УДК 004.021
ГРНТИ 50.05.13: ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ. АВТОМАТИЗАЦИЯ ПРОГРАММИРОВАНИЯ.

Vechur O.V.
Candidate of Engineering Sciences, Associate Professor,
Kharkiv National University of Radioelectronics
Shevchenko D.O.
Student,
Kharkiv National University of Radioelectronics

EVALUATION AND OPTIMIZATION OF SOFTWARE PRODUCT INFRASTRUCTURE

Вечур А.О.
Кандидат технических наук, доцент,
Харьковский национальный университет радиоэлектроники
Шевченко Д.А.
Студент,
Харьковский национальный университет радиоэлектроники

ОЦЕНКА И ОПТИМИЗАЦИЯ ИНФРАСТРУКТУРЫ ПРОГРАММНОГО ПРОДУКТА

Abstract. The work is devoted to study methods that can be used for evaluation of software product infrastructure and for its optimization.

Software product infrastructure contains of vide variety of components. There are three main vectors in modern reality: deploy on virtual machines, containers or cloud functions. Software product was deployed in two installations: deployed on virtual machines and cloud functions. For analyze of both infrastructures performance testing was performed. A graph is constructed between the response time and the number of requests.

For infrastructure optimization algorithm was developed. Based on experimental data mathematical model was built and applied to several test installations. After analyzing received data was made a conclusion about method efficiency.

Аннотация. Данная работа посвящена методам, которые возможно использовать для оценки существующей инфраструктуры программного продукта, а также проведению оптимизации этой инфраструктуры.

Инфраструктура программного продукта состоит из множества различных компонентов. Зачастую в современности она идёт по трём направлениям развёртывания: развёртывание на виртуальных машинах, контейнеризация или использование облачных функций. Программный продукт был развёрнут в двух инсталляциях: виртуальные машины и облачные функции. Для анализа обеих инфраструктур было проведено нагрузочное тестирование и построен график времени отклика от количества запросов.

Был разработан алгоритм для оптимизации инфраструктуры. На основании полученных в ходе эксперимента данных была построена математическая модель и применена к ещё нескольким тестовым инсталляциям. Основываясь на полученных данных были сделаны выводы относительно эффективности данного подхода.

Key words: scalability, usability, cost-efficiency.
Ключевые слова: масштабируемость, юзабилити, экономическая эффективность.

Введение: Стоимость программного обеспечения состоит из множества факторов: количество затраченых человеко-часов, уникальности программного решения, инфраструктуры необходимой для его работы. Первый фактор упирается в управленческие методы применяемые на проекте. Второй фактор – упирается в существующий рынок. Третий же фактор зависит как от системных требований продукта так и от нефункциональных требований.

Затраты на инфраструктуру являются постоянной статьёй расходов для продукта. Возможность оптимизации инфраструктуры с целью уменьшения расходов на её содержание без утраты производительности самого приложения является важной задачей в современном мире.

В проектных командах на данный момент существует две крайности в отношении инфраструктуры. Первый подход заключается в использовании максимально производительной инфраструктуры, которую позволяет использовать бюджет. Недостатком такого метода является то, что доступные мощности не используются, а бюджет просто уходит владельцам облачных сервисов или на счета за электричество. Второй подход напротив идёт по пути минимальной инфраструктуры и её расширению по мере развития продукта. Этот подход более напоминает рациональный, но ведёт к тому, что на изменения инфраструктуры постоянно затрачивается время специалистов. Таким образом бюджет тратится не

на полезные работы, а на постоянное изменение ресурсов.

Постановка проблемы: В результате работы необходимо исследовать существующие методы анализа инфраструктуры и способы её оптимизации.

Необходимо проанализировать универсальность выбранного метода анализа, а также, разобрав существующие подходы к управлению инфраструктурой, разработать метод для оптимизации существующей инфраструктуры и исследовать его эффективность на нескольких уже существующих программных продуктах. Схему метода оптимизации можно увидеть на рисунке 1.1.

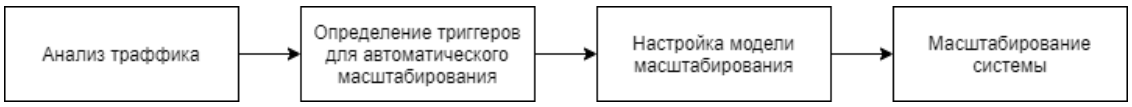

Рисунок 1.1 – Схема алгоритма оптимизации

Анализ последних исследований и публикаций: При исследовании методов анализа инфраструктуры было выделено несколько подходов.

Первый подход анализирует инфраструктуру относительно кода программного продукта: эксперт анализирует требования программного продукта, существующую программную реализацию и на основании этих вещей, а также своего экспертного мнения выдвигает вердикт касательно необходимой инфраструктуры в случае первичной развёртки приложения или оптимизации его в случай анализа уже развёрнутого приложения [1].

Недостатками данного метода является субъективность опыта эксперта, неоднородность описания требований и недостатки программной реализации. Неоднородность описания требований продукта приводит к сложности их понимания, также это требует наличия какой-либо проектной документации, что на практике не всегда соответствует реальности: часто реализация функциональных требований приложения опережает написание технической документации, а в связи с применением Agile-методологий приводит к тому, что в погоне за предоставлением новой функциональности клиенту приходится жертвовать удобством поддержания приложения. Недостатки программной реализации проекта ведут к тому, что при одинаковых требованиях возможно превышение необходимого числа количества коммуникаций межд частями приложения: часто в приложениях встречаются проблемы в коммуникации сервера приложения с сервером базы данных из-за неоптимального количества запросов в базу данных или неоптимальных запросов. Такие недоработки в кодовой базе приложения приводят к неожиданным просадкам в производительности, а как следствие к повышенным требованиям к инфраструктуре. Субъективность эксперта же может запросто привести заказчика в ситуацию, когда он поддерживает инфраструктуру более мощную, чем необходимо, в связи с тем, что эксперт решил перестраховаться и заложить запас по производительности больший, чем необходимо. Это также связано с тем, что зачастую компании в договорах акцентируют внимание на финансовой стороне, а не инфраструктурной.

Второй подход за свою основу берёт работу с программным продуктом как с «чёрным ящиком»: человек, анализирующий программный продукт не знает ничего про его устройство, а анализирует только входящие и исходящие данные [1]. Одним из таких методов можно считать нагрузочное тестирование. Нагрузочное тестирование составляет собой выполнение ряда функциональных тестов относительно уже развёрнутого приложения и анализа таких составляющих как время отклика и корректность обработки запроса. Также во время проведения нагрузочного тестирования при мониторинге инфраструктуры можно выявить не только то, что при увеличении количества запросов увеличивается время отклика, но и то, что с какого-то момента запросы в базу данных перестают обрабатываться или то, что приложение поймало deadlock.

Среди плюсов данного метода для анализа инфраструктуры можно выделить его универсальность: при наличии функциональных требований к программному продукту возможно провести нагрузочное тестирование и найти слабые места работы приложения.

Недостатком же данного подхода является тот факт, что тестирование укажет вам на проблемы с инфраструктурой, но не с конкретной реализацией функциональных требований. Стоит заметить, что для мониторинга также необходимо привлечения опытного эксперта, который сможет найти проблемы, которые приводят к появлению тех или иных проблем.

Предоставители облачных сервисов уже имеют сервисы помогающие оптимизировать инфраструктуру существующих приложений. Например Azure Advisor анализирует текущую конфигурацию и телеметрию приложения и на их основании предоставляет советы для оптимизации инфраструктуры. Такой подход позволяет сделать выводы относительно эффективности использования текущей инфраструктуры, но игнорирует функциональные и нефункциональные требования. AWS Trusted Advisor применяет аналогичные подходы для анализа инфраструктуры. Также оба предоставителя

облачных сервисов предоставляют поддержку от technical account manager для более детальной оптимизации и экономии.

Выделение нерешенных ранее частей общей проблемы: Таким образом необходимо разработать метод позволяющий на основании функциональных требований приложений предоставлять советы по оптимизации инфраструктуры или модифицировать текущие методы оптимизации для того, чтобы учесть функциональные требования.

Цель статьи: Предоставить данные полученные в ходе экспериментальной проверки инфраструктуры программных продуктов. Предоставить данные полученные в ходе анализа предложенного метода оптимизации инфраструктуры. Проанализировать результаты оптимизации и предоставить результаты данного исследования.

Оценка инфраструктуры программного продукта: Для анализа инфраструктуры путём нагрузочного тестирования было выбрано два приложения: развёрнутое на виртуальных машинах и мощности которого используют облачные функции. В ходе подготовки к эксперименту по одному набору функциональных требований было реализовано два приложения. Это связано с тем, что в современных реалиях возможности развёртывания приложения напрямую зависят от его реализации. Таким образом было получено java-приложения с базой данных MongoDB предполагающее развёртывание на виртуальной машине и набор функций AWS Lambda на языке python с использованием базы данных DynamoDB.

Первое приложение было установленно на двух виртуальных машинах t2.mini в облаке AWS. Второе было также развёрнуто в облаке AWS. На этом этапе стоит отдельно остановиться и рассмотреть разницу между подходами к развёртыванию приложений.

Развёртывания приложения на виртуальных машинах подразумевает под собой поддержание некоторого количества виртуальных серверов с копиями исходного кода приложения, развёрнутой базой данных и её репликами. Такой подход является стандартом в мире программных продуктов. Преимущества данного подхода заключаются в том, что данный подход проверен временем и большинство специалистов умеют с ним работать. Недостатками же такого подхода являются сложности масштабирования. Для масштабирования таких приложений существует два подхода: горизонтальное и вертикальное масштабирование. Вертикальное масштабирование полагается на увеличение объёма ресурсов доступных приложению за счёт увеличения ресурсов на виртуальной машине. Горизонтальное же масштабирование полагается на балансировку нагрузки и увеличение ресурсов приложения засчёт увеличения количества виртуальных машин. Вертикальное маштабирование имеет свои ограничения со стороны предоставителя облачных сервисов, а горизонтальное масштабирование требует продумывания архитектуры, отсутствие которой на ранних этапах может привести к усложнению рефакторинга и работы приложения [1].

Подход облачных функций напротив в свою основу берёт простоту масштабирования. Для выполнения каждого запроса происходит вызов «функции» - независимого фрагмента кода, который выполняет только требуемую задачу. Разворачивается контейнер в котором выполняется код. На каждый запрос создаётся новый контейнер или используется свободный. Это приводит нас к тому, что помимо выполнения самого кода приложения на большую часть запросов дополнительно добавляется время создания контейнера. В облачных сервисах это называется временем холодного старта. Время холодного старта зависит от размера функции, а также от её языка [2].

Во время подготовки к следующей фазе исследования был проведён эксперимент для выяснения этого времени. Результаты можно увидеть в таблице 1.1. Выяснилось, что для java время холодного старта в среднем составляет 500 миллисекунд, а для python 200 миллисекунд. При развёртывании обычного сервера приложений таких проблем не наблюдается.

Таким образом разобрав основные различия между архитектурой наших решений приступим к сбору данных. Для сбора данных на основании функциональных требований был построен ряд нагрузочных тестов, количество запросов в секунду постепенно увеличивалось от 100 до 1500. Результаты можно увидеть на графике 1.1 и в таблице 1.2.

Таблица 1.1

Время холодного старта облачной функции

Количество выделенной оперативной памяти	128 Mb	1024 Mb	3008 Mb
Java	650 мс	567 мс	480 мс
Python	253 мс	223 мс	189 мс
Node JS	199 мс	212 мс	210 мс
Golang	350 мс	320 мс	300 мс

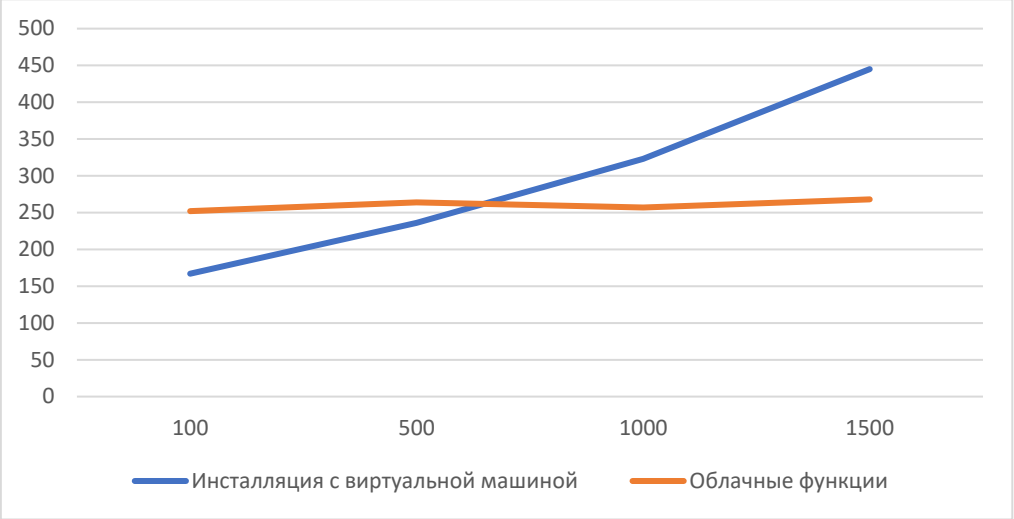

График 1.1 – Максимальное время отклика приложения (мс) от числа запросов

Таблица 1.2

Максимальное время отклика приложения

Число запросов	100	500	1000	1500
Время отклика монолита, мс	167	236	323	445
Время отклика serverless, мс	252	264	257	268

При анализе полученных данных можно заметить, что с ростом количества запросов время отклика облачных функций практически не изменялось и держалось в приделах 250 миллисекунд, что соответствует 200 миллисекундам холодного старта и времени выполнения самой функции. Регулируя количество одновременно живущих контейнеров можно найти оптимальную комбинацию по производительности.

Производительной же инсталляции на виртуальной машине оставляет желать лучшего. Проанализировав данные был сделан вывод, что проблемы с производительностью были связаны с тем, что Tomcat может обрабатывать около 350 транзакций в секунду [3].

В целом мы можем наблюдать, что примерно на 350 запросах инсталляция на виртуальной машине достигла границы в 200 миллисекунд на ответ на запрос. В целом, для динамической составляющей сайта это нормальное время отклика, но с ростом трафика время отклика достигло достаточно неудобных для конечного пользователя чисел [4].

Оптимизация инфраструктуры программного продукта: проанализировав данные полученные в ходе нагрузочного тестирования был построен метод для оптимизации инфраструктуры развёрнутой на виртуальных машинах. Суть метода заключается в том, что нагрузка реального приложения приводится к математической модели и анализируется после чего на основании этих данных мы производим автоматическое увеличение или уменьшение мощностей в течении дня. Для анализа возьмём трафик одного небольшого приложения. Сервер развёрнут в AWS на виртуальной машине c5.large. Увидеть его нагрузку можно на графике 1.2 и в таблице 1.3.

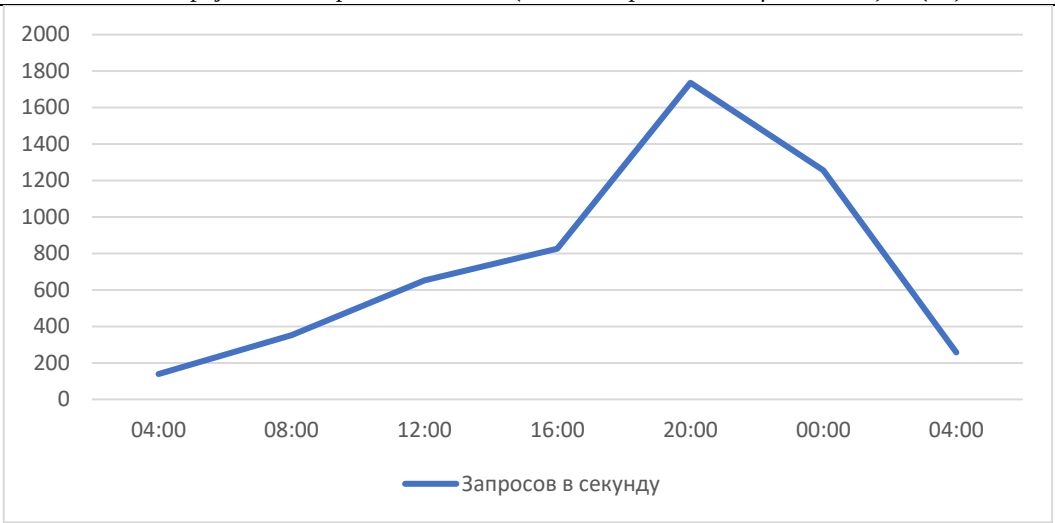

График 1.2 – Количество запросов в секунду в течении суток анализируемого продукта

Таблица 1.3

Суточная нагрузка анализируемого продукта

Время суток	4:00	6:00	9:00	12:00	15:00	18:00	21:00	00:00	2:00
Запросов в секунду	139	226	421	653	763	1236	1672	1257	805

Поделим данные про нагрузку на промежутки и проведём сглаживание методом скользящего среднего по формуле 1.1.

$$SMA = \frac{\sum_{i=1}^{n} P_i}{n} \quad (1.1)$$

где P_i – трафик,
n – основной параметр – длинна сглаживания или период SMA

После этого проведём аппроксимацию каждого из участков при помощи поллиномиальной регрессии и найдя производные от полученных функций получим скорость с которой увеличивается количество запросов в моменты, когда время обработки запроса пересекает черту в 200 миллисекунд для одного сервера [5].

В рамках эксперимента было определено, что это значение в нашем случае равно 350 запросам. Найдём значения производной для всех множеств значений с шагом в 350 запросов. Так мы получили данные по скорости роста количества запросов в секунду, которые будем использовать в качестве триггера для изменения конфигурации.

Настроим load-balancer таким образом, чтобы при увеличении количества запросов он поднимал новые виртуальные машины, а при уменьшении – уменьшал их количество. Применим данные подход к существующему продукту и развернём несколько виртуальных машин для обеспечения работоспособности приложения. Результаты можно увидеть на графике 1.3 и таблице 1.4.

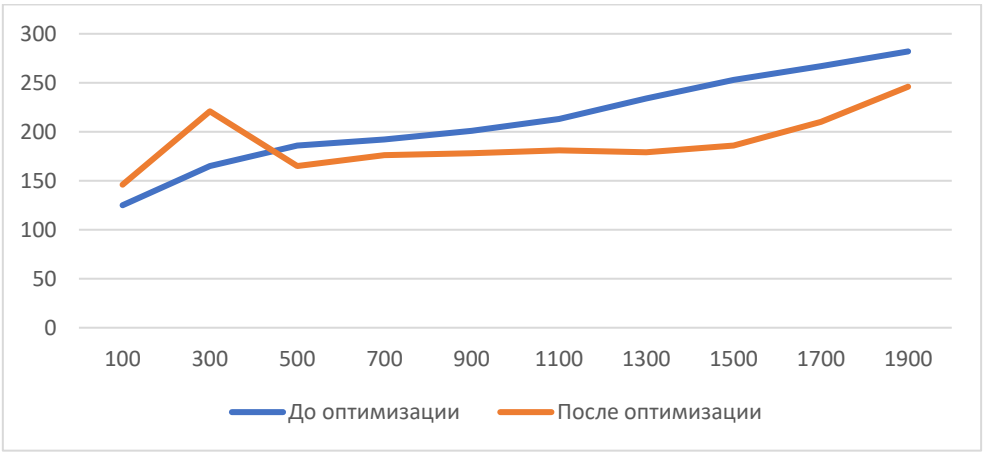

График 1.3 – Максимальное время отклика в миллисекундах от количества запросов в секунду до и после оптимизации

Проанализировав полученные данные мы видим, что с увеличением роста запросов максимальное время отклика у приложения до оптимизации постепенно возрастает. После оптимизации мы можем заметить, что на малом числе запросов время их обработки больше, чем у инсталляции до оптимизации. Это может быть обусловленно тем, что для оптимизированной инфраструктуры были применены виртуальные машины t2.small, которые по своим характеристикам проигрывают c5.large, которая использовалась в первичной конфигурации. Пик на 300 запросах обусловлен тем, что правило применённое для поднятия новых виртуальных машин ещё не отработало и второй сервер для обработки запросов не был поднят. В дальнейшем динамически поднятые виртуальные машины позволяют держать время отклика в приемлимых границах.

Таблица 1.4

Результаты оптимизации

Количество запросов в секунду	100	300	500	700	900	1100	1300	1500	1700	1900
Время отклика до оптимизации	125	165	186	192	201	213	234	253	267	282
Время отклика после оптимизации	146	221	165	176	178	181	179	186	210	246

Увеличение времени отклика после 1500 запросов свидетельствует о том, что данный триггер плохо реагирует на непредсказуемые нагрузки, которые превышают данные на основании которых данный триггер строился, что ставит под сомнение его применимость и вынуждает к анализу других триггеров для изменения инфраструктуры.

С финансовой точки зрения исходная инфраструктура стоила 72 доллара в месяц: считаем, что сервер c5.large был активен каждый день в течении месяца при стоимость 0,097 долларов за час. Инфраструктура после оптимизации стоит 65 долларов: в ходе эксперимента получилось, что одна виртуальная машина была активна 24 часа в течении месяц, ещё две – 16 часов и ещё две 8 часов при цене в 0,027 долларов за одну виртуальную машину в час. Это связано с тем, что облачные сервисы поощряют эффективное использование вычислительных ресурсов за счёт своих ценовых политик. Использование большого количества временных виртуальных машин (в нашем эксперименте от 1 до 5) оказалось не только выгоднее с точки зрения пользовательской удобности, но и с финансовой точки зрения. Проведя дальнейший анализ было выяснено, что при увеличении стоимости виртуальной машины для оптимизированной инфраструктуры на 1 цент привело бы к проигрышу по финансовой составляющей, а именно то, что инфраструктура обходилась бы нам в 86 долларов.

Выводы и предложения: В ходе проведённой работы был проанализирован предложенный метод оптимизации инфраструктуры. Полученные данные свидетельствую о том, что метод применим и даёт возможность поддерживать инфраструктуру в положении при котором приложение будет без затруднений использоваться пользователями. То, что финансово оптимизированная инфраструктура оказалась выгоднее, является сопутствующим результатом и никак не коррелирует ни с чем, кроме ценовых политик предоставителей облачных сервисов. Предложенный триггер, а именно скорость роста количества запросов, необходимо сравнить с другими возможными, а именно с количеством запросов в секунду и максимальным временем отклика. Существует вероятность, что триггеры на количество запросов и время отклика будут эффективнее для более непредсказуемой нагрузки.

Список литературы:

1. Цваліна К., Абрамс Б. Інфраструктура програмних проектів [Текст] / Цваліна К., Абрамс Б. М. : Вильямс, 2011.-418 с.

2. Nathan Malishev. AWS Lambda Cold Start Language Comparisons, 2019 edition. URL: https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244

3. Peter Lin. So You Want High Performance / URL: https://tomcat.apache.org/articles/performance.pdf

4. Jordan Kasteler. What Is Time To First Byte, And How To Improve It/ URL: https://www.searchenginepeople.com/blog/16081-time-to-first-byte-seo.html

5. Michael A Murphy, Brandon Kagey, Michael Fenn, and Sebastien Goasguen. Dynamic provisioning of virtual organization clusters. In Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 364–371. IEEE Computer Society, 2009.