КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Наумкін В.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Наумкін Віталій Володимирович

УДК 004.4

Порівняльний аналіз процесів розробки та адаптації моделей та шаблонів хмарних рішень та сервісів

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____ Наумкін В.В.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.ф-м.н, доц. Бойчук А.М.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Наумкіну Віталію Володимировичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Порівняльний аналіз процесів розробки та адаптації моделей та шаблонів хмарних рішень та сервісів

керівник роботи:
Бойчук Андрій Михайлович, кандидат фізико-математичних наук, доцент

затверджена наказом вищого навчального закладу від « 26 » червня 2023 року № 32/1 с

2. Термін подання студентом роботи 16.02.2024

3. Вихідні дані роботи: Формальні моделі, методи та алгоритми.

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)
   1. Огляд та аналіз хмарного програмного забезпечення
   2. Дослідження моделей хмарного програмного забезпечення
   3. Розробка шаблонів хмарних рішень та сервісів
   4. Розробка та адаптація моделей та шаблонів хмарних рішень та сервісів

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд та аналіз хмарного програмного забезпечення	26.09.2023	Виконано
2.	Дослідження моделей хмарного програмного забезпечення	20.10.2023	Виконано
3.	Розробка шаблонів хмарних рішень та сервісів	15.11.2023	Виконано
4.	Розробка та адаптація моделей та шаблонів хмарних рішень та сервісів	30.11.2023	Виконано
5.	Формування висновків	09.12.2023	Виконано
6.	Оформлення пояснювальної записки	22.12.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	11.01.2024	Виконано

Студент _____ Наумкін В.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бойчук А.М.
(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
16	Залізний трикутник обмежень	47	Співвідношення між силами служб контейнеризації
21	Розподіл ринку хмарних провайдерів	49	Запуск контейнерного образу Ubuntu із введеними змінними середовища
22	Інформаційна панель Google App Engine	53	Архітектура проекту
27	Ініціативи з оптимізації бюджету в хмарі	54	Візуалізація проміжних компонентів для випадку використання зчитування електрокардіографії (ЕКГ) у домашньому середовищі медичного працівника

27	Проблеми хмарних обчислень	55	Діаграма послідовності, що демонструє передачу повідомлень між взаємодіючими службами у випадку використання запису показань ЕКГ пацієнта
28	Проблеми хмарних обчислень за розвитком бізнесу	65	Мова шаблонів для розробки програмного забезпечення для хмарних сервісів, що зображує зв’язки між шаблонами і категоріями
32	Збій роботи соціальної мережі Twitter	71	Система на основі архітектури мікросервісу для збору та збереження показників температури середовища
34	Візуальне представлення шаблонів хмарних обчислень	72	Взаємозв'язок між обмеженнями системи обміну повідомленнями
35	Еталонна архітектура веб-хостингу Amazon Web Services	75	Зв’язок між трьома службами, що виконують маршрутизацію через систему обміну повідомленнями
37	Карта шаблонів для керування мікросервісами	77	Чотири об’єкти інфраструктури, кожен з яких містить свій сервіс
39	Карта шаблонів, що представляє мову шаблонів для стійкості хмарних програм	78	Взаємозв'язок між силами виявлення служби (сервісу)
40	Візуальне представлення каталогу контейнерів шаблонів, організованого за трьома кольоровими категоріями: розробка, час виконання та впровадження	80	Приклад конфігурації проксі
41	Екосистема хмарних інструментів, згідно з Cloud Native Computing Foundation		

АНОТАЦІЯ

Кваліфікаційна робота присвячена дослідженню моделей і виконанню порівняльного аналізу процесів розробки та адаптації моделей та шаблонів хмарних рішень та сервісів шляхом розробка методології шаблонізації процесу проектування хмарної інфраструктури рішення.

В першому розділі проаналізовано предметну область побудови хмарних рішень та сервісів. Розглянуто інженерне програмне забезпечення для хмарних рішень та описані сервісні моделі та хмарні послуги.

В другому розділі представлено моделі хмарного програмного забезпечення, наведені особливості розробки хмарних рішень. Запропоновано хмарні шаблони проектування дизайну хмарних рішень, наведена сутність мова шаблонів для мікросервісів та представлені шаблони контейнеризації та масштабування хмарних сервісів.

В третьому розділі виконано розробка та адаптація моделей та шаблонів хмарних рішень та сервісів, досліджені хмарні архітектури. Запропоновано каталог шаблонів для хмарних сервісів та емпірична оцінка проекту. Розглянуто технологію автоматизованого управління інфраструктурою хмарного сервісу з використанням шаблонів, наведено дослідження моделей шаблонів для хмарних сервісів та методологія балансування трафіку між службами хмарних сервісів.

КЛЮЧОВІ СЛОВА: ПРОЦЕС АДАПТАЦІЇ, ХМАРНІ СЕРВІСИ, ХМАРНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, МОДЕЛЬ, МАСШТАБУВАННЯ, ІНФРАСТРУКТУРА, МІКРОСЕРВІС, ШАБЛОНИ ПРОЕКТУВАННЯ, МІКРОСЕРВІС, ХМАРНА АРХІТЕКТУРА.

SUMMARY

The qualification work is devoted to the study of models and the performance of a comparative analysis of the processes of development and adaptation of models and templates of cloud solutions and services through the development of a methodology for tinplating the process of designing a cloud infrastructure solution.

In the first section, the subject area of building cloud solutions and services is analyzed. Engineering software for cloud solutions is considered and service models and cloud services are described.

In the second section, models of cloud software are presented, features of cloud solution development are given. Cloud design templates for cloud solution design are proposed, the essence of the template language for microservices is provided, and cloud service containerization and scaling templates are presented.

In the third section, the development and adaptation of models and templates of cloud solutions and services was carried out, cloud architectures were studied. A catalog of templates for cloud services and an empirical evaluation of the project are offered. The technology of automated cloud service infrastructure management using templates is considered, the study of template models for cloud services and the methodology of traffic balancing between cloud services are given.

KEY WORDS: ADAPTATION PROCESS, CLOUD SERVICES, CLOUD SOFTWARE, MODEL, SCALING, INFRASTRUCTURE, MICROSERVICE, DESIGN PATTERNS, MICROSERVICE, CLOUD ARCHITECTURE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

RPC - Remote Procedure Call;

AMQP - Advanced Message Queuing Protocol;

MQTT - Message Queue Telemetry Transport;

CERN - Conseil European pour la Recherche Nucleaire.

ВСТУП

Актуальність теми дослідження. Доступ до Інтернету став можливим у цивілізованих країнах, уможливлюючи надшвидке спілкування та співпрацю, сприяючи новому поколінню веб-додатків. Ці додатки висувають безпрецедентні вимоги до розробки програмного забезпечення, такі як відмовостійкість і масштабованість, гарантуючи, що вони завжди доступні навіть за змінного попиту. Хмарні обчислення надають ресурси, які дозволяють розробникам впоратися з цими вимогами, але вимагають зміни парадигми від традиційного підходу до розробки програмного забезпечення та, головним чином, експлуатації. Ця робота досліджує, як розробляється хмарне програмне забезпечення, від проектування до експлуатації.

Основним питанням дослідження є питання про те, як розробляється хмарне програмне забезпечення, від проектування до експлуатації. Виконаний аналіз показав брак емпірично перевірених знань у цій області які можна попвнити за допомогою набору з типових рішень повторюваних проблем (шаблонів), які об’єднуються специфікованою мовою шаблонів. Це дозволяє хмарним інженерам приймати обґрунтовані рішення під час розробки свого хмарного програмного забезпечення.

Хмарні обчислення – це нова парадигма для надання ресурсів на вимогу (наприклад, інфраструктури, платформи, програмного забезпечення, тощо) клієнтам. Поточна архітектура хмарних обчислень забезпечує три рівні послуг. По-перше, Software as Service (SaaS) надає доступ до повних програм як послуг, наприклад управління взаємовідносинами з клієнтами (CRM). По-друге, платформа як послуга (PaaS) забезпечує платформу для розробки інших додатків поверх неї, наприклад як Google App Engine (GAE). Нарешті, інфраструктура як послуга (IaaS) забезпечує середовище для розгортання, запуск і керування віртуальними машинами та сховищами. Технічно IaaS пропонує поступову масштабованість (збільшення та зменшення)

обчислювальних ресурсів і зберігання на вимогу. Завдяки низці бізнес-переваг, які пропонують хмарні обчислення, багато організацій почали створювати програми хмарної інфраструктури для підвищення гнучкості бізнесу шляхом використання гнучких і еластичних хмарних сервісів. Але перемістити додатки та/або дані в хмару непросто. Існують численні виклики, щоб використати весь потенціал які обіцяють хмарні обчислення. Ці виклики часто пов'язані з тим, що існуючі програми мають специфічні вимоги та характеристики, яким необхідно відповідати. Окрім цього, із зростанням загальнодоступних хмарних пропозицій, клієнтам Cloud стає все важче вирішити, який постачальник може забезпечити їхню якість обслуговування (QoS) вимоги. Кожен провайдер Cloud пропонує подібні послуги за різними цінами та рівнями продуктивності. Хоча певний провайдер може бути дешевим за наданням терабайтів пам’яті, оренду потужних віртуальних машин від них може коштувати дорого. Тому, враховуючи різноманітність пропозицій хмарних послуг, важливим завданням для клієнтів є виявити, хто є «правильними» хмарними постачальниками, які можуть задовольнити їхні вимоги. Часто можуть бути компроміси між різними функціональними та нефункціональними вимогами, які виконуються різними постачальниками Cloud. Це ускладнює оцінку рівнів обслуговування для різних хмарних провайдерів в такий спосіб, який вимагає якості, надійності чи безпеки додатків, що може бути забезпечено в хмарах. Тому це не так достатньо просто відкрити кілька хмарних служб, але це так також важливо оцінити, яка хмара є найбільш підходящою для обслуговування. У цьому контексті індекс вимірювання хмарних послуг Консорціум (CSMIC) визначив індекси вимірювання які об’єднані у формі Індексу вимірювання обслуговування (SMI) і важливий для оцінки хмарної служби. Ці індекси вимірювання можуть використовуватися клієнтами для порівняння різних хмарних сервісів. У цьому дослідженні виконано ще один крок вперед, запропонувавши фреймворк, який може порівнювати різні хмари постачальників на основі вимог користувачів. SMICloud дозволяють

користувачам порівнювати різні хмарні пропозиції відповідно до їхніх пріоритетів та вздовж кількох вимірів і вибирати відповідно все, що відповідає їхнім потребам. Під час реалізації релевантної моделі вирішується кілька проблем для оцінки QoS і рейтингу хмарних провайдерів. Перший як виміряти різні атрибути SMI. Багато таких атрибути змінюються з часом. Кожен окремий параметр впливає на сервіс процес відбору, і його вплив на загальний рейтинг залежить його пріоритет у загальному процесі відбору та від механізму ранжирування для вирішення проблеми присвоєння ваги характеристикам з урахуванням взаємозалежності між ними, таким чином забезпечуючи вкрай необхідний кількісний показник основа для ранжування хмарних сервісів.

Таким чином, хмарні постачальники досягли ефективності завдяки економії на масштабі, приносячи користь як постачальникам, так і клієнтам. Цей підхід дозволив невеликим компаніям і окремим особам стати конкурентоспроможними у створенні програмного забезпечення для глобального ринку. Частково це стало можливим завдяки можливості динамічно масштабувати свою систему, враховуючи поточний трафік, створений їхніми користувачами в будь-який момент часу .

Мета і завдання дослідження. Метою кваліфікаційної роботи є виконання порівняльного аналізу процесів розробки та адаптації моделей хмарних рішень і сервісів та розробка методології шаблонізації процесу проектування хмарної інфраструктури хмарного рішення.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

- проаналізувати новітні засоби побудови хмарних рішень;
- дослідити особливості розробки хмарного програмного забезпечення;
- вдосконалити існуючі моделі хмарного програмного забезпечення;

- описати технологію автоматизованого управління інфраструктурою хмарного сервісу з використанням шаблонів;
- реалізувати власні оригінальні шаблони проектування хмарних рішень та сервісів.

Об’єктом дослідження є самі процеси розробки та адаптації моделей та шаблонів хмарних рішень та сервісів.

Предметом дослідження є порівняльний аналіз процесів розробки та адаптації моделей та шаблонів хмарних рішень та сервісів.

Методи дослідження базуються на використанні формальних методів та алгоритмічних стратегій, що лежать в основі побудови хмарних рішень та їх абстрагування засобами відповідних шаблонів.

Наукова новизна одержаних результатів полягає у тому, що на основі порівняльного аналізу було запропоновано сучасні шаблони проектування для розробки програмного забезпечення хмарних сервісів, еталонну архітектуру для хмарних обчислень та каталог шаблонів.

Практичне значення одержаних результатів полягає в імплементації засобів автоматизованого управління інфраструктурою хмарних сервісів на основі адаптивних шаблонів.

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 88 сторінок. Список використаних джерел містить – 54 позиції.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ХМАРНИХ РІШЕНЬ ТА СЕРВІСІВ

1.1 Опис предметної області дослідження

Інтернет сильно впливає на наше життя. Всесвітня павутина (WWW) дає нам змогу купувати продукти, дивитися телешоу та мати доступ до всіх наших колег, друзів і родини з іншого боку смартфона, який ми носимо в кишені. Понад 53% населення світу щодня є онлайн. Ці особи є потенційними користувачами для будь-якого онлайн-бізнесу, що робить WWW привабливим каналом для компаній, щоб охопити цільову аудиторію. Ця мотивація призвела до створення понад двох мільярдів веб-сайтів.

З роками веб-сайти ставали дедалі складнішими, починаючи від статичних HTML-сайтів і закінчуючи дуже складними браузерними програмами, такими як Google Drive або Facebook. Вони також виросли, щоб керувати великими обсягами даних для десятків мільйонів користувачів. Попутно інфраструктурі та технологіям довелося адаптуватися до зростання складності, що спонукало до народження хмарних обчислень.

Орієнтуючись на глобальний ринок, командам розробників доводилося турбуватися про масштабування апаратного та програмного забезпечення. Інфраструктура масштабування вимагала запобіжних інвестицій в апаратне забезпечення, що могло бути непомірно високим для деяких компаній. Без належної підготовки стрибки трафіку можуть перевантажити інфраструктуру, що підтримує програму, внаслідок чого вона не відповідає, що є неприйнятним сценарієм, оскільки користувачі погано справляються з несправністю програмного забезпечення та швидко переходять до наступної доступної альтернативи. З глобальним охопленням веб-додатків і коли попит змінюється протягом дня, динамічне масштабування програмного забезпечення стало прихованою необхідністю.

Щоб скористатися цією можливістю, у 2006 році Amazon представив поняття хмарних обчислень. Cloud відмовився від необхідності інвестувати в інфраструктуру для роботи програмного забезпечення через Інтернет. Замість цього він запровадив модель обслуговування, де програмне забезпечення, платформи чи інфраструктура можуть надаватися на вимогу, як послуга, на основі оплати за використання, працюючи як товар, як і електроенергія. Хмарні постачальники досягли ефективності завдяки економії на масштабі, приносячи користь як постачальникам, так і клієнтам. Цей підхід дозволив невеликим компаніям і окремим особам стати конкурентоспроможними у створенні програмного забезпечення для глобального ринку. Частково це стало можливим завдяки можливості динамічно масштабувати свою систему, враховуючи поточний трафік, створений їхніми користувачами в будь-який момент часу.

Хмара дозволила розробникам працювати зі своїм програмним забезпеченням у більших масштабах, одночасно вводячи завдання роботи в такому масштабі. Для прийняття цієї нової парадигми були потрібні нові архітектури, фреймворки та інструменти.

Розробка програмного забезпечення є однією з галузей інженерії, яка розвивається найшвидше, додатковим мотивом чого є широке розповсюдження Інтернету та вибухове зростання програмного бізнесу, побудованого на його основі. Попит на нових інженерів зростає вищими темпами, ніж темпи, з якими вони закінчують навчання. Не лише бракує людських ресурсів, але також бракує високоякісних допоміжних матеріалів щодо розробки для хмари.

Враховуючи обмежені людські ресурси та відсутність надійних джерел знань, для інженерів непросто приймати обґрунтовані рішення під час створення програмного забезпечення. Щоб полегшити їх роботу, ми повинні розуміти складність проблем, які вони найчастіше вирішують. Які періодичні проблеми ми можемо спостерігати під час розробки програмного забезпечення для хмари? Чи є ключові характеристики, які впливають на

виникнення проблем? Які стратегії ми можемо прийняти для їх вирішення? Як проблеми змінюються залежно від контексту і як ми можемо адаптуватися до нього? Як ми можемо отримати ці знання та зробити їх доступними для інших хмарних інженерів? Чи знають компанії про існування цих проблем?

Пов’язані з хмарою архітектури, технології та загальні знання зросли до такого рівня, що стало складно орієнтуватися. Існує очевидна нестача досліджень, що підтримують архітектуру хмарного програмного забезпечення, а саме визначення того, що сприяє успішному хмарному програмному забезпеченню, і вказівок щодо їх оптимізації. Хоча деякі автори працюють над цим, вони рідко підтверджуються науковими доказами.

У цій роботі ми пропонуємо дослідити проблему проектування програмного забезпечення для хмари, відмовившись від необхідності багатомісячних інвестицій у дослідження та розробки або запобігаючи неоптимальним рішенням під час розробки хмарних продуктів. Ми хочемо надати розробникам емпірично підтверджені знання, які допоможуть їм приймати обґрунтовані рішення щодо їхніх хмарних архітектур.

1.2 Інженерне програмне забезпечення для хмарних рішень

Запровадження хмарних технологій зростає, але не обходиться без інженерних проблем. Інженерам-програмістам доручено розробити програмне забезпечення для хмари та забезпечити його успіх. Думаючи про це, як ми вирішуємо, як розробляти програмне забезпечення? Що робить це завдання складним? Чи можемо ми відрізнити хороший дизайн від поганого? Що впливає на наші впровадження?

Щоб відповісти на ці запитання, давайте почнемо з розуміння того, що таке дизайн у контексті програмного забезпечення. Словник Merriam-Webster визначає дизайн як:

1. «конкретну мету або намір, який має на увазі окрема особа або група»,

2. «план або протокол для виконання або досягнення чогось»,

3. «основна схема, яка керує функціонуванням, розвитком або розгортанням».

Розробка програмного забезпечення стосується цих трьох описів. Інженери повинні визначити мету продукту, який вони хочуть створити, і синтезувати план його створення, визначивши схему або конструкцію, яка підтримуватиме його розвиток.

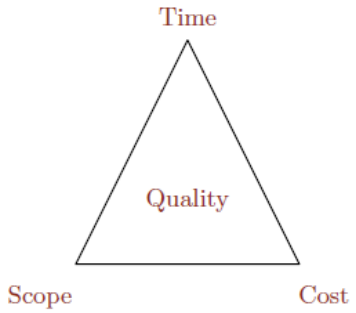


Рисунок 1.1 – Залізний трикутник обмежень

Трикутник обмежень зображує конкуруючі обмеження, які розробникам програмного забезпечення потрібно збалансувати під час розробки програмного забезпечення: час, вартість, якість і обсяг. Зміна одного завжди матиме вплив на інших.

Процес розробки керує залізним трикутником і обмеження: час, вартість і обсяг, і як вони впливають на якість, зображений на рисунку 1.1. Для успішного результату їх потрібно належним чином збалансувати. Наприклад, щоб досягти того самого результату за менший час, потрібно або збільшити ресурси (вартість) або прийняти вплив на якість.

Ми віримо, що, як і у випадку із залізним трикутником, хороший дизайн позитивно впливає на фазу розробки та забезпечує стійкий продукт. Кореляція між хорошим дизайном і кінцевою якістю програмного забезпечення може бути досягнута. Правильний дизайн призведе до вищої якості програмного забезпечення, реалізованого за менший час і з тими самими витратами. Стверджується, що якщо ви вважаєте, що хороша

архітектура дорога, спробуйте погану архітектуру, посилаючись на той факт, що поганий дизайн може призвести до збитків для програмного проекту.

Складність розробки програмного забезпечення полягає в тому, що це не точна наука. Не існує золотого правила вирішення всіх проблем. Те саме рішення може не бути життєздатним вирішенням подібної проблеми в іншому контексті або такої, яка потребує іншого балансу обмежень. Тим не менш, є незмінні якості, які позитивно впливатимуть на хмарне програмне забезпечення від його розробки до операцій, як у випадку з тестованістю, масштабованістю, розширюваністю тощо.

Найчастіше розробники програмного забезпечення розробляють рішення для проблем, над якими вже працювали інші, користуючись наявними знаннями замість того, щоб заново винаходити велосипед. Відповідно, правильні знання дизайну можуть допомогти інженерам забезпечити якість програмного забезпечення без впливу на час або вартість, що призведе до підвищення ефективності розробки для певного обсягу. Ці проекти, як правило, природним чином виникають у багатьох і незалежних підходах до тієї самої проблеми та часто мають однакові якості.

Шаблони часто пропонують альтернативні рішення, адаптуючись до балансу сил у кожному конкретному контексті. Обмеження ідентифікують характеристики проблеми та її контекст і часто протистоять одна одній, доводячи, чому проблему складно вирішити. Знайти рішення, яке врівноважує сили, необхідно, щоб рішення відповідало проблемі. Прикладом того, як сили повинні бути збалансовані, є залізний трикутник. Неможливо створити складне та якісне програмне забезпечення за короткий час і без невеликих інвестицій. Таким чином, обсяг, якість, час і вартість є конкуруючими силами, які потрібно збалансувати відповідно до контексту, щоб знайти оптимальну стратегію розвитку для того самого контексту.

Разом із шаблонами введено поняття мов шаблонів. Патерни розглядаються як елементи граматики, які, будучи пов’язаними, визначають мову, яка говорить нам, як все об’єднати разом. Мови шаблонів – це набір

шаблонів, застосовних у певній області, які можна використовувати разом для вирішення споріднених проблем.

Використання шаблонів і мов шаблонів як способу обміну знаннями пізніше було прийнято розробниками програмного забезпечення для документування ефективних рішень щодо розробки програмного забезпечення. Шаблони проектування: елементи багаторазового об'єктно-орієнтованого програмного забезпечення є найвідомішою роботою, яка використовує шаблони для документування практик проектування об'єктно-орієнтованого програмування (ООП). Цю парадигму часто використовують для навчання практикам проектування ООП у більшості програмних інженерів. Шаблони та мови шаблонів надали не лише знання, але й словниковий запас, який користувачі можуть використовувати, щоб сперечатися про свої проблеми та рішення.

Хмарні обчислення описують як інфраструктуру, яка сприяла революції WWW. Розглянемо DevOps як спосіб мислення, який мотивує автономні команди, які автоматизують контроль якості, розгортання та операції, усуваючи розподіл обов'язків і потребу в людському втручанні в операції програмного забезпечення. Нарешті, ми описуємо шаблони проектування як джерело знань для розробників програмного забезпечення, щоб оптимізувати свої дизайнерські рішення, мінімізуючи інвестиції в дослідження та розробки

WWW було винайдено в Європейській раді з ядерних досліджень (CERN) наприкінці 1980-х років. В цей час було створено перший веб-сервер і браузер. У своїй початковій формі це була колекція документів, ідентифікованих за допомогою уніфікованого покажчика ресурсів (URL), доступних через Інтернет. Документи будуть або статичними веб-сторінками, або файлами, доступ до яких можна зробити віддалено за допомогою веб-сервера. WWW порівнювалось зі складним графом, вершини якого є документами, а ребра — зв'язками між ними.

На той час зробити програмне забезпечення доступним для користувачів було клопітким процесом. Програмістам потрібно було

скомпілювати своє програмне забезпечення для певної кількості платформ, розповсюдити свої двійкові файли користувачеві, а потім він міг інсталювати програмне забезпечення на свій комп'ютер або інші пристрої. Під час цього процесу виникає кілька проблем, а саме: несумісність програмного забезпечення між платформами, дозволи користувача, недостатні можливості апаратного забезпечення або просто відсутність у користувача технічних знань для встановлення програмного забезпечення.

Протягом 1999 року WWW пройшов через велику еволюцію, яку названо Web 2.0. Web 2.0 представив контент, створений користувачами, на відміну від традиційних статичних веб-сайтів. Веб-програми були створені з використанням веб-технологій, які використовують веб-браузери як клієнти, вимагаючи лише підключення до Інтернету для взаємодії між користувачами та віддаленим сервером. Асинхронний JavaScript ще більше покращив цей досвід, дозволивши взаємодіяти з сервером без необхідності перезавантажувати веб-сторінку, що призвело до появи інтерактивних і динамічних веб-сторінок. Протягом цього часу програми були розгорнуті в попередньо виділеній інфраструктурі. Компанії повинні були інвестувати в центри обробки даних та інфраструктуру та наймати спеціалізовані команди для їх експлуатації. Попередні інвестиції були непомірними для більшості компаній, через що меншим гравцям було дуже важко розпочати бізнес в Інтернеті. Однак із зростанням Інтернету він став ідеальним каналом для розповсюдження контенту чи підтримки бізнесу. Можливість для хмарних обчислень з'явилася, коли програми повинні були стати еластичними та масштабуватися горизонтально на кількох машинах. Хмарні обчислення надають послуги для підтримки розробки, впровадження та роботи таких горизонтально масштабованих програм, задовольняючи потреби в гнучкій інфраструктурі і економічній роботі.

Amazon Web Services (AWS) представила хмарні обчислення у 2006 році як набір керованих служб, які забезпечують будівельні блоки для створення програмного забезпечення. Ці послуги надавалися як товар, який

можна було придбати за потреби, так само як воду чи електроенергію. Хмару названо новою парадигмою надання послуг на основі оплати за використання. Було усунуто більшість авансових витрат на створення IT-інфраструктури, перенісши вартість інфраструктури з капітальних витрат (CAPEX) до операційних витрат (OPEX), а також дозволив організаціям коригувати ресурси на вимогу. Назва хмара з'явилася через використання хмари як представлення Інтернету на більшості діаграм архітектури. Хмарні послуги доступні в трьох різних моделях обслуговування: програмне забезпечення як послуга (SaaS), платформа як послуга (PaaS) та інфраструктура як послуга (IaaS).

AWS вийшла на ринок із трьома послугами:

- Elastic Compute Cloud (EC2) забезпечувала обчислювальну інфраструктуру на вимогу,
- Simple Storage Service (S3) — кероване сховище файлів,
- Simple Queue Service (SQS) — службу черги повідомлень для співпраці між службами.

Усі три керувалися програмно та виставлялися рахунки за використання. Протягом багатьох років AWS додала десятки послуг до свого списку.

Незабаром виникла конкуренція з AWS. Google Cloud Platform запустила свої сервіси, а потім з'явилися сервіси від Microsoft Azure. Разом вони управляють більшою частиною хмарного ринку (рис. 1.2).

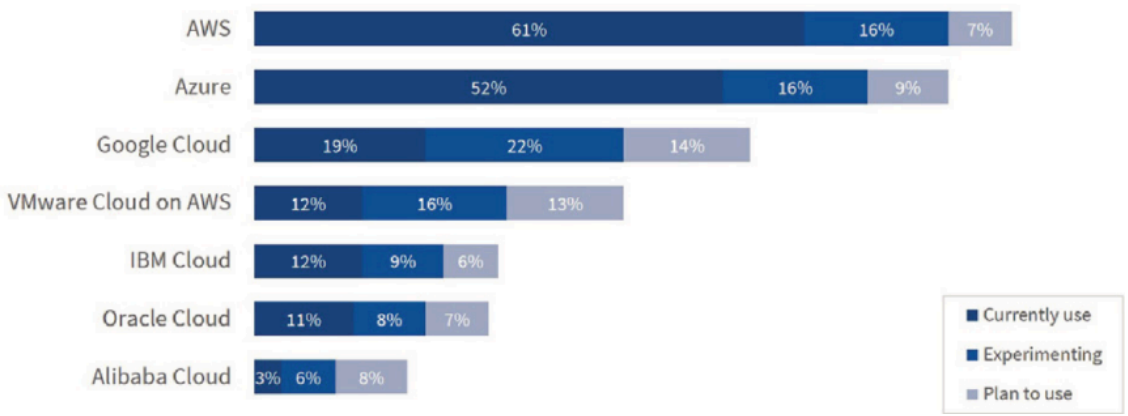


Рисунок 1.2 – Розподіл ринку хмарних провайдерів

1.3 Сервісні моделі та хмарні послуги

Хмарні обчислення дозволяють розробляти програмне забезпечення, надаючи повторно використовувані компоненти як послугу, яку часто називають «все як послуга» (XaaS). Компоненти часто поділяють на одну з трьох категорій комерціалізації послуг: SaaS, PaaS та IaaS. Усі три рівні обслуговування були описані у визначенні хмарних обчислень Національним інститутом стандартів і технологій (NIST).

Програмне забезпечення як послуга SaaS надає послуги на вимогу. Сьогодні доступний широкий спектр послуг, таких як бази даних або служби надсилання електронної пошти. Ці послуги можна адаптувати для сприяння розробці нових послуг шляхом аутсорсингу частини необхідних будівельних блоків для них. Gartner описує це як програмне забезпечення, яким володіє, доставляє та керує дистанційно один або кілька постачальників. Постачальник надає програмне забезпечення за моделлю «один-до-багатьох» усім клієнтам, які мають контракт, на основі оплати за використання або підписки на основі показників використання. У 2019 році дохід від SaaS сягнув 19,5 мільярдів доларів США, що на 18 відсотків більше, ніж у попередньому році. SaaS продовжував неухильно зростати, досягнувши загального доходу в 100 мільярдів доларів у всіх постачальників публічних хмар до 2021 року. Мотивація використовувати SaaS подібна до мотивації розробників використовувати сторонні бібліотеки під час розробки програмного забезпечення.

Платформа як послуга PaaS забезпечує повністю кероване та масштабоване робоче середовище для додатків, створених за допомогою підтримуваних мов програмування та бібліотек. Платформа керує всіма серверними компонентами, дозволяючи розробникам зосередитися лише на самій програмі. PaaS зазвичай добре інтегрується з керуванням вихідним кодом команди розробників, маючи можливість інтегрувати розгортання з їхньою системою контролю версій (VCS) , часто Git.

Google Cloud Platform була першим великомасштабним постачальником послуг PaaS з їх App Engine, випущеним у 2008 році. PaaS часто надає панелі моніторингу для виділеного середовища, показуючи такі показники, як запити в секунду або виділені машини. Команда часто може вручну змінити розподіл обладнання на цих інформаційних панелях. Інформаційна панель двигуна Google App показана на рисунку 1.3. Постачальники PaaS часто доповнюють свої пропозиції службами SaaS , які надають компоненти для розробників для створення своїх програм, наприклад бази даних або служби кешу.

Інфраструктура як послуга IaaS знаходиться на найнижчому рівні, надаючи такі інфраструктури, як віртуальні машини та балансувальники навантаження, забезпечуючи управління фактичним обладнанням. Користувачі несуть відповідальність за створення та керування своїм середовищем, а потім керують своїм програмним забезпеченням поверх нього. Це найнижчий рівень абстракції хмарного сервісу, який часто включає обчислення, сховище, балансувальники навантаження, мережі та інше обладнання (часто віртуалізоване), яке користувач повинен налаштувати для адаптації до розміщеної програми.

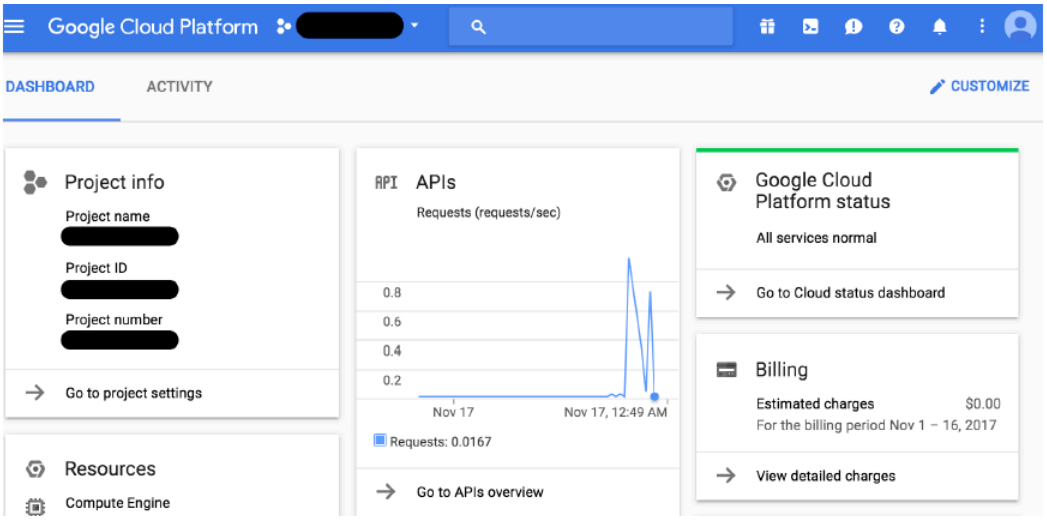


Рисунок 1.3 – Інформаційна панель Google App Engine, на якій показано вхідні запити за секунду, виділені обчислювальні екземпляри та приблизні рахунки

Багатохмарні хмарні постачальники виросли і пропонують сотні послуг із географічно надлишковою доступністю. Тим не менш, деякі програми мають особливі вимоги до хмари, які не можуть бути вирішені одним постачальником хмари. Розглянемо програму, яка через затримку потребує розгортання в Китаї та центральних Сполучених Штатах. Якщо ми розглядаємо AWS і Google Cloud, AWS присутня в Пекіні, Китай, але не в центральних США, тоді як Google Cloud має центр обробки даних в Айові, але не в Китаї. Такі додатки можуть використовувати кілька хмар або мультихмару, де один або кілька хмарних провайдерів використовуються під час розробки хмарного рішення.

Загальна стратегія розширення до мультихмари полягає в розгортанні відокремлених або реплікованих компонентів із програми в різних провайдерах. Співпраця між ними здійснюється за допомогою Інтернету та сервісного спілкування за допомогою брокера або API. Стверджується, що для оптимального використання багатохмарність потребує автоматичного керування інфраструктурою. Застосування мультихмарності не є тривіальним рішенням, оскільки воно вимагає від команди запобігання блокуванню постачальника, природної спокуси, враховуючи пропозицію SaaS , яку надає кожен постачальник, що може зменшити час розробки та складність.

Разом із загальнодоступними хмарами можна використовувати приватні хмари для підвищення конфіденційності та економічності за рахунок початкових інвестицій у обладнання, фізичного розподілу простору та людських ресурсів.

Відомі дослідження де було описано таксономію багатохмарних архітектур і проаналізовано 20 багатохмарних проектів, дійшовши висновку, що вони покращують продуктивність угоди про рівень обслуговування (SLA) і часто вимагають брокера для підключення кількох компонентів. Проект Uni4Cloud продемонстрував, як збільшити резервування за рахунок використання мультихмари шляхом розгортання програми в IaaS від двох різних постачальників. SeaClouds — це ще один дослідницький проект ЄС,

який представив інструменти для безперебійного адаптивного керування кількома хмарами.

Висновки до розділу 1

В даному розділі проведено огляд хмарного програмного забезпечення та рішень, описано предметну область дослідження, наведено інженерне програмне забезпечення для хмарних рішень та сервісні моделі.

РОЗДІЛ 2. МОДЕЛІ ХМАРНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Особливості розробки хмарного програмного забезпечення

Хмарні обчислення дозволили командам створювати програми, які охоплюють користувачів у глобальному масштабі. Культура DevOps розширює концепцію автономних команд, успадковану від гнучких методологій, щоб також керувати своїм програмним забезпеченням за допомогою автоматизації. Проте, розробляючи програмне забезпечення для хмари, інженери стикаються з безліччю нових проблем, особливостей розробки хмари, яких раніше не спостерігалося. Від необхідності масштабувати свої системи горизонтально, щоб переконатися, що система запущена та працює належним чином, або відновлюватися в разі збою, ці складнощі роблять розробку хмари нетривіальною. У цьому розділі описані існуючі найкращі методи створення програмного забезпечення для хмари. Вони можуть керувати розробниками під час розробки свого хмарного програмного забезпечення.

Експлуатація хмарних обчислень вимагає знань у певній галузі, які відходять від традиційної розробки програмного забезпечення в кількох напрямках, а саме в архітектурі програмного забезпечення та організації команди. Саме програмне забезпечення більше не є єдиним робочим елементом, створеним групою інженерів; автоматизація, забезпечення якості та операції стали однаково важливими.

Хмарні обчислення сприяли швидкому зростанню нової парадигми програмного забезпечення, що призвело до вибуху технологій і проектів програмного забезпечення, які його підтримують. Через таке швидке розширення інженерам було дуже важко залишатися на одному рівні з новими тенденціями, що вимагало від більшості значних досліджень, щоб розпочати розробку хмари. Хоча деякі з них навчилися працювати з хмарою,

їх було недостатньо, щоб задовольнити попит на інженерів, які розуміються на хмарних обчисленнях.

Хмарна безпека. Витоки даних або зловживання ними часто обговорюються, оскільки хмарні продукти експоненціально збільшують обсяг даних, які вони збирають від своїх користувачів. Закони про конфіденційність даних, такі як Європейський загальний регламент захисту даних (GDPR), разом із проблемами безпеки можуть стримувати команди розробників від тривіального переміщення своїх програм у приватні хмари, але є необхідністю для забезпечення захисту даних і порушенням, яке може знизити довіру споживачів до товару. Стверджується, що клієнти повинні довіряти своїм хмарним провайдерам. Кожен рівень хмари займається своєю безпекою, і кожен рівень повинен довіряти своїм базовим рівням, щоб бути надійним.

Сервіси сторонніх розробників нижчі в інфраструктурі як послуги (IaaS), зростають із платформою як послугою (PaaS) і найвищими з програмним забезпеченням як послугою (SaaS) , оскільки складність послуги зростає на кожному з цих рівнів. Приватні хмари, як правило, більш безпечні, ніж загальнодоступні, оскільки вони існують у спеціальному обладнанні, яке працює в ізольованих комп’ютерних мережах. Обмеження доступу до Інтернету та відсутність спільного використання ресурсів між декількома об’єктами призводить до зменшення векторів атак на нього. Конфіденційність даних у хмарі є ще однією головною проблемою. Деякі хмарні провайдери не мають зовнішнього аудиту, який забезпечує їх безпеку. Незважаючи на це, користувачам хмарних технологій доведеться довіряти свої особисті дані своїм постачальникам хмарних технологій.

Управління. Управління хмарою стосується процесу роботи хмари. Він відповідає на питання про те, як має працювати хмарна оркестровка, розглядаючи підзвітність, права прийняття рішень, ризики та управління ресурсами. У ньому розглядаються такі конкретні питання, як зміна дозволів, балансування обмежень і ризик.

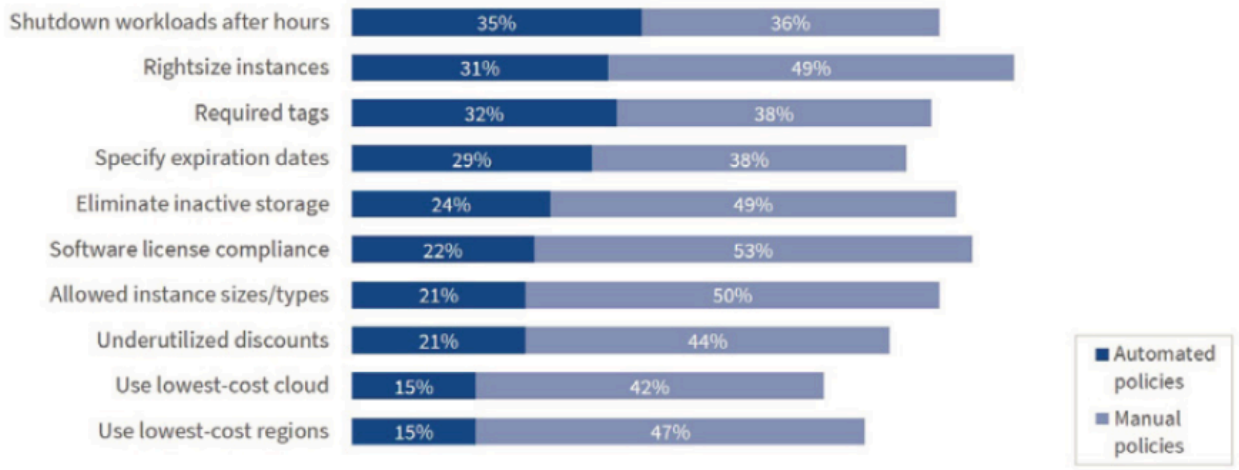


Рисунок 2.1 – Ініціативи з оптимізації бюджету в хмарі

Згідно даного рисунку 2.1 закриття робочих навантажень у неробочий час, коригування розмірів екземплярів і позначення ресурсів були найбільш відповідними автоматизованими політиками.

Хмарні витрати. Хмарні обчислення спрощують доступ до обчислювальних ресурсів у великих масштабах, але вони все ще значно впливають на щомісячний бюджет для компаній, що розробляють програмне забезпечення для хмари. Понад 50% респондентів стверджують, що витрачають на це понад 1,2 мільйона доларів. Існує кілька стратегій оптимізації, таких як річні контракти або оптимізація ресурсів, які не є тривіальними для інженера, який не знайомий з ними.

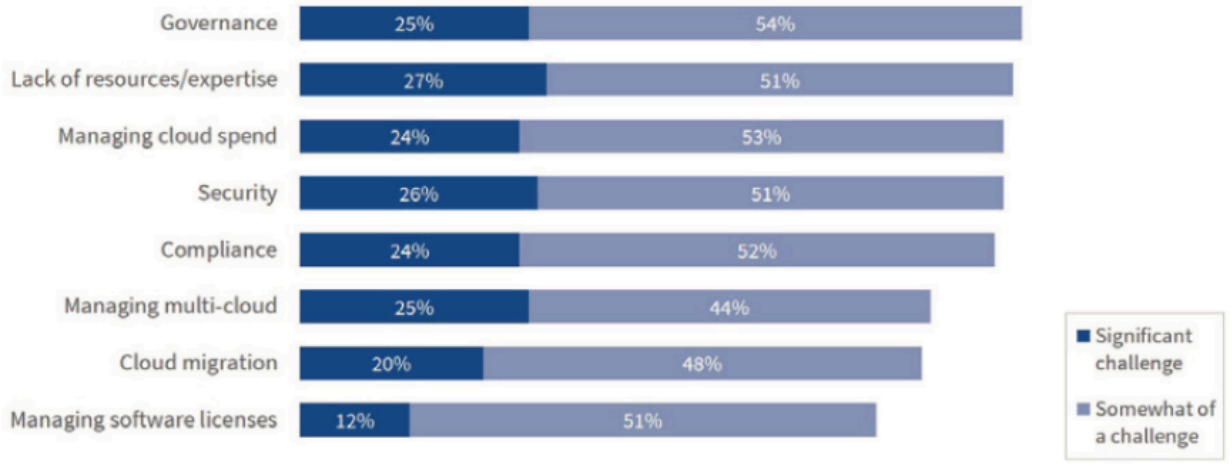


Рисунок 2.2 – Проблеми хмарних обчислень

Рисунок 2.1 перераховує найбільш прийнятні стратегії оптимізації витрат, на думку респондентів опитувань. Розглядаючи як ручні, так і автоматизовані ініціативи, коригування розмірів екземплярів є найбільш часто застосовуваною стратегією для оптимізації витрат.

Загалом брак ресурсів і досвіду є найважливішими відкритими проблемами впровадження хмарних обчислень.

BEGINNER	INTERMEDIATE	ADVANCED
1. Governance (86%)	1. Governance (78%)	1. Governance (78%)
2. Lack of resources/expertise (85%)	2. Managing cloud spend (77%)	2. Managing cloud spend (76%)
3. Managing cloud spend (84%)	3. Lack of resources/expertise (76%)	3. Compliance (76%)
4. Cloud migration (83%)	4. Security (76%)	4. Security (74%)
5. Security (82%)	5. Compliance (73%)	5. Lack of resources/expertise (73%)

Рисунок 2.3 – Проблеми хмарних обчислень за розвитком бізнесу

Ми бачимо, наскільки ці виклики важливі для респондентів на рисунку 2.2 і розбиті за рівнем знань респондентів на рисунку 2.3. І все ж найбільшою проблемою для впровадження хмарних технологій є брак ресурсів і знань для керівництва інженерами, за якими слідують управління та багатохмарне управління.

Розробка програмного забезпечення для хмари вимагає складної зміни парадигми від традиційної розробки програмного забезпечення. Тепер розробники мають враховувати вимоги, яких не існувало в минулому.

Розширюється за дизайном. При розробці програмного забезпечення з можливістю масштабування очікується, що коли програма досягне межі вертикальної масштабованості, вона зможе продовжувати масштабуватись горизонтально. За задумом монолітні служби можуть масштабуватися лише вертикально, що робить їх непридатними для масштабованих хмарних архітектур. Нові архітектури, такі як мікросервіси, мають бути адаптовані для

розділення служб на менші компоненти, які масштабуються окремо, полегшуючи роботу зі змінним попитом, який часто спостерігається в хмарних програмах. Масштабованість має стосуватися всіх компонентів програми, таких як зберігання даних, інфраструктура обміну повідомленнями тощо. Тим не менш, існують ризики превентивної оптимізації, заявляючи, що це природна еволюція для служб, які починаються як моноліти та розбиваються на мікросервіси, коли це необхідно.

Динамічна інфраструктура. Щоб скористатися перевагами архітектури мікросервісів і забезпечити економічно ефективне масштабування, інфраструктура має адаптуватися до навантаження програми, динамічно масштабуючись вгору та вниз.

Масштабованість сервісу. Працюючи у великому масштабі зі службами, які існують на десятках і сотнях серверів, непрактично керувати службами, що працюють на кожному сервері окремо. Стан серверів і використання ресурсів мають бути абстраговані, що дозволяє розміщувати служби в кластері, причому їх виділення окремому серверу є автономним. Крім того, кластер має бути автономним для виявлення несправних екземплярів служб, виконання необхідних дій для їх відновлення: або перезапустити його, або запустити на іншому сервері.

Відкриття служби. Розгортання служб у динамічній інфраструктурі створює для них необхідність відкривати один одного, щоб вони могли співпрацювати, як очікувалося. Динамічна інфраструктура ускладнює це, оскільки на момент розгортання немає фіксованої адреси служби.

Моніторинг. Програмне забезпечення виходить з ладу. Під час розгортання в хмарі програмне забезпечення працюватиме на апаратному забезпеченні з динамічним налаштуванням, можливо, на багатьох територіально розподілених серверах. Щоб переконатися, що система працює належним чином, автоматичний моніторинг кожного сервера та служби має важливе значення, щоб дозволити командам швидше виявляти та виправляти проблеми.

Обмін повідомленнями. Розподілений характер хмарних додатків вимагає інфраструктури обміну повідомленнями, яка полегшує зв’язок між службами у слабко зв’язаний спосіб для забезпечення масштабованості. Асинхронний обмін повідомленнями широко використовується і надає багато переваг, але також створює проблеми, такі як упорядкування повідомлень, керування шкідливими повідомленнями, ідемпотентність тощо.

Доступність. Доступність або відсоток часу, протягом якого система доступна, має важливе значення для дотримання контрактів і забезпечення задоволеності клієнтів і користувачів. Збої можуть виникати через програмні та апаратні помилки, зловмисні атаки та некероване завантаження системи. Хмарні програми зазвичай надають користувачам угоду про рівень обслуговування (SLA), тому програми повинні бути розроблені таким чином, щоб забезпечити їх доступність.

Надійність. Надійність вимірює ймовірність того, що система вироблятиме очікуваний результат протягом часу. Надійний сервіс працює стабільно відповідно до того, для чого він був розроблений.

Стійкість. Стійкість оцінює здатність системи обробляти та відновлюватися після збоїв, що часто називають відмовостійкою. Хмарні програми часто використовують служби спільної платформи, є багатокористувачами, конкурують за ресурси та пропускну здатність, спілкуються через Інтернет і працюють на звичайному апаратному забезпеченні, що означає підвищену ймовірність виникнення як тимчасових, так і більш постійних збоїв. Виявлення збоїв і швидке й ефективне відновлення необхідні для підтримки відмовостійкості .

Безпека. Хмарні служби під впливом Інтернету є ідеальною мішенню для хакерських атак. Програмне забезпечення має бути розроблено з урахуванням міркувань безпеки, щоб мінімізувати ймовірність зламу, тоді як інфраструктура має бути ретельно захищена, щоб запобігти несанкціонованому доступу до ресурсів і даних.

Хмарні збої: хмарне програмне забезпечення, як і будь-яке програмне забезпечення, схильне до збоїв. Ми обговорили, як надійність впливає на утримання користувачів, що означає, що невдачі можуть бути причиною припинення деяких хмарних бізнесів. Gadish визначив наступні збої як повторювані.

Людська помилка. Впровадження людської помилки в програмне забезпечення залежить не від того, чи це станеться, а від того, коли це станеться. Від розробки до експлуатації кожен день без людської помилки збільшує ймовірність того, що помилка незабаром буде внесена. Холлнагель описує людську помилку як ідентифіковану людську дію, яка розглядається як причина небажаного результату. Він оцінює стратегії оцінки надійності людини, а саме те, як їх можна передбачити. У контексті хмари людська помилка має потенційно катастрофічний вплив і навіть призвести до втрати хмарного сховища. Крім того, налагодження в Amazon Web Services (AWS) у 2017 році призвело до простою служби та оцінюваних втрат у 150 мільйонів доларів для великих американських компаній.

Помилки програми. Збій програми через помилки програмного забезпечення є ще однією регулярною причиною збою. Традиційні IT-практики можуть допомогти пом’якшити проблему за допомогою конвеєрів автоматизації для тестування та розгортання програм. Відомо, як важко виправляти помилки в хмарних програмах, враховуючи, що часто складно піддавати програмне забезпечення рівню стресу, якого воно досягає у виробництві.

Час простою хмарного провайдера. Хмарні постачальники стають дуже надійними, надаючи інструменти та стратегії для забезпечення резервування для підвищення надійності. Наприклад, AWS надає кілька зон доступності в кожному регіоні розгортання, щоб додатки могли розгортатися з резервуванням у певній території. Тим не менш, деякі проблеми неможливо запобігти, як-от природні причини, і навіть постачальники вищого рівня

можуть постраждати. Незначний збій мережі в AWS у 2015 році призвів до понад п’ятигодинного збою.

Збій в AWS 28 лютого 2017 року почався з припинення роботи Simple Storage Service (S3) і швидко погіршив роботу інших служб. Сама інформаційна панель статусу AWS не працювала, оскільки залежала від S3. Зрештою причину пояснили людською помилкою. Постраждало ще кілька хмарних додатків, а саме Medium та Slack.

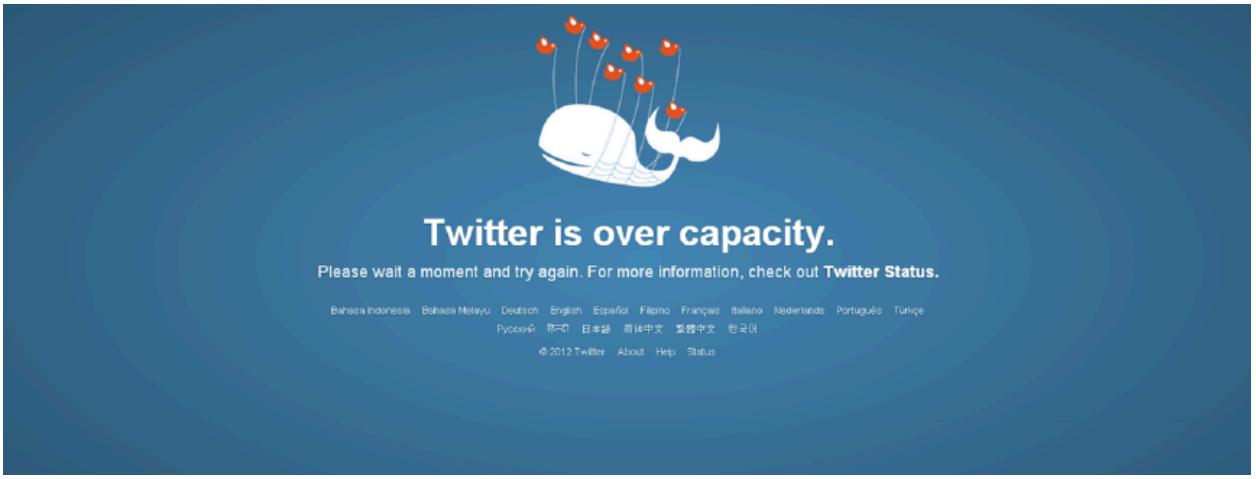


Рисунок 2.4 – Збій роботи соціальної мережі Twitter

Якість обслуговування. Визначення функціонуючої послуги залежить від наданого бізнесу. Хоча для деяких додатків може бути прийнятним збільшення часу відповіді протягом певної частини дня, для інших, таких як індустрії потокового передавання відео чи музики, будь-яка несподівана затримка або неможливість потокової передачі вмісту має критичний вплив на бізнес. Крім того, попит користувачів може завдати шкоди інфраструктурі, якщо ресурси не розподілені завчасно, налаштовані на автоматичне масштабування. Добре відома історія про збої в роботі сервісу була помічена в перші роки Twitter, коли було звичайною практикою побачити зображення, що подано на рисунку 2.4, статична сторінка, яка повідомляє, що щось не працює належним чином.

Порушення конфіденційності та безпеки. Вимоги безпеки є важливими для забезпечення конфіденційності даних клієнтів, а для деяких компаній є обов'язковими для роботи, як-от GDPR. Корпоративні клієнти часто вимагають сертифікатів управління інформаційною безпекою, перш ніж виносити рішення щодо свого бізнесу, як-от сімейство сертифікатів ISO/IEC 27000. Хмарні провайдери самі не зможуть забезпечити точність розгорнутих додатків, і, таким чином, команди розробників все ще повинні забезпечити постійну безпеку своїх додатків.

Відсутність процедур аварійного відновлення. Аварійне відновлення було звичайною практикою протягом десятиліть у фізичних центрах обробки даних. Хмарні провайдери пропонують безліч стратегій для усунення збоїв: багатозонне розгортання, автоматичне резервне копіювання, автоматичне відновлення після збою та багато іншого. Це може призвести до розслаблення команди інженерів, делегуючи відповідальність за безперервність бізнесу своєму хмарному провайдеру. Таке розслаблення неодноразово змушувало команду повірити, що у них були належні стратегії відновлення, лише щоб дізнатися, що вони не були належним чином налаштовані, коли вони були потрібні. Опис резервного копіювання та відновлення GitLab у 2017 році є чудовим прикладом цієї невдачі: команда вважала, що у них є кілька стратегій резервного копіювання, але більшість з них не працювали, як очікувалося. Після того, як людська помилка призвела до часткового стирання бази даних, їм знадобилося кілька годин, щоб знайти життєздатне рішення для відновлення, і зрештою довелося відновити базу даних із холодного сховища, що зайняло понад 18 годин.

2.2 Хмарні шаблони проектування дизайну хмарних рішень

Актуальність шаблонів серед спільнот програмної інженерії спонукала дослідників хмарних обчислень також прийняти їх щоб задокументувати спостережувані проблеми та рішення. Шаблони проектування можуть

допомогти командам розробників впевнено та швидше запустити свій хмарний додаток.

Cloud Computing Patterns описує компоненти та практики хмари. Карту шаблонів та їхніх зв'язків зображено на рисунку 2.5. Шаблони впорядковано за такими категоріями.

Основи хмарних обчислень. Описує моделі хмарних служб і стратегії розгортання, докладно пояснюючи, як вибрати між ними для різних програм.

Хмарні пропозиції. Описує послуги, доступні через хмарні постачальники, і те, як їх використовувати для створення програм.

Архітектури хмарних додатків. Докладно описано, як програми мають бути організовані в хмарі.

Управління хмарними додатками. Уточнює, як слід керувати програмами, щоб гарантувати, що вони безперервно працюють належним чином.

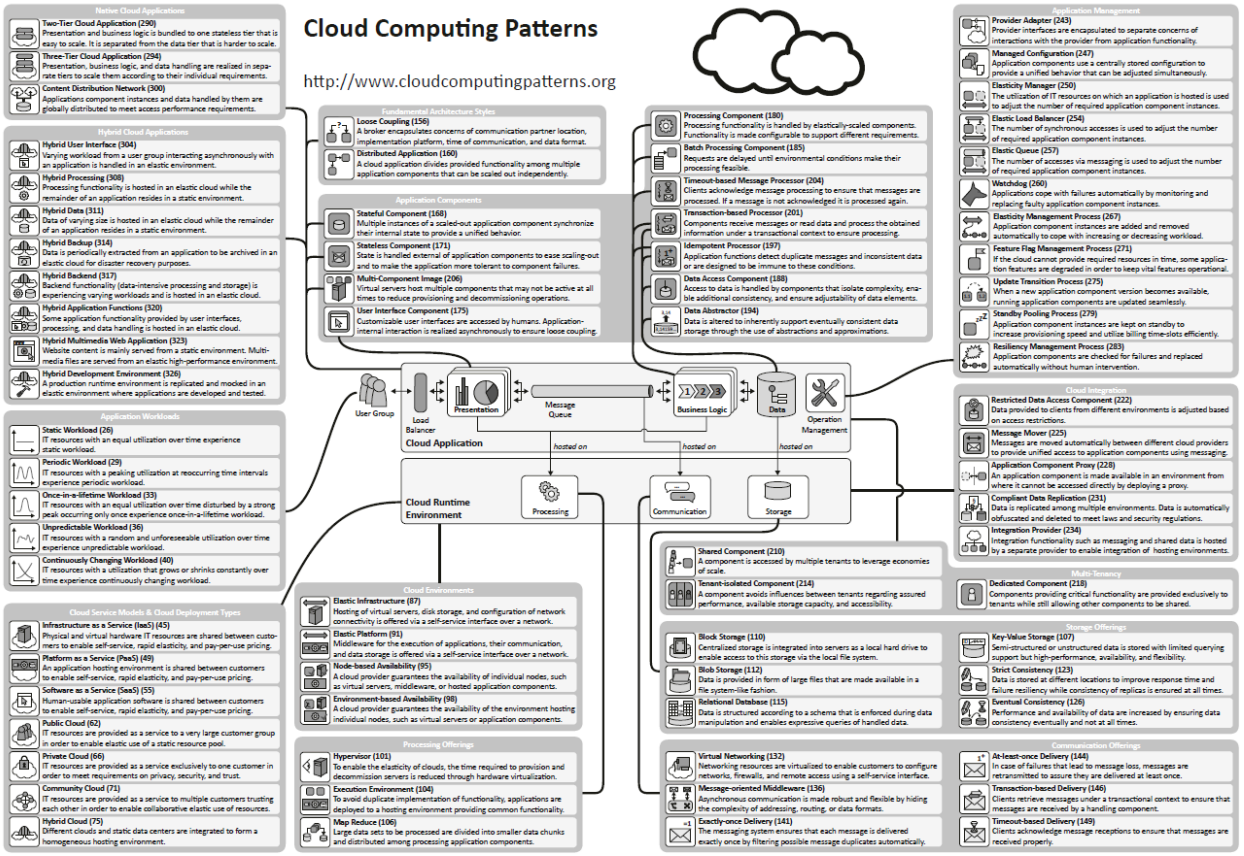


Рисунок 2.5 – Візуальне представлення шаблонів хмарних обчислень

Композитні хмарні програми. Використовує кілька шаблонів разом для вирішення складних сценаріїв.

Ці шаблони складаються з анотації до шаблону, проблеми у формі запитання, контексту, рішення, результату, варіації рішення, якщо це можливо, пов’язаних шаблонів і відомих способів використання. Візуальне представлення часто підтримує рішення.

AWS Reference Architectures — це репозиторій еталонних архітектур Amazon, що описує запропоновані рішення для конкретних сценаріїв додатків, таких як розміщення веб-додатків або пакетна обробка. Наразі Amazon пропонує загалом 16 еталонних архітектур. Кожна еталонна архітектура візуально описана, ідентифікуючи служби, які складають рішення, і те, як вони взаємодіють одна з одною. Ці вказівки не написані у формі шаблонів і не залежать від контексту, де може бути реалізоване рішення.

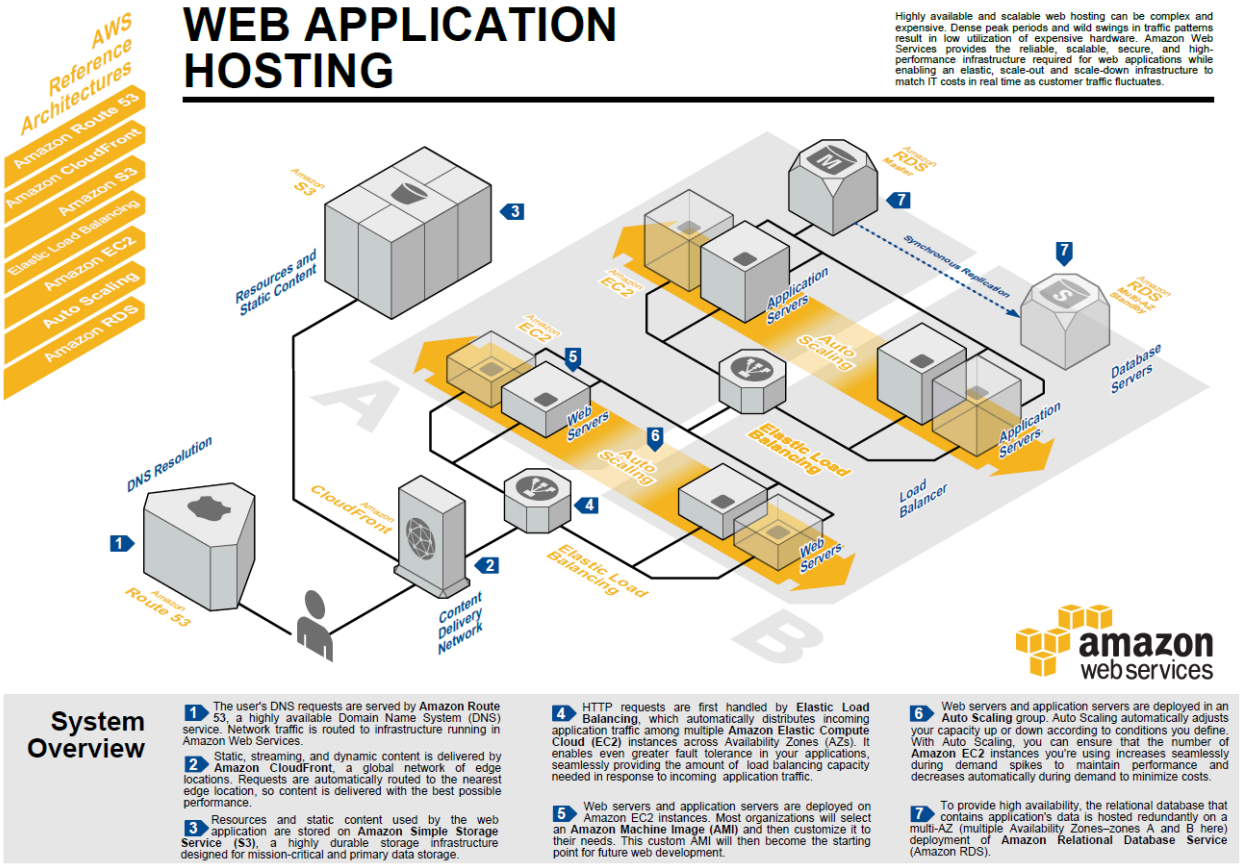


Рисунок 2.6 – Еталонна архітектура веб-хостингу Amazon Web Services

На рисунку 2.6 зображено еталонну архітектуру веб-хостингу AWS. Кожна послуга, що входить до складу рішення, коротко описана внизу.

Таблиця 2.1

Інформація про перші дванадцять шаблонів проектування Azure

Pattern Name	Description
Cache-Aside	Load data on demand into a cache from a data store
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource
Command and Query Responsibility Segregation	Segregate operations that read data from operations that update data by using separate interfaces
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain
External Configuration Store	Move configuration information out of the application deployment package to a centralized location
Federated Identity	Delegate authentication to an external identity provider
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service validates and sanitizes requests, and passes requests and data between them
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries

Таблиця 2.2

Інформація про наступні дванадцять шаблонів проектування Azure

Pattern Name	Description
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances
Materialized View	Generate prepopulated views over the data in one or more data stores when the data is not ideally formatted for required query operations
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes to smooth intermittent heavy loads
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed
Runtime Reconfiguration	Design an application so that it can be reconfigured without requiring redeployment or restarting the application
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources
Sharding	Divide a data store into a set of horizontal partitions or shards
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client
Throttling	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

Наразі Azure надає список із 24 шаблонів проектування для хмарних обчислень. Шаблони проектування Azure відповідають традиційній структурі шаблону, включаючи контекст і опис проблеми, рішення, опис проблем і міркувань під час прийняття шаблону, приклад у вигляді коду або описаний за допомогою діаграм, а також список пов'язаних шаблонів. У деяких випадках доступний приклад для завантаження. Анотації шаблонів наведено в таблицях 2.1 і 2.2.

2.3 Мова шаблонів для мікросервісів

На даний час мова шаблонів складається з більш ніж 40 шаблонів, зображених на рисунку 2.7.

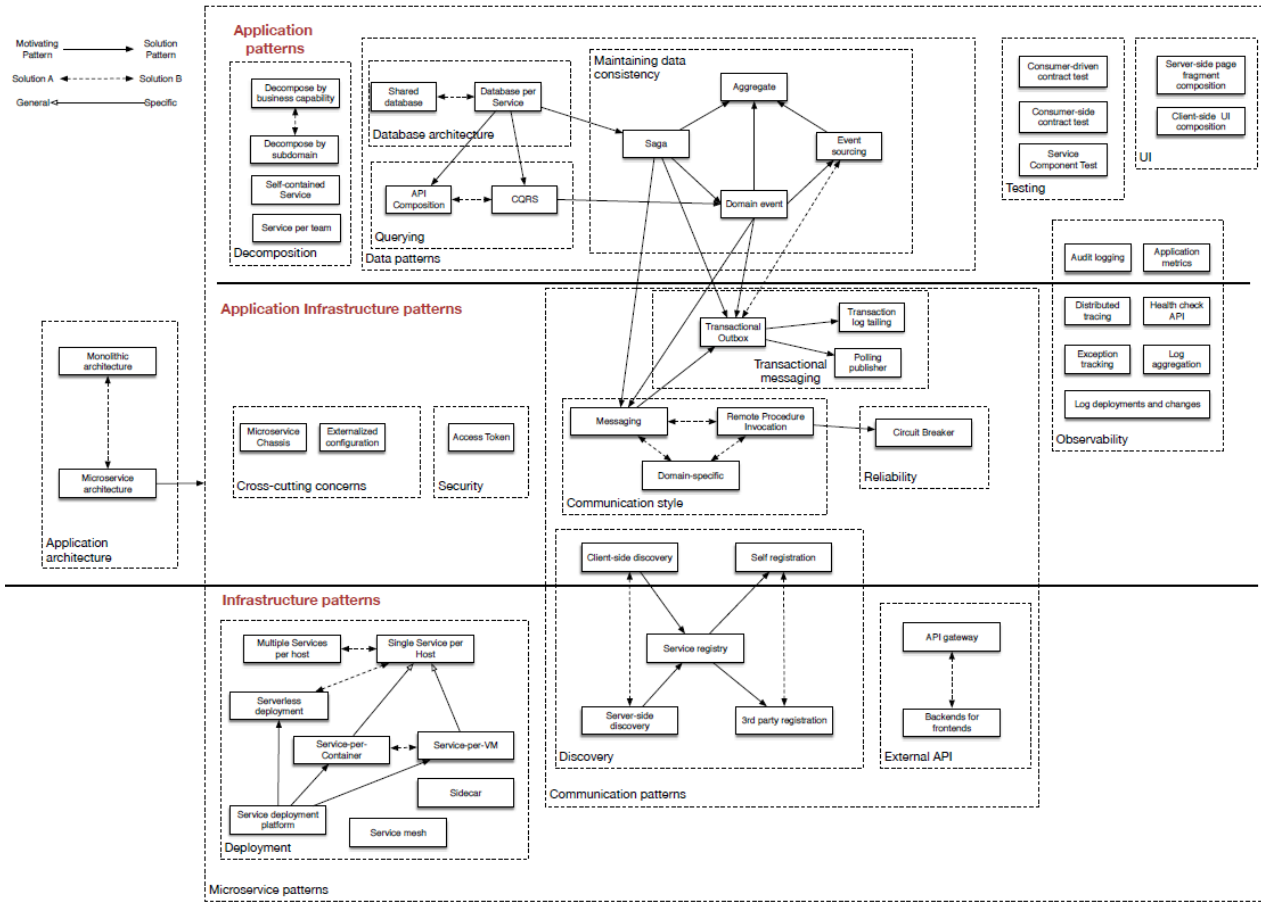


Рисунок 2.7 – Карта шаблонів для керування мікросервісами

Мова шаблонів постійно розширюється завдяки знанням, зібраним консалтинговими службами щодо архітектури мікросервісів. Веб-ресурс містить опис шаблонів у формі шаблону, використовуючи проблему, сили, рішення, приклад, результуючий контекст і пов’язану форму шаблонів. Часто шаблони в цьому форматі не описуються, а натомість демонструється, як їх можна реалізувати технічно, включаючи кілька прикладів вихідного коду та діаграм архітектури.

Існує узагальнення шаблонів та антишаблонів у життєвому циклі програмного забезпечення. Воно представляє шаблони для керування конфігурацією, тестування, розробки, розгортання, збирання та сценаріїв розгортання, розгортання та випуску додатків, інфраструктури та середовищ, даних. Наприклад, для паралельних тестів шаблоном є паралельне виконання кількох тестів на екземплярах апаратного забезпечення, щоб зменшити час виконання тестів і антишаблон виконання тестів на одній машині чи екземплярі. Запуск залежних тестів, які не можна запускати паралельно. Дюваль описує п’ятдесят шаблонів із цією стратегією, яка може підвищити обізнаність розробника щодо цих рішень і як запобігти помилкам реалізації за допомогою опису антишаблонів. Тим не менш, цих підсумків недостатньо, щоб деталізувати, як розробник міг би продовжити впровадження.

Архітектури SOA можна розширити за допомогою інфраструктури, наданої хмарними обчисленнями, для створення більшої та складнішої програми. Також можна використовувати кілька хмар, щоб збільшити подальшу надмірність і, отже, доступність хмарної програми. Описується потребу в кількох онтологіях, які повинні бути визначені та прийняті хмарними провайдерами для опису їхніх хмарних послуг, а саме щодо зберігання, обчислення та зв’язку, полегшуючи розподіл послуг між хмарними провайдерами. Тому переходять концепції хмарних брокерів, які могли б розподіляти ресурси з кількох хмар, описаних такими онтологіями, сприяючи переносимості та сумісності хмари. Еталонна архітектура багатохмарної програми описана та продемонстрована в прототипі, який

використовує Google Cloud для обчислень і Microsoft Azure для бази даних SQL. Цю концепцію незначно прийняли хмарні провайдери. Як приклад, AWS надає Service Broker, який дозволяє відкривати служби AWS безпосередньо через програми сторонніх розробників, такі як Red Hat OpenShift.

Також існує 22 моделі в концепції Patterns of resilience, карта шаблонів полегшує навігацію, як показано на рисунку 2.8. Ця мова є особливо актуальною для забезпечення словника для обговорення стійкості, оскільки шаблони недостатньо описані для застосування командою розробників.

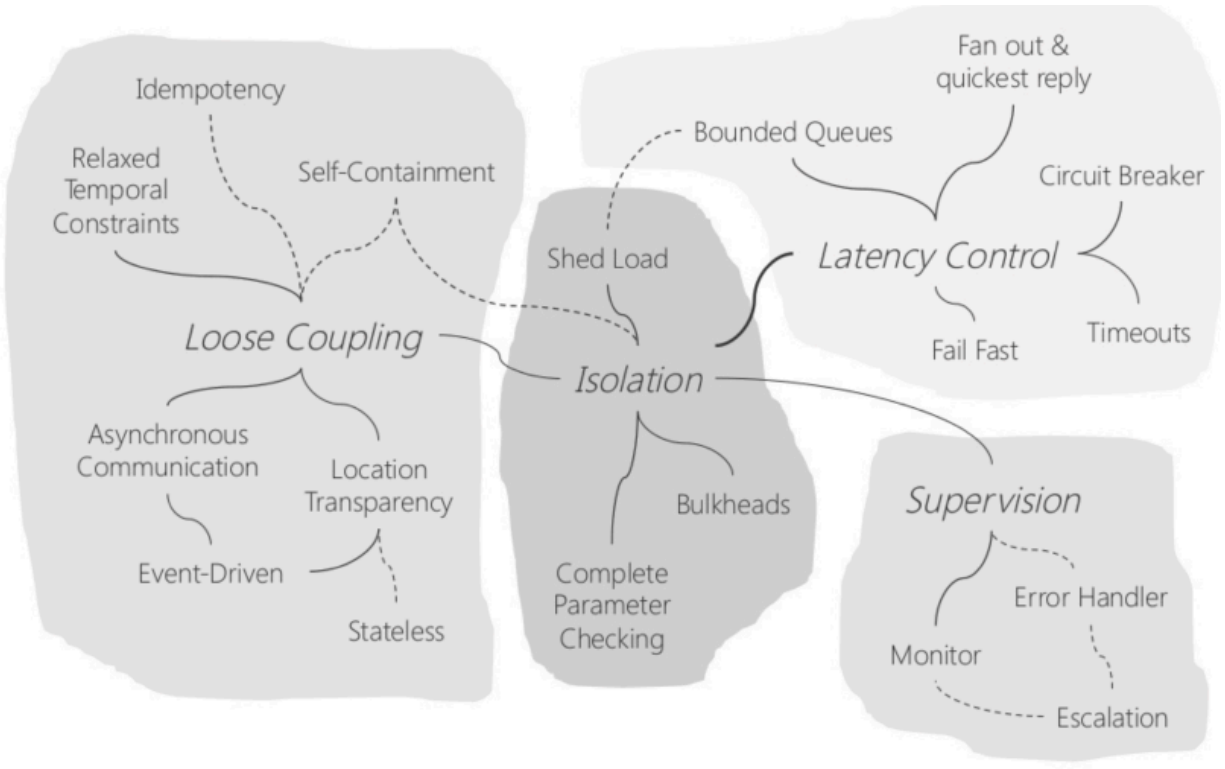


Рисунок 2.8 – Карта шаблонів, що представляє мову шаблонів для стійкості хмарних програм

Також є відомим каталог шаблонів, призначений для використання контейнерів. Шаблони організовані за трьома категоріями: розробка, із 6 шаблонами, 2 шаблонами в розповсюдженні та 10 шаблонами для виконання.

Кожен шаблон коротко описує контекст, пов’язаний з контейнером, із коротким описом і схемою, що демонструє його використання (рис. 2.9).

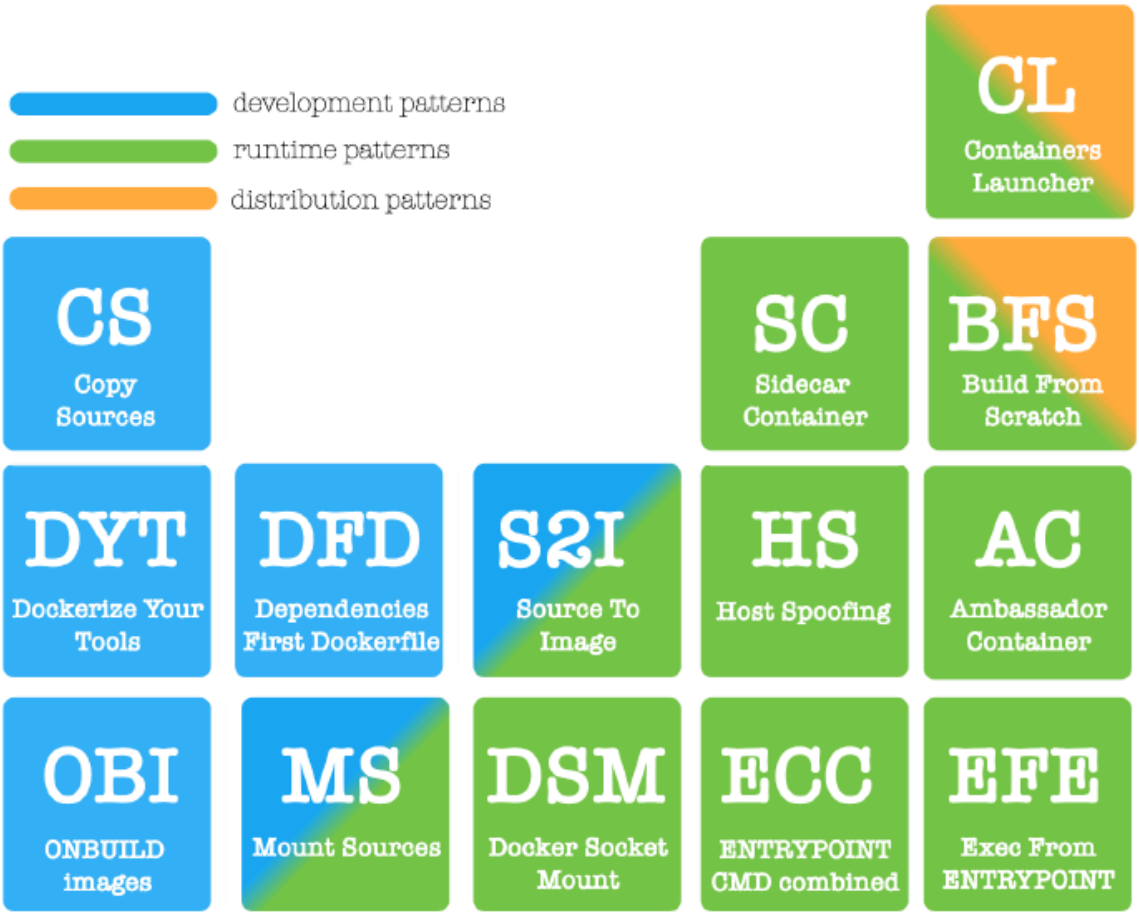


Рисунок 2.9 – Візуальне представлення каталогу контейнерів шаблонів, організованого за трьома кольоровими категоріями: розробка, час виконання та впровадження

Cloud Native Landscape – це ініціатива Cloud Native Computing Foundation, метою якої є визначення та класифікація найбільш відповідних інструментів для підтримки розробки хмарних технологій.

На сьогоднішній день вони зібрали загалом 1171 інструмент, класифікований за визначенням і розробкою додатків, оркестровкою та керуванням, часом виконання, ініціалізації, платформами, можливістю спостереження та аналізом, безсерверними та спеціальними технологіями використання. Екосистему зображено на рисунку 2.10.

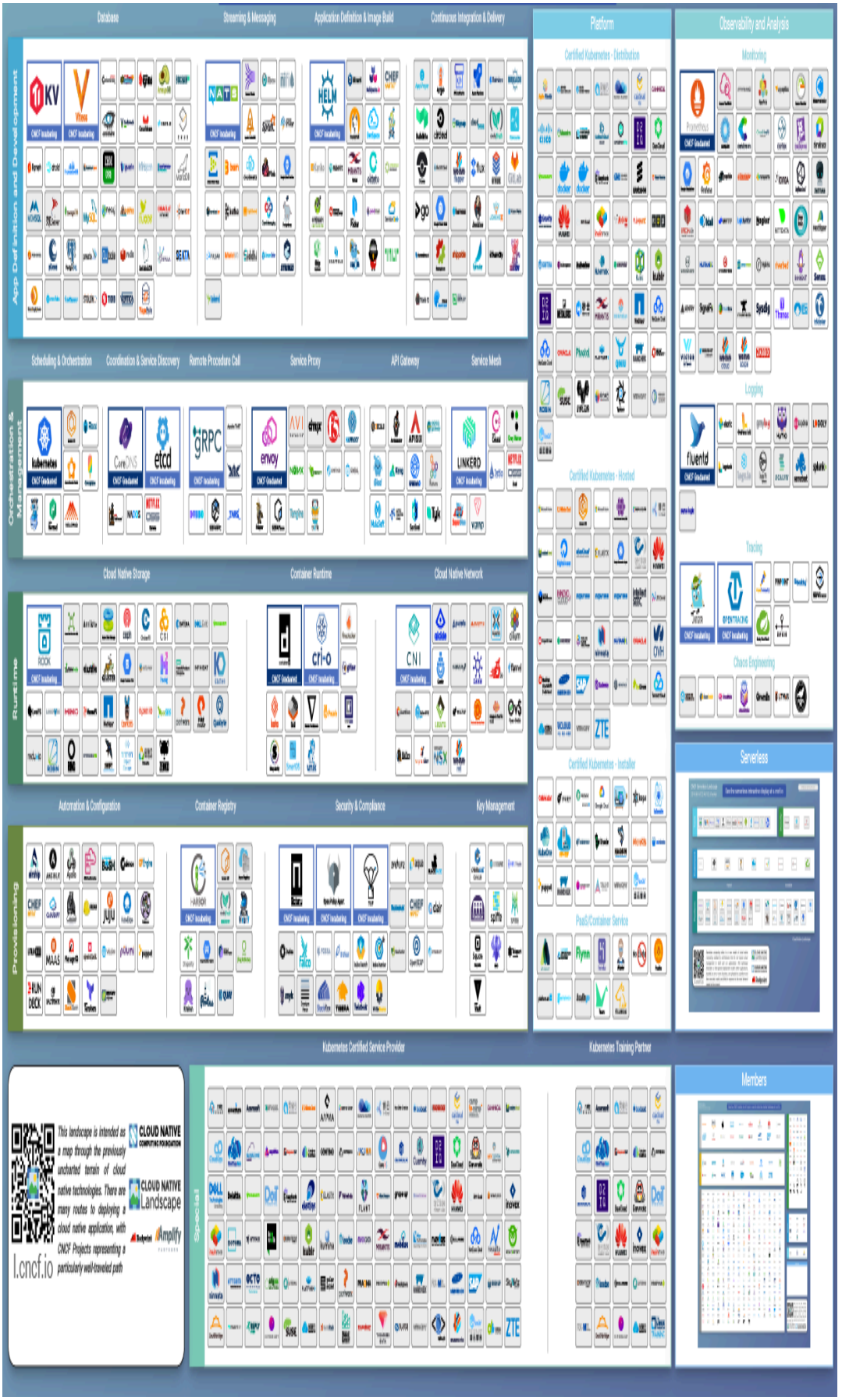


Рисунок 2.10 – Екосистема хмарних інструментів, згідно з Cloud Native Computing Foundation

Хмарні шаблони проектування є визнаним активом для розширення можливостей інженерів для створення кращого програмного забезпечення. Незважаючи на наявність таких знань, однією з головних проблем розробки хмарних технологій все ще є брак ресурсів і досвідчених інженерів для створення хмарних додатків.

На даний час доступні знання величезні, однак часто занадто суворі та не адаптуються до різних контекстів. Крім того, автори прагнуть документувати знання з власного досвіду, не перевіряючи їх достовірність іншими фахівцями. Наскільки відомо, не було емпіричних досліджень, які б оцінювали вплив шаблонів як керівних принципів для визначення архітектури хмарних програм. Вважається, що шаблони є гарною стратегією для накопичення знань про дизайн для хмари, а мова шаблонів для хмари ідеально підходить для підтримки дизайнерських рішень. Для того, щоб більшість професіоналів могли застосовувати такі шаблони, вони повинні виходити за межі поточного рівня техніки, залучати сили та альтернативні рішення.

Хмарні обчислення вимагали від розробників програмного забезпечення значного переходу до адаптації до нових вимог. Опитування RightScale показує, що брак ресурсів або досвіду є найбільш актуальною проблемою впровадження хмари. Це є фактором у введенні нових вимог до програмного забезпечення, таких як масштабованість за проектом, динамічна інфраструктура, оркестровка послуг, доступність, надійність, стійкість та інші. З багатьох причин збої в хмарі є звичайною реальністю серед практиків через людські помилки, помилки, простої провайдера, безпеку, відсутність процедур відновлення тощо.

Хороші знання та практики проектування програмного забезпечення можуть покращити цей сценарій, а шаблони забезпечують ідеальний спосіб обміну цими знаннями.

Шаблони хмарного дизайну можуть допомогти командам розробників проектувати архітектури, які швидше задовольняють вимоги хмари, скорочуючи необхідний час дослідження.

2.4 Шаблони контейнеризації та масштабування хмарних сервісів

Хмарне програмне забезпечення вимагає інфраструктури, де воно виконується. У минулому таке середовище вимагало придбання апаратного забезпечення, налаштування всього цього, включаючи операційну систему, встановлення всіх залежностей, а потім встановлення самого програмного забезпечення. Сьогодні більшість хмарних провайдерів використовують віртуалізацію, що дозволяє створювати та видаляти віртуальні машини на вимогу за допомогою API. Віртуальні машини надаються як майже безмежний ресурс, що полегшує розподіл обчислювальної потужності на вимогу. Існують також платформи для налаштування приватних хмарних рішень, які дозволяють такий самий динамічний розподіл ресурсів поверх приватних кластерів «голого металу» за допомогою схожого API. Ця категорія представляє п’ять шаблонів для нагляду:

- контейнеризація;
- менеджер агрегації (оркестровки);
- автоматичне відновлення;
- планувальник завдань;
- збій.

Створення середовища розробки або виробництва вручну є трудомістким процесом. Імовірність помилки висока, враховуючи зазвичай велику кількість залежностей і необхідних конфігурацій. Крім того, вони забруднюють хост, можливо, не дозволяючи йому розміщувати декілька програм. Хоча віртуалізація може створити портативне середовище всього стека апаратного та програмного забезпечення, вона завжди віртуалізує весь стек апаратного та програмного забезпечення, що дуже вимагає ресурсів.

Контейнеризація є кращою альтернативою, що дозволяє створювати незмінні, відтворювані, портативні та безпечні середовища виконання програмного забезпечення. Контейнери значно легші, ніж віртуалізація з повним стеком, оскільки немає необхідності віртуалізувати рівень операційної системи. Контейнери запобігають забрудненню хоста залежностями та конфігураціями, полегшуючи керування та розгортання в масштабі. Цей підхід також необхідний для індивідуального масштабування кожної послуги.

Програмне забезпечення для розширення можливостей інфраструктури в хмарі зазвичай нестабільне та динамічно розподіляється. Таким чином, оркестровка відіграє життєво важливу роль у динамічному визначенні налаштувань виконання та адаптації програмного забезпечення для роботи з нею.

Сервери в кластері відрізнятимуться деталями апаратного забезпечення. У той час як деякі можуть мати більше центрального процесора (CPU), інші можуть мати більший обсяг доступної оперативної пам’яті (RAM). Не всі послуги однакові. Таким чином, їх потрібно розташувати разом із обладнанням, яке краще відповідає їхнім вимогам. Крім того, деякі служби потрібно розміщувати на одному хості через кілька причин, наприклад затримку. Розподіл послуг не є тривіальним завданням. Менеджер агрегації може абстрагувати базову інфраструктуру, що складається з різної кількості серверів з різнорідними ресурсами, і автоматично вирішувати розподіл послуг між апаратним забезпеченням.

Асинхронні завдання, такі як обслуговування бази даних, надсилання електронних листів або резервне копіювання, часто потрібні, щоб забезпечити виконання завдань у найкращий можливий час. Вони можуть працювати з заданою частотою або в один момент часу. Планувальник завдань може бути використаний для оркестрування виконання цих програм у кластері та оцінки їх результату, створюючи звіти про помилки, коли це необхідно.

Це припущення ще більш актуальне під час організації програмного забезпечення в хмарі, враховуючи його типово великий масштаб. Визнаючи, що неможливо запобігти збою програмного забезпечення, нагляд гарантує, що служби працюють належним чином, виконуючи належні дії для їх відновлення у разі збою.

Сервіси, що працюють всередині контейнерів, повинні бути стійкими у разі збою має відбутися автоматичне відновлення. Використовуючи незмінність контейнерів, контейнер автоматично перезапускається, щоб спробувати відновити службу щоразу, коли виявляє несправність. Розширені стратегії можуть бути застосовані для відновлення послуги або набору служб, наприклад перезапуск списку служб у певному порядку. Менеджер оркестровки повинен визначити найкращу стратегію для кожного сценарію.

Механізми підвищення стійкості програмного забезпечення можна побудувати, визнавши, що програмне забезпечення дає збій. Роблячи це, розробники можуть сподіватися, що система відновиться за неочікуваних сценаріїв, але не можуть оцінити свою довіру до них, не перевіривши несподівані сценарії. Щоб забезпечити надійність і відмовостійкість, механізм ін’єкції відмов може періодично або постійно інжектувати несподівані події в системі, оцінюючи, чи продовжує вона вести себе належним чином. Ін’єкція помилок може оцінити надійність шляхом ін’єкції неочікуваних значень у службу та спостереження за будь-якою неочікуваною поведінкою. Відмовостійкість можна перевірити випадковим вимкненням серверів, гарантуючи, що вони негайно масштабуються, не впливаючи на якість обслуговування.

Шаблон Контейнерізації. Розгортання служби на хості з’єднує її з операційною системою, можливо, створюючи побічні ефекти з іншими службами на тому самому хості або з самим хостом. Використовуйте контейнер, щоб упакувати службу та її залежності та активувати її ізольоване програмне розгортання.

Розгортання служби на хості поєднує її з операційною системою, можливо, створюючи побічні ефекти з іншими службами на тому самому хості або на самому хості. Використовуйте контейнер, щоб упакувати службу та її залежності та активувати її ізольоване програмне розгортання.

Сучасне апаратне забезпечення з багатоядерними та багатопроцесорними архітектурами створене для одночасного виконання кількох програм. Хмарні обчислення часто використовують спільний доступ до ресурсів для виконання кількох служб на одному хості. Спільне використання операційної системи хоста з розміщеними службами може призвести до несумісності програмного забезпечення між ними або швидко захаращити хост, оскільки він повинен змінити свою файлову систему, щоб відповідати залежностям кожної служби. Це викликало потребу в ізольованих середовищах. Віртуалізація з повним стеком швидко стала де-факто стандартним підходом до забезпечення спільного використання ресурсів, дозволяючи виконувати служби у виділеній установці операційної системи. Паравіртуалізація ще більше покращила цей підхід, надавши апаратні ресурси безпосередньо віртуалізованому середовищу. Тим не менш, ізоляція досягається за рахунок збільшення вартості використання апаратного забезпечення, необхідного для віртуалізації стека операційної системи в кожному розміщеному середовищі.

Розглянемо веб-додаток, який має три служби:

1. HTTP-сервер.
2. Базу даних.
3. Службу кешування об’єктів.

Ці служби спільно використовують деякі основні бібліотеки, але кожна залежить від різних версій. Команда розробників використовує кілька різних дистрибутивів Linux для розробки, але виробничі середовища використовують інший. Усі три служби мають бути розгорнуті на тимчасовому хості з метою тестування, а потім розгорнуті в виробниче середовище з відповідними конфігураціями. Обслуговувати кілька середовищ

вручну, зберігаючи базову інфраструктуру відокремленою від програмних залежностей, непросто.

Розглянемо ймовірну проблему. Розгортання служби на хості з'єднує її з операційною системою, можливо, створюючи побічні ефекти з іншими службами на тому самому хості або з самим хостом. Розгортання програмного забезпечення, як правило, поєднує служби з хост-середовищем, модифікуючи його відповідно до своїх потреб. Під час розміщення кількох служб, які спільно використовують ресурси, а саме файлову систему, центральний процесор, пам'ять і доступність мережі, може спостерігатися несподівана поведінка, оскільки вони конкурують за ці ресурси. Крім того, існують ситуації, коли дві служби не можуть співіснувати в одному середовищі через несумісні залежності, що вимагає спеціального середовища для кожної служби.

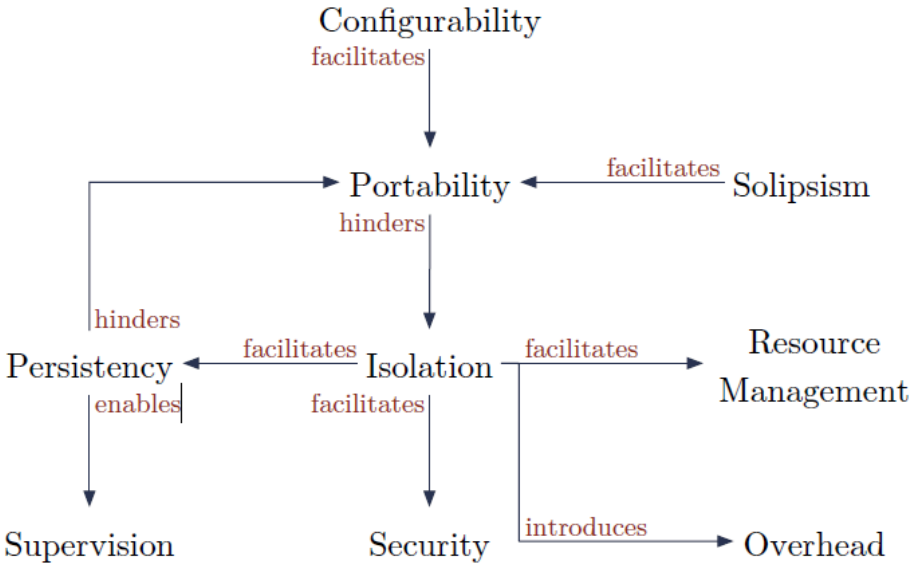


Рисунок 2.11 – Співвідношення між силами служб контейнеризації

Наступні обмеження, зображені на рисунку 2.11, необхідно збалансувати при розгляді прийняття цього шаблону:

- управління ресурсами. Невикористання всіх ресурсів на сервері є нерентабельним, а надмірне виділення служб погіршить їх продуктивність;

- накладні витрати. Відокремлення служб від операційної системи може призвести до накладних витрат на обчислення;
- нагляд. Необхідно відстежувати стан служби, запускаючи відновлення в разі збоїв;
- ізоляція. Встановлення залежностей змінює хост, що може призвести до побічних ефектів з іншими службами на тому самому хості;
- портативність. Програмне розгортання системи вимагає легкого розгортання упакованого програмного забезпечення в різних середовищах;
- конфігурація. Програмне розгортання системи вимагає стратегії конфігурації під час виконання;
- безпека. Різні підходи до ізоляції вводять різні рівні безпеки за замовчуванням;
- наполегливість. Зберігати дані на хості після закінчення терміну виконання служби, можливо, повторно використовувати їх у майбутніх виконаннях.

Рішення. Треба використати контейнер, щоб упакувати службу та її залежності та активувати її ізольоване програмне розгортання. Віртуалізація з повним стеком забезпечує ізольоване середовище для запуску програмного забезпечення. Незважаючи на це, вартість віртуалізації операційної системи для кожного середовища створює значні накладні витрати на ЦП, пам’ять. Портативність також обмежена, враховуючи збільшення використання диска. Таким чином, цей підхід не є оптимальним рішенням для хмарного програмного забезпечення.

Краще рішення існує у віртуалізації на рівні операційної системи, також відомої як контейнери. Контейнер — це самостійне ізольоване середовище з віртуальною файловою системою, мережею та розподілом ресурсів, яке виконується в операційній системі хоста.

Контейнер можна створити та запустити програмним шляхом, за допомогою конфігурацій, наданих внутрішньому програмному забезпеченню як змінних середовища, що робить його переносним між хостами. Суворий

розподіл ресурсів гарантує, що контейнер не буде перевикористовувати наявні апаратні ресурси. Рисунок 2.12 демонструє, як налаштувати та надрукувати змінні середовища для контейнера.

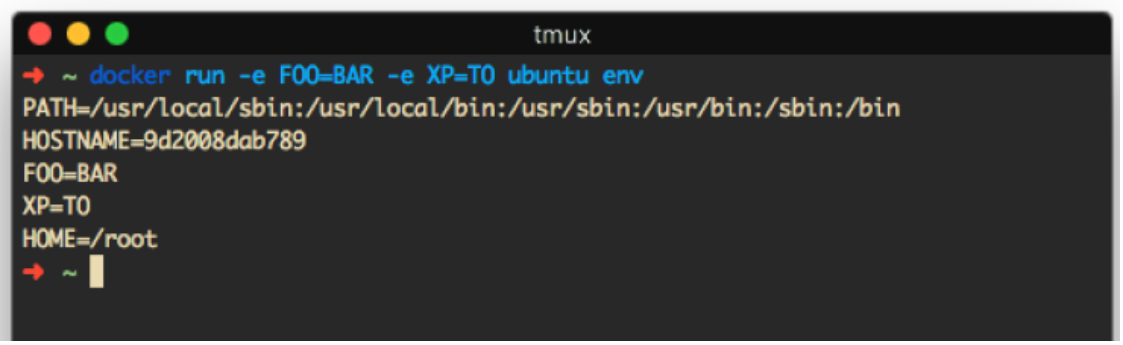


Рисунок 2.12 – Запуск контейнерного образу Ubuntu із введеними змінними середовища

Постійне сховище можна налаштувати в контейнері, відкриваючи файли або папки з хост-сервера всередині контейнера. Доступ до файлової системи обмежений. Коли контейнер видаляється з хосту, усі його дані також видаляються. Залишаються лише папки, відкриті для контейнера, якщо такі є.

У разі невдачі він може перезапустити себе з тими самими конфігураціями та чистим середовищем.

Змінні середовища надаються за допомогою аргументу -e . Цей приклад виконує команду ENV і завершує роботу, яка просто друкує змінні середовища. Змінні середовища можуть читатися програмним забезпеченням, що працює всередині контейнера, як спосіб надання конфігурацій часу виконання.

Кожна послуга буде упакована в окремий контейнер. У середовищі розробки три контейнери можуть бути запущені на одному хості. В окремому виробничому середовищі кожен контейнер може виконуватися на незалежному хості. До контейнерів не потрібно буде вносити жодних змін, окрім запуску їх із правильною конфігурацією як змінних середовища, які можна легко автоматизувати. За потреби кожну службу можна масштабувати

незалежно від інших, збільшуючи кількість екземплярів для цього конкретного контейнера.

Цей шаблон дає такі переваги:

- Використання ресурсів оптимізовано, накладні витрати зменшуються порівняно з віртуалізацією повного стека, оскільки потрібно віртуалізувати лише тонкий рівень, покращуючи продуктивність, яку досягає хост.
- Ресурси можуть бути виділені для контейнера, використовуючи доступні ресурси хоста між кількома контейнерами, а також те, що надається з контейнера на хост і навпаки.
- Аргументи можна надати контейнеру під час виконання, щоб налаштувати службу, що працює всередині нього. Завдяки своїй незмінності, у разі збою контейнер може перезапуститися з початковою конфігурацією.
- Ізольоване середовище можна легко перенести між розробкою та виробництвом, оскільки розмір образу лише упаковує службу та її залежності, залишаючи поза увагою всі компоненти операційної системи.
- Шаблон також вводить наступні зобов’язання:
- Паравіртуалізація — це техніка віртуалізації, яка надає доступ до частини апаратного забезпечення хоста безпосередньо віртуальній машині. У деяких сценаріях низькорівневого доступу до обладнання паравіртуалізація може забезпечити підвищення продуктивності.
- Упаковка служб як контейнерів все одно призведе до накладних витрат порівняно зі встановленням служб безпосередньо на хості.

Конфігурація може знадобитися, щоб контейнер адаптувався до кількох хостів і сценаріїв. Використовуючи шаблон конфігурації на основі середовища, можна використовувати змінні середовища для налаштування запущених служб під час виконання.

Деякі контейнери можуть потребувати збереження інформації між виконаннями в хості. Це стосується окремих баз даних, які не можуть втратити свої дані, якщо машина перезавантажиться. Маючи на увазі цю

мету, шаблон локальних томів можна використовувати, щоб відкрити папку з хоста всередині контейнера.

Висновки до розділу 2

В даному розділі наведені моделі хмарного програмного забезпечення та шаблони контейнеризації та масштабування хмарних сервісів. Також детально досліджені хмарні шаблони проектування дизайну хмарних рішень та сервісів.

РОЗДІЛ 3. РОЗРОБКА ТА АДАПТАЦІЯ МОДЕЛЕЙ ТА ШАБЛОНІВ ХМАРНИХ РІШЕНЬ ТА СЕРВІСІВ

3.1 Дослідження хмарних архітектур

Розглянемо проект, що має на меті допомогти людям похилого віку (або опікунам) збільшити свою незалежність, оснастивши їх будинок набором датчиків, які могли б допомогти їхньому повсякденному житті. Було реалізовано кілька варіантів використання. Наприклад, датчики дверей і вікон можуть попереджати про те, що вони помилково залишені відкритими, запобігаючи загрозі безпеці. Іншим успішним прикладом використання було безперервне відстеження пацієнтів з хворобою Альцгеймера, забезпечуючи механізм тривоги, який вони могли спрацьовувати, якщо вони заблукали або заплуталися, як у своєму домі, так і поза ним.

Проект вимагає універсальної програмної платформи для агрегування інформації, що надходить в екосистему, а також для керування й ідентифікації користувачів і послуг. Враховуючи медичну природу даних, що обробляються, передача повідомлень вимагає централізованої системи зв’язку для забезпечення належної автентифікації, маршрутизації та безпеки всіх даних, якими обмінюються.

Рисунок 3.1 містить огляд архітектури проекту. Для цього проекту було розроблено Ambient Assisted Living Message Queue (AALMQ), хмарну програму для організації зв’язку між двома іншими об’єктами проекту, наприклад, датчиком частоти серцевих скорочень і приладовою панеллю пацієнта.

Система надає чергу повідомлень, де сторонні служби та пристрої обмінюються даними за умови автентифікації та авторизації. Існували обмеження щодо того, яку інформацію могли отримувати передплатники, тому ми надали доступ до черг і те, якими повідомленнями можна було

обмінюватися в кожній черзі, щоб лише авторизовані видавці та споживачі могли взаємодіяти з певною чергою, запобігаючи витоку даних.

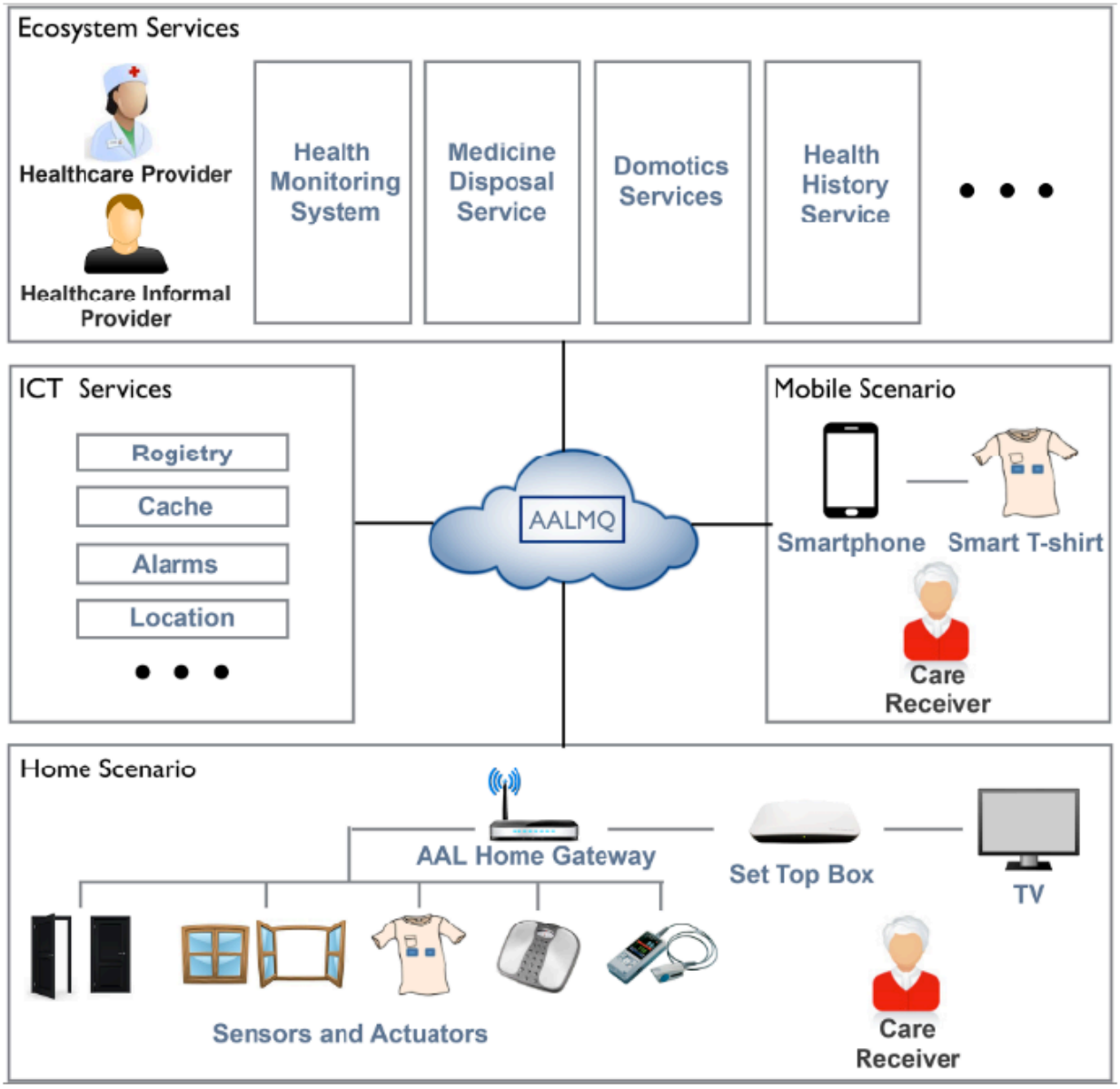


Рисунок 3.1 – Архітектура проекту

Сервіси екосистеми дозволяють опікунам і медичним працівникам взаємодіяти з отримувачем допомоги. Послуги ІКТ забезпечили функціональні можливості всього проекту. Опікун має мобільний і домашній сценарії, де за ним контролювалися різні датчики, які дозволяли сповістити доглядача в разі надзвичайної ситуації.

Рисунок 3.2 (стор. 58) демонструє сценарій, коли особа, що отримує медичну допомогу (пацієнт), має вдома ваги, пристрій для зчитування електрокардіографії (ЕКГ) і домашній шлюз AAL. Десь у хмарі вузол AALMQ забезпечує зв'язок між домашнім шлюзом AAL і веб-додатком моніторингу та мобільним додатком моніторингу, які використовуються відповідно постачальником медичних послуг і неформальним постачальником медичних послуг.

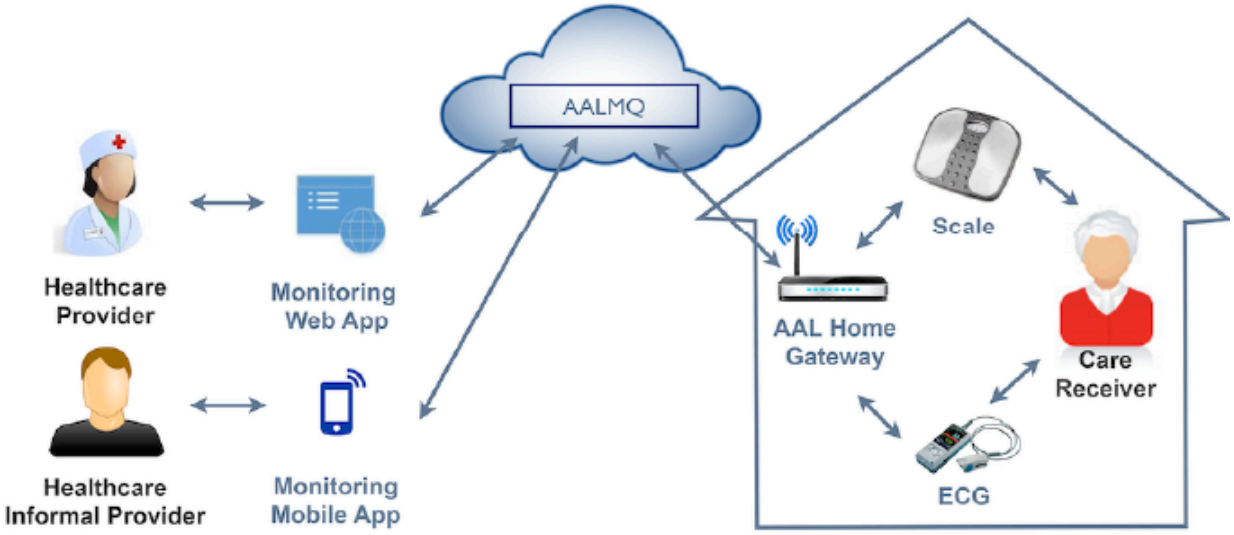


Рисунок 3.2 – Візуалізація проміжних компонентів для випадку використання зчитування електрокардіографії (ЕКГ) у домашньому середовищі медичного працівника

Постачальник представляє організацію (лікаря, медсестру), яка відповідає за моніторинг і реагування на будь-які проблеми, які можуть виникнути з потребуючим медичної допомоги. Водночас неофіційний постачальник медичних послуг – це неспеціалізована особа (родина, друг), яка хоче бути в курсі статусу потребуючого медичної допомоги.

Можна визначити різні сценарії спілкування, один з яких проілюстровано на рисунку 3.3 (стор. 59). У цьому сценарії описано механізм забезпечення того, що постачальник медичних послуг (p) отримує

сповіщення, коли серцевий ритм постачальника медичних послуг (r) виходить за межі вказаного діапазону.

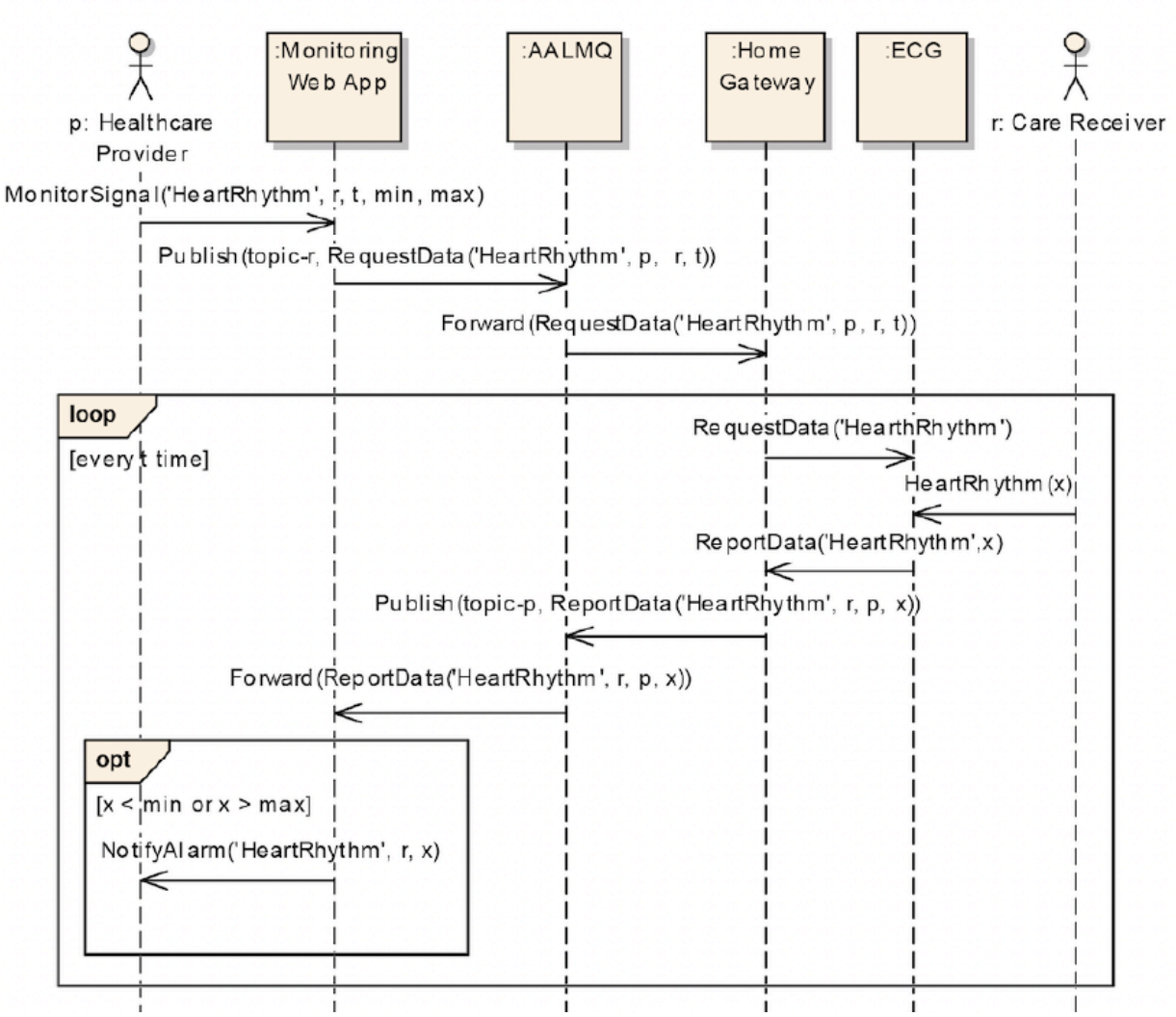


Рисунок 3.3 – Діаграма послідовності, що демонструє передачу повідомлень між взаємодіючими службами у випадку використання запису показань ЕКГ пацієнта

Це опис того, що всі учасники були попередньо налаштовані та мали дозвіл на обмін повідомленнями. Сценарій працює наступним чином:

Постачальник медичних послуг (p) починає взаємодію з веб-додатком моніторингу взаємодія абстрактно представлена повідомленням MonitorSignal на рисунку 3.3.

Веб-додаток моніторингу готує повідомлення RequestData для надсилання на шлюз AAL, який обслуговує отримувача медичної допомоги, маючи в якості параметрів тип сигналу, ідентифікацію постачальника медичних послуг, який запитує дані, ідентифікацію отримувача медичної допомоги, яку потрібно контролювати, та період збору даних. Веб-програма Monitoring Web App передає повідомлення через вузол AALMQ , надсилаючи повідомлення Publish до вузла AALMQ, яке має як параметри ключ маршрутизації та повідомлення RequestData. У цьому випадку ключ маршрутизації — це тема, яка визначає ціль повідомлення, тобто одержувача медичної допомоги. Якщо потрібно, повідомлення RequestData може бути зашифровано веб-програмою моніторингу, стаючи повністю непрозорим для вузла AALMQ .

Припускаючи, що вказаний домашній шлюз AAL раніше підписався на тему, яка ідентифікує отримувача допомоги r, вузол AALMQ пересилає повідомлення RequestData до шлюзу AAL.

Домашній шлюз AAL потім розшифровує (за потреби) та інтерпретує отримане повідомлення. На основі конфігураційної інформації він спочатку визначає пристрій, який вимірює запитуваний сигнал (ритм серця) від запитуваної особи (r). Потім, як показано оператором взаємодії циклу на рисунку, він періодично запитує (з періодом t) зчитування від пристрою, який повідомляє виміряне значення (x). Для кожного значення, повідомленого пристроєм, шлюз готує повідомлення ReportData для надсилання запитувачу, що має як параметри тип сигналу, ідентифікацію отримувача медичної допомоги, що контролюється (r), ідентифікацію медичної допомоги.

Постачальник, який запитав дані (p), і фактичне вимірювання (x). Шлюз передає повідомлення через вузол AALMQ, надсилаючи повідомлення Publish, яке має як параметри ключ маршрутизації та повідомлення ReportData. Як і в попередньому випадку, ключ маршрутизації — це тема, яка визначає ціль повідомлення, яким у цьому випадку є постачальник послуг.

Знову ж таки, якщо потрібно, повідомлення ReportData може бути зашифрованим, стаючи повністю непрозорим для вузла AALMQ.

Якщо припустити, що веб-програма моніторингу попередньо підписалася на тему, яка ідентифікує постачальника послуг p, вузол AALMQ пересилає повідомлення ReportData до веб-програми моніторингу.

Веб-додаток моніторингу потім розшифровує (за потреби) та інтерпретує отримане повідомлення. Він додає отримане значення до своєї внутрішньої бази даних для подальшої консультації та, якщо він виходить за межі зазначеного діапазону, надсилає сповіщення про тривогу постачальнику послуг із зазначенням типу сигналу, ідентифікації отримувача допомоги (r) та спостережуваного значення (x).

Представлений приклад базується на невеликому випадку, який було проведено для перевірки та вдосконалення підходу до тестування та сертифікації, представленого в цьому документі. Інфраструктура тестування, яка включає багаторазові тестові драйвери, була розроблена для полегшення впровадження та виконання модульних тестів.

Визначені раніше випадки використання дозволили нам визначити вимоги до системи. Враховуючи те, що вона регулюватиме передачу повідомлень між будь-якими іншими двома компонентами в екосистемі, це вважалося критичним у проекті. Таким чином, його проектування, розробка та експлуатація повинні були запобігти простою. Зміни в ньому, як-от оновлення або зміни в інфраструктурі, необхідні для забезпечення мінімального простою або відсутності його, мінімізуючи вплив на екосистему.

Щоб забезпечити відповідність програми вимогам, було кілька проблем. Зокрема, ми повинні були розглянути:

Доставка. Переміщення програмного забезпечення між середовищами вручну, наприклад копіювання файлів між хостами, непрактично. У той час була представлена технологія контейнерів Docker. Ми використовували Docker, щоб упакувати дві служби, які складали AALMQ незалежно:

програмне забезпечення черги повідомлень і супутнє програмне забезпечення, яке керує доступом до неї. Використовуючи Apache Mesos, програмне забезпечення для агрегації кластерів, можна розгортати контейнери з резервуванням в інфраструктурі без необхідності доступу до кожного сервера вручну, що робить розгортання тривіальним.

Агрегація інфраструктури. Для роботи з додатком доведеться використовувати кілька комп'ютерів. Одного комп'ютера було недостатньо для обробки трафіку під час пікового використання, і потрібно створити резервну систему, яка продовжувала б працювати в разі збою одного вузла. Не було б тривіально керувати кожною машиною, незважаючи на будь-яку автоматизацію.

Потрібно абстрагувати віртуальні машини, що лежать в основі, і якимось чином вказати, як у них працює програмне забезпечення. Для цього можна використати Mesos і Marathon як технологію агрегації для розгортання образів Docker. Відповідно було розгорнуто дві служби на всіх трьох серверах, які складали кластер, для продуктивності та резервування. Додаткові сервери можуть додавати на вимогу, а також розгортати служби, налаштовуючи просту конфігурацію.

Автоматизація розгортання. Початкове розгортання було запущено вручну. Було виявлено, що цей процес займав багато часу та вимагав чіткого дотримання сценарію. Незначне відхилення від цього сценарію може перевести систему в автономний режим. Тому необхідно впровадити автоматизацію, щоб сценарій міг виконуватися комп'ютером, запобігаючи помилкам людини. Було реалізовано невеликий сценарій розгортання, який гарантував наявність правильної версії вихідного коду локально, створив і опублікував із нього образ докера, а потім витягнув цей образ у оркестратор виробничого середовища. У той час інфраструктури автоматизації розгортання не були широко поширені, тому ми самі розробили цю стратегію автоматизації.

Доступність і стійкість. Дослідницький проект мав кілька випробувань, як на синтетичних даних. Оскільки AALMQ був єдиною точкою відмови для всієї екосистеми AAL4ALL, його потрібно було розробити за допомогою стратегій, які мінімізували ймовірність того, що він стане недоступним. Поки цей вузол був доступний, запити, що надходили в систему, пересилалися до одного з вузлів випадковим чином. Упорядкування повідомлень не стосувалося проекту. Якщо розгорнута служба перестала працювати, оркестратор спробував виконати автоматичне відновлення. Якщо це не спрацює, зовнішня система моніторингу сповістить нас електронною поштою, що дозволить швидко відповісти вручну.

Трафік і масштаб. Будучи раннім доказом концепції, яка проводила польові випробування, обсяг даних, якими обмінювався через AALMQ , був мінімальним. Піковий трафік рідко буде перевищувати десятки повідомлень в секунду. Одна машина могла б керувати цим обсягом даних. Надлишковість була здебільшого мотивована для забезпечення доступності та стійкості, ніж продуктивності та масштабованості. Тим не менш, прийнята архітектура може швидко масштабуватися до нових машин для обробки додаткового трафіку.

Отже, впровадження AALMQ дало чудову можливість експериментувати з новими хмарними технологіями та їх агрегаціями. В AALMQ було зроблено три внески, які розширили знання про хмару. Mesos і Marathon виконували функції менеджера та абстрагували інфраструктуру, полегшуючи роботу сервісу та масштабованість. Docker забезпечив контейнеризацію сервісу, дозволяючи виконувати ізольовано та середовищ. RabbitMQ надав службу черги повідомлень із обмеженою взаємодією з чергою.

Ці конструктивні рішення впоралися з визначеними вимогами, забезпечивши масштабовану та безпечну систему для маршрутизації повідомлень між розподіленими взаємодіючими службами.

3.2 Каталог шаблонів для хмарних сервісів та емпірична оцінка проекту

Експериментів із хмарними технологіями було недостатньо, щоб зрозуміти, чи узгоджується уявлення про хмарну екосистему з найкращими практиками, прийнятими іншими інженерами. Чи прийнято оптимальні рішення? Чи використано найкращі технології ? Як можна додатково оптимізувати середовище ? Щоб відповісти на ці питання, вже було недостатньо експериментувати з технологіями.

Для цього виконано емпіричну оцінку для збору цієї інформації і використано каталог практик Cloud і DevOps із 13 шаблонами.

Оцінка охоплювала такі категорії:

- Продукт. У розділі продукту треба зрозуміти, що зробила компанія по-друге, якщо були якісь особливі вимоги, які б вплинули на вибір компанії.
- Керування командою. Розмір команди, взаємодія, методи управління проектами.
- Конвеєр доставки програмного забезпечення. У цьому пункті визначено, чи робили групи безперервну інтеграцію, як вони справлялися зі створенням середовищ для кожного стану конвеєра та які команди що робили в кожному стані.
- Управління інфраструктурою. Тут відображається як компанії обробляли свою інфраструктуру. Вони використовували хмару? Які процеси вони автоматизували ?
- Моніторинг і обробка помилок. У цьому визначається чи стежили компанії за своєю інфраструктурою, як вони це робили та як реагували на виявлення помилок ?

Емпіричне оцінювання дозволяє визначити тенденції, які зрештою призвели до створення каталогу шаблонів із 13 шаблонів розробки та операцій:

- Оповіщення. Визначення стратегії сповіщення команди, коли система зазнає збоїв. Вся команда може бути сповіщена або може бути підгрупа, відповідальна за оцінку сповіщень протягом певного періоду. Члени цієї підгрупи можуть періодично змінюватись.
- Аудит. Відстеження стану програми та журналів, надсилаючи сповіщення про збої для швидшого реагування команди, коли це необхідно.
- Хмара. Розробка продукту для використання хмарних обчислень забезпечує швидшу розробку завдяки використанню хмарних служб.
- Огляд коду. Експертна перевірка вихідного коду перед його прийняттям, щоб мінімізувати помилки та поділитися знаннями.
- Спілкування. Визначення канали зв’язку для команди, щоб вони могли легко підтримувати синхронізацію.
- Безперервна інтеграція. Використання системи безперервної інтеграції, щоб автоматично тестувати та створювати систему, сповіщаючи групу розробників про збої в цьому процесі.
- Розгортання. Автоматизація налаштування середовища за допомогою сценаріїв розгортання та розгортання продукту за допомогою відтворюваних середовищ.
- Планування роботи. Складання графіків виконання робіт в інфраструктурі.
- Відтворювані середовища. Інтеграція програмного забезпечення переносним способом між системами.
- Масштабування. Використання вертикального або горизонтального масштабування для більшої доступності продукту.
- Командна робота. Налаштування розміру команди розробників так, щоб вона була менше десяти членів, і сприяння гнучким процесам і комунікації.
- Контроль версій. Використання системи контролю версій, щоб полегшити співпрацю між розробниками.

Таблиця 3.1 визначає, скільки компаній можуть використувати кожен шаблон.

Таблиця 3.1

Прийняття каталогу шаблонів

Pattern name	Adoption count
Communication	25
Version Control	24
Cloud	21
Auditability	17
Continuous Integration	16
Reproducible Environments	15
Error Handling	15
Scaling	12
Deployment	11
Code Review	11
Team Orchestration	9
Alerting	9
Job scheduling	3

Потрібно визначити, яким був вплив застосування цих шаблонів і визначення покращення показників, вимірюючи ключові показники продуктивності до та після застосування шаблонів.

Мотивацією дослідження полягає в покращенні хмарної стратегії, а саме:

- скорочення часу, витраченого на розгортання програмного забезпечення, застосовуючи автоматизацію;
- скорочення часу, необхідного для налаштування нових середовищ розробки та розгортання, шляхом програмного впровадження відтворюваних середовищ;
- мінімізація варіації стану між машинами для розробки та виробництва;
- модернізація стеку технологій.

Шаблон розгортання було прийнято, щоб гарантувати створення середовища під час кожного розгортання, завдяки тому, що процес створює образ контейнера, придатний для виробництва, який потім використовуватиме процес розгортання під час розгортання виробничого середовища.

Таблиця 3.2 визначає показники отримані на початку та в кінці експерименту. Автоматизація стратегії розгортання дещо сповільнила тривалість розгортання як побічний ефект постійного виконання всіх тестів, що, з іншого боку, підвищило впевненість команди під час кожного розгортання. Ця автоматизація підвищила обізнаність розробників про проблеми з кодовою базою, зменшивши частоту, з якою спостерігалися помилки збірки.

Таблиця 3.2.

Показники продуктивності до та після експерименту

Metric	Initial value	Final value
Deployment strategy (staging)	Manual	Automatic
Deployment duration (staging)	3 to 10 minutes	15 minutes
Deployment frequency (staging)	When needed	On every push
Environment set up strategy	Manual	Automatic
Environment set up time	5 minutes to hours, depending on developer	Usually less than 5 minutes
Ease to make environment changes	Low	High
Build errors frequency	Low	Medium

Налаштування нових середовищ, як для нових розробників, інсценування, так і для виробництва, мало найзначніший вплив, у будь-якому випадку це було зроблено за 5 хвилин, витягнувши правильний образ Docker. Істотне вдосконалення для завдання, яке часто вимагало кількох годин для менш досвідчених членів команди. Частота, з якою спостерігалися помилки збірки, була результатом удосконалення процесу збірки, що збільшило

кількість помилок, виявлених під час збірки програмного забезпечення, і справді була позитивною зміною.

3.3 Технологія автоматизованого управління інфраструктурою хмарного сервісу з використанням шаблонів

Розглянемо мову шаблонів для розробки програмного забезпечення для хмарних сервісів. Пропонується десять шаблонів і посилаємося на два інших, пов’язуючи їх мовою шаблонів.

Під час запису цієї мови шаблонів було враховано деякі міркування, щоб забезпечити індивідуальну якість кожного шаблону. Було розглянуто наступні атрибути:

- Повнота. Чи повний опис шаблону ? Повний шаблон забезпечує рівень деталізації, який дає змогу читачеві ідентифікувати проблему та реалізувати її.
- Стислість. Чи шаблон містить більше інформації, ніж те, що необхідно ? Короткий шаблон переходить прямо до суті, його легко читати та відтворювати.
- Термін дії. Чи заявлене рішення дійсне та чи описано достатньо відомих застосувань? Дійсний шаблон документує прийняте гарне рішення та обґрунтовує його конкретними прикладами.

Мова шаблонів складається з дванадцяти шаблонів, десять з яких новітніх, упорядкованих у чотири категорії: автоматизоване керування інфраструктурою, агрегація та моніторинг, відкриття та комунікація (рис. 3.4) зображує закономірності в мові та розкриває їхні зв’язки.

Автоматизоване управління інфраструктурою як категорія включає два шаблони: автоматизована масштабованість та інфраструктура як код. Операції можуть мати вирішальне значення для успіху продукту. Управління операціями вручну є повільним, схильним до помилок і дорогим, що ускладнює відстеження змін і розвиток інфраструктури.

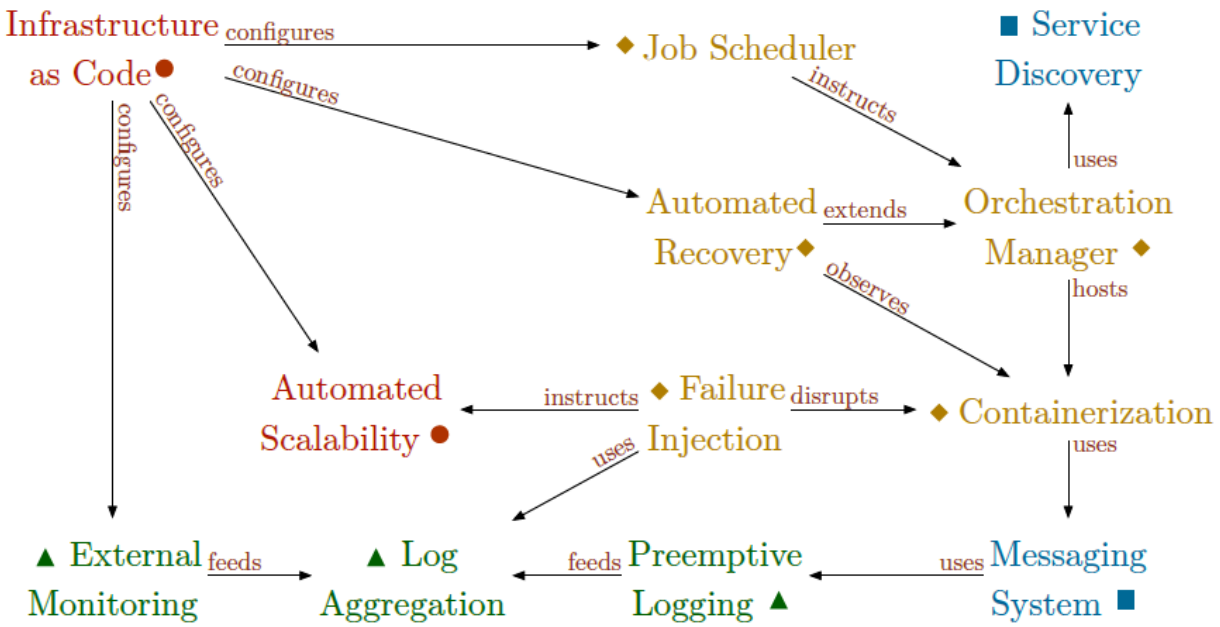


Рисунок 3.4 – Мова шаблонів для розробки програмного забезпечення для хмарних сервісів, що зображує зв’язки між шаблонами і категоріями

Щоб команди були ефективними, вони повинні автоматизувати свою роботу. Часто обговорюється автоматизоване тестування якості за допомогою впровадження автоматизованих тестів і безперервної інтеграції. Операції мають бути так само автоматизовані, реалізовані як частина процесу розробки. Інфраструктура як код дозволяє цю практику.

Використання мікросервісної архітектури дає змогу розділити відповідальність на кілька невеликих служб, які можна розробляти, масштабувати або працювати незалежно. Кілька мікросервісів можна використовувати для створення складних хмарних програм.

Хмарні програми можуть швидко перейти з майже простою в обслуговування мільйонів запитів на секунду. При розробці програмного забезпечення для хмари важливо не відставати від високих піків трафіку, щоб забезпечити надійну роботу користувача. Автоматизована масштабованість є важливою для досягнення безперервної продуктивності служби, шляхом моніторингу ресурсів, щоб вирішити, коли масштабувати систему автоматично.

Після виділення хмарної інфраструктури розробникам програмного забезпечення потрібно виділити та використовувати своє програмне забезпечення поверх неї. Ця категорія представляє п’ять шаблонів, які допомагають розробникам керувати своїм програмним забезпеченням.

Традиційно розгортання програмного забезпечення на хості поєднувало його з операційною системою, вимагаючи встановлення залежностей і визначення конфігурацій, які могли викликати побічні ефекти з іншими службами на тому ж хості.

Контейнеризація передбачає використання контейнерів для упаковки та розгортання служб ізольовано, уникаючи їх впливу один на одного або на хост.

Масштабне розгортання й оновлення програмного забезпечення вручну загрожує помилками, є повільним і дорогим. Менеджер агрегації може допомогти автоматизувати цей процес, надаючи програмний спосіб оркестровки служб, одночасно абстрагуючи базову інфраструктуру.

Оскільки час безвідмовної роботи програмного забезпечення має вирішальне значення, розробники, впроваджують стратегії автоматичного відновлення, деякі з яких описано в шаблонах вище. Оскільки програмне забезпечення є програмним забезпеченням, самі стратегії відновлення схильні до збоїв, і їх потрібно часто використовувати, щоб переконатися в їх правильності.

Шаблони моніторингу. Програмне забезпечення, що працює в хмарі, може бути піддано величезній кількості трафіку, який, зрештою, і за наявності достатнього часу призведе до несподіваних сценаріїв. Незважаючи на те, що неможливо запобігти виникненню проблем, розробники повинні впроваджувати необхідні стратегії, щоб визначити, коли та чому виникають проблеми з їхнім програмним забезпеченням, щоб вони могли вирішити їх швидко та відразу з першого разу, запобігаючи повторному впливу на програмне забезпечення в майбутньому. Ця категорія представляє три

шаблони для полегшення спостереження за поведінкою програми під час виконання.

Превентивне ведення журналу описує серію практик, які гарантують, що інформація про час виконання буде захоплена та доступна розробникам для вирішення проблем при їх першому виникненні. Це робиться шляхом попередньої оптимізації рівня журналювання, яке створює програма.

Агрегація журналів потім описує, як ці журнали мають бути централізовані для полегшеного доступу, що переважно стосується розподілених систем.

Збережена інформація буде найбільш актуальною для виявлення проблем. Автоматизовані стратегії моніторингу є періодичними та важливими та можуть бути внутрішніми або зовнішніми щодо інфраструктури, на якій працює хмарне програмне забезпечення.

Зовнішній моніторинг описує, як внутрішній моніторинг підлягає упередженому спостереженню, нездатному виявити, наприклад, проблеми з підключенням до Інтернету, оскільки він тестує програмне забезпечення з тієї ж локальної мережі та пропонує запровадження зовнішнього об’єкта для контролю за загальнодоступними інтерфейсами програми.

Послідовність для веб-додатку. Розглянемо сценарій, коли спеціалісту з хмарних технологій потрібно створити та розгорнути резервну веб-програму, що складається з клієнтського HTTP-сервера та бази даних. Практик повинен розробити свій HTTP-сервер і базу даних як два взаємодіючі мікросервіси. Використовуючи контейнеризацію та одну службу на контейнер, він створив два зображення контейнера, по одному для кожної служби. Ці контейнери були б легко переносимими між кількома середовищами, такими як локальне, проміжне або виробниче середовища, налаштовані за допомогою доступних змінних середовища. З інфраструктурою як кодом практик описує інфраструктуру, необхідну для налаштування системи. Після виконання цього програмного опису необхідна інфраструктура стане доступною.

Automated scalability може налаштувати інфраструктуру на горизонтальне масштабування, якщо це необхідно. Щоб розгорнути його служби ізольованим і масштабованим способом, інфраструктура повинна бути абстрагованою.

Менеджер агрегації буде відповідати за оптимальний розподіл машин-контейнерів в інфраструктурі, беручи до уваги загальні та доступні ресурси на кожній машині.

Планувальник завдань буде відповідати за виконання щоденного процесу резервного копіювання бази даних на зовнішній сайт.

Веб-сервер використовуватиме локальний мережевий порт 12345 для підключення до бази даних, увімкнений виявленням служби. Шаблон вводить локальний зворотний проксі на всіх машинах, який надає статичний порт служби для кожної служби в кластері. Цей сценарій не вимагає системи обміну повідомленнями .

Щоб переконатися, що служба працює належним чином, практикуючий спеціаліст повинен застосувати методи моніторингу.

Служба зовнішнього моніторингу може відстежувати всі кінцеві точки, що виходять в Інтернет, гарантуючи, що вони обидва онлайн і реагують належним чином. попереджувальне ведення журналу може додатково підвищити обізнаність про стан системи, налаштувавши служби з відповідним рівнем журналізації. Обґрунтування полягає в тому, що відповідна інформація про час виконання повинна бути захоплена для можливих проблем налагодження, коли виникають проблеми, що ускладнює їх налагодження в іншому випадку.

Агрегація журналів може централізувати ці журнали, індексувати та зробити їх доступними для запитів для ефективного використання.

Нарешті, фахівець повинен переконатися, що його стратегії стійкості активні та ефективні. Ін’єкція відмов може задіяти існуючі механізми стійкості шляхом випадкового введення помилок в інфраструктуру, таких як

випадкове завершення роботи машин, і перевірки того, що система відновлюється автоматично.

Отже, мова організована за чотирма категоріями шаблонів: автоматизоване керування інфраструктурою, агрегація та моніторинг, виявлення та зв’язок. Гіпотетичний сценарій впровадження шаблонів описується за допомогою послідовності їх прийняття.

3.4 Дослідження моделей шаблонів для хмарних сервісів

В архітектурі мікросервісів кілька служб повинні співпрацювати, щоб забезпечити програму в цілому.

Співпраця вимагає, щоб служби спочатку виявили та створили канали зв’язку між собою. Це представляє систему обміну повідомленнями та виявлення служб.

Розглянемо наступні шаблони:

- Система обміну повідомленнями. У міру збільшення обсягу та складності взаємодіючих служб канали зв’язку «точка-точка» стають некерованими, що перешкоджає відмовостійкості та масштабованості. Для цього потрібно використовувати систему обміну повідомленнями , яка ще називається чергою повідомлень, щоб абстрагувати розміщення служб і оркеструвати повідомлення з оптимальною стратегією маршрутизації між ними.
- Відкриття служби. У динамічно розподіленій інфраструктурі служби потребують стратегії виявлення для встановлення каналу зв’язку. Абстрактні деталі мережі послуг, покладаючись на зовнішній механізм, який полегшує зв’язок і балансує трафік між двома службами.

Прийняття мікросервісів як архітектурного стилю запровадило потребу в співпраці сервісів у децентралізованому та, можливо, ненадійному середовищі. Не гарантується, що кожен компонент постійно буде онлайн, а

також те, що кожна служба має стабільну IP-адресу (протокол Інтернету) або фіксовану кількість запущених екземплярів.

Ці тонкощі хмарних обчислень вводять кілька вимог. А саме, служби повинні спілкуватися один з одним у постійно мінливому середовищі, процес зв'язку має бути стійким до збоїв, забезпечуючи стійкість системи в цілому, коли вона стикається з нестандартною поведінкою будь-якої сторони зв'язку, а передача повідомлень повинна бути асинхронним, роз'єднаним, розвиватися, використовуючи канал зв'язку , що не залежить від вмісту .

Розглянемо рішення домашньої автоматизації, яке керує системами кондиціонування повітря (АС). Три служби складають рішення: зчитувач датчиків, приймач даних і менеджер змінного струму. Sensor Reader розгортається в будинку користувача. Він відповідає за отримання та передачу даних про температуру. Data Receiver — це веб-сервер, який отримує показники температури та зберігає їх у базі даних. Data Receiver також може забезпечувати агрегації даних, що зберігаються в базі даних. Менеджер кондиціонування відповідає за керування блоками кондиціонування, оцінюючи середню температуру за останні 10 хвилин, налаштовуючи кондиціонер для генерування холодного або теплого повітря. Три служби повинні співпрацювати, щоб забезпечити повне рішення для автоматизованого керування кондиціонером. Очікувана взаємодія між ними зображена на рис 3.5.

Служби в хмарній програмі повинні спілкуватися один з одним, щоб співпрацювати. Типова стратегія зв'язку використовує підхід клієнт-сервер, обмежуючи зв'язок двома проміжними екземплярами служби та вимагаючи, щоб клієнт знав, як підключитися на сервер, а саме його ім'я хоста та порт сервера. Хмарні програми розгортаються на динамічному обладнанні, що означає, що адреси внутрішнього сервера недоступні під час розробки, що ускладнює використання прямого зв'язку між службами. Крім того, коли існує кілька екземплярів служби, трафік потрібно збалансувати між усіма екземплярами.

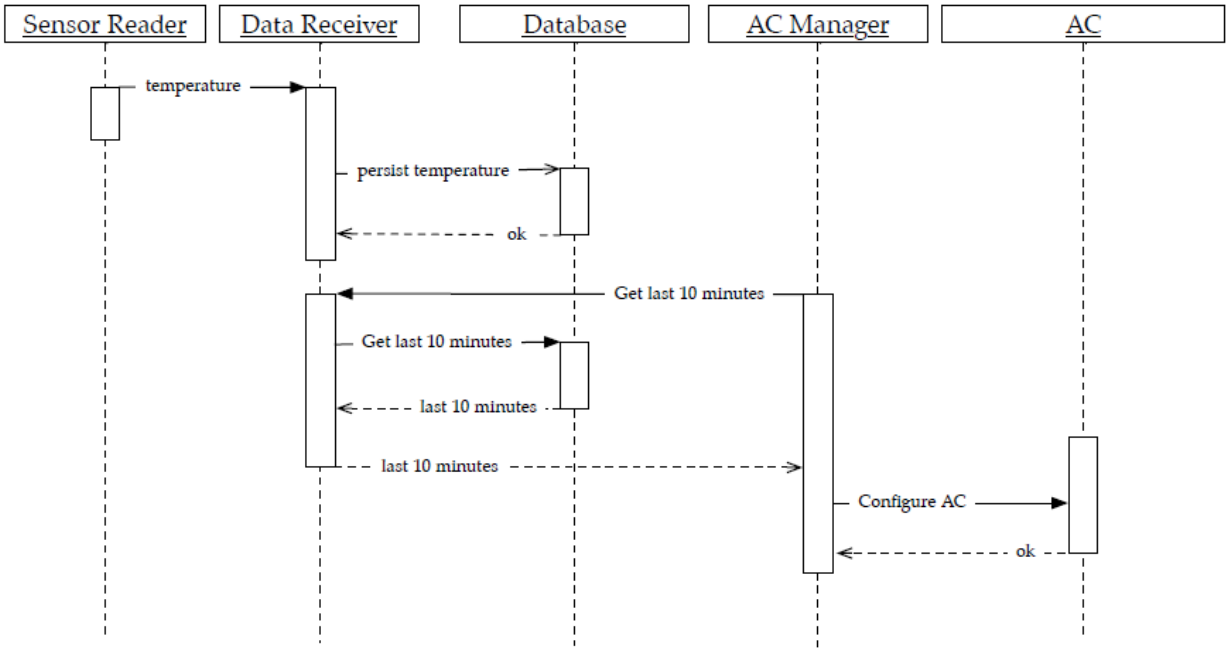


Рисунок 3.5 – Система на основі архітектури мікросервісу для збору та збереження показників температури середовища

Враховуючи вищезазначене, визначено необхідність абстрагування зв’язку між службами. Такий канал має забезпечувати передачу будь-яких повідомлень і правильно ідентифікувати відправника та одержувача таких повідомлень. Канал зв'язку має бути масштабованим, забезпечуючи дотримання вимог щодо затримки навіть при обробці великих обсягів повідомлень.

Обмеження

Наступні обмеження, зображені на рис 3.5, необхідно збалансувати при розгляді прийняття цього шаблону:

На цю модель впливають наступні обмеження:

2. Розчеплення. Відправнику не потрібно знати мережеву адресу служби-одержувача, щоб спілкуватися з нею.

3. Масштабованість. Сам канал зв'язку повинен бути масштабованим.

4. Стійкість. Комунікація повинна бути стійкою, незважаючи на збої в каналі зв'язку.

5. Наполегливість. Повідомлення між службами мають зберігатися, доки не буде підтвердження їх обробки.

6. Динамічна і гнучка топологія. Топологія системи буде розвиватися з часом, нові служби приєднуватимуться до існуючих, а інші залишатимуться в режимі реального часу.

7. Безпека корисного навантаження. Канал зв'язку повинен підтримувати зашифровані повідомлення.

8. Безпека каналу. Сам канал зв'язку повинен бути зашифрований.

9. Затримка. Введення непрямого зв’язку збільшує час затримки, необхідний для передачі повідомлення між двома службами.

10. Порядок і пріоритет. Команда повинна оцінити, чи порядок і пріоритет повідомлень актуальні для впровадження.

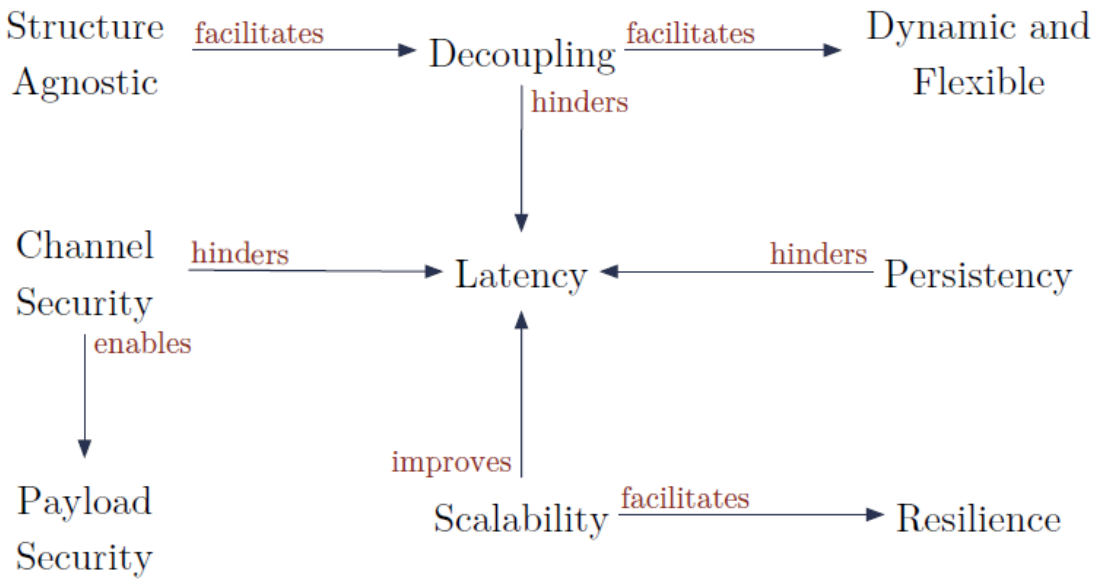


Рисунок 3.5 – Взаємозв'язок між обмеженнями системи обміну повідомленнями

Для вирішення даної проблеми потрібно використати систему обміну повідомленнями щоб абстрагувати розміщення служб і оркеструвати повідомлення з оптимальною стратегією маршрутизації між ними.

Система обміну повідомленнями відповідає за маршрутизацію повідомлень між службами, які можуть бути як виробниками, так і споживачами повідомлень. Повідомлення можуть відрізнятися за розміром і вмістом, оскільки канал не залежить від їх внутрішньої структури, якщо вони дотримуються прийнятого протоколу.

Система обміну повідомленнями працює шляхом створення однієї або кількох черг, які працюють як структура даних « першим прийшов, першим вийшов» (FIFO). Деякі реалізації надають можливість пріоритезації повідомлень у черзі. Також можна застосувати політику якості обслуговування (QoS), яка змушує споживачів підтвердити, що вони успішно обробили повідомлення, перш ніж воно буде вилучено з черги. QoS гарантує, що несправна служба не видалить повідомлення з черги без його обробки. Якщо служба не підтверджує, що повідомлення було оброблено протягом прийнятного періоду, повідомлення стає доступним для обробки іншим споживачем.

Більшість реалізацій підтримують кілька стратегій доставки повідомлень. RabbitMQ, яка є однією з найпоширеніших реалізацій, підтримує прості черги, обмін із кількома чергами, маршрутизацію, споживання на основі тем і RPC (Remote Procedure Call).

При реалізації Remote Procedure Call (RPC) служби можуть надсилати запити до черги повідомлень і блокувати очікування відповіді. Споживач підбирає запит, обробляє його та надсилає назад у чергу, призначену відправнику запиту. Тоді перший сервер отримає його запит і відновить обчислення.

Перенесення відповідальності за обробку всіх комунікацій до служби черги повідомлень робить її єдиною точкою відмови. З цієї причини служби обміну повідомленнями зазвичай розгортаються з резервуванням, що гарантує, що зв’язок між службами продовжуватиме працювати, якщо деякі екземпляри вийдуть з ладу.

Концепція систем передачі повідомлень була доступна протягом кількох років, оскільки проміжне програмне забезпечення забезпечує комунікаційні стратегії, які добре спостерігаються, а саме зв’язок «один до багатьох», забезпечуючи динамічні зв’язки між службами. Початкове посилання на програми обміну повідомленнями як засіб зв’язку між серверами було вперше представлено в патентній системі сервера черги повідомлень. Нещодавно було введено кілька стандартів, а саме Advanced Message Queuing Protocol (AMQP) і Message Queue Telemetry Transport (MQTT).

Більшість реалізацій дозволить каналу зв’язку використовувати алгоритм шифрування для захисту каналу зв’язку. Будучи незалежним від вмісту повідомлення, саме корисне навантаження також може бути зашифровано за потреби, запобігаючи витоку даних, навіть якщо систему обміну повідомленнями скомпрометовано.

Порядок може бути дотриманий або ні, залежно від прийнятої реалізації. RabbitMQ, наприклад, може забезпечити порядок обробки повідомлень і навіть підтримує пріоритетні та сегментовані черги, зі стратегіями, відмінними від FIFO для доставки повідомлень.

Розглядаючи попередній приклад три служби можуть спілкуватися за допомогою розподілу на основі черги повідомлень у системі повідомлень, як показано на рис. 3.6.

На даному рисунку 3.6 немає двох служб, які спілкуються безпосередньо. Стрілки представляють повідомлення, якими обмінюються системи.

Черги повідомлень можна ідентифікувати за назвою та вимагати від споживачів підписки на черги, з яких вони хочуть отримувати повідомлення. Спочатку служба одержувача даних підписуватиметься на показники черг і запити. AC Manager підписався б на чергу, названу на його честь, менеджер. Усередині будинку Sensor Reader збиратиме показники температури та надсилатиме їх до черги повідомлень за допомогою черги метрик.

Асинхронно Data Receiver споживатиме ці повідомлення та зберігатиме їх у базі даних.

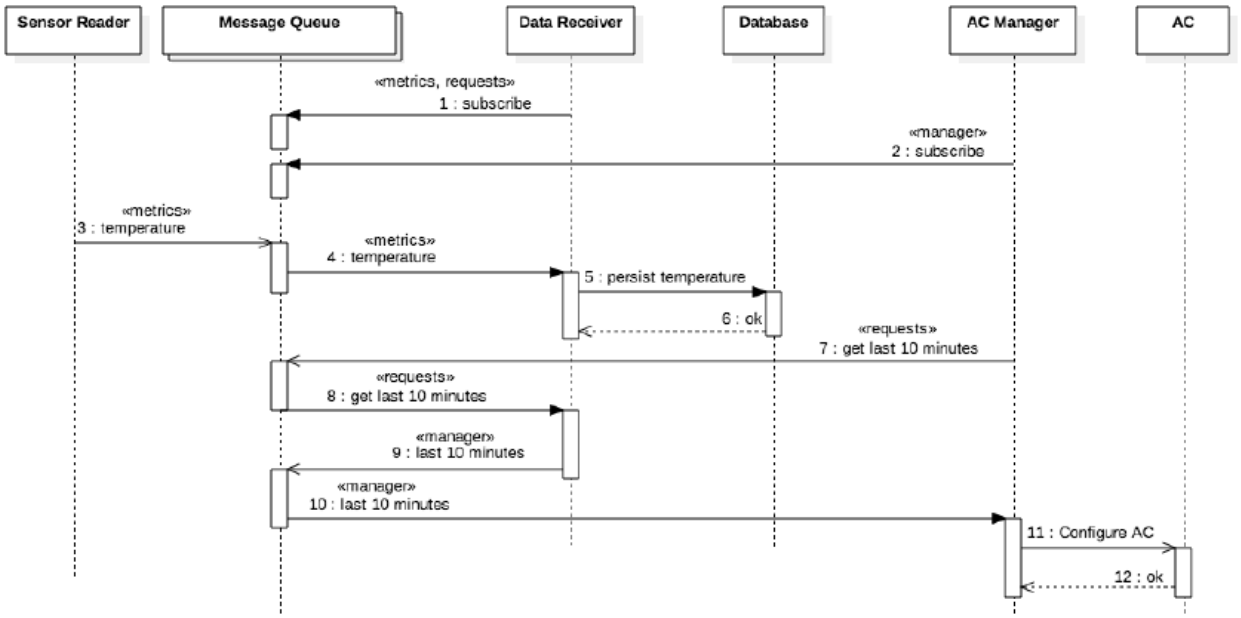


Рисунок 3.6 – Зв’язок між трьома службами, що виконують маршрутизацію через систему обміну повідомленнями

Періодично AC Manager запитує кожні 10 хвилин показники температури для черги повідомлень у черзі запитів. Одержувач даних споживе це повідомлення, зібрати цю інформацію з бази даних і надіслати її назад до черги повідомлень за допомогою черги менеджера. Нарешті, диспетчер змінного струму споживатиме ці повідомлення та налаштовуватиме відповідну поведінку системи змінного струму.

У контексті розробки програмного забезпечення для хмари черги повідомлень можуть абстрагувати розташування служб, усуваючи потребу в механізмах виявлення між ними. Кожна служба може спілкуватися безпосередньо з однією або кількома чергами, вимагаючи лише адреси служби черги повідомлень.

Використання черг повідомлень також полегшує масштабування служби. Сервіси, які отримують трафік із зовнішніх каналів, повинні бути

масштабовані, щоб обробляти трафік. Потім вони вставлятимуть повідомлення в черги, які споживаються іншими службами. У такій архітектурі розмір черги можна використовувати, щоб зрозуміти, чи потрібно масштабувати послугу і як, маючи на меті завжди зберігати якомога меншу чергу повідомлень. Системи обміну повідомленнями можна використовувати для реалізації агрегації журналів, якщо служби передають свої журнали як повідомлення, які потім агрегуються службою централізації журналів.

Відомі використання система обміну повідомленнями має широкий спектр застосувань. У Conseil European pour la Recherche Nucleaire (CERN) його використовували для надання інформації для багатьох інструментів моніторингу в кількох проектах, а саме у Великому адронному колайдері.

3.5 Методологія балансування трафіку між службами хмарних сервісів

У динамічно розподіленій інфраструктурі служби потребують стратегії виявлення для встановлення каналу зв’язку. Абстрактні деталі мережі послуг, спираючись на зовнішній механізм, який полегшує зв’язок і балансує трафік між двома службами.

Хмарні програми зазвичай складаються з безлічі служб, які можуть бути розподілені на кількох фізичних серверах у різних мережах. Для того, щоб служби могли співпрацювати, вони повинні знати, як спілкуватися одна з одною, що передбачає необхідність конфігурації або виявлення імені хоста або IP-адреси та порту, через який можна отримати доступ до необхідної служби. Крім того, коли служба має кілька екземплярів, що вимагається в налаштуваннях високої доступності, може виникнути необхідність рівномірно розподілити трафік між існуючими екземплярами.

Розглянемо приклад.

Сервер додатків отримує запити HTTP та запитує сервер бази даних щодо інформації, необхідної для обробки відповіді HTTP. З метою

масштабованості база даних поширюється з кількома копіями для читання, кількість яких різниться залежно від середнього навантаження на систему. Служба не має інформації про те, як можна отримати доступ до серверів баз даних через динамічний розподіл примірників бази даних. Рисунок 3.7 представляє можливий розподіл послуг між існуючими серверами такої системи.

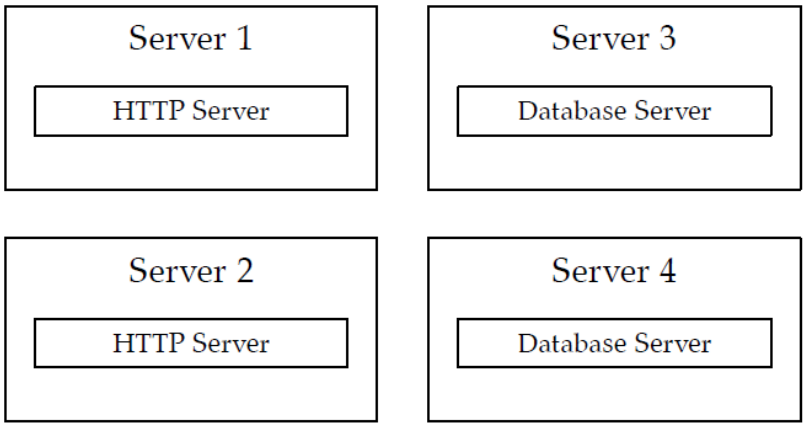


Рисунок 3.7 – Чотири об’єкти інфраструктури, кожен з яких містить свій сервіс

Опишемо проблему. У динамічно розподіленій інфраструктурі служби потребують стратегії виявлення для встановлення каналу зв’язку.

Потрібне відокремлення служби, оскільки програмне забезпечення автоматично розгортається та масштабується в хмарі, що дозволяє масштабувати окремі компоненти програмного забезпечення при динамічному використанні забезпечене обладнання. Розгортання в таких умовах залишає клієнтські служби невідомими про те, де розміщені інші служби, що вимагає стратегії виявлення для забезпечення синхронного зв’язку між ними.

Обмеження. Наступні обмежуючі показники, зображені на рис 3.8, необхідно збалансувати при розгляді прийняття цього шаблону:

1. Відкриття в реальному часі. Стан потрібно оновлювати, коли кількість екземплярів у службі змінюється.

2. Відокремлення місця розташування. Службам не потрібно знати, де розміщені інші, щоб спілкуватися з ними.

3. Діагностика протоколу. Робота на мережевому рівні, підтримка будь-якого прийнятого сервісами протоколу.

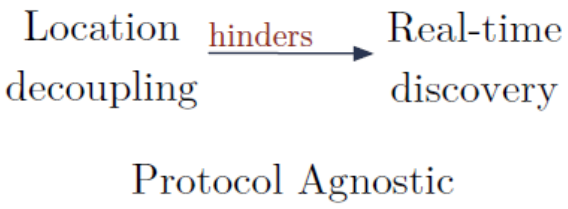


Рисунок 3.8 – Взаємозв'язок між силами виявлення служби (сервісу)

Рішення пролеми. Абстрактні деталі мережі послуг, покладаючись на зовнішній механізм, який полегшує зв’язок і балансує трафік між двома службами.

Потрібно використати новий компонент, щоб надати клієнтській службі вказівки щодо того, як досягти служби призначення. Реалізації можуть відрізнятися від використання DNS-сервера до зворотного проксі-сервера на кожному сервері.

Перший підхід полягає у використанні служби DNS, яка знає про розгортання служби, створенні одного запису DNS для кожної служби та підтримці його в актуальному стані, щоб він завжди дозволяв список серверів, на яких розгорнуто службу. Цей підхід вимагає змусити розгорнуті клієнтські служби використовувати цей DNS-сервер.

Підхід зворотного проксі базується на розгортанні проксі на кожному сервері. Проксі-сервер відкриває порт служби для кожної служби та знає про стан розгортання, щоб перенаправляти кожен локальний порт туди, де служба фактично розгорнута в інфраструктурі.

Проксі-сервери працюють на мережевому рівні, що робить їх незалежними від протоколів, без проблем обробляючи TPC, UDP або HTTP.

Обидві стратегії вимагають, щоб проксі або DNS-сервер постійно знав про стан розгортання. Для цього існує кілька стратегій. Одна з них полягає в тому, щоб мати реєстр послуг, де кожна служба повідомляє про себе, а також спеціальне програмне забезпечення, яке періодично зчитує цю інформацію та оновлює проксі-сервери. Інша альтернатива — запитати цю інформацію в менеджера оркестровки.

Як проксі-сервери, так і DNS-сервери можна налаштувати на маршрутизацію трафіку, коли доступно кілька екземплярів служби, яка діє як балансувальник навантаження. Алгоритм балансування може працювати, наприклад, розподіляючи запити за допомогою циклічної техніки або розумнішим способом, відповідно до доступності ресурсів цілі.

Отже, приклад вирішено

Ця техніка вимагає зовнішнього механізму оркестровки для збереження метаінформації про служби, що працюють в інфраструктурі, щодо хостів і портів. Кожна хост-машина має проксі-сервер, який періодично запитує диспетчер оркестровки та пересилає відомий локальний порт на хост(и) і порт, де доступна служба в інфраструктурі. Програми очікують, що конкретний порт буде доступним локально, що абстрагуватиме точний порт і хост, на якому запущено службу. Розглянемо приклад, описаний раніше: веб-додаток розгортається з двома HTTP-серверами, які отримують зовнішні запити, які повинні спілкуватися з одним із двох інших серверів баз даних, щоб створити відповідь. Щоб HTTP-сервери спілкувалися з базою даних, вони підключаються до відомого локального порту замість встановлення прямого з’єднання, залишаючи проксі-серверу пересилати запит на доступний сервер бази даних. Масштабованість досягається незалежною зміною кількості серверів баз даних або HTTP, покладаючись на проксі-сервери на стороні HTTP для належного визначення доступних

серверів баз даних і розподілу навантаження між ними, діючи як внутрішній балансир навантаження. Цей приклад представлено на малюнку 3.9.

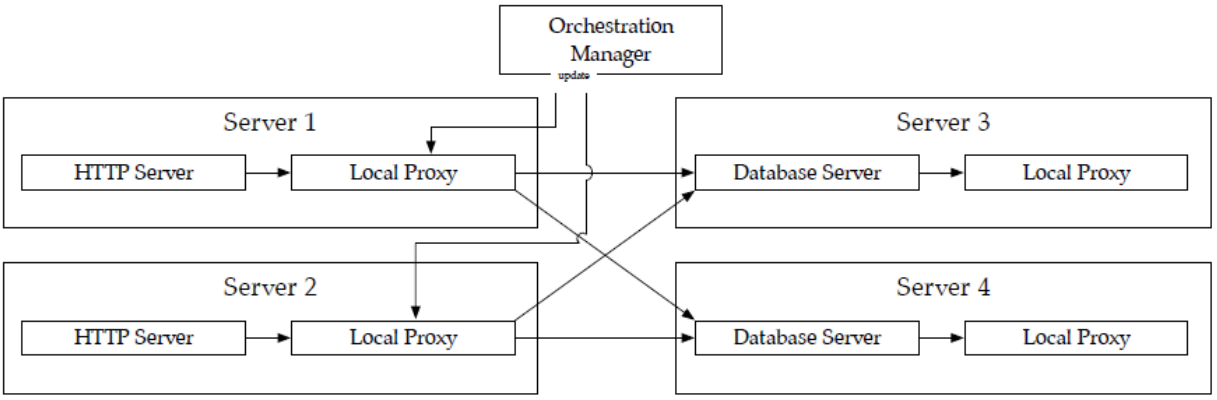


Рисунок 3.9 – Приклад конфігурації проксі

Цей шаблон дає наступні переваги:

1. Відкриття в реальному часі. Зміни в інфраструктурі негайно визначаються менеджером оркестровки, який переконфігурує проксі-сервери.

2. Відокремлення місця розташування. Розробка сервісу може ігнорувати фактичне фізичне розташування інших сервісів, з якими він інтегрується, покладаючись на зворотний проксі для перенаправлення трафіку до виконання сервісу.

3. Покращений протокол. Проксі-сервери працюють на транспортному рівні OSI або нижчому, отже, не залежать від протоколу.

Шаблон також вводить наступні зобов’язання: моніторинг. Зіставлення між службою та її запущеними екземплярами має підтримуватися постійно, щоб зворотні проксі-сервери були правильно налаштовані та перенаправляли трафік лише до активних служб.

Цей шаблон може бути застосований, коли контейнеризація використовується для ізоляції додатків, сприяння зв’язку між контейнерами, розміщеними на різних серверах, без необхідності індивідуальної інтеграції додатків із механізмами виявлення. Інформацію про службові порти в кожному контейнері можна ввести за допомогою змінних середовища.

Цей шаблон залежить від зовнішнього механізму, який відстежує кожну службу в інфраструктурі. Менеджер оркестровки зберігає цю інформацію, і його можна отримати.

Висновки до розділу 3

Отже, в цьому розділі виконаноа розробка та адаптація моделей та шаблонів хмарних рішень та сервісів. Запропоновано шаблони для підтримки співпраці служб, відповідно система обміну повідомленнями вводить передачу повідомлень як стратегію асинхронного обміну повідомленнями між службами, тоді як виявлення служби полегшує виявлення служби в кластері, підтримуючи синхронну взаємодію. Виявлення служб і зв’язок є важливими для забезпечення співпраці служб і вертикального масштабування служб хмарних сервісів.

ВИСНОВКИ

В уваліфікаційній роботі досліджено процеси розробки та адаптації моделей та шаблонів хмарних рішень та сервісів, запропоновано моделі шаблонів хмарних рішень. При розробці хмарних сервісів виникають проблеми, а саме: здатність масштабуватися, використання динамічної інфраструктури, моніторингу, безпечного обміну повідомленнями, проблеми доступності, надійності, відмовостійкості і безпека. Незважаючи на це, розробка хмарних технологій швидко розвивається, хоча вирішення цих проблем залишається складним, оскільки існує брак якісних ресурсів і досвіду у розробці хмарних технологій. В даній роботі основна увага була зосереджена на проблемах у хмарних обчисленнях. Очевидно, що під час розробки програмного забезпечення для хмарних сервісів виникають десятки чи сотні інших актуальних проблем, які можна досліджувати в майбутніх дослідженнях. В контексті даної роботи розглядаються наступні аспекти:

- розгортання служби на хості, що поєднує її з операційною системою, і дослідженням потенційних небезпек в даному процесі;
- масштабоване керування програмним забезпеченням вручну, особливо в архітектурах, які використовують мікросервіси;
- питання планування динамічної інфраструктури без постійного розподілу ресурсів, для виконання яких, необхідно використовувати додаткове апаратне забезпечення;
- механізми стійкості хмарних сервісів і вплив помилкової роботи;
- інформація, необхідна для пошуку і від лагодження помилок, які втрачаються недостатню детальність процесу журналізації.

В роботі запропоновано сучасні шаблони проектування для розробки програмного забезпечення хмарних сервісів, еталонну архітектуру для хмарних обчислень, каталог шаблонів і відповідно виконано прикладне дослідження хмарних технологій і практик DevOps, використана мова шаблонів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Giuseppe Aceto et al. “Cloud monitoring: A survey.” In: Computer Networks 57.9 (2013), pp. 2093–2115. issn: 13891286. doi: 10.1016

2. William C. Adams. “Conducting Semi-Structured Interviews.” In: Handbook of Practical Program Evaluation: Fourth Edition August (2015), pp. 492–505. issn: 0190-0447. doi: 10.1002/9781119171386.ch19. arXiv: arXiv:1011.1669v3 (cit. on pp. 150, 151).

3. Tiago Almeida, H.S. Hugo Sereno Ferreira, and Tiago Boldt Sousa. “A Collaborative Expandable Framework for Software End-Users and Programmers.” In: 9th Cooperative Design, Visualization, and Engineering. Vol. 7467 LNCS. Osaka, Japan: Springer Berlin Heidelberg, 2012, pp. 163–166. isbn: 9783642326080. doi: 10.1007/978-3-642-32609-7_22 (cit. on p. 261).

4. Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A Pattern Language: Towns, Buildings, Construction. Ed. by S Ishikawa and M Silverstein. Vol. 2. Center for Environmental Structure 0. Oxford University Press, 1977. Chap. 52, pp. 503–521. isbn: 0195019199 (cit. On pp. 5, 21).

5. Réka Albert, Hawoong Jeong, and Albert László Barabási. “Diameter of the world-wide web.” In: Nature 401.6749 (1999), pp. 130–131. issn: 00280836. doi: 10.1038/43601. arXiv: 9907038 [cond-mat]. url: http://www.nature.com/doifinder/10.1038/43601 (cit. on p. 12).

6. Stephen Abrams, John Kunze, and David Loy. “An Emergent Micro-Services Approach to Digital Curation Infrastructure.” In: International Journal of Digital Curation 5.1 (2010), pp. 172–186. issn: 1746-8256. doi: 10.2218/ijdc.v5i1.151.

7. C Alexander. The Nature of Order, Book 2: The Process of Creating Life. Center for Environmental Structure, 2002, p. 636. isbn: 9780972652926 (cit. on pp. 21, 74).

8. Christopher Alexander. Notes on the Synthesis of Form. 1964. isbn:0-674-62751-2 (cit. on p. 21).

9. Christopher Alexander. The Timeless Way of Building. Oxford University Press, 1979. isbn: 0195024028 (cit. on pp. 6, 21, 71).

10. Saleem Alhabash and Mengyan Ma. “A Tale of Four Platforms: Motivations and Uses of Facebook, Twitter, Instagram, and Snapchat Among College Students?” In: Social Media and Society 3.1 (2017). issn: 20563051. doi:10.1177/2056305117691544 (cit. on p. 194).

11. Amazon. AWS Architecture Center. url: https://aws.amazon.com/architecture/

12. Amazon. Amazon EC2 Container Service. 2015. url: https://aws.amazon.com/docker/

13. Amazon. Amazon Cloudtrail. 2017. url: https://aws.amazon.com/cloudtrail/

14. Amazon. Scheduled Tasks (cron). 2017. url: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/scheduled%7B%5C_%7Dtasks.html

15. Paul Anderson. Web 2.0 and beyond: Principles and technologies. CRC Press, 2016, pp. 1–412. isbn: 9781439828687. doi: 10.5860/choice.50-3893 (cit. On pp. 2, 12).

16. Arcitura Education Inc. Dynamic Failure Detection and Recovery. url: http://cloudpatterns.org/design%7B%5C_%7Dpatterns/dynamic%7B%5C_%7Dfailure

17. Arcitura Education Inc. Cloud Patterns. 2019. url: https://patterns.arcitura.com/cloud-computing-patterns (cit. on pp. 34, 122).

18. Michael Armbrust et al. “A view of cloud computing.” In: Communications of the ACM 53.4 (2010), pp. 50–58. issn: 00010782. doi:0.1145/1721654.1721672. url: http://portal.acm.org/citation.cfm?doid=1721654.1721672 (cit. on pp. 30, 31).

19. Roger Atkinson. “Project management: Cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria.” In: International Journal of Project Management 17.6 (1999), pp.

20. Azure. Azure Logging and Auditing. 2017. url: https://docs.microsoft.com/enus/azure/security/azure-log-audit.

21. Prith Banerjee et al. “Everything as a service: Powering the new information economy.” In: Computer 44.3 (2011), pp. 36–43. issn: 00189162. doi: 10.1109/MC.2011.67 (cit. on pp. 2, 15).

22. Richard Baskerville et al. “How internet software companies negotiate quality.” In: Computer 34.5 (2001), p. 51. issn: 00189162. doi: 10 . 1109 /2.920612 (cit. on p. 18).

23. Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. Tech. rep. 10. Maryland, USA: Univ. of Maryland, Dep. of Computer Science, 1996, pp. 751–761.

24. Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. “Patterns for Software Orchestration on the Cloud.” In: 22nd Conference on Pattern Languages of Programs. Pittsburgh, Pennsylvania, USA., 2015

25. Kent Beck et al. Manifesto for Agile Software Development. 2001. url: http: //agilemanifesto.org/.

26. Dominique Bellenger et al. “Scaling in cloud environments.” In: Recent Researches in Computer Science - Proceedings of the 15th WSEAS International Conference on Computers, Part of the 15th WSEAS CSCC Multiconference (2011), pp. 145–150 (cit. on p. 2).

27. Tiago Boldt Sousa and Hugo Sereno Ferreira. “Object-Functional Patterns: Re-thinking Development in a Post-Functional World.” In: 8th Quality of Information and Communications Technology (QUATIC). Lisbon, Portugal: IEEE, 2012, pp. 348–352 (cit. on p. 262).

28. Frank Buschmann, Kevlin Henney, and Douglas Schmidt. Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. 2007, p. 639.

29. Jim Bird. Devops for Finance. Vol. 1. 2015. isbn: 9788578110796 (cit. On p. 30).

30. Fernando Brito e Abreu and Walcelio Melo. “Evaluating the impact of object-oriented design on software quality.” In: International Software Metrics Symposium, Proceedings (1996), pp. 90–99.

31. F Bushmann, R Meunier, and H Rohnert. Pattern-oriented software architecture: A System of Patterns, Volume 1. Vol. 1. Wiley Publishing, 1996, p. 476. isbn: 9780471958697.

32. Tiago Boldt Sousa et al. “Engineering Software for the Cloud – Patterns and Sequences.” In: 11th Latin American Conference on Pattern Languages of Programs Programs. 11. Buenos Aires, Argentina, 2016,

33. Tiago Boldt Sousa et al. “Engineering Software for the Cloud: Messaging Systems and Logging.” In: 22nd European Conference on Pattern Languages of Programs. Irsee, Bavaria, Germany, 2017. isbn: 978-1-4503-4848-5.

34. Tiago Boldt Sousa et al. “Engineering Software for the Cloud: Automated Recovery and Scheduler.” In: 23rd European Conference on Pattern Languages of Programs. Irsee, Bavaria, Germany., 2018 (cit. on p. 258).

35. Tiago Boldt Sousa et al. “Engineering Software for the Cloud: External Monitoring and Fault Injection.” In: 23rd European Conference on Pattern Languages of Programs. Irsee, Bavaria, Germany., 2018 (cit. on p. 258).

36. Jonas Bonér et al. “The Reactive Manifesto (Version 2.0).” In: Reactivemanifesto.Org 2.16 September 2014 (2014), pp.1–2.

37. Jonas Bonér. Reactive Microservices Architecture Design Principles for Distributed Systems. 2016, p. 54. isbn: 978-1-491-95779-0 (cit. on pp. 29, 152).

38. Grady Booch. Object-Oriented Analysis and Design with Applications (3rd Edition). Addison Wesley Longman Publishing Co., Inc., 2004.

39. Morgan Bruce and Paulo A. Pereira. Microservices in action. Manning Publications, 2019, p. 366. isbn: 1617294454 (cit. on pp. 30, 170).

40. Antonio Brogi et al. “SeaClouds.” In: ACM SIGSOFT Software Engineering Notes 39.1 (2014), pp. 1–4. issn: 01635948. doi: 10.1145/2557833.2557844.

41. Malcolm Bronte-stewart. “Beyond the Iron Triangle: Evaluating Aspects of Success and Failure using a Project Status Model.” In: Computing & Information Systems 19.2 (2015), pp. 21–37 (cit. on p. 4).

42. Thanh Bui. “Analysis of Docker Security.” In: Computing Research Repository abs/1501.0 (2015), p. 7.url:http://arxiv.org/abs/1501.02967.

43. James Casey et al. “A messaging infrastructure for WLCG.” In: Journal of Physics: Conference Series 331.PART 6 (2011). issn: 17426596. doi: 10.1088/1742-6596/331/6/062015 (cit. on pp. 30, 143).

44. D Cohen and B Crabtree. Semi-structured Interviews. 2006. url: http://www.qualres.org/HomeSemi-3629.html.

45. Omar Castro, Hugo Sereno Ferreira, and Tiago Boldt Sousa. Collaborative Web Platform for UNIX-Based Big Data Processing. Seattle, WA, USA, 2014.

46. Robert N. Charette. Why Software Fails. 2005. doi: 10.1109/MSPEC.2005. 1502528. url: http://spectrum.ieee.org/computing/software/whyoftwarefails

47. John Chinneck, Marin Litoiu, and Murray Woodside. “Real-time multi-cloud management needs application awareness.” In: ICPE 2014 - Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (2014), pp. 293–296. doi: 10.1145/2568088.2576763.

48. Mario Callegaro, Katja Lozar Manfreda, and Vasja Vehovar. Web survey methodology. Sage, 2015 (cit. on p. 192).

49. Fernando Dias Correia et al. “Home-based Rehabilitation With A Novel Digital Biofeedback System versus Conventional In-person Rehabilitation

after Total Knee Replacement: a feasibility study.” In: Scientific Reports 8.1 (2018), pp. 1–12. issn: 20452322. doi: 10.1038/s41598-018-29668-0.

50. Mick P. Couper. “Web Surveys.” In: Public Opinion Quarterly 64.4 (2000), pp. 464–494. issn: 0033362X. doi: 10.1086/318641.

51. Ward Cunningham. Let It Crash. 2014. url: http://wiki.c2.com/?LetItCrash (cit. on p. 109).

52. Michael Cusumano. “Cloud computing and SaaS as new computing platforms.” In: Communications of the ACM 53.4 (2010), pp. 27–29. issn:00010782. doi: 10.1145/1721654.1721667 (cit. on pp. 16, 17).

53. Cycligent. Continuous Delivery Patterns for Design and Deployment. Tech. rep. 2015 (cit. on pp. 41, 89).

54. Maximilien De Bayser, Leonardo G. Azevedo, and Renato Cerqueira. “ResearchOps: The case for DevOps in scientific applications.” In: Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015 (2015), pp. 1398–1404. doi: 10 . 1109 / INM . 2015.

метадані

Заголовок

Порівняльний аналіз процесів розробки та адаптації моделей та шаблонів хмарних рішень та сервісів

Автор

Наумкін В.В. Науковий керівник / Експерт

підрозділ

King Danylo University

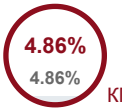
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

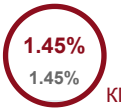
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	49

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КП 2

25
Довжина фрази для коефіцієнта подібності 2

16145
Кількість слів

125762
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua:8080/bitstream/handle/123456789/379/%D0%91%D0%B0%D0%BB%D0%B0%D0%BD%D1%8E%D0%BA_%D0%86_%D0%86_%D0%9A%D0%92%D0%90%D0%9B%D0%86%D0%A4%D0%86%D0%9A%D0%90%D0%A6%D0%86%D0%98%CC%86%D0%9D%D0%90_%D0%A0%D0%9E%D0%91%D0%9E%D0%A2%D0%90.pdf?sequence=1	46	0.28 %
2	http://repository.ukd.edu.ua:8080/bitstream/handle/123456789/379/%D0%91%D0%B0%D0%BB%D0%B0%D0%BD%D1%8E%D0%BA_%D0%86_%D0%86_%D0%9A%D0%92%D0%90%D0%9B%D0%86%D0%A4%D0%86%D0%9A%D0%90%D0%A6%D0%86%D0%98%CC%86%D0%9D%D0%90_%D0%A0%D0%9E%D0%91%D0%9E%D0%A2%D0%90.pdf?sequence=1	38	0.24 %