проблеми, критичні ситуації, недотримання дедлайнів, незадоволення клієнтів тощо.

Приклад документу про аналіз вимог за даними критеріями наведено на таблиці 1.1. Результати такого аналізу завжди є суб'єктивними, залежить від контексту конкретного ПЗ та є допоміжним інструментом.

Таблиця 1.1. Аналіз вимог на архітектурну значущість

Вимога	Оцінка	Обґрунтування
Відповідність GDPR	Висока	GDPR вимагає можливості доступу та повного видалення збережених про користувача даних на вимогу, високої безпеки користувацьких даних тощо. Оперування ПЗ в ЄС при недотриманні даних правил загрожує високими штрафами.
Підтримка 10 тисяч одночасних користувачів	Висока	Критичні QoS-вимоги, що радикально відрізняються від звичайного навантаження. Впливає на всю архітектуру: балансування навантаження, масштабування БД, кешування тощо.
Обчислення щомісячної статистики за даними	Висока	Вимагає масового доступу до даних та можливості їх швидкої масової обробки, що вимагає залучення особливих інструментів як на рівні баз даних, так і на рівні організації даних обчислень.
Підтримка відеодзвінків у медичній платформі	Висока	Нова функціональність для команди, наскрізна (безпека, аудіо/відео потоки, мережа), є ресурсоемною.
Автоматичне резервне копіювання кожні N годин	Помірна	Впливає на архітектуру зберігання даних, навантаження на БД, потребує планувальника завдань, але стандартні рішення існують.
Сортування, пошук та пагінація за певною сутністю	Нехтовна	Тривіальна зміна.
Зміна кольору кнопки	Нехтовна	Тривіальна зміна.

Іншою та не менш важливою площиною відмінності між різними вимогами є розділення між *функціональними* та *нефункціональними* вимогами.

Функціональні вимоги описують *що* робить ПЗ для виконання поставлених задач чи вирішення проблем. Їх метою є формалізація і фіксування очікувань кінцевого користувача. Так, наприклад, вимога до ПЗ мати пошук серед екземплярів певної сутності в базі даних є функціональною.

Нефункціональні вимоги описують *яким* є ПЗ. Також відомі як *атрибути якості*), вони описують операційні характеристики, обмеження тощо. Наприклад, вимога надавати відповідь на HTTP-запит протягом 500 мс. від здійснення користувачем запиту є нефункціональною. Нефункціональні вимоги часто впливають на архітектуру.

Функціональні вимоги визначають реакцію програмного забезпечення на ті чи інші стимули, та задовільняються розподіленням відповідальності та порядку дій між частинами ПЗ. Архітектурні рішення виникають з перетину обох типів вимог.

Якісні вимоги відрізняються за наступними критеріями [2]:

Необхідність забезпечує ефективність витрати часу розробників та фокус на вирішенні проблем. Необхідна вимога відповідає реальній потребі користувачів або вирішує конкретну задачу — без її реалізації система не зможе виконувати свою основну функцію або буде значно менш цінною. Включення непотрібних вимог призводить до марної витрати часу та коштів, збільшення складності системи, накопичення технічного боргу та відволікання від дійсно важливих функціональностей. При виникненні сумнівів у необхідності вимоги, варто скористатись контрольним питанням “І що?” (англ. “So what?”) щодо відсутності даної вимоги у списку.

Однозначність забезпечує єдине та правильне розуміння між усіма учасниками проекту незалежно від їхнього досвіду чи ролі. Неоднозначна вимога може бути інтерпретована по-різному різними людьми, що призводить до реалізації неочікуваної функціональності, конфліктів між командами, необхідності переробки коду та невідповідності кінцевого продукту очікуванням. Однозначність досягається шляхом використання конкретних, вимірюваних цілей замість розмитих (“Відповідність ДСТУ EN 301 549:2022 Вимоги щодо доступності продуктів та послуг ІКТ” замість “Зручний інтерфейс”), чітке визначення граничних умов та винятків, уникнення слів з множинними значеннями, а також структурованих форматів опису поведінки системи. Добре сформульована однозначна вимога не залишає простору для припущень — кожен, хто її читає, розуміє точно те саме, що мав на увазі автор.

Перевірюваність забезпечує можливість об'єктивно визначити, чи була вимога реалізована правильно та повністю. Непереверювані вимоги унеможливають контроль якості, приймання роботи, конфліктів між учасниками процесу розробки тощо. Переверюваність досягається через формулювання конкретних, вимірюваних критеріїв успіху (наприклад, “відповідь надходить протягом 200 мс” замість “система працює швидко”), визначення точних умов виконання та очікуваних результатів, специфікацію форматів вхідних і вихідних даних, а також опис поведінки у граничних та виняткових ситуаціях. Добре сформульована перевірювана вимога дозволяє однозначно підтвердити або спростувати факт її реалізації без необхідності суб'єктивної інтерпретації.

Досяжність забезпечує реалістичність реалізації вимоги в межах наявних ресурсів, технологій та обмежень проекту. Недосяжна вимога призводить до демотивації команди, провалу дедлайнів, перевитрати бюджету, або навіть повної неможливості завершення проекту, що в результаті може спричинити скасування всієї розробки. Досяжність задачі з інженерної точки зору може бути оцінена проведенням окремого дослідження, однак навіть технічно можлива вимога може бути недосяжною з точки зору доступних ресурсів, часових рамок та інших обмежень.

1.1.1 Функціональні вимоги

Документування функціональних вимог є ключовим не тільки для прийняття та обґрунтування правильних архітектурних рішень, але і для процесу розробки програмного забезпечення загалом. Цілісно описані функціональні вимоги є фундаментальними для:

- побудови співрозуміння між учасниками процесу;
- уникнення випадків необхідності виконання однієї роботи двічі;
- контролю розростання обсягів проекту;
- передачі знань, у т.ч. адаптації нових членів команди;
- контролю якості тощо.

Функціональні вимоги можуть бути описані як у текстовому вигляді: вільному, тест-специфікаціях, користувацьких історіях, варіантах використання, EARS тощо, так і візуально: блок-схемах, діаграмах потоків даних, діяльності, станах, послідовностях, варіантах використання, прототипах екранних форм (як каркасах так і завершених дизайнах) тощо.

Типовим вільним записом функціональних вимог є “X має Y”:

Приклад**ФВ-001: Відображення оцінок**

Сторінка предмету, відкрита студентом, має відображати оцінки студента з даного предмету.

Продуктовий контекст є важливим для розуміння контексту вимоги, тому написання функціональних вимог часто починається з користувацької історії.

Користувацька історія — напівформальний опис бажаного мовою предметної області, що виконується від імені кінцевого користувача або іншого учасника використання програмного забезпечення та пояснює *нащо* тієї чи іншої частини ПЗ. Типовим форматом користувацьких історій є шаблон Connextra “Як *<роль>*, я хочу *<виконати дію>* для *<вигоди>*”.

Приклад**ФВ-001: Відображення оцінок**

Користувацька історія: Як студент, я хочу знати свої оцінки, для того щоб бути впевненим у власному прогресі.

Користувацькі історії використовуються як на початкових етапах розробки для запису користувацьких потреб та зацікавленостей, так і для формування вимог та опису можливостей ПЗ. Важливо звернути увагу на відмінність між вищенаведеним прикладом та користувацькими історіями вигляду “Як студент, я хочу мати сторінку поточного оцінювання, для того щоб знати свої оцінки” та іншими схожими формами [3], [4]. Перші фокусуються на описі проблеми, у той час як другі звертають увагу на конкретне рішення проблеми. Використання проблемо-орієнтованої форми залишає спосіб досягнення цілі і отримання вигоди відкритим для подальшої інтерпретації, що дозволяє знаходити простіші та примітивніші способи задоволення вимоги, що дозволяє зекономити час при першому підході до реалізації та ітеративно досягати фінального бачення ПЗ. Використання ж вужчої, рішення-орієнтованої форми може призвести до передчасного прийняття рішення, витрати часу на реалізацію ще не потрібного функціоналу, прив’язки до субоптимальних рішень.

Після фіналізації користувацька історія доповнюється *критеріями прийняття*.

Критерії прийняття — список умов, які ПЗ має виконувати, аби та чи інша користувацька історія була здійсненою, або задача чи проблема — вирішеною. Даними умовами вимірюється прогрес розробки та забезпечується відповідність реалізованих функціональностей потребам користувача.

Якісні критерії прийняття є об'єктивними, перевірюваними, написаними простою мовою та сфокусованими на бажаному результаті. Дотримання даних принципів дозволяє уникнути непорозумінь між учасниками процесу, забезпечує чітке розуміння того, коли функціональність можна вважати завершеною та що є необхідним для цього, та створює основу для ефективного тестування.

Критерії прийняття можуть бути описані як у вільній формі (X має Y), так і у відповідності поведінковим діаграмам чи прототипам екранних форм, так і в більш формальних виглядах, наприклад Дано-Коли-Тоді (англ. Given-When-Then) [5]: “ДАНО <передумова/стан/вхідні дані> КОЛИ <дія> ТОДІ <результат>”. У критерії також може бути декілька передумов, послідовних дій або результатів, які можуть бути поєднані логічними сполучниками ТА чи АБО. Одній користувацькій історії часто відповідає декілька критеріїв прийняття:

Приклад

ФВ-002: Завантаження лабораторних робіт

Користувацька історія: Як студент, я хочу здати виконані лабораторні роботи, для того щоб підтвердити прогрес у предметі.

Критерії прийняття:

1. ДАНО студент увійшов у систему ТА відкрито сторінку предмета, що вивчається студентом КОЛИ студент натискає кнопку ‘завантажити лабораторну роботу’ ТОДІ має відкритися модальне вікно, що дозволить обрати лабораторну роботу та прикріпити файл.
2. ДАНО студент увійшов у систему ТА відкрито сторінку предмета, що вивчається студентом ТА студентом вже завантажено певну лабораторну роботу КОЛИ викладач виставив оцінку за цю лабораторну роботу ТОДІ студенту має бути заблоковано внесення змін в здану лабораторну роботу.

Іншим способом формалізувати критерії прийняття є шаблон EARS (Easy Approach to Requirement Syntax, Простий Підхід до Синтаксису Вимог). EARS пропонує п'ять шаблонів (табл. 1.2).

Таблиця 1.2. Шаблони EARS

Назва	Шаблон
Типовий	<частина ПЗ> має <відгук>
Подійно-керований	КОЛИ <тригер> <опціональна передумова>, <частина ПЗ> має <відгук>
Небажана поведінка	ЯКЩО <небажаний стан або подія>, тоді <частина ПЗ> має <відгук>
Стано-керований	ПОКИ <стан>, <частина ПЗ> має <відгук>
Опціональна можливість	ДЕ <можливість увімкнена>, <частина ПЗ> має <відгук>
Складні	Комбінація вищезазначених шаблонів за допомогою логічних сполучників ТА і АБО

Використання EARS замість вільної форми написання вимог чи критеріїв прийняття дозволяє уникнути типових помилок, що призводять до хибної інтерпретації вимог, але при цьому є менш формалізованим, ніж Дано-Коли-Тоді чи варіанти використання. Розглянемо EARS на прикладі вимоги ФВ-002:

Приклад

ФВ-002: Завантаження лабораторних робіт

Користувацька історія: Як студент, я хочу здати виконані лабораторні роботи, щоб підтвердити прогрес у предметі.

Критерії прийняття:

1. Кнопка 'завантажити лабораторну роботу' на сторінці предмета має відкривати модальне вікно, що дозволить обрати лабораторну роботу та прикріпити файл
2. КОЛИ настав дедлайн здачі, модальне вікно має заблокувати завантаження лабораторної роботи
3. ЯКЩО студентом завантажено лабораторну роботу у хибному форматі, модальне вікно має відобразити повідомлення про помилку

Приклад (продовження)**ФВ-002: Завантаження лабораторних робіт**

Користувацька історія: Як студент, я хочу здати виконані лабораторні роботи, щоб підтвердити прогрес у предметі.

Критерії прийняття:

4. ПОКИ лабораторну роботу не перевірено викладачем, модальне вікно має надавати студенту можливість замінити файл лабораторної роботи
5. ДЕ увімкнено версіонування лабораторних робіт, система має зберігати усі завантажені студентом файли та дати їх надання
6. КОЛИ викладач повертає лабораторну роботу на доопрацювання ПОКИ не настав дедлайн здачі, модальне вікно має надавати студенту можливість замінити файл лабораторної роботи

Додатковим способом формалізувати вимоги та доповнити користувацькі історії деталізованою інформацією про бажану поведінку ПЗ є *варіанти використання*:

Варіанти використання є сукупністю усіх шляхів досягнення певної мети певним діячем (актором) засобами певної системи [6], [7], включаючи як успішні сценарії, так і сценарії з ускладненнями чи такі, що закінчуються невдачею.

Діяч (актор, дійова особа) представляє будь-кого чи будь-що наділене власною поведінкою у взаємодії з системою, що розглядається. Це може бути як людина, так і організація, інституція чи зовнішнє до системи програмне забезпечення. Діяч описує певну роль, що виконується у взаємодії з ПЗ.

Первинний діяч ініціює варіант використання для досягнення власної мети.

Підтримуючий діяч є тим, до якого система звертається у процесі виконання варіанта використання.

Базовий сценарій — очікуваний шлях досягнення мети, також відомий як “основний сценарій успіху” або “основний потік”.

Розширення — альтернативні шляхи, особливі випадки, опціональні кроки та помилки, які потребують обробки. Також відомі як “альтернативні потоки”.

Варіант використання складається з базового сценарію та множини розширень. Базовий сценарій описується як послідовність кроків, кожен з яких включає взаємодію між системою та одним або кількома діячами. Розширення визначають

точки, де виконання може відхилитися від базового сценарію, та може включати послідовність кроків у даному випадку. Нумерація розширень відображає крок базового сценарію, від якого це розширення відштовхується. Додатково, варіант використання може бути доповнений передумовами, обсягом (частина системи, якої стосується) тощо.

Таблиця 1.3. Приклад варіанту використання

Первинний діяч: Студент	Здача виконаної лабораторної роботи	
Базовий сценарій: 1. Студент автентифікується у системі; 2. Студент переходить на сторінку предмету; 3. Система відображає список доступних лабораторних робіт; 4. Студент обирає лабораторну роботу для задачі; 5. Система відкриває форму завантаження; 6. Студент прикріплює файл з виконаною роботою; 7. Студент підтверджує задачу; 8. Система зберігає файл та оновлює статус роботи; 9. Система відображає підтвердження успішної задачі.		Розширення: 4а. Лабораторна робота недоступна для задачі; 4а1. Система пояснює недоступність ЛР для задачі, наприклад дедлайн. 6а. Файл має неприпустимий формат; 4а1. Система дає можливість обрати інший файл та повідомляє про підтримувані формати. 6б. Розмір файлу перевищує дозволений; 8а. Помилка при збереженні файлу.

Для отримання загального огляду функціональних вимог до ПЗ, сукупність варіантів використання може бути зображена на діаграмі варіантів використання.

Діаграма варіантів використання є UML-діаграмою, що зображає відношення між діячами та варіантами використання ПЗ, що їм доступні (рис. 1.2). Діаграма надає високорівневий огляд можливостей програмного забезпечення з точки зору зовнішніх користувачів та систем, абстрагуючись від деталей реалізації.

Основними елементами діаграми є:

Діячі (актори) — зображуються як фігурки людей або прямокутники зі стереотипом «астог», представляють зовнішні сутності, що взаємодіють із системою;

- Варіанти використання** — зображуються як овали з назвою всередині, представляють функціональності, які система надає діячам;
- Межі системи** — прямокутник, що окреслює варіанти використання та відділяє внутрішні елементи системи від зовнішніх діячів;
- Асоціації** — лінії між діячами та варіантами використання, що показують участь діяча у варіанті використання;
- Відношення включення («include»)** — пунктирна стрілка з підписом, що вказує на обов'язкове включення одного варіанту використання в інший;
- Відношення розширення («extend»)** — пунктирна стрілка з підписом, що вказує на опціональне розширення базового варіанту використання;
- Узагальнення** — суцільна стрілка з порожнім трикутником, що показує спеціалізацію діячів або варіантів використання.

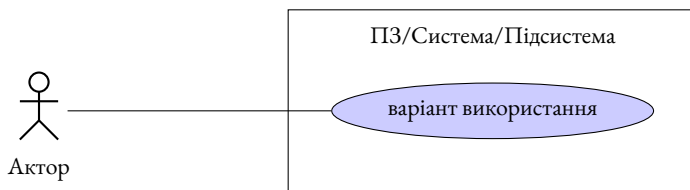


Рис. 1.2. Базові елементи діаграми варіантів використання

При побудові діаграми важливо дотримуватись балансу між повнотою та читабельністю — надмірна деталізація або велика кількість елементів може зробити діаграму важкою для сприйняття. Для складних систем рекомендується створювати кілька діаграм різного рівня деталізації або розділяти систему на підсистеми з окремими діаграмами.

Розглянемо складніший приклад (рис. 1.3).

На діаграмі представлено чотири типи діячів:

- Студент** — первинний діяч, що використовує систему для перегляду оцінок, здачі лабораторних робіт та доступу до навчальних матеріалів;
- Викладач** — первинний діяч, відповідальний за оцінювання робіт, управління навчальними матеріалами та встановлення часових обмежень;
- Гість** — базовий тип користувача з обмеженим доступом, від якого наслідуються студент;
- Система сповіщень** — підтримуючий діяч (зовнішня система), що забезпечує розсилку повідомлень.

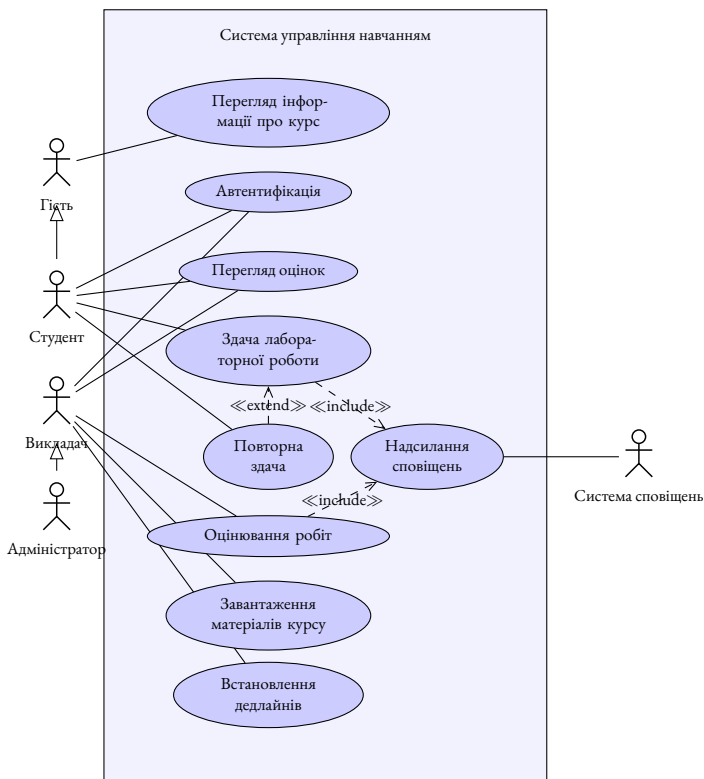


Рис. 1.3. Діаграма варіантів використання системи управління навчанням

Діаграма демонструє три типи відношень між варіантами використання:

Відношення включення («include») використовується для виділення спільної поведінки: “Оцінювання робіт” завжди включає “Надсилання сповіщень” — забезпечує автоматичне інформування студентів про виставлені оцінки.

Відношення розширення («extend») моделює опціональну поведінку: “Повторна задача роботи” розширює базовий варіант “Задача лабораторної роботи” і активується лише за певних умов (наприклад, коли викладач повертає роботу на доопрацювання, як описано в критеріях прийняття ФВ-002).

Узагальнення показує ієрархію діячів: Студент наслідує можливості гостя (доступ до публічних матеріалів) та додає власні функціональності.

Варіант використання “Автентифікація” є спільним для студентів та викладачів, що відповідає вимозі ФВ-003 про багатоканальну автентифікацію. Діаграма не

деталізує різні методи входу (email, телефон, соціальні мережі), вони описуються в текстових специфікаціях обраної форми.

Важливо зауважити, що діаграми мають давати узагальнений погляд — варто уникати перевантаження інформацією. Детальніші аспекти, такі як версіонування лабораторних робіт, різні типи оцінок або складні умови доступу, краще документувати в текстових описах варіантів використання та інших текстових формах опису функціональних вимог або на окремих, більш деталізованих діаграмах підсистем. Складні системи з великої кількості компонентів та можливих дій варто розбивати на декілька діаграм, кожна з яких описує окремі модулі.

Завершальним етапом деталізації функціональних вимог для ПЗ з інтерфейсом користувача є побудова прототипів екранних форм у вигляді каркасів, макетів чи інтерактивних прототипів.

Каркас (англ. wireframe) є низькодеталізованим структурним ескізом користувацького інтерфейсу, що показує розташування основних елементів, їх ієрархію та базову функціональність без деталей дизайну (кольори, шрифти, зображення).

Макет (англ. mockup) є статичним візуальним представленням інтерфейсу з деталізованим дизайном, кольорами, типографікою та стилізацією, але без інтерактивності. Макети наповнюються штучними даними для реалістичного представлення інтерфейсу.

Інтерактивний прототип є функціональною моделлю інтерфейсу, симулює взаємодію користувача з системою через переходи між макетами, анімації та відгуки на дії користувача.

Кожен тип прототипу відповідає різним етапам уточнення вимог. Каркасні моделі використовуються на початкових етапах для швидкої перевірки концепцій та структури інформації. Макети допомагають узгодити візуальні рішення та забезпечити відповідність бренду чи стилістичним вимогам. Інтерактивні прототипи дозволяють перевірити логіку взаємодії та виявити проблеми користувацького досвіду до початку розробки.

Прототипи не замінюють текстову документацію вимог, а доповнюють її, надаючи візуальний контекст та полегшуючи комунікацію між учасниками процесу розробки. Їх використання дозволяє виявити неточності та прогалини у вимогах на ранніх етапах, коли вартість змін є мінімальною.

На етапі проєктування архітектури програмного забезпечення каркаси інтерфейсу допомагають візуально контекстуалізувати вимоги при прийнятті рішень, та є достатніми.

1.1.2 Нефункціональні вимоги

Нефункціональні вимоги формують архітектуру програмного забезпечення через накладення наскрізних обмежень можливих рішень, що визначаються необхідною якістю виконання функціональних вимог. Так, вимога до високої доступності призводить до потреби у наявності “запасних” компонентів системи, швидкого відновлення після критичних помилок, тощо, а вимога до обслуговування десятків тисяч одночасних користувачів на декількох континентах – розподіленої архітектури з балансуванням навантаження, наявності у декількох датацентрах та синхронізації даних між ними.

Нефункціональні вимоги описуються *атрибутами якості*.

Атрибут якості (англ. quality attribute) — вимірювана або перевірювана властивість системи, що характеризує ступінь, до якого система задовольняє потреби та очікування зацікавлених сторін щодо її поведінки та характеристик. Атрибути якості описують операційні та еволюційні властивості системи, та можуть конфліктувати між собою (наприклад, підвищення безпеки може знизити час відгуку системи), що вимагає архітектурних компромісів.

Серед атрибутів якості програмного забезпечення виділимо наступні:

Продуктивність характеризує здатність ПЗ виконувати свої функції в межах визначених часових обмежень при заданому навантаженні. Включає метрики часу відгуку (латентність між стимулом та реакцією), пропускну здатності (кількість транзакцій за одиницю часу) та використання ресурсів (процесор, пам’ять, мережа). Типовий сценарій: “ПЗ повинно обробляти 2000 транзакцій на хвилину при середньому часі відгуку у 500мс”.

Безпека характеризує здатність ПЗ захищати дані та інформацію від неавторизованого доступу при забезпеченні доступу авторизованим користувачам. Може включати конфіденційність (захист від несанкціонованого розкриття), цілісність (захист від несанкціонованої модифікації), доступність (захист від атак типу denial-of-service), автентичність (підтвердження ідентичності учасників транзакції), невідмовність (неможливість заперечення виконаних дій) та підзвітність (можливість простежити дії до конкретного суб’єкта).

Модифікованість характеризує вартість внесення змін у ПЗ. Вимірюється витратами часу, зусиль та ресурсів на реалізацію, тестування та розгортання змін. Включає розширюваність (додавання нових функцій), конфігуруваність (зміна поведінки через параметри), адаптованість (пристосування до нових середовищ) та корегованість (виправлення дефектів). Типовий сценарій: “додавання нового типу платіжного методу має займати не більше

40 людино-годин (включаючи розробку і тестування) при відсутності не передбачених побічних ефектів (регресійних дефектів тощо)”.

Доступність характеризує ступінь операційної готовності ПЗ до виконання своїх функцій у довільний момент часу. Вимірюється як відношення часу справної роботи до загального часу (наприклад, 99.9% означає не більше 8 годин 46 хвилин простою на рік). Визначається через середній час між відмовами (MTBF) та середній час відновлення (MTTR) за формулою:

$$\text{Доступність} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}.$$

Надійність характеризує здатність ПЗ продовжувати коректну роботу протягом визначеного часу в заданих умовах експлуатації. Є атрибутом, спорідненим з доступністю. Вимірюється через середній час між відмовами (MTBF), імовірність відмови при запиті (PFD), середній час до відновлення працездатності (MTTR). Включає відмовостійкість (здатність функціонувати при наявності помилок), відновлюваність (здатність повернутися до робочого стану) та передбачуваність поведінки.

Зручність використання характеризує легкість використання ПЗ для досягнення конкретних цілей з ефективністю, результативністю та задоволенням у визначеному контексті використання. Включає здатність до вивчення (швидкість опанування), ефективність використання (продуктивність досвідчених користувачів), запам'ятовуваність (легкість відновлення навичок), стійкість до помилок (здатність до мінімізації кількості та серйозності помилок користувача) та суб'єктивне задоволення. На перший погляд, зручність використання здається неважливим атрибутом для архітектури ПЗ, однак, наприклад, нефункціональні вимоги щодо прозорості прогресу дії чи можливості скасувати дію з гарантованим поверненням до попереднього стану можуть набувати архітектурної значущості.

Тестованість характеризує легкість виявлення дефектів або перевірки відповідності ПЗ специфікаціям. Визначається можливістю контролювати вхідні дані та стан системи (керуваність), спостерігати вихідні дані та зміни стану (спостережуваність), ізолювати компоненти для незалежного тестування та автоматизувати тестові процедури. Часто вимірюється у кількості гілок коду, що перевіряються автоматичними тестами.

Портативність характеризує придатність ПЗ до перенесення з одного середовища в інше. Середовище може включати апаратне забезпечення (різні процесорні архітектури, обсяги пам'яті), програмне забезпечення (операційні системи, версії рантайм-середовищ, бібліотеки), організаційне оточення (різні компанії, процеси розробки) або їх комбінації. Для програмного забезпечення, що використовує хмарні ресурси — також здатність до міграції

до іншого постачальника хмарних послуг або на локальні обчислювальні потужності.

Супроводжуваність характеризує легкість підтримки ПЗ в робочому стані та її еволюції відповідно до вимог, що змінюються. Включає в себе модифікованість, портативність та тестованість, доповнені аналізованістю (складність розуміння системи у роботі, пошуку та діагностики проблем) та стабільністю (відсутність неочікуваних ефектів від змін).

Масштабованість характеризує здатність ПЗ ефективно адаптуватися до збільшення навантаження через додавання нових ресурсів (в автоматичному або ручному режимі), також — здатність ПЗ використати додані ресурси для адаптації до підвищеного навантаження. Масштабованість визначає, наскільки добре ПЗ може впоратися зі зростанням кількості користувачів, обсягів даних, інтенсивності транзакцій або інших параметрів навантаження без деградації продуктивності або функціональності. Також включає в себе географічну масштабованість — здатність ПЗ до функціонування при розподіленні користувачів та компонентів ПЗ між різними локаціями при збереженні продуктивності, надійності, доступності та інших атрибутів якості.

Окрім вищезгаданих, також можуть використовуватись налаштовуваність (здатність ПЗ до конфігурації під конкретне оточення чи вимоги), здатність до інтернаціоналізації (можливості потенційного використання у іншому культурному та мовному просторі) та локалізації (адаптації до використання у конкретній країні конкретною мовою), перевикористовуваність (здатність окремих частин ПЗ до використання для вирішення початково незапланованих задач), інтероперабельність (здатність до обміну інформацією та взаємодії з іншими ПЗ, системами, обладнанням тощо) та інші атрибути якості.

Унікальні атрибути якості можуть виникати з особливостей предметної області, для якої розробляється програмне забезпечення (наприклад, аудитованість для банківського чи фінансового ПЗ — придатність до перевірки процесів, дій, рішень тощо для підтвердження коректності реалізованих процесів, даних тощо). Своєчасна ідентифікація специфічних предметній області атрибутів якості є важливою при плануванні архітектури ПЗ, оскільки їм притаманний надмірний вплив на архітектурні рішення.

Самі по собі атрибути якості є абстрактними концепціями — твердження “система має бути продуктивною” або “система має бути безпечною” не надає достатньої інформації для прийняття архітектурних рішень чи перевірки їх виконання.

Нефункціональні вимоги можуть бути конкретизовані як у вільній формі (як приклади у визначеннях атрибутів якості вище), так і у вигляді сценаріїв атрибуту якості.

Сценарій атрибуту якості — конкретний, вимірюваний опис того, як система повинна поводитися в певній ситуації для задоволення нефункціональної вимоги. Сценарій перетворює абстрактний атрибут якості на перевірюване твердження, що може бути використане для оцінки архітектури та приймання системи.

Сценарій атрибуту якості складається з наступних частин (рис. 1.4):

Джерело стимулу є сутністю (людина, система, або будь-що інше), що генерує стимул;

Стимул є умовою або подією, що має бути оброблена системою;

Артефакт є частиною системи, що отримує стимул;

Середовище є умовою, за якої відбувається стимул (нормальна робота, пікове навантаження, відмова компонента);

Відгук є реакцією системи у відповідь на стимул;

Міра відгуку є кількісною або якісною характеристикою, що дозволяє перевірити задовільність відгуку.

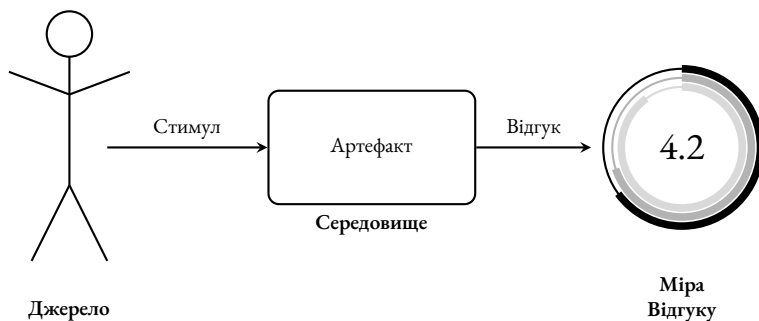


Рис. 1.4. Сценарій атрибуту якості

Розглянемо базовий приклад. Загальний атрибут портативності може використовуватись як “система має працювати в різних середовищах”, однак для того щоб на основі даної вимоги можна було приймати архітектурне рішення, даний атрибут має бути конкретизованим. Для цього, необхідно ідентифікувати реальні ситуації використання, що виникають у цільовій предметній області. Наприклад, в освітній

галузі типовою є ситуація, коли навчальні заклади мають різні інфраструктурні обмеження — від повністю хмарних рішень до суворо локальних розгортань через існуючі інвестиції в інфраструктуру (табл. 1.4)

Таблиця 1.4. Приклад сценарію атрибуту якості: Портативність

Елемент	Опис
Джерело	Університет-замовник
Стимул	Вимога розгорнути систему управління навчанням на власній інфраструктурі університету (наприклад, on-premise VMware vSphere) замість рекомендованої постачальником конфігурації (AWS EKS та S3)
Артефакт	Система управління навчанням як ціле
Середовище	Початкове розгортання у новому університеті-клієнті перед початком навчального року
Відгук	Система адаптується до локальної інфраструктури через конфігурацію, підтримує розгортання на VMware Tanzu чи іншому локальному варіанті Kubernetes
Міра відгуку	Розгортання виконується за 80 людино-годин силами ІТ-відділу університету з мінімальною підтримкою постачальника; 100% функціональності доступні незалежно від обраної платформи; документація покриває усі підтримувані конфігурації

З архітектурної перспективи, даний сценарій вимагає впровадження шарів абстракції на декількох рівнях: абстрагування від платформи-специфічних сервісів зберігання, незалежність від конкретної реалізації оркестрації контейнерів (EKS проти Tanzu), уніфікація механізмів моніторингу та логування. Кількісна міра в 80 людино-годин накладає жорстке обмеження на складність процесу адаптації, виключаючи архітектурні рішення, що потребують суттєвої модифікації коду або архітектури для кожного варіанту розгортання. Вимога функціональної еквівалентності (100% доступності функцій) унеможливорює спрощений підхід до портованості через обмеження функціональності залежно від платформи.

Доступність також може набувати менш очікуваних нюансів під час конкретизації (табл. 1.5).

Таблиця 1.5. Приклад сценарію атрибуту якості: Доступність

Елемент	Опис
Джерело	Приймальна комісія
Стимул	Масовий імпорт інформації про студентів першого курсу
Артефакт	Декілька сервісів, база даних
Середовище	Нормальне навантаження на систему
Відгук	Система поступово обробляє надані дані, створює облікові записи студентів, інші сутності, встановлює зв'язки з існуючими сутностями, розсилає сповіщення тощо.
Міра відгуку	Імпорт першого курсу у 2000 осіб відбувається менш ніж за 10 хвилин, обслуговування інших запитів не припиняється та відповідає вимогам продуктивності.

Дана вимога включає в себе доступність не тільки відносно відмов, що спричинені недоступністю системи, а й щодо таких, що спричинені раптовим та короточасним ростом навантаження — пакетної обробки даних — що впливає на архітектуру через виключення синхронних підходів до обробки інформації, оскільки містить необхідність продовження нормального обслуговування запитів. Тому, для виконання даної вимоги, ПЗ має передбачати черги та асинхронну обробку даних без блокування обробки інших запитів.

Розглянемо цю ж ситуацію з точки зору зручності у використанні (табл. 1.6).

Таблиця 1.6. Приклад сценарію атрибуту якості: Зручність у використанні

Елемент	Опис
Джерело	Приймальна комісія
Стимул	Масовий імпорт інформації про студентів першого курсу
Артефакт	Декілька сервісів, база даних
Середовище	Нормальне навантаження на систему
Відгук	Система поступово обробляє надані дані, створює облікові записи студентів, інші сутності, встановлює зв'язки з існуючими сутностями, розсилає сповіщення тощо.
Продовження на наступній сторінці	

Таблиця 1.6 – Продовження

Елемент	Опис
Міра відгуку	Система має надавати підтвердження коректного прочитання імпортованих даних з прикладом перед початком процедури імпорту, прозоро повідомляти користувачу про прогрес, надати можливість зупинити транзакцію з повним відкатом змін.

Вимоги зручності використання можуть мати найбільш непередбачувані архітектурні наслідки. Так, ця вимога впливає на архітектурні рішення щодо баз даних через необхідність у:

- проведенні довгих транзакцій (напр. патерн Saga) (що особливо ускладнено у випадку мікросервісної архітектури з посервісними базами даних, що вимагає розподілення транзакцій);
- збереження проміжних станів виконання;
- відстеження прогресу у асинхронних чергах, що виникають з попередньої вимоги, тощо.

Те, що на перший погляд здається простою UX-вимогою (“показати прогрес та дозволити скасування”), трансформується в складну розподілену систему з управлінням станом, транзакційною семантикою та механізмами відновлення. При цьому архітектурна складність зростає зі збільшенням гранулярності контролю користувача над процесом.

Обробка послаблених (англ. degraded) чи аварійних станів системи, вимоги до швидкості відновлення та інші споріднені вимоги можуть бути описані як у термінах доступності, так і у термінах надійності.

Доступність зосереджується на мінімізації часу, протягом якого система недоступна для користувачів. Ключовим питанням є “Чи можуть користувачі отримати доступ до системи?” Метрики доступності вимірюють відсоток часу безперебійної роботи та швидкість відновлення після збою. З архітектурної перспективи, забезпечення високої доступності вимагає механізмів швидкого виявлення збоїв, автоматичного перезапуску, резервування компонентів та балансування навантаження.

Надійність, натомість, концентрується на здатності системи продовжувати коректну роботу попри виникнення помилок. Центральним є питання “Чи правильно система виконує свої функції під час та після збою?” Надійність вимірюється через ймовірність безвідмовної роботи, цілісність транзакцій та здатність

системи ізолювати помилки. Архітектурні рішення для надійності включають транзакційність, черги з відмітками виконання, механізми відновлення стану, обробку виняткових ситуацій та запобігання каскадним відмовам.

Розглянемо випадок аварійного стану системи (табл. 1.7).

Таблиця 1.7. Приклад сценарію атрибуту якості: Доступність в аварійному стані

Елемент	Опис
Джерело	Довільний компонент системи
Стимул	Виникнення критичної помилки та припинення роботи
Артефакт	Програмне забезпечення
Середовище	Підвищене навантаження (8000 одночасних користувачів)
Відгук	ПЗ виявляє збій, перезавантажує проблемний компонент, перенаправляє запити на резервні копії, зберігає стан користувачьких сесій.
Міра відгуку	Роботу компоненти відновлено протягом 20 секунд без втручання адміністратора, випадок зареєстровано та сповіщено адміністратора про випадок.

У цьому сценарії успіх визначається швидкістю відновлення доступу до ПЗ. Архітектурні імплікації включають необхідність механізмів моніторингу з малою затримкою виявлення (health checks кожні 5 секунд), оркестраторів для автоматичного перезавантаження (Kubernetes з налаштованими liveness probes), та наявності декількох копій компоненти для негайного перенаправлення трафіку. Критичним є час між виникненням збою та повним відновленням обслуговування.

Цей же сценарій, описаний з точки зору надійності, акцентує увагу на мінімізації наслідків аварії та збереженні коректності операцій (табл. 1.8)

Таблиця 1.8. Приклад сценарію атрибуту якості: Надійність

Елемент	Опис
Джерело	Довільний компонент системи
Стимул	Виникнення критичної помилки та припинення роботи
Артефакт	Програмне забезпечення
Середовище	Підвищене навантаження (8000 одночасних користувачів)
Продовження на наступній сторінці	

Таблиця 1.8 – Продовження

Елемент	Опис
Відгук	Система виявляє збій, перезапускає проблемний компонент, перенаправляє запити на резервні копії, зберігає стан користувачьких сесій.
Міра відгуку	Жодна виконувана транзакція не втрачається, аварійний стан не поширюється на інші компоненти системи

Тут успіх визначається збереженням цілісності даних та ізоляцією помилки. Архітектурні наслідки можуть включати транзакційні гарантії, механізми запобігання каскадним відмовам, ідемпотентні операції (такі, повтор яких не змінює результат) для безпечного повтору запитів, та event sourcing або write-ahead logging для відновлення стану. Акцент є на гарантії або успішного завершення операції, або коректного відкидання без часткових змін.

Сценарії доступності, портативності та надійності розглядались з точки зору локально розгорнутої системи. Масштабованість розглянемо на прикладі централізованого хмарного програмного забезпечення виду SaaS (Something-as-a-Service) (табл. 1.9).

Таблиця 1.9. Приклад сценарію атрибуту якості: Масштабованість

Елемент	Опис
Джерело	Відділ продажів
Стимул	Підписання контрактів з 5 новими університетами протягом кварталу, що додає 45,000 користувачів до існуючої мультитенантної платформи
Артефакт	Спільна хмарна система управління навчанням
Середовище	Система обслуговує 18 університетів (150,000 активних користувачів) з одного кластера в одному дата-центрі
Відгук	Система автоматично виділяє нові репліки необхідних сервісів для підтримки навантаження, доєднує їх до черги обробки завдань, масштабує бази даних.
Міра відгуку	Вимоги до продуктивності продовжують виконуватись протягом та після активізації нових користувачів на платформі

Дана вимога описує “гумовість” програмного забезпечення, забезпечення якої впливає на архітектуру на всіх рівнях, від можливості автоматичного додавання

нових реплік сервісів та їх автоматичного розпізнавання іншими частинами ПЗ до коректної обробки стану реалізаціями окремих функцій, а також вибору баз даних та їх конфігурації для масштабованості.

Продуктивність безпосередньо впливає на користувацький досвід та економічну ефективність системи управління навчанням. Високе навантаження під час реєстрації на курси, здачі екзаменаційних робіт або масового перегляду результатів створює критичні моменти, коли недостатня продуктивність може призвести до втрати довіри користувачів та репутаційних втрат. Розглянемо типовий сценарій пікового навантаження (табл. 1.10):

Таблиця 1.10. Приклад сценарію атрибуту якості: Масштабованість

Елемент	Опис
Джерело	Студенти всіх курсів
Стимул	Відкриття реєстрації на вибіркові дисципліни о 9:00 в понеділок
Артефакт	Модуль реєстрації на курси, API gateway, база даних
Середовище	12,000 студентів намагаються зареєструватися одночасно протягом перших 5 хвилин; кожен студент обирає 3-5 курсів з обмеженою кількістю місць
Відгук	Система обробляє запити реєстрації, перевіряє доступність місць, резервує місця, оновлює лічильники, надсилає підтвердження, відхиляє запити при заповненні курсу
Міра відгуку	P50 латентність < 300 мс, P99 < 2 с; пропускна здатність ≥ 5000 транзакцій/сек; 0% втрачених реєстрацій через timeout; відсутність deadlocks при конкурентному доступі до однакових курсів

Метрики продуктивності часто деталізуються шляхом опису за допомогою квантилів (“скільки відсотків запитів мусять відповідати метриці”), посилаючись на певний розподіл можливих значень метрики, замість середніх значень. Таким чином, користувацький досвід оцінюється точніше: P50 (медіана) < 300 мс означає, що половина всіх запитів обробляється швидше, ніж за 300 мілісекунд, забезпечуючи плавний досвід для більшості користувачів. P99 < 2 с гарантує, що навіть 99% найповільніших запитів (лише 1% може бути повільнішим) завершуються за прийнятний час — це критично, оскільки саме найповільніші запити часто визначають сприйняття системи користувачами та можуть вказувати на приховані проблеми з конкуренцією або ресурсами.

Сценарій (табл. 1.10) ілюструє класичну проблему “thundering herd” — раптового сплеску навантаження, що перевищує нормальний рівень у десятки разів. У таких випадках, система повинна забезпечити, наприклад, щоб двоє студентів не могли одночасно зайняти останнє місце на курсі — для цього використовується перевірка версії запису при оновленні (якщо хтось встиг змінити дані раніше, друга спроба буде відхилена). Також для полегшення навантаження можуть бути використані кеші загальної інформації, що не змінюється швидко (готові відповіді на запити інформації про курс, пошуку курсів тощо) та інші засоби оптимізації для пікових сценаріїв.

1.1.3 Архітектурні рішення

Важливі вибори, які впливають на структуру та поведінку системи фіксуються у вигляді *записів про архітектурні рішення* (англ. Architecture Decision Record).

Architecture Decision Record є документом, що фіксує окреме архітектурне рішення, спричинене функціональною чи нефункціональною вимогою(-ами), разом із його контекстом, обґрунтуванням та наслідками. ADR вирішують фундаментальну проблему втрати архітектурного знання в процесі розробки програмного забезпечення, роблячи неявні припущення явними та зберігаючи історію прийняття рішень для майбутніх розробників. Сукупність усіх ADR, створених і підтримуваних у проекті, формує *журнал архітектурних рішень* (Architectural Decision Log, ADL).

Журнал архітектурних рішень є “живим” документом — ведеться, доповнюється та розширюється протягом усього життєвого циклу програмного забезпечення.

Поняття “архітектура” у цьому контексті не слід трактувати надто обмежувально. ADR можуть документувати широкий спектр рішень: вибір технологій, мов програмування, бібліотек або функціональних можливостей (при виконанні функціональних вимог, які є зовнішніми до ADR). Будь-яке рішення, що потенційно впливає на архітектуру системи, може вважатися архітектурним рішенням.

Базовий формат запису про архітектурне рішення — Y-Statement: “В контексті <задачі>, зіткнувшись з <проблемою>, (опціонально: нехтуючи <альтернативами>), було прийнято <рішення> для досягнення <мети, бажаного результату, властивості системи>, приймаючи <недоліки, негативні наслідки>, (опціонально: оскільки <додаткові причини чи переваги>)”.

Розширеною та деталізованою структурою є MADR: Markdown Architectural Decision Records.

Обов'язкові компоненти включають *заголовок* — короткий опис проблеми та знайденого рішення; *контекст та формулювання проблеми* — вільна форма опису ситуації у двох-трьох реченнях або як ілюстративна історія; *розглянуті опції* — список альтернатив, що оцінювалися; *результат рішення* — обрана опція з обґрунтуванням.

Опціональні елементи дозволяють розширити документацію: *рушійні сили рішення* (англ. Decision Drivers) описують фактори, що впливають на вибір; наслідки, як позитивні, так і негативні; *підтвердження* визначає методи верифікації дотримання ADR (перевірка коду, тест тощо); *детальний аналіз опцій* включає структуровані переваги та недоліки кожної альтернативи; *додаткова інформація* може містити докази, згоду команди, часові рамки реалізації або посилання на пов'язані рішення.

MADR передбачає розміщення даних документів безпосередньо поряд з кодовою базою, у docs/decisions у вигляді markdown-файлів, названих за шаблоном NNNN-назва-через-дефіс.md, де NNNN є чотиризначним порядковим номером рішення.

Номери ADR ніколи не використовуються повторно. Якщо рішення змінюється, старий ADR зберігається з позначкою “замінено” (англ. superseded) і посиланням на новий ADR, що забезпечує збереження повної історії рішень. У метаданих YAML це виражається як “статус: замінений на ADR-0123”.

Кумулятивно прийняті архітектурні рішення варто відображати за допомогою діаграм. Для цього можуть слугувати як окремі UML-діаграми (напр. діаграма компонент чи діаграма розгортання), так і багаторівневі моделі, такі як C4.

ADR вважаються (і мають бути) незмінними документами, на практиці багато команд застосовують прагматичну змінність: нова інформація може додаватися до існуючих ADR із позначками дати та нотатками про те, що інформація надійшла після початкового рішення — у процесі розробки розуміння рішень завжди еволюціонує. Головним є недоторканість вже записаної в ADR інформації — прозорість історії рішень є необхідною для розуміння майбутніми розробниками не лише поточного стану ПЗ, а і процесу та причин його еволюції. Також, зберігання ADR запобігає втраті знань при перебігу часу чи звільненні окремих членів команди.

Рекомендований розмір ADR — 1-2 сторінки. Має бути витриманий баланс між повнотою та лаконічністю. Надто короткі ADR можуть не надавати достатнього контексту; надто довгі стають важкими для читання та підтримки.

Поточний стан архітектури ПЗ — кумулятивний результат історії прийнятих архітектурних рішень — варто відображати за допомогою діаграм. Для цього можуть слугувати як окремі UML-діаграми (напр. діаграма компонент чи діаграма розгортання), так і багаторівневі моделі, такі як C4.

Модель С4 візуалізує архітектуру за допомогою ієрархічного набору діаграм архітектури програмного забезпечення. Кожен з рівнів С4 представляє власний рівень абстракції;

Контекст є першим рівнем С4 та представляють загальний обсяг програмного забезпечення, глобальну взаємодію та відносини з користувачами та іншим ПЗ;

Контейнери є другим рівнем С4 та декомпонують ПЗ на взаємопов'язані контейнери, які представляють певний застосунок чи сховище даних;

Компоненти є третім рівнем С4 та розкладають контейнери на взаємопов'язані компоненти, відображають відношення цих компонент до інших компонент, контейнерів чи ПЗ;

Код є четвертим рівнем С4 та надає додаткові, відносно низькорівневі, деталі про будову програмного забезпечення, що мають пряму відповідність до реалізації ПЗ.

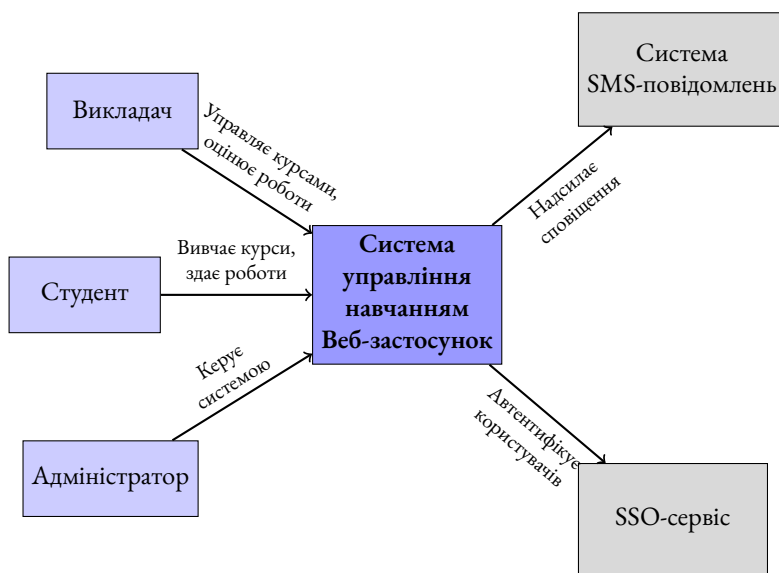


Рис. 1.5. Діаграма контексту системи управління навчанням (C4 Level 1)

Взаємозв'язок між ADR та C4-діаграмами є двостороннім. ADR документають рішення, що формують архітектуру, видиму на діаграмах. Натомість, C4-діаграми надають візуальний контекст для розуміння масштабу та впливу архітектурних рішень. Коли архітектурне рішення змінюється (новий ADR замінює

старий), відповідні С4-діаграми мають бути оновлені для відображення поточного стану системи.

Розглянемо приклад візуалізації архітектури системи управління навчанням на різних рівнях С4.

Діаграма контексту (рис. 1.5) надає високорівневий огляд системи та її взаємодій. Цей рівень абстракції корисний для комунікації з зацікавленими сторонами, що не мають технічної експертизи. На прикладі система управління навчанням обслуговує три типи користувачів та інтегрується з двома зовнішніми системами.

У моделі С4 програмна система як ціле є найабстрактнішим представленням розроблюваного ПЗ та представляє те, що створюється, підтримується та розвивається в певному процесі розробки.

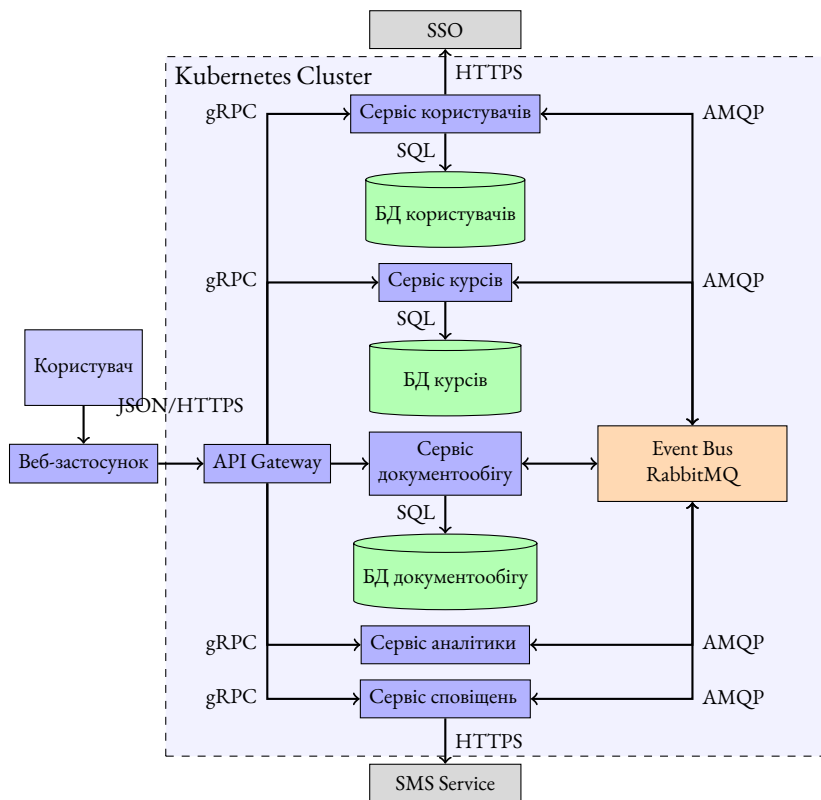


Рис. 1.6. Діаграма контейнерів системи управління навчанням (C4 Level 2)

У програмної системи є чітко визначена межу відповідальності — усе всередині цієї межі доступне для модифікації, а усе зовні є зовнішніми залежностями. Межі системи та чітке визначення, де починається і закінчується “система”, залежить від конкретної компанії та інших факторів. В багатьох випадках, межею системи на діаграмі першого рівня C4 є межі однієї команди.

Діаграма контейнерів (рис. 1.6) деталізує внутрішню структуру системи, показуючи основні технологічні рішення. На прикладі відокремлення фронтенду, API Gateway та бекенд-сервісів відображає архітектурне рішення про використання мікросервісної архітектури. Вибір окремих баз даних для кожного сервісу забезпечує слабку зв’язаність, як і RabbitMQ для обміну подіями для зв’язку між сервісами — кожне з цих рішень має бути задокументоване окремим ADR.

Діаграма компонентів (рис. 1.7) деталізує внутрішню структуру сервісу документообігу, піднімаючись на один рівень абстракції над кодом. У моделі C4 компонента є групою споріднених функціональностей, які інкапсульовані за чітко визначеним інтерфейсом.

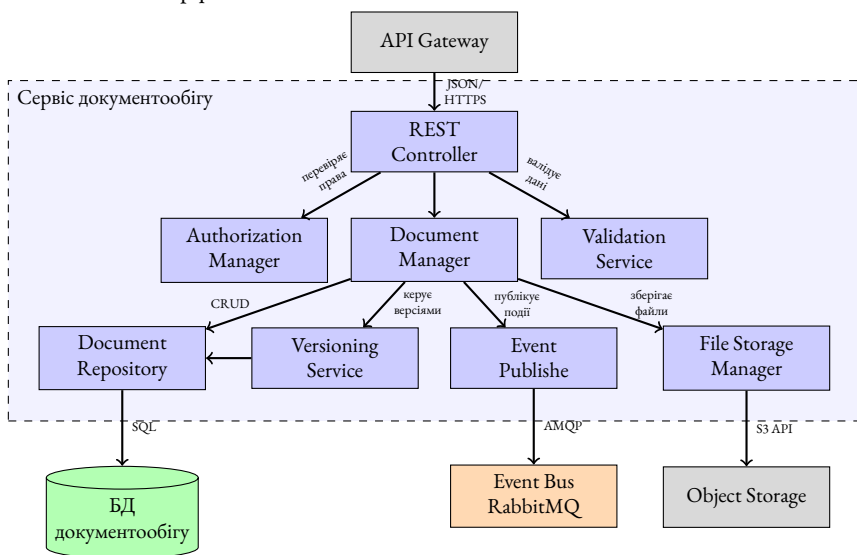


Рис. 1.7. Діаграма компонентів сервісу документообігу (C4 Level 3)

На прикладі, REST Controller приймає HTTP-запити та делегує обробку спеціалізованим компонентам: Authorization Manager перевіряє права доступу, Validation Service валідує вхідні дані, а Document Manager координує основну бізнес-логіку. Компоненти не розгортаються окремо — натомість усі компоненти всередині контейнера виконуються в одному процесному просторі. В залежності

від мови програмування та інших факторів, компонент може бути набором класів, функцій, файлів тощо.

Четвертий рівень С4 — код — описує імплементаційні деталі: конкретні класи, інтерфейси, еніти, функції та об'єкти, з яких побудовані компоненти. Якщо діаграма компонентів показує, що Document Manager координує бізнес-логіку, то діаграма коду розкриває, які саме класи реалізують цю логіку: DocumentService, DocumentValidator, DocumentMapper, DocumentEntity тощо, і як вони взаємодіють через виклики методів, успадкування та композицію. Ключова відмінність між рівнями полягає в меті: третій рівень призначений для архітектурних дискусій про розподіл відповідальності та залежності між компонентами, у той час як четвертий рівень слугує для розуміння конкретної реалізації цих компонентів мовою програмування. Для більшості архітектурних обговорень та документування достатньо діаграм до третього рівня включно, оскільки вони абстрагуються від імплементаційного шуму, зберігаючи фокус на архітектурно значущих рішеннях.

1.2 Архітектурне представлення

Архітектурне представлення відображає систему у площині певної точки зору. Різні представлення використовуються для опису архітектури та відповіді на питання різних точок зору, розв'язання занепокоєнь різних зацікавлених сторін: кінцевого користувача, розробника, проектного менеджера, системного інтегратора тощо.

Кожне з необхідних представлень описується певною моделлю чи набором моделей, які в свою чергу найчастіше містять діаграми, але окрім них можуть містити також таблиці, текст, графіки тощо. Кожне з окремих представлень також може відображати систему з точки зору певного *архітектурного стилю*, що дозволяє сумістити в одному програмному забезпеченні декілька архітектурних стилів [8].

Архітектурний стиль є узагальненням архітектур, що мають спільні риси та поведінки, що виражається у схожих взаємодіях між компонентами ПЗ.

Архітектурний патерн описує певну повторювану *проблему*, що виникає у певних *контекстах* та надає перевірене *рішення* проблеми. Рішення полягає у описі ролей компонент, що беруть участь, їх відповідальностей та взаємодій. Архітектурний стиль відповідає рішенням у патерні.

Архітектура може бути описана різними способами:

- Вільний — довільний набір прямокутників, тексту та ліній, що їх поєднують;
- Напів-вільний — такі як UML;

- Формалізований — такі як AADL.

Для систематичного опису архітектури програмного забезпечення застосовуються різноманітні фреймворки та методології організації представлень. Одним з найпоширеніших підходів до структурування архітектурної документації є модель 4+1 представлень (рис. 1.8), запропонована Крухтеном [8].



Рис. 1.8. Модель представлень “4+1”

Логічне представлення є об’єктною моделлю проектування (коли використовується об’єктно-орієнтований метод проектування). Логічна архітектура насамперед підтримує функціональні вимоги — що система повинна надавати своїм користувачам з точки зору послуг. Система декомпонується на набір ключових абстракцій, взятих (переважно) з проблемної області.

Представлення розробки описує статичну організацію програмного забезпечення в його середовищі розробки. Архітектура розробки представлена діаграмами модулів та підсистем, які показують відношення “експорту” та “імпорту” між компонентами. Переважно архітектура розробки враховує внутрішні вимоги, пов’язані з легкістю розробки, управлінням програм-

ним забезпеченням, повторним використанням чи спільністю коду, а також обмеження, накладені інструментарієм або мовою програмування.

Процесне представлення охоплює аспекти паралелізму та синхронізації проектування. Процесна архітектура враховує деякі нефункціональні вимоги, такі як продуктивність та доступність. Вона розглядає питання паралелізму та розподілу, цілісності системи, відмовостійкості, а також те, як основні абстракції з логічного представлення вписуються в процесну архітектуру — в якому потоці керування фактично виконується операція для об'єкта.

Фізичне представлення описує відображення програмного забезпечення на апаратне забезпечення та відбиває його розподілений аспект. Воно стосується топології програмних компонентів на фізичному рівні, а також фізичних з'єднань між цими компонентами. Фізична архітектура враховує переважно нефункціональні вимоги системи, такі як доступність, надійність (відмовостійкість), продуктивність (пропускна здатність) та масштабованість. Програмне забезпечення виконується в мережі комп'ютерів або вузлів обробки. Різні ідентифіковані елементи — мережі, процеси, задачі та об'єкти — потрібно відобразити на різні вузли. Для представлення фізичного подання використовується UML-діаграма розгортання, яка показує апаратні вузли, програмні компоненти, розміщені на них, та комунікаційні шляхи між ними.

Опис архітектури — прийняті рішення — може бути організований навколо цих чотирьох представлень та проілюстрований кількома обраними варіантами використання або сценаріями, які стають п'ятим представленням. Елементи в чотирьох представленнях демонструють свою безшовну взаємодію за допомогою невеликого набору важливих сценаріїв — екземплярів більш загальних варіантів використання — для яких ми описуємо відповідні сценарії (послідовності взаємодій між об'єктами та між процесами).

Для моделювання архітектури ПЗ користуючись підходом 4+1 представлень часто використовують UML-діаграми (табл. 1.11).

Таблиця 1.11. Відповідні UML-діаграми для подань моделі 4+1

Назва подання	Відповідні UML-діаграми
Логічне представлення	Діаграма класів, діаграма об'єктів, діаграма компонентів, діаграма пакетів, діаграма композитної структури
Продовження на наступній сторінці	

Таблиця 1.11 – Продовження

Назва подання	Відповідні UML-діаграми
Процесне представлення	Діаграма діяльностей, діаграма станів, діаграма послідовності, діаграма синхронізації, діаграма взаємодії
Фізичне представлення	Діаграма розгортання
Представлення розробки	Діаграма компонентів, діаграма пакетів
Представлення сценаріїв	Діаграма варіантів використання

Важливо зазначити, що не всі представлення є обов’язковими для кожного проєкту. Вибір необхідних представлень залежить від розміру та складності системи, кількості зацікавлених сторін, а також специфічних вимог проєкту. Так, для невеликих систем з простою архітектурою може бути достатньо лише логічного представлення та сценаріїв.

1.3 Дизайн-документ

Розглянуті у попередніх розділах підходи до документування вимог, архітектурних рішень та представлень архітектури інтегруються у дизайн-документ — центральний артефакт, що формалізує та фіксує ключові архітектурні рішення, прийняті на основі архітектурно значущих вимог. Цей документ слугує мостом між високорівневими бізнес-цільми та детальною технічною реалізацією та забезпечує спільне розуміння архітектури серед усіх учасників процесу розробки.

Структура дизайн-документа відображає процес прийняття архітектурних рішень від розуміння контексту та обмежень через аналіз альтернатив до обґрунтування конкретних виборів:

- Мета та контекст
- Архітектурно значущі вимоги
- Архітектурні рішення
- Архітектурна специфікація

Важливо пам’ятати, що дизайн-документ є живим артефактом, який повинен оновлюватися синхронно зі змінами в архітектурі системи. Регулярний перегляд та актуалізація документа забезпечує його цінність як довідкового матеріалу та запобігає розходженню між документацією та реалізацією.

Мета та контекст Розділ повинен чітко артикулювати, як технічна архітектура підтримує стратегічні цілі організації. Це включає:

- Основну бізнес-проблему, яку вирішує система
- Очікувану бізнес-цінність
- Часові рамки та критичні віхи проекту
- Зацікавлені сторони (англ. stakeholder) та їхні очікування

Стратегія компанії та контекст, у якому існує/існуватиме програмне забезпечення, мають безпосередній вплив на функціональні та нефункціональні вимоги, і таким чином – на архітектуру програмного забезпечення.

Архітектурно значущі вимоги Даний розділ містить вимоги до програмного забезпечення, як функціональні, так і нефункціональні, описані у будь-якій з розглянутих (чи інших) форм:

- Вільна форма;
- Користувацька історія;
- Критерії прийняття;
- Шаблон EARS;
- Варіанти використання;
- Атрибути якості;
- тощо.

Вимоги до програмного забезпечення деталізують контекст програмного забезпечення, фіксують та формалізують основні впливи на його архітектуру.

Архітектурні рішення Розділ мусить містити усі *важливі* архітектурні рішення, а також ті, що мають найбільшу ймовірність викликати питання в майбутньому – неортодоксальні, неочікувані. Аналіз альтернативних рішень, вказання очікуваних недоліків прийнятих рішень та обґрунтування незначущості недоліків чи засобів їх пом'якшення є важливими частинами даного розділу.

Архітектурна специфікація Даний розділ описує *виконання* архітектурних рішень, деталізує архітектуру, описує особливості взаємодії окремих елементів архітектури (REST API, структури повідомлень, формати даних тощо). Даний розділ може бути як невід'ємною частиною дизайн-документу, так і окремим набором супровідних документів.

Дизайн-документ, створений з урахуванням цих принципів та практик, є інструментом підтримки архітектурної цілісності, комунікації та еволюції програмного забезпечення протягом усього життєвого циклу. Систематичне документування архітектури — від вимог через рішення до їхнього втілення — забезпечує збереження архітектурного знання та ефективну співпрацю всіх учасників процесу розробки.

Розділ 2

Тактики та патерни

2.1 Архітектурні тактики

Архітектурні рішення формуються на основі застосування тактик атрибутів якості та архітектурних патернів. Розуміння взаємозв'язку між атрибутами якості, тактиками їх досягнення, архітектурними патернами та стилями є необхідним для прийняття обґрунтованих архітектурних рішень.

Тактика атрибуту якості є проектним примітивом, спрямованим на керування відгуком програмного забезпечення на стимул з точки зору окремого атрибуту якості [9] (рис. 2.1).

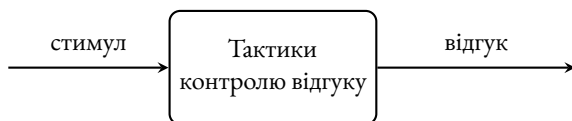


Рис. 2.1. Місце архітектурної тактики

Тактики є базовими будівельними блоками архітектурного проектування, аналогічними до атомів у хімії. Кожна тактика зосереджується на вирішенні однієї архітектурної проблеми, пов'язаної з конкретним атрибутом якості. Архітектурні рішення часто включають комбінації декількох тактик для задоволення складних нефункціональних вимог. Тактики організовані навколо атрибутів якості програмного забезпечення. Для кожного атрибуту якості існує набір тактик, що дозволяють архітектору систематично підходити до забезпечення відповідних характеристик системи. Розглянемо основні категорії тактик для ключових атрибутів якості.

2.1.1 Тактики доступності

Тактики доступності спрямовані на перетворення стимулу — відмови компоненту або частини програмного забезпечення — у відгук, що мінімізує або усуває наслідки відмови. Метою є маскування відмови, її запобігання або виконання відновлення системи.

Тактики доступності класифікуються за трьома основними категоріями:

Виявлення відмов включає механізми ідентифікації проблем до того, як вони призведуть до повної недоступності системи. До таких тактик належать моніторинг стану компонентів (ping/echo, heartbeat), часові мітки для виявлення застарілих даних, перевірка умов функціонування, санітарна перевірка вхідних даних, голосування між паралельними обчисленнями, виявлення виняткових ситуацій та самотестування компонентів.

Відновлення після відмов охоплює тактики повернення системи до працездатного стану після виявлення проблеми. Підкатегорія підготовки та відновлення включає резервні копії компонентів, відкат до попереднього стану, обробку винятків, оновлення програмного забезпечення, повторні спроби виконання операцій, ігнорування хибної поведінки, контрольована деградація (“елегантна”, англ. Graceful Degradation) — поступове обмеження функціоналу при відбоях. Підкатегорія повторного введення в експлуатацію містить тактики shadow-режиму (паралельне виконання на резервній системі), ресинхронізації стану, ескаляції перезапуску (від простого до складного) та безперервного переадресування запитів.

Запобігання відмовам спрямовані на усунення потенційних джерел проблем до їх виникнення. Включають вилучення компонентів з обслуговування для профілактики, використання транзакцій для забезпечення атомарності операцій, прогнозні моделі для передбачення відмов, запобігання винятковим ситуаціям та збільшення множини компетенцій системи для обробки ширшого спектру ситуацій.

2.1.2 Тактики продуктивності

Тактики продуктивності контролюють час, необхідний для обробки подій (стимулів), що надходять до системи. Мета полягає у забезпеченні обробки подій у межах визначених часових та ресурсних обмежень. Тактики продуктивності розділяються на дві основні категорії:

Керування попитом на ресурси зменшує навантаження на систему або робить його більш передбачуваним. Включає керування робочими запитами

(управління чергами, пріоритизація), обмеження відгуку на події (вибіркова обробка, дроселювання), пріоритизацію подій за важливістю, зменшення обчислювального навантаження через оптимізацію алгоритмів, обмеження часу виконання операцій та підвищення ефективності обчислень.

Керування ресурсами збільшує доступність або ефективність використання ресурсів. Тактики включають збільшення ресурсів (додаткові сервери, пам'ять), введення паралелізму (багатопотоковість, розподілені обчислення), підтримку декількох копій обчислень для паралельної обробки, підтримку декількох копій даних (кешування, реплікація), обмеження розмірів черг для запобігання перевантаженню та планування ресурсів для оптимального розподілу навантаження.

Продуктивність характеризується латентністю (час відгуку на окремий запит), пропускну здатністю (кількість операцій за одиницю часу) та використанням ресурсів. Вимоги до продуктивності часто формуються з використанням квантилів: P50 (медіана), P95, P99 тощо, що дозволяє точніше описати розподіл часу відгуку та уникнути впливу викидів на оцінку системи.

2.1.3 Тактики розгортуюваності

Тактики розгортуюваності забезпечують ефективне впровадження нових елементів програмного забезпечення у продуктивне середовище в межах визначених обмежень часу, вартості та якості.

Керування конвеєром розгортання включає тактики масштабованих випусків (поступове розгортання на зростаючу кількість вузлів), скриптування команд розгортання для автоматизації та можливість відкату до попередньої версії при виявленні проблем.

Керування розгорнутою системою охоплює тактики управління взаємодіями між сервісами (забезпечення сумісності інтерфейсів), пакування залежностей разом з компонентом для ізоляції та перемикачі функціональностей (feature toggles) для керування доступністю нових можливостей без повторного розгортання.

Розгортуюваність безпосередньо впливає на швидкість доставки змін користувачам та здатність системи до еволюції. Автоматизація процесів розгортання, мінімізація часу простою та забезпечення можливості швидкого відкату є критичними для сучасних програмних систем, особливо тих, що функціонують у хмарному середовищі.

2.1.4 Тактики інтегрованості

Тактики інтегрованості забезпечують ефективне об'єднання нових компонентів із існуючою системою в межах визначених обмежень часу, вартості та якості.

Обмеження залежностей зменшує зв'язаність між компонентами через інкапсуляцію деталей реалізації, використання проміжних компонентів для непрямой взаємодії, обмеження шляхів комунікації між елементами, дотримання стандартів взаємодії та абстрагування спільних сервісів.

Адаптація дозволяє компонентам пристосовуватися до різних контекстів через механізми виявлення можливостей інших компонентів, налаштування інтерфейсів під конкретні потреби та конфігурування поведінки без зміни коду.

Координація забезпечує узгоджену роботу компонентів через оркестрацію взаємодій, керування ресурсами та синхронізацію дій.

Інтегрованість є критичною для систем, що повинні взаємодіяти з численними зовнішніми сервісами, або для систем, де окремі компоненти розробляються різними командами чи постачальниками. Високий рівень інтегрованості дозволяє швидко впроваджувати нові можливості без суттєвих змін в архітектурі.

Окрім розглянутих вище, існують тактики для інших атрибутів якості:

Модифікованість включає тактики локалізації змін, запобігання ефекту ряби (ripple effect) та відкладання прив'язки до конкретних реалізацій.

Безпека охоплює тактики виявлення, протидії та відновлення після атак, а також реагування на інциденти безпеки.

Тестованість включає контроль та спостереження за станом системи, обмеження складності для спрощення тестування.

Зручність використання забезпечує підтримку ініціативи користувача, зворотний зв'язок про стан системи та можливість скасування дій.

Вибір відповідних тактик базується на аналізі сценаріїв атрибутів якості. Кожен сценарій визначає стимул, артефакт, середовище, відгук та міру відгуку, що дозволяє архітектору ідентифікувати, які саме тактики необхідні для задоволення вимоги.

2.2 Архітектурні патерни

Архітектурні патерни є наступним рівнем абстракції після тактик, об'єднуючи декілька проєктних рішень у цілісне рішення повторюваної проблеми.

Архітектурний патерн описує певну повторювану *проблему*, що виникає у певних *контекстах* та надає перевірене *рішення* проблеми. Рішення полягає у описі ролей компонент, що беруть участь, їх відповідальностей та взаємодій. Архітектурний стиль відповідає рішенням у патерні.

На відміну від тактик, які є атомарними та зосереджені на одному аспекті якості, патерни є “молекулярними” — вони конструюються з декількох тактик та проектних рішень для вирішення комплексної архітектурної проблеми. Структура опису архітектурного патерну включає:

- Контекст — умови, за яких виникає проблема;
- Проблема — специфічна архітектурна задача, що потребує вирішення;
- Рішення — опис структури компонентів, їхніх ролей та взаємодій;
- Наслідки — позитивні та негативні ефекти застосування патерну.

Архітектурні патерни часто асоціюються з конкретними атрибутами якості або архітектурними характеристиками. Основні категорії архітектурних патернів включають:

Патерни доступності та стійкості забезпечують безперебійне функціонування системи при виникненні відмов. Circuit Breaker запобігає каскадним відмовам шляхом припинення викликів до недоступного сервісу та надання можливості відновлення. Bulkhead ізолює ресурси між компонентами для обмеження впливу відмови однієї частини на решту системи. Health Endpoint Monitoring реалізує моніторинг стану компонентів через спеціалізовані кінцеві точки перевірки працездатності. Throttling обмежує споживання ресурсів окремими користувачами або компонентами для запобігання перевантаженню системи.

Патерни управління даними визначають організацію зберігання, доступу та узгодженості даних у розподілених системах. Cache-Aside забезпечує завантаження даних до кешу на вимогу при їх запиті. Event Sourcing зберігає повну послідовність подій, що змінюють стан даних, замість збереження лише поточного стану. Sharding розподіляє дані між множиною сховищ для горизонтального масштабування.

Патерни проєктування та реалізації визначають структурну організацію компонентів системи. Ambassador розміщує допоміжні компоненти для мережових запитів поруч з основним сервісом. Anti-Corruption Layer реалізує фасад між новою системою та застарілою або зовнішньою системою для трансляції між несумісними моделями даних. Gateway Offloading переносить спільну функціональність з окремих сервісів до шлюзу. Sidecar розміщує

допоміжні компоненти в окремому процесі поруч з основним застосунком. Strangler Fig поступово замінює застарілу систему новою шляхом послідовної міграції функціональності.

Патерни обміну повідомленнями організують асинхронну комунікацію між компонентами. Asynchronous Request-Reply розділяє запит та відповідь у часі через використання черг повідомлень. Competing Consumers дозволяє множині споживачів обробляти повідомлення з однієї черги паралельно. Priority Queue надає різні пріоритети повідомленням для забезпечення першочергової обробки критичних запитів. Publisher-Subscriber розв'язує відправників та отримувачів повідомлень через проміжний канал підписки дозволяючи асинхронний обмін даними.

Архітектор має формувати власний репертуар патернів на основі досвіду та вивчення різноманітних джерел, оцінюючи застосовність кожного патерну у конкретному контексті системи, що проектується.

Розділ 3

Архітектурні стилі

Вибір архітектурного стилю є одним з найбільш фундаментальних архітектурних рішень, оскільки він визначає базову організаційну структуру системи, типи компонентів та способи їхньої взаємодії. Різні стилі забезпечують різні архітектурні характеристики та по-різному задовольняють нефункціональні вимоги. Розуміння властивостей, переваг та обмежень основних архітектурних стилів є необхідним для обґрунтованого вибору та документування архітектурних рішень.

Архітектурний стиль є узагальненням сімейства архітектур, що мають спільні структурні характеристики та поведінку. Стиль описує іменовані відносини між компонентами, що охоплюють різноманітні архітектурні характеристики [10].

Архітектурний стиль одним з рішень у визначенні архітектурного патерну (де патерн є проблемою в контексті що має рішення). Архітектурний стиль визначає:

- Типи компонентів та їхні ролі;
- Топологію зв'язків між компонентами;
- Механізми взаємодії;
- Принципи розподілення відповідальності;
- Обмеження на організацію системи.

Вибір архітектурного стилю базується на аналізі архітектурно значущих вимог та архітектурних характеристик, що мають найвищий пріоритет для конкретної системи. Один з методів вибору стилю — звуження можливостей: спочатку обирається загальний стиль на основі вимог, потім конкретизуються технологічні

рішення для обчислень, зберігання даних та обміну повідомленнями, після чого формується конкретна архітектура застосунку з використанням референсних архітектур, принципів проектування та архітектурних патернів. Сучасні системи часто поєднують декілька архітектурних стилів у різних підсистемах. Така гібридна архітектура дозволяє оптимізувати характеристики окремих частин системи відповідно до їхніх специфічних вимог. Наприклад, система може використовувати подієво-керовану архітектуру для асинхронної обробки великих обсягів даних та шарову архітектуру для користувацького інтерфейсу та бізнес-логіки. Архітектурний стиль впливає на організацію команди розробників, процеси розгортання, інфраструктурні вимоги та технологічний стек. Рішення про вибір стилю є одним з найбільш фундаментальних архітектурних рішень і має бути ретельно обґрунтоване та задокументоване у відповідному записі про архітектурне рішення.

3.1 Монолітна архітектура

Монолітна архітектура — це підхід, за якого вся функціональність програмного продукту реалізована в одному кодовому базисі й розгортається як єдине ціле. Інакше кажучи, уся бізнес-логіка, інтерфейси користувача, обробка даних і доступ до бази даних знаходяться у межах однієї програми. У термінах C4 — у межах одного контейнеру.

З технічної точки зору моноліт можна уявити як багат шарову структуру, де існують рівень інтерфейсу (UI), рівень бізнес-логіки та рівень доступу до даних. Проте, попри логічний поділ на шари, фізично це все — один застосунок, який компілюється, розгортається і масштабується як єдине ціле.

Монолітна архітектура є найстарішим і найпростішим підходом до побудови програмних систем. Вона виникла ще в ті часи, коли програмування було тісно пов'язане з апаратним забезпеченням і основним завданням розробників було забезпечити коректне та швидке виконання інструкцій.

У 60–70-х роках перші інформаційні системи для бізнесу, наприклад бухгалтерські або банківські програми, часто реалізовувалися як цілісні монолітні застосунки. Операційні системи також існували як моноліти, у яких усі модулі взаємодіяли напряму і були невіддільними один від одного. У той час це не вважалося недоліком, адже головне завдання полягало у створенні працездатної системи, а не у масштабуванні чи інтеграції з іншими системами.

Згодом ця модель поширилась і на корпоративні додатки та програмне забезпечення масового користування. Фактично будь-яка рання ERP-система, більшість перших CRM та майже всі настільні програми кінця XX століття — це класичні моноліти:

- Microsoft Word або Excel у 90-х роках — монолітні системи. Усі функції — від інтерфейсу до збереження документів — реалізовані в одному додатку.
- Перші версії інтернет-магазинів та порталів працювали як моноліти: одна база даних, одна кодова база, один веб-сервер.
- SAP R/3 у своїх ранніх ітераціях була монолітною платформою: модулі бухгалтерії, управління персоналом, логістики зливалися у єдину систему, яка потребувала централізованого оновлення.

Наведені приклади демонструють універсальність монолітного підходу незалежно від предметної області та масштабу застосування. Домінування монолітної архітектури у період становлення програмної інженерії пояснюється не лише технологічними обмеженнями епохи, але й об'єктивними перевагами цього підходу для систем із чітко визначеним обсягом функціональності та передбачуваним навантаженням:

Простота. Немає складної інфраструктури. Розробник може розгорнути всю систему локально і працювати з нею.

Швидкість старту. Для створення MVP (мінімально життєздатного продукту) моноліт підходить найкраще. Команди стартапів можуть створити працюючий прототип за кілька тижнів.

Єдина база даних. Усі дані знаходяться у спільному сховищі, що полегшує задачу збереження узгодженості, а також реалізацію транзакційності.

Продуктивність викликів. Оскільки всі модулі працюють у межах одного процесу, виклики між ними виконуються миттєво, без мережових затримок.

Спрощене тестування. Легше перевірити систему як єдине ціле, не турбуючись про інтеграцію через мережові API.

Переваги та недоліки монолітної архітектури не є абсолютними характеристиками, а залежать від масштабу системи, розміру команди розробників та специфіки предметної області. Система, що на початковому етапі демонструє переваги простоти та швидкості розробки, може з часом зіткнутися з проблемами при зростанні навантаження, розширенні команди та функціональності:

Складність масштабування. Якщо навантаження зростає лише на одну частину системи, масштабувати доводиться весь застосунок.

Крихкість. Невелика помилка в одному модулі може призвести до падіння всієї системи.

Повільне оновлення. Для внесення змін у будь-який модуль потрібно перекомпілювати й розгортати всю систему.

Важкість підтримки великим колективом. Коли над одним монолітом працює десяток чи більше команд, виникають конфлікти, дублювання логіки, складнощі з координацією.

Обмежена гнучкість. Усі модулі зазвичай пишуться на одній мові програмування та в одному технологічному стеку, що може гальмувати інновації.

Перехід від переваг до недоліків відбувається поступово та визначається кількома факторами. Кодова база поступово зростає від кількох тисяч до сотень тисяч, а потім і мільйонів рядків коду, що ускладнює навігацію та розуміння системи. Команда розробників розширюється від невеликої групи до десятків чи сотень осіб, що призводить до координаційних витрат та конфліктів при одночасній роботі над спільною кодовою базою. Навантаження на систему зростає нерівномірно між різними функціональними модулями, однак монолітна природа архітектури унеможливує селективне масштабування окремих компонентів. Критичним моментом є втрата можливості ефективної модифікації системи. Те, що на початку розробки сприймається як перевага єдиної кодової бази, поступово трансформується у технічний борг, коли зміна одного модуля вимагає ретельного аналізу впливу на інші частини системи, повного перезавантаження програмного забезпечення та ризику непередбачуваних побічних ефектів. Простота початкового розгортання поступається місцем складності підтримки узгодженості великої кількості взаємозалежних компонентів.

Моноліт можна порівняти з багатопверховим будинком, де все інтегровано в єдину конструкцію: електрика, вода, опалення, ліфти. Коли будинок невеликий — це зручно. Але якщо будинок виростає у хмарочос, будь-яка проблема (наприклад, поломка центральної системи опалення) впливає на всіх мешканців. Натомість у “містечку” кожен будинок може мати власну систему і поломка одного будинку не паралізуватиме життя інших — це стало підґрунтям для створення сервіс-орієнтованої архітектури.

3.2 Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура (англ. *Service-Oriented Architecture*, SOA) з’явився як реакція на обмеження монолітних систем у період, коли інформаційні технології почали швидко інтегруватися у бізнес-процеси великих організацій. Наприкінці 90-х і на початку 2000-х років компанії зіткнулися з тим, що програмні продукти, які були зручними на старті, з часом ставали “каменем на шії”: оновити їх ставало все складніше, розширювати — майже неможливо, а інтегрувати з іншими системами — справжнім викликом.

Наприклад, умовний банк на початку 2000-х. У роботі — десятки різних програм: одна для обліку депозитів, інша для кредитів, ще одна для роботи з банкоматами. Усі вони були написані як окремі моноліти, часто різними командами і навіть різними компаніями. Коли виникла потреба “зшити” їх разом, з’явилася величезна проблема — моноліти погано інтегруються. Саме тоді і виникла ідея організувати систему не як єдиний величезний блок, а як набір незалежних сервісів, які взаємодіють через стандартизовані інтерфейси.

Основна ідея SOA полягає у тому, що складна система розбивається на менші функціональні одиниці — сервіси. Кожен сервіс виконує обмежену, але чітко визначену функцію: наприклад, “обробка платежів”, “ідентифікація користувача”, “збереження документів”.

Важливо підкреслити, що у SOA сервіси не просто є “модулями” всередині однієї програми. Вони існують як самостійні сутності, які можна запускати, оновлювати, масштабувати незалежно від інших. Взаємодія між ними відбувається через мережеві протоколи — спочатку це був SOAP (з XML-повідомленнями), пізніше — REST (з JSON), а сьогодні часто використовують gRPC чи асинхронні повідомлення через брокери (Kafka, RabbitMQ).

Аналогією до сервісу є невеликий магазин у торговельному центрі. Кожен магазин продає щось своє: одяг, техніку, продукти. Але всі вони користуються спільною інфраструктурою — електрикою, водопроводом, охороною. Аналогічно, у SOA кожен сервіс вирішує конкретне завдання, але при цьому вся система працює як єдиний “торговельний центр”.

Розглянемо детальніше сильні сторони сервіс-орієнтованої архітектури:

Масштабованість. Якщо у системі зростає навантаження лише на один модуль, наприклад, сервіс авторизації, можна додати більше його екземплярів, не чіпаючи решту. Це набагато ефективніше, ніж масштабувати весь моноліт.

Гнучкість. Різні сервіси можуть бути реалізовані на різних мовах програмування чи навіть розміщені на різних платформах. Наприклад, сервіс машинного навчання можна написати на Python, а сервіс обліку транзакцій — на Java. Вони все одно будуть взаємодіяти, бо їх поєднує стандартний інтерфейс.

Надійність. Якщо один сервіс виходить з ладу, уся система не обов’язково “падає”. Наприклад, якщо тимчасово не працює сервіс рекомендацій товарів, клієнти все одно можуть робити покупки.

Зручність командної роботи. У великих компаніях різні команди можуть відповідати за різні сервіси. Це дозволяє працювати паралельно, зменшуючи залежності та конфлікти.

Проте переваги сервіс-орієнтованої архітектури супроводжуються істотними обмеженнями та компромісами, які необхідно враховувати при прийнятті архітектурних рішень. Основні недоліки SOA впливають безпосередньо з її розподіленої природи та організаційних вимог:

Складність інфраструктури. Щоб SOA працювала стабільно, потрібні потужні інструменти моніторингу, логування, балансування навантаження, автоматичного розгортання. Без цього система швидко перетворюється на хаос.

Високі накладні витрати. Кожен запит у SOA проходить через мережу. Якщо у моноліті виклик функції виконується “всередині програми” за мікросекунди, то у SOA — це мережевий запит, що може займати мілісекунди і навіть більше. При великій кількості викликів це створює серйозні затримки.

Складність тестування. Тестування окремого сервісу — просте завдання, але тестування всієї системи як єдиного цілого вимагає складних інтеграційних сценаріїв.

Зростання організаційних витрат. Кожна команда має не лише писати код, але й підтримувати CI/CD, документацію API, тестові середовища. Це підвищує вартість розробки.

Практичне застосування сервіс-орієнтованої архітектури демонструє широкий спектр результатів залежно від контексту впровадження, масштабу організації та дотримання архітектурних принципів. Аналіз конкретних випадків дозволяє ідентифікувати фактори успіху та типові помилки. Успішне впровадження сервіс-орієнтованої архітектури характерне для організацій, що мають достатні ресурси, чітко визначені межі сервісів та зрілі процеси розробки:

Amazon. Класичний приклад: спочатку система існувала як моноліт, але зі зростанням масштабів компанія перейшла на SOA. Це дозволило окремо масштабувати модулі “кошика”, “каталогу”, “платежів”. Згодом SOA еволюціонувала у мікросервіси, інфраструктура яких стала основою AWS.

Банківські системи. Наприклад, обробка платежів відокремлена від управління рахунками чи модулю AML (anti-money laundering). Завдяки цьому банки можуть незалежно оновлювати сервіси і підключати нові зовнішні платформи (наприклад, Apple Pay).

Медіаплатформи. Netflix відомий тим, що активно використовує сервісну архітектуру: сервіси для потокового відтворення, рекомендацій, управління користувачами працюють незалежно, що дозволяє масштабувати їх під пікові навантаження.

Невдалі впровадження зазвичай пов'язані з недооцінкою складності розподілених систем, передчасною оптимізацією або неправильним визначенням меж сервісів:

Невеликі компанії, які намагаються “почати” відразу з SOA. Якщо стартап з 5 людей намагається побудувати систему з десятків сервісів, то більшість часу команда витратить на інфраструктуру, а не на продукт. Такі приклади не рідкісні: компанії банкрутують, бо не встигають вивести продукт на ринок.

Неправильна гранулярність сервісів. Іноді архітектори створюють занадто дрібні сервіси, що призводить до “мережевої павутини”: запит користувача проходить через десятки сервісів, кожен додає затримку, і система стає неефективною.

Відсутність належного моніторингу. Є випадки, коли компанії впроваджували SOA, але не будували системи моніторингу. У результаті у випадку збою сервісу було неможливо швидко зрозуміти, де саме виникла проблема.

На перший погляд, різниця між монолітною та сервіс-орієнтованою архітектурою здається очевидною: у першій все об'єднано в одну систему, а в другій — розподілено на окремі сервіси. На практиці ця різниця має наслідки для розробки, супроводу, масштабування та навіть для бізнес-моделі компанії.

Таблиця 3.1. Порівняння монолітної та сервіс-орієнтованої архітектури

Критерій	Монолітна	SOA
Простота старту	Висока: один проєкт, швидке розгортання	Низька: потрібна інфраструктура (CI/CD, моніторинг)
Масштабованість	Масштабується лише вся система цілком	Можна масштабувати окремі сервіси
Надійність	Збій у модулі може вплинути на всю систему	Вихід з ладу сервісу не обов'язково “ламає” все
Розробка командою	Складно паралельно працювати великій кількості команд	Команди можуть працювати незалежно
Продовження на наступній сторінці		

Таблиця 3.1 – Продовження

Критерій	Монолітна	СОА
Продуктивність	Виклики швидкі, внутрішньо-програмні	Кожен виклик між сервісами — мережевий, із затримкою
Тестування	Легко протестувати як єдине ціле	Необхідні інтеграційні тести між сервісами
Гнучкість у виборі технологій	Обмежена одним стеком	Можна використовувати різні мови та фреймворки

Монолітна архітектура підходить для невеликих проєктів, стартапів, локальних рішень, де головне — швидко створити працюючий продукт і підтримувати його мінімальними силами. Вона проста, зручна на старті, але погано масштабується.

Сервіс-орієнтована архітектура є доцільною тоді, коли продукт орієнтований на тисячі чи мільйони користувачів, має складну бізнес-логіку та потребує гнучкого масштабування. Вона дозволяє розділити відповідальність між командами, але вимагає інвестицій у DevOps, інфраструктуру та моніторинг.

3.3 Мікросервісна архітектура

Мікросервісна архітектура з’явилася як подальший розвиток ідей сервіс-орієнтованої архітектури (SOA). Якщо у SOA під “сервісом” часто розумівся великий модуль, що виконував цілий спектр функцій, то мікросервіси стали відповіддю на потребу ще більшої автономності та гнучкості.

На початку 2010-х років великі ІТ-компанії, такі як Netflix, Amazon, eBay, стикнулися з проблемою: навіть сервісна архітектура виявлялася надто громіздкою, а розробка окремих модулів часто блокувалася залежностями від інших. У результаті виникла ідея подрібнити систему на набагато менші компоненти — так звані “мікросервіси”, кожен з яких виконує лише одну чітку бізнес-функцію і може розвиватися незалежно.

Мікросервісна архітектура — це підхід до проєктування програмних систем, у якому застосунок складається з набору невеликих незалежних сервісів. Кожен із них реалізує одну бізнес-можливість, часто має власну базу даних або сховище, а взаємодія між сервісами відбувається через чітко визначені інтерфейси (як правило, HTTP API, gRPC або черги повідомлень).

Основні характеристики мікросервісів:

Малі та автономні. Один мікросервіс виконує вузьку функцію, наприклад “автентифікація користувача”, “обробка платежів” чи “рекомендації товарів”.

Незалежність розробки. Кожна команда може працювати над своїм мікросервісом, не залежачи від інших.

Різноманітність технологій. Кожен мікросервіс може бути реалізований на різних мовах програмування і з різними базами даних.

Автоматизація. Для мікросервісів характерна потреба у DevOps-культури: CI/CD, контейнеризації (Docker), оркестрації (Kubernetes).

Наприклад, система онлайн-ритейлера, який продає товари по всьому світу, може складатися з сотень мікросервісів:

- один відповідає за каталог товарів,
- інший за пошук,
- окремий сервіс перевіряє наявність товару на складі,
- ще один — за обробку платежів,
- окремий сервіс формує рекомендації для користувачів.

Якщо потрібно масштабувати пошук (бо мільйони користувачів одночасно вводять запити), компанія може зробити це, не торкаючись інших сервісів — таких як сервісу платежів. Так само, мікросервіси платежів можуть бути окремими для різних країн (з банківських та податкових причин) і бути ізольованими одне від одного.

Інший приклад — Netflix — класичний приклад мікросервісної архітектури. Сервіси відповідають за різні функції: завантаження контенту, рекомендації, буферизацію відео, логування, білінг. Якщо один мікросервіс виходить з ладу, це не блокує роботу всієї системи, а користувач може навіть не помітити проблеми. У банківських застосунках кожна операція може бути окремим мікросервісом: відкриття рахунку, переказ грошей, перевірка кредитної історії. Це підвищує безпеку, оскільки злам одного мікросервісу не означає доступ до всієї системи.

Мікросервісна архітектура забезпечує низку переваг, що впливають безпосередньо з принципу розподілу функціональності на незалежні компоненти. Ці переваги стають особливо помітними у великомасштабних системах з високими вимогами до доступності та еволюційності:

Масштабованість. Можна масштабувати лише ті сервіси, які потребують цього.

Гнучкість у розробці. Команди можуть обирати власний стек технологій.

Надійність. Вихід з ладу одного мікросервісу не обов'язково паралізує всю систему.

Швидкий розвиток. Незалежні команди можуть одночасно додавати нові функції.

Легше впроваджувати інновації. Можна переписати один мікросервіс з нуля без впливу на решту системи.

Проте мікросервісна архітектура вносить істотну операційну та організаційну складність, що може перевищити її переваги у певних контекстах. Недоліки мікросервісів є прямим наслідком розподіленої природи архітектури та необхідності координації між незалежними компонентами:

Складність інфраструктури. Потрібні Kubernetes, Docker, системи моніторингу (Prometheus, Grafana).

Проблеми з інтеграцією. Взаємодія відбувається через мережу, що додає затримки й ризик відмов.

Збільшені витрати. Розробка, тестування й моніторинг вимагають більше ресурсів.

Організаційні виклики. Потрібно чітко визначати межі відповідальності між командами.

Складніше налагодження. Виявити проблему, яка виникає через взаємодію кількох мікросервісів, набагато важче, ніж у моноліті.

Мікросервіси можна уявити як екосистему невеликих підприємств. Кожне підприємство займається чимось одним: одне випікає хліб, інше ремонтує автомобілі, третє виробляє меблі. Усі вони незалежні, але разом формують економіку міста. Якщо одне підприємство тимчасово зупинило роботу, інші продовжують функціонувати.

У протиполог цьому, моноліт схожий на величезний комбінат, який виробляє одразу все: від їжі до автомобілів. У ньому простіше налагодити процеси, але якщо комбінат зупиниться, усе місто залишиться без товарів.

Однією з найбільш поширених помилок сучасних команд є переконання, що мікросервісна архітектура — це універсальне рішення для будь-якої задачі. Така позиція призводить до передчасного ускладнення проекту, перевитрат і, врешті-решт, до втрати швидкості розробки. Потрібно чітко розуміти: мікросервіси є ефективними лише в тих випадках, коли масштаб і складність системи виправдовують витрати на їх впровадження та підтримку.

Монолітна архітектура залишається найкращим вибором у ситуаціях, коли:

- продукт перебуває на ранній стадії розвитку і головна мета — швидко створити прототип (MVP);

- команда невелика (2–10 осіб), і всі можуть працювати в одній кодовій базі без суттєвих конфліктів;
- очікуване навантаження не надто велике і його можна масштабувати вертикально (потужнішими серверами);
- швидкість виходу на ринок важливіша за довгострокову гнучкість.

Приклад

Багато успішних стартапів починали як моноліти: Instagram, GitHub, Basecamp. Вони змогли швидко завоювати аудиторію, не витрачаючи ресурси на складну інфраструктуру. Лише після масштабного зростання постало питання переходу до більш складних архітектурних підходів.

Сервіс-орієнтована архітектура (SOA) може бути компромісним рішенням між монолітом та мікросервісами у випадках, коли:

- система стає надто великою, але повний перехід на мікросервіси ще не виправданий;
- організація вже має кілька великих модулів, які логічно розділити на сервіси;
- потрібна інтеграція з іншими корпоративними системами через стандартизовані протоколи (SOAP, ESB тощо);
- існує потреба у чіткій функціональній ізоляції (наприклад, облік користувачів відділений від бухгалтерського модуля).

Приклад

Великі корпоративні системи розробки ПЗ 2000-х (Oracle E-Business Suite, IBM WebSphere) розвивалися саме як SOA: модулі були достатньо незалежними, але залишалися відносно великими, а їхня взаємодія здійснювалася через корпоративну шину даних.

Мікросервісна архітектура є оптимальною у таких випадках:

- система обслуговує велику кількість користувачів і вимагає горизонтального масштабування;
- команда складається з десятків і сотень розробників, яких потрібно розділити на автономні групи;

- різні частини системи мають суттєво відмінні вимоги до продуктивності та технологій;
- продукт швидко розвивається, і важливо, щоб додавання нових функцій не блокувало інші команди;
- компанія вже має зрілу DevOps-культуру і ресурси для підтримки складної інфраструктури.

Приклад

Netflix, Spotify, Amazon — класичні історії успіху мікросервісної архітектури. Усі ці компанії вийшли на глобальний рівень завдяки можливості масштабувати сервіси незалежно, швидко розгортати нові функції та підтримувати роботу системи навіть при збоях окремих компонентів.

Вибір архітектури завжди має бути обґрунтованим. Моноліт підходить для швидкого старту, SOA — для середніх і великих систем із чіткою модульною структурою, а мікросервіси — для глобальних продуктів, що вимагають масштабованості, гнучкості й незалежності команд. Найбільша помилка — копіювати чужий досвід без урахування власного контексту.

3.4 Подійно-орієнтована архітектура

Подійно-орієнтована архітектура (Event-Driven Architecture, EDA) — це спосіб побудови програмних систем, у якому вся робота організована навколо подій. Компоненти не чекають, поки хтось їх “викличе напряму” (як у моноліті чи мікросервісах із REST API), вони реагують на події, які надходять у систему.

Подійний підхід виник не випадково. У традиційних системах (монолітах і навіть мікросервісах) часто виникала проблема синхронності: один компонент викликає інший і чекає на відповідь. Якщо відповідь затримується — система “зависає”.

Коли з’явилися розподілені системи (інтернет-магазини, банківські сервіси, соціальні мережі), стало очевидно: все відбувається надто швидко, а користувачів надто багато, щоб усі чекали одне на одного. Потрібна була модель, де все працює асинхронно — тобто “подія сталася, всі хто хоче на неї реагують, але основний процес не зупиняється”.

EDA складається з трьох ключових елементів:

Виробник подій (event producer). створює подію. Наприклад, кнопка “Купити” в інтернет-магазині.

Посередник (event broker). передає події тим, хто на них підписався. Це може бути черга повідомлень (RabbitMQ), система потокової обробки (Kafka) або хмарний сервіс (AWS EventBridge).

Споживач подій (event consumer). “підписані” на певні події і реагують на них. Це може бути сервіс обробки платежів, модуль доставки чи система повідомлень.

Таким чином, виробник не знає, хто саме обробить подію. Він просто “оголошує” факт, що щось сталося, а далі інші самі вирішують, як на це реагувати.

Приклад

Коли ви публікуєте фотографію в Instagram — це подія. Далі запускається ланцюг реакцій:

1. фото зберігається на сервері;
2. надсилається push-сповіщення вашим друзям;
3. система рекомендацій аналізує новий пост;
4. аналітика реєструє вашу активність.

Ви зробили одну дію, а ряд підсистем відреагували автоматично.

Подійно-орієнтована архітектура забезпечує ряд переваг, що впливають з асинхронної природи взаємодії між компонентами та слабкої зв’язності виробників і споживачів подій. Основні переваги EDA проявляються у системах з високими вимогами до масштабованості та гнучкості:

Масштабованість. Легко додавати нових “слухачів” подій без змін у “виробниках подій”.

Гнучкість. Система легко пристосовується до нових вимог: можна додати нову реакцію на подію без переписування інших частин.

Асинхронність. Компоненти не чекають відповіді, тому система працює швидше й витримує більше навантаження.

Відмовостійкість. Якщо один споживач події не працює, решта продовжують роботу.

Природність для реального світу. Усе в житті відбувається подійно: ми реагуємо на сигнали, а не чекаємо синхронних викликів.

Асинхронна та розподілена природа подійно-орієнтованої архітектури вносить специфічні складнощі у розробку, тестування та експлуатацію системи. Не-

доліки EDA стають особливо помітними у контекстах, що вимагають строгої послідовності операцій або детермінованої поведінки:

Складність відстеження. Події можуть породжувати інші події, і стає важко зрозуміти, що саме викликало певну дію.

Надмірна асинхронність. Якщо потрібно мати чіткий порядок виконання, подійна система може створити хаос.

Залежність від інфраструктури. Потрібні брокери подій, системи моніторингу, логування.

Тестування. Важко перевірити систему як єдине ціле, оскільки події можуть надходити у будь-якому порядку.

Реалізація подійно-орієнтованої архітектури вимагає відповідної інфраструктури для обробки, маршрутизації та зберігання подій. Існує кілька технологічних платформ, що надають різні підходи до управління потоками подій, кожна з яких характеризується специфічними архітектурними рішеннями та компромісами. Розглянемо основні платформи та їхнє застосування у виробничих системах.

Apache Kafka є розподіленою платформою потокової обробки подій, що реалізує модель журналу подій з можливістю багаторазового читання. Kafka використовується у великомасштабних системах, де критичними є пропускну здатність та можливість відтворення історії подій.

Архітектурно Kafka організує події у топіки (англ. topics) — іменовані категорії, до яких публікуються повідомлення. Кожен топик розбивається на частини (англ. partitions), що забезпечує горизонтальну масштабованість та паралельну обробку. Події в межах частин зберігаються у строгому хронологічному порядку та ідентифікуються послідовними зміщеннями (англ. offsets). Споживачі можуть читати події з довільного зміщення, що дозволяє повторно обробляти історичні дані або відновлювати стан після збоїв.

Модель зберігання подій у Kafka відрізняється від традиційних черг повідомлень: події не видаляються після прочитання, а зберігаються протягом визначеного періоду утримання (англ. retention period). Це дозволяє декільком незалежним споживачам обробляти один потік подій у різний спосіб та у різний час.

RabbitMQ реалізує модель брокера повідомлень, що базується на концепціях обмінників (англ. exchanges), черг (англ. queues) та зв'язків (англ. bindings). На відміну від Kafka, RabbitMQ орієнтований на гнучку маршрутизацію повідомлень між виробниками та споживачами з гарантіями доставки.

Виробники публікують повідомлення до обмінників, які маршрутизують їх до черг відповідно до визначених правил. RabbitMQ підтримує кілька типів обмінників: `direct` (маршрутизація за точним ключем), `topic` (маршрутизація за патерном), `fanout` (широкомовна розсилка) та `headers` (маршрутизація за атрибутами заголовків). Споживачі читають повідомлення з черг, після чого повідомлення зазвичай видаляються.

Приклад: інтернет-магазин

Подія створення замовлення публікується до обмінника типу `fanout`. Обмінник маршрутизує копії події до трьох незалежних черг: черги перевірки наявності товару, черги ініціювання платежу та черги відправлення повідомлень клієнту. Кожна черга обслуговується окремим сервісом. Якщо сервіс платежів тимчасово недоступний, повідомлення залишаються у черзі до відновлення сервісу. RabbitMQ забезпечує підтвердження обробки (`acknowledgments`), що запобігає втраті повідомлень при збоях споживачів.

AWS EventBridge є керованим сервісом для інтеграції застосунків через події. EventBridge автоматично отримує події з сервісів AWS, сторонніх SaaS-застосунків та користувацьких джерел, маршрутизуючи їх до цільових сервісів відповідно до визначених правил.

Правила EventBridge дозволяють фільтрувати події за атрибутами та спрямовувати їх до різних цілей: Lambda-функцій, черг SQS, топіків SNS, кластерів ECS або HTTP-ендпоінтів. Сервіс забезпечує автоматичне масштабування інфраструктури обробки подій, що усуває необхідність ручного управління брокерами повідомлень.

Приклад: система обробки зображень

Завантаження файлу до S3 генерує подію, яка автоматично надходить до EventBridge. Правило EventBridge фільтрує події за розширенням файлів (лише зображення) та викликає Lambda-функцію аналізу зображень. Результат аналізу публікується як нова подія, яка маршрутизується до Lambda-функції формування звітів. Фінальна подія спрямовується до SNS для розсилки результатів. Усі сервіси взаємодіють асинхронно через події без прямих викликів.

Подійно-орієнтована архітектура забезпечує слабку зв'язність компонентів та асинхронну взаємодію, що дозволяє незалежно масштабувати окремі частини

системи та підвищує відмовостійкість через ізоляцію збоїв. Архітектурний стиль EDA виявляється доцільним у системах з високою частотою подій, де множина споживачів обробляє один потік даних різними способами: платформи соціальних мереж, системи фінансових транзакцій, телеметрія пристроїв Інтернету речей. Проте впровадження подійно-орієнтованої архітектури вимагає інфраструктури для керування потоками подій (брокери повідомлень, системи моніторингу розподілених транзакцій), а також технічної компетентності команди розробки у сферах асинхронного програмування, обробки збоїв у розподілених системах та забезпечення узгодженості даних у умовах відкладеної обробки. Архітектурне рішення про застосування EDA має базуватись на аналізі співвідношення між перевагами слабкої зв'язності та операційними витратами на підтримку розподіленої асинхронної системи.

3.5 Безсерверна архітектура

Термін “безсерверна архітектура” (англ. *Serverless Architecture*) на перший погляд звучить дивно. Адже всі ми знаємо: будь-який застосунок зрештою працює на серверах. Як тоді може існувати архітектура “без серверів”?

Секрет у тому, що сервери все одно є, але розробники більше не зобов'язані ними керувати. У традиційних підходах команди повинні були думати про налаштування серверів, оновлення операційних систем, балансування навантаження, резервне копіювання тощо. У безсерверній моделі вся ця робота перекладається на хмарного провайдера (наприклад, Amazon, Google, Microsoft). Розробники пишуть лише функції та бізнес-логіку, а все інше відбувається автоматично.

У 2000-х роках більшість застосунків працювали на виділених серверах. Компанія купувала фізичну машину, встановлювала на неї систему, деплоїла код. Якщо користувачів ставало більше — доводилося купувати ще одну машину.

Згодом з'явилися **віртуальні машини**, які дозволили використовувати один фізичний сервер для кількох різнорідних непов'язаних навантажень. Це знизило витрати. Далі прийшла **контейнеризація** (Docker, Kubernetes), яка зробила розгортання ще гнучкішим.

При цьому все одно залишався тягар: хтось повинен був управляти цими контейнерами, слідкувати за їхнім масштабуванням і доступністю.

І ось у середині 2010-х компанії запропонували нову модель: “А навіщо взагалі думати про сервери? Давайте ви просто писатимете функції, а ми все решта зробимо за вас”. Так народилася ідея безсерверної архітектури. Одним із перших рішень став **AWS Lambda** (2014), а згодом з'явилися аналоги від Google (Cloud Functions), Microsoft (Azure Functions), IBM та інших.

У безсерверній архітектурі застосунок складається з набору окремих функцій. Кожна функція виконується у відповідь на подію: HTTP-запит, повідомлення у черзі, завантаження файлу, зміна запису в базі.

Ключові риси:

Виконання на вимогу. Функції “сплять”, поки не надійде подія. Тільки тоді хмара запускає код.

Автоматичне масштабування. Якщо подій стає більше — хмарний провайдер запускає більше копій функцій. Якщо подій мало — масштаб зменшується.

Оплата за використання. Ви платите не за сервери цілодобово, а лише за час, коли виконується ваша функція (наприклад, за мілісекунди роботи).

Подійна природа. Serverless часто використовується разом із подійно-орієнтованою архітектурою: функції реагують на події з інших сервісів.

Приклад

Уявіть автомат із кавою. У традиційній архітектурі ви повинні тримати баристу, навіть якщо за день заходить лише кілька клієнтів. У безсерверній моделі бариста “з’являється” тільки тоді, коли ви натискаєте кнопку. Він робить вам каву й зникає, поки знову не знадобиться.

Приклад

Коли користувач завантажує фото у хмару, подія запускає функцію. Вона змінює розмір зображення, додає водяний знак і зберігає у сховище.

До переваги безсерверної архітектури можна віднести:

Відсутність управління інфраструктурою. Розробники пишуть код і не думають про сервери.

Автоматичне масштабування. Не потрібно вручну додавати чи зменшувати кількість серверів.

Оптимізація витрат. Платите лише за час виконання функцій.

Швидкість розробки. Легко створювати нові сервіси, оскільки не треба налаштовувати інфраструктуру.

Ідеально для подійних систем. Добре інтегрується з EDA, IoT, аналітикою.

Напротивагу їм, безсерверна архітектура має ряд **недоліків та викликів**:

Vendor lock-in. Ви залежите від конкретного хмарного провайдера (AWS, Azure, Google). Перенести код на іншу платформу може бути складно.

Холодний старт. Якщо функція не виконувалась певний час, при першому запуску виникає затримка (кілька сотень мілісекунд).

Обмеження виконання. Функції мають ліміт часу (зазвичай від кількох секунд до 15 хвилин). Це не підходить для довготривалих процесів.

Складність налагодження. Важче тестувати й відслідковувати систему, де все побудовано на подіях і короткочасних функціях.

Витрати при високому навантаженні. Якщо функції виконуються мільярди разів, іноді дешевше мати власні сервери.

Отже, безсерверна архітектура — це наступний крок в еволюції програмних систем, що дає змогу зосередитися на бізнес-логіці, а не на інфраструктурі. Вона чудово підходить для систем, де важлива швидка реакція на події, гнучке масштабування й мінімальні витрати на утримання серверів. Проте, як і будь-який підхід, вона має свої обмеження й найкраще працює тоді, коли застосовується свідомо, з урахуванням контексту та потреб продукту.

3.6 Простір-орієнтована архітектура

Термін “простір-орієнтована архітектура” (англ. *Space-Based Architecture*, SBA) може звучати загадково. Насправді йдеться про особливу модель організації даних і процесів у розподіленій системі. Ключова ідея SBA — усунути “вузькі місця” традиційних систем, таких як бази даних чи централізованих сервісів, і забезпечити масштабування за рахунок розподіленого простору даних та обчислень.

Приклад

Якщо уявити традиційну архітектуру як місто, де все життя зосереджене в центрі (центральный ринок, центральный вокзал, головна площа), то SBA — це місто, де центру немає. Є багато маленьких площ, складів і доріг, які рівномірно розподілені. Люди можуть отримати необхідні товари й послуги в будь-якому районі, не створюючи черг у центрі.

SBA виникла як відповідь на проблему масштабування традиційних систем у 2000-х роках. У міру зростання навантаження більшість застосунків “упиралися” у базу даних. Навіть якщо додати більше серверів для обробки запитів, усі вони

продовжували тягнутися до однієї бази. Це створювало “пляшкове горло”, через яке система починала сповільнюватися.

Ідея була проста: а що, як замість одного централізованого сховища ми створимо розподілений “простір даних”, доступний одночасно для всіх компонентів? І що, як бізнес-логіку теж винесемо ближче до цього простору, аби обчислення відбувалися “на місці”, а не через віддалені запити? Так з’явився підхід SBA.

В основну SBA лягли наступні **принципи**:

Розподілений простір даних. Замість єдиної бази даних у системі існує набір вузлів, які зберігають дані в пам’яті. Цей простір єдина логічна сутність, але фізично він розподілений.

Обробка подій у просторі. Компоненти публікують події у простір, інші компоненти “слухають” їх і реагують.

Еластичність. Якщо навантаження зростає, у простір можна додати більше вузлів. Якщо навантаження падає — вузли можна прибрати.

Відсутність “центру”. Немає єдиної бази даних чи сервера, який би міг стати точкою відмови.

Збереження у пам’яті. Оскільки дані часто зберігаються в оперативній пам’яті вузлів, доступ до них відбувається дуже швидко.

Приклад

Під час “чорної п’ятниці” мільйони користувачів одночасно додають товари у кошики й роблять замовлення. У класичній архітектурі база даних може просто не витримати. У SBA ж усі дані про кошики зберігаються в розподіленому просторі, і кожен користувач працює з найближчим вузлом.

Приклад

Банки чи біржі обробляють сотні тисяч транзакцій щосекунди. SBA дозволяє виконувати перевірку транзакцій у пам’яті, розподіляючи навантаження по вузлах. Це гарантує швидкість і відмовостійкість.

До переваг SBA ми можемо віднести:

Масштабованість. Легко додати нові вузли у простір для обробки зростаючого навантаження.

Висока продуктивність. Дані зберігаються в оперативній пам'яті, доступ до них миттєвий.

Відмовостійкість. Вихід з ладу одного вузла не призводить до зупинки всієї системи.

Природна інтеграція з подійними системами. SBA добре поєднується з EDA.

Також, використання SBA йде з рядом **недоліків та викликів**:

Складність. Проектування SBA вимагає глибоких знань і розуміння розподілених систем.

Витрати на інфраструктуру. Необхідно мати достатньо серверів/вузлів, аби зберігати дані в пам'яті.

Синхронізація. Потрібно гарантувати узгодженість даних у розподіленому просторі.

Ніша застосування. SBA найкраще підходить для високонавантажених систем, але є надмірною для малих проєктів.

На практиці застосовується ряд існуючих рішень, такі як:

GigaSpaces XAP. Одна з найвідоміших платформ, що реалізує SBA.

Oracle Coherence. Рішення для розподіленого кешування й обчислень у просторі.

Hazelcast. Розподілений кеш і обчислювальний простір, який часто використовується в Java-проєктах.

Отже, простір-орієнтована архітектура — це спеціалізований, але потужний підхід, створений для високонавантажених і критично важливих систем. Її головна ідея — усунути “пляшкові горла” традиційних архітектур і забезпечити масштабованість за рахунок розподілених обчислень у пам'яті.

Якщо порівнювати SBA з іншими архітектурами, вона є природним вибором для фінансових систем, IoT, великих інтернет-магазинів і будь-яких рішень, де йдеться про обробку мільйонів подій у реальному часі.

Розділ 4

Реалізація архітектури

4.1 Масштабування

Масштабованість (англ. *scalability*) є фундаментальним атрибутом якості (див. підрозділ 1.1.2), що характеризує здатність системи адаптуватися до зростання навантаження шляхом додавання обчислювальних ресурсів, зберігаючи при цьому узгоджений рівень продуктивності. Навантаження може вимірюватися у різних ви- мірах, таких як кількість одночасних користувачів, обсяг оброблюваних даних, чи інтенсивність транзакцій. Важливо розрізняти масштабованість та продуктивність (англ. *performance*). Продуктивність вимірює ефективність системи (наприклад, час відгуку або пропускну здатність) за *поточного* навантаження, тоді як масшта- бованість описує *функцію* залежності продуктивності від зміни навантаження та ресурсів. Ідеальна система демонструє лінійну масштабованість, де подвоєння ресурсів призводить до подвоєння пропускну здатності за незмінного часу відгуку. Однак на практиці лінійна масштабованість є майже недосяжною через накладні витрати на координацію, конкуренцію за спільні ресурси (англ. *contention*) та забезпечення когерентності даних.

Для формального моделювання меж масштабованості часто застосовується Універсальний закон масштабованості (Universal Scalability Law, USL), запропонований Нейлом Гюнтером. USL описує пропускну здатність C системи з N вузлами за формулою:

$$C(N) = \frac{N}{1 + \alpha(N - 1) + \beta N(N - 1)} \quad (4.1)$$

де α — коефіцієнт конкуренції (блокування), а β — коефіцієнт когерентності (витрати на синхронізацію розподіленого стану). Ця модель математично обґрунтовує, чому додавання нових вузлів зрештою призводить до зменшення приросту продуктивності і навіть до її деградації (закон спадної віддачі). Таким чином, архітектурні рішення мають бути спрямовані на мінімізацію коефіцієнтів α та β .

4.1.1 Стратегії масштабування: вертикальне та горизонтальне

Існує дві фундаментальні стратегії додавання ресурсів для масштабування системи:

Вертикальне масштабування полягає у збільшенні потужності *окремого вузла* (сервера) шляхом додавання ресурсів, таких як потужніший центральний процесор (CPU), збільшення обсягу оперативної пам'яті (RAM) або використання швидших систем зберігання даних (наприклад, перехід з HDD на SSD). Перевагою цього підходу є відносна простота, оскільки він зазвичай не вимагає суттєвих змін в архітектурі програмного забезпечення. Однак вертикальне масштабування має жорсткі обмеження: існує фізична та економічна "стеля" потужності одного вузла, а вартість високопродуктивного обладнання зростає нелінійно. Крім того, цей підхід не вирішує проблему єдиної точки відмови (Single Point of Failure, SPOF), про яку йдеться у розділі 4.2.

Горизонтальне масштабування передбачає додавання *більшої кількості вузлів* (серверів, контейнерів) до системи, які працюють паралельно для розподілу навантаження. Це домінуючий підхід у сучасних хмарних (*cloud-native*) та розподілених системах. Головною перевагою є висока еластичність та теоретично необмежений потенціал зростання шляхом додавання стандартизованих, відносно недорогих вузлів. Водночас горизонтальне масштабування вносить значну архітектурну складність, оскільки вимагає механізмів балансування навантаження, управління розподіленим станом, забезпечення узгодженості даних та координації між вузлами.

Для більш глибокого розуміння багатовимірної природи горизонтального масштабування часто використовується "Куб Масштабованості" (*Scalability Cube*), популяризований у праці "The Art of Scalability". Ця модель визначає три ортогональні осі, вздовж яких може розвиватися система:

- **Вісь X (Горизонтальне дублювання):** Це найпоширеніша форма масштабування (scale-out), що полягає у клонуванні повного екземпляра застосунку та

розподілі запитів між цими копіями за допомогою балансувальника навантаження. Кожен клон є ідентичним і обробляє частину загального потоку запитів.

- **Вісь Y (Функціональна декомпозиція):** Ця вісь передбачає розбиття системи за функціями або сервісами. Замість єдиного моноліту (див. розділ 3.1) система декомпозується на набір незалежних сервісів (наприклад, сервіс автентифікації, сервіс замовлень, сервіс каталогів), що відповідає архітектурним стилям SOA (розділ 3.2) та мікросервісів (розділ 3.3). Це дозволяє масштабувати кожну функцію незалежно; наприклад, якщо сервіс пошуку зазнає високого навантаження, можна додати вузли лише для нього, не впливаючи на сервіс платежів.
- **Вісь Z (Партиціонування даних):** Ця вісь передбачає розподіл даних між вузлами. Кожен вузол (або екземпляр сервісу) відповідає лише за *підмножину* даних (наприклад, партиціонування (*sharding*) бази даних за ідентифікатором користувача або регіоном). Це дозволяє масштабувати системи з величезними обсягами даних, оскільки кожен вузол оперує меншим набором даних.

Зрілі, високонавантажені системи, як правило, використовують комбінацію всіх трьох осей масштабування одночасно для досягнення максимальної продуктивності та еластичності.

Для реалізації стратегій масштабування, особливо горизонтального, архітектор має застосовувати специфічні тактики проектування (див. розділ 2.1). Ключовою передумовою для масштабування по осі X є проектування компонентів **без стану (Stateless Component Design)**. Якщо компонент зберігає стан сесії локально (наприклад, у пам'яті процесу), балансувальник навантаження не може вільно перенаправляти послідовні запити від одного клієнта на різні вузли. Архітектурна тактика полягає у *екстерналізації стану* — винесенні сесійних даних до спільного сховища, доступного всім вузлам (наприклад, розподілений кеш як Redis або Memcached, або база даних).

Для реалізації масштабування по осі Y (функціональна декомпозиція) ключовою тактикою є **асинхронна комунікація** з використанням черг повідомлень (наприклад, RabbitMQ, Kafka), що детально розглядається у розділі 3.4 (Подійно-орієнтована архітектура). Асинхронність розриває прямі залежності між сервісами, дозволяючи виробникам (*producers*) та споживачам (*consumers*) повідомлень масштабуватися незалежно та обробляти тимчасові піки навантаження без блокування всієї системи. Додатковою важливою тактикою є **кешування (Caching)**, яке зменшує навантаження на базові сховища даних (розділ 4.5) шляхом зберігання часто запитуваних даних у швидкому шарі (наприклад, *in-memory*).

У більшості розподілених систем саме сховище даних стає головним вузьким місцем. Для масштабування баз даних застосовують дві основні тактики, що мають фундаментальні архітектурні наслідки:

1. **Реплікація (Replication):** Створення копій бази даних. Зазвичай використовується модель "Primary-Replica" (Лідер-Послідовник), де всі операції запису (INSERT, UPDATE, DELETE) йдуть на основний вузол (Primary), а потім асинхронно копіюються на вузли-репліки (Replicas). Це дозволяє горизонтально масштабувати операції *читання* (READ), розподіляючи їх між численними репліками. Компромісом цього підходу є **кінцева узгодженість (eventual consistency)**. Через затримку реплікації (replication lag) дані, прочитані з репліки, можуть бути застарілими.
2. **Партиціонування (Partitioning або Sharding):** Відповідає осі Z "Куба Масштабованості". Дані фізично розділяються на частини (шарди) і розподіляються між різними вузлами. Кожен вузол відповідає лише за свою частину даних. Цей підхід є єдиним дієвим способом масштабування операцій *запису* (WRITE) у великих системах. Однак партиціонування суттєво ускладнює архітектуру: запити, що вимагають даних з кількох шардів (cross-shard queries), стають складними та неефективними, а забезпечення транзакційної цілісності (ACID) для операцій, що охоплюють декілька шардів, стає неможливим і вимагає застосування складних патернів, таких як Saga.

Вибір між цими тактиками та прийняття компромісу між узгодженістю та доступністю є центральною проблемою проектування розподілених систем, що детально розглядається у контексті **теореми CAP** у розділі 4.6.

4.2 Відмовостійкість

Відмовостійкість (англ. *Fault Tolerance*) — це здатність системи продовжувати коректну роботу, можливо на деградованому рівні, за наявності апаратних або програмних відмов (faults). Це поняття є фундаментальним для побудови надійних (Reliable) та високодоступних (Available) систем, як визначено у підрозділі 1.1.2. Важливо розрізняти таксономію збоїв: **відмова (fault)** — це дефект у компоненті (наприклад, програмна вада або збій диска); **помилка (error)** — це некоректний внутрішній стан системи, спричинений відмовою; **збій (failure)** — це зовнішня, спостережувана девіація поведінки системи від її специфікації. Архітектура, що є відмовостійкою, має на меті розірвати цей ланцюг: вона повинна запобігати перетворенню відмов на помилки, або помилок — на системні збої.

У той час як доступність обраховується за формулою:

$$A = \frac{MTBF}{MTBF + MTTR} \quad (4.2)$$

Формула (4.2) визначає кількісно визначає частку часу, коли система є операційною, відмовостійкість описує *методи*, за допомогою яких ця доступність досягається. Головна мета відмовостійкості — максимізувати час між відмовами (MTBF) та мінімізувати час відновлення (MTTR) шляхом автоматичного реагування на збої без людського втручання.

Для досягнення відмовостійкості застосовується набір архітектурних тактик, які можна згрупувати у три категорії, частково описані у підрозділі 2.1.1 як тактики доступності. Тут ми розглядаємо їх у контексті повноцінної реалізації:

1. **Виявлення відмов (Fault Detection):** Система не може відреагувати на відмову, яку вона не виявила. Тактики включають:
 - **Heartbeat (Пульс):** Компоненти періодично надсилають повідомлення "я живий" монітору. Відсутність повідомлення впродовж певного часу сигналізує про відмову. Це розширення тактики ping/echo;
 - **Voting (Голосування):** Кілька надлишкових компонентів (реплік) виконують однакове обчислення. Їхні результати порівнюються. Якщо один результат відрізняється, цей компонент вважається несправним;
 - **Watchdog Timer (Сторожовий таймер):** Зовнішній процес очікує на періодичний "скид" від компонента. Якщо компонент "зависає" і не скидає таймер, він примусово перезапущається.
2. **Відновлення після відмов (Fault Recovery):** Після виявлення відмови система має відновити коректний стан.
 - **Надлишковість (Redundancy):** Це ключова тактика, що передбачає наявність дублюючих компонентів. Вона реалізується у формах: *Active redundancy* (гарячий резерв), де всі репліки обробляють запити паралельно; *Passive redundancy* (теплий/холодний резерв), де резервний компонент активується (відбувається *failover*) лише після відмови основного;
 - **Graceful Degradation (Контрольована деградація):** Замість повного збою, система відключає менш критичні функціональності, щоб зберегти працездатність основних. Наприклад, сайт може відключити рекомендації, але залишити можливість покупки;

- **Checkpointing (Збереження стану):** Періодичне збереження стану системи, що дозволяє відновити роботу з останньої відомої коректної точки замість повного перезапуску.
3. **Запобігання відмовам (Fault Prevention):** Тактики, спрямовані на уникнення відмов.
- **Removal from Service (Виведення з експлуатації):** Тимчасове виведення компонента з-під навантаження для оновлення або профілактики;
 - **Transactions (Транзакції):** Використання ACID-транзакцій для гарантування атомарності операцій, що запобігає виникненню некоректних, частково змінених станів.

Архітектурні тактики комбінуються у високорівневі патерни проєктування, що забезпечують стійкість розподілених систем:

- **Circuit Breaker (Автоматичний вимикач):** Цей патерн запобігає каскадним відмовам. Він діє як програмний "запобіжник" що моніторить виклики до зовнішнього сервісу. Якщо кількість відмов перевищує поріг, вимикач "розмикається" (стан *Open*), і всі наступні виклики миттєво завершуються помилкою, не створюючи навантаження на сервіс, що, ймовірно, відмовив. Через певний час вимикач переходить у стан *Half-Open*, пропускаючи тестовий запит. Якщо він успішний, вимикач "закривається" (стан *Closed*); якщо ні — залишається розімкненим;
- **Bulkhead (Ізоляція відсіків):** Патерн, названий за аналогією з перебірками у корпусі корабля. Він ізолює ресурси (наприклад, пули потоків, пули з'єднань) для різних частин системи. Якщо одна функціональність (наприклад, запит до повільного зовнішнього API) вичерпує всі потоки у своєму пулі, це не впливає на потоки, виділені для інших, критичних функціональностей. Це запобігає повному вичерпанню ресурсів системи через відмову одного компонента;
- **Retry (Повторні спроби):** У розподілених системах поширені тимчасові відмови (наприклад, збій мережі). Патерн *Retry* передбачає повторне виконання операції, що завершилася збоєм. Критично важливо поєднувати його з **Exponential Backoff** (експоненційною затримкою) — збільшенням інтервалу між повторними спробами, щоб не перевантажити сервіс, який, можливо, намагається відновитися. Повторні спроби безпечно застосовувати лише до **ідемпотентних** операцій (операцій, повторне виконання яких не змінює результат).

У розподілених системах досягнення відмовостійкості нерозривно пов'язане з компромісами, що формалізовані у **Теоремі CAP** (див. розділ 4.6). Теорема стверджує, що неможливо одночасно гарантувати Узгодженість (Consistency), Доступність (Availability) та Стійкість до розділення мережі (Partition tolerance). Оскільки мережеві розділення (відмови) є неминучим фактом у розподілених системах, архітектор змушений обирати між узгодженістю (CP) та доступністю (AP) під час відмови.

Мікросервісні (розділ 3.3) та подійно-орієнтовані (розділ 3.4) архітектури за своєю природою проєктуються для стійкості до розділення. Вони часто обирають доступність (AP), жертвуючи сильною узгодженістю на користь **кінцевої узгодженості** (Eventual Consistency). На противагу, традиційні монолітні системи (розділ 3.1) з єдиною реляційною базою даних (ACID) часто обирають узгодженість (CP) — у разі втрати зв'язку з базою даних система стає недоступною, але гарантує коректність даних.

Перевірка коректності реалізації відмовостійкості є нетривіальною задачею. Недостатньо просто протестувати "щасливий шлях" (happy path). **Хаос-інжиніринг** (Chaos Engineering) — це дисципліна експериментального тестування розподіленої системи з метою побудови впевненості у її здатності витримувати турбулентні умови в продакшені. Замість очікування на реальні відмови, інженери проактивно вносять їх у контрольоване середовище.

Канонічним прикладом є інструмент *Chaos Monkey*, розроблений Netflix, який у робочий час випадковим чином зупиняє віртуальні машини та контейнери у *виробничому* (production) середовищі. Мета — не "зламати" систему, а перевірити, чи спрацювають автоматичні механізми відновлення (наприклад, failover, перезапуск вузлів, балансування навантаження) так, як очікувалось, і чи залишається система доступною для користувачів.

4.3 Безпека

Безпека характеризує здатність ПЗ захищати дані та інформацію від неавторизованого доступу при забезпеченні доступу авторизованим користувачам. Архітектурні рішення щодо безпеки впливають на всі аспекти проєктування системи — від вибору протоколів автентифікації до організації взаємодії між компонентами.

4.3.1 Управління ідентифікацією та доступом

Управління ідентифікацією (англ. Identity Management, IdM) (також — управління ідентифікацією та доступом, Identity and Access Management, IAM) є методом верифікації ідентичності мережевих сутностей та визначення

рівня доступу до ресурсів підприємства. Основною метою управління ідентифікацією є забезпечення того, що лише автентифіковані користувачі — як фізичні особи, так і пристрої — отримують доступ до конкретних застосунків, компонент та систем, для яких вони авторизовані.

Управління ідентифікацією базується на трьох базових концепціях:

Ідентифікація є здатністю однозначно визначити користувача, пристрій або застосунок у мережі підприємства на основі його атрибутів. Прикладами ідентифікаторів є імена користувачів, ідентифікатори процесів, електронні адреси та табельні номери співробітників. Системи безпеки використовують цю ідентичність при визначенні, чи може суб'єкт отримати доступ до об'єкта.

Автентифікація є процесом верифікації ідентичності, заявленої мережевою сутністю, на основі її облікових даних. Для верифікації можуть використовуватися три фактори:

1. Фактор знання (англ. knowledge factor) — базується на чомусь, що користувач знає, наприклад, пароль або PIN-код;
2. Фактор володіння (англ. ownership factor) — базується на предметі, який має користувач, наприклад, ідентифікаційна картка, смарт-картка або токен безпеки;
3. Фактор властивості (англ. inherence factor) — базується на невідчужуваному атрибуті користувача, такому як відбитки пальців або інші біометричні дані.

Авторизація є процесом надання доступу до мережевих ресурсів для конкретної сутності або типу ідентичності користувача у спосіб, що узгоджується з політиками та управлінням підприємства. Автентифікація встановлює, ким є користувач, а авторизація встановлює, що цей користувач може робити. Для підтримки безпечного мережевого середовища автентифікація має відбуватися перед авторизацією.

У контексті організації управління ідентифікацією реалізується через централізовану систему (постачальника ідентичності), що забезпечує єдину точку автентифікації для всіх систем у корпоративному домені (рис. 4.1).

Постачальник ідентичності (англ. Identity Provider, IdP) створює, підтримує та керує інформацією про ідентичність, а також може надавати послуги автентифікації іншим застосункам.

Типовими реалізаціями постачальників ідентичності для організацій є:

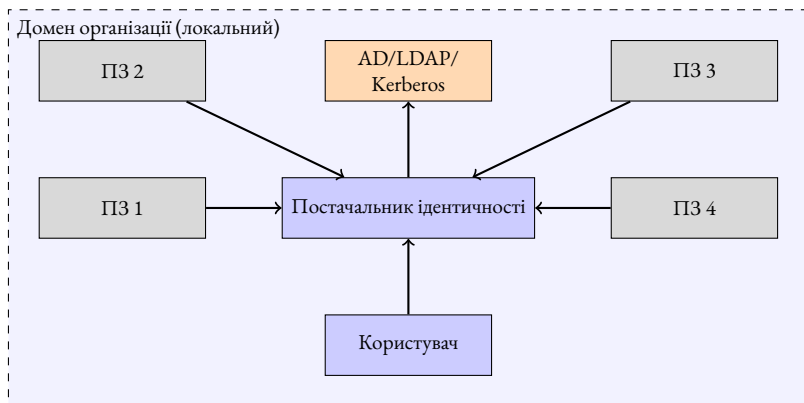


Рис. 4.1. Місце постачальника ідентичності в локальній інфраструктурі організації

- LDAP (Lightweight Directory Access Protocol);
- Доменні користувачі на основі Kerberos;
- Active Directory (AD).

У випадку, коли системи організації частково розміщені у хмарному середовищі або є необхідність надати доступ до зовнішніх систем партнерів (рис 4.2), застосовується федеративне управління ідентифікацією.

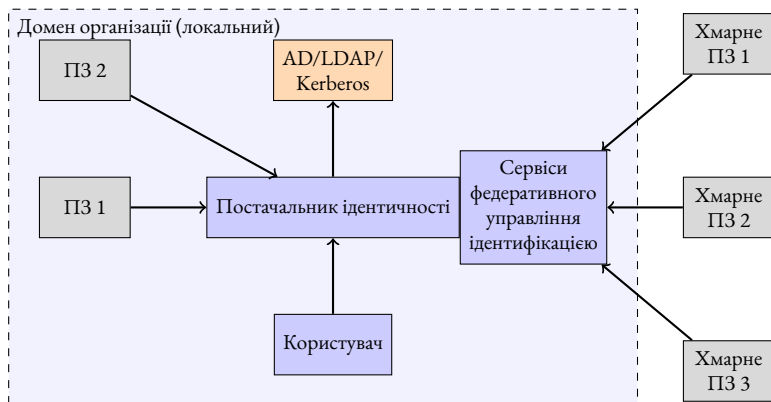


Рис. 4.2. Використання федеративного управління ідентифікацією в організації

Федерована ідентичність (англ. Federated Identity) є методом зв'язування ідентичності користувача між кількома окремими системами управління

ідентифікацією. Федеративна ідентичність дозволяє користувачам швидко переміщуватися між системами, зберігаючи безпеку.

У випадках коли компанія переважно користується хмарним програмним забезпеченням найчастіше використовуються хмарні системи *єдиного входу* (англ. Single Sign-On (SSO)) (рис. 4.3).

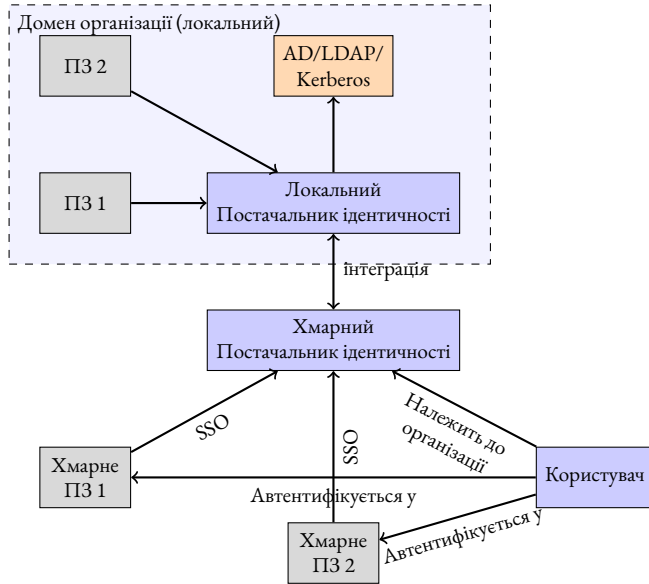


Рис. 4.3. Використання системи єдиного входу в організації

Єдиний вхід (англ. Single Sign-On, SSO) є механізмом автентифікації, що дозволяє користувачу автентифікуватися один раз та отримувати доступ до множини систем без необхідності повторної автентифікації.

При проектуванні системи розробники мусять вирішити низку питань щодо управління ідентифікацією:

- Профіль користувача — як користувачі реєструються у системі;
- Автентифікація — хто відповідає за автентифікацію користувачів;
- SSO для федеративних користувачів;
- Облікові дані для керованих користувачів:
 - Хто зберігає облікові дані користувачів;

- Як користувачі автентифікуються;
- Доступ користувачів:
 - Як користувач отримує дозволи;
 - Яку модель контролю доступу використовувати.

Існує три основних підходи до реалізації управління ідентифікацією для програмного забезпечення:

Внутрішнє управління ідентифікацією передбачає реалізацію функціональності управління ідентифікацією безпосередньо у застосунку. Цей підхід може базуватися на використанні бібліотек або на повністю власній реалізації. Переваги: зменшення залежностей від зовнішніх систем, повний контроль над реалізацією. Недоліки: ризики помилок у власній реалізації, необхідність реалізації повного спектру функціональності (включаючи багатофакторну автентифікацію, контроль доступу), ризик неправильних архітектурних рішень, додаткові проблеми при проходженні аудитів.

Готові рішення (англ. out-of-the-box solutions) — використання спеціалізованого програмного забезпечення для управління ідентифікацією, що розгортається та підтримується організацією самостійно. Прикладами є Keycloak та WSO2. Переваги: готова функціональність, безкоштовні продукти з відкритим кодом, можливість розширення. Недоліки: необхідність самостійного управління хостингом, оновлення до нових версій, міграція власних компонент при оновленні версії продукту.

Управління ідентифікацією як послуга (англ. Identity-as-a-Service, IDaaS) — використання хмарних сервісів для управління ідентифікацією. Прикладами є AWS Cognito, Auth0 by Okta, Org Kratos. Переваги: готове рішення, керований хостинг, висока доступність. Недоліки: витрати на використання (необхідність розрахунку вартості), залежність від постачальника послуг.

Вибір підходу до управління ідентифікацією є одним з найважливіших архітектурних рішень на початку проекту, оскільки від нього залежить безпека системи, складність розробки та підтримки, а також можливості інтеграції з іншими системами.

Питання авторизації користувачів також є важливим при плануванні безпечних рішень. Контроль доступу визначає, які дії користувачі можуть виконувати з ресурсами системи після автентифікації. Існує декілька моделей контролю доступу, кожна з яких має свої характеристики та сфери застосування.

Контроль доступу на основі розсуду (англ. Discretionary Access Control, DAC) базується на визначенні доступу власником цільового ресурсу. Власник ресурсу самостійно вирішує, хто має та не має доступу, та точно визначати, який саме доступ дозволений.

Контроль доступу на основі обов'язку (англ. Mandatory Access Control, MAC) переносить відповідальність з власника ресурсу на окрему сутність, групу чи особу що має окремі повноваження встановлювати права доступу до ресурсів.

Контроль доступу на основі ролей (англ. Role-Based Access Control, RBAC) є розширенням MAC, у якому уповноважена сутність чи особа замість гранулярного розподілення доступів кожному користувачеві визначає ролі, їх права доступу до ресурсів та призначає користувачам доступи через присвоєння їм цих ролей.

Атрибутний контроль доступу (англ. Attribute-Based Access Control, ABAC) базується на оцінці атрибутів для прийняття рішень про доступ. Атрибути можуть належати до користувача (суб'єкта), ресурсу, дії або середовища. Правила доступу в ABAC формуються як булеві вирази, що оцінюють комбінації цих атрибутів для визначення, чи слід надати доступ. Правила можуть бути як локальними (щодо одного ресурсу) так і глобальними (для усіх ресурсів). На відміну від RBAC, де доступ визначається виключно роллю користувача, ABAC надає гнучкість у визначенні контекстно-залежних правил доступу, що враховують множину факторів одночасно.

Рольовий контроль доступу складається з наступних елементів:

Адміністратори ідентифікують ролі, надають дозволи та підтримують системи безпеки.

Роль групує працівників на основі завдань, які вони виконують.

Дозволи визначають доступ та дії, прикріплені до кожної ролі, та окреслюють, що користувачі можуть та не можуть робити.

RBAC є придатним для організацій з чітко визначеною структурою ролей та передбачуваними патернами доступу. Модель забезпечує простоту управління правами доступу через призначення користувачів до ролей.

ABAC надає більшу гнучкість порівняно з RBAC, дозволяючи визначати права доступу на основі множини атрибутів:

Суб'єкт — атрибути користувача (посада, відділ, рівень допуску тощо).

Ресурс — атрибути ресурсу (тип файлу, рівень конфіденційності, власник тощо).

Дія — операція, що виконується (читання, редагування, видалення тощо).

Середовище — контекстуальні атрибути (час, місцезнаходження, протокол комунікації тощо).

АВАС є придатним для складних програмних засобів з динамічними вимогами до контролю доступу, де рольова модель є занадто жорсткою. Проте АВАС вимагає більш складної конфігурації та підтримки порівняно з RBAC.

При інтеграції програмного забезпечення з системою управління ідентифікацією організації виникає питання надання доступу новим користувачам. Існує два основних підходи:

- Типові ролі для нових користувачів — кожен новий користувач автоматично отримує певний набір базових ролей;
- Призначення ролей на основі claims у токени ідентичності — система аналізує атрибути користувача, надані постачальником ідентичності, та на їх основі призначає відповідні ролі.

Другий підхід дозволяє централізовано керувати доступом користувачів до множини систем організації без необхідності ручного налаштування ролей у кожній системі окремо.

Рішення про авторизацію користувача на основі будь-якого з підходів до контролю доступу може виконуватись на різних прошарках програмного забезпечення:

Авторизація на рівні застосунку — застосунок знає про специфічні дозволи та приймає рішення про авторизацію самостійно. Цей підхід надає застосунку повний контроль над логікою авторизації та дозволяє реалізувати складні правила доступу, специфічні для бізнес-логіки.

Авторизація на рівні сервера авторизації — сервер авторизації обмежує доступ до ресурсів на основі загальних правил. Цей підхід централізує управління правилами доступу, але може бути недостатньо гнучким для складних сценаріїв бізнес-логіки.

Обидва підходи є правомірними, та вибір між ними залежить від специфіки системи. У складних системах часто застосовується комбінований підхід: сервер авторизації виконує базову перевірку прав доступу, а застосунок додатково перевіряє специфічні для бізнес-логіки умови.

4.3.2 Безпека у різних архітектурних стилях

Архітектурний стиль програмного забезпечення суттєво впливає на підходи до забезпечення безпеки та управління ідентифікацією.

У монолітній архітектурі безпека традиційно базується на серверних сесіях:

1. Користувач автентифікується, надаючи облікові дані чи користуючись SSO;
2. Сервер створює сесію та зберігає її у сховищі стану сесій;
3. Сервер надсилає ідентифікатор сесії клієнту;
4. Клієнт включає cookie чи HTTP-заголовок з ідентифікатором сесії в кожен наступний запит;
5. Сервер використовує ідентифікатор сесії для отримання інформації про автентифікованого користувача.

Цей підхід є простим у реалізації для монолітних застосунків, оскільки всі компоненти мають доступ до спільного сховища стану. Проте він створює труднощі при масштабуванні та переході до розподілених архітектур — чи навіть просто при горизонтальному масштабуванні моноліту.

Однак, при використанні мікросервісів використання звичайного ідентифікатора сесії ускладнене, оскільки відсутній спільний між сервісами стан, аж до відокремлення баз даних. Між мікросервісами може бути утворене спільне сховище сесій, однак дані рішення мають проблему з масштабуванням — при кожному запиті мікросервіс мусить переконатись у наявності даної сесії в сховищі та видобути пов'язані з нею дані. При цьому, використання реплікації для сховища сесій ускладнене — якщо нова сесія не встигла “дійти” до репліки, мікросервіс який користується даною реплікою відмовить користувачу у виконанні запиту, який мав би виконатись. При цьому, розділення сховища сесій на декілька реплік що містять різні дані також ускладнене, оскільки при кожному запиті ідентифікатор сесії *мусить* бути знайдений в сховищі.

Для вирішення цих проблем застосовуються токени:

Токен є самодостатньою одиницею інформації, що містить дані про ідентичність користувача та його права доступу. Токени зберігаються на стороні клієнта та надсилаються з кожним запитом до сервера.

Використання токенів дозволяє:

- Уникнути необхідності спільного сховища стану між сервісами;
- Уникнути необхідності зберігання стану сесії на стороні сервера взагалі;
- Масштабувати сервіси незалежно один від одного.

Робота з токенами може бути реалізована в різних частинах програмного забезпечення:

1. API Gateway відповідає за централізовану автентифікацію

У цьому підході API Gateway виконує автентифікацію користувачів та передає токени внутрішнім сервісам. Сервіси довіряють Gateway та не виконують власну автентифікацію.

Критика підходу:

- Відсутнє повноцінне управління ідентифікацією — лише автентифікація;
- Потенційна можливість використання API Gateway для прийняття всіх рішень про авторизацію, що призводить до порушення принципу нульової довіри (Zero Trust), оскільки внутрішні точки доступу залишаються незахищеними.

2. Окремий мікросервіс управління ідентифікацією

У цьому підході створюється спеціалізований мікросервіс, відповідальний за управління ідентифікацією, автентифікацію та видачу токенів. Інші мікросервіси перевіряють токени та приймають рішення про авторизацію самостійно.

Переваги:

- Повноцінне управління ідентифікацією;
- Кожен сервіс перевіряє токени та контролює доступ;
- Відповідність принципу нульової довіри.

3. API Gateway + мікросервіс управління ідентифікацією

Комбінований підхід, у якому:

- Мікросервіс управління ідентифікацією відповідає за автентифікацію та видачу токенів;
- API Gateway виконує базову перевірку токенів та маршрутизацію запитів;
- Кожен мікросервіс додатково перевіряє токени та виконує специфічні для нього правила авторизації.

Цей підхід є найбільш збалансованим та рекомендованим для складних систем.

Альтернативно, робота з токенами може бути виконана на рівні *сервісної сітки*.

Сервісна сітка (англ. service mesh) є виділеним інфраструктурним рівнем для управління комунікацією між сервісами у мікросервісній архітектурі. Service

mesh реалізується через розгортання проксі-серверів (зазвичай у вигляді *sidecar*-контейнерів) поряд з кожним екземпляром мікросервісу, що перехоплюють весь вхідний та вихідний мережевий трафік сервісу. Ця архітектура абстрагує міжсервісну комунікацію від реалізації самого мікросервісу та дозволяє централізовано управляти маршрутизацією, балансуванням навантаження, відмовостійкістю, спостережуваністю та безпекою комунікації без модифікації коду самих сервісів.

Service mesh надає додатковий рівень абстракції для управління комунікацією між мікросервісами та може централізувати певні аспекти безпеки:

- Автентифікація та авторизація на рівні service mesh, незалежно від мови програмування чи фреймворку сервісів;
- Шифрування трафіку між сервісами (mutual TLS);
- Централізоване управління політиками доступу.

Service mesh є корисним у гетерогенних середовищах, де сервіси написані різними мовами програмування, оскільки надає єдиний механізм безпеки для всіх сервісів.

Serverless-архітектура вимагає специфічних підходів до управління безпекою:

- API Gateway виконує автентифікацію користувачів;
- Lambda-функції або інші serverless-компоненти виконують авторизацію на основі токенів;
- Використання спеціалізованих сервісів управління ідентифікацією (наприклад, AWS Cognito для AWS Lambda).

У serverless-архітектурі API Gateway часто приймає більше рішень щодо авторизації порівняно з іншими архітектурними стилями, оскільки це дозволяє уникнути виклику функцій для запитів, що не мають необхідних прав доступу, та зменшити витрати.

4.3.3 Стандарти автентифікації та авторизації

Для забезпечення сумісності та безпеки розроблено низку стандартів автентифікації та авторизації.

JSON Web Token (JWT) є відкритим стандартом, що визначає компактний та самодостатній спосіб безпечної передачі інформації між сторонами у вигляді JSON-об'єкта. JWT можуть бути верифіковані та є довіреними, оскільки вони цифрово підписані.

JWT складається з трьох частин, розділених крапками:

- **Заголовок** (Header) містить тип токена та алгоритм підпису;
- **Корисне навантаження** (Payload) містить claims — твердження про сутність та додаткові дані;
- **Підпис** (Signature) використовується для верифікації відправника JWT та забезпечення цілісності повідомлення.

Безпека JWT забезпечується наступними механізмами:

- Токен підписується приватним ключем при створенні;
- Токен верифікується публічним ключем при отриманні;
- Токен може бути зашифрований для додаткової конфіденційності.

JWT є особливо придатним для розподілених систем, оскільки дозволяє верифікувати токени без звернення до централізованого сховища.

OAuth 2.0 є протоколом делегування для доступу до API та є індустрічним стандартом для управління ідентифікацією та доступом. Відкритий протокол авторизації OAuth 2.0 дозволяє застосунку отримувати доступ до ресурсів, розміщених іншими вебзастосунками, від імені користувача без надання користувачем своїх облікових даних.

Основні учасники OAuth 2.0:

- Клієнт (Client) — застосунок, що запитує доступ;
- Сервер ресурсів (Resource Server) — сервер, що зберігає захищені ресурси;
- Сервер авторизації (Authorization Server/OAuth Server) — сервер, що видає токени доступу.

Типовий потік OAuth 2.0:

1. Клієнт запитує токен доступу у сервера авторизації;
2. Сервер авторизації верифікує клієнта та отримує згоду користувача;
3. Сервер авторизації видає токен доступу клієнту;
4. Клієнт використовує токен доступу для запиту ресурсу;
5. Сервер ресурсів перевіряє токен та надає доступ до ресурсу.

OAuth 2.0 визначає два типи токенів:

Токен доступу (Access Token) є короткостроковим токеном, що надає доступ до захищених ресурсів. Типова тривалість життя токена доступу — від декількох хвилин до декількох годин.

Токен оновлення (Refresh Token) є довгостроковим токеном, що використовується для отримання нових tokenів доступу без повторної автентифікації користувача. Токен оновлення зберігається клієнтом та не надсилається до сервера ресурсів.

Використання tokenів оновлення дозволяє:

- Обмежити час життя tokenів доступу для зменшення ризику їх компрометації;
- Уникнути необхідності повторної автентифікації користувача;
- Відкликати доступ шляхом анулювання токена оновлення.

OpenID Connect (OIDC) є простим рівнем ідентичності, що розміщується поверх OAuth 2.0. OIDC спрощує верифікацію ідентичності користувача та отримання базової інформації про профіль від постачальника ідентичності.

OIDC розширює OAuth 2.0 шляхом додавання токена ідентичності (ID Token), що містить інформацію про автентифікованого користувача. На відміну від OAuth 2.0, що є протоколом авторизації, OIDC є протоколом автентифікації.

Типовий потік OIDC:

1. Користувач переходить до веб-застосунку;
2. Веб-застосунок перенаправляє користувача до постачальника ідентичності;
3. Користувач вводить облікові дані та надає згоду;
4. Постачальник ідентичності повертає токен ідентичності браузеру;
5. Браузер перенаправляє токен ідентичності до веб-застосунку;
6. Веб-застосунок валідує токен ідентичності та встановлює сесію.

OIDC є придатним для застосунків, орієнтованих на споживачів (consumer-facing applications), де необхідна автентифікація користувачів через зовнішніх постачальників ідентичності (Google, Facebook тощо).

Security Assertion Markup Language (SAML) є відкритим стандартом на основі XML, що дозволяє підприємствам передавати інформацію про автентифікацію та авторизацію користувачів партнерським компаніям та корпоративним застосункам.

SAML є старішим стандартом порівняно з OAuth 2.0 та OIDC, але продовжує широко використовуватися для єдиного входу у корпоративних застосунках. SAML підтримує як автентифікацію, так і авторизацію, та використовує XML для tokenів замість JSON.

Вибір стандарту залежить від контексту використання:

Таблиця 4.1. Порівняння стандартів автентифікації та авторизації

	SAML	OAuth 2.0	OIDC
Фокус	Автентифікація + авторизація	Авторизація	Автентифікація
Основне застосування	SSO для корпоративних застосунків	Авторизація API	SSO для споживацьких застосунків
Формат токена	XML	JSON (JWT)	JSON (JWT)
Походження	Oasis (2001)	Twitter, Google (2006)	OpenID Foundation (2014)

- SAML — для інтеграції з корпоративними системами ідентифікації;
- OAuth 2.0 — для авторизації доступу до API;
- OIDC — для автентифікації користувачів через зовнішніх постачальників ідентичності.

4.3.4 Безпека у багатоорганізаційній архітектурі

Програмне забезпечення типу Software-as-a-Service (SaaS) часто обслуговує множини організацій у єдиній розгорнутій інстанції. Це створює специфічні виклики для управління ідентифікацією та контролю доступу.

Існує два основних підходи до організації користувачів у багатоорганізаційних системах:

Ізольовані користувачі — кожна організація має власний набір користувачів, що не перетинається з користувачами інших організацій. Користувач може бути членом лише однієї організації. Цей підхід є простішим у реалізації, але менш гнучким.

Спільні користувачі — користувачі можуть бути членами множини організацій. Один користувач може мати різні ролі та дозволи у різних організаціях. Цей підхід є складнішим у реалізації, але надає більшу гнучкість для сценаріїв, коли користувачі працюють з множиною організацій.

Вибір підходу залежить від бізнес-вимог та сценаріїв використання системи.

Критичним аспектом багатоорганізаційної архітектури є забезпечення ізоляції даних між організаціями. Існує два основних підходи:

- **Фізична ізоляція** — кожна організація має власну репліку програмного забезпечення, базу даних або окремий простір у базі даних. Цей підхід надає найвищий рівень ізоляції, але ускладнює масштабування та підтримку.
- **Логічна ізоляція** — всі організації використовують спільну базу даних, а ізоляція забезпечується на рівні застосунку через фільтрацію даних за ідентифікатором організації. Цей підхід є простішим у масштабуванні, але вимагає ретельної реалізації для запобігання витоку даних між організаціями.

Логічна ізоляція вимагає:

- Додавання ідентифікатора організації до всіх запитів до бази даних;
- Перевірки належності даних до організації користувача на кожному рівні застосунку;
- Автоматизованого тестування для виявлення потенційних витоків даних між організаціями.

4.3.5 Автентифікація сервісних облікових записів

Окрім користувачів, системи часто містять *службові облікові записи* — автоматизовані процеси, що потребують доступу до ресурсів без участі користувача.

Службовий обліковий запис (англ. Service Account) є обліковим записом, що використовується застосунками або службами для автентифікації та виконання автоматизованих задач. Сервісні облікові записи не можуть інтерактивно вводити облікові дані.

Підходи до автентифікації службових облікових записів:

API ключі є простим підходом, де кожен сервіс отримує унікальний ключ для автентифікації. API ключі зазвичай є довгостроковими та не мають механізму автоматичного оновлення.

Переваги: простота реалізації та використання.

Недоліки: складність ротації ключів, відсутність детальної інформації про дозволи у самому ключі.

OAuth 2.0 Client Credentials Grant дозволяє сервісам отримувати токени доступу шляхом автентифікації з використанням `client_id` та `client_secret`. Отримані токени є короткостроковими та автоматично оновлюються.

Переваги: стандартизований підхід, підтримка короткострокових токенів, можливість централізованого управління дозволами.

Недоліки: вища складність порівняно з API ключами.

Для службових облікових записів рекомендується:

- Використання окремих облікових записів для кожного сервісу;
- Надання мінімально необхідних дозволів (принцип найменших привілеїв);
- Регулярна ротація облікових даних;
- Аудит використання сервісних облікових записів;
- Використання механізмів управління секретами (наприклад, HashiCorp Vault, AWS Secrets Manager).

4.3.6 Децентралізоване управління ідентифікацією

Традиційні системи управління ідентифікацією є централізованими — постачальник ідентичності зберігає та контролює дані про користувачів. Децентралізоване управління ідентифікацією пропонує альтернативний підхід.

Децентралізоване управління ідентифікацією (англ. Decentralized Identity Management) є підходом, у якому користувачі контролюють власні дані ідентичності без залежності від централізованого постачальника. Децентралізована ідентичність базується на трьох учасниках: емітентах (issuers), користувачах та верифікаторах (verifiers).

Архітектура децентралізованої ідентичності відрізняється від централізованих систем розподілом відповідальності між учасниками та використанням криптографічних методів для верифікації без необхідності звертання до центрального реєстру. Процес роботи децентралізованої ідентичності складається з наступних етапів:

1. **Емітент** (організація, установа) видає цифрові посвідчення (verifiable credentials) користувачу. Приклади: диплом про освіту, водійські права, підтвердження працевлаштування.
2. **Користувач** зберігає цифрові посвідчення у власному цифровому гаманці (digital wallet) на своєму пристрої.
3. **Верифікатор** (застосунок, сервіс) запитує підтвердження певних атрибутів у користувача.
4. Користувач надає верифікатору необхідні посвідчення зі свого гаманця.
5. Верифікатор перевіряє автентичність посвідчень, звертаючись до емітента або використовуючи криптографічні методи.

Переваги децентралізованої ідентичності:

- Користувачі контролюють власні дані ідентичності;

- Зменшення ризиків витоку даних — відсутня централізована мішень;
- Можливість вибіркового розкриття атрибутів (користувач може підтвердити лише необхідні атрибути, не розкриваючи всі дані);
- Незалежність від окремих постачальників ідентичності.

Недоліки та виклики:

- Складність реалізації та підтримки;
- Необхідність управління ключами на пристрої користувача;
- Ризик втрати доступу при втраті пристрою;
- Відсутність широкої підтримки у існуючих системах;
- Питання відкликання посвідчень.

Децентралізоване управління ідентифікацією є перспективним напрямком, але на поточний момент має обмежене практичне застосування. Для більшості корпоративних та комерційних систем централізоване або федеративне управління ідентифікацією залишається більш придатним підходом.

4.4 Випуск програмного забезпечення

Випуск програмного забезпечення є процесом доставки функціональності кінцевим користувачам через перенесення коду з середовища розробки до середовища кінцевого користувача (production) (далі — СКК). Організація даного процесу впливає на швидкість доставки змін, якість програмного забезпечення та складність управління кодовою базою.

Частота випусків програмного забезпечення варіюється залежно від типу продукту, організаційних можливостей та бізнес-вимог:

Звичайне програмне забезпечення Випуск раз на шість місяців. Характерний для традиційного програмного забезпечення з складним процесом встановлення та оновлення.

Хмарний продукт (квартальні випуски) Випуск раз на три-шість місяців. Проміжний підхід між традиційною моделлю та частими оновленнями.

Хмарний продукт (регулярні випуски) Випуск раз на один-два тижні. Забезпечує швидшу доставку функціональності при збереженні контрольованого процесу випуску.

Хмарний продукт (безперервне розгортання) Кілька випусків на день. Кожна зміна автоматично доставляється до СКК після проходження автоматизованої перевірки.

Вибір частоти випусків визначає організацію системи контролю версій, процесу тестування та архітектурні рішення щодо зворотної сумісності.

4.4.1 Патерни розгалуження

Патерни розгалуження (англ. branching patterns) визначають організацію гілок у системі контролю версій для управління паралельною розробкою, інтеграцією змін та випуском версій [11].

Основна гілка (англ. mainline, master, main) є центральною гілкою розробки, до якої інтегруються зміни з інших гілок. Стан основної гілки визначає готовність кодової бази до випуску.

Функціональна гілка (англ. feature branch) ізолює розробку окремої функціональності від основної гілки. Після завершення розробки зміни інтегруються до основної гілки через злиття (merge) або перебазування (rebase).

Організація процесу випуску програмного забезпечення може відбуватися двома основними способами:

Випуск з основної гілки (англ. release from mainline) Основна гілка завжди перебуває у стані, готовому до випуску. Кожен коміт до основної гілки потенційно може бути випущений до СКК. Даний підхід є передумовою для безперервної доставки (Continuous Delivery). Випуск здійснюється через створення тегу на відповідному коміті основної гілки.

Гілка випуску (англ. release branch) Створюється окрема гілка від основної гілки у момент, коли набір функціональності готовий до випуску. У гілці випуску виконуються лише коміти для стабілізації версії: виправлення дефектів, оновлення документації, налаштування конфігурації. Основна гілка продовжує розвиватися незалежно. Після стабілізації з гілки випуску створюється реліз до виробництва.

Вибір між випуском з основної гілки та гілкою випуску визначається частотою випусків та можливостями організації підтримувати основну гілку у постійно готовому до СКК стані.

Після випуску в СКК можуть бути знайдені дефекти, які не було помічено під час підготовки випуску. Виправлення таких дефектів часто відбувається терміново, поза звичним циклом розробки. Для підтримки цього використовуються *гілки термінового виправлення*.

Гілка термінового виправлення (англ. hotfix branch) призначена для виправлення критичних дефектів у робочому середовищі, які потребують негайного усунення.

Організація гілок термінового виправлення залежить від обраного шляху до виробництва:

1. **При випуску з основної гілки:** Гілка термінового виправлення створюється від тегу відповідної версії у СКК. Після виправлення дефекту гілка зливається назад до основної гілки та створюється новий випуск.
2. **При випуску з гілки випуску:** Гілка термінового виправлення створюється від гілки випуску відповідної версії. Виправлення додається до гілки випуску, після чого зміни зливаються як до гілки випуску, так і до основної гілки для забезпечення, що виправлення не втрачається у майбутніх версіях.
3. **При безперервній доставці:** У разі постійно готової до випуску основної гілки, термінове виправлення може бути виконане безпосередньо у основній гілці та випущене як звичайний реліз.

Різні підходи до організації випуску програмного забезпечення накладають різні вимоги до стану основної гілки. У патернах з гілками випуску основна гілка може містити незавершені функціональності або зміни, що потребують додаткової стабілізації перед випуском. Альтернативним підходом є забезпечення постійної готовності основної гілки до безпосереднього випуску у будь-який момент часу.

Готова до випуску основна гілка (англ. release-ready mainline) є підходом до управління кодовою базою, за якого основна гілка постійно перебуває у стані, достатньому для безпосереднього випуску до СКК.

Підтримка готової до випуску основної гілки вимагає:

- Високого покриття автоматизованими тестами для виявлення регресій;
- Автоматизованої перевірки якості коду;
- Дисципліни розробників щодо інтеграції лише завершених змін;
- Використання прапорців функціональності (feature flags) для приховування незавершених функцій;
- Можливості швидкого відкату змін у разі виявлення проблем.

Готова до випуску основна гілка є передумовою для:

Безперервної доставки (англ. Continuous Delivery) Здатність виконувати часті випуски програмного забезпечення, але зберігаючи ручне прийняття рішення про момент випуску. Кожна зміна проходить через автоматизований конвеєр збірки та тестування, після чого стає доступною для випуску до СКК за рішенням відповідальної особи.

Безперервного розгортання (англ. Continuous Deployment) Повна автоматизація процесу випуску, за якої кожна зміна, що пройшла автоматизовану перевірку, автоматично розгортається до виробничого середовища без ручного втручання. Результатом є велика кількість випусків до СКК щодня.

Альтернативним підходом до підтримки постійно готової до випуску основної гілки є явне відділення виробничого коду від коду, що активно розробляється. У ситуаціях, коли основна гілка використовується для інтенсивної розробки та інтеграції експериментальних функціональностей, доцільним може бути збереження стабільних випусків в окремій гілці.

Гілка зрілості (англ. maturity branch, production branch) є окремою гілкою, що містить лише той код, який пройшов усі етапи перевірки та знаходиться у СКК або готовий до випуску.

У патерні гілки зрілості, після досягнення кодовою базою певного рівня готовності, відповідний стан копіюється до гілки зрілості. Основна гілка продовжує розвиватися, в той час як гілка зрілості містить лише стабільні випуски. Коміти до гілки зрілості відбуваються через злиття зі стабілізованих гілок випуску або безпосередньо з основної гілки після завершення усіх перевірок.

Гілка середовища (англ. environment branch) створюється для зберігання конфігураційних змін, специфічних для певного середовища виконання (тестове, проміжне, виробниче).

Гілки середовища є антипатерном у сучасній розробці програмного забезпечення з наступних причин:

- Порушують принцип єдиного джерела істини;
- Ускладнюють відстеження відмінностей між середовищами;
- Створюють ризик розбіжності поведінки між середовищами;
- Утруднюють автоматизацію розгортання.

Рекомендованою альтернативою є зовнішня конфігурація (англ. externalized configuration), за якої конфігураційні параметри зберігаються поза кодовою базою (у змінних середовища, конфігураційних сервісах, файлах параметрів) та застосовуються під час розгортання до відповідного середовища.

4.4.2 Стратегії розгалуження

Стратегія розгалуження визначає конкретний набір правил щодо створення, іменування, злиття та видалення гілок, а також розподіл відповідальності між різними типами гілок. Розглянемо найпоширеніші стратегії розгалуження.

4.4.2.1 Git Flow

Git Flow є стратегією розгалуження, орієнтованою на організацію розробки з плановими випусками та підтримкою кількох версій у СКК.

Git Flow складається з наступних видів гілок:

master Містить лише код у виробництві. Кожен коміт до master є випуском.

develop Основна гілка розробки, містить найновіший код для наступного випуску.

feature/* Функціональні гілки, створюються від develop, зливаються назад до develop.

release/* Гілки випуску, створюються від develop для підготовки нової версії. Зливаються як до master (після готовності), так і до develop (для збереження виправлень).

hotfix/* Гілки термінових виправлень, створюються від master для виправлення критичних дефектів у виробництві. Зливаються як до master, так і до develop.

Розробка відбувається у декілька етапів, в яких зміна “просочується” через декілька гілок:

1. Нова функціональність розробляється у гілці feature/*, створеній від develop;
2. Після завершення feature/* зливається до develop;
3. Коли develop готовий до випуску, створюється гілка release/*;
4. У гілці release/* виконуються фінальні виправлення та підготовка до випуску;
5. Гілка release/* зливається до master та позначається тегом версії;
6. Зміни з release/* зливаються назад до develop;
7. У разі критичного дефекту створюється hotfix/* від master, виправляється, та зливається до обох master та develop.

Git Flow підходить для:

- Програмного забезпечення з плановими випусками (щоквартально, щомісячно);
- Проектів, що потребують підтримки кількох версій у виробництві;
- Команд, що працюють над кількома функціональностями паралельно.

Git Flow характеризується рядом практичних недоліків, що обмежують його застосовність у сучасних умовах розробки програмного забезпечення. Модель передбачає підтримку множини довгоживучих гілок (master, develop, feature, release,

hotfix), що призводить до значної складності управління та високих витрат на злиття коду між гілками. Частота конфліктів злиття зростає пропорційно до кількості паралельно активних гілок та тривалості їхнього існування. Git Flow проектувався для циклів випуску програмного забезпечення з тривалими інтервалами між релізами, що суперечить практикам безперервної інтеграції та безперервного розгортання (CI/CD), де код інтегрується та розгортається до виробничого середовища декілька разів на день. Процес створення hotfix-гілок від master та їхнього подальшого злиття як у master, так і у develop створює додаткові точки можливих помилок та вимагає дисципліни від команди. Для невеликих команд чи проєктів з короткими циклами випуску накладні витрати на підтримку цієї моделі перевищують її переваги щодо організації релізів. Додатково, release-гілки, які створюються для підготовки випуску, можуть існувати паралельно з активною розробкою в develop, що ускладнює інтеграцію критичних виправлень та призводить до дублювання зусиль команди.

4.4.2.2 GitHub Flow

GitHub Flow є спрощеною стратегією розгалуження, орієнтованою на безперервну доставку та швидкі ітерації.

GitHub Flow має всього два види гілок:

main (master) Основна гілка, завжди готова до випуску та відображає стан СКК.

feature/* Функціональні гілки для розробки будь-яких змін (функціональності, виправлення дефектів тощо).

Процес роботи з GitHub Flow:

1. Створюється гілка від main для будь-якої зміни;
2. Виконуються коміти до гілки функціональності;
3. Відкривається запит на злиття (Pull Request);
4. Код переглядається іншими розробниками;
5. Після схвалення код може бути розгорнуто до тестового середовища;
6. Після перевірки запит на злиття зливається до main;
7. Зміни у main автоматично створюють артефакти, формально готові до розгортання на СКК.

Рішення про розгортання артефакту конкретної версії основної гілки може бути прийняте за консенсусом готовності стану гілки на момент цієї версії до розгортання — наприклад, проходженням артефакту через оточення, на яких виконується повторне та регресійне тестування перед випуском.

GitHub Flow підходить для:

- Хмарних продуктів з безперервною доставкою;
- Команд, що практикують швидкі ітерації;
- Проектів, що не потребують підтримки кількох версій у СКК.

GitHub Flow характеризується надмірною спрощеністю, що обмежує його застосовність у контекстах, які вимагають більш структурованого управління релізами. Модель передбачає існування лише однієї довгоживучої гілки (main/master) та короткоживучих feature-гілок, що унеможлиблює підтримку декількох виробничих версій програмного забезпечення одночасно — критична вимога для корпоративного програмного забезпечення, мобільних застосунків та інших систем, де користувачі можуть використовувати різні версії продукту. Відсутність формалізованого процесу підготовки релізу (release-гілок) призводить до того, що тестування та стабілізація відбуваються безпосередньо в main-гілці, що підвищує ризик випуску дефектів у виробниче середовище. GitHub Flow припускає можливість розгортання будь-якої зміни безпосередньо у виробництво після злиття, що вимагає надзвичайно високої зрілості процесів автоматизованого тестування, моніторингу та відновлення — вимоги, яким не завжди відповідають організації. Модель не розрізняє між звичайними функціональними змінами та критичними виправленнями (hotfix), що ускладнює пріоритизацію та відстеження термінових змін. Для програмного забезпечення з регламентованими циклами випуску (наприклад, квартальні релізи, узгоджені з маркетинговими кампаніями чи зовнішніми залежностями) відсутність механізму координації множини змін у рамках одного релізу створює організаційні складнощі. GitHub Flow не надає підтримки для release candidate-версій чи beta-тестування, що є стандартною практикою для багатьох типів програмного забезпечення.

4.4.2.3 One Flow

OneFlow є стратегією розгалуження, що поєднує елементи Git Flow та GitHub Flow, забезпечуючи гнучкість для різних моделей випуску.

Основні принципи OneFlow:

- Основна гілка (master або main) є єдиною постійною гілкою;
- Функціональні гілки створюються від основної гілки та зливаються назад;
- Гілки випуску створюються від основної гілки лише тимчасово та після готовності випуску — гілка зливається назад до основної;

Варіації OneFlow:

OneFlow з основною гілкою як основною гілкою розробки Аналогічно GitHub Flow, але з можливістю створення гілок випуску для стабілізації версії.

OneFlow з окремою гілкою розробки `develop` слугує основною гілкою розробки, `master` містить стабільні випуски (гілка зрілості). Даний варіант наближається до Git Flow, але без обов'язкових гілок `release/*`.

OneFlow представляє компромісне рішення між складністю Git Flow та спрощеністю GitHub Flow, зберігаючи структурованість при зменшенні накладних витрат на управління гілками. Модель природно інтегрується з практиками безперервної інтеграції, оскільки `main` завжди містить останній стабільний стан коду, придатний для автоматизованого тестування та розгортання. OneFlow зберігає можливість паралельної підтримки декількох версій продукту через `release`-гілки, що критично для корпоративного програмного забезпечення, одночасно уникаючи постійної синхронізації між довгоживучими гілками. Проте, модель все ще вимагає дисципліни команди щодо управління `release`-гілками та може бути надмірно складною для проектів з дуже частими випусками (декілька разів на день), де GitHub Flow залишається оптимальним вибором. OneFlow найкраще підходить для проектів з регулярними, але не надто частими релізами (тижневі або місячні цикли), де необхідна підтримка декількох версій та структурований процес підготовки випуску.

4.4.3 Безперервна інтеграція

Традиційний підхід до інтеграції змін від кількох розробників характеризується періодичним злиттям довготривалих гілок розробки, що призводить до:

- Складних конфліктів злиття через накопичення великої кількості розбіжностей;
- Виявлення проблем сумісності на пізніх етапах розробки;
- Непередбачуваності часу, необхідного для інтеграції;
- Зниження якості через складність перевірки інтегрованого коду.

Ці проблеми мають спільну причину: затримка інтеграції збільшує дивергенцію кодових баз, що експоненційно ускладнює злиття. Чим довше зміни залишаються ізольованими в окремих гілках, тим більша ймовірність конфліктів та тим складнішими стають ці конфлікти. Логічним рішенням є мінімізація часу існування паралельних гілок через максимально часту інтеграцію змін до спільної кодової бази, що формує основу практики *безперервної інтеграції*.

Безперервна інтеграція (англ. Continuous Integration, CI) є практикою розробки програмного забезпечення, за якої розробники регулярно (до кількох

разів на день) інтегрують свої зміни до спільного репозиторію, після чого автоматично виконується збірка та тестування для швидкого виявлення проблем інтеграції.

Основні практики безперервної інтеграції формують набір технічних та організаційних заходів, спрямованих на мінімізацію часу між написанням коду та виявленням проблем інтеграції:

1. Підтримка єдиного репозиторію вихідного коду;
2. Автоматизація збірки;
3. Забезпечення самоперевірки збірки через автоматизовані тести;
4. Щоденне інтегрування змін до основної гілки;
5. Збірка кожного коміту до основної гілки;
6. Швидкість виконання збірки (до 10 хвилин);
7. Тестування у середовищі, ідентичному виробничому;
8. Легкий доступ до останньої збірки;
9. Прозорість процесу збірки та тестування;
10. Автоматизація розгортання.

Ці практики взаємопов'язані та доповнюють одна одну: автоматизація збірки та тестування забезпечує швидкий зворотний зв'язок, частота інтеграції зменшує обсяг змін у кожному коміті, а прозорість процесу дозволяє команді оперативно реагувати на проблеми.

4.4.3.1 Сервери безперервної інтеграції

Сервер безперервної інтеграції є інструментом автоматизації процесу збірки, тестування та розгортання програмного забезпечення. Сервер CI відстежує зміни у репозиторії, ініціює збірку при виявленні нових комітів, виконує визначені кроки перевірки та повідомляє про результати.

Конвеєр безперервної інтеграції, що виконується сервером CI включає різні типи автоматизованих перевірок:

Модульні тести (англ. unit tests) Перевіряють коректність окремих компонентів у ізоляції. Виконуються найшвидше, мають найвище покриття коду.

Збірка Перевіряє компіляцію коду та відсутність синтаксичних помилок, а також успішне створення артефакту.

Статичний аналіз коду Автоматизована перевірка коду на відповідність стандартам, потенційні дефекти та вразливості безпеки.

Інтеграційні тести Перевіряють взаємодію між компонентами системи, включаючи бази даних, зовнішні сервіси, черги повідомлень.

API-тести Перевіряють коректність програмних інтерфейсів, включаючи валідацію запитів, відповідей, обробку помилок.

Контрактне тестування (англ. contract testing) Перевіряє відповідність між очікуваннями споживача API та реальною реалізацією постачальника.

Можливість розгортання Перевірка успішного розгортання артефакту до тестового середовища.

Наскрізнi тести (англ. end-to-end tests, E2E) Перевіряють повні сценарії роботи системи з точки зору користувача. Виконуються найповільніше, часто є найменш стабільними.

Послідовність перевірок організовується за принципом піраміди тестування: велика кількість швидких модульних тестів, менша кількість інтеграційних тестів, мінімальна кількість повільних наскрізних тестів.

Автоматизація збірки та тестування у рамках безперервної інтеграції створює передумови для наступного етапу автоматизації процесу розробки — доставки програмного забезпечення до СКК. Наявність автоматизованих перевірок дозволяє з високою впевненістю визначити готовність зміни до випуску, однак безперервна інтеграція сама по собі не вирішує питання організації процесу випуску.

Безперервна доставка (англ. Continuous Delivery, CD) є практикою розробки програмного забезпечення, за якої зміни можуть бути випущені до виробництва у будь-який момент часу через повну автоматизацію процесу збірки, тестування та підготовки до розгортання. Рішення про випуск приймається вручну.

Безперервне розгортання (англ. Continuous Deployment) є розширенням безперервної доставки, за якого кожна зміна, що пройшла автоматизовану перевірку, автоматично розгортається до виробничого середовища без ручного втручання.

Відмінність між безперервною доставкою та безперервним розгортанням полягає у автоматизації останнього кроку: при безперервній доставці рішення про розгортання приймає людина, при безперервному розгортанні — автоматична система після успішного проходження усіх перевірок. В обох варіантах ПЗ проходить через набір певних автоматизованих етапів:

Конвеєр випуску (англ. release pipeline, deployment pipeline) є автоматизованою послідовністю етапів, через які проходить зміна коду від коміту розробника до розгортання у СКК.

Для забезпечення якості програмного забезпечення на кожному етапі розробки, конвеєр випуску організовується як послідовність середовищ з наростаючим

рівнем подібності до виробничого середовища. Типова структура конвеєру випуску включає наступні етапи та середовища:

1. **Локальна машина розробника** Початковим етапом конвеєру є локальне середовище розробника, де виконуються базові перевірки перед інтеграцією коду до спільної кодової бази:
 - Виконання модульних тестів перед комітом;
 - Перевірка стилю коду;
 - Коміт до системи контролю версій.
2. **Сервер безперервної інтеграції** Після коміту змін до системи контролю версій, код автоматично обробляється сервером безперервної інтеграції, який виконує складніші перевірки та створює артефакт для подальшого розгортання:
 - Автоматичне виявлення нових комітів;
 - Збірка артефакту;
 - Виконання модульних тестів;
 - Виконання інтеграційних тестів;
 - Статичний аналіз коду;
 - Перевірка стійкості до відмов (fault-tolerance tests);
 - Завантаження артефакту до репозиторію артефактів (Nexus, Artifactory).
3. **Середовище контролю якості (QA)** Артефакт, що успішно пройшов етап збірки та автоматизованих перевірок, одночасно з розгортанням у середовище для розробників, розгортається у спеціалізованому середовищі для функціонального та навантажувального тестування:
 - Автоматичне розгортання артефакту;
 - Тести прийняття (acceptance tests);
 - Навантажувальне тестування (performance testing);
 - Дослідницьке тестування (exploratory tests);
 - Тестування API та користувацького інтерфейсу.
4. **Проміжне середовище (Staging)** Перед розгортанням у СКК артефакт проходить валідацію у проміжному середовищі, яке максимально наближене до виробничої конфігурації:
 - Середовище, ідентичне СКК;
 - Димові тести (smoke tests) з синтетичними транзакціями;
 - Навантажувальне тестування у реалістичному середовищі;
 - Моніторинг метрик;

- Можливість експозиції середовища третім сторонам.
5. **Середовище кінцевого користувача** Фінальним етапом конвеєру є розгортання у СКК, де програмне забезпечення стає доступним кінцевим користувачам:
- Розгортання затвердженого артефакту;
 - Димові тести з синтетичними транзакціями;
 - Активний моніторинг;
 - Централізоване логування;
 - Збір зворотного зв'язку для оцінки бізнес-гіпотез.

Артефакт просувається до наступного середовища лише після задовільного виконання усіх перевірок поточного етапу. Критерії просування визначають умови, за яких артефакт вважається готовим.

4.5 Сховища даних

Вибір типу сховища даних є одним з фундаментальних архітектурних рішень, що визначає можливості програмного забезпечення щодо збереження, доступу та обробки інформації. Різні типи сховищ даних оптимізовані для різних патернів доступу, обсягів даних та вимог до консистентності, що робить розуміння їхніх характеристик критично важливим для архітекторів програмного забезпечення.

Еволюція сховищ даних відображає зміни у вимогах до програмного забезпечення протягом десятиліть — від ієрархічних баз даних 1960-х років через реляційні системи 1970-х до різноманітних спеціалізованих рішень сучасності. Кожен тип сховища виник як відповідь на специфічні обмеження попередніх підходів та нові вимоги предметних областей.

4.5.1 Ієрархічні бази даних

Ієрархічні бази даних, що з'явилися у 1960-х роках, організують дані у деревоподібні структури, де кожен запис має єдиного батька та може мати декілька нащадків. Дана модель відображає природні ієрархічні відношення між сутностями.

Ієрархічна база даних є системою управління базами даних, що організує дані у деревоподібну структуру, де записи пов'язані відношеннями батько-нащадок.

Основні характеристики ієрархічних баз даних включають:

- Швидкий доступ до даних за наявності відомого шляху в ієрархії;

- Ефективне використання пам'яті завдяки відсутності дублювання посилань;
- Обмеженість у представленні відношень багато-до-багатьох;
- Складність реструктуризації при зміні вимог.

Типовими випадками використання ієрархічних баз даних є управління запасами, де продукти природно групуються за категоріями та підкатегоріями; управління мережами, де обладнання організоване в ієрархічні топології; організаційні структури підприємств. Прикладами реалізацій є IBM Information Management System (IMS), Windows Registry, LDAP (Lightweight Directory Access Protocol).

Незважаючи на обмеженість у порівнянні з сучаснішими підходами, ієрархічні бази даних досі використовуються у системах, що мають чітко визначену ієрархічну структуру та потребують високої продуктивності при навігації деревом даних.

4.5.2 Реляційні бази даних

Обмеження ієрархічних баз даних стали особливо критичними в корпоративних системах 1960-1970-х років, де складність даних та вимоги до гнучкості зростали. Необхідність у більш гнучкій та математично обґрунтованій моделі даних стимулювала дослідження альтернативних підходів.

Реляційні бази даних, засновані на реляційній алгебрі та формалізовані Едгаром Коддом у 1970 році, революціонізували управління даними через введення табличної організації з формальними математичними основами.

Реляційна база даних є типом бази даних, що зберігає та надає доступ до пов'язаних між собою точок даних. Реляційні бази даних засновані на реляційній моделі — інтуїтивному та формалізованому способі представлення даних у таблицях. У реляційній базі даних кожен рядок таблиці є записом з унікальним ідентифікатором, званим ключем. Стовпці таблиці містять атрибути даних, і кожен запис зазвичай має значення для кожного атрибута, що спрощує встановлення зв'язків між точками даних. Таблиці можуть бути пов'язані між собою через включення ідентифікатору одної таблиці як стовпця іншої, що дозволяє встановити зв'язок між рядками таблиці.

Ключові властивості реляційних баз даних:

- Структурована організація даних у таблиці з визначеною схемою;
- ACID-гарантії (Atomicity, Consistency, Isolation, Durability) для транзакцій;
- Декларативна мова запитів SQL (Structured Query Language);
- Нормалізація даних для зменшення надмірності;
- Підтримка складних зв'язків через зовнішні ключі.

Реляційні бази даних залишаються домінуючим вибором для програмного забезпечення більшості сфер та доменів завдяки зрілості технології, стандартизації SQL та надійним гарантіям консистентності.

4.5.3 NoSQL бази даних

Термін NoSQL (спочатку “non-SQL”, пізніше інтерпретований як “not only SQL”) охоплює різноманітні нереляційні системи управління базами даних, що з’явилися як відповідь на обмеження реляційних баз даних у контексті розподілених систем та великих обсягів даних.

NoSQL база даних є нереляційною системою управління базами даних, оптимізованою для специфічних моделей даних та патернів доступу, що часто жертвує деякими ACID-гарантіями заради горизонтальної масштабованості та продуктивності.

Основними категоріями NoSQL баз даних є ключ-значення, документо-орієнтовані, колонкові та графові сховища. Кожна категорія оптимізована для різних випадків використання та має унікальні компроміси між консистентністю, доступністю та стійкістю до поділу мережі (CAP-теорема).

Сховище ключ-значення (англ. Key-value store) є типом нереляційної бази даних, що зберігає дані як колекцію пар ключ-значення, де ключ слугує унікальним ідентифікатором. Як ключі, так і значення можуть бути будь-чим — від простих об’єктів до складних складених об’єктів. Ключ-значення бази даних гарно розділяються та дозволяють горизонтальне масштабування на рівні, недосяжному для інших типів баз даних.

Характеристики сховищ ключ-значення:

- Надзвичайно висока продуктивність операцій читання та запису;
- Проста модель даних без підтримки складних запитів;
- Горизонтальна масштабованість через шардування (розділення між різними екземплярами бази даних) за ключем;
- Відсутність схеми для значень;
- Оптимізація для точкових запитів за ключем.

Типовими випадками використання є кешування результатів обчислень, управління сесіями користувачів, зберігання конфігурацій, реалізація обмежувачів швидкості (rate limiting).

Документо-орієнтована база даних є базою даних, що зберігає інформацію у документах. Замість зберігання даних у фіксованих рядках та стовпцях, документо-орієнтовані бази даних використовують гнучкі документи — такі, поля яких не є чітко зафіксованими та можуть змінюватись від документу до документа. Документо-орієнтовані бази даних є найпопулярнішою альтернативою табличним реляційним базам даних.

Ключові властивості:

- Гнучка схема — документи в одній колекції можуть мати різну структуру;
- Вкладені структури даних без необхідності з'єднань (joins);
- Індекссування за довільними полями документів;
- Підтримка складних запитів з агрегацією;
- Природне відображення об'єктів предметної області.

Документо-орієнтовані бази даних оптимальні для систем, що не мають необхідності у складних реляційних структурах та навігації ними, а також для систем, що вимагають роботи з об'єктами змінної чи нечіткої структури.

Колонкова база даних є системою управління базами даних, оптимізованою для швидкого отримання стовпців даних, типово в аналітичних застосунках. Колонкова організація зберігання табличних даних є важливим чинником продуктивності аналітичних запитів, оскільки драматично зменшує загальні вимоги до дискового вводу-виводу та зменшує обсяг даних, що має бути завантажено з диска. На відміну від реляційних, що зберігають дані рядками, колонкові БД організовують дані за стовпцями, що покращує продуктивність для аналітичних навантажень.

Колонкові бази даних використовуються для аналітики великих даних, сховищ даних (англ. data warehousing), обробки подій та метрик.

4.5.3.1 Графові бази даних

Графова база даних є систематичною колекцією даних, що акцентує увагу на відношеннях між різними екземплярами даних. Такі бази даних використовують математичну теорію графів для відображення зв'язків між даними — вершини є сутностями, а ребра — відношеннями. На відміну від реляційних баз даних, що зберігають дані у жорстких табличних структурах, графові бази даних зберігають дані як мережу сутностей та відношень. У результаті, ці бази даних часто забезпечують кращу продуктивність та гнучкість, оскільки більше підходять для моделювання деяких реальних сценаріїв.

Основні компоненти графових баз даних:

- Вершини (вузли) — представляють сутності предметної області;
- Ребра — представляють відношення між сутностями;
- Властивості — атрибути вершин та ребер;
- Мітки — категоризація вершин та ребер.

Графові бази даних виявляють найбільшу ефективність при операціях, що потребують переходу за декількома зв'язками між сутностями. Типовими випадками використання є соціальні мережі (друзі друзів, спільноти), рекомендаційні системи (схожі товари, користувачі), виявлення шахрайства (аналіз складних патернів транзакцій), управління знаннями.

Об'єктно-орієнтована база даних є системою управління базами даних, що зберігає дані як об'єкти — екземпляри класів з атрибутами та методами. Об'єктно-орієнтовані бази даних усувають невідповідність між об'єктною моделлю програми та реляційною моделлю бази даних.

Характеристики:

- Безпосереднє зберігання складних об'єктних структур;
- Підтримка успадкування та поліморфізму;
- Можливість визначення методів об'єктів у базі даних;
- Відсутність необхідності об'єктно-реляційного відображення (ORM).

Історично об'єктно-орієнтовані бази даних не отримали широкого поширення, однак їхні концепції знайшли застосування у сучасних мобільних застосунках для локального зберігання даних з можливістю швидкого доступу до складних об'єктних структур.

4.5.4 Спеціалізовані сховища даних

Окрім загальнопризначених типів баз даних, існує множина спеціалізованих рішень, оптимізованих для конкретних патернів даних, вимог тощо.

База даних часових послідовностей (англ. Time series database) є базою, оптимізованою для зберігання, пошуку та обробки даних з часовими мітками — часових рядів. Дані часових рядів є вимірюваннями або подіями, що відстежуються, беруться зразки та виконується агрегація. Цими даними можуть бути метрики серверів, моніторинг продуктивності застосунків, мережні дані, дані датчиків, події, кліки, транзакції на ринку та багато інших типів аналітичних даних.

Ключові характеристики часових рядів:

- Упорядкованість за часовою міткою;
- Монотонне додавання нових записів (append-only);
- Рідко виконувані операції оновлення чи видалення;
- Типові запити охоплюють часові діапазони;
- Потреба в агрегації за часовими вікнами.

Бази даних часових послідовностей застосовують спеціалізовані структури даних та алгоритми стиснення для ефективної роботи з послідовностями метрик.

Бази даних в пам'яті є спеціалізованими базами даних, що покладаються переважно на оперативну пам'ять для зберігання даних. Ці бази забезпечують мінімальні часи відгуку, усуваючи потребу доступу до стандартних дискових накопичувачів.

Властивості баз даних в пам'яті:

- Затримка доступу, що вимірюється у мікросекундах;
- Висока пропускна здатність операцій;
- Необхідність механізмів персистентності для збереження даних;
- Обмеженість обсягу оперативної пам'яті.

Бази даних в пам'яті використовуються як кеш перед традиційними базами даних, для зберігання сесій користувачів, лічильників та метрик реального часу, обмежувачів швидкості API, черг повідомлень.

Сховище конфігурації є спеціалізованими розподіленими сховищами ключ-значення для управління конфігурацією, що забезпечують узгоджений доступ до конфігураційних даних для множини екземплярів сервісів, часто з підтримкою динамічного оновлення та сповіщень про зміни.

Характеристики сховищ конфігурації:

- Строга консистентність даних між екземплярами через консенсус-протоколи;
- Спостереження за змінами (watch) для реактивного оновлення ;
- Ієрархічна організація ключів;
- Управління версіями конфігурацій;
- Виявлення запущених сервісів та моніторинг коректності їх роботи.

Типові застосування включають централізоване управління конфігурацією мікросервісів, виявлення сервісів (service discovery), розподілену координацію та блокування, управління виборами ведучого екземпляру сервісу (leader election).

Сховище подій (англ. event store) є спеціалізованою базою даних для збереження послідовності подій, що описують зміни стану програмного забезпечення. Замість зберігання лише поточного стану, сховище подій зберігає повну історію змін, що дозволяє відтворити будь-який попередній стан та аналізувати еволюцію даних. Сховища подій зберігають зміни стану системи як послідовність незмінних подій, реалізуючи патерн Event Sourcing.

Властивості сховищ подій:

- Незмінність записаних подій (immutability);
- Впорядкованість подій за часом та номером версії;
- Можливість підписки на потік подій;
- Реконструкція стану через застосування послідовності подій;
- Збереження повного аудиторського сліду.

Сховища подій використовуються у системах, що потребують повної історії змін (фінансові транзакції, медичні записи), для побудови множини проєкцій (англ. read models) з однієї послідовності подій, для аудиту та відповідності регуляторним вимогам.

Повнотекстова пошукова система є спеціалізованою системою управління даними, оптимізованою для індексування великих обсягів текстового контенту та виконання складних текстових запитів з підтримкою ранжування релевантності, морфологічного аналізу та фасетного пошуку, зараз часто і за допомогою вкладених представлень (англ. embeddings).

Ключові можливості:

- Токенізація та лінгвістичний аналіз тексту;
- Інвертовані індекси для швидкого пошуку термінів;
- Ранжування результатів за релевантністю;
- Підтримка синонімів, стемінгу, лематизації;
- Фасетний пошук та агрегації;
- Підсвічування знайдених фрагментів.

Повнотекстові пошукові системи застосовуються для пошуку в контенті (документи, статті, продукти), централізованого збирання та пошуку логів, аналітики текстових даних.

Геопросторова база даних (англ. Geospatial database) є базою даних, оптимізованою для зберігання та запиту даних, що представляють об'єкти у географічному просторі. Геопросторові бази даних підтримують спеціальні типи

даних (точки, лінії, полігони) та просторові операції (відстань, перетин, включення).

Типові просторові запити:

- Пошук об'єктів у певному радіусі від точки;
- Визначення об'єктів усередині полігону;
- Обчислення відстаней між об'єктами;
- Перевірка перетину геометричних об'єктів;
- Просторові з'єднання (spatial joins).

Геопросторові можливості застосовуються у системах геолокації (знаходження найближчих об'єктів), логістиці та доставці (оптимізація маршрутів), картографічних сервісах, аналізі географічних даних.

Об'єктне сховище (англ. blob storage) є сервісом зберігання даних, що організує дані як об'єкти, на відміну від файлових систем (файли в директоріях) або блокових сховищ (блоки у томах). Кожен об'єкт містить дані, метадані та унікальний ідентифікатор. Об'єктні сховища оптимізовані для зберігання великих обсягів неструктурованих бінарних даних.

Характеристики об'єктних сховищ:

- Горизонтальна масштабованість до петабайтів даних;
- Адресація об'єктів через URL або ключі;
- Підтримка метаданих довільної структури;
- Версіонування об'єктів;
- Економічність зберігання у порівнянні з блоковими сховищами;
- Інтеграція з CDN для глобальної доставки даних.

Об'єктні сховища використовуються для статичних активів веб-застосунків (зображення, відео, документи), резервного копіювання та архівування, дата-лейків для аналітики, зберігання логів та метрик IoT-пристроїв.

Дата-лейк (англ. Data lake) є архітектурним патерном зберігання великих обсягів сирих даних у їхньому джерельному форматі до моменту потреби в їхньому використанні. Дата-лейк дозволяє зберігати дані без попереднього визначення схеми, забезпечуючи гнучкість для різноманітних типів аналізу.

Ключові характеристики дата-лейків:

- Зберігання сирих, необроблених даних;

- Підтримка структурованих, напівструктурованих та неструктурованих даних;
- Схема-на-читання замість схеми-на-запис;
- Масштабованість до петабайтів даних;
- Інтеграція з інструментами аналітики та машинного навчання.

Основним викликом архітектури дата-лейку є те, що сирі дані зберігаються без нагляду за змістом. Для того, щоб дата-лейк робив дані використовуваними, необхідні механізми каталогізації та захисту даних. Без цих елементів дані не можуть бути знайдені або їм не можна довіряти, що призводить до “болота даних” (data swamp). Задоволення потреб ширшої аудиторії вимагає від дата-лейків наявності управління, семантичної консистентності та контролю доступу.

Дата-лейки будуються на основі об’єктних сховищ (Amazon S3, Azure Data Lake Storage) з додатковим рівнем метаданих та управління.

Lakehouse є архітектурою даних, що поєднує можливості дата-лейків (гнучкість зберігання, масштабованість) та сховищ даних (управління, ACID-транзакції, схема) через додавання рівня управління метаданими та транзакціями поверх дата-лейку.

4.6 Кластерні системи. Теорема CAP.

Розподілені системи є фундаментом сучасного програмного забезпечення, забезпечуючи масштабованість, доступність та продуктивність через розподіл обчислювальних ресурсів та даних між множиною незалежних вузлів.

Однак, розподілена природа таких систем створює фундаментальні обмеження щодо гарантій, які можуть бути надані одночасно. Розуміння цих обмежень є критичним для прийняття архітектурних рішень при проектуванні розподілених систем.

Традиційні системи управління базами даних забезпечують строгі гарантії коректності операцій через властивості ACID.

ACID є акронімом чотирьох властивостей транзакцій у системах управління базами даних: атомарність (англ. Atomicity), узгодженість (англ. Consistency), ізолюваність (англ. Isolation) та довговічність (англ. Durability). Сукупність даних властивостей забезпечує узгодженість даних.

Атомарність гарантує, що транзакція виконується повністю або не виконується зовсім. Часткове виконання транзакції неможливе — у разі виникнення помилки на будь-якому етапі, усі зміни, внесені транзакцією, скасовуються.

Узгодженість забезпечує збереження цілісності бази даних. Транзакція переводить базу даних з одного коректного стану в інший коректний стан, зберігаючи усі визначені обмеження цілісності, тригери та каскадні правила.

Ізольованість гарантує невидимість проміжних даних транзакції поза її межами. При роботі двох транзакцій з одними даними, одна з них буде призупинена до завершення другої.

Довговічність забезпечує збереження результатів успішно завершеної транзакції навіть у разі подальшого збою системи. Зміни записуються у постійне сховище та не можуть бути втрачені.

ACID-властивості є стандартом для традиційних реляційних баз даних та забезпечують сильну узгодженість, що є критичною для багатьох застосунків, особливо у сферах, де втрата або некоректність даних може мати серйозні наслідки.

Розподілені системи, що оперують на множині незалежних вузлів, часто не можуть забезпечити усі властивості ACID без значної деградації продуктивності та доступності. Це призвело до розробки альтернативного підходу BASE.

BASE є акронімом трьох принципів розподілених систем: Доступність загалом (Basically Available), м'який стан (Soft state) та Кінцева узгодженість (Eventual consistency). BASE представляє послаблені гарантії порівняно з ACID, віддаючи перевагу доступності над миттєвою узгодженістю.

Доступність загалом передбачає, що система залишається операційною навіть при часткових відмовах. Реплікація даних знижує ймовірність повної недоступності інформації, а шардинг (partitioning) розподіляє дані між множиною серверів зберігання, роблячи потенційні відмови частковими. Результатом є система, що залишається доступною навіть якщо підмножини даних стають тимчасово недоступними.

М'який стан означає, що система допускає тимчасову неузгодженість даних, делегуючи обробку таких ситуацій розробникам застосунків. На відміну від ACID-систем, що вважають узгодженість даних жорсткою вимогою, BASE-системи дозволяють даним перебувати у проміжних, потенційно неузгоджених станах.

Кінцева узгодженість гарантує, що у певний невизначений майбутній момент часу дані досягнуть узгодженого стану. На відміну від ACID-систем, що забезпечують узгодженість при завершенні транзакції, узгодженість гарантується лише "десь у майбутньому".

Вибір між ACID та BASE залежить від специфічних вимог застосунку. ACID є необхідним для систем, де критична абсолютна коректність кожної операції

(банківські транзакції, інвентаризація товарів з обмеженою кількістю). BASE є прийнятним для систем, де висока доступність та масштабованість важливіші за миттєву узгодженість (соціальні мережі, системи рекомендацій тощо). Обидва підходи можуть бути використані в різних підсистемах однієї системи в залежності від вимог.

Узгодженість даних у розподілених системах може набувати різних форм залежно від гарантій, що надаються системою.

Сильна узгодженість (англ. strong consistency) гарантує, що після успішного завершення операції запису усі наступні операції читання з будь-якого вузла системи повертають оновлене значення. Це відповідає поведінці, очікуваній у ACID-системах. Сильна узгодженість вимагає координації між вузлами при кожній операції, що може призводити до значної затримки та зниження доступності.

Кінцева узгодженість (англ. eventual consistency) гарантує, що за відсутності нових оновлень, усі репліки даних згодом досягнуть ідентичного стану. Період, протягом якого різні репліки можуть повертати різні значення, називається вікном неузгодженості. Системи з евентуальною узгодженістю віддають перевагу доступності над узгодженістю.

Вибір типу узгодженості є архітектурним компромісом. Сильна узгодженість забезпечує простішу модель програмування та передбачувану поведінку, але вимагає жертвування продуктивністю та доступністю. Кінцева узгодженість дозволяє досягати кращої продуктивності та доступності, але ускладнює логіку застосунок, що має враховувати можливість отримання застарілих даних.

Фундаментальні обмеження розподілених систем формалізовані теоремою CAP, сформульованою Еріком Брюером у 1998 році.

Теорема CAP (також відома як теорема Брюера) стверджує, що розподілена система з реплікованими даними може одночасно гарантувати не більше двох з наступних трьох властивостей: узгодженість (Consistency), доступність (Availability) та стійкість до розбиття мережі (Partition tolerance).

Три властивості теореми CAP визначаються наступним чином:

Узгодженість (англ. Consistency) — кожен запит на читання отримує відповідь що або дорівнює останньому записаному значенню, або є помилкою. Узгодженість у теоремі CAP має інше значення від ACID.

Доступність (англ. Availability) — кожен запит, що обробляється робочим вузлом системи, отримує відповідь за скінченний час. Доступність за CAP відрізняється від нефункціональних вимог про доступність.

Стійкість до розбиття мережі (англ. Partition tolerance) — система продовжує функціонувати попри втрату або затримку довільної кількості повідомлень між вузлами. Розбиття мережі (partition) розділяє систему на дві або більше підмножин вузлів, що не можуть комунікувати між собою.

Теорема CAP стверджує, що можливі лише три комбінації властивостей:

CA (Consistency + Availability) — системи, що забезпечують узгодженість та доступність, але не толерують розбиття мережі. Прикладом є традиційні реляційні бази даних з розподіленою транзакційною семантикою, які можуть функціонувати лише за умови відсутності розбиття між вузлами-репліками.

CP (Consistency + Partition tolerance) — системи, що забезпечують узгодженість та толерують розбиття мережі, але жертвують доступністю. У разі виникнення розбиття, система може блокувати певні операції до відновлення зв'язку, щоб уникнути ризику порушення узгодженості.

AP (Availability + Partition tolerance) — системи, що забезпечують доступність та толерують розбиття мережі, але жертвують узгодженістю. Система продовжує обробляти запити навіть при розбитті, приймаючи ризик тимчасової неузгодженості між вузлами.

Розглянемо систему з двох вузлів А та В, що зберігають спільну частину даних V. Нехай обидва вузли зберігають значення V_0 . Розглянемо наступну послідовність подій:

1. Виникає розбиття мережі між вузлами А та В;
2. До вузла А надходить запит на запис, що оновлює значення до V_1 ;
3. До вузла В надходить запит на читання.

У цій ситуації вузол В має лише два варіанти дій:

1. **Обрати доступність (AP):** Прийняти запит на читання та повернути значення V_0 , усвідомлюючи, що вузол А зберігає нові дані V_1 , недоступні до відновлення зв'язку. Це порушує узгодженість.
2. **Обрати узгодженість (CP):** Відхилити запит на читання, усвідомлюючи, що клієнт може не мати можливості зв'язатися з вузлом А до відновлення зв'язку. Це порушує доступність.

Неможливо одночасно повернути актуальні дані (узгодженість) та обслужити запит (доступність) у разі розбиття мережі. Вибір одного автоматично виключає інше, що доводить теорему CAP.

Важливим нюансом теореми CAP є те, що розбиття мережі не є подією, якою можна керувати — це факт роботи будь-якої розподіленої системи. Мережеві відмови відбуваються регулярно через апаратні збої, перевантаження мережі, помилки конфігурації або інші причини. Тому практичний вибір завжди лежить між CP та AP, а не між усіма трьома комбінаціями.

CA-системи можливі лише у теоретичному сценарії гарантованої відсутності мережевих збоїв, що є нереалістичним для будь-якої розподіленої системи у реальному світі. Таким чином, реальний архітектурний вибір полягає у визначенні пріоритету: узгодженість (CP) або доступність (AP) у разі виникнення розбиття мережі.

Вибір між CP та AP визначається специфічними вимогами застосунку та наслідками тимчасової неузгодженості або недоступності даних. **AP-системи** є доцільними для застосунків, де критична безперервність обслуговування, а тимчасова неузгодженість є прийнятною. **CP-системи** є необхідними для застосунків, де критична абсолютна коректність даних, а тимчасова недоступність є прийнятною.

Попри фундаментальні обмеження теореми CAP, на практиці можливим є підхід, що дозволяє системі забезпечувати узгодженість та доступність більшу частину часу, а у разі виникнення розбиття — застосовувати спеціалізовані стратегії для мінімізації негативних наслідків.

Така стратегія складається з трьох етапів:

1. **Виявлення розбиття** — застосовуються механізми моніторингу для виявлення втрати зв'язку між вузлами.
2. **Режим розбиття** — після виявлення розбиття, система входить у спеціальний режим, що може обмежувати певні операції. Наприклад, дозволяти лише операції читання, блокувати операції, що модифікують критичні дані, або застосовувати кінцеву узгодженість для некритичних даних.
3. **Відновлення** — після усунення розбиття, система ініціює процес відновлення узгодженості, що може включати синхронізацію реплік, розв'язання конфліктів та компенсацію помилок, допущених у режимі розбиття.

Для забезпечення коректного відновлення після розбиття мережі можуть використовуватись вільні від конфліктів репліковані види даних (англ. Conflict-free Replicated Data Types, CRDT).

CRDT є структурами даних, що можуть реплікуються між множиною вузлів у мережі, при цьому будь-яка репліка може бути оновлена незалежно, паралельно та без координації з іншими репліками; будь-які два вузли, що отримали однаковий набір оновлень (незалежно від порядку їх отримання), гарантовано досягають ідентичного стану.

CRDT забезпечують математичні гарантії кінцевої узгодженості через властивості комутативності, асоціативності та ідемпотентності операцій. Це дозволяє застосовувати оновлення у довільному порядку без впливу на кінцевий результат.

Література

- [1] O. Zimmermann. “Architectural Significance Criteria and Some Core Decisions Required”. (2020), url: <https://ozimmer.ch/practices/2020/09/24/ASRTestECSADecisions.html> (дата зверн. 29.07.2025).
- [2] I. Hooks, “WRITING GOOD REQUIREMENTS”, *INCOSE International Symposium*, т. 4, № 1, с. 1247—1253, 1994. DOI: <https://doi.org/10.1002/j.2334-5837.1994.tb01834.x>.
- [3] A. Marcano, *It starts with a story...* 2014. url: <https://antonymarcano.com/blog/2014/05/it-starts-with-a-story/> (дата зверн. 29.07.2025).
- [4] A. Marcano, *How the industry broke the Connextra Template*, 2016. url: <https://antonymarcano.com/blog/2016/08/how-the-industry-broke-the-connextra-template/> (дата зверн. 29.07.2025).
- [5] M. Gorman та E. Gottesdiener, *Using “Given-When-Then” to Discover and Validate Requirements*, 2012. url: <https://www.ebgconsulting.com/blog/using-given-when-then-to-discover-and-validate-requirements-2/> (дата зверн. 29.07.2025).
- [6] I. Jacobson та A. Cockburn, *Use-Case Foundation*, вер. 1.1. url: <https://alistaircockburn.com/Use%20Case%20Foundation.pdf> (дата зверн. 29.07.2025).
- [7] A. Cockburn, *Writing Effective Use Cases*, 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 2000, ISBN: 0201702258.
- [8] P. Kruchten, “The 4+1 View Model of architecture”, *IEEE Software*, т. 12, № 6, с. 42—50, 1995, ISSN: 0740-7459. DOI: 10.1109/52.469759. url: <http://dx.doi.org/10.1109/52.469759>.
- [9] L. Bass, P. Clements та R. Kazman, *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, квіт. 2003, ISBN: 0321154959. url: <https://people.ece.ubc.ca/matei/EECE417/BASS/index.html>.

- [10] М. Richards та N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Incorporated, 2020, ISBN: 9781492043454. url: https://books.google.com.ua/books?id=_pNdwgEACAAJ.
- [11] М. Fowler. "Patterns for Managing Source Code Branches". (трав. 2020), url: <https://martinfowler.com/articles/branching-patterns.html> (дата зверн. 06.10.2025).