Міністерство освіти та науки України
Харківський національний університет радіоелектроніки

Факультет *Інфокомунікацій*
(повна назва)
Кафедра *Інформаційно – мережна інженерія*
(повна назва)

# АТЕСТАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)
*Проектування системи автоматизованого розгортання інфраструктури веб–проекту*
(тема)

Виконав:
студент *2* курсу, групи *ІМІм-18-1*
*Лисенко М. І.*
(прізвище, ініціали)

Спеціальність *172 "Телекомунікації та радіотехніка"*
(код і повна назва спеціальності)
Тип програми *освітньо–професійна*
(освітньо-професійна або освітньо-наукова)
Освітня програма *Інформаційно – мережна інженерія*
(повна назва освітньої програми)
Керівник *доц. к.т.н. Костромицький А. І.*
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ _____
(підпис) (прізвище, ініціали)

20 *19* р.

Не містить відомостей, заборонених до відкритого публікування

Студент _____

Керівник_____

Харківський національний університет радіоелектроніки

Факультет *Інфокомунікацій*
Кафедра *Інформаційно – мережна інженерія*
Рівень вищої освіти другий (магістерський)

Спеціальність *172 "Телекомунікації та радіотехніка"*
(код і повна назва)
Тип програми *освітньо – професійна*
(освітньо – професійна або освітньо – наукова)

Освітня програма *Інформаційно – мережна інженерія*
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2019 р.

# ЗАВДАННЯ
## НА АТЕСТАЦІЙНУ РОБОТУ

студентові *Лисенко Максиму Ігоровичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Проектування системи автоматизованого розгортання інфраструктури веб – проекту*
затверджена наказом по університету від *31 10* 2019р. № *1609 Ст*
2. Термін подання студентом роботи до екзаменаційної комісії ____ _____ 2019 р.
3. Вихідні дані до роботи *Огляд та порівняльна характеристика інструментів автоконфігурації та оркестрації, вибір підходящих, виходячі з вихідних даних, побудова проекту на основі вибранних інструментів та вихідних даних*
4. Перелік питань, що потрібно опрацювати в роботі
*Вступ*
*1. Загальні принципи побудови інфраструктури веб – проекту*
*2. Аналіз та порівняльна характеристика інструментів автоматизованого розгортанння інфраструктури веб – проекту*
*3. Аналіз та порівняльна характеристика інструментів оркестрації ресурсів веб – проекту*
*4. Проектування системи автоматизованого розгортання ірнфраструктури веб – проекту*
*Висновки*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) _____

*Слайди у форматі PowerPoint(мета дослідження, порівняння методологій розробки веб – проекту, архітектура інструментів автоконфігурації, висновки, тощо)*

# КАЛЕНДАРНИЙ ПЛАН

| № | Назва етапів роботи | Терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | *Ознайомлення із завданням. Уточнення ТЗ.* | *31.10.19* | |
| 2 | *Підбір літератури за темою роботи.* | *17.11 – 20.11.19* | |
| 3 | *Виконання розділу 1* | *21.11 – 23.11.19* | |
| 4 | *Виконання розділу 2* | *24.11 – 26.11.19* | |
| 5 | *Виконання розділу 3* | *27.06 – 28.11.19* | |
| 6 | *Виконання розділу 4* | *28.11 – 01.12.19* | |
| 7 | *Оформлення презентаційного матеріалу,* | *01.12 – 06.12.19* | |
| | *підготовка до захисту у ДЕК* | | |
| | | | |
| | | | |
| | | | |
| | | | |

Дата видачі завдання *31 жовтня 2019 р.*

Студент _____
(підпис)

Керівник роботи _____ *доц. к.т.н. Костромицький А. І.*
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ

Пояснювальна записка: – 91 с., 32 рис., 3 табл., 11 джер, 1 додаток.

Об'єкт дослідження – веб – проект.

Мета роботи: огляд загальних принципів побудови веб – проекту; аналіз та порівняння найпоширеніших засобів автоматизації інфраструктури веб – проекту та оркестрації відповідних ресурсів; побудова веб – проекту на основі вхідних данних.

Передбачувані наукові результати: демонстрація можливостей автоматизації, оркестрації та автоконфігуріції; формування рекомендацій щодо впровадження автоматизації до інфраструктури веб – проекту; побудова автоматизованої інфраструктури веб – проекту.

ВЕБ ПРОЕКТ, АВТОМАТИЗАЦІЯ, ОРКЕСТРАЦІЯ, БЕЗПЕКА, АВТОКОНФІГУРАЦІЯ

# THE ABSTRACT

Explanatory note: – 91 p., 32 fig., 3 tab., 11 sources, 1 app.

Target of research – web – project.

Objective: general principles overview of web – project construction; analysis and comparison the most common ways of web – project infrastructure automation and related resources orchestration; building web – project based on input data.

Prospective scientific results: demonstration of automation, orchestration and autoconfiguration abilities; the formation of recommendations related to automation introducing to web – project infrastructure; building the automated web – project infrastructure.

WEB – PROJECT, AUTOMATION, ORCHESTRATION, SECURITY, AUTOCONFIGURATION

ЗМІСТ

# ПЕРЕЛІК СКОРОЧЕНЬ

API – (Application Programming Interface) – інтерфейс програмування додатків, програмний інтерфейс програми;
AWS – (Amazon Web Services) – комерційна публічна хмара;
CI – (Continious Integration) – практика розробки програмного забезпечення;
CD – (Continious Deployment) – практика розробки програмного забезпечення;
DevOps – (development + operations) – методологія взаємодії системних адміністраторів та розробників;
DNS – (Domain Name System) – комп'ютерна розподілена система для отримання інформації про домени;
ECS – (Elastic Container Service) – високопродуктивний сервіс оркестрації контейнерів;
CSS – (Cascading Style Sheets) – формальна мова опису зовнішнього вигляду документа, написаного з використанням мови розмітки;
HCL – (HashiCorp Configuration Language) – мова конфігурації, автором якої є HashiCorp;
HTML – (HyperText Markup Language) – стандартизована мова розмітки документів;
IaC – (Infrastructure as Code) – підхід для управління і опису інфраструктури центру обробки даних через конфігураційні файли;
QA – (quality assurance) – фахівець із забезпечення якості програмного забезпечення;
SCM – (Software Configuration Management) – інструмент для управління конфігурацією;
SSH – (Secure Shell) – мережний протокол прикладного рівня;
SQL – (Structured Query Language) – мова управління базами даних для реляційних баз даних;
VM – (Virtual Machine) – віртуальна машина;
ЦОД – (Центр обробки даних) – центр зберігання та обробки даних.

# ВСТУП

На сьогоднішній день існує безліч інструментів які забезпечують автоматизований підхід до розгортання інраструктури веб – проекту. Відомі інструменти автоконфігурації, такі як: Puppet, Chef, Ansible в данний час користуються популярністю за рахунок широкої аудиторії користувачів та вкладників. Інструменти оркестрації, такі як: Terraform, Cloud Formation завоювали сердця користувачів за рахунок низького порогу входження в інструментарій та прозорість можливостей та способів налаштування. Інстументи, які реалізують CI/CD концепції: Jenkins TeamCity, Bamboo, CircleCI мають дуже широкий діапазон можливостей, які відносяться до автоматизованої побудови бекенда веб – проекту та мають майже нескінченну можливість розширення функціоналу в залежності до вимог.

Актуальність автоматизованого підходу полягає у скороченні використання людських ресурсів за рахунок відмови від ручних дій, а отже, є можливість виділити більше часу на впровадження нових функцій до продукта, засобів безпеки веб – проекту. При впровадженні автоматизації, компанія стає більш конкурентоспроможною, за рахунок використання новітніх технологій та підходів, за рахунок більш гручкого розподілу людських ресурсів.

Метою даної роботи є огляд найпоширеніших способів побудови інфраструктури веб – проекту, опис та порівняльна характеристика інструментів автоконфігурації та оркестрації ресурсів веб – проекту, вибір оптимальних інструментів для веб – проекту виходячи з вихідних даних.

# 1 ЗАГАЛЬНІ ПРИНЦИПИ ПОБУДОВИ ІНФРАСТРУКТУРИ ВЕБ – ПРОЕКТУ

## 1.1 Цикл розробки веб – проекту

Незважаючи на загальноприйняту думку, центральне місце в процесі дизайну і розробки веб – сайтів не завжди займає фаза написання коду. В першу чергу приходять на розум технології, такі як HTML, CSS і JavaScript, і справді створюють образ мережі, до якого ми звикли і визначають способи нашої взаємодії з інформацією. Що зазвичай залишається поза увагою, але в той же час є чи не найважливішою частиною процесу розробки, так це стадії попереднього збору інформації, ретельного планування, а також підтримки вже після запуску сайту.

У цьому розділі ми поговоримо про те, як може виглядати типовий процес розробки веб – сайту. Можна виділити різну кількість етапів, з яких складається процес розробки. Зазвичай це число від п'яти до восьми, але в кожному випадку загальна картина залишається приблизно однаковою. Давайте зупинимося на середньому значенні. Отже, сім основних етапів розробки:

- Збір інформації;
- Планування;
- Дизайн;
- Створення контенту;
- Розробка;
- Тестування, огляд і запуск;
- Підтримка.

Коли настає час планування процесу розробки веб – сайту, дві головні проблеми, з якими можна стикнутися, це час і вартість розробки. Ці два значення багато в чому залежать від розміру і складності проекту. Для того, щоб представляти в загальних рисах, як буде протікати робота над проектом, можна створити графік процесу розробки, який буде містити основні завдання проекту, а також етапи, з яких він складається. Це дозволить зручно стежити за загальною картиною і завжди бути впевненим в тому, що поставлені завдання будуть вирішені точно в строк. Для даної я вважаю за краще використовувати GanttPRO, зручну діаграму Гантта для управління проектами та завданнями онлайн (рис 1.1).

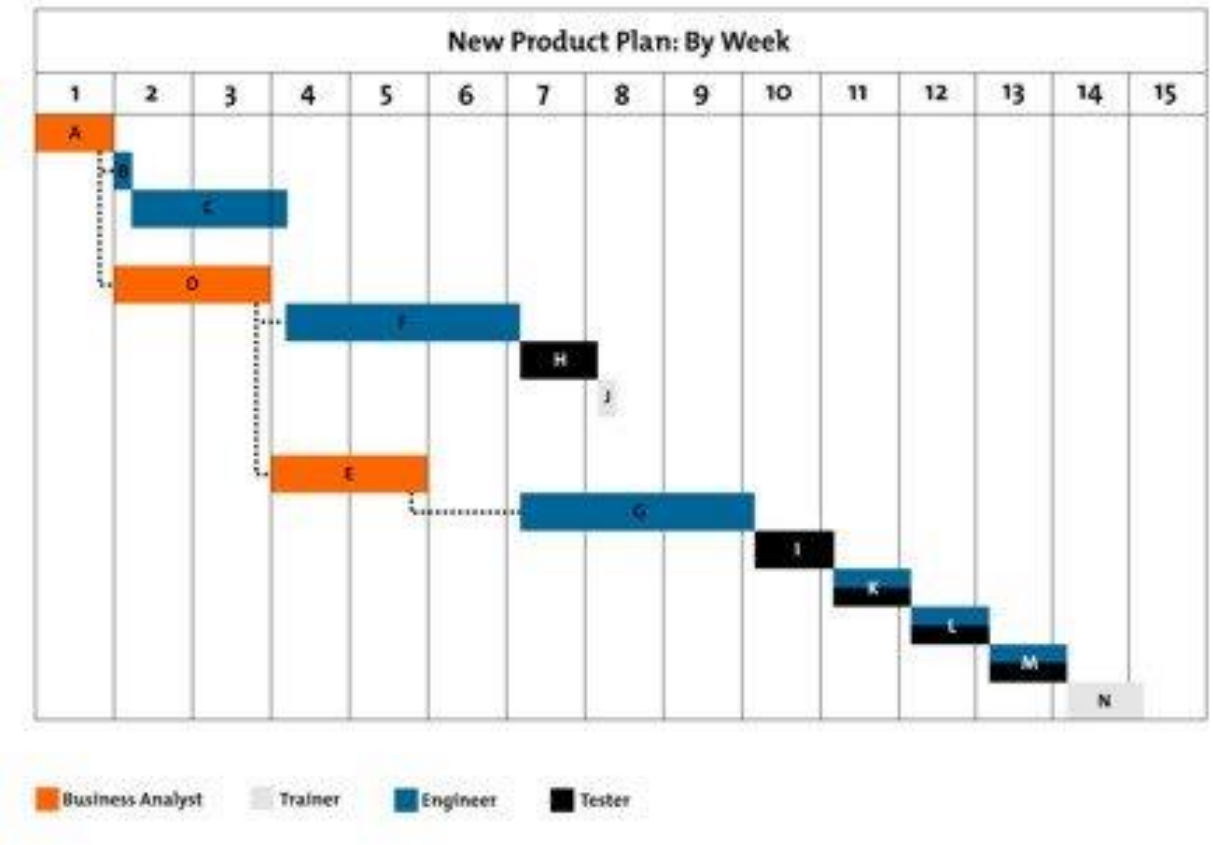


Рисунок 1.1 – Діаграма Гантта

Етап 1. Збір інформації: призначення, основні цілі та цільова аудиторія. Етап попереднього дослідження та збору інформації визначає те, як будуть протікати всі наступні стадії розробки. Найважливіше на цьому етапі – отримати чітке і повне розуміння того, яким буде призначення вашого майбутнього сайту, з якою метою ви хочете досягти з його допомогою, а також яка цільова аудиторія, яку ви хочете на нього залучити[1].

Така своєрідна анкета веб – розробки дозволить визначити найкращу стратегію подальшого розвитку проекту. Новинні портали відрізняються від розважальних сайтів, а сайти для підлітків відрізняються від таких для дорослої аудиторії. Різні сайти надають відвідувачам різну функціональність, а значить різні технології повинні використовуватися в тому чи іншому випадку. Детально складений план, створений на основі даних, отриманих на цьому етапі, може запобігти від витрати додаткових ресурсів на рішення непередбачених труднощів,

таких як зміна дизайну або додавання функціоналу, непередбаченого спочатку. Приблизний час: від 1 до 2 тижнів.

Етап 2. Планування: створення карти сайту і макета. На цій стадії розробки замовник вже може отримати уявлення про те, яким буде майбутній сайт. На основі інформації, зібраної на попередній стадії, створюється карта сайту (sitemap). Так, наприклад, виглядає карта сайту XB Software (рис. 1.2):

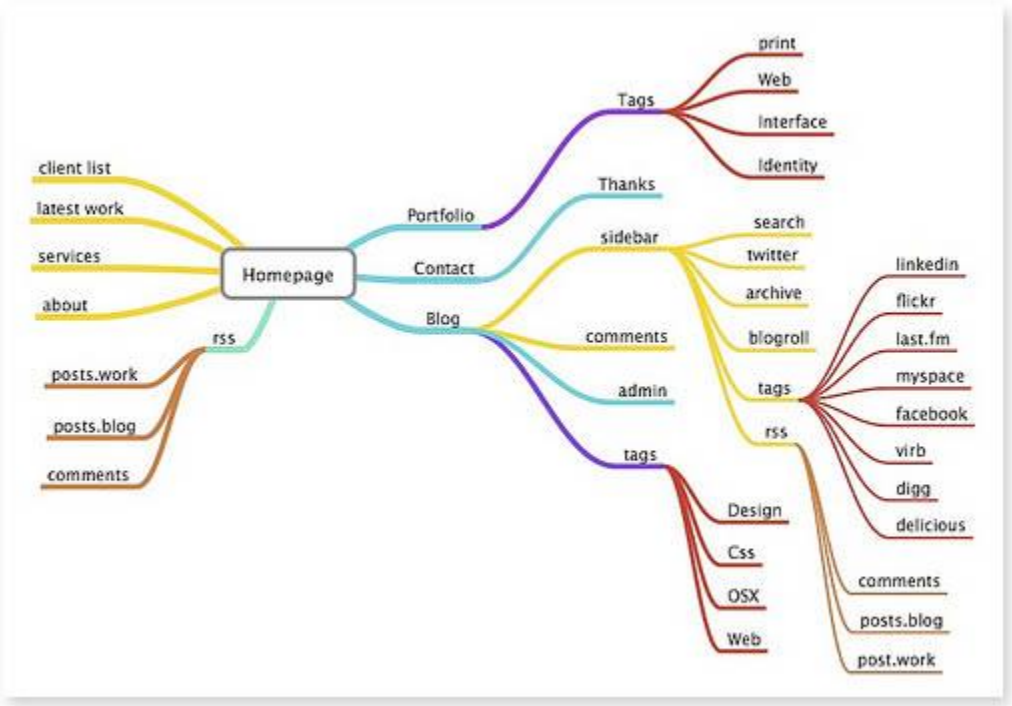


Рисунок 1.2 – Карта сайту XB Software

Карта сайта описує взаємозв'язок між різними частинами сайту. Це допомагає зрозуміти, наскільки зручним у використанні він буде. По карті сайту можна визначити «відстань» від головної сторінки до інших сторінок, що допомагає судити про те, наскільки просто користувачеві буде дістатися до цікавить його інформацією. Основна мета створення карти сайту – створити легкий з точки зору навігації і дружній до користувача продукт[1]. Це дозволяє зрозуміти внутрішню структуру майбутнього сайту, але не описує те, як сайт буде виглядати. Іноді, перш ніж приступити до написання коду або до розробки дизайну, може бути важливим отримати схвалення замовника.

У цьому випадку створюється макет (wireframe або mock – up). Макет являє з себе візуальне уявлення майбутнього інтерфейсу сайту. Але, на відміну, наприклад, від шаблону, про який ми поговоримо далі, він не містить елементів дизайну, таких як колір, логотипи, і т.п. Він тільки описує, які елементи будуть поміщені на сторінку і як вони будуть розташовані. Макет являє собою свого роду начерк майбутнього сайту. Можна використовувати один з доступних онлайн – сервісів для створення макетів. Приблизний час: від 2 до 6 тижнів.

Етап 3. Дизайн: шаблон сторінки, огляд та затвердження. На цьому етапі веб – сайт стає ще ближче до своєї остаточної форми. Весь візуальний контент, наприклад, на зображеннях, фото і відео, створюється саме зараз. І знову вся інформація, яка була зібрана на першій стадії проекту, вкрай важлива на цьому кроці. Інтереси замовника, а також цільова аудиторія повинні враховуватися в першу чергу під час роботи над дизайном. Дизайнером на даному етапі створюється шаблон сторінки (page layout). Основне призначення шаблону – візуалізувати структуру сторінки, її вміст, а також відобразити основний функціонал.

На цей раз, на відміну від макета, використовуються елементи дизайну. Шаблон містить кольори, логотипи і зображення. Він дає можливість судити про те, як в кінцевому результаті буде виглядати готовий сайт. Після створення шаблон може бути відправлений замовнику[1]. Після огляду замовником виконаної роботи, він надсилає свій відгук. Якщо його не влаштовують якісь аспекти дизайну, треба змінити існуючий шаблон і знову відправити його замовнику. Цей цикл повторюється до тих пір, поки замовник не буде повністю задоволений результатом. Приблизний час: від 4 до 12 тижнів.

Етап 4. Створення контенту. Процес створення контенту зазвичай проходить паралельно з іншими стадіями розробки і його роль не варто недооцінювати. На даному етапі необхідно описати саму суть того, що хочеться донести до аудиторії свого веб – сайту. Ця стадія включає в себе також створення привабливих і помітних заголовків, написання і редагування тексту, компіляція існуючих текстів і т.д. Все це вимагає витрати додаткового часу і зусиль. Як правило, замовник надає контент, вже готовий до того, щоб бути розміщеним на сайті. Важливо, щоб весь контент був підготовлений до або під час стадії розробки. Приблизний час: від 5 до 15 тижнів.

Етап 5. Верстка та розробка. Тепер можна нарешті перейти безпосередньо до верстки сайту. Всі графічні елементи, розроблені раніше, використовуються на

даній стадії. Зазвичай в першу чергу створюється домашня сторінка, а потім до неї додаються інші сторінки у відповідності з ієрархією, розробленої на етапі створення карти сайту. Також на цьому етапі відбувається установка CMS. Всі статичні елементи веб – сайту, дизайн яких був розроблений раніше при створенні шаблону, перетворюються в реальні динамічні інтерактивні елементи веб – сторінки. Важлива задача – проведення SEO – оптимізації (Search Engine Optimization), яка являє собою оптимізацію елементів веб – сторінки (заголовків, опису, ключових слів) з метою підняття позицій сайту в результатах видачі пошукових систем. Валідність коду є вкрай важливою в цьому випадку. Приблизний час: від 6 до 15 тижнів.

Етап 6. Тестування та запуск. Тестування є, напевно, найбільш рутинною частиною розробки. Кожне посилання має бути перевірене, кожна форма і кожен скрипт повинні бути протестовані. Текст повинен бути перевірений програмою перевірки орфографії для виявлення можливих помилок. Валідатори коду використовуються для того, щоб бути впевненим, що створений на попередньому етапі код повністю відповідає сучасним веб – стандартам. Це може виявитися вкрай важливим, якщо для вас критична, наприклад, кросбраузерна сумісність. Після того, як сайт перевірений, він може бути завантажений на сервер. Зазвичай для цього використовується FTP – клієнт[1]. Після завантаження сайту на сервер, необхідно провести ще один тест для того, щоб бути впевненим, що під час завантаження не відбулося непередбачених помилок і всі файли цілі та неушкоджені. Приблизний час: від 2 до 4 тижнів.

Етап 7. Підтримка: відгуки користувачів і регулярні оновлення. Важливо розуміти, що веб – сайт являє собою скоріше сервіс, ніж продукт. Недостатньо просто «доставити» його споживачеві. Також важливо бути впевненим в тому, що все працює, як і було заплановано, а користувачі задоволені кінцевим продуктом. Потрібно також бути готовим швидко вносити зміни, якщо це буде необхідно. Система відгуків дозволить виявляти виниклі проблеми, з якими стикаються відвідувачі сайту. Самим критичним завданням в подібних випадках буде вирішення виниклих проблем настільки швидко, наскільки це можливо. В іншому випадку, користувачі швидше віддадуть перевагу іншому ресурсу, ніж будуть миритися з незручностями. Також не слід забувати про регулярне оновлення CMS. Регулярні оновлення позбавлять вас від помилок і проблем з безпекою. Безперервний процес.

Висновок. Потрібно постійно пам'ятати, що процес розробки веб – сайту не починається з написання коду і не закінчується після запуску сайту. Етап підготовки зачіпає всі наступні етапи, визначаючи те, наскільки продуктивним виявиться процес роботи над проектом. Грунтовне і глибоке дослідження таких аспектів, як стать, вік і інтереси кінцевих користувачів може виявитися визначальним. Підтримка сайту вже після його запуску також вкрай важлива.

Треба бути досить оперативними, щоб мати можливість швидко виправляти виниклі помилки і вирішувати виниклі у користувачів проблеми. Розуміння того, що серед етапів розробки веб – сайту немає таких, які можна було б вважати маловажливими або необов'язковими, допоможе уникнути зайвого клопоту і дасть впевненість в тому, що робота над проектом рухається так, як і було задумано та існує повний контроль над процесом розробки (рис. 1.3).

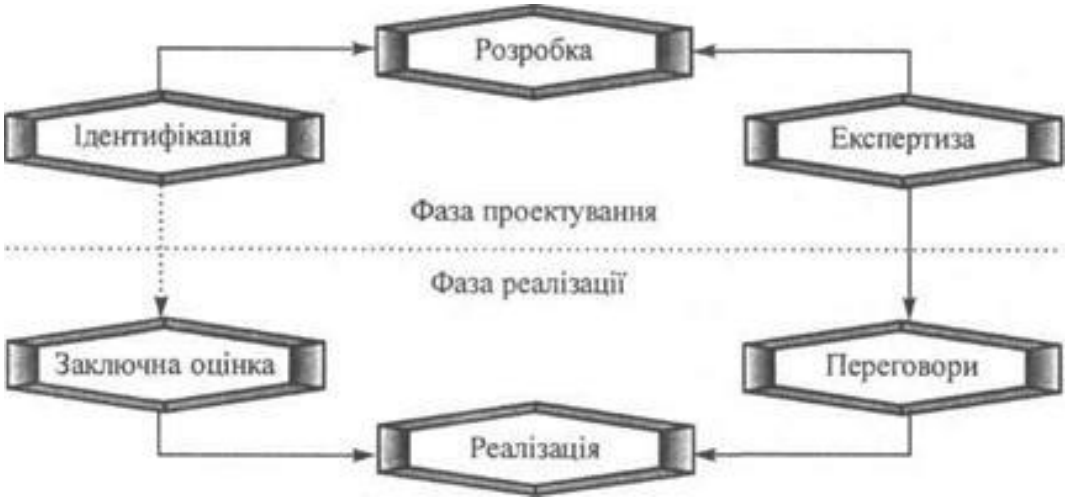


Рисунок 1.3 – Цикл розробки веб – проекту

## 1.2 Методології розробки веб – проекту

Перед тим, як перейти до методологій, ми розглянемо 8 основних принципів планування розробки, які значно спрощують життя.

Етап планування передбачає усвідомлену і цілеспрямовану діяльність команди на шляху до досягнення результату. Визначити завдання, розбити на кроки, передбачити терміни – необхідний крок на шляху втілення задуманого. Особливо, якщо справа стосується гнучкої методології Agile, яку я вважаю

найкращою. Команди розробників допускають наступні помилки при плануванні створення ПО.

Планувати терміни повинні програмісти, а не менеджери.

Часта помилка, коли керівник проекту, який знає належним чином обсяг і специфіку завдань, встановлює терміни проекту не відповідно до досвіду, можливостям і компетенціям команди, а спираючись на власні уявлення, бажання або запити клієнта. Програмістам в подібних групах не позаздриш. Розбіжності запланованих і реальних термінів становить 40 – 80%[2]. Атмосфера в колективі створюється погана і відбиває бажання працювати. Проблеми слідують одна за одною, а винними виставляються безпосередньо розробники.

Необхідно заздалегідь визначати приблизні терміни здачі всього проекту і реальний час вирішення завдання.

Відпускати процеси на самоплив ні в якому разі не можна. Ігнорування процедури планування призводить до низької мотивації розробників в попередні дедлайну періоди, до нерозуміння командою, що робити, куди рухатися і що потрібно отримати в результаті. В об'єднаннях, де приблизні терміни здачі проекту не визначаються, бажано задуматися про те, що подібний хаос до добра не доведе.

Проект розбивайте на дрібні етапи з чіткими цілями і обов'язковим обговоренням результатів.

Застосування принципу необхідно для протидії закону Паркінсона, який визначає, що загальний обсяг роботи завжди буде збільшуватися, щоб заповнити весь виділений на роботу час. Слідуючи раді, можна уникнути бажання старанно працювати лише незадовго до терміну здачі проекту. Розбиття процесу досягнення глобальної мети на контрольні періоди з необхідністю виконання конкретних завдань протягом тижня – двох, дозволить використовувати робочий потенціал команди по максимуму. При зазначеному підході підтримується високий рівень мотивації і працездатності розробників протягом усього періоду створення ПЗ і збільшується ймовірність досягнення бажаних цілей.

Члени команди повинні максимально взаємодіяти один з одним. Перш за все підвищується згуртованість колективу і стимулюється надання взаємодопомоги. При недостатньому спілкуванні між членами команди, відсутній «командний дух», що забезпечує злагоджену роботу. Спільна продуктивна діяльність задовольняє соціальні потреби людини у відчутті значущості виконуваної роботи. Дотримання принципу дозволяє безболісно замінити будь – якого члена команди, тому що учасники в курсі хто, що і як робить.

Включайте резерв часу для покриття форс – мажору, нових вимог замовника, відпусток і свят, на інтеграцію і тестування. На початковому етапі планування можна передбачити всі ситуації. Тому потрібно зарезервувати час про запас, щоб команді не довелося поспішати і, як наслідок, робити помилки. Не варто ігнорувати необхідність налагодження і доведення ПО до рівня стабільної роботи і прийнятої кількості багів. Випуск сирого продукту через жорстку економію часу не розумний. Методологія Agile передбачає мінливість зовнішніх умов і необхідність швидкої і безболісної адаптації до них.

Не можна поспішати, порушувати план і зменшувати час розробки ПЗ. Часта помилка менеджерів, які думають, що програмісти зможуть потягнути будь – які терміни. По – перше, команда демотивує, саботує процес праці або пише заяви за власним бажанням. По – друге, різке прискорення робочих операцій виснажує ресурси організму і психіки людини, веде до професійного вигорання[2]. По – третє, завищений темп призводить до збільшення кількості помилок в коді. На налагодження і виправлення в майбутньому буде потрібно значно більше часу, ніж вийде заощадити подібним чином.

Документуйте планування за допомогою відповідного таск – менеджера. Вибір конкретної програми є питання смаку. Плани слід фіксувати. Потрібна наочність як для розробників, так і для клієнтів, зберігаючи можливість для внесення змін. Це дозволяє поліпшити взаєморозуміння команди розробників, менеджменту і клієнта. Знижується кількість суперечок щодо трактування робочих дій. Чіткість формулювання плану допоможе уникнути двоякого тлумачення.

Розставляйте пріоритети завданням і концентруйтеся на головному. Намагайтеся в першу чергу реалізовувати найбільш важливий функціонал. Майте на увазі, що якимись функціями в процесі розробки доведеться пожертвувати, так само як і реалізацією частини ідей. А розставити пріоритети можливо виключно через спілкування і обмін думками.

«Waterfall Model» (каскадна модель або «водоспад»).

Одна з найстаріших, передбачає послідовне проходження стадій, кожна з яких має завершитися повністю до початку наступної. У моделі Waterfall легко керувати проектом. Завдяки її жорсткості, розробка проходить швидко, вартість і термін заздалегідь визначені. Але це палиця з двома кінцями. Каскадна модель буде давати відмінний результат тільки в проектах з чітко і заздалегідь визначеними вимогами і способами їх реалізації. Немає можливості зробити крок назад, тестування починається тільки після того, як розробка завершена або майже

завершена. Продукти, розроблені за цією моделлю без обґрунтованого її вибору, можуть мати недоліки (список вимог не можна скорегувати в будь – який момент), про які стає відомо лише в кінці через суворої послідовності дій (рис. 1.4). Вартість внесення змін висока, так як для її ініціалізації доводиться чекати завершення всього проекту[2]. Проте, фіксована вартість часто переважує мінуси підходу. Виправлення усвідомлених в процесі створення недоліків можливо, і вимагає від одного до трьох додаткових угод до контракту з невеликим ТЗ.

Переваги:

- Повна і узгоджена документація на кожному етапі;
- простота використання;
- стабільні вимоги;

Недоліки:

- Велика кількість документації;
- не дуже гнучка система;
- немає можливості повернутися на крок назад.

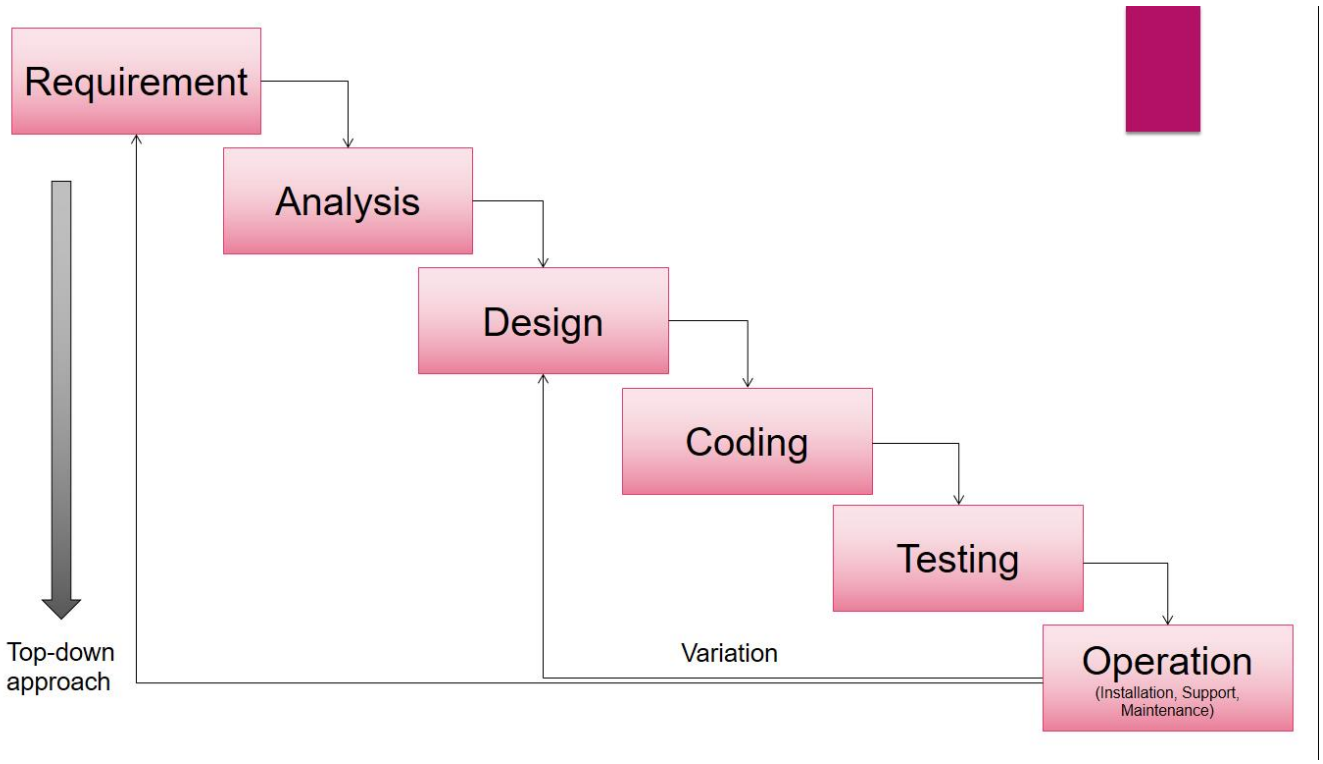


Рисунок 1.4 – Waterfall Model

«V – Model».

Успадкувала структуру «крок за кроком» від каскадної моделі. V – образна модель застосовна до систем, яким особливо важливо безперебійне функціонування. Наприклад, прикладні програми в клініках для спостереження за пацієнтами, інтегроване ПО для механізмів управління аварійними подушками безпеки в транспортних засобах і так далі. Особливістю моделі можна вважати те, що вона спрямована на ретельну перевірку і тестування продукту, що знаходиться вже на початкових стадіях проектування[2].

Використовування V – модель. Якщо потрібне ретельне тестування продукту, то V – модель виправдає закладену в себе ідею (рис. 1.5); validation and verification; для малих і середніх проектів, де вимоги чітко визначені і фіксовані; в умовах доступності інженерів необхідної кваліфікації, особливо тестувальників.

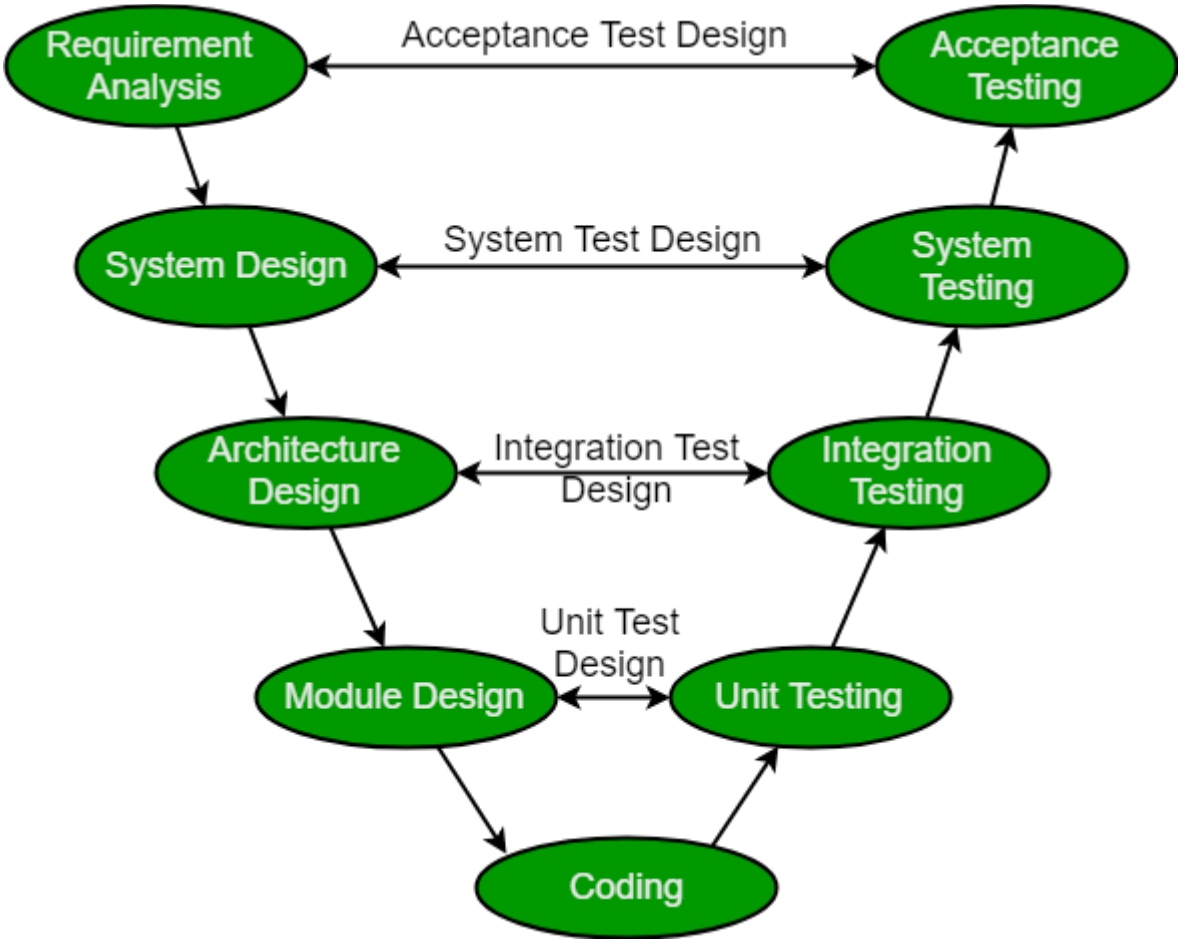


Рисунок 1.5 – V – Model

«Incremental Model» (інкрементна модель).

У інкрементної моделі повні вимоги до системи діляться на різні збірки. Термінологія часто використовується для опису поетапної збірки ПО. Мають місце кілька циклів розробки, і разом вони складають життєвий цикл «мульти – водоспад». Цикл розділений на більш дрібні легко створювані модулі. Кожен модуль проходить через фази визначення вимог, проектування, кодування, впровадження та тестування[2]. Процедура розробки по інкрементній моделі передбачає випуск на першому великому етапі продукту в базовій функціональності, а потім вже послідовне додавання нових функцій, так званих «інкрементів». Процес триває до тих пір, поки не буде створена повна система.

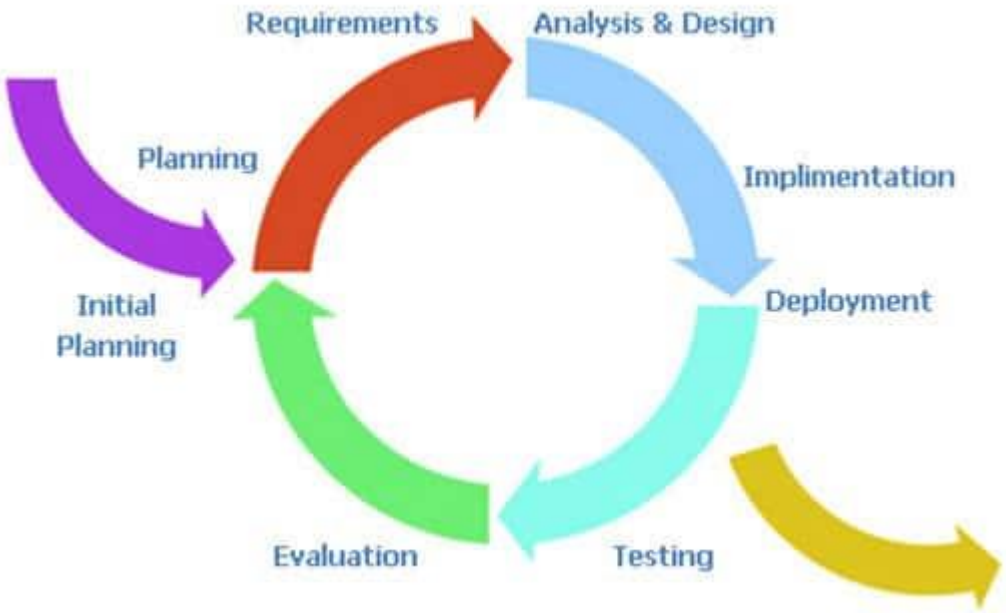


Рисунок 1.6 – Incremental Model

Коли використовувати інкрементальну модель. Коли основні вимоги до системи чітко визначені і зрозумілі (рис. 1.6); у той же час деякі деталі можуть доопрацьовуватися з плином часу; потрібний ранній висновок продукту на ринок; є кілька ризикових функцій або цілей.

«RAD Model» (rapid application development model або швидка розробка додатків).

RAD – модель – різновид інкрементної моделі. У RAD – моделі компоненти або функції розробляються декількома висококваліфікованими командами паралельно, ніби кілька міні – проектів[3]. Тимчасові рамки одного циклу жорстко обмежені. Створені модулі потім інтегруються в один робочий прототип. Це дозволяє дуже швидко надати клієнту для огляду щось робоче з метою отримання зворотного зв'язку і внесення змін.

Модель швидкої розробки додатків включає наступні фази:

– Бізнес – моделювання: визначення переліку інформаційних потоків між різними підрозділами;

– Моделювання даних: інформація, зібрана на попередньому етапі, використовується для визначення об'єктів і інших сутностей, необхідних для циркуляції інформації. Моделювання процесу: інформаційні потоки пов'язують об'єкти для досягнення цілей розробки;

– Збірка програми: використовуються кошти автоматичного складання для перетворення моделей системи автоматичного проектування в код;

– Тестування: тестуються нові компоненти і інтерфейси.

Коли використовується RAD – модель. Може використовуватися тільки при наявності висококваліфікованих і вузькоспеціалізованих архітекторів. Бюджет проекту великий, щоб оплатити цих фахівців разом з вартістю готових інструментів автоматизованого складання (рис. 1.7). RAD – модель може бути обрана при впевненому знанні цільового бізнесу і необхідності термінового виробництва системи протягом 2 – 3 місяців.

RAD – методологія, яка на перше місце ставить швидкість і зручність розробки. Одна з головних умов – використання мови швидкої розробки. Ця назва абстрактної мови програмування, за допомогою якого програміст здатний вирішувати завдання швидше, ніж з представниками третього покоління (C / C ++, Pascal або Fortran). Ось ще кілька пунктів концепції:

– Використання фокус – груп для збору вимог;

– Прототипування і призначене для користувача тестування конструкцій;

– Повторне використання програмних компонентів;

– Використання плану не включає переробку або дизайн наступної версії продукту;

– Проведення неформальних нарад за запитом однієї зі сторін.

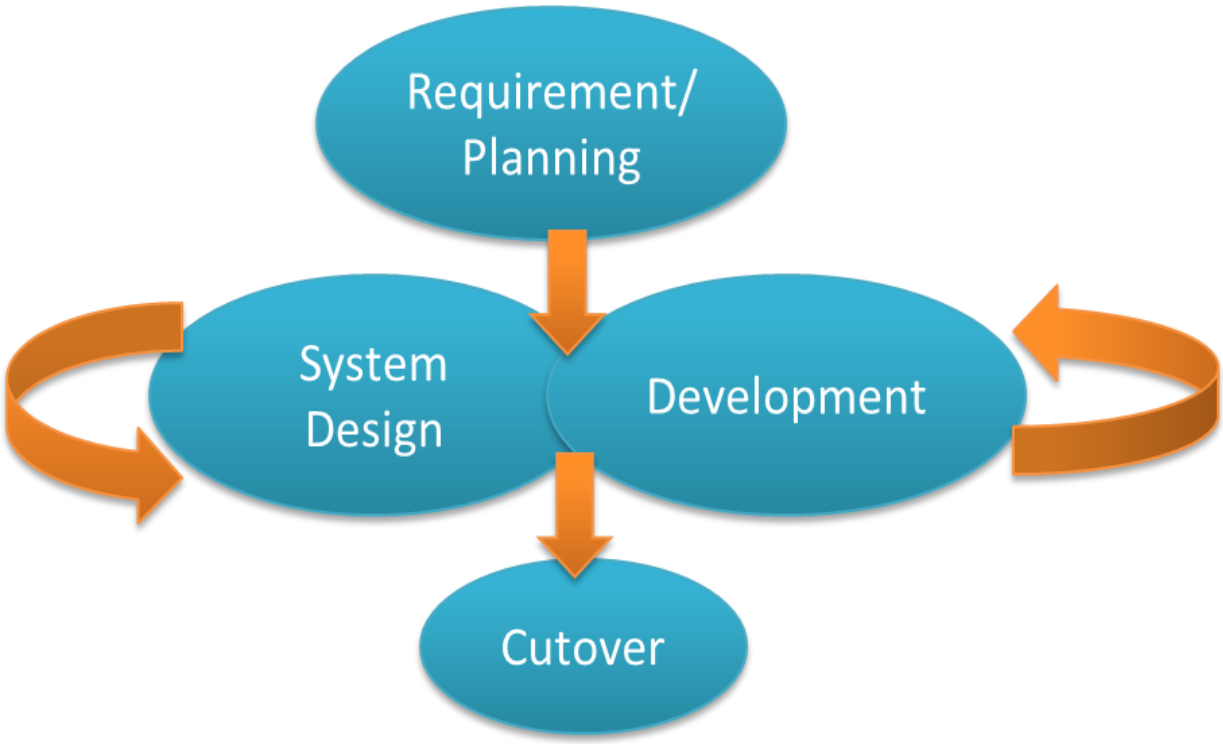


Рисунок 1.7 – RAD Model

«Agile Model» (гнучка методологія розробки).

У «гнучкії» методології розробки після кожної ітерації замовник може спостерігати результат і розуміти, задовольняє він його чи ні. Це одна з переваг гнучкої моделі. До її недоліків відносять те, що через відсутність конкретних формулювань результатів складно оцінити трудовитрати і вартість, необхідні на розробку. Екстремальне програмування (XP) є одним з найбільш відомих застосувань гнучкої моделі на практиці[3].

В основі такого типу – нетривалі щоденні зустрічі – «Scrum» і регулярно повторювані зборів (раз в тиждень, раз на два тижні або раз на місяць), які називаються «Sprint». На щоденних нарадах учасники команди обговорюють:

– звіт про виконану роботу з моменту останнього Scrum'a;
– список завдань, які співробітник повинен виконати до наступних зборів;
– труднощі, що виникли в ході роботи.

Методологія підходить для великих або націлених на тривалий життєвий цикл проектів, постійно адаптуються до умов ринку. Відповідно, в процесі реалізації вимоги змінюються. Варто згадати клас творчих людей, яким властиво

генерувати, видавати і випробувати нові ідеї щотижня або навіть щодня. Гнучка розробка найкраще підходить для цього психотипу керівників.


Рисунок 1.8 – Agile Model

Коли використовувати Agile. Коли потреби користувачів постійно змінюються в динамічному бізнесі; зміни на Agile реалізуються за меншу ціну через часті інкрементів; на відміну від моделі водоспаду, в гнучкій моделі для старту проекту достатньо лише невеликого планування (рис. 1.8).

Agile – метод гнучкої розробки програмного забезпечення, що передбачає велику кількість ітерацій. Документ Agile Manifesto описує 4 ідеї і 12 принципів гнучкого підходу, коротко його можна описати всього двома пунктами:

Неформальні відносини важливіше задокументованих. Тобто усні домовленості між співробітниками, між замовником і виконавцем важливіші ніж ті, що відображено в планах, договорах і технічному завданні. Інакше кажучи, клієнт завжди правий.

Працюючий продукт – головна оцінка прогресу. Важливі не інструменти, рішення, продуктивність і витонченість, а той факт, що всі заплановані можливості реалізовані. Незважаючи на недоліки, Agile стала фундаментальною концепцією для розробки ПО і знайшла відображення в інших методиках, мова про які піде далі.

«Iterative Model» (ітеративна модель).

Ітераційна модель життєвого циклу не вимагає для початку повної специфікації вимог. Замість цього, створення починається з реалізації частини функціоналу, що стає базою для визначення подальших вимог. Цей процес повторюється. Версія може бути неідеальна, головне, щоб вона працювала. Прикладом ітеративної розробки може служити розпізнавання голосу. Перші дослідження і підготовка наукового апарату почалися давно, на початку – в думках, потім – на папері. З кожною новою ітерацією якість розпізнавання поліпшувалася[3]. Проте, ідеальне розпізнавання ще не досягнуто, отже, завдання ще не вирішена повністю.

Коли оптимально використовувати ітеративну модель.

– Вимоги до кінцевої системі заздалегідь чітко визначені і зрозумілі.

– Основне завдання повинна бути визначена, але деталі реалізації можуть еволюціонувати з плином часу.

«Spiral Model» (спіральна модель).

«Спіральна модель» схожа на інкрементальну, але з акцентом на аналіз ризиків. Вона добре працює для вирішення критично важливих бізнес – задач, коли невдача несумісна з діяльністю компанії, в умовах випуску нових продуктових лінійок, при необхідності наукових досліджень і практичної апробації.

Спіральна модель передбачає 4 етапи для кожного витка:

– Планування;

– аналіз ризиків;

– конструювання;

– оцінка результату і при задовільній якості перехід до нового витка.

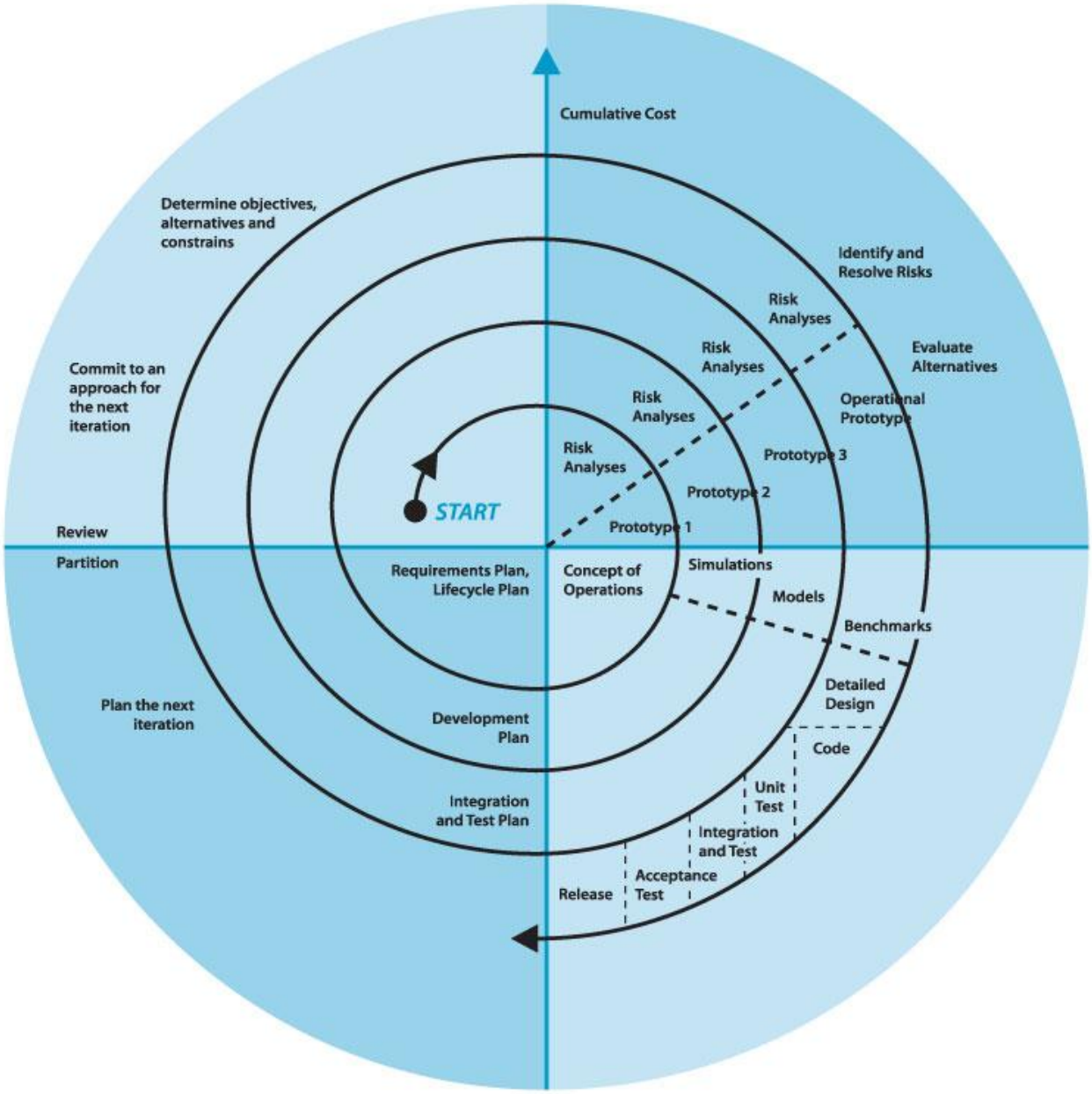


Рисунок 1.9 – Spiral Model

Модель спірального життєвого циклу – це складна організація життєвого циклу ПО, яка фокусується на ранньому виявленні та зменшенні проектних ризиків. Розробка починається в невеликому масштабі, вирішуються локальні завдання, оцінюються ризики та шляхи їх зменшення (рис. 1.9). Наступний крок охоплює більш комплексні завдання – наступний виток спіралі. Перевага підходу не в збільшенні швидкості розробки, а в зниженні рівня виникнення ризиків.

Успішність спірального методу залежить від сумлінного, уважного і компетентного управління, а розмір проекту не має принципового значення.

Ця модель не підійде для малих проектів, вона влучна для складних і дорогих, наприклад, таких, як розробка системи документообігу для банку, коли кожен наступний крок вимагає більшого аналізу для оцінки наслідків, ніж програмування.

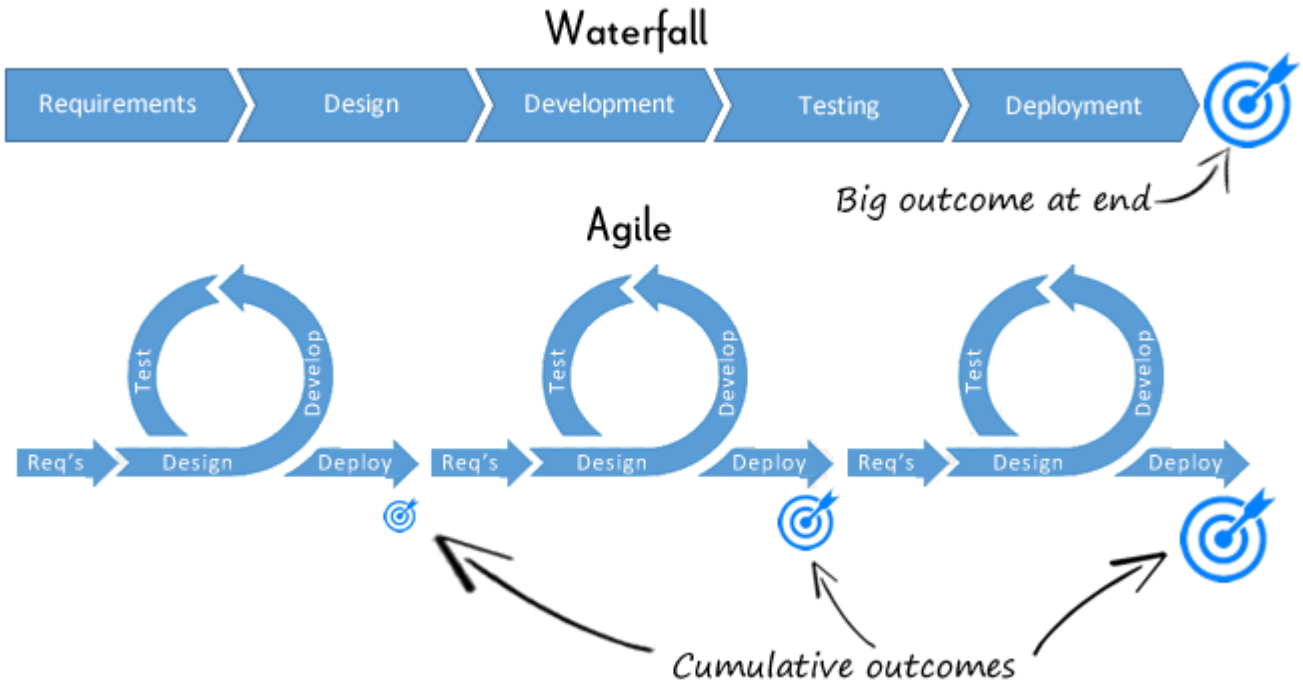


Рисунок 1.10 – Каскадна та гнучка методології

Kanban.

Kanban – система, побудована на візуалізації процесу виконання завдань команди. Основна ідея в цій системі зменшувати кількість завдань виконуються в даний момент (в колонці «in progress»). У скрам орієнтація команди на успішне виконання спринтів, в Канбан на першому місці завдання[4]. Гарний для проектів. які в стадії підтримки. де основний функціонал вже розроблений і залишилися мінімальні доопрацювання і багофіксінг (рис. 1.10). У канбані завдання здаються індивідуально. Завдання незалежно від інших завдань проходить по всім етапам на дошці і як тільки вона виконана її можна показати замовнику. Канбан дошка складається з колонок, кожна з яких це окремий процес розробки. На деякі стовпці

(наприклад, in progress) вводять обмеження по кількості завдань, які там можуть знаходитися. Це допомагає легко і швидко знаходити проблемні місця в розподілі завдань. На зображенні приклад самої просто такий дошки[4]. Кількість колонок і назви можуть змінюватися, назвемо найпоширеніші:

- To do – список завдань, які треба зробити;
- In progress – завдання над якими ведеться робота в даний момент;
- Code review – завдання, які зроблені і відправлені на рев'ю;
- In testing – завдання, готові до тестування;
- Done – зроблені завдання.

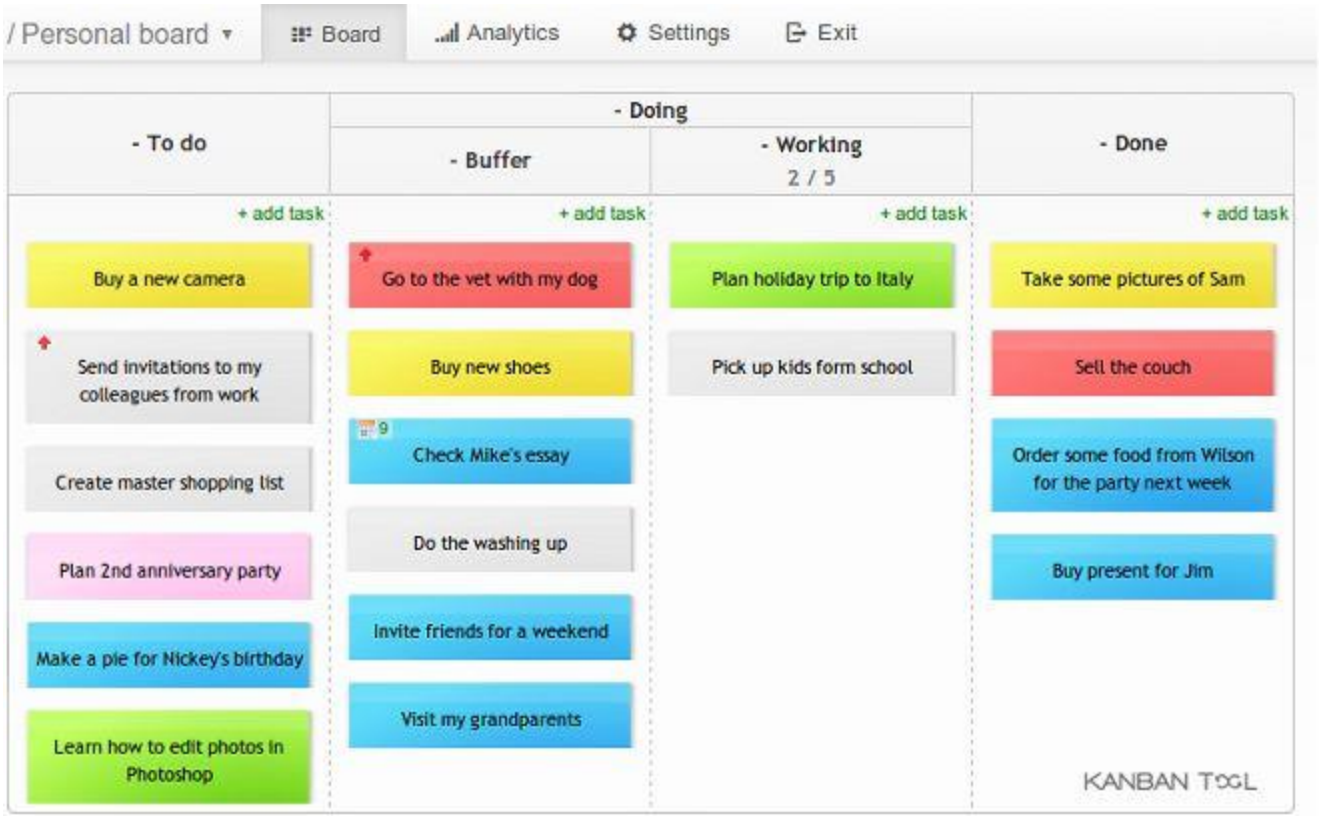


Рисунок 1.11 – Kanban методологія

Переваги:

- Простота використання;
- наочність. (Допомагає в знаходженні вузьких місць, спрощує розуміння);
- висока залученість команди в сам процес;
- висока гнучкість в розробці.

Недоліки:

- Нестабільний список завдань;
- складно застосовувати на довгострокових проектах;
- відсутність жорстких дедлайнів.

FDD.

FDD – процес для забезпечення масштабованості і повторюваності, при цьому заохочує творчість та інновації. Ось основні принципи: Розробка кожного великого проекту повинна мати системність. Процеси повинні бути простими і опрацьованими. Цінність і логічність процесу повинна бути ясна кожному члену команди. Перевага віддається коротким ітеративним циклам розробки. Це зменшує кількість помилок і дозволяє швидше нарощувати функціональність. FDD регламентує час, який повинен витрачатися на кожен з процесів. Організаційної діяльності в циклі повинна займати не більше 23 – 25%, в той час як на безпосередню розробку, складання і тестування функцій необхідно витрачати 75 – 77% часу.

JAD.

JAD – це методологія, націлена на максимальну зайнятість в розробці кінцевого користувача. Відбувається це за допомогою зустрічей і проведення спільних семінарів. JAD була придумана в 1970 – х роках співробітниками IBM і націлена на бізнес в цілому. Однак з часом ця концепція стала успішно застосовуватися і для розробки програмного забезпечення[4]. На відміну від підходу Waterfall, JAD призводить до скорочення часу розробки, більшої задоволеності клієнтів і економії коштів на вивченні ринку. З іншого боку, це вимагає великої клієнтської вибірки і необхідності розробників працювати не зі строгими вимогами ТЗ, а з мінливою думкою.

LD.

Ощадлива розробка ПО – ще одне відгалуження гнучкою методології, яка передбачає збереження високого морально – функціонального стану розробників. Це виражається в: Заохочення співробітників за успішну роботу. Зміні поточних завдань тільки в міру необхідності або на вимогу замовника. Суворому виконанні плану: все, що понад – вважається втратами часу і ресурсів. Впровадженні загальної концепції «Мислити широко, робити мало, помилятися швидко, вчитися стрімко».

XP.

Екстремальне програмування – можливість вести розробку в умовах постійно мінливих вимог. Ось кілька ознак:

– Гра в планування. На початку проекту є тільки приблизний план, після кожної ітерації його чіткість зростає;

– висока частота релізів. Нова версія продукту має незначні зміни в порівнянні з попередньою, але час на проходження митного кордону при цьому мінімальний;

– контакт з клієнтом. Для задоволення вимог кінцевої аудиторії необхідне оперативне реагування на зауваження та побажання;

– рефакторинг. Поліпшення якості коду без зменшення функціональності.

– стандарт виконання коду. Або застосовуються загальні правила, або розбіжності в оформленні не підлягають обговоренню і критиці.

– колективна відповідальність. Незважаючи на те, що кожен член команди виконує свою ділянку робіт, за код в цілому відповідає весь колектив.

DevOps методологія.

DevOps (від англ. Development & operations) – набір практик, націлених на активну взаємодію фахівців з розробки з фахівцями з інформаційно – технологічного обслуговування і взаємну інтеграцію їх робочих процесів один в одного[4]. Базується на ідеї про тісну взаємозалежність розробки та експлуатації програмного забезпечення і націлений на те, щоб допомагати організаціям швидше створювати і оновлювати програмні продукти і послуги. Динамічні зміни, що відбуваються в світі технологій, пред'являють до сучасних компаніям більш суворі вимоги до швидкості реакції, часу випуску релізів IT – систем і здатності фахівців різного профілю відкрито спілкуватися і плідно співпрацювати.

Методологія почала розвиватися з першої конференції DevOpsDays 2008 року. Суть DevOps полягає в об'єднанні спільноти розробників програмного забезпечення, тестування та обслуговування в єдине ціле. Development Operations охоплює кілька дисциплін. Це дуже практична область, яка вимагає розуміння технічних основ і культурних аспектів. Слово «DevOps» – це поєднання «development» і «operations» (розробка та операції). Гра слів відображає основну ідею, закладену в DevOps – співпраця між різними дисциплінами розробки. Своїм корінням DevOps йде в принципи «гнучкою» розробки. Маніфест Agile був

написаний в 2001 році декількома людьми. Вони хотіли поліпшити стан справ в розробці систем і знайти нові способи роботи в IT – індустрії.

Ось витяги з маніфесту Agile:

- Люди і взаємодія важливіше процесів та інструментів;
- працюючий продукт важливіше вичерпної документації;
- співпраця з замовником важливіше узгодження умов контракту;
- готовність до змін важливіше проходження попереднім планом;
- тобто, не заперечуючи важливості того, що справа, ми все – таки більше цінуємо те, що зліва.

DevOps відноситься до першого принципу: «Люди і взаємодія важливіше процесів та інструментів»[4]. Якщо ви працювали у великій організації, ви знаєте, що замість цього правила діє протилежний принцип. Стіни між відділами зводяться навіть в невеликих компаніях, де на перший погляд здається, що такі стіни сформувати неможливо.

Кому потрібна і не потрібна методологія.

Багато IT – експертів вважають, що DevOps принесе користь будь – якій організації, яка займається розробкою програмного забезпечення. Це справедливо навіть в тому випадку, якщо компанія є простим споживачем IT – сервісів і не розробляє власні програми. В цьому випадку впровадження DevOps – культури допоможе сконцентруватися на інноваціях.

Виняток становлять стартапи, але і тут все залежить від масштабів проекту. Якщо мета – запустити мінімально життєздатний продукт (minimum viable product, MVP), щоб протестувати нову ідею, то можна обійтися і без DevOps. Наприклад, засновник Groupon на початку роботи над сервісом сам вручну розміщував все пропозиції на сайті і збирав замовлення. Ніяких інструментів автоматизації він не використав.

Впроваджувати методологію і інструменти автоматизації має сенс тільки тоді, коли додаток починає набирати популярність. Це допоможе налагодити бізнес – процеси і прискорити вихід оновлень.

Концепція Development Operations.

DevOps підкреслює важливість взаємодії між людьми. Технології служать тригерами спілкування і допомагають зруйнувати бар'єри всередині організацій. Головне – правильне використання і вибір інструментів. Ось приклад: вибір систем для баг – репортів. Справа в тому, що розробники і QA використовують різні

системи для обробки завдань і багів. Різні інструменти створюють непотрібні тертя між командами і ще більше розділяють їх, коли колективи повинні зосередитися на спільній роботі. Сисадміни можуть використовувати третю систему для обробки запитів деплоя на сервери організації.

DevOps інженер відразу зрозуміє, що всі три системи потрібні для управління робочим процесом з подібними властивостями. Всі три команди повинні використовувати одну і ту ж систему (рис. 1.12). Її можна налаштувати під всі ролі. Головна перевага – менші витрати на обслуговування, оскільки три системи замінюються однією. Інше завдання Development Operations – автоматизація і безперервна доставка (CD)[4]. Автоматизація рутинних завдань залишає більше часу для взаємодії з людиною, звідки можна отримати більше користі.

Чи не є винятком і розробники вбудованих систем. Затребувана методологія безперервного постачання рішень висуває особливі умови до організації проектних процесів, причому тестування перестає бути заключним і ізольованим етапом всього циклу розробки (рис. 1.13).

Основною метою переходу на ідеологію DevOps є бажання поліпшити і прискорити обслуговування клієнтів. Скорочення витрат, збільшення автоматизації, поліпшення управління конфігурацією, виконання бізнес – запитів – це лише супутні завдання. При цьому різним типам організацій може знадобитися різна структура команд для ефективної взаємодії Dev і Ops. З появою в 2009 році методології DevOps, яка повинна була відповісти на зростаючу кількість питань до організації активного, безперервного і продуктивної взаємодії IT – фахівців в рамках проекту, компанії розділилися на лідерів і відстаючих в залежності від того, наскільки добре їм вдалося адаптувати процес тестування до декларованим принципам.

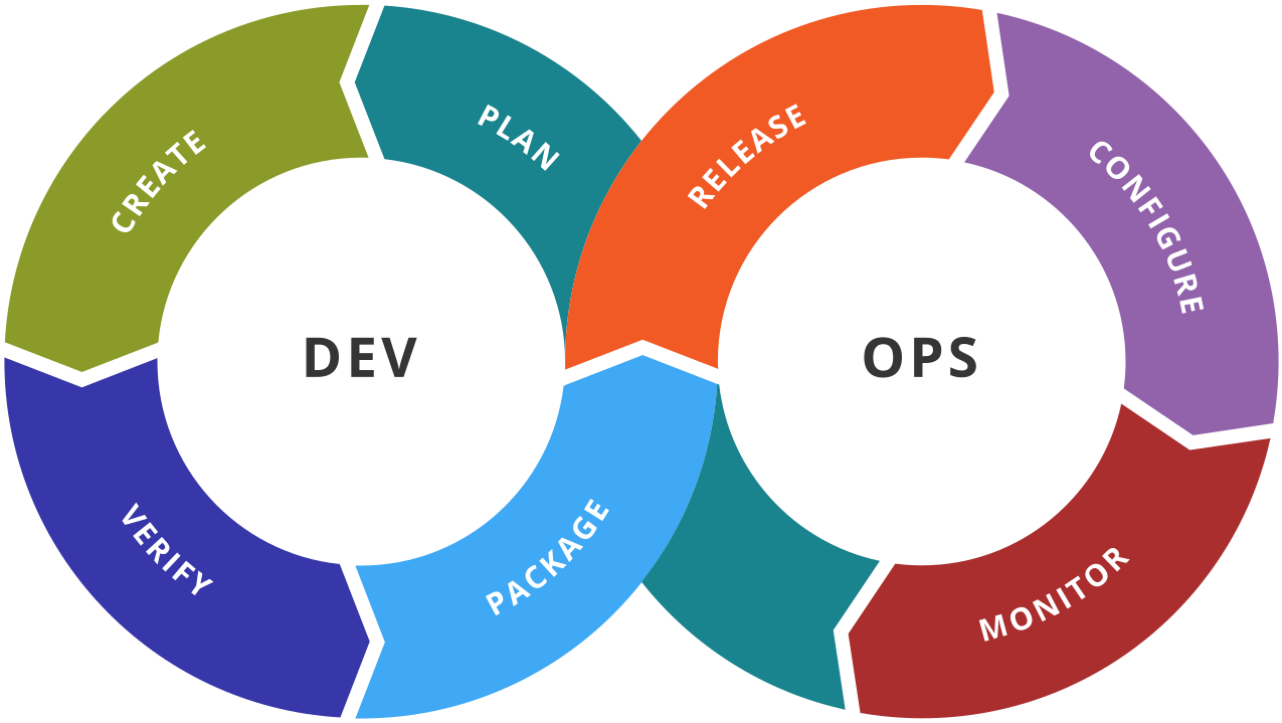


Рисунок 1.12 – Розробка програмного забезпечення по DevOps

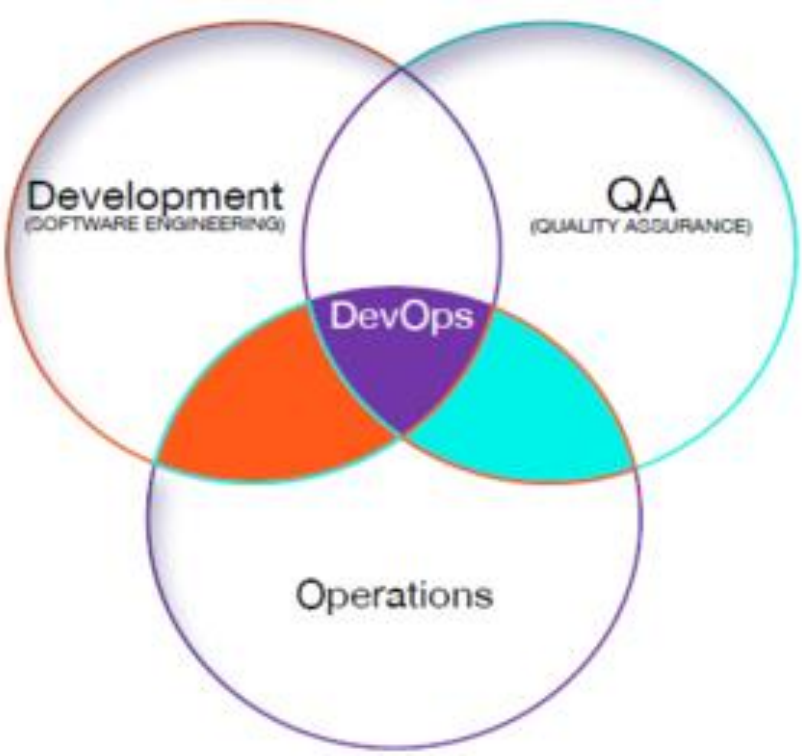


Рисунок 1.13 – DevOps концепція

Впровадження DevOps – непросте завдання для багатьох опитаних компаній: адже нова практика часто вимагає іншого способу мислення і фундаментального зсуву в корпоративній культурі. І все ж переваги, які DevOps і Agile обіцяють розробникам, тестувальникам і всій команді, набагато перевершують будь – які витрати.

Швидкість прийняття рішення.

Процеси в DevOps повинні бути швидкими. Потрібно розглядати час виходу на ринок у великій і зосереджуватися на завданнях в меншій перспективах. Таку ж лінію підтримує інженерний підхід безперервної доставки (Continuous Delivery). Багато ідей в DevOps і Continuous Delivery є різними назвами одних концепцій. Між двома поняттями немає розбіжностей. Вони – дві сторони однієї медалі. Інженери DevOps працюють над прискоренням, ефективністю і надійністю корпоративних процесів. Повторюваний ручна праця, який схильний до помилок, видаляється при першій можливості[4].

Але при роботі з реалізаціями Development Operations легко втратити мету. Нікому не принесе користь швидке виконання непотрібних завдань. Замість цього варто стежити за збільшенням вартості бізнесу. Наприклад, цінність налагодженої комунікації між різними ролями в організації очевидна. Власник товару може зацікавитися процесом розробки. У цій ситуації корисна можливість швидко і ефективно впроваджувати поступові поліпшення коду в тестову середу виконання. У тестових середовищах власники продуктів і QA можуть стежити за ходом розробки.

DevOps інженер.

Хороший DevOps інженер просуне компанію вперед, змінить прийняті правила і сценарії будь – якого існуючого бізнесу. Він забезпечить безперебійну роботу систем і постійний моніторинг всієї платформи для вирішення проблем при їх виникненні. DevOps інженер відповідає за оновлення коду і серверної інфраструктури після нових коммітов. А також за автоматизацію рутинних завдань при збільшенні часу складання і зростання компанії. Цим він звільняє програмістів, які можуть зосередитися на створенні логіки додатка.

Такий фахівець відповідає за:

- Використання широкого діапазону інструментів, технологій;
- навички програмування і скриптинга;
- управління серверами: моніторинг, операції, оновлення інфраструктури;

- навички автоматизації скриптами або спеціальним софтом;
- ефективне управління розподілом пам'яті і даних;
- спілкування і співпраця.

DevOps інструменти:

- Планування та оцінка – JIRA;
- контроль версій – Github і Git;
- підтримка і відображення злиття коду – SVN;
- сборка – Maven, Gradle;
- тестування – Selenium, JUNIT, Jenkins;
- деплой – Chef, Puppet і Ansible;
- операції – Salt, Red Hat Ansible;
- моніторинг – Nagios, Sensu і New Relic, Pagerduty, Cloudmapю

Критика DevOps. Хоча методологія допомагає організаціям швидше приймати рішення, що стосуються розробки додатків, скорочує кількість помилок в ПЗ і заохочує співробітників вчитися новому, у неї є і критики[4]. Є думка, що програмісти не повинні розбиратися в деталях роботи системних адміністраторів. Нібито DevOps призводить до того, що в компанії замість фахівців з розробки або адміністрування з'являються люди, котрі розуміються в усьому, але поверхово.

Також вважається, що DevOps не працює при поганому менеджменті. Якщо у команд розробників і адміністраторів немає загальних цілей, в цьому винні менеджери, які організовують взаємодію між командами. Щоб вирішити цю проблему, потрібна не нова методологія, а система оцінки менеджерів на основі відгуків від підлеглих.

DevOps – відносно нова концепція, і у неї все ще немає чіткого, загальноприйнятого визначення. По суті, це набір практик, націлених на активну взаємодію та інтеграцію фахівців з розробки, тестування і супроводу. Використання DevOps забезпечує стабільне, швидке і надійне розгортання ПО, в тому числі і завдяки безперервності тестування, що дозволяє уникнути затримок і проблем з якістю, властивих для класичної моделі проектної розробки.

Популярність методології значно зросла в останні роки. За даними RightScale 2016 State of the Cloud Report: DevOps Trends, прийняття DevOps збільшилася з 66% в 2017 році до 74% в 2018 році, а серед великих організацій прийняття методології ще вище – 81%.

Теперь ми визначилися з поняттям DevOps, що воно означає, для чого використовується. Далі ми перейдемо до опису CI/CD концепцій.

На закінчення про методологію розробки ПО. На мою думку, грунтовно розбиратися в методології розробки ПО потрібно людям, які займають управлінські посади або прагнуть до них, але розуміти хоча б основи бажано всім. Це невід'ємна частина процесу розробки і використовується не тільки в IT – сфері.

У сучасній практиці моделі розробки програмного забезпечення багатоваріантні. Немає єдино вірної для всіх проектів, стартових умов і моделей оплати. Навіть така улюблена всіма нами Agile не може застосовуватися повсюдно через неготовність деяких замовників або неможливості гнучкого фінансування. Методології частково перетинаються в засобах і частково схожі один на одного. Деякі інші концепції використовувалися лише для пропаганди власних компіляторів і не внесли в практику нічого нового.

## 1.3 CI/CD концепція

CI (Безперервна інтеграція) – це практика, яка допомагає розробникам доставляти краще програмне забезпечення більш надійним та передбачуваним способом. Цей підрозділ стосується проблем, з якими стикаються розробники під час написання, тестування та доставки програмного забезпечення кінцевим користувачам. Досліджуючи безперервну інтеграцію, буде розкрито, як можна подолати ці проблеми[5].

Безперервна інтеграція, безперервне розгортання і безперервна доставка – немов вектори, напрямок яких збігається, а модуль відрізняється. Мета всіх трьох прийомів однакова: підвищити надійність розробки і релізів ПО, а також прискорити розробку та релізи. Основна відмінність між трьома підходами – обсяг автоматизації при кожному з них. Новачки часто плутають ці феномени, оскільки не здогадуються, що вони не взаємовиключні, а входять один в одного.

CI/CD – платформи, на базі яких реалізується концепція, підтримують виконання регулярної автоматизованої збірки проектів для оперативного виявлення дефектів і рішення інтеграційних проблем. При стандартному підході (каскадна розробка ПО або водоспад – методика), де розробники незалежно трудяться над різними приватними системами, стадійна інтеграція є заключною, і при відкритій появі помилок може бути непередбачуванно затримана працівником.

Перехід до безперервної інтеграції дозволяє знизити трудову роботу і зробити її більш передбачуваною для щасливого раннього і постійного виявення і усунення помилок і протидії.

По своїй суті концепція CI/CD реалізує ідеологічну розробку розробок та експлуатацію ПЗ (Development & Operations, DevOps) та відповідає основним принципам Agile у частинах рекомендацій за допомогою автоматичного тестування для побудови відлагоджених робочих версій продукту.

Основна стихія переходу на CI/CD надається в адаптаційному підході, де на першому місці наступають процеси, а технології – тільки на другому. Необхідно будувати нові процеси, визначати нові ролі людей, шукати точки інтеграції вже існуючих і нових процесів. Розміщення акцентів з продуктом і «залізом» на людей і організаційні аспекти роботи для багатьох людей є серьозним викликом. Другий момент – в CI/CD відповідальність розробника – щоб результат був найкращим. Тепер він не просто пише абстрактний код за ТЗ, а ещё і його перевіряє своїми силами[5].

Спочатку буде розглянуто джерело проблеми, яка полягає в циклі розробки програмного забезпечення. Далі буде висвітлено деякі конфлікти змін, які можуть відбутися під час цього процесу, а потім буде пояснення, як безперервна інтеграція вирішує ці проблеми.

Джерело проблеми.

Давайте подивимось, як виглядає традиційний цикл розробки програмного забезпечення. Кожен розробник отримує копію коду з центрального сховища. Вихідною точкою зазвичай є остання стабільна версія програми. Усі розробники починають з однієї і тієї ж відправної точки і працюють над додаванням нової функції або виправленням помилки. Кожен розробник досягає успіху, працюючи самостійно або в команді. Вони додають або змінюють класи, методи та функції, формуючи код відповідно до своїх потреб, і врешті вони виконують завдання, яке їм було призначено. Тим часом інші розробники та команди продовжують працювати над власними завданнями, змінюючи код або додаючи новий код, вирішуючи завдання, які їм були призначені.

Якщо ми зробимо крок назад і подивимось на велику картину, тобто весь проект, ми можемо побачити, що всі розробники, які працюють над проектом, змінюють контекст для інших розробників, коли вони працюють над вихідним кодом. Коли команди закінчують свої завдання, вони копіюють свій код у центральний сховище. Є два сценарії, які можуть мати місце в цей момент.

Код у центральному сховищі не змінюється.

Код такий самий, як і початковий примірник. Якщо це так, то справа проста, оскільки система незмінна. Усі ідеї, які ми мали щодо системи, досі стоять. Це завжди так, якщо ви єдиний розробник, який працює над програмою, і якщо ви закінчили свою роботу перед іншими членами вашої команди. Так чи інакше, справи виглядають добре для вас. Створену вами та протестовану систему можна доставити користувачам без додаткових змін.

Код у центральному сховищі змінився.

Другий сценарій полягає в тому, що додаток, над яким ви працювали, змінився, і ви виявите це в момент, коли ви намагаєтесь скопіювати свій код у центральний сховище. Зміни коду можуть суперечити тим, які ви внесли. Якщо виникають конфлікти, вам потрібно їх вирішити, щоб мати змогу успішно доставити свій код користувачам. У цьому випадку все може ускладнитися.

Конфлік змінень.

Існує кілька типів конфліктів змін, які можуть виникати при інтеграції коду. Ось кілька найпоширеніших[5]. Почнемо з найпростіших сценаріїв і поступово вивчимо складніші.

Деталі реалізації змінилися – ви відновили метод, але це зробив розробник, який вже інтегрував їх код у центральний сховище. Поведінка методу однакова у всіх трьох реалізаціях. Вам потрібно буде вибрати версію, яка залишиться, та видалити інші реалізації. Можна навіть придумати четверту реалізацію. Це простий тип конфлікту, який зазвичай можна вирішити протягом декількох хвилин.

API, на які ви покладалися, змінилися. Наприклад, змінилася поведінка певного методу. Це може вплинути на ваш код різними способами – від незначних змін, які вам можуть знадобитися внести, до великих структурних змін. Срібної кулі в таких випадках немає. Вам потрібно буде уважно вивчити зміни та внести всі виправлення.

Вся підсистема програми поводиться по – іншому – у таких випадках ви майже напевно зіткнетесь з частковим, якщо не повним перезаписом свого рішення. Якщо це так, вам, ймовірно, доведеться поговорити з усіма розробниками, які працюють над додатком, оскільки така суттєва зміна не повинна відбутися, не повідомляючи про це іншу команду.

Рішення: інтегруватись постійно. Вирішення проблеми управління великою кількістю змін великих подій інтеграції концептуально просте. Потрібно розділити

ці великі інтеграційні події на значно менші інтеграційні події. Таким чином, розробникам потрібно боротися зі значно меншою кількістю змін, які легше зрозуміти та керувати ними. Щоб події інтеграції були невеликими та легко керованими, потрібно, щоб вони траплялися часто[5]. Пару разів на день ідеально. Практику робити невеликі інтеграції часто називають постійною інтеграцією (рис. 1.13).

Ідея проста, але в той же час часто представляється неможливим втілити в життя. Це тому, що зміна процесу вимагає від нас змінити деякі власні звички, а змінити звички важко.

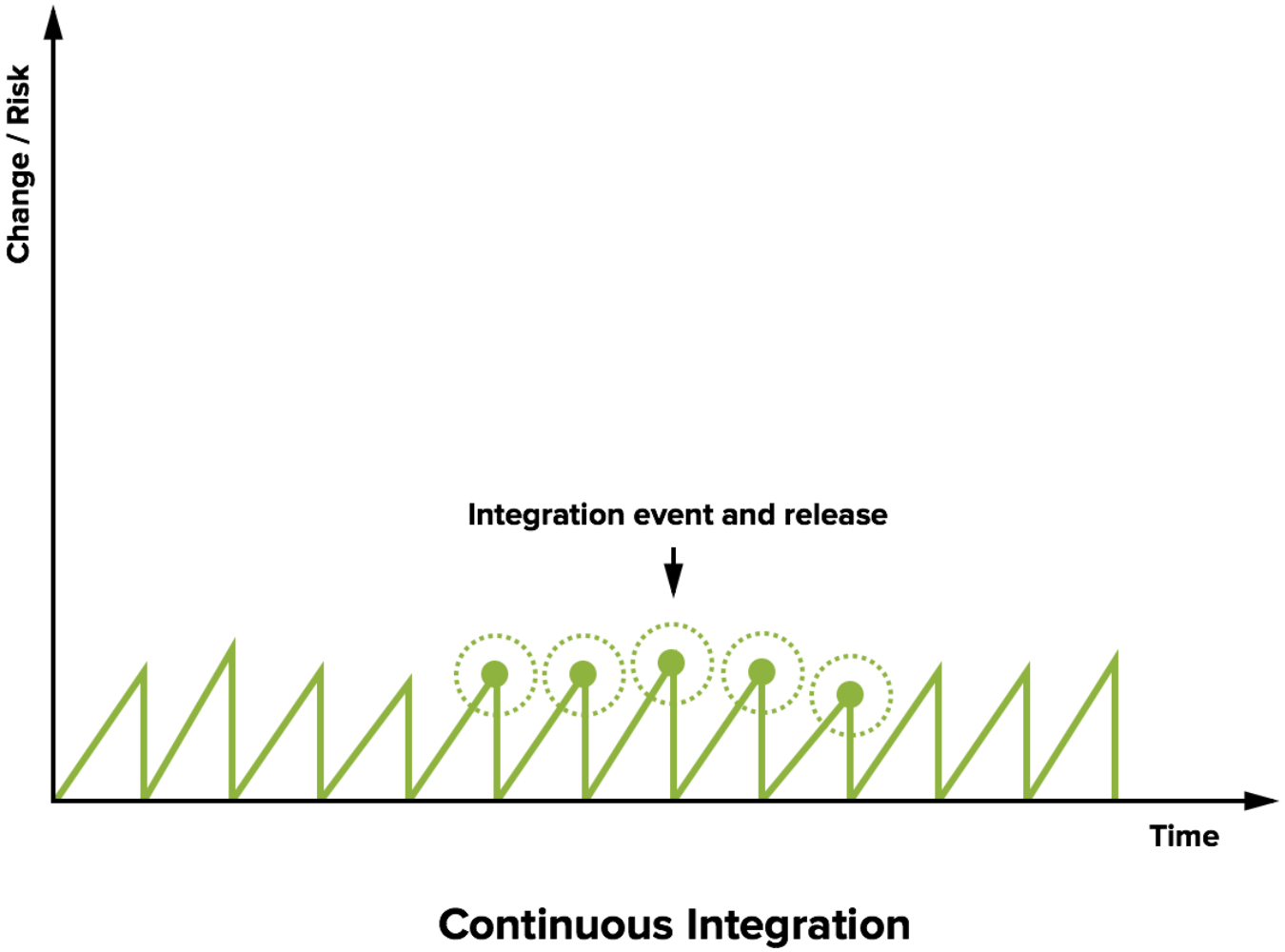


Рисунок 1.13 – Безперервна інтеграція

Щоб уникнути описаних раніше проблем, розробникам потрібно щодня або навіть пару разів на день інтегрувати частково повну роботу назад у основне

сховище. Для цього їм спочатку потрібно ввести всі зміни, додані до основного сховища під час роботи над кодом. Вони також повинні переконатися, що їх код запрацює після його інтеграції в основний сховище. Єдиний спосіб забезпечити це – протестувати кожну функцію програми. Перше, що виникає на увазі, коли ми починаємо розглядати постійну інтеграцію, це те, що розробникам потрібно було б витрачати половину свого часу щодня на тестування коду, щоб не порушити код у головному сховищі для всіх інших[5].

Ось чому необхідною умовою постійної інтеграції є автоматизований тестовий набір. Автоматизовані тести знімають з розробників тягар ручного, повторюваного та схильного до помилок процесу тестування. Вони також роблять весь процес тестування набагато швидшим. Комп'ютер може замінити години ручного тестування за кілька хвилин автоматизованого тестування. Розробка, орієнтована на поведінку та тест, – це методи, які допомагають розробникам писати чистий, доступний код під час написання тестів одночасно.

Тести мають сенс лише в тому випадку, якщо вони виконуються щоразу, коли вихідний код змінюється без винятку. Служба безперервної інтеграції, така як Semaphore, – це інструмент, який може автоматизувати цей процес, контролюючи центральне сховище коду та виконуючи тести на кожну зміну вихідного коду. Крім запущених тестів, вони також збирають результати тестів і передають ці результати всій команді, яка працює над проектом. Результат постійної інтеграції настільки важливий, що для багатьох команд є правило припинити роботу над своїм поточним завданням, якщо версія в центральному сховищі порушена. Вони приєднуються до команди, яка працює над виправленням коду, поки знову не пройдуть тести. Роль служби безперервної інтеграції полягає в поліпшенні зв’язку між розробниками, повідомляючи про статус вихідного коду проекту.

В CI/CD важливий швидкий цикл зворотного зв'язку, що дозволяє майже миттєво визначити, наскільки якісними є зміни в коді і функціональності продукту. При waterfall підході можна швидко вносити зміни в код, але без постійних перевірок виявлення багів зажадає набагато більше часу і може статися вже після початку комерційної експлуатації, причому таких «відкладених» багів в одному релізі може виявитися трохи.

Інструменти CI допомагають швидко відповісти на питання про появу дефектів для кожного чоловіка, що гарантує наявність раннего виявлення та помилки помилок. CI/CD – платформа допомагає не тільки оперативно тестувати, а й виводити нову функціональність для конечного користувача таким чином, щоб у

випадку виявлення помилок завжди існувала можливість або швидко встигати, або «скасовувати» ітераційну розв’язку на шаг назад.

Головні плюси CI/CD:

– Швидкість виведення нової функціональності від запиту клієнта до запуску в експлуатацію. CI/CD дозволяє запускати оновлення за лічені дні або тижні в порівнянні з цілим календарним роком при класичному waterfall підході. Нові сервіси – нові конкурентні бізнес – переваги. З'являється можливість не просто відтворювати функціональність рішень конкурентів, але і значно випереджати їх у розробці та впровадженні нових інструментів;

– Можливість вибору оптимального варіанту за рахунок оперативного тестування і більшого числа ітерацій. Відмовившись від роботи над безперспективними рішеннями, ви заощадите ресурси компанії;

– Якість підсумкового результату вище: автотестування охоплює всі аспекти продукту, що важко буде реалізувати при стандартному релізний підході. Всі помилки і тонкі місця виявляються і видаляються ще на ранніх етапах розробки.

Головні мінуси CI/CD:

– Спокуса перевести на Agile, DevOps і CI/CD відразу все, що пов'язано з корпоративними IT – системами, включаючи core – рівень, без придбання первинного досвіду[5]. Це може серйозно порушити роботу компанії, особливо при поганій організації переходу на нову методологію.

– Підтримка належного рівня координації між CI і CD. Швидкі та якісні результати від застосування методики можливі тільки після тривалої і ретельної настройки взаємодії між командами DevOps, інженерами, scrum – експертами і керівництвом компанії. Найскладніше в CI/CD – людський фактор, налагодження здорової командної роботи, яку запрограмувати і автоматизувати неможливо.

Впровадження безперервної інтеграції та автоматизованого тестування в процес розробки змінює спосіб розробки програмного забезпечення від початку. Це вимагає зусиль усіх членів команди та культурних зрушень в організації. Великі зміни в робочому процесі не просто швидко зняти. Зміни потрібно вводити поступово, і всі члени команди та зацікавлені сторони повинні бути в курсі цієї ідеї.

# 2 АНАЛІЗ ТА ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА ІНСТРУМЕНТІВ АВТОМАТИЗОВАНОГО РОЗГОРТАННЯ ІНФРАСТРУКТУРИ ВЕБ – ПРОЕКТУ

Оркестрація – це автоматизована конфігурація, координація та управління комп'ютерними системами та програмним забезпеченням.

Оркестрація часто обговорюється в контексті сервісно орієнтованої архітектури, віртуалізації, забезпечення, конвергентної інфраструктури та тем динамічного центру обробки даних. Оркестрація в цьому сенсі – це узгодження бізнес – запиту з додатками, даними та інфраструктурою. Основна відмінність робочого процесу "автоматизації" від "оркестрації" (в контексті хмарних обчислень) полягає в тому, що робочі процеси обробляються і завершуються як процеси в межах одного домену для цілей автоматизації, тоді як оркестрація включає в себе робочий процес і забезпечує спрямовану дію на більші цілі та завдання[6]. У цьому контексті та з загальною метою досягти конкретних цілей та завдань (описаних через параметри якості обслуговування), наприклад, відповідати цілям ефективності програми, використовуючи мінімізовані витрати та максимізувати ефективність програми в межах бюджетних обмежень.

У цьому розділі основна увага буде сфокусована на двух гігантах у розрізі порівняння інструментів оркестрації ресурсів веб – проекту. Також, для того щоб продемонструвати можливості та функції роботи даних інструментів, буде обрано хмарний провайдер AWS.

Більшості сучасних підприємств цікаво слово "автоматизація" і все з ним пов'язане. Автоматизовані процеси все більше і більше витісняють людську працю зі сфери бізнесу і, здається, через кілька років керувати всім будуть програми і машини. Але, не будемо забігати наперед і для початку розберемося.

Процес автоматизації передбачає нівелізацію людської праці (або значне його приборкання) за допомогою програм і техніки. За фактом, процеси, що виконуються людиною, замінюються процесами, виконуваними технікою. Це нововведення збільшує продуктивність, скорочує час виконання завдання і, в більшості випадків, значно покращує якість результатів.

Автоматизація – хороший спосіб підвищення прибутковості підприємства. У розрізі бізнесу, можна сказати що є безліч процесів, щоб знайти та

проконфігурувати потрібний вам, треба перерити усі, наткнутися на підводні камні і таке інше. Це являється марною тратою грошей. Так ось автоматизація виконує цей пошук за вас. Як висновок – глобальна економія людських ресурсів та ресурсів бізнесу.

І все ж, незважаючи на вищесказане, більшість все ще сумнівається в практичності і необхідності автоматизації бізнес – процесів.

Плюси автоматизації:

- Підвищення продуктивності;
- поліпшення якості;
- економія витрат часу;
- напрямок вивільненого часу на більш важливі завдання;
- збільшення прибутку.

Деякі мінуси:

– Складність впровадження – для початку впровадження автоматизації потрібно змінити поточний стан речей, через що можуть виникнути деякі складності;

– несприйняття (відкритий протест) колективом – іноді, людям важко сприймати щось нове, особливо коли вони звикли працювати по старому і їх цілком все влаштовувало.

Автоматизація бізнесу – процес складний і тривалий, але виразно стоїть. Адже оптимізувавши роботу підприємства, можна застрахувати себе і свій бізнес від багатьох проблем, таких як: брак контролю або його надлишок, несвоєчасна звітність, порушення дисципліни і інших, які глобально впливають на роботу і прибуток.

Управління конфігураціями є важливою частиною методології DevOps, і інструменти на зразок Ansible, Chef, Puppet або SaltStack складають основу сучасних систем розробки програмного забезпечення. Terraform – приклад системи наступного покоління для створення, управління і конфігурації хмарної інфраструктури[6].

Отже, поговоримо про утиліту для так званого "оркестрування" хмарної інфраструктури, що дозволяє будь – якому оператору працювати з будь – яким публічним або приватним провайдером послуг хмарного хостингу. Плюс до цього за допомогою такого інструменту можна легко створювати хмарну інфраструктуру як код (Infrastructure as Code), управляти нею і покращувати надалі. Будучи

частиною екосистеми Hashicorp, що включає також Vagrant, Packer, Consul, Vault і Nomad, Terraform дозволяє розмістити будь – який додаток, написаний на будь – якій мові на будь – яку інфраструктуру.

## 2.1 Terraform

Terraform – інструмент, що дозволяє розробляти, змінювати і версіонувати інфраструктуру сервісу в форматі коду. Інфраструктура описується в конфігураційних файлах і може застосовуватися як до працюючій системі, так і до абсолютно нової. За своїм форматом, розробка terraform конфігурації нагадує опис інфраструктурних компонентів, необхідних для роботи вашого додатка. В цілому формат опису інфраструктури досить декларативний. Для нього використовується спеціально розроблена мова програмування HashiCorp Configuration Language (HCL). Але, можна використовувати і просто JSON.

Лише кілька з ряду переваг Terraform:

- Оркестрування, а не просто конфігурація інфраструктури;
- побудова незмінної інфраструктури;
- декларативний, а не процедурний код;
- архітектура, що працює на стороні клієнта.

Всі згадані раніше утиліти створювалися для спрощення конфігурації серверів, тобто їх головна мета полягає в установці і управлінні ПО на вже існуючих серверах. Terraform ж сконцентрований на завданнях по створенню серверів з нуля, залишаючи роботу по розміщенню контейнерів з ПО платформам типу Docker або Packer. Коли вся інфраструктура хмарної екосистеми працює як код і всі параметри записані в декларативні файли конфігурації, фахівці команди можуть працювати з ними і змінювати ці файли, як і будь – який інший код[6].

При використанні Chef, Salt, Puppet, або Ansible, будь – яке оновлення ПЗ потрібно проводити там, де воно встановлено. Таким чином, з часом кожен сервер набуває свою унікальну конфігурацію і історію оновлень ПО. Це призводить до так званого "конфігураційному дрифту", коли незначні відмінності в вживаному в хмарної екосистемі ПО призводять до появи потенційних точок доступу для хакерів і уразливості системи.

Terraform працює за концепцією незмінної інфраструктури, де кожна зміна будь – якого компонента (оновлення ПЗ, видалення або додавання компонентів, і

т.д.) призводить до створення окремого стану інфраструктури, тобто до побудови нової системи і видалення попередньої конфігурації. Це означає, що процес оновлення ПЗ йде легко і швидко по всій системі відразу і захищений від можливих помилок. У той же час, повернення до будь – якої попередньої конфігурації системи працює не складніше, ніж вибір необхідної конфігурації зі списку і створення нового оточення за необхідними параметрами.

При використанні Chef або Ansible ми змушені писати покрокові процедурні інструкції по досягненню необхідного стану системи. Навпаки, Terraform, Salt або Puppet воліють відписувати кінцеві стану системи, залишаючи конфігурацію на розсуд утиліти. Це просто, тому що досить обмежена бібліотека шаблонів коду може задовольнити будь – які запити по конфігурації інфраструктури, а вбудовані примітиви дозволяють писати досить складний за рівнем впливу, але дуже легко читаємий модульний код.

Застосовуючи процедурний код, необхідно пам'ятати про всі поточні в системі процеси і недавні події, щоб створити просту і однозначну інструкцію. При використанні Terraform, просто вказується, які зміни потрібно зробити з поточним станом системи, що дозволяє обходитися досить компактною і дуже простою бібліотекою шаблонів коду.

Terraform використає API, що надаються постачальником хмарного хостингу. З їх допомогою утиліта будує інфраструктуру, що означає відхід від зайвих перевірок безпеки, відсутність необхідності в роботі окремого сервера для управління конфігураціями і виділення ресурсів на роботу численних програм – агентів. Ansible досягає східного результату шляхом роботи через SSH[6]. але можливості даного методу досить обмежені. Завдяки тому, що Terraform працює через API, цей інструмент надає буквально безмежну свободу дій і можливих варіантів конфігурацій. Такий підхід набагато краще з точки зору безпеки, надійності роботи і загальної простоти використання системи.

Terraform робить тільки одну задачу. І робить її добре. Завдання це – створити, зібрати і налаштувати ресурси. Будь – які ресурси, які можна описати у вигляді набору властивостей, зрозумілих провайдеру цих самих ресурсів. В першу чергу мова йде про ресурси наших улюблених обчислювальних хмар. AWS і Azure. І безлічі інших. Terraform – це єдиний спосіб опису ресурсів (рис. 2.1). Можливо, відразу в декількох хмарах. Але от самі типи ресурсів, їх імена і атрибути, будуть в кожній хмарі свої. І в них доведеться розібратися.

Зате опис ресурсів буде в текстовому вигляді. У файлах формату HCL (це щось середнє між JSON і YAML). У вашій системі контролю версій. Це і називається Infrastructure as Code.

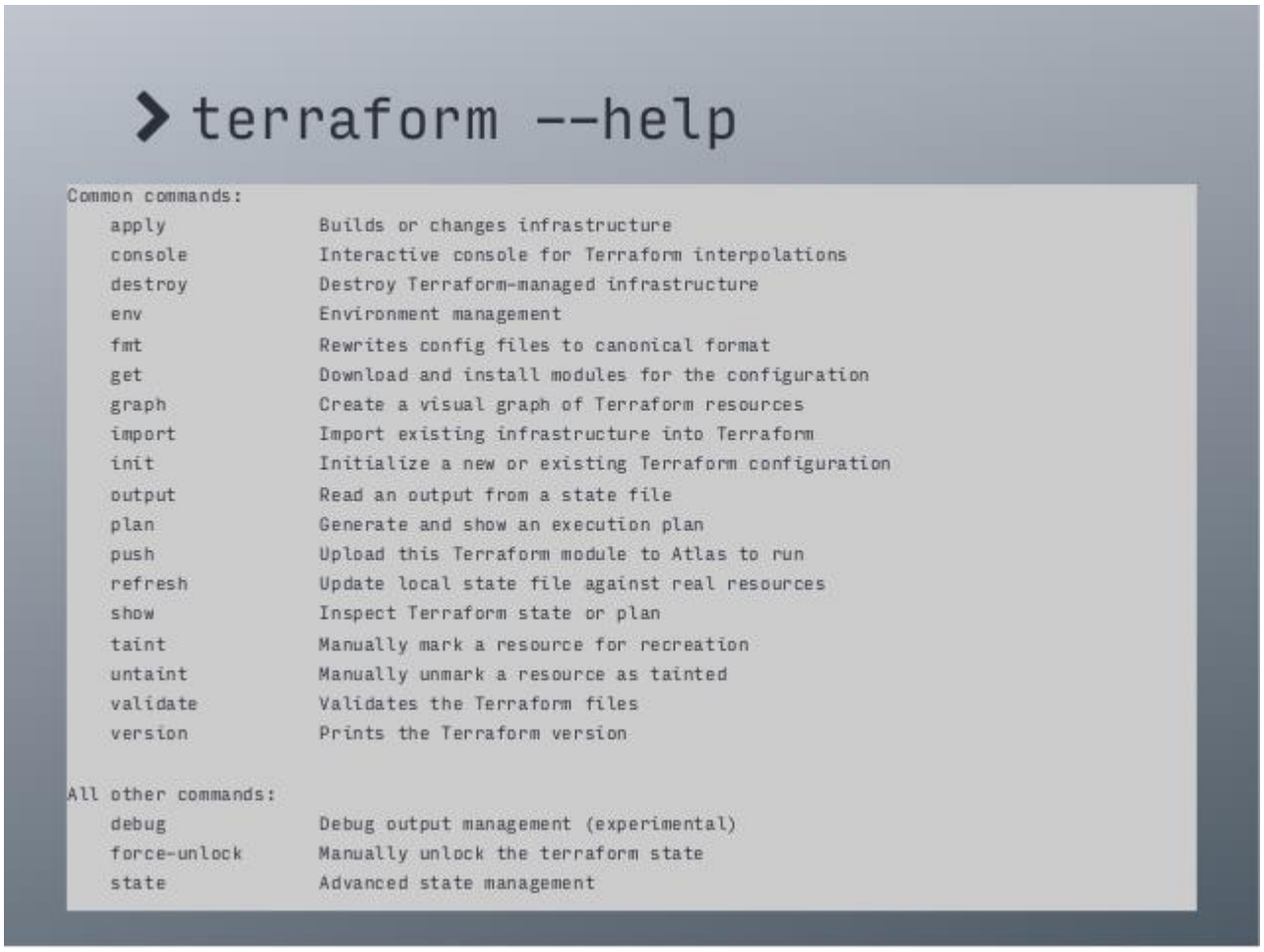


Рисунок 2.1 – Набір команд Terraform

Розберемося у термінології.

Конфігурація. Configuration. Це набір * .tf файлів в поточному каталозі, тобто де ви запустили terraform. У цих файлах ви описуєте те, що вам потрібно створити і підтримувати. Не важливо, як ви ці файли назвете[6]. Не важливо, в якому порядку в них все опишете. Є тільки невелику угода що повинні бути файли: main.tf – типу заголовні визначення, variables.tf – всі вхідні змінні, outputs.tf – всі вихідні змінні даної конфігурації.

Провайдери. Providers. Модулі, окремі бінарники, які визначають весь інший набір того, що можна задати в конфігурації (рис. 2.2). Вони викачуються при ініціалізації конфігурації. Ініціалізація робиться командою terraform init.
Ось як може виглядати опис провайдера для AWS:

```
provider "aws" {
    version = "~> 1.20"
    region = "${var.aws_region}"
    profile = "${var.aws_profile}"
}
```

Рисунок 2.2 – Опис провайдера AWS

Конструкції з фігурними дужками – це interpolations. В даному випадку беруться значення двох вхідних змінних.

Змінні. Variables. Рівні введення для конфігурації (рис. 2.3). Їх треба явно описати:

```
variable "aws_profile" {}
variable "aws_region" {}
```

Рисунок 2.3 – Змінні

Змінні бувають трьох типів: рядок, список, map. В даному випадку тип не вказано, і мається на увазі рядок. З підтримкою вкладених типів, типу списку maps, все настільки невизначено, що краще вважати, що вкладених типів немає. Значення для змінних, як і належно, можна передавати купою різних способів. Через змінні оточення типу TF_VAR_aws_profile. Через параметри командного рядка типу – var 'aws_profile = "terraform"'. Через файли * .tfvars, які потрібно явно передавати в командному рядку: –var–file ../test.tfvars. Через файли * .auto.tfvars в поточному каталозі, які завантажуються автоматично[6]. Ясна річ, є правила перевизначення змінних, а мари навіть об'єднуються.

Є ще вихідні змінні. Це деякі значення, які є виходом вашої конфігурації. Ці значення потім можна легко отримати командою terraform output.

Ресурси. Resources. Те, заради чого все це і затівається. Ресурси, підтримувані провайдером, які потрібно створити, або модифікувати, або видалити (рис. 2.4).

У кожного ресурсу є тип, який залежить від провайдера, ім'я, в контексті даної конфігурації, і набір аргументів, які потрібно передати ресурсу. Також ресурс має деякі атрибути, це якісь, можливо, обчислювані значення, які стають відомі, коли ресурс створений. Ось як може виглядати, наприклад, створення S3 корзини:

```
resource "aws_s3_bucket" "frontend_bucket" {
    bucket = "${var.environment}-frontend"
    acl = "public-read"
    website {
        index_document = "index.html"
        error_document = "index.html"
    }
    tags {
        Name = "${var.deployment} Frontend"
        Environment = "${var.environment}"
        Deployment = "${var.deployment}"
    }
}
```

Рисунок 2.4 – Опис ресурсу S3 корзини

Найчастіше ресурси один в один відповідають тим ресурсам, що можна створити через API або в консолі управління хмарою. Але іноді зустрічаються і «псевдоресурси», які являють собою команди модифікації «справжніх» ресурсів. Наприклад, Security Groups в AWS[6]. Часто потрібно, щоб одна Security Group посилалася на іншу, а та посилалася на цю. В консолі це проблем не викликає, створюємо обидві, а потім модифікуємо права доступу, щоб вони посилалися один на одного.

Terraform так не вміє. Він вважає, що всі ресурси повинні бути створені за один запуск, без багатокрокових модифікацій, в порядку, згідно з їх залежностей (а якщо немає залежностей, то навіть паралельно). А тут виходить циклічна залежність, яка Terraform не влаштовує. Для вирішення проблеми придумали «псевдоресурс» aws_security_group_rule, який фактично є кроком модифікації

«справжнього» ресурсу aws_security_group. Циклічна залежність розривається. Справжні ресурси створюються в кілька кроків.

Джерела даних. Data Sources. Провайдери вміють не тільки створювати ресурси, а й запитувати атрибути вже наявних ресурсів. Наприклад, щоб вставити якісь властивості в ваші ресурси. Для цього і потрібні джерела даних. Але є і більш цікаві застосування. Наприклад, провайдер template дозволяє читати локальні файли – шаблони, підставляючи туди змінні. Синтаксис шаблонів повністю повторює синтаксис interpolations.

Модулі. Modules. Будь – яка конфігурація в Terraform може розглядатися як модуль. Поточна конфігурація, в поточному каталозі, називається root module. Будь – які інші конфігурації, в інших каталогах файлової системи, з реєстру модулів від HashiCorp, зі сховищ на GitHub або Bitbucket, просто доступні по HTTP, можна підключити до поточної конфігурації і використовувати.

При підключенні модулю дається ім'я. Під цим ім'ям він буде тут доступний. Потрібно вказати місце розташування модуля. Одне і те ж місце розташування модуля можна підключати кілька разів під різними іменами. Так що думайте про опубліковані конфігураціях як про класи, які можна перевикористати[6]. А конкретне підключення тут можна вважати екземпляром класу. Успадкування тільки немає. Зате делегування – будь ласка. Модулі можуть використовувати інші модулі.

Крім імені, модулю при підключенні потрібно задати аргументи. Це ті самі вхідні змінні. Всередині модуля вони будуть використовуватися як ${var.var_name}. А результат роботи модуль надає у вигляді вихідних змінних. На них в даній конфігурації можна посилатися як ${module.module_name.var_name}. В загалому, майже аналогічно використанню ресурсів.

Резюмуємо. Маємо набір * .tf файлів в поточному каталозі. Це – конфігурація. Або root module. Який може підключати інші модулі з різних місць. Інші модулі – це такі ж * .tf файли. Вони приймають змінні і повертають змінні. По ходу справи описують ресурси і використовують джерела даних. Маємо повне декларативне опис того, що ми хочемо отримати.

Стан. State. Друге, після конфігурації, ключове поняття Terraform. Стан зберігає стан ресурсів. Як воно відбувається насправді в цій хмарі. За замовчуванням стан – це один великий JSON файл terraform.tfstate, який створюється в поточному каталозі. Цей файл стану можна зберігати в системі контролю версій. Але, якщо сильно багато розробників будуть правити

конфігурацію Terraform, стан теж буде часто змінюватися. І доведеться постійно правити конфлікти, не кажучи вже про те, що не варто забувати робити pull.

Тому краще використовувати remote state. Стан можна зберігати в Consul. А, в разі AWS, в S3. Теж буде один файлик, але в хмарі, і з версіонування. І буде всім доступний.

Спочатку нашу конфігурацію потрібно ініціалізувати. Команда terraform init викачує плагіни (провайдери, бекенд для remote state), модулі (з GitHub, наприклад), і звалює це в прихований підкаталог .terraform. Цей підкаталог не потрібно зберігати в системі контролю версій. Але він потрібен для нормальної роботи Terraform. Тому terraform init –input=false – обов'язковий крок, щоб запускати Terraform з CI. Друга команда для використання Terraform в звичайному житті: terraform apply. Насправді вона виконує кілька кроків.

Крок перший**.** Refresh. Terraform порівнює поточний відомий стан з реальним станом ресурсів в хмарі. Провайдер виробляє купу запитів на читання через хмарниц API. І стан оновлюється. При першому запуску стан є порожнім. Значить, оновлювати нічого, і Terraform вважає, що жодного ресурсу не існує.

Крок другий. Plan. Terraform порівнює поточний відомий (і тільки що оновлене) стан з конфігурацією. Якщо в стані ресурсу немає, а в конфігурації він є, ресурс буде створений. Якщо в стані ресурс є, а в конфігурації його немає, ресурс буде видалений. Якщо змінилися аргументи ресурсу, то, якщо це можливо, ресурс буде оновлено на місці. Або ж ресурс буде видалений, а на його місці буде створений новий ресурс того ж типу, але з новими аргументами[6].

План буде представлений користувачу. З повним зазначенням того, що буде створено, видалено, або змінено. У тій мірі, наскільки це відомо до початку реального виконання. (Ідентифікатори ресурсів, наприклад, як правило призначаються випадково, і стають відомі лише після цього створення ресурсу.) Треба відповісти «yes», щоб перейти до наступного кроку.

Крок третій. Apply. Власне, застосування плану, створеного на попередньому кроці. Згідно залежностям. При цьому знову змінюється стан, туди записуються всі справжні властивості створених ресурсів. Ну от і все. Все просто. Звіряємося, порівнюємо, вираховуємо різницю, робимо зміни. На відміну від Ansible, звірка стану робиться один раз перед побудовою плану і для всієї конфігурації відразу.

Terraform – простий і впертий. І він не робить rollback. На жаль, в більшості випадків створення навіть одного ресурсу – не атомарно. Навіть створення S3 bucket – це з десяток різних викликів, в основному, щоб окремо з'ясувати різні

властивості цього бакету (get запити). Якщо якийсь крок не виконався (частою причиною помилок є недолік прав на окремі операції), Terraform вважає, що створення ресурсу провалилося.

Але в реальності ресурс таки міг створитися. Але це може не знайти відображення в стані Terraform. І при повторному запуску може трапитися повторна спроба створення ресурсу, яка може зламатися тепер уже через конфлікт імен. Або в хмарі може виявитися більш одного бажаного ресурсу. Крім того, Terraform зовсім не в курсі ресурсів, створених автоматично при виклику API хмари (приклад: Elastic Network Interface в AWS створюються неявно). І не в курсі ресурсів, що не описаних в конфігурації, але від яких можуть залежати його ресурси (приклад: Security Group в AWS не вийде видалити, поки хоч хтось її використовує, але ось хто цей хто, Terraform знати не завжди може).

Але Terraform упертий. Так що після редагування прав доступу, редагування помилок в конфігурації, і декількох запусків terraform apply, хмара таки невблаганно перейде в бажаний стан. Плюс може залишитися трохи сміття – непотрібних ресурсів, які були створені, але не були видалені. Це доведеться підчистити ручками. Потім, гадаю, прийде культура все робити через Terraform, і відразу вгадувати потрібні права. І сміття буде менше. Можна перенести інфраструктуру, набиту руками в консолі, в Terraform[6].

Пишете конфігурацію. Краще додавати один ресурс за одним. Робите terraform import. По суті, ви зіставляєте імена ресурсів в конфігурації (resource_type.resource_name) з реальними ідентифікаторами існуючих ресурсів (вони різні для різних типів ресурсів). Terraform намагається прочитати атрибути ресурсів і записати їх в стан. Робите terraform plan і дивіться, чи не намагається Terraform щось поправити. Якщо намагається, керуєте конфігурацію, і знову дивіться на план. В ідеалі Terraform скаже, що все ок, і нічого правити не буде. У разі гірше він все – таки що небудь створе заново.

Аналогічним чином, тільки використовуючи terraform refresh, можна привести конфігурацію у відповідність з тими змінами, які хтось зробив своїми руками. Імпорт не завжди працює ідеально. Зовсім не працює для «псевдоресурсів», про які я говорив раніше. Складнощі виникають зі складними ресурсами, з безліччю вкладених сутностей, на зразок тих же Security Group або Route Table в AWS.

Рефакторинг. Рефакторинг конфігурації Terraform. Він можливий. Потрібно тільки діяти акуратно. Перейменування ресурсу. Швидше за все Terraform

запропонує видалити старий ресурс, і створити новий, з новим ім'ям. Якщо це неприйнятно, можна спробувати видалити ресурс зі стану командою terraform state rm, а потім зробити terraform import. Є ще спеціальна команда terraform state mv, яка ніби як спеціально призначена для цього рефакторінга. Але я за допомогою state mv якось домігся стабільного crash of Terraform state. З тих пір остерігаюся.

Розбиття однієї великої конфігурації на кілька маленьких. На це є кілька причин. Причина перша. terraform apply виконується не сильно швидко. І чим більше ресурсів є в конфігурації, тим повільніше. Йому ж треба перевірити стан кожного ресурсу, навіть якщо в конфігурації цей ресурс і не змінювався. Має сенс виділити частини конфігурації виходячи з частоти змін і «охоплення території». Скажімо, VPC доведеться налаштовувати лише один раз, і потім майже ніколи не міняти. ECS кластер ви будете створювати кожен раз, коли буде з'являтися нове оточення, і потім знову без змін. А сервіси потрібно підновляти кожен індивідуально майже при кожному деплої. Причина друга. Terraform поки погано працює з версіон ресурсами. Типовий приклад: ECS Task Definition. Цей ресурс не можна видалити, тільки позначити як неактивний. Цей ресурс не можна змінити, лише створити нову ревізію. Тому на кожен terraform apply буде «що знаходиться на відстані» старий Task Definition, і створений новий Task Definition, навіть якщо в конфігурації нічого не змінювалося. І оновлений ECS Service, який цей Task Definition використовує[6].

Це призведе до того, що цей сервіс оновить і перезапустить свої завдання. Тобто відбудеться справжній редеплой. Це добре, тому що можна робити редеплой ECS сервісів за допомогою Terraform. Це погано, тому що я не хочу редеплоіть все два десятка моїх сервісів одночасно по terraform apply однієї великої конфігурації. Сервіси потрібно виносити в окремі конфігурації Terraform.

Добре. Створити ще одну папку з * .tf файлами поруч – не проблема. Перенести туди управління ресурсами за допомогою terraform import теж можна. Але ж наші сервіси повинні дещо знати про кластер: ім'я кластера, ті ж Security Groups, ще параметри. Конфігурації залежать від інших конфігурацій. З більш загальних змін необхідно передати параметри більш конкретним конфігураціям. Кластер повинен знати про VPC, сервіси повинні знати про кластери.

Можна передати все, що потрібно, кілка потрібних ідентифікаторів, руками. Як вхідні змінні нашої конфігурації. Буде працювати. Якщо ви правильно скопіювали ці ідентифікатори.

Дві основні переваги Terraform:

– Супер – портативність – одна утиліта і одна мова застосовується для опису хмарної інфраструктури в Google Cloud, AWS, OpenStack і роботи з будь – яким іншим постачальником послуг. Зміна постачальника хмарного хостингу більше ніколи не буде проблемою.

– Простота повноцінного запуску додатків – припустимо, на ваших серверах Amazon запущені Docker контейнери під керуванням Kubernetes, в яких працює широкий спектр додатків – і все це з легкістю управляється за допомогою одного інструмента.

Недоліки Terraform.

Так як Terraform з'явився відносно недавно, він ще далеко не ідеальний. Наприклад, розробник одного разу виправив баг в Terraform ignition provider і прибрав indents з JSON, що призвело до необхідності перенастроювання всіх раніше розміщених конфігурацій. Інший важливий момент полягає в тому, що як диригентську паличку може використовувати тільки один диригент, так і Terraform бажано використовувати одному оператору – або хоча б з одного терміналу. Досить часто, запуск однієї і тієї ж команди з різних машин (з різними конфігураціями Terraform та іншого ПО) приводить до різних результатів.

Само собою, це піднімає невирішені питання про управлінні процесом і можливості знаходження відповідального за ту чи іншу проблему. Поки ці питання не вирішені розробником, працювати з Terraform краще поодинці (або позмінно, але з одного терміналу). Третій суттєвий недолік Terraform полягає в тому, що утиліта була розроблена тільки під хмару і непридатна для використання з кластерами виділених серверів, хоча більш досвідчені конкуренти типу Salt, Ansible, Puppet прекрасно з цим справляються, адже їх розробили більше 5 років тому саме для такої роботи. Це робить Terraform досить специфічним рішенням, яке не може підійти в 100% сценаріїв[6].

Однак, я впевнений, що протягом кількох наступних років всі ці недоліки Terraform будуть усунені, а його безсумнівні переваги виявляться ще більш яскраво.

Я вважаю, що Terraform в даний момент є однією з кращих утиліт для управління конфігураціями хмарної інфраструктури. Можливо, цей інструмент ще не такий популярний як Chef, Ansible, або Puppet, але впевнений, що з часом ця утиліта придбає набагато більшу популярність. Розробники Vagrant і Consul ще раз довели свою майстерність і надали зручний і корисний продукт. Так як Terraform

це рішення з відкритим кодом, його розвиток підтримується великою і безперервно зростаючим спільнотою розробників. Terraform, безсумнівно, крутий, і з часом стане тільки краще. Ця утиліта не стане "вбивцею" Salt, Ansible, або Puppet, але по праву займе гідне місце в інструментарії будь – якого DevOps інженера.

## 2.2 Cloud Formation

AWS CloudFormation надає універсальну мову для опису і виділення всіх ресурсів інфраструктури в хмарному середовищі. CloudFormation дозволяє використовувати мови програмування або простий текстовий файл для автоматичного безпечного моделювання і виділення всіх ресурсів, необхідних для додатків по всіх регіонах і акаунтів користувача. Так ви отримаєте єдине джерело достовірної інформації про ресурси AWS. Сервіс AWS CloudFormation надається безкоштовно – оплаті підлягають тільки ресурси AWS, необхідні для запуску додатків.

AWS CloudFormation дозволяє моделювати всю інфраструктуру за допомогою якого текстового файлу, або мов програмування. Такий підхід дозволяє отримати єдине джерело достовірної інформації про ресурси AWS і стандартизувати компоненти інфраструктури, які використовуються в організації, забезпечуючи відповідність конфігурації вимогам і прискорене усунення неполадок.

Найпростіший спосіб описати, що таке CloudFormation, це те, що це інструмент від AWS, який дозволяє без особливих зусиль відкручувати ресурси. Ви визначаєте всі ресурси, які ви хочете, щоб AWS розгорнувся в документі – план, натискаєте кнопку, а потім AWS магічно створює все це. Цей план називається шаблоном у програмі CloudFormation. CloudFormation гарантує, що залежні ресурси у вашому шаблоні створюються у належному порядку. Наприклад, скажімо, що ми хочемо створити запис DNS Route53 та екземпляр EC2, який має точку запису DNS на екземпляр EC2[6]. CloudFormation піклується про те, щоб спочатку надати екземпляр EC2, дочекатися його готовності, а потім створити запис DNS після цього. AWS CloudFormation «оркеструє» надання необхідних ресурсів.

Тож замість того, щоб писати сценарій з купою викликів API AWS, циклів очікування та повторювати логіку, ви просто скажете, що опишіть, що ви хочете, і скажіть CloudFormation, щоб він це зробив за вас. Гарно.

Ось коротке пояснення того, що кожен означає із параметрів:

– AWSTemplateFormatVersion: Вказує версію шаблону AWS CloudFormation.

– Опис: Текстовий рядок, що описує шаблон.

– Відображення: відображення ключів та пов'язаних з ними значень, які можна використовувати для визначення умовних значень параметрів. Це версія CloudFormation висловлювання "case".

– Вихідні дані: описує значення, які повертаються під час перегляду властивостей вашого стека. Це відображається на консолі AWS CloudFormation.

– Параметри: Вказує значення, які можна передати у ваш шаблон під час виконання.

– Ресурси: Вказує ресурси стека та їх властивості, як наш екземпляр EC2. Це єдина необхідна властивість.

AWS CloudFormation виділяє ресурси безпечним і відтвореним чином, дозволяючи створювати і пересоздавати інфраструктуру і додатки без необхідності виконувати ручні дії або писати власні скрипти. CloudFormation сам визначає, які операції слід виконувати при управлінні стеком, і автоматично скасовує зміни, якщо виявляються помилки (рис. 2.5).

Інфраструктуру можна створити за допомогою будь – якого редактора коду, реєструвати в системі управління версіями і перевіряти отримані файли разом з колегами перед розгортанням в робочому середовищі.

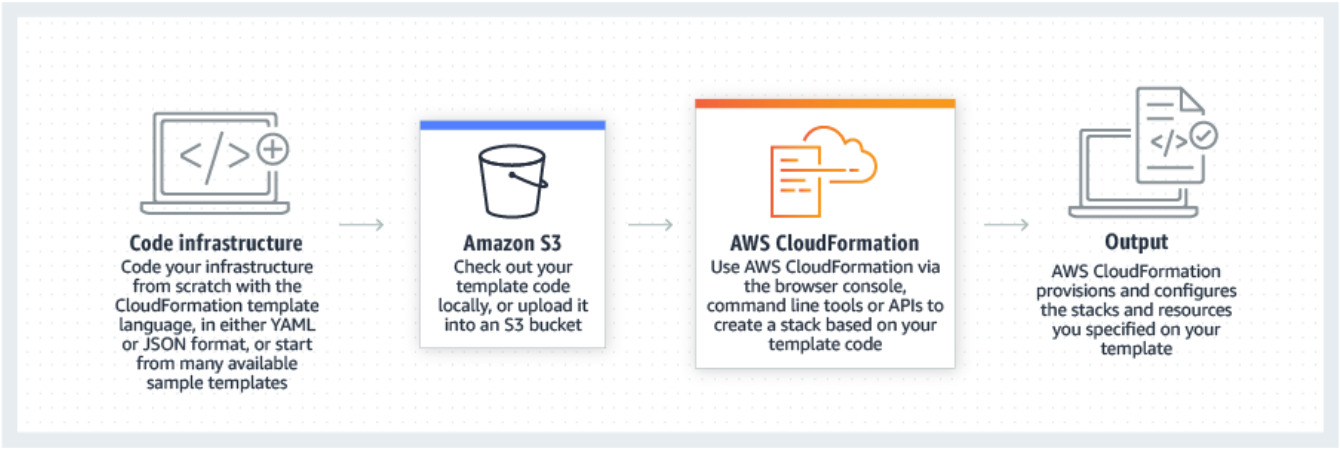


Рисунок 2.5 – Принцип роботи AWS CloudFormation


Рисунок 2.6 – Опис ресурсу AWS CloudFormation

З CloudFormation ви можете зробити набагато більше, ніж було зроблено в цьому розділі. CloudFormation відома своєю крутою кривою навчання. Сподіваюсь, ви можете побачити з цього опису, що крута крива навчання – це насправді не саме CloudFormation, а насправді вивчаючи всі самі ресурси, які ви зможете обернути за допомогою CloudFormation (рис. 2.6). Це як вивчення мови програмування. Саму мову програмування не так складно вивчити, це стандартні бібліотеки, зовнішні бібліотеки, екосистему, яка потребує певного часу, щоб навчитися та вільно володіти.

## 2.3 Порівняльна характеристика Terraform та CloudFormation

Отже, який же інструмент вибрати для оркестрації ресурсів.
Для початку треба відобрази відмінності один від одного, після чого все ж таки вибрати найбільш відповідний для нас.

Terraform – це відповідь громади на інфраструктуру як код. Він часто менш багатослівний, ніж CloudFormation, і має чудову модульну систему[8].
CloudFormation розроблений і підтримується AWS, він дуже тісно інтегрований і підтримує лише AWS. Тісна інтеграція піддається великому користувальницькому

інтерфейсі, чудова можливість перегляду всіх ваших стеків для даного облікового запису. Це дозволяє з легкістю посилатися на поперечні стеки, що також є великим виграшем модульності та руйнуванням цього моноліту.

Інтерфейс командного рядка та робочий потік.

Найпоширеніша командна функція Terraform init та plan. Оскільки Terraform є хмарноактивним, потрібно чітко сказати, де ви будете зберігати свій файл стану, це частина команди init. Запуск плану Terraform використовується для створення плану виконання, залежно від вашого державного файла та наявної інфраструктури.

CloudFormation має подібний процес, коли ви створюєте та виконуєте набори змін. Ви будете використовувати комбінацію створення – зміни – набору, виконання – зміни – набору та розгортання. create –change–set схожий на план Terraform, тоді як набір Execute–change – це застосувати і розгорнути вихід обох команд в одну[8]. Основна відмінність полягає в тому, що ви будете розглядати та вирішувати проблеми через інтерфейс CloudFormation.

Незмінний стан.

Я провів простий експеримент, де обидва інструменти створили простий екземпляр EC2. Я видалив обидва екземпляри і спробував запустити обидва етапи розгортання знову (рис. 2.7). Приховане в плані Terraform – це оновлення, яке використовується для узгодження вашого state файла з реальною інфраструктурою світу. Це означає, що Terraform зміг виявити видалення та завантажив новий екземпляр. CloudFormation не має такого узгодження і залежить від наявного стека. Таким чином, він був непохитний, нічого не змінилося. Єдиним виправленням CloudFormation було перейменування ресурсу EC2.


Рисунок 2.7 – Порівняння опису ресурсів

Умовні розгортання.

Умовні розгортання є важливою частиною будь – якого розгортання, яке ви хочете розгортати з одного стека, незалежно від його виробництва чи розвитку. Але іноді виробництво просто вимагає додаткових ресурсів, або навпаки[8]. Тільки CloudFormation явно підтримує умови, ви можете позначати кожен ресурс умовним прапором.

Модулі.

Модулі, що ексклюзивні до Terraform, є дуже потужним способом зруйнувати розгортання Terraform. Модуль Terraform – це сукупність файлів та ресурсів Terraform, які визначають набір інфраструктури. Це дозволяє вводити та виходити. Щоб використовувати модуль, ви просто передаєте свої входи як змінні, і під кришкою Terraform витягне модуль і виконає код Terraform як завжди.

Таблиця 2.1 – Порівняння характиристика CloudFormation і Terraform

| | Terraform | CloudFormation |
|---|---|---|
| Тип | Оркестрація | Оркестрація |
| Хмара | AWS, GCP, Azure та інші | Тільки AWS |
| Формат конфігурації | JSON | HCL/JSON |
| Контроль стану | + | – |
| Запуск без застосування | terraform plan | deploy –no-execute-change-set |
| Контроль застосування | + | – |
| Управління вже створенними ресурсами | + | – |
| Можливість розширення | Провайдери | – |
| Управління секретними даними | Видимі у state файлі | Не видимі |
| Перевірка змінень | Детальний план перед виконанням | Change Sets |

CloudFormation використовує вкладені стеки для виконання того ж завдання. Проблема з вкладеними стеками полягає в тому, що якщо дочірній стек виходить з ладу, весь стек також. Це спричиняє болі при розгортанні.

CloudFormation має чудовий інтерфейс як для налагодження, так і загального огляду всього, що відбувається. Перехресне посилання є потужним і робить розділення стеків простим, тоді як мені особисто дуже подобається синтаксис. Відсутність підтримки – це моя справжня прихильність, оскільки думка про необхідність використання користувацьких ресурсів або скриптів оболонки для отримання певних функцій не викликає відкритого джерела[8].

Terraform чудовий тим, що це яскраве співтовариство з відкритими джерелами, це проста модульна парадигма і це хмарна агностика. Terraform може бути важким для налагодження через кліп, і часто ви стикаєтесь з великим монолітним репо, що включає всю вашу інфраструктуру. Ви можете розділити його на модулі, але це не однакове розділення проблем, які ви можете отримати із CloudFormation.

Отже, як висновок, можна сказати що обидва інструмента мають свої переваги та недоліки. В залежності від прикладного завдання, керуючись описаними функціями CloudFormation і Terraform, користувач має сам вибрати що йому до до вподоби.

У нашому конкретному випадку, керуючись вихідними даними, обираємо Terraform.

# 3 АНАЛІЗ ТА ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА ІНСТРУМЕНТІВ АВТОКОНФІГУРАЦІЇ РЕСУРСІВ ВЕБ – ПРОЕКТУ

Швидкий розвиток віртуалізації укупі зі збільшенням потужності серверів, які відповідають промисловим стандартам, а також доступність «хмарних» обчислень привели до значного зростання числа потребуючих управління серверів, як всередині, так і поза організацією. І якщо колись це робили за допомогою стійок з фізичними серверами в центрі обробки даних поверхом нижче, то тепер треба управляти набагато більшою кількістю серверів, які можуть бути розподілені по всій земній кулі.

У цей момент засоби управління конфігураціями і вступають в гру. У багатьох випадках, ми управляємо групами однакових серверів, на яких запущені однакові додатки і сервіси. Вони розміщуються на системах віртуалізації всередині організації, або ж запускаються як «хмарні» і гостьові в віддалених ЦОД. У деяких випадках, ми можемо говорити про велику кількість обладнання, яке існує тільки для підтримки дуже великих додатків або про обладнання, що обслуговує міріади невеликих сервісів.

У будь – якому випадку, можливість змусити їх усіх виконати волю системного адміністратора не може бути знецінена. Це єдиний шлях управляти величезними і зростаючими інфраструктурами. Puppet, Chef і Ansible були задумані щоб спростити налаштування та обслуговування десятків, сотень та навіть тисяч серверів. Це не означає, що маленькі компанії не отримають вигоди від цих інструментів, так як автоматизація зазвичай робить життя простіше в інфраструктурі будь – якого розміру.

У даному розділі буде розлянуто 2 найпоширеніших інструменти автоконфігурації – Puppet та Chef.

## 3.1 Puppet

Puppet вважається найбільш використовуваним з інструментів автоконфігурації[9]. Він найбільш повний з точки зору можливих дій, модулів і призначених для користувача інтерфейсів, представляючи повну картину ЦОД, охоплюючи майже кожну операційну систему і надаючи утиліти для всіх основних

ОС. Початкова установка відносно проста, вимагає розгортання головного сервера і клієнтських агентів на кожній керованій системі.

Інтерфейс командного рядка дозволяє завантажувати і встановлювати модулі за допомогою команди puppet. Потім потрібні зміни в конфігураційних файлах, необхідні для налаштування модуля під потрібні завдання, клієнти, які повинні отримати інструкції, отримають їх при наступному зверненні до головного сервера, або через запит від сервера, який ініціює процес зміни негайно.

Також є модулі, за допомогою яких виконується налаштування віртуальних і розміщених в «хмарах» серверів. Всі модулі та конфігурації будуються за допомогою вбудованої, заснованої на ruby, мови, або ж на самому ruby. Це вимагає деяких знань програмування, на додаток до навичок системного адміністрування.

У Puppet Enterprise найбільш повний веб – інтерфейс з усіх, що дозволяє контролювати керовані вузли в реальному часі за допомогою попередньо створених модулів , розміщених на головних серверах. Веб – інтерфейс хороший для управління, але не дозволяє проводити «тонку» настройку модулів. Інструменти для побудови звітів добре розроблені, надаючи глибоку деталізацію про поведінку агентів і про внесені зміни[9].

Архітектура Puppet побудована на двох взаємодії двох окремих модулів: PuppetMaster, що встановлюється на керуючі вузли, і PuppetAgent, що встановлюється на керовані. Подібне рішення має на увазі або централізовану, або слабо – розподілену архітектуру і забезпечує високу ступінь горизонтального масштабування.

У Puppet використовується декларативний стиль опису конфігурацій на власній предметно – орієнтованої мови, заснованії на синтаксисі Ruby. В якості методів доставки конфігурації дане рішення поєднує "Push" і "Pull" технології.

Існують Open – source і Enterprise версії Puppet, що відрізняються наявністю додаткових вбудованих компонент управління інфраструктурою (зокрема Web – інтерфейсу), засобів моніторингу, додатковими модулями, доступними в офіційній бібліотеці. Дана система підтримує операційні системи Unix, MacOCX, Windows для керованих вузлів і системи сімейства Unix для керуючих.

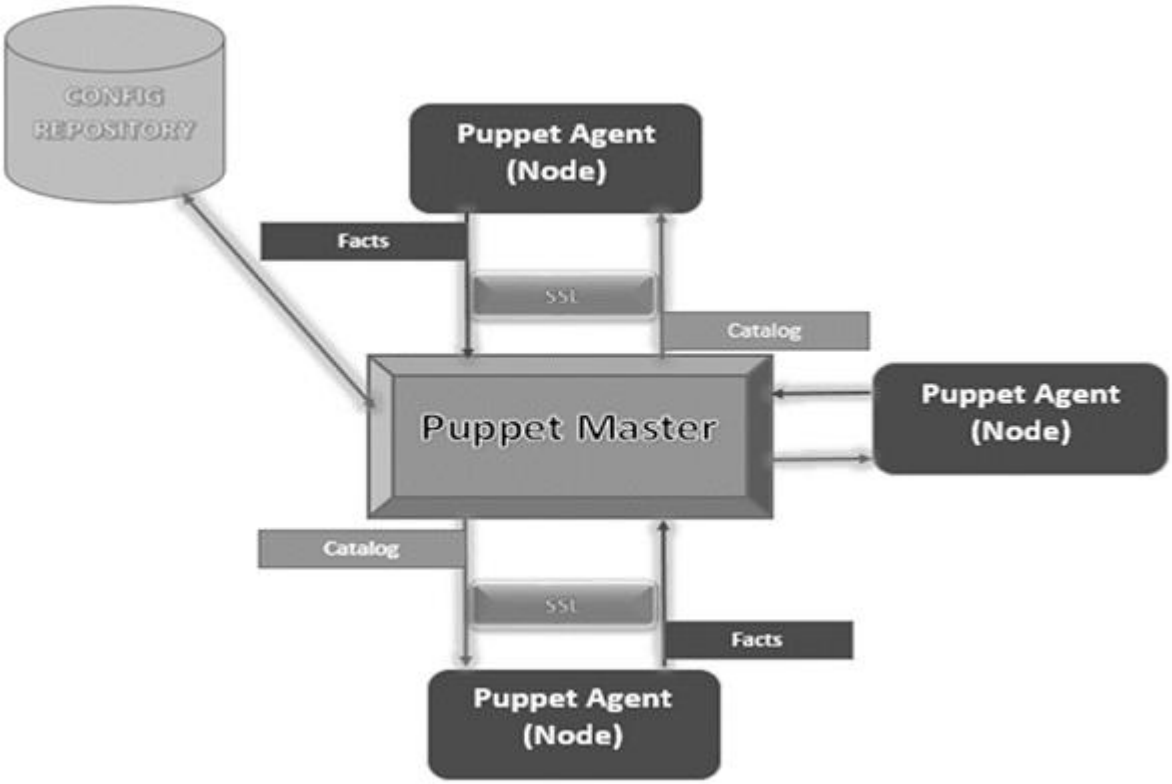


Рисунок 3.1 – Архітектура Puppet

Серверна частина програми створена для зберігання маніфестів, так в програмної термінології Puppet називаються файли конфігурації. В процесі роботи сервер приймає звернення з клієнтських машин і автоматично відсилає їм оновлені файли конфігурації ОС для роботи в мережі. На клієнтських комп'ютерах також має бути встановлено програмне забезпечення Puppet, вже у вигляді клієнтської частини. Як правило, дані установчі пакети включаються в саму операційну систему, що дозволяє швидко розгортати комп'ютерну мережу, проте, в разі їх відсутності, доведеться завантажувати необхідну збірку з сайту розробника (рис. 3.1).

Додаткова зручність даного рішення в тому, що один адміністратор за допомогою сервера може здійснити налаштування і управління сотень і тисяч машин, об'єднаних в мережу. Якщо виникнуть якісь проблеми, то відгук з місць дозволить адміну швидко поправити код і усунути їх. Хоча в даному випадку зростають вимоги до уважності адміна – один невірно написаний рядок коду конфігурації може привести до неполадок по всій мережі. Хоча, якщо розібратися,

в даному випадку можна запустити працюючий маніфест попередньої збірки і відновити все досить оперативно.

Крім всіх компонентів, які беруть участь у формуванні описаного потоку даних, існує безліч типів даних, що використовуються Puppet в ході виконання операцій внутрішнього взаємодії. Ці типи даних є критичними, так як з їх допомогою здійснюються всі взаємодії та вони є публічним типами даних, які можуть приймати або генерувати будь – які інші інструменти.

Найбільш важливими типами даних є:

– Набори фактів (Facts): Системні дані, що збираються на кожній машині і використовуються для компіляції специфікацій конфігурації.

– Маніфест (Manifest): Файли, що містять код Puppet і зазвичай об'єднуються в рамках колекцій, званих "модулями".

– Каталог (Catalog): Граф ресурсів заданого вузла для управління і встановлення залежностей між ними.

– Звіт (Report): Набір всіх подій, що генеруються під час використання заданого каталогу.

Крім наборів фактів, маніфестів, каталогів і звітів, Puppet підтримує типи даних для роботи з файлами, сертифікатами (які він використовує для аутентифікації), а також деякі інші.

Багато модулів перед використанням вимагають доопрацювання і / або виправлення помилок[9]. Наприклад, може не перевірятися поточна операційна система і встановлюватися пакет 'apache2' в CentOS (правильна назва пакета в цьому випадку – 'httpd'), як ніби це Ubuntu. Крім того, наявність великої кількості модулів ускладнює вибір. Частково дана проблема вирішується сортуванням за релевантністю і числу завантажень в репозиторії Puppet Forge, а також набором модулів від самого виробника. Крім того, у вбудованій мові присутні підводні камені. Зокрема, пов'язані з залежностями.

Установка і настройка Puppet для роботи в клієнт – серверному режимі досить проста, якщо дотримуватися кількох умов. Потрібно обов'язково синхронізувати час на машинах і перевірити роботу розпізнавання імен (name resolution) за допомогою DNS або файлу hosts. У першому випадку, якщо цього не зробити, виникнуть помилки перевірки сертифіката при підключенні Puppet до

сервера. У другому він не зможе визначити ім'я машини, на якій працює агент. В автономному режимі роботи таких проблем немає.

```
class { 'mysql::server' :
  root_password => 'password'
}

mysql::db{ ['test', 'test2', 'test3']:
  ensure => present,
  charset => 'utf8',
  require => Class['mysql::server'],
}

mysql::db{ 'test4':
  ensure => present,
  charset => 'latin1',
}

mysql::db{ 'test5':
  ensure => present,
  charset => 'binary',
  collate => 'binary',
}
```

Рисунок 3.2 – Приклад Puppet маніфесту

Абстракція Puppet приблизно наступна:

– Вузол (node) – це сукупність конфігурації для конкретної цільової системи. На ділі це окремий клас.

– Клас (class) – це набір декларативної логіки, яка включається в конфігурацію вузла або інші класи. У класу немає ні примірників, ні методів, зате є параметри та змінні, описані всередині логіки. На ділі, це скоріше процедура, яка може успадковувати іншу процедуру банальним нарощуванням коду і не зовсім банальною областю видимості змінних (рис. 3.2).

– Тип (type) – а ось це вже більше скидається на класичний клас – у нього передбачаються екземпляри з ім'ям і виразно заданими параметрами, але нічого більше. Конкретна реалізація типу може бути написана у вигляді Puppet скрипта

через define, який створює екземпляри інших типів або ж у вигляді розширення на Ruby.

– Ресурс (resource) – власне це і є іменовані екземпляри типів. Ім'я кожного ресурсу унікально в рамках конкретного типу в межах конфігурації вузла (каталогу).

– Змінні (variable) – загалом, це константи. До версії Puppet 4 з їх областю видимості було все ще гірше. Тепер вона адекватна: для використання ззовні місця визначення потрібно задавати повністю кваліфікований ідентіфіактор, за винятком випадку успадкування класів.

Ресурс – це найменша одиниця абстракції в Puppet.

Ресурсами можуть бути:

– Файли;

– пакети (Puppet підтримує пакетні системи багатьох дистрибутивів);

– сервіси;

– користувачі;

– групи;

– завдання cron.

Ресурси можна (і зазвичай потрібно) групувати в класи. Наприклад, клас для Postfix буде містити ресурси і для установки і для настройки поштового сервера Postfix. Одним з головних елементів опису конфігурації є вузол (node). Вузол дозволяє описувати функціональність певного типу[9]. Скажімо, можна описати вузол «офісний десктоп», або «веб – сервер». Відповідно, для офісного десктопа буде встановлено і налаштовано все необхідне для офісу ПЗ, а для веб – сервера який – небудь apache або nginx.

Puppet може використовуватися для локального розгортання без мережі і відповідної інфраструктури. Це може використовуватися для створення образів контейнерів. Є навіть цілий напрям який полягає в відмові від централізованого сервера.

Потрібно розуміти, що Puppet Server стає вразливим місцем всієї IT інфраструктури, тому що визначає кінцеву конфігурацію всіх систем. В особливих випадках має сенс робити поділ – окремий сервер для критичних елементів інфраструктури з вкрай обмеженим доступом і ручним оновленням і другий для всього іншого.

Доступність Puppet Server визначає можливість управління всією інфраструктурою. Має сенс розміщувати Puppet Server на віртуальній машині в більш надійній сторонній хмарі, ніж власні можливості. Або ж слід встановлювати кілька серверів[9].

Розглянутий програмний продукт дозволяє виконувати практично всі заходи, пов'язані з адмініструванням системи, організовувати автоматичне виконання повторюваних операцій, а також здійснювати управління процесом розгортання стеків програмного забезпечення для швидкого впровадження програмних продуктів, що є ключовою функцією в сучасному світі хмарних сервісів і масштабованих систем.

## 3.2 Chef

Chef схожий на Puppet з точки зору загальної концепції, в ньому також є головний сервер і агенти, які встановлені на керованих вузлах. На додаток до головного сервера, установка Chef також вимагає робочої станції, для управління ним. Агенти можуть бути встановлені з робочої станції за допомогою утиліти knife, яка використовує протокол SSH для розгортання, полегшуючи процес установки. Після цього, керовані вузли автентифіковані з головним за допомогою сертифікатів.

Конфігурація Chef тісно пов'язана з системою управління версіями Git, тому знання того, як працює Git необхідно для роботи. Також як і Puppet, Chef заснований на ruby, тому буде потрібно і знання цієї мови. Як і у випадку з Puppet, модулі можуть бути завантажені або написані «з нуля», після чого встановлені на приймаючі вузли, відповідно до необхідних налаштувань.

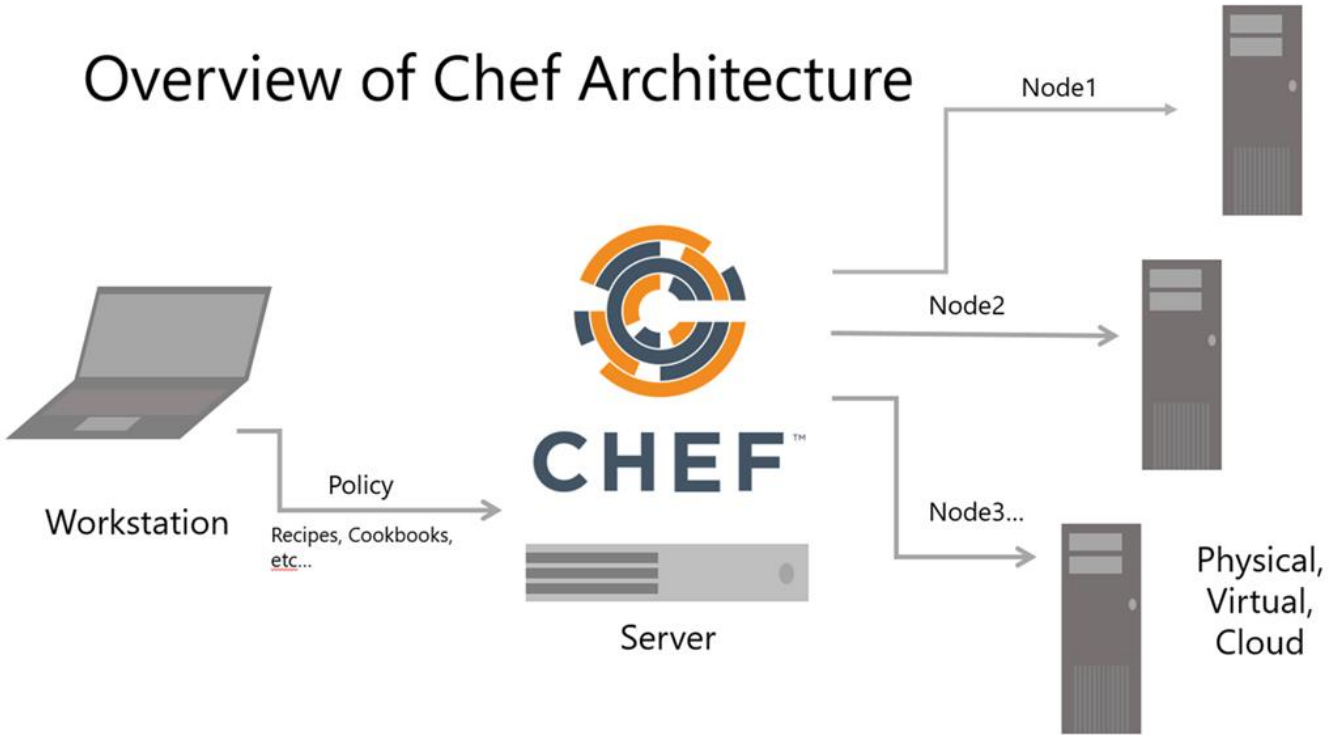


Рисунок 3.3 – Архітектура Chef

Призначений для користувача веб – інтерфейс функціональний, але не надає можливості модифікувати конфігурації. Він не так сповнений, як веб – інтерфейс Puppet Enterprise, поступається в побудові звітів і деяких інших функціях, але дозволяє вести облік обладнання та організацію вузлів. Як і у Puppet, у Chef великий набір модулів і рецептів налаштувань, переважно на ruby. З цієї причини, Chef добре підходить для інфраструктур, орієнтованих на розробку (рис. 3.3).

Chef – це клієнт – серверна архітектура, є Chef – сервер і Chef – клієнт. Конфігурація заснована на пошуку, написана на Ruby, має Ruby DSL. У Chef величезна спільнота і найбільший набір інструментів серед усіх SCM. Ось так виглядає код на Chef, це розгортання Jetty.

```
#
# Cookbook Name:: dg-app-edl
# Recipe::fe
#

node.normal[:jetty][:home] = "/usr/share/jetty"
node.normal[:jetty][:group] = "deploy"

include_recipe "dg-auth::deploy"
include_recipe "newrelic::repository"
include_recipe "newrelic::server-monitor"
include_recipe "dg-jetty::jetty9"
include_recipe "newrelic::java-agent"

directory "edl" do
  action :create
  owner
  group "deploy"
  mode "0775"
  path "/usr/share/where/edl"
  recursive true
end
```

Рисунок 3.4 – Приклад Chef рецепту

Інфраструктура Chef.

– Nodes (Ноди) – це будь – який ваш сервер, фізичний чи віртуальний, який ви будете налаштовувати за допомогою Chef.

– The Server (Chef – сервер) – безпосередньо сам Chef – сервер до якого звертаються клієнти (Nodes), сервер складається з декількох компонентів: Web – UI – додаток на RoR представляє собою веб – інтерфейс для сервера;

– Erchef – ядро сервера, починаючи з Chef 11.x написано на Erlang, повністю сумісний з колишньою версією на Ruby[10];

– Bookshelf – сховище для "куховарських книг"; Nginx – йде в комплекті, всі API – запити проходять через нього;

– PostgreSQL – також включений до складу Chef – сервера, очевидно виконує роль сховища інформації.

– Workstations (робоча станція) – робоче місце адміністратора Chef, тобто ваш ПК на якому ви буде готувати рецепти, куховарськиі книги і управляти всіми вузлами за допомогою knife.

– Knife – це основний інструмент для роботи з Chef з консолі. Саме за допомогою "ножа" ви будете керувати нодами і Chef – сервером (рис. 3.4).

Терміни Chef:

– Cookbook (куховарська книга) – сховище атрибутів, рецептів, шаблонів і файлів.

– Attribute (атрибути) – параметри конфігурацій які ви можете задавати для нод як через рецепти, так і через ролі.

– Templates (шаблони) – ви можете поширювати конфігурації як звичайними файлами так і параметризованними erb – шаблонами. Значення параметрів шаблону в певному порядку підставляються з атрибутів куховарських книг або з ролей.

– Files – будь – які файли, які ви зможете поширювати за допомогою рецептів попередньо описавши їх шлях призначення та права.

– Definitions (визначення) – використовуються для опису нових або існуючих ресурсів (наприклад служба, або віртуальний хост apache) і / або дій над ними[10].

– Libraries (бібліотеки) – ні що інше як вставки коду на Ruby, як розширення для рецептів.

– Resources (ресурси) – ресурс це будь – який об'єкт системи – файл, користувач, група, сервіс та ін.

## 3.3 Порівняльна характеристика Puppet та Chef

Тепер, коли ми маємо уявлення, звідки береться кожен із цих інструментів, порівняємо іх. Вони більш схожі чи різні, і що краще. Ми отримаємо відповіді, розбивши інструменти на основі деяких важливих характеристик:

Доступність.

У цьому контексті доступність визначається як доступність інструментів під час позапланових перерв обслуговування. Якщо основний Chef сервер виходить з ладу, резервний сервер займає його місце. Puppet включає в себе архітектуру з

декількома майстрами, тож якщо активний майстер не працює, для нього потрібно доповнити іншого майстра.

Терміни та поняття.

Оскільки менеджери конфігурації абстрагують файли конфігурації, важливо ознайомитись з термінами, унікальними для кожного інструменту. Chef пропонує вам працювати з кулінарними книгами та рецептами, а Puppet працює з маніфестами та модулями. Рецепти та маніфести, як правило, описують окремі поняття, тоді як кулінарні книги та рецепти описують більш загальні поняття.

Вартість підприємства: Chef Automate має щорічну плату в розмірі 137 доларів за вузол, і ця ціна дає вам все необхідне для створення та розгортання. Ціни на puppet коливаються від 112 доларів за вузол / рік, якщо ви вибираєте стандартний план підтримки, до 199 доларів за комплект вузла / рік у плані підтримки преміум – класу.

Спільні ознаки.

Налаштування.

Chef функціонує з архітектурою майстер – клієнт, при цьому сервер працює на головній машині, а клієнт працює як агент на кожній клієнтській машині. Chef також включає в себе додатковий компонент під назвою «Робоча станція», який обробляє всі конфігурації, які тестуються, зберігаючи, а потім переміщуючи їх на центральний сервер[10]. Puppet також використовує розташування архітектури головного агента. Puppet сервер працює на головній машині, тоді як puppet клієнти працюють як агент на кожній клієнтській машині. Також є підписання сертифіката – мастера – агента.

Отже, іншими словами, обидва інструменти важко встановити. Керування інструментами: якщо ви використовуєте Chef, треба мати навички програмування, оскільки вони вам знадобляться для успішного керування конфігураціями. Клієнт Chef витягує конфігурації з сервера, і ці конфігурації знаходяться в Ruby DSL.

На жаль, Puppet не менш задіяний. Він використовує власну мову під назвою PuppetDSL (Domain Specific Language). Процес орієнтований на системних адміністраторів і має функції негайного віддаленого виконання. Підсумок, обидва інструменти використовують pull конфігурації. Це означає, що ведені вузли автоматично витягують конфігурації з центрального сервера без необхідності жодних команд. Це на відміну від конфігурації push, де всі конфігурації, присутні в центральному сервері, висуваються до вузлів.

Масштабованість.

Chef і Puppet дуже легко масштабуються. Вони можуть працювати з великою інфраструктурою, якщо користувач надає IP – адресу та ім'я хостів вузлів, які потрібно налаштувати. Інструменти впораються з рештою.

Мова конфігурації.

Chef використовує Ruby DSL (мова, що визначається доменом), мову, орієнтовану на розробників, яку важко вивчити. Puppet використовує вищезгадану PuppetDSL, ще одну складну для вивчення мови.

Таблиця 3.1 – Переваги та недоліки Puppet та Chef

| | Puppet | Chef |
|---|---|---|
| Переваги | Повний інтерфейс користувача | Підхід, керований кодом, означає більшу гнучкість та контроль конфігурацій |
| | Потужні можливості звітування | Інструмент «knife» зменшує головний біль у установці |
| | Надає доступ до добре створеної спільноти підтримки | Надає широку колекцію рецептів конфігурації та модулів |
| Недоліки | Його модельний підхід дорівнює меншому контролю в порівнянні з підходами, керованими кодами | Якщо ви ще не знаєте Ruby та процедурного кодування, приготуйтеся до крутої кривої навчання |
| | Розширені завдання вимагають CLI, і оскільки він заснований на Ruby, вам потрібно буде ознайомитися з ним | Це складний інструмент |

Зрештою, ідеальний інструмент DevOps залежить від потреб та цілей організації. Chef існує давно в порівнянні з більшістю інших інструментів DevOps і демонструє чудові можливості для вирішення надзвичайно складних завдань.

Якщо у вашій організації є команди, орієнтовані на розвиток і оточення, то Chef – ідеальний вибір.

Puppet також був створенний давно і був розгорнутий та перевірений у виборі великих, вимогливих середовищ. Виходячи з цього, Puppet є хорошим вибором для великих підприємств, які віддають перевагу добре перевіреним та давнім інструментам. Остаточний вибір зводиться до усвідомлення того, що важливо для вашого бізнесу[10].

Виходячи з віхідних даних, обираємо Puppet.

# 4 ПРОЕКТУВАННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО РОЗГОРТАННЯ ІНФРАСТРУКТУРИ ВЕБ – ПРОЕКТУ

Отже, ми визначилися з інструментами автоконфігурації (Puppet) та оркестрації (Terraform). Нагадаємо, що в якості хмарного провайдера буде використано AWS.

В даному розділі будет побудована платформа для тестування знань студентів в області системної інженерії за допомогою інструментів автоконфігурації та оркестрації.

Вихідні дані до проекту:

– Декілька віртуальних серверів для розгортання додатку;

– База даних MySQL для зберігання результатів тестування та списку студентів;

– Налаштування мережного захисту за допомогою AWS Security Groups.

## 4.1 Установка та налаштування Puppetmaster

Для початку, треба встановити Puppetmaster на віртуальну машину. В нашому конкретному випадку, це буде віртуальна машина у хмарі AWS. Установка буде проводитися в ручному режимі, для того щоб показати та освітити процедуру установки без впровадження автоматизації (рис. 4.1).

Отже, піднімемо VM на платформі Ubuntu та з розміром m5.xlarge (16.0 GiB RAM та 4 vCPUs) та встановимо на неї Puppetmaster.

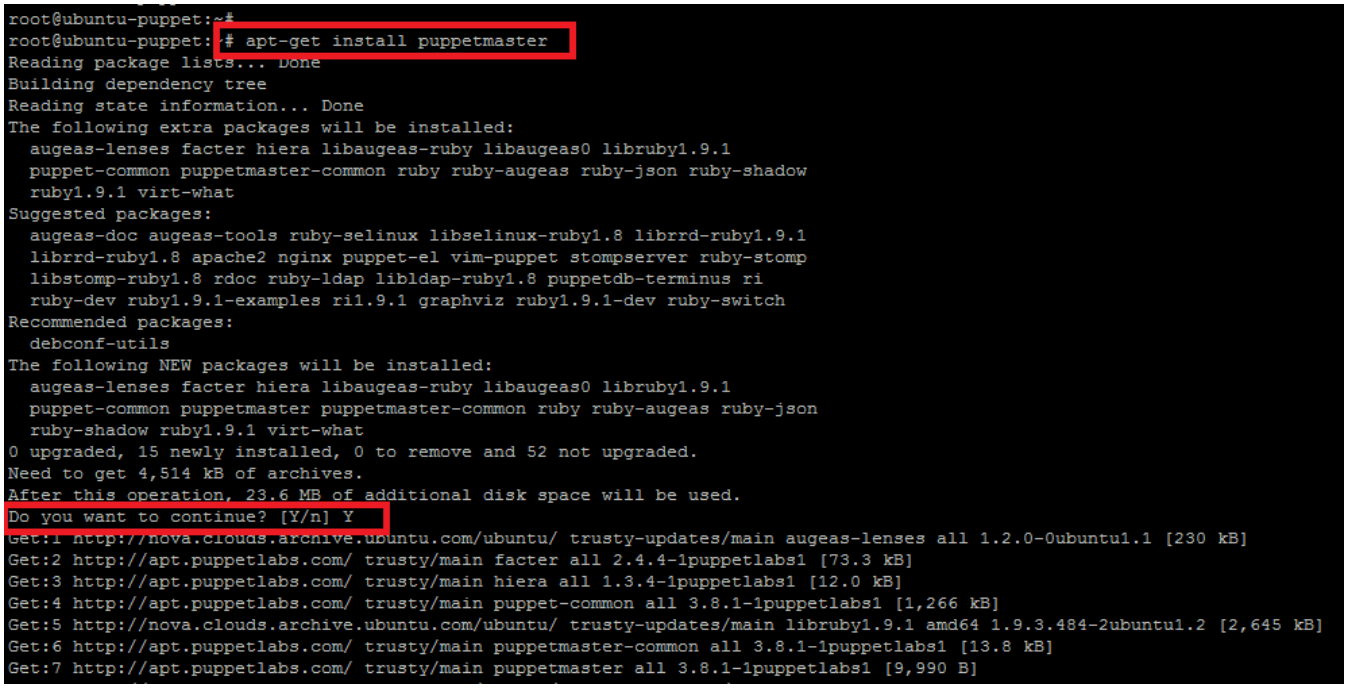


Рисунок 4.1 – Установка Puppetmaster

Наступний крок – створити роль, яка буде встановлювати MySQL базу на віртуальну машину.

В першу чергу треба встановити MySQL Puppet module за допомогою команди «sudo –i puppet module install puppetlabs –mysql --version 3.11».

Тепер нам треба вказати цей класс, щоб Puppet міг використати його для нашої віртуальної машини. Для цього треба написати «include ::mysql::server» в «/etc/puppetlabs/code/environments/production/manifests/site.pp» на машині, де встановленний Puppetmaster (рис. 4.2). У даному випадку, MySQL база буде встановлюватися на кожен хост, та нам це не потрібно. Отож, треба описати застосування модулю так, як приведенно нижче на рисунку. В даному випадку модуль буде застосованний тільки для mysql_server ноди. mysql_server ім’я для віртуальної машини ми дамо пізніше.

```
node mysql_server {
  include ::mysql::server
}
```

Рисунок 4.2 – Застосування mysql модуля для mysql_server ноди

Ті ж самі дії треба зробити, щоб бекенд віртуальна машина змогла підключатися до Puppetmaster та застосовувати свою конфігурацію. На бекенд машини ми встановимо Apache веб – сервер (рис. 4.3).

```
node apache_web_server {

  file { '/mnt/reaxys_app/tomcat':
    ensure => directory,
    mode   => '0755',
    owner  => root,
    group  => root,
  } ->

  file { '/opt/tomcat':
    ensure => link,
    target => '/mnt/reaxys_app/tomcat',
  } ->

  remote_file { '/mnt/reaxys_app/download/apache-tomcat-9.0.16.tar.gz':
    ensure      => present,
    source      => "https://www-eu.apache.org/dist/tomcat/tomcat-9/v9.0.27/bin/apache-tomcat-9.0.27.tar.gz"
  } ->
  exec { 'tar zxvf /mnt/reaxys_app/download/apache-tomcat-9.0.16.tar.gz':
    cwd     => '/opt/tomcat',
    creates => '/opt/tomcat/apache-tomcat-9.0.16/',
    path    => ['/bin', '/usr/bin', '/usr/sbin'],
    onlyif  => "test -e /mnt/reaxys_app/download/apache-tomcat-9.0.16.tar.gz"
  } ->
  file { '/opt/tomcat/apache-tomcat-9.0.16' :
    recurse        => true,
    mode           => '0754',
    owner          => reaxys,
    group          => reaxys,
  }

  exec { 'find /opt/tomcat/ -type f -exec chmod o+r {} \;':
    path    => ['/bin', '/usr/bin', '/usr/sbin']
  } ->

  exec { 'systemctl start tomcat':
    path        => ['/bin', '/usr/bin', '/usr/sbin']
    refreshonly => true
  }
}
```

Рисунок 4.3 – Застосування Puppet коду для установки Apache

## 4.2 Робота з Terraform

Тепер необхідно встановити Puppet agent на наші віртуальні машини.

Але перед цим, ці віртуальні машини треба підняти. Тут ми і використаємо Terraform, як інструмент оркестрації.

Для початку, опишемо хмарний провайдер, в даному випадку AWS (рис. 4.3).

```
provider "aws" {
  version             = "~> 2.30"
  allowed_account_ids = ["<account_id>"]
  region              = "us-east-1"
}
```

Рисунок 4.3 – Опис Terraform провайдеру

```
c:\>terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.30.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.30"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Рисунок 4.4 – Результат виконання команди terraform init

Тепер можна описувати та піднімати наші віртуальні сервера. Нижче представлено Terraform код, який це зробить. Треба звернути увагу на те, що всі параметри були взяті на думку автора. В інших випадках, вони можуть і будуть відрізнятись (рис. 4.5). Так як віртуальних серверів буде декілька (у нашому випадку 7), використаємо Auto Scaling Group та Elastic Load Balancer AWS ресурси.

```
resource "aws_elb" "backend_elb" {
  name            = "backen-elb"
  subnets         = [<subnets_id>]
  security_groups = [<security_groups_id>]
  internal        = true

  listener {
    instance_port     = 8080
    instance_protocol = "http"
    lb_port           = 8080
    lb_protocol       = "http"
  }

  health_check {
    interval            = "10"
    healthy_threshold   = "2"
    unhealthy_threshold = "3"
    target              = "HTTP:8080"
    timeout             = 10
  }

  connection_draining_timeout = 60
  connection_draining         = true
  cross_zone_load_balancing   = true
  idle_timeout                = "4000"

}
```

Рисунок 4.5 – Створення ресурсу Elastic Load Balancer

Після створення Elastic Load Balancer, треба створити Auto Scaling Group та Launch Configuration ресурси та зв’язати іх з уже створеним балансером.

Вираш використовування Auto Scaling Group та Elastic Load Balancer полягає в тому, що машини будуть ідентичні та будуть мати одну конфігурацію (яку ми описали в Puppet маніфесті). Отже, нема необхідності вручну встановлювати веб – сервер на кожну машину. У цьому і полягає вираш автоконфігурації ресурсів веб – проекту.

```
resource "aws_autoscaling_group" "backend" {
  name                      = "backend-asg"
  vpc_zone_identifier       = [<subnets_id>]
  launch_configuration      = "${aws_launch_configuration.backend_lc.id}"
  max_size                  = "7"
  min_size                  = "5"
  desired_capacity          = "7"
  health_check_type         = "ELB"
  load_balancers            = ["${aws_elb.backend_elb.name}"]
  health_check_grace_period = "3000"
}
resource "aws_launch_configuration" "backend-lc" {
  name_prefix     = "backend-lc"
  image_id        = "<centos_ami>"
  instance_type   = "t2.large"
  key_name        = "<key_name>"
  security_groups = [<security_groups_ids>]
  user_data       = "${data.template_file.init.rendered}"

  root_block_device {
    volume_type           = "gp2"
    volume_size           = "10"
    delete_on_termination = true
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

Рисунок 4.6 – Створення ресурсу Auto Scaling Group та Launch Configuration

Для створення віртуальної машини с потім встановленною базою використаємо Terraform ресурс aws_instance.

```
resource "aws_instance" "sql" {
  ami                    = "<centos_ami>"
  instance_type          = "t2.large"
  key_name               = "<key_name>"
  vpc_security_group_ids = [<security_groups_ids>]
  user_data              = "${data.template_file.init.rendered}"
  subnet_id              = [<subnets_id>]

  root_block_device {
    delete_on_termination = true
    volume_size           = "16"
    volume_type           = "gp2"
  }

  ebs_block_device {
    device_name           = "/dev/xvdf"
    volume_type           = "gp2"
    volume_size           = "100"
    delete_on_termination = false
  }
}
```

Рисунок 4.7 – Створення ресурсу aws_instance

Тепер ми маємо 8 запущенних віртуальних машин у хмарі AWS (7 для розгортання додатку та 1 – база даних).

Залишився останній важливий крок. Налаштування Security Groups у хмарі AWS.

Нагадаємо, у нас є балансер, бекенд машини та база даних.

Вхідний трафік на балансер на 443, 80 або 8080 порт має бути відкритий з мережі університету, щоб користувачі могли користуватися ресурсом. Також, вихідний трафік на 8080 порт на бекенд машини повинній бути відкритим для того, щоб балансер зміг помітити машину як «healthy» та перенапрявляти трафік на неї.

Також, бекенд машини повинні мати вихідний трафік на порт 3306 (MySQL сервер порт).

База даних не повинна бути видна з інтернету, але повинна мати доступ до інтернету для того, щоб можна було завантажити оновлення і таке інше. Також, трафік на 8140 порт має бути відкритий с бази даних (щоб можна було ходити на Puppetmaster). Окрім цього, повинен бути відкритий трафік на 3306 порт з віртуальної машини с додатком.

Таблиця 4.1 – Security groups для ресурсів проекту

| | Load Balancer | Backend | MySQL DB |
|---|---|---|---|
| Inbound | 80, 8080, 443 з мережі університету (в залежності від налаштувань) | 80, 8080, 443 з балансеру (в залежності від налаштувань) | 3306 с Security Group бекенд машини |
| Outbound | 80, 8080, 443 на бекенд машини (в залежності від налаштувань) | 80, 8080, 443 для доступу до інтернет ресурсів | 80, 8080, 443 через NAT Gateway |
| | | 3306 до Security Group MySQL DB машини | 8140 на Puppetmaster |
| | | 8140 на Puppetmaster | |

Приведемо приклад user_data сценарію, який буде застосовано до віртуальної машини (в майбутньому MySQL серверу) під час її запуску (рис. 4.8). За допомогою сценарію, віртуальна машина зможе сама підключатися до Puppetmaster та шукати роль, яку ми описали вище та автоматично встановити MySQL сервер. Також, щоб нода змогла інедтифікувати свої налаштування серез інших, треба указати ім’я «mysql_server» у файлі «/etc/hosts» та привести його до виду «localhost = mysql_server».

```
#cloud-config
write_files:
  - path: /etc/puppetlabs/facter/facts.d/instance_classification.yaml
    permissions: '0444'
    owner: root:root
    content: |
      ---
      env:           'production'

package_upgrade: false

runcmd:
  - echo 'Restart rsyslog  to get correct hostname in log entries'
  - systemctl restart rsyslog
  - echo 'Installing Puppet of version 5'
  - rpm -ivh https://yum.puppet.com/puppet5/puppet5-release-el-7.noarch.rpm
  - yum -y install puppet-agent
  - echo 'Configure puppet with proper values'
  - /opt/puppetlabs/bin/puppet config set --section main server ${ puppetmaster_dns }
  - /opt/puppetlabs/bin/puppet config set --section agent environment production
  - /opt/puppetlabs/bin/puppet config set --section agent show_diff true
  - echo 'Remove any certificates remaining from AMI'
  - rm -rf /opt/puppetlabs/puppet/cache/ssl/*
  - rm -rf /etc/puppetlabs/puppet/ssl
  - echo 'Removed unused fact files'
  - find /etc/puppetlabs/facter/facts.d/ -type f ! -name instance_classification.yaml -exec rm {} \;
  - puppet agent -t --waitforcert=60
  - echo "Start up puppet service in case initial run fail"
  - systemctl start puppet
```

Рисунок 4.8 – Приклад user_data сценарію для бази даних

User_data сценарій для бекенд машин використовується ідентичний. У головному маніфесті Puppet була описана конфігурація як для бази, так і для бекенд машин.

Після цих шагів, нам треба лише запустити terraform apply. User_data сценарії зроблять усе інше.

# ВИСНОВКИ

На сьогоднішній день майже неможливо уявити процеси бізнесу, які б не використовували автоматизований підхід в тій чи іншій мірі. Автоматизація процесів дає широкий спектр можливостей фокусування часу та людських ресурсів на розвиток бізнесу, на підняття фактору конкурентноспроможності. Автоматизація дає повний контроль над процесами и вівільняє людські ресурси з буденних завдань.

Впровадження автоматизації не зовсім легкий процес. Досить невелика компанія, з досить недовгостроковими проектами та процесами, без рутинних завдань, не потребує автоматизації, так як це впровадження недоцільно с точки зору витрати як людського ресурсу, так і бізнес ресурсів.

У даній роботі були розглянуті найбільш відомі методології розробки веб-проекту. Також, були наведені приклади використання тої чи іншої методології в залежності від вхідних даних, типу проекту, його масштабу.

Значна увага була приділена методології DevOps та принципам CI/CD, дана характеристика цих підходів а також, формування рекомендацій щодо впровадження DevOps методології та автоматизації до проекту чи бізнесу.

У роботі було представлене та описане поняття оркестрації ресурсів веб – проекту, дана порівняльна характеристика таким інструментам як Terraform та CloudFormation. Проаналізувавши області їх призначення, плюси та мінуси, наявність підтримки, та виходячи з вхідних даних до проекту, був обраний Terraform.

Також, були описані та проаналізовані інструменти автоконфігурації ресурсів веб – проекту: Puppet та Chef. Було освітлено іх призначення, дана порівняльна характеристика, описані методи доставки конфігурації, представлена логіка роботи та ключові компоненти. Виходячи з вхідних даних та вимог проекту, був обраний Puppet як інструмент автоматизованної доставки налаштувань на кінцеві вузли.

Використовуючи вище описані інструменти, був побудованний проект для тестування знань студентів в області системної інженерії. Також, була впроваджена мережна безпека до проекту.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Ещё раз про семь основных методологий разработки [Електронний ресурс] // Habr. – 2015. – Режим доступу до ресурсу: https://habr.com/ru/company/edison/blog/269789/.

2. Тестирование. Фундаментальная теория. Часть 2 — Методологии разработки ПО [Електронний ресурс] // DOU. – 2015. – Режим доступу до ресурсу: https://dou.ua/forums/topic/14015/.

3. Методы разработки веб – приложений и сафтов — каскадные, Agile, Scrum [Електронний ресурс] // Depix. – 2017. – Режим доступу до ресурсу: https://depix.ru/articles/metody_razrabotki_veb_prilozheniy_i_saytov__kaskadnye_agile_scrum.

4. Непрерывная интеграция, непрерывная доставка, непрерывное развертывание [Електронний ресурс] // Habr. – 2017. – Режим доступу до ресурсу: https://habr.com/ru/company/piter/blog/343270/.

5. Методология разработки CI/CD [Електронний ресурс] // ITGlobal. – 2019. – Режим доступу до ресурсу: https://itglobal.com/ru-ru/company/blog/development-method-ci-cd/.

6. AWS Provider [Електронний ресурс] // Terraform Doc. – 2019. – Режим доступу до ресурсу: https://www.terraform.io/docs/providers/aws/index.html.

7. Terraform — Orchestration Automation on Any Cloud [Електронний ресурс] // Medium. – 2019. – Режим доступу до ресурсу: https://medium.com/avmconsulting-blog/terraform-orchestration-automation-on-any-cloud-25afc6dfbc72.

8. Your Infrastructure as Code. CloudFormation Vs Terraform? [Електронний ресурс] // Hackernoon. – 2018. – Режим доступу до ресурсу: https://hackernoon.com/your-infrastructure-as-code-cloudformation-vs-terraform-34ec5fb5f044.

9. Puppet — особенности, плюсы и минусы [Електронний ресурс] // DataArt. – 2014. – Режим доступу до ресурсу: https://dataart.ua/news/puppet-osobennosti-plyusy-i-minusy/.

10. An Overview of Chef Infra [Електронний ресурс] // Chef Doc. – 2019. – Режим доступу до ресурсу: https://docs.chef.io/chef_overview.html.

11. Lysenko M., Kvaratskheliia T. Дослідження загроз та методів забезпечення безпеки хмарних обчислень // Topical issues of the development of modern science. Abstracts of the 3rd International scientific and practical conference. Publishing House “ACCENT”. Sofia, Bulgaria. 2019. Pp. 129-136. URL: http://sci-conf.com.ua.