СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы. В настоящее время вычислительные мощности многих предприятий организованы с использованием облачных платформ: в публичных или частных облачных системах [1, 2]. Это обусловлено не только низкой стоимостью использования таких решений, но также быстротой получения требуемых ресурсов [3]. Для предприятий малого и среднего размера дополнительным преимуществом является возможность оплаты вычислительных ресурсов не по общему количеству, а по факту их использования. [4].

Помимо существенных достоинств, использование такого подхода обладает также следующими недостатками. Информационная безопасность инфраструктуры и конфиденциальность хранимых там данных могут быть под угрозой вследствие ошибки поставщика облачных услуг [5]. Кроме того, аппаратное обеспечение облачной инфраструктуры может выйти из строя в результате сбоя, вследствие чего предприятие теряет доступ ко всей своей инфраструктуре, что приводит к убыткам [6]. Для нивелирования последнего недостатка обычно прибегают к использованию нескольких облачных систем.

Кроме этого, у таких компаний может возникнуть необходимость мигрировать всю инфраструктуру с одной облачной платформы на другую [7]. Такая необходимость может быть вызвана рядом причин, среди которых выделяют:

- ожидаемое снижение стоимости облачной инфраструктуры;
- возросшие потребности в вычислительных мощностях, которые не могут быть удовлетворены текущими решениями.

Для таких случаев активно исследуются и разрабатываются программные модули, применение которых не привязывает компанию к конкретной реализации облачной платформы, а позволяет миграцию инфраструктуры в другое облако без доработок программного обеспечения.

С учётом тенденций, описанных выше, сервис управления виртуализированной инфраструктурой является актуальным и востребованным реше-

нием.

Цель и задачи исследования. Целью работы является упрощение реализации компонентов управления ресурсами систем с различными технологиями виртуализации инфраструктуры. Для достижения поставленной цели необходимо решить ряд **задач**.

1. Проанализировать предметную область и определить типовые подходы к управлению виртуализированной инфраструктурой.
2. Выявить недостатки существующих решений; сформулировать новый подход к управлению виртуализированной инфраструктурой.
3. На основе сформулированного подхода разработать сервис управления ресурсами систем с различными технологиями виртуализации инфраструктуры.

Область исследования. Проведённое исследование виртуализированной инфраструктуры полностью соответствует специальности «Программная инженерия», а содержание выпускной квалификационной работы — техническому заданию.

Объектом исследования являются платформы виртуализации инфраструктуры.

Предметом исследования является метод управления виртуализированной инфраструктурой с поддержкой нескольких существующих реализаций платформ виртуализации инфраструктуры.

Методологическую основу исследования составляют метод обобщения и эксперимент.

Практическая значимость исследования. Разработанный сервис управления может быть использован как в новых, так и в существующих облачных системах. Такой сервис позволяет осуществлять мониторинг и контроль разнородных платформ облачных вычислений при использовании единого унифицированного программного интерфейса (API). Это позволяет использовать одни и те же системы мониторинга и управления с разными платфор-

мами облачных вычислений, а так же позволяет заменять одну виртуализированную инфраструктуру на другую без доработок в существующих программных модулях.

Объем и структура работы. Выпускная квалификационная работа содержит 69 страниц машинописного текста, 12 рисунков, 17 таблиц и список литературы, включающий 33 источника. Структурно работа состоит из введения, трех частей и заключения. Во введении обоснована актуальность исследования, определены цели и задачи, объект и предмет исследования. Первая часть посвящена теоретическим основам исследования, обзору предметной области и постановке задачи. Во второй части сформулирован и представлен список требований к реализации решения. Третья часть исследования посвящена описанию разработанного решения. В заключении приведены основные результаты работы.

1 УПРАВЛЕНИЕ ВИРТУАЛИЗИРОВАННОЙ ИНФРАСТРУКТУРОЙ

1.1 Облачные платформы

Для организации своей вычислительной инфраструктуры компании в настоящее время всё чаще выбирают решения, основанные на так называемых облачных платформах [8]. Такие решения имеют ряд достоинств, среди которых для данной работы наиболее важно следующее: загрузка аппаратного обеспечения может быть выше при организации инфраструктуры в облачное решение, чем при традиционной организации [9]. Загрузка аппаратного обеспечения показывает какое количество аппаратных ресурсов используется относительно общего их количества [10]. Чем выше загрузка, тем меньше аппаратных ресурсов работает вхолостую или простаивает. Как холостая работа, так и простой ведут к прямым убыткам, в связи с чем более высокая загрузка обеспечивает финансовую выгоду [11].

В облачных инфраструктурах повышение загрузки достигается с помощью разделения вычислительных ресурсов одного и того же сервера между несколькими задачами [12]. В традиционной инфраструктуре такое разделение не применяется по причинам безопасности, а также из-за возможного захвата всех аппаратных ресурсов одной задачей и последующим простаиванием других [13]. Таким образом, в традиционной архитектуре для выполнения того же объёма работы нужно больше аппаратных ресурсов, чем в облачной, как проиллюстрировано на рис. 1.

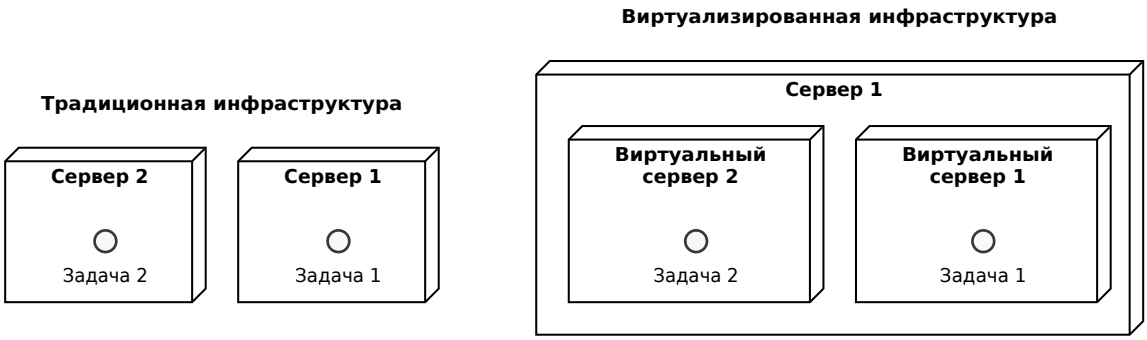

Рис. 1: Сравнение виртуализированной и традиционной организации инфраструктуры

1.2 Виртуализированная инфраструктура

”Виртуализированная инфраструктура” — специальный обобщающий термин, введённый в этой работе для объединения двух способов организации облачной инфраструктуры:

- при помощи контейнерной организации приложений;
- с использованием виртуальных машин для организации приложений.

Виртуализированной инфраструктурой в данной работе называется такая инфраструктура, которая:

1. Управляет вычислительными мощностями.
2. Предоставляет возможность запускать программные приложения на вычислительных мощностях этой инфраструктуры.
3. Позволяет выделять каждому из запущенных программных приложений строго определённое количество ресурсов. Количество ресурсов при этом необязательно кратно количеству аппаратных ресурсов в этой инфраструктуре.
4. Предоставляет возможность изменять количество выделенных ресурсов в случае, если потребности приложения изменились.
5. Позволяет запускать дополнительные программные приложения на свободных вычислительных мощностях.

1.3 Масштабирование

В облачных решениях захват всех аппаратных ресурсов невозможен вследствие ограниченного выделения ресурсов каждой из задач [14]. В общем случае, это способы ограничения ресурсов делятся на два типа [15]:

- решения на базе виртуальных машин;
- решения на базе контейнеров.

В обоих подходах количество ресурсов, доступных задаче, управляются с помощью масштабирования. Различают два типа масштабирования.

1. Горизонтальное.

   Горизонтальным называется масштабирование, при котором изменяется количество виртуальных машин или контейнеров, доступных задаче. Количество ресурсов, доступных каждой из виртуальных машин или каждому из контейнеров, при этом остаётся неизменным.

2. Вертикальное.

   Вертикальным масштабированием называется такое масштабирование, при котором изменяется количество ресурсов, выделенных каждой из виртуальных машин или контейнеров. Их количество при этом остаётся неизменным.

Разница между ними проиллюстрирована на рис. 2.

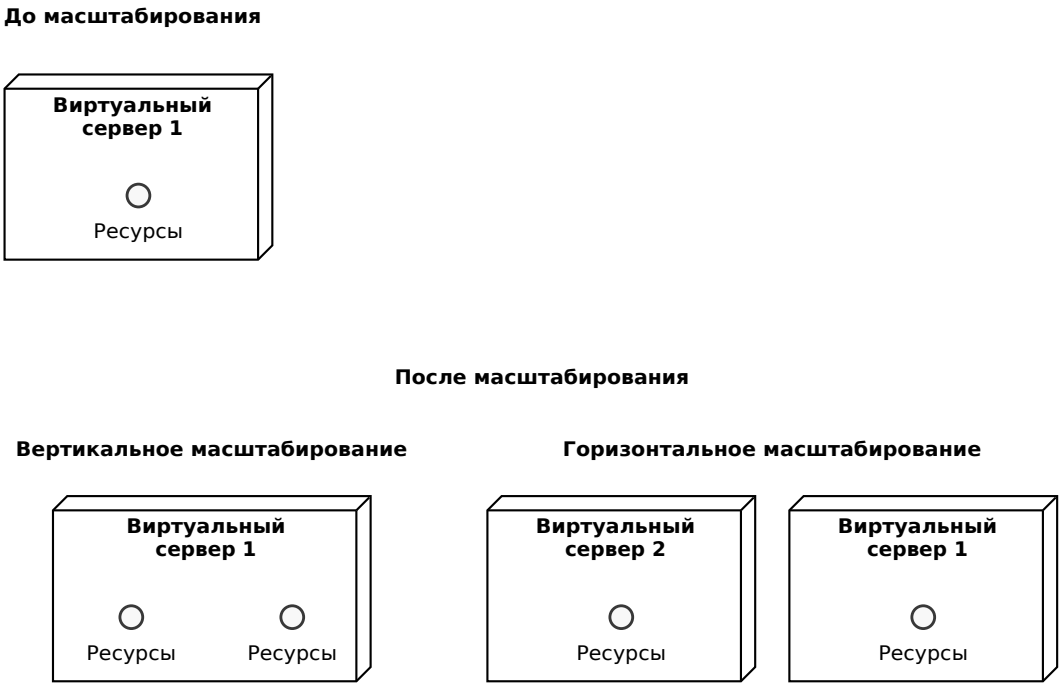

Рис. 2: Сравнение горизонтального и вертикального масштабирования

Горизонтальное масштабирование в настоящее время применяется чаще [16] по ряду причин, неполный перечень которых представлен далее.

1. Возможность выделения на одно приложение (задачу) несколько серверов.

   Это достигается при помощи запуска виртуальных машин или контейнеров на каждом из из серверов [17].

2. Более высокая загрузка аппаратного обеспечения.

   В случае отсутствия нагрузки на приложение, будет простаивать только то количество аппаратных ресурсов, которое выделено одной виртуальной машине или контейнеру приложения, другие ресурсы при этом будут высвобождены [18]. В случае горизонтального масштабирования есть возможность выделить одной виртуальной машине или контейнеру минимально необходимое количество ресурсов для обработки минимальной нагрузки на приложение В то же время в случае вертикального масштабирования, при выделении ресурсов, достаточных для обработ-

ки пиковой нагрузки, будет наблюдаться простой в обычном режиме работы [19].

Таким образом, существует задача масштабирования приложений. Для решения этой задачи существуют технологии, имеющие общее название ”автомасштабирование” [20]. При помощи таких технологий осуществляется автоматическое масштабирование в зависимости от текущей нагрузки на приложение или других факторов.

Во многие платформы облачных вычислений сервисы автомасштабирования уже встроены [21], однако существуют сервисы автомасштабирования, являющиеся внешними по отношению к платформе, как показано на рис. 3.

Рис. 3: Внешний и внутренний сервисы автомасштабирования

Причины появления таких сервисов могут быть разными, некоторые примеры приведены далее.

1. Отсутствие решений автомасштабирования в используемой облачной платформе [22].

2. Недостаточная эффективность встроенного сервиса автомасштабирования.

   Это может быть обусловлено спецификой конкретного приложения, которую встроенный сервис не учитывает, так как чаще всего они спроектированы без учёта специфики конкретных приложений. [23].

На практике зачастую внешние сервисы автомасштабирования спроектированы и разработаны под одну конкретную платформу облачных вычислений [24]. При этом взаимодействие с другими платформами требует доработки реализации сервиса автомасштабирования как показано на рис. 4. Это обусловлено значительной разницей в предоставляемых программных интерфейсах (API) разными платформами облачных вычислений [25].

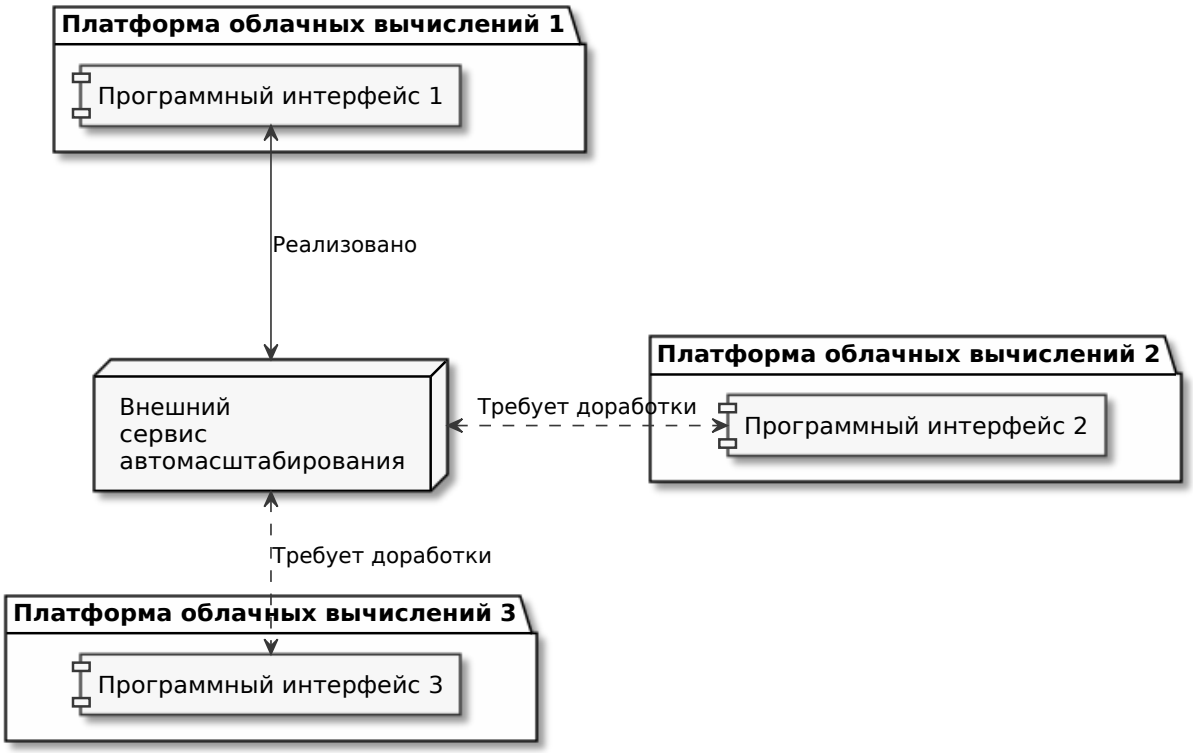

Рис. 4: Проблема взаимодействия с разными платформами

1.4 Типовые подходы к управлению виртуализированной инфраструктурой

Далее в этой главе будет представлен обзор научных работ, статей и патентов, посвящённых решению описанной проблемы.

Программируемый инфраструктурный шлюз для обеспечения работы гибридных облачных сервисов

Программируемый инфраструктурный шлюз, используемый для обеспечения работы гибридных облачных сервисов в сетевом окружении [26] разра-

ботан с целью обеспечения организации взаимодействия частей гибридного облачного окружения.

Под гибридным облачным окружением в патенте понимается облачное окружение, где часть вычислительных ресурсов являются внутренними, а управление ими осуществляется самой организацией (или частным лицом). При этом другая часть вычислительных ресурсов управляется сторонней организацией (или сторонним частным лицом) и является внешней. В качестве примера приводится использование публичного облачного сервиса Amazon S3™ для хранения архивных данных компании и внутренней инфраструктуры организации для хранения корпоративных данных по текущим операциям.

Инфраструктурный шлюз, как и сервис управления, должен решать задачу предоставления унифицированного интерфейса к различным реализациям облачных окружений. Отсюда следует, что разработка описываемого инфраструктурного шлюза, как и сервиса управления виртуализированной инфраструктурой, предполагает разработку так называемых облачных адаптеров. Под облачным адаптером здесь понимается программный модуль, управляющий передачей данных и команд из частного облака (внутренних ресурсов) в публичную (внешнюю) инфраструктуру.

Авторы программируемого инфраструктурного шлюза не решали задачу мониторинга используемых приложениями ресурсов в реальном времени. Сервис управления, в свою очередь, должен решать такую задачу.

Сервис управления виртуализированной инфраструктурой, в отличие от описываемого шлюза, должен работать не только с гибридными облачными сервисами, но и в инфраструктуре, состоящей только из частного облака или только из публичного. При этом сервис управления не позволяет передавать данные между двумя частями инфраструктуры.

Метод оркестровки гибридными облачными сервисами

Метод оркестровки гибридными облачными сервисами [27] позволяет при помощи одного унифицированного интерфейса взаимодействовать с гибридным облачным окружением, состоящем из нескольких разнородных плат-

форм облачных вычислений. Таким образом, этот метод, как и сервис управления виртуализированной инфраструктурой, предоставляет единый унифицированный интерфейс, но в случае сервиса управления интерфейс скрывает всегда одну платформу облачных вычислений. Метод оркестровки предполагает создание интерфейса, скрывающего несколько разнородных реализаций виртуализированной инфраструктуры, работающих одновременно и формирующих таким образом единое гибридное облачное окружение.

С учётом вышесказанного, можно заметить, что и метод оркестровки, и сервис управления подразумевают возможность работы с различными реализациями виртуализированной инфраструктуры. Сюда включаются сразу две задачи:

1. Сбор статистики использования ресурсов.
2. Изменение количества выделенных приложению ресурсов.

При этом метод оркестровки не ставит перед собой именно эти задачи, он ставит перед собой задачу предоставления ”всеобъемлющего” API, который, будет решать эти задачи, если их решает каждая из платформ в гибридном облачном окружении.

Задача предоставления возможности адаптации дополнительной реализации виртуализированной инфраструктуры к единому унифицированному API ставится как перед методом оркестровки, так и перед сервисом управления виртуализированной инфраструктурой.

Программная архитектура сервиса мониторинга гибридных платформ облачных вычислений

Программная архитектура сервиса мониторинга гибридных платформ облачных вычислений [28] была спроектирована для решения двух основных задач:

- Мониторинг в режиме реального времени. Это может быть использовано в целях так называемого превентивного или планового обслуживания, то есть для проведения ряда мероприятий, необходимого для из-

бежания неожиданных сбоев.

- Постфактум мониторинг состояния системы. С помощью такого мониторинга можно обнаружить проблемы в работе системы или даже проблемы, связанные с безопасностью системы.

Гибридность целевой платформы (платформы под мониторингом) заключается в том, что она может быть заменена на другую без необходимости вносить изменения в сам сервис мониторинга. Таким образом, задача предоставления унифицированного интерфейса решается как сервисом мониторинга, так и сервисом управления.

Сервис управления виртуализированной инфраструктурой, как и сервис мониторинга, решает задачу сбора статистики в реальном времени. При этом сервис мониторинга решает дополнительную задачу постфактум мониторинга, но такой задачи не стоит перед сервисом управления.

Перед сервисом мониторинга не стоит задачи предоставления интерфейса, который позволяет изменять количество выделенных каждому из приложений ресурсов. Сервис управления, в свою очередь, должен решать такую задачу и позволять не только узнать сколько ресурсов используется сейчас, но и высвобождать неиспользуемые ресурсы, а также выделять дополнительные.

Портативный сервис автомасштабирования для управления различными масштабируемыми облачными платоформами

Портативный сервис автомасштабирования для управления различными масштабируемыми облачными платформами [20] был разработан для автоматического масштабирования разнородных реализаций виртуализированной инфраструктуры. Этот сервис не предоставляет внешний программный интерфейс, только интерфейс для администратора системы с доступом к параметрам сервиса, а так же собранной статистике.

Рассматриваемый сервис автомасштабирования является примером прикладного назначения сервиса управления, потому что сервис управления решает задачи двух модулей сервиса автомасштабирования:

1. Provision Manager - модуль, позволяющий изменять количество доступных каждому запущенному приложению ресурсов.
2. Monitoring Engine - модуль, предоставляющий статистику использования ресурсов, собранную платформой облачных вычислений.

Таким образом, сервис автомасштабирования решает задачи, поставленные перед сервисом управления, но не предоставляет внешний программный интерфейс. Напротив, сервис автомасштабирования использует унифицированный интерфейс самостоятельно и не позволяет применять другие алгоритмы автомасштабирования, кроме заложенных в него.

Федерация облачных сервисов с использованием прокси-слоя виртуального API в распределённом облачном окружении

Известен способ объединения одной из реализаций виртуализированной инфраструктуры OpenStack в федерацию с другими реализациями с помощью прокси-слоя виртуального API, который представляет другие облака как псевдо-зону доступности в OpenStack [29]. Используя этот способ, можно создать систему из нескольких облачных платформ, объединённую в OpenStack. Это позволяет использовать модули управления, мониторинга и так далее из OpenStack с другими платформами без необходимости их изменения.

Таким образом, эта работа решает все задачи сервиса управления виртуализированной инфраструктурой:

1. задача предоставления единого интерфейса решена предоставлением интерфейса OpenStack;
2. задача предоставления возможности добавлять дополнительные платформы облачных вычислений решена при помощи псевдо-зон доступности в OpenStack, куда можно добавить необходимую облачную платформу;
3. задача сбора статистики решена с помощью сбора статистики в OpenStack;
4. задача изменения количества выделенных приложению ресурсов так же решается с помощью модуля управления OpenStack.

Как видно из этого перечисления, все задачи решены, но решены при помощи использования дополнительного облачного окружения, которое будет избыточно в той прикладной задаче, для которой предлагается использовать сервис управления виртуализированной инфраструктуры. Дополнительная облачная платформа внесёт существенные задержки и сложность в итоговую систему, а так же создаст дополнительную нагрузку на вычислительные мощности. Иными словами, этот способ рекомендуется использовать только в ситуациях, когда OpenStack уже является частью системы, но добавлять его только ради получения единого унифицированного интерфейса будет избыточно.

Программа для управления гибридными облачными сервисами и облачными хранилищами "ACCENTOS"

Программа с коммерческим названием "AccentOS", запатентованная в Российской Федерации [30], предназначена для управления гибридными облачными сервисами и облачными хранилищами. Согласно описания патента, в данной программе реализовано:

- управление физическими серверами, входящими в облачную инфраструктуру;
- мониторинг состояния серверов в реальном времени; автоматическое восстановление облачных сервисов в случае отказа или аварии;
- унификацию интерфейсов управления для публичных, частных и смешанных облачных сервисов и облачных хранилищ;
- получение статистических данных об основных характеристиках производительности серверов;
- интефацию с системами мониторинга и резервного копирования сторонних производителей;
- предоставление расширенной поддержки драйверов для взаимодействия с серверами в облачной инфраструктуре;
- предоставление расширенной поддержки динамического управления

серверами в облачной инфраструктуре.

Перечень не является исчерпывающим и полным и содержит лишь те пункты, которые являются важными в данной работе.

Основываясь на этом перечне, можно сделать вывод о том, что в данной программе реализованы все требования к сервису управления виртуализированной инфраструктурой, кроме одного: предоставление унифицированного программного интерфейса (API). Действительно, согласно патенту, программа ”AccentOS” предоставляет интерфейс администратору, но не предусматривает возможности интеграции внешних систем. Таким образом, данную программу нельзя считать сервисом управления. Наоборот, эта программа могла бы быть реализована с использованием такого сервиса, но, в отсутствие общедоступной реализации, программа содержит аналог такого сервиса как один из модулей.

Поддержка мульти-облачных систем в нативных облачных приложениях с использованием платформы эластичных контейнеров

Существует решение [31], позволяющее организовать облачную систему на основе разнородных поставщиков услуг облачных сервисов и разнородных платформ эластичных контейнеров. В качестве примера организации такой облачной системы и доказательств работоспособности решения в статье приведена система следующей конфигурации:

- Поставщики облачной инфраструктуры:
  - Google Cloud Engine
  - Amazon AWS
  - OpenStack (в датацентре университета исследователя)
- Платформы эластичных контейнеров:
  - Kubernetes
  - Docker Swarm

Само решение состоит из утилиты командной строки, принимающей на

вход *желаемое* и *текущее* состояния всей облачной системы. Затем утилита производит необходимые для достижения *желаемого* состояния мероприятия. Таким образом, утилита решает задачи сервиса управления виртуализированной инфраструктуры:

1. предоставление унифицированного программного интерфейса (API);
2. предоставить возможность изменять количество выделенных каждому из приложений ресурсов.

В работе так же освещена проблема интеграции дополнительных реализаций виртуализированной инфраструктуры. В частности, для интеграции дополнительной платформы облачных вычислений или для интеграции дополнительного поставщика облачной инфраструктуры необходимо реализовать адаптер API дополнительно интегрируемого модуля к API всей системы. Таким образом, задача предоставления возможности адаптации новых модулей решена.

В работе не освещён вопрос мониторинга *текущего* состояния системы и, как следствие, задача сбора статистики использования приложениями вычислительных ресурсов, не решена.

Поддержка программируемых правил автомасштабирования для контейнеров и виртуальных машин на облачных платформах

Policy Keeper [32], осуществляющий автомасштабирование контейнеров и виртуальных машин на различных облачных платформах, был спроектирован с поддержкой раздельного автомасштабирования узлов (виртуальных машин) системы и контейнеров системы. Иными словами, Policy Keeper позволяет изменять количество контейнеров приложения без изменения количества узлов приложения и наоборот. В круг задач, решаемых таким сервисом автомасштабирования, входят:

1. осуществлять мониторинг набора показателей использования вычислительных ресурсов по каждому из приложений;
2. позволять администратору изменять набор показателей, по которым со-

бирается статистика, для каждого приложения;

3. осуществлять автомасштабирование каждого из приложений по заданному администратором набору правил;
4. позволять создавать условия на основе статистики использования вычислительных ресурсов в каждом из правил автомасштабирования;
5. позволять адаптировать сервис автомасштабирования к облачным системам, построенным на базе других реализаций виртуализированной инфраструктуры.

Из перечисленных выше требований к Policy Keeper, можно сделать следующие выводы:

1. требование к сервису управления виртуализированной инфраструктурой об адаптации к новым облачным платформам здесь реализовано в полной мере;
2. требование о предоставлении унифицированного интерфейса здесь не реализовано, так как Policy Keeper не предоставляет доступ к собранной им информации, он позволяет только создавать правила по которым он осуществляет автомасштабирование системы;
3. требование о сборе статистики здесь реализовано и Policy Keeper использует эту статистику для принятия решений по заданным правилам;
4. требование об изменении количества выделенных ресурсов здесь так же реализовано, Policy Keeper умеет включать и отключать дополнительные узлы и контейнеры в системе.

Таким образом, Policy Keeper реализует часть требований к сервису управления, но не все из них. Более того, нужно отметить, что сервис управления мог бы выступать как часть (модуль) системы автомасштабирования по программируемым правилам.

1.5 Постановка задачи

В результате обзора предметной области было выявлено, что в настоящее время не существует решений, удовлетворяющих всем предъявляемым требованиям. В связи с этим возникает необходимость разработки подхода к управлению виртуализированной инфраструктурой, удовлетворяющего таким требованиям.

Подход к управлению виртуализированной инфраструктурой должен решать задачу взаимодействия с платформами облачных вычислений, в то время как компоненты управления ресурсами будут принимать решения о необходимости осуществления такого взаимодействия. Таким образом, компонент управления ресурсами получает возможность управлять ресурсами систем с различными технологиями виртуализации инфраструктуры без значительных доработок, потому что задача адаптации новой платформы облачных вычислений будет решена разработанным подходом к управлению.

Для принятия решения об осуществлении управляющего взаимодействия, компоненту управления ресурсами необходимо знать какое количество ресурсов сейчас простаивает[21]. Для этого подход к управлению должен предусмотреть возможность сбора статистики использования вычислительных мощностей каждым из приложений[33].

Суммируя вышесказанное, можно сказать, что задача управления виртуализированной инфраструктурой включает в себя две основных подзадачи.

1. Предоставление доступа к статистике использования вычислительных мощностей каждым из приложений.
2. Предоставление возможности изменять количество выделенных вычислительных мощностей каждому из приложений.

2 ПРЕДЛОЖЕННЫЙ ПОДХОД К УПРАВЛЕНИЮ ВИРТУАЛИЗИРОВАННОЙ ИНФРАСТРУКТУРОЙ

2.1 Описание предложенного подхода

Для решения задачи, поставленной в пункте 1.5 в данной работе предлагается подход, основанный на введении в облачную систему дополнительного сервиса как показано на рис. 5. Такой сервис будет называться далее сервисом управления виртуализированной инфраструктурой.

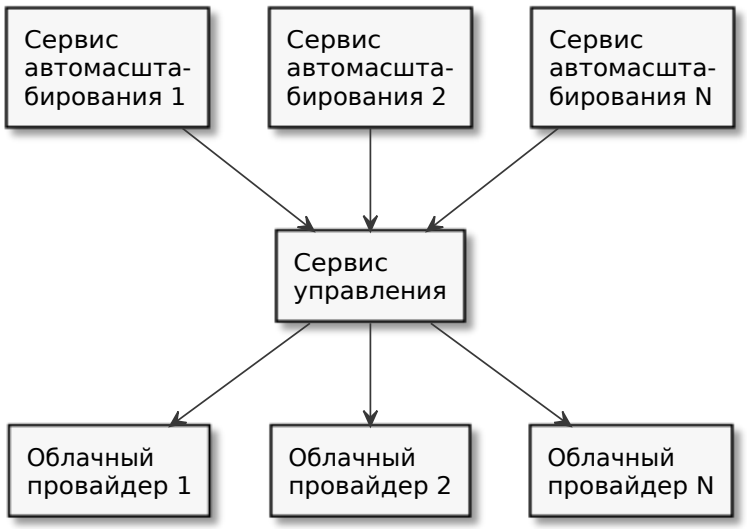

Рис. 5: Облачная система с дополнительным сервисом управления

При помощи введения в систему дополнительного промежуточного компонента решается задача абстрагирования компонентов управления ресурсами от конкретных объектов управления - платформ виртуализации инфраструктуры. Таким образом, при разработке компонента управления ресурсами (например, сервиса автомасштабирования), нужно будет учитывать только программный интерфейс (API) сервиса управления, даже если в целевой системе будет несколько различных платформ облачных вычислений.

Тем не менее, при необходимости введения в систему новой платформы виртуализации, возникнет необходимость доработки сервиса управления с учётом программного интерфейса добавляемой платформы. Однако это по-

требуется сделать лишь один раз и далее можно будет использовать неограниченное количество облачных платформ, предоставляющих такой же программный интерфейс.

2.2 Аутентификация в управляемых платформах

В целях уменьшения затрат на реализацию компонентов управления ресурсами, аутентификацию в управляемой системе должен выполнять тоже сервис управления. Для этого у администратора системы должна быть возможность сохранить параметры подключения, такие как логин, пароль, токен и так далее. После сохранения параметров, администратору системы должен быть предоставлен идентификатор сохранённого набора параметров. После этого администратор должен будет передать этот идентификатор компоненту управления ресурсами. Дальнейшее взаимодействие с сервисом управления потребует от клиентов только идентификатора. Таким образом, реализация компонента управления ресурсами не будет требовать реализации процесса аутентификации ни в одной из целевых платформ виртуализации.

Как проиллюстрировано на рис. 6, после начальной конфигурации компонентов системы, участие администратора в работе системы не требуется. Для взаимодействия с сервисом управления, компоненту управления требуется знать лишь идентификатор, имея который есть возможность осуществлять, например, горизонтальное масштабирование.

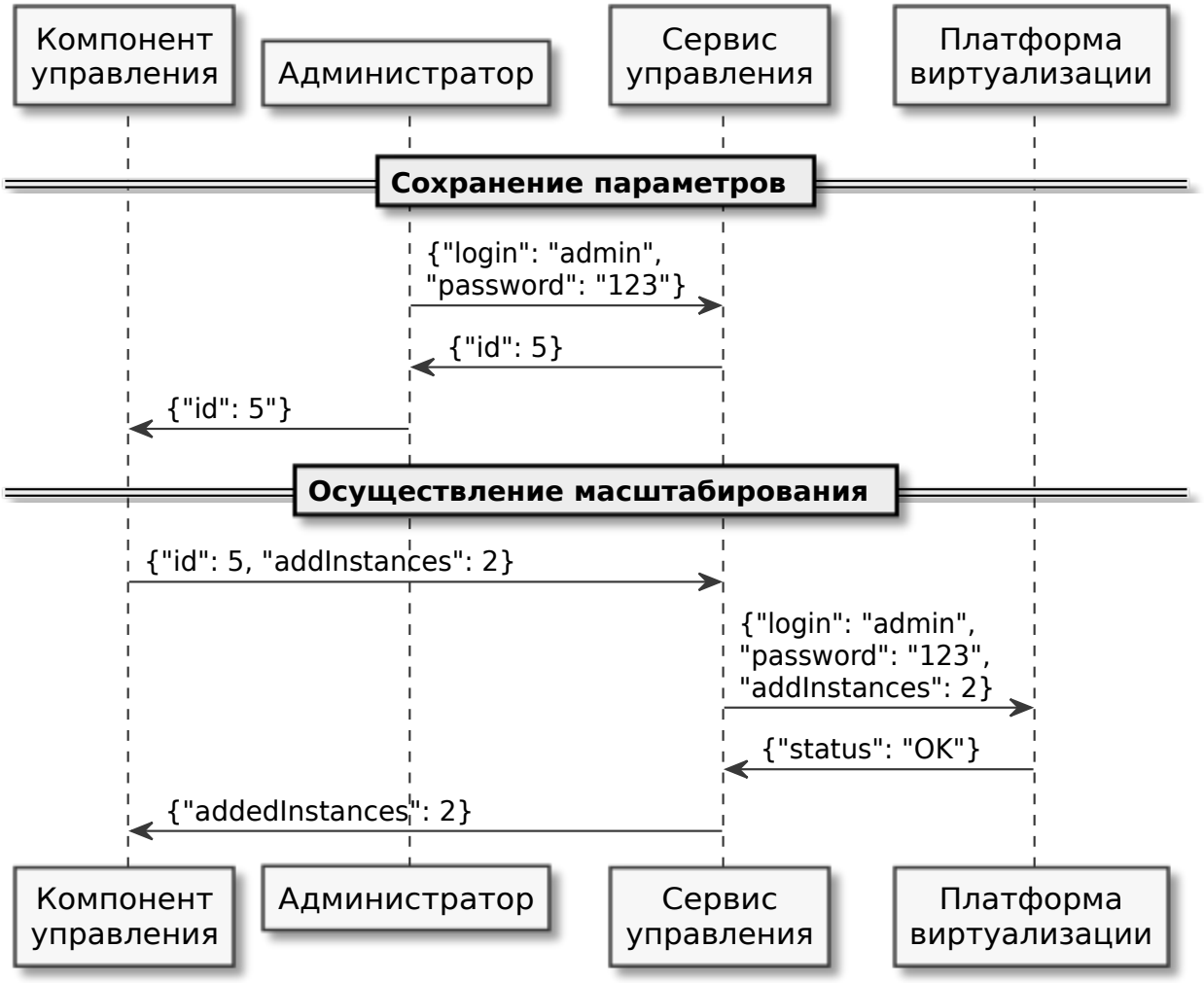

Рис. 6: Пример конфигурации компонента управления и дальнейшего взаимодействия

2.3 Требования к сервису управления

К разрабатываемому сервису управления виртуализированной инфраструктурой предъявлен следующий список функциональных и нефункциональных требований:

1. Интегрируемость с Kubernetes и OpenNebula. Требование интегрируемости именно с этими реализациями виртуализированной инфраструктуры обусловлено существенной разницей в их организации и, соответственно, программных интерфейсах (API), которые они предоставляют. Такая существенная разница позволит продемонстрировать преимущества использования сервиса управления наиболее полно.
2. Адаптируемость к дополнительным реализациям виртуализированной инфраструктуры с использованием программных адаптеров. Это нефункциональное требование объясняется необходимостью поддержки различных платформ облачных вычислений, в том числе не существующих на сегодняшний день, в связи с чем должна быть предусмотрена возможность адаптировать сервис к дополнительным платформам.
3. Возможность описать и сохранить параметры подключения к виртуализированной инфраструктуре, включая параметры конкретного приложения. Это функциональное требование возникает вследствие необходимости поддержки гибридных облачных окружений, состоящих из нескольких облачных платформ и нескольких приложений.
4. Возможность запросить список текущих приложений, то есть список параметров подключения к каждому из приложений.
5. Возможность обновить параметры подключения для каждого из приложений.
6. Возможность удалить параметры подключения для каждого из приложений.
7. Возможность запросить информацию по приложению, включающую в себя:

- текущее количество ВМ (виртуальных машин) или контейнеров, выделенное данному приложению в данной платформе облачных вычислений;
- имя каждого из контейнеров или ВМ в платформе облачных вычислений;
- текущую загрузку центрального процессора в каждом из контейнеров или ВМ;
- текущее количество используемой каждой ВМ или контейнером оперативной памяти (RAM).

Это требование является частью задачи управления, решаемой сервисом управления, а именно подзадача сбора статистики использования вычислительных ресурсов (мониторинга).

8. Возможность осуществления масштабирования, то есть возможность изменять текущее количество выделенных контейнеров или ВМ для каждого из приложений. Это требование является частью задачи управления, решаемой сервисом управления, а именно подзадача изменения количества выделенных вычислительных мощностей.
9. Предоставление единого унифицированного программного интерфейса (API), не зависящего от типа управляемых виртуализированных инфраструктур, а так же их количества.

3 РАЗРАБОТКА СЕРВИСА УПРАВЛЕНИЯ ВИРТУАЛИЗИРОВАННОЙ ИНФРАСТРУКТУРОЙ

3.1 Описание компонентов сервиса

Для решения задачи, поставленной в разделе 1.5, с учётом требований в разделе 2.3 было спроектировано решение, представленное на рис. 7.

1. СУБД.

   Конфигурацию приложений необходимо надёжно хранить для того, чтобы у администраторов системы не было необходимости заново настраивать сервис управления в случае аварий.

2. Веб-сервис — сервис, осуществляющий взаимодействие с внешними компонентами, в частности с сервисами автомасштабирования.

   Он необходим для предоставления API этим сервисам, а так же для взаимодействия с СУБД. Состоит из двух компонентов:

   - менеджер;
   - REST API.

3. Клиентская библиотека — программный модуль (библиотека), который будет подключаться к веб-сервису.

   Данная библиотека будет содержать API для взаимодействия с каждой из поддерживаемых платформами облачных вычислений, а также предоставлять унифицированный интерфейс для взаимодействия, который будет использоваться веб-сервисом для обработки запросов.

Приложение и внешний компонент не являются частью разрабатываемого сервиса. Напротив, сервис служит промежуточным звеном между ними для организации независимого взаимодействия.

Пример взаимодействия компонентов представлен в виде диаграммы последовательности на рис. 8

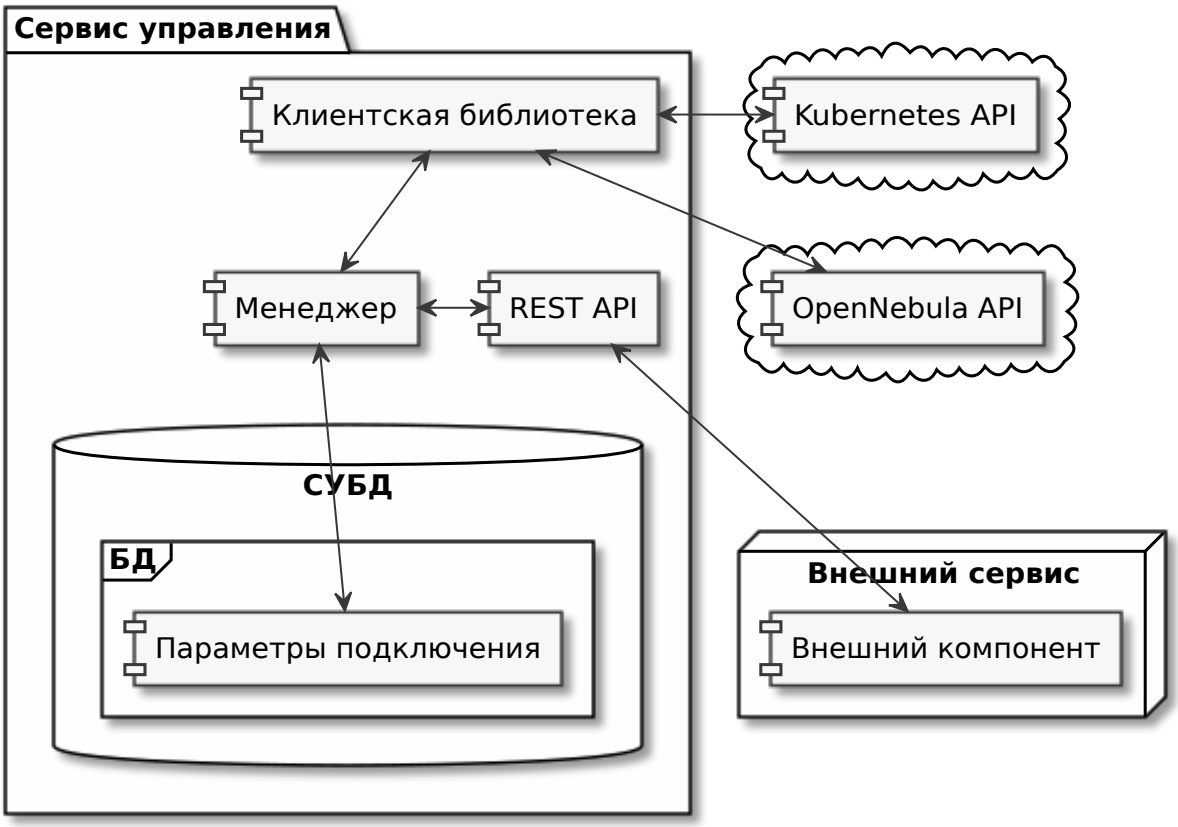

Рис. 7: Предложенная организация решения

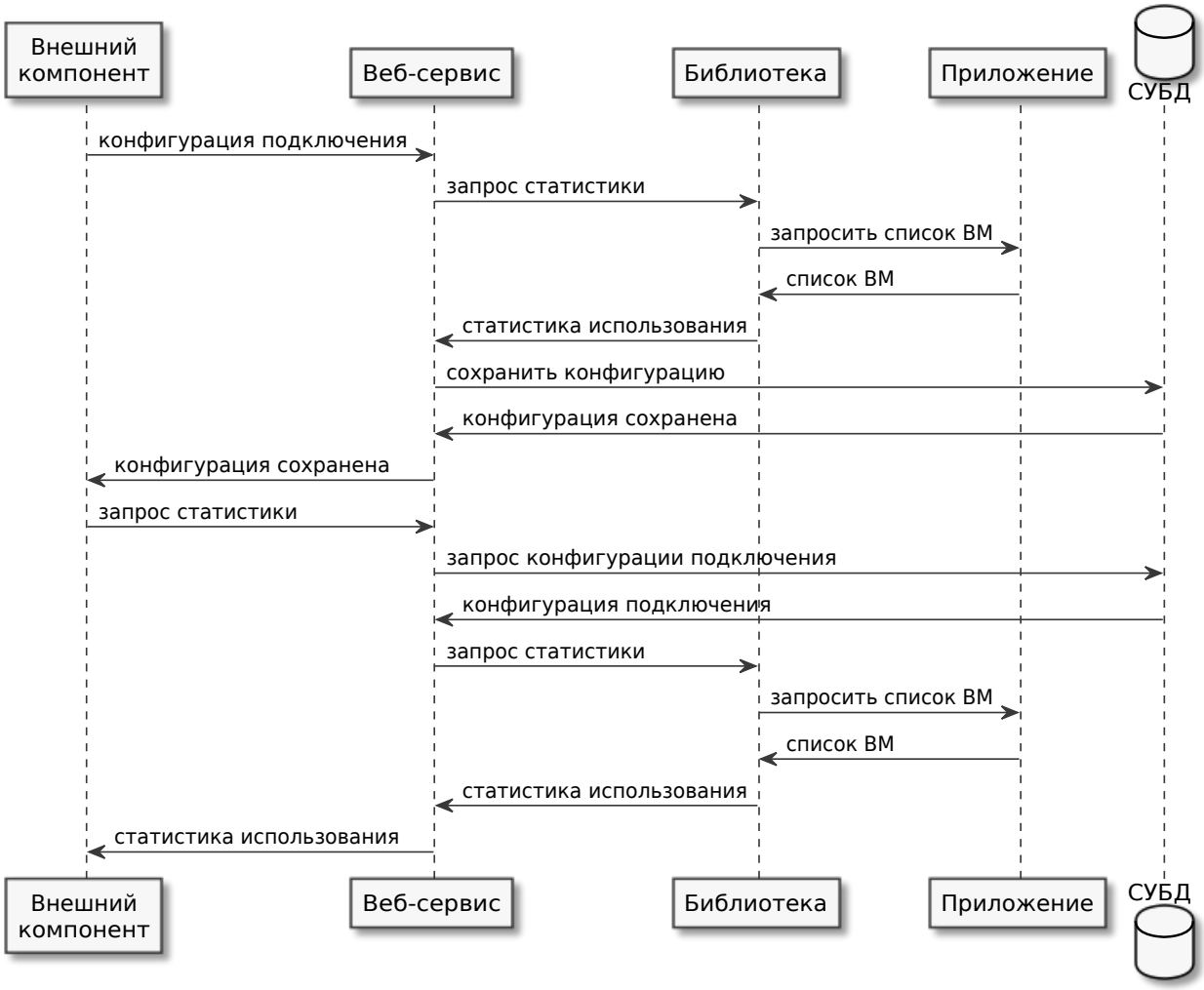

Рис. 8: Диаграмма последовательности взаимодействия компонентов

3.2 Описание схемы базы данных

Организация базы данных, используемой сервисом, представлена на рис. 9.

Таблица ”application_entity” содержит записи с информацией об известных системе приложениях. Описание полей сущности представлено в табл. 2.

Таблица ”kubernetes_config_entity” содержит записи с информацией о параметрах подключения к платформе ”Kubernetes”, а так же о конкретном приложении под управлением данной платформы. Описание полей сущности представлено в табл. 3.

Таблица ”open_nebula_config_entity” содержит записи с информацией о параметрах подключения к платформе ”OpenNebula”, а так же о конкретном приложении под управлением данной платформы. Описание полей сущности представлено в табл. 4.

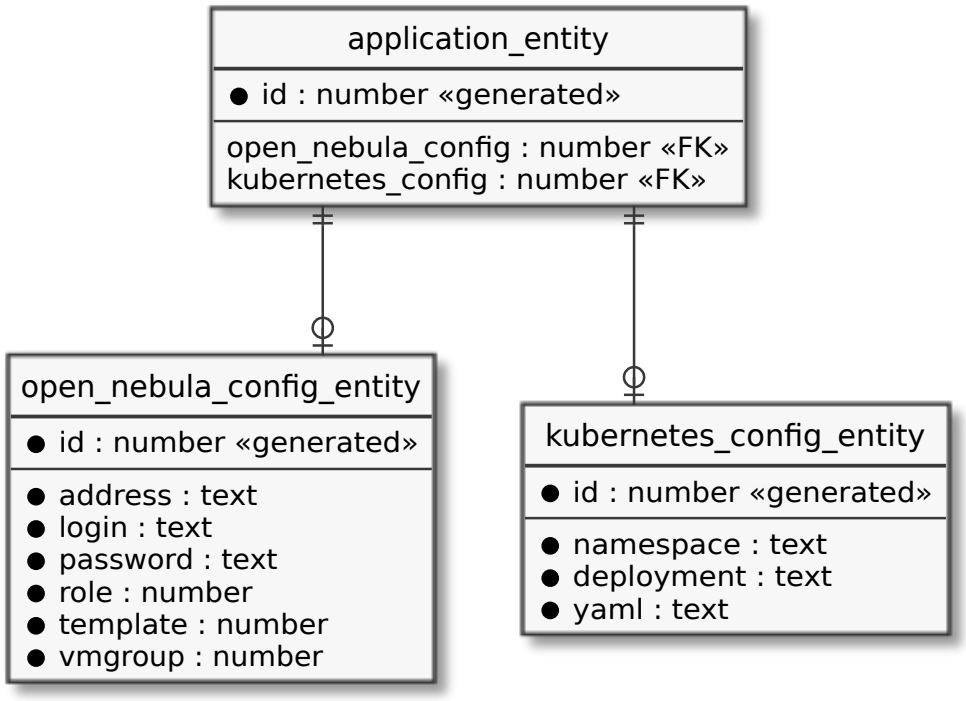

Рис. 9: Схема базы данных

3.3 Описание REST API веб-сервиса

В этом разделе содержится перечисление всех вызовов HTTP REST API, которые поддерживает веб-сервис, а так же описание запросов и ответов.

Запросы

Передать для сохранения параметры подключения к приложению под управлением ”Kubernetes” можно с помощью запроса, описание которого приведено в табл. 5.

Передать для сохранения параметры подключения к приложению под управлением ”OpenNebula” можно с помощью запроса, описание которого приведено в табл. 12.

Обновить параметры ранее созданного подключения к приложению под управлением ”OpenNebula” можно с помощью запроса, описание которого приведено в табл. 13.

Обновить параметры ранее созданного подключения к приложению под управлением ”Kubernetes” можно с помощью запроса, описание которого приведено в табл. 7.

Запросить список приложений, параметры подключения к которым сохранены в СУБД, можно с помощью запроса, описание которого приведено в табл. 8.

Удалить ранее сохранённые параметры подключения к приложению вне зависимости от платформы, под управлением которой оно находится, можно с помощью запроса, описание которого приведено в табл. 9.

Запросить список ВМ или контейнеров относящихся к приложению, параметры подключения к которому ранее были сохранены, вне зависимости от платформы, под управлением которой оно находится, а так же статистическую информацию об использованных ими ресурсах, можно с помощью запроса, описание которого приведено в табл. 10.

Осуществить масштабирование приложения, параметры подключения к которому ранее были сохранены, вне зависимости от платформы, под управлением которой оно находится, можно с помощью запроса, описание которого приведено в табл. 11.

Ответы

Ответ на любой запрос может содержать либо описание ошибки в формате, который является общим для всех запросов, либо содержать тело ответа в формате, зависящем от запроса, либо содержать только HTTP-код ответа без тела сообщения.

Описание общего формата ошибки приведено в табл. 14.

Ответ на запрос сохранения параметров вне зависимости от платформы, параметры подключения к которой были сохранены, а так же ответ на запрос удаления приложения описан в табл. 6.

Ответ на запрос списка сохранённых приложений, то есть списка известных параметров подключения к приложениям, описан в табл. 16.

Ответы на запрос состояния приложения и на запрос масштабирования приложения не зависят от платформы, под управлением которой находится само приложение. Формат ответа описан в табл. 17.

Ответ на запрос обновления конфигурации так же не зависит от платформы облачных вычислений, к которой относится конфигурация. Формат ответа описан в табл. 15.

3.4 Описание клиентской библиотеки

Клиентская библиотека — программный модуль в составе сервиса управления виртуализированной инфраструктурой, который отвечает за взаимодействие с каждой из поддерживаемых платформ облачных вычислений, а так же предоставляет унифицированный программный интерфейс (API), не зависящий от конкретной платформы.

На рис. 10 приведена диаграмма классов клиентской библиотеки. В центре диаграммы находятся два интерфейса ”AppClient” и ”AppInstance”, которые служат для абстрагирования пользователей библиотеки от такого с какой именно платформой облачных вычислений они работают.

В частности, ”AppClient” предоставляет интерфейс взаимодействия с

приложением, а именно два метода:

1. getAppInstances — метод для получения списка ВМ или контейнеров, выделенных этому приложению;
2. scaleInstances — метод для изменения количества выделенных приложению ВМ или контейнеров.

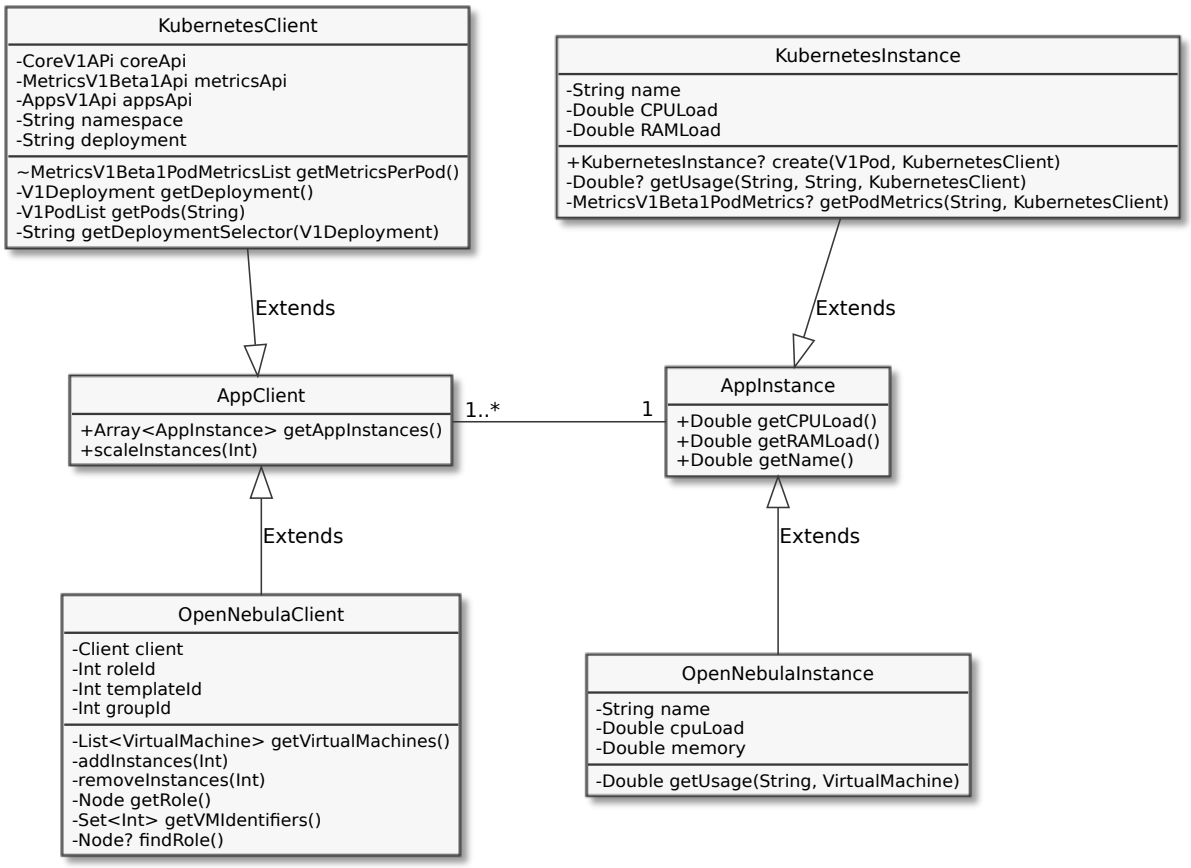

Рис. 10: Диаграмма классов

Выделенные приложению ВМ или контейнеры (в зависимости от платформы) представлены интерфейсом ”AppInstance”, который позволяет узнать своё имя, заданное на платформе облачных вычислений, а так же собранную статистическую информацию по использованным ресурсам. В частности, ”getCPULoad” возвращает загрузку вычислительных ядер в течение последней минуты, а ”getRAMLoad” возвращает количество используемых приложением байт оперативной памяти.

Остальные классы на данной диаграмме являются реализациями двух

описанных выше интерфейсов под конкретные виртуализированные инфраструктуры. Как видно из набора полей и методов этих классов, внутренне они существенно различаются, но, несмотря на это, пользователи библиотеки могут взаимодействовать с ними через один и тот же интерфейс, что ускоряет и упрощает разработку новых модулей облачных систем.

3.5 Демонстрация упрощения реализации компонента управления ресурсами

На рис. 11 изображена диаграмма последовательности взаимодействия компонентов системы, в которой нет сервиса управления. Как видно на диаграмме, в такой системе реализация компонента управления ресурсами (сервиса автомасштабирования) требует реализации получения, обработки и хранения параметров подключения к системе, а так же их последующего использования. Причём, как видно на диаграмме, сервис автомасштабирования должен уметь обрабатывать различные параметры подключения, специфичные для конкретной платформы виртуализации. Кроме того, в такой системе сервис автомасштабирования должен содержать реализацию взаимодействия с платформой виртуализации согласно тому программному интерфейсу (API), который она предоставляет.

Например, платформа OpenNebula оперирует понятиями ”группа ВМ” и ”шаблон ВМ”, а платформа Kubernetes не использует ВМ совсем, зато использует понятия ”namespace” и ”deployment”, которых нет в OpenNebula.

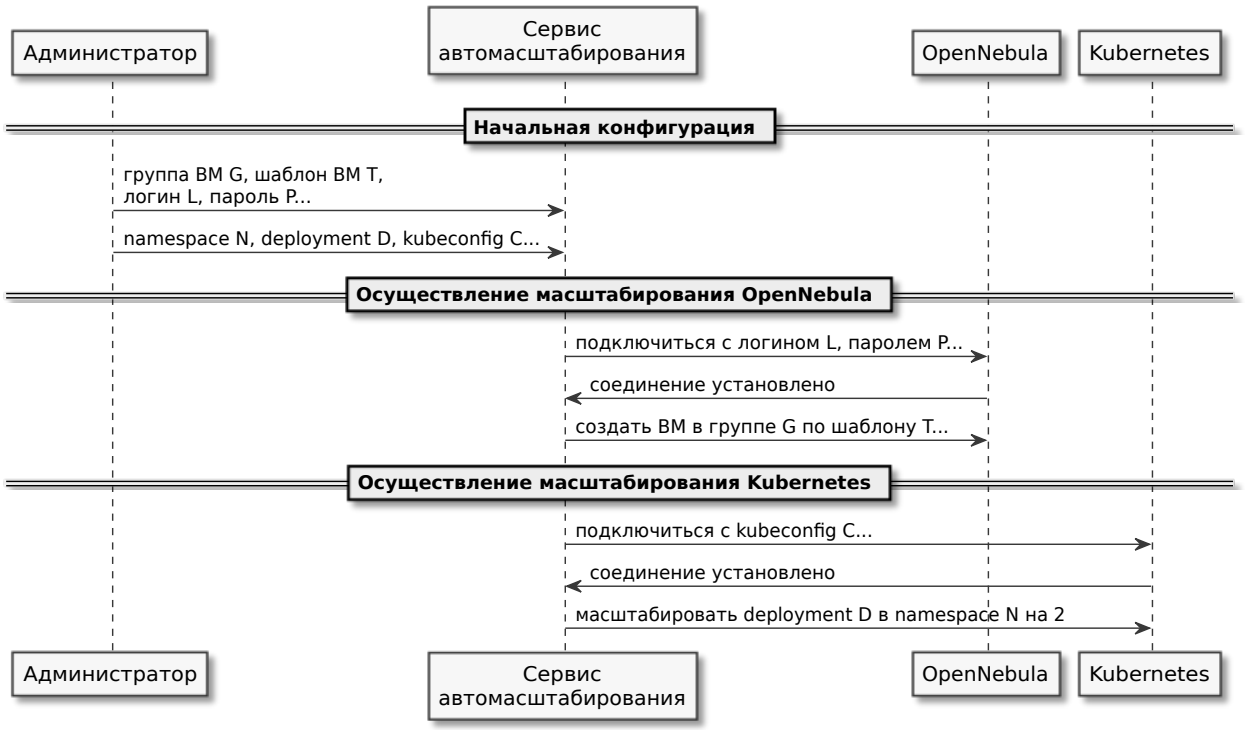

Рис. 11: Диаграмма последовательности взаимодействия компонентов в системе без сервиса управления

С введением в систему сервиса управления, как показано на рис. 12, взаимодействие с платформами виртуализации и работа администратора не изменились. Однако реализация компонента управления ресурсами теперь предполагает взаимодействие по одному и тому же унифицированному интерфейсу, не зависящему от используемой платформы виртуализации. Компонент управления ресурсами должен уметь обрабатывать и хранить только идентификатор управляемого приложения. Для осуществления самого масштабирования компонент управления теперь не должен уметь оперировать понятиями ”группа ВМ”, ”шаблон ВМ” и тому подобными. Напротив, необходимо реализовать лишь команду ”incrementBy”, при получении которой сервис управления сам осуществит горизонтальное масштабирование в соответствии с тем программным интерфейсом, который предоставляет конкретная платформа виртуализации.

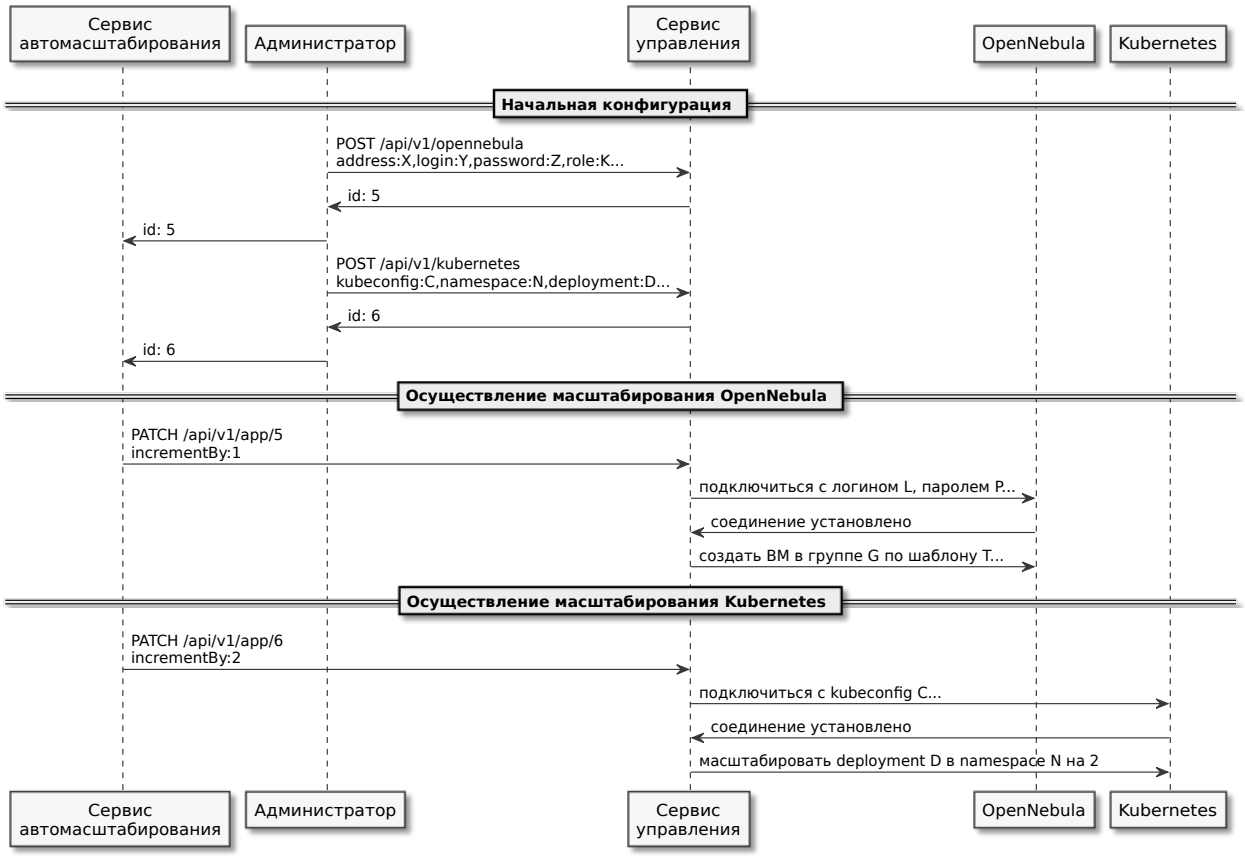

Рис. 12: Диаграмма последовательности взаимодействия компонентов в системе с сервисом управления

Упрощение разработки можно показать с помощью формулы расчёта разницы трудозатрат T на реализацию компонента управления ресурсами (сервиса автомасштабирования) в системе с сервисом управления и в системе без такого сервиса.

Пусть затраты на реализацию взаимодействия с платформой виртуализации будут O, а на реализацию взаимодействия с сервисом управления — M. При этом платформ виртуализации в системе будет предусмотрено N, а затраты на реализацию самого компонента управления ресурсами — S.

Предположим также, что зависимость количества трудозатрат на реализацию от количества платформ виртуализации N линейная. Тогда затраты T_1 на реализацию компонента управления ресурсами в системе без сервиса управления можно рассчитать по формуле 1.

$$T_1 = S + \sum_{i=1}^{N} O_i \tag{1}$$

Трудозатраты T_2 на реализацию компонента управления в системе, где предусмотрен сервис управления, можно рассчитать по формуле 2.

$$T_2 = S + M \tag{2}$$

Разницу в трудозатратах T на реализацию компонента управления в системе с сервисом управления и в системе без такого сервиса можно рассчитать по формуле 3.

$$T = T_2 - T_1 = M - \sum_{i=1}^{N} O_i \tag{3}$$

Чем меньше T, тем сложнее реализация компонента управления ресурсами без сервиса управления. В частности, при отрицательных значениях T, реализация компонента управления ресурсами в системе с сервисом управления будет проще, чем в такой же системе без него.

3.6 Реализованные требования

Таблица 1 содержит краткий перечень реализованных требований к сервису управления виртуализированной инфраструктурой.

Таблица 1: Перечень реализованных требований к сервису

Требование	**Описание реализации**
Интегрируемость с Kubernetes и OpenNebula.	Реализация интеграции с обеими платформами включена в модуль ”клиентская библиотека”.

Продолжение на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Требование	Описание реализации
Адаптируемость к дополнительным платформам.	Паттерн программирования ”Фабрика”, применённый для реализации клиентской библиотеки, позволяет неограниченно расширять список поддерживаемых платформ.
Возможность описания и сохранения параметров подключения.	При помощи запросов к REST API веб-сервиса, описанных в табл. 5 и табл. 12, передаются параметры подключения для хранения в БД в формате, описанном в табл. 3 и табл. 4 соответственно.
Возможность запроса списка текущих приложений.	Список известных параметров подключения предоставляется по REST API с помощью запроса, описанного в табл. 8, ответ на который приведён в табл. 16.
Возможность обновления параметров подключения.	Сохранённые ранее параметры обновляются с помощью запросов к REST API, описанных в табл. 7 и 13.
Возможность удаления параметров подключения.	Сохранённые ранее параметры удаляются с помощью запроса к REST API, описанного в табл. 9.
Возможность запроса текущей информации о приложении.	Текущая информация о приложении предоставляется по REST API с помощью запроса, описанного в табл. 10, ответ на который описан в табл. 17.
Возможность осуществления масштабирования.	Масштабирование приложения осуществляется по REST API с помощью запроса, описанного в табл.11.
Предоставление унифицированного интерфейса.	Сервис предоставляет унифицированный интерфейс, так как запросы текущей информации и масштабирования не зависят от управляемой платформы.

Таким образом, все требования, предъявленные к сервису в пункте 2.3, были реализованы в полном объёме.

ЗАКЛЮЧЕНИЕ

В выпускной квалификационной работе были получены следующие результаты.

1. Выявлены типовые подходы к управлению виртуализированной инфраструктурой, преимущества и недостатки существующих решений задачи управления такой инфраструктурой, сформулированы функциональные и нефункциональные требования к сервису управления и сделан вывод, что ни один из существующих сервисов не удовлетворяет этим требованиям.
2. Разработан подход к управлению виртуализированной инфраструктурой, а также сервис, реализующий этот подход. Выявлено, что все сформулированные требования удовлетворены.
3. Сформулированы выражения для вычисления трудозатрат на реализацию компонентов управления ресурсами в системах с сервисом управления и без него, с помощью которых показаны преимущества разработанного подхода.
4. На практике разработан сервис управления виртуализированной инфраструктурой и продемонстрировано существенное упрощение процесса реализации компонентов управления ресурсами за счёт использования разработанного подхода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Petcu, D., Macariu, G., Panica, S., & Crăciun, C. (2013). Portable cloud applications — from theory to practice. Future Generation Computer Systems, 29(6), 1417-1430.

[2] Matthew Portnoy. Virtualization Essentials, 2nd Edition. – New York: Wiley / Sybex, 2016. – 336 с.

[3] Мартынчук И. Г., Жмылёв С. А. Архитектура и организация сервисов автомасштабирования в облачных системах // Альманах научных работ молодых ученых Университета ИТМО – 2017.

[4] Singh J. et al. Data flow management and compliance in cloud computing //IEEE Cloud Computing. – 2015. – Т. 2. – No. 4. – С. 24-32.

[5] Managing competing elastic Grid and Cloud scientific computing applications using OpenNebula / S Bagnasco, D Berzano, S Lusso [и др.] // Journal of Physics: Conference Series / IOP Publishing. Т. 664. 2015. С. 022004.

[6] Kaushal Vishonika, Bala Anju. Autonomic Fault Tolerance Using HAProxy in Cloud Enviorment. Ph.D. thesis: The Thapar Institute of Engineering and Technology. 2011.

[7] Mr. Ray J Rafaels. Cloud Computing: From Beginning to End. – CreateSpace Independent Publishing Platform, 2015. – 152 с.

[8] Жмылёв С.А., Мартынчук И.Г., Киреев В.Ю. Оценка длины периода нестационарных процессов в облачных системах // VI научно-практическая конференция с международным участием «Наука настоящего и будущего» для студентов, аспирантов и молодых ученых. 2018. С. 41–43.

[9] Naresh Kumar Sehgal, Pramod Chandra P. Bhatt. Cloud Computing: Concepts and Practices — Springer, 2018. — 269 с.

[10] Bogatyrev V., Vinokurova M. Control and safety of operation of duplicated

computer systems // International Conference on Distributed Computer and Communication Networks / Springer. 2017. С. 331–342.

[11] Мартынчук И.Г., Жмылёв С.А. Модели и методы композиции нестационарных распределений // Сборник тезисов докладов конгресса молодых ученых. – СПб., 2019.

[12] Мартынчук И.Г. Разработка программного комплекса для автоматического управления ресурсами облачной вычислительной системы // Аннотированный сборник научно-исследовательских выпускных квалификационных работ бакалавров Университета ИТМО. 2017. С. 44–46.

[13] Mao M., Humphrey M. A performance study on the vm startup time in the cloud // Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. – IEEE, 2012. – С. 423-430.

[14] Chhibber A., Batra S. Security analysis of cloud computing //International Journal of Advanced Research in Engineering and Applied Sciences. – 2013. – Т. 2. – No. 3. – С. 2278-6252.

[15] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu and W. Zhou, ”A Comparative Study of Containers and Virtual Machines in Big Data Environment,” // 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2018, pp. 178-185, doi: 10.1109/CLOUD.2018.00030.

[16] Rodriguez I., Llana L., Rabanal P. A General Testability Theory: Classes, properties, complexity, and testing reductions //IEEE Transactions on Software Engineering. – 2014. – Т. 40. – No. 9. – С. 862-894.

[17] Медведев Алексей. Облачные технологии: тенденции развития, примеры исполнения // Современные технологии автоматизации. 2013. Т. 2. С. 6–9.

[18] He S. et al. Elastic application container: A lightweight approach for cloud resource provisioning //Advanced information networking and applications (aina), 2012 ieee 26th international conference on. – IEEE, 2012. – С. 15-22.

[19] Optimal cloud resource auto-scaling for web applications / Jiang J., Jie Lu,

Guangquan Zhang [и др.] // 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. IEEE, 2013. С. 58–65.

[20] Y. Wadia, R. Gaonkar and J. Namjoshi, ”Portable Autoscaler for Managing Multi-cloud Elasticity,” // 2013 International Conference on Cloud & Ubiquitous Computing & Emerging Technologies, Pune, 2013, с. 48-51.

[21] Barr Jeff, Narin Attila, Varia Jinesh. Building fault-tolerant applications on aws // Amazon Web Services. 2011. С. 1–15.

[22] I. Bermudez, S. Traverso, M. Mellia. Exploring the cloud from passive measurements: The Amazon AWS case // 2013 Proceedings IEEE INFOCOM. IEEE, 2013. С. 230–234.

[23] Analytical methods of nonstationary processes modeling / Sergei Zhmylev, Ilya Martynchuk, Valeriy Kireev [и др.] // CEUR Workshop Proceedings. 2019.

[24] Using machine learning for black-box autoscaling / Muhammad Wajahat, Anshul Gandhi, Alexei Karve [и др.] // 2016 Seventh International Green and Sustainable Computing Conference (IGSC) / IEEE. 2016. С. 1–8.

[25] Comparison of open-source cloud management platforms: OpenStack and OpenNebula / Xiaolong Wen, Genqiang Gu, Qingchun Li [и др.] // 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery / IEEE. 2012. С. 2457–2461.

[26] Патент US 9755858 B2, 2017.

[27] Патент WO 2014021849 Al, 2014.

[28] Aktas, MS. Hybrid cloud computing monitoring software architecture. Concurrency Computat Pract Exper. 2018; 30:e4694. https://doi.org/10.1002/cpe.4694 (дата обращения 21.05.2020)

[29] M. M. Shreyas, ”Federated Cloud Services using Virtual API Proxy Layer in a Distributed Cloud Environment,” // 2017 Ninth International Conference on Advanced Computing (ICoAC), Chennai, 2017, с. 134-141.

[30] Патент RU 2020612651, 2020.

[31] Nane Kratzke. 2017. Smuggling Multi-cloud Support into Cloud-native Applications using Elastic Container Platforms. // In Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017). SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 57–70. DOI:https://doi.org/10.5220/0006230700570070 (дата обращения 21.05.2020)

[32] Kovács, J. (2019). Supporting Programmable Autoscaling Rules for Containers and Virtual Machines on Clouds. Journal of Grid Computing, 17(4), 813-829.

[33] Lorido-Botrán Tania, Miguel-Alonso José, Lozano Jose Antonio. Auto-scaling techniques for elastic applications in cloud environments // Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09. 2012. T. 12. C. 2012.

ПРИЛОЖЕНИЕ 1. ЛИСТИНГИ РАЗРАБОТАННЫХ ПРОГРАММНЫХ МОДУЛЕЙ

В листинге 1 представлен исходный код компонента ”Менеджер”, который обрабатывает запросы, получаемые от компонента ”REST API”. Компонент ”REST API” предоставлен фреймворком ”Spring”.

Листинг 1: Исходный код компонента ”Менеджер” (ApiController.kt)

```
@RestController
class ApiController {
    private val logger = LoggerFactory.getLogger(javaClass)

    @Autowired
    private lateinit var appRepository: ApplicationRepository

    @Autowired
    private lateinit var kubernetesConfigRepo: KubernetesConfigRepository

    @Autowired
    private lateinit var openNebulaConfigRepo: OpenNebulaConfigRepository

    @PostMapping("/api/v1/kubernetes/{namespace}/{deployment}")
    fun createKubernetes(
            @RequestBody yaml: String,
            @PathVariable("namespace") namespace: String,
            @PathVariable("deployment") deployment: String
    ): AppIdResponse {
        logger.info("createKubernetes(namespace: \"$namespace\", deployment:
        ↪ \"$deployment\", yaml: \"$yaml\")")

        val config = KubernetesConfigEntity(deployment, namespace, yaml)
        checkAppConfig(config)
        val id = saveConfiguration(config)

        return result("createKubernetes", AppIdResponse(id))
    }

    @PostMapping("/api/v1/opennebula")
    fun createOpenNebula(
            @RequestBody request: OpenNebulaRequest
    ): AppIdResponse {
        logger.info("createOpenNebula(request = $request)")
```

```
34
35          val config = OpenNebulaConfigEntity(request.address, request.login,
            ↪  request.password, request.role, request.template, request.vmgroup)
36          checkAppConfig(config)
37          val id = saveConfiguration(config)
38
39          return result("createOpenNebula", AppIdResponse(id))
40      }
41
42      @GetMapping("/api/v1/app/{id}")
43      fun getApplicationInfo(@PathVariable("id") id: Long): Array<AppInstanceResponse> {
44          logger.info("getApplicationInfo(id: \"$id\")")
45
46          val client = getApp(id).getAppClient()
47
48          val result = getAppInfo(client)
49
50          return result("getApplicationInfo", result)
51      }
52
53      @GetMapping("/api/v1/app")
54      fun getAppConfigurations(): AppConfigResponse {
55          logger.info("getAppConfigurations()")
56
57          val apps = appRepository.findAll()
58          val kubernetesApps = apps.mapNotNull {
59              KubernetesConfigResponse(
60                      namespace = it.kubernetesConfig?.namespace ?: return@mapNotNull null,
61                      deployment = it.kubernetesConfig.deployment,
62                      appId = it.id ?: throw IllegalStateException("Id must not be null if
                        ↪  data is retrieved from DB")
63              )
64          }
65
66          val openNebulaApps = apps.mapNotNull {
67              OpenNebulaConfigResponse(
68                      address = it.openNebulaConfig?.address ?: return@mapNotNull null,
69                      appId = it.id ?: throw IllegalStateException("Id must not be null if
                        ↪  data is retrieved from DB"),
70                      role = it.openNebulaConfig.role,
71                      template = it.openNebulaConfig.template,
72                      vmgroup = it.openNebulaConfig.vmgroup
73              )
74          }
75
76          return result("getAppConfigurations", AppConfigResponse(openNebulaApps,
            ↪  kubernetesApps))
```

```
77      }
78
79      @DeleteMapping("/api/v1/app/{id}")
80      fun removeApplication(@PathVariable("id") id: Long): AppIdResponse {
81          logger.info("removeApplication(id: \"$id\")")
82
83          if (appRepository.existsById(id))
84              appRepository.deleteById(id)
85          else
86              throw NoSuchApplicationException(id)
87
88          return result("removeApplication", AppIdResponse(id))
89      }
90
91      @PutMapping("/api/v1/kubernetes/{namespace}/{deployment}/{id}")
92      fun updateKubernetes(
93              @RequestBody yaml: String,
94              @PathVariable("namespace") namespace: String,
95              @PathVariable("deployment") deployment: String,
96              @PathVariable("id") id: Long
97      ) {
98          logger.info("updateKubernetes(namespace: \"$namespace\", deployment:
            ↪ \"$deployment\", id = $id, yaml: \"$yaml\")")
99
100         val app = getApp(id)
101         val currentConfig = app.kubernetesConfig ?: throw NoSuchApplicationException(id)
102         val updated = KubernetesConfigEntity(deployment, namespace, yaml, app,
            ↪ currentConfig.id)
103
104         checkAppConfig(updated)
105         saveConfiguration(updated, app)
106     }
107
108     @PutMapping("/api/v1/opennebula/{id}")
109     fun updateOpenNebula(
110             @RequestBody request: OpenNebulaRequest,
111             @PathVariable("id") id: Long
112     ) {
113         logger.info("updateOpenNebula(request = $request, id = $id)")
114
115         val app = getApp(id)
116         val currentConfig = app.openNebulaConfig ?: throw NoSuchApplicationException(id)
117         val config = OpenNebulaConfigEntity(
118                 request.address, request.login, request.password, request.role,
119                 request.template, request.vmgroup, app, currentConfig.id
120         )
```

```
121
122         checkAppConfig(config)
123         saveConfiguration(config, app)
124     }
125
126     @PatchMapping("/api/v1/app/{id}")
127     fun scaleApplication(@PathVariable("id") id: Long, @RequestBody request:
    ↪  ScaleRequest): Array<AppInstanceResponse> {
128         logger.info("scaleApplication(id = $id, request = $request)")
129
130         val client = getApp(id).getAppClient()
131
132         client.scaleInstances(request.incrementBy)
133
134         val result = getAppInfo(client)
135
136         return result("scaleApplication", result)
137     }
138
139     private fun checkAppConfig(config: AppConfiguration) {
140         logger.info("checkAppConfig(config = $config)")
141
142         val instances: Array<AppInstance>
143         try {
144             instances = config.getAppClient().getAppInstances()
145         } catch (e: AppClientException) {
146             logger.error("checkAppConfig: no connection", e)
147             throw NoAppConnectionException(e)
148         }
149
150         logger.info("checkAppConfig: instances count = ${instances.size}")
151         for (instance in instances) {
152             try {
153                 logger.info("checkAppConfig: instance(name = \"${instance.getName()}\",
                    ↪  CPU = ${instance.getCPULoad()}), RAM = ${instance.getRAMLoad()}")
154             } catch (e: AppClientException) {
155                 logger.error("checkAppConfig: unable to get instance info", e)
156                 throw InvalidAppConfigException(e)
157             }
158         }
159     }
160
161     private fun saveConfiguration(config: KubernetesConfigEntity, oldApp:
    ↪  ApplicationEntity? = null): Long {
162         val saved: KubernetesConfigEntity
163
```

```
164             try {
165                 saved = kubernetesConfigRepo.save(config)
166             } catch (e: DataIntegrityViolationException) {
167                 val existing = kubernetesConfigRepo.findByNamespaceAndDeploymentAndYaml(
168                         config.namespace, config.deployment, config.yaml
169                 ) ?: throw e
170                 val id = existing.application?.id ?: throw e
171                 throw ExistingApplicationException(id)
172             }
173 
174             val app = ApplicationEntity(kubernetesConfig = saved, id = oldApp?.id)
175             return saveApplication(app)
176         }
177 
178         private fun saveConfiguration(config: OpenNebulaConfigEntity, oldApp:
        ↪  ApplicationEntity? = null): Long {
179             val saved: OpenNebulaConfigEntity
180 
181             try {
182                 saved = openNebulaConfigRepo.save(config)
183             } catch (e: DataIntegrityViolationException) {
184                 val existing =
                    ↪  openNebulaConfigRepo.findByAddressAndLoginAndPasswordAndRoleAndTemplateAndVmgroup(
185                         config.address, config.login, config.password, config.role,
                            ↪  config.template, config.vmgroup
186                 ) ?: throw e
187                 val id = existing.application?.id ?: throw e
188                 throw ExistingApplicationException(id)
189             }
190 
191             val app = ApplicationEntity(openNebulaConfig = saved, id = oldApp?.id)
192             return saveApplication(app)
193         }
194 
195         private fun saveApplication(app: ApplicationEntity): Long {
196             return appRepository.save(app).id
197                     ?: throw IllegalStateException("Id for a new application was not
                        ↪  generated")
198         }
199 
200         private fun <Result> result(method: String, result: Result): Result {
201             logger.info("$method result: $result")
202             return result
203         }
204 
205         private fun <Result> result(method: String, result: Array<Result>): Array<Result> {
```

```
206             logger.info("$method result: ${result.contentToString()}")
207             return result
208         }
209 
210         private fun getApp(id: Long) = appRepository.findByIdOrNull(id) ?: throw
            ↪ NoSuchApplicationException(id)
211 
212         private fun getAppInfo(client: AppClient) = client.getAppInstances().map {
213             try {
214                 AppInstanceResponse(it.getCPULoad(), it.getRAMLoad(), it.getName())
215             } catch (e: AppClientException) {
216                 logger.error("getAppInfo: unable to get instance info", e)
217                 throw e
218             }
219         }.toTypedArray()
220 }
```

Листинг 2 содержит исходный код фабрики объектов типа ”AppClient”, входящей в состав клиентской библиотеки. С помощью фабрики можно получить доступ к программному интерфейсу поддерживаемой платформы облачных вычислений, имея необходимые параметры подключения.

Листинг 2: Исходный код фабрики объектов ”AppClient” (AppClientFactory.kt)

```
1  open class AppClientFactory {
2      companion object {
3          @JvmStatic
4          fun getClient(kubeClient: ApiClient, namespace: String, deployment: String):
           ↪ AppClient {
5              return KubernetesClient(kubeClient, namespace, deployment)
6          }
7 
8          @JvmStatic
9          fun getClient(openNebulaClient: Client, groupID: Int, roleId: Int, templateId:
           ↪ Int): AppClient {
10             return OpenNebulaClient(openNebulaClient, groupID, roleId, templateId)
11         }
12     }
13 }
```

Интерфейс ”AppClient” представлен в листинге 3. Это унифицированный интерфейс, с помощью которого осуществляется взаимодействие с конкретной платформой облачных вычислений.

Листинг 3: Исходный код интерфейса ”AppClient” (AppClient.kt)

```
/**
 * Interface to access elastic application working in a virtualized infrastructure.
 */
interface AppClient {
    /**
     * Requests information about currently working application instances from
↪   infrastructure provider.
     */
    @Throws(AppClientException::class)
    fun getAppInstances(): Array<AppInstance>

    /**
     * Scales application instances count by {@param count}.
     * Instances are added if {@param count} is positive and removed if it is negative.
     * If {@param count} is zero nothing will be done.
     */
    @Throws(AppClientException::class)
    fun scaleInstances(count: Int)
}
```

Реализация описанного выше интерфейса ”AppClient” для одной из интегрируемых платформ под названием ”Kubernetes” представлена в листинге 4.

Листинг 4: Исходный код реализации интерфейса ”AppClient” для ”Kubernetes” (KubernetesClient.kt)

```
private const val DEPLOYMENT_SELECTOR = "app"

open class KubernetesClient(
    apiClient: ApiClient, private val namespace: String, private val deployment: String
) : AppClient {
    private val coreApi = CoreV1Api(apiClient)
    private val metricsApi = MetricsV1Beta1Api(apiClient)
    private val appsApi = AppsV1Api(apiClient)

    override fun getAppInstances(): Array<AppInstance> {
        val dep = getDeployment()

        val selector = getDeploymentSelector(dep)
        val labelSelector = "$DEPLOYMENT_SELECTOR=$selector"

        val pods = getPods(labelSelector)

```

```
18          return pods.items.mapNotNull { KubernetesInstance.create(it, this)
            ↪  }.toTypedArray()
19      }
20
21      override fun scaleInstances(count: Int) {
22          if (count == 0) return
23
24          val scale: V1Scale
25
26          try {
27              scale = appsApi.readNamespacedDeploymentScale(deployment, namespace,
                ↪  false.toString())
28          } catch (e: ApiException) {
29              throw AppClientException(e)
30          }
31
32          val currentCount = scale.spec?.replicas
33              ?: throw AppClientException("Unable to get replica count for deployment
                ↪  \"$deployment\"")
34          scale.spec?.replicas = currentCount + count
35
36          try {
37              appsApi.replaceNamespacedDeploymentScale(deployment, namespace, scale, null,
                ↪  null, null)
38          } catch (e: ApiException) {
39              throw AppClientException(e)
40          }
41      }
42
43      internal fun getMetricsPerPod() = metricsApi.getPodMetrics(namespace)
44
45      private fun getDeployment() =
46          try {
47              appsApi.readNamespacedDeployment(deployment, namespace, null, null, null)
48                  ?: throw AppClientException("Deployment \"$deployment\" was not found in
                    ↪  namespace \"$namespace\"")
49          } catch (e: ApiException) {
50              throw AppClientException(e)
51          }
52
53
54      private fun getPods(labelSelector: String) =
55          try {
56              coreApi.listNamespacedPod(namespace, null, null, null, null, labelSelector,
                ↪  null, null, null, null)
57                  ?: throw AppClientException("Pods list of \"$deployment\" was not found
                    ↪  in namespace \"$namespace\"")
```

```
58          } catch (e: ApiException) {
59              throw AppClientException(e)
60          }
61
62      private fun getDeploymentSelector(dep: V1Deployment) =
        ↪  dep.spec?.selector?.matchLabels?.get(DEPLOYMENT_SELECTOR)
63          ?: throw AppClientException("Deployment \"$deployment\" has no
            ↪  \"$DEPLOYMENT_SELECTOR\" selector")
64  }
```

Взаимодействие с ”OpenNebula” осуществляется при помощи ”OpenNebulaClient”, представленного в листинге 5.

Листинг 5: Исходный код реализации интерфейса ”AppClient” для ”OpenNebula” (OpenNebulaClient.kt)

```
1   open class OpenNebulaClient(
2       private val client: Client, private val groupId: Int, private val roleId: Int,
        ↪  private val templateId: Int
3   ) : AppClient {
4
5       override fun getAppInstances(): Array<AppInstance> =
6           getVirtualMachines().map { OpenNebulaInstance(it) }.toTypedArray()
7
8       override fun scaleInstances(count: Int) {
9           if (count > 0) addInstances(count)
10          else if (count < 0) removeInstances(-count)
11      }
12
13      private fun getVirtualMachines(): List<VirtualMachine> {
14          val pool = VirtualMachinePool(client)
15          throwIfError(pool.info())
16          val vmIDs = getVMIdentifiers()
17
18          return pool.filter { vmIDs.contains(it.id()) }
19      }
20
21      private fun addInstances(count: Int) {
22          val template = Template(templateId, client)
23          val roleName = getString(getRole(), "NAME")
24          val groupTemplate = "VMGROUP = [ VMGROUP_ID = \"$groupId\", ROLE = \"$roleName\"
            ↪  ]"
25          for (i in 0 until count) {
26              val response = template.instantiate("", false, groupTemplate, false)
27              throwIfError(response)
28          }
29      }
```

```
30
31      private fun removeInstances(count: Int) {
32          val list = getVirtualMachines()
33
34          val toRemove = min(count, list.size)
35          if (toRemove == 0) return
36
37          list.take(toRemove).forEach { throwIfError(it.terminate()) }
38      }
39
40      private fun getRole(): Node =
41          findRole() ?: throw AppClientException("Failed to find role with id $roleId in VM
            ↪ group $groupId")
42
43      private fun getVMIdentifiers(): Set<Int> {
44          val role = getRole()
45          val vmList = getString(role, "VMS")
46          return vmList.split(",").filter { it.isNotBlank() }.map { it.toInt()
            ↪ }.toSortedSet()
47      }
48
49      private fun findRole(): Node? {
50          val group = VMGroup(groupId, client)
51
52          val root = getRootElement(group.info())
53          val roleList = getNodeList(root, "ROLES/ROLE")
54          for (i in 0 until roleList.length) {
55              val item = roleList.item(i)
56              val id = getNumber(item, "ID").toInt()
57              if (id == roleId)
58                  return item
59          }
60
61          return null
62      }
63  }
```

Для получения доступа к статистике использования ресурсов каждым конкретным контейнером или виртуальной машиной используется интерфейс ”AppInstance”. Его исходный код представлен в листинге 6;

Листинг 6: Исходный код интерфейса ”AppInstance” (AppInstance.kt)

```
1  /**
2   * Interface to access elastic application instance information.
3   * It is either a container or a virtual machine.
4   *
```

```
 * Be aware: all of the information is cached in object's fields.
 * Instance might not exist by the time data is requested.
 */
interface AppInstance {
    /**
     * CPU load is represented in CPU core usage in the last minute.
     * For instance, if this value is 0.5, then half of a CPU core was used in the minute before data was requested.
     * CPU usage is always absolute value, it is the same on 12-core machine and 36-core machine.
     */
    fun getCPULoad(): Double

    /**
     * RAM usage is represented in bytes of RAM used in the last minute.
     * For example, if this value is 1024 then 1 KiB of RAM was used by this instance in the minute before data was requested.
     * RAM usage is always absolute value, it is the same both on 16 GiB RAM machine and 1 GiB RAM machine.
     */
    fun getRAMLoad(): Double

    /**
     * Name of the instance in terms of virtualized infrastructure provider.
     */
    fun getName(): String
}
```

Листинг 7 содержит реализацию описанного ранее интерфейса ”AppInstance” для ”Kubernetes”.

Листинг 7: Исходный код реализации интерфейса ”AppInstance” для ”Kubernetes” (KubernetesInstance.kt)

```
internal data class KubernetesInstance(
    private val name: String,
    private val CPULoad: Double,
    private val RAMLoad: Double
) : AppInstance {

    companion object {
        fun create(pod: V1Pod, client: KubernetesClient): KubernetesInstance? {
            val name = pod.metadata?.name ?: throw AppClientException("Pod name or whole metadata is unknown")
            val cpuLoad = getUsage(name, "cpu", client) ?: return null
            val ramLoad = getUsage(name, "memory", client) ?: return null

```

```
13              return KubernetesInstance(name, cpuLoad, ramLoad)
14          }
15
16          private fun getUsage(podName: String, metricName: String, client:
            ↪ KubernetesClient): Double? {
17              val podContainers = getPodMetrics(podName, client)?.containers
18
19              return podContainers?.fold(DoubleAdder()) { acc, container ->
20                  val usage = container.usage?.get(metricName)?.number
21                      ?: throw AppClientException("Usage of \"$metricName\" was not found
                        ↪  for container \"${container.name}\"")
22                  acc.add(usage.toDouble())
23                  acc
24              }?.toDouble()
25          }
26
27          private fun getPodMetrics(podName: String, client: KubernetesClient):
            ↪ MetricsV1Beta1PodMetrics? {
28              val allMetrics = client.getMetricsPerPod().items ?: return null
29
30              for (item in allMetrics) {
31                  if (item.metadata?.name == podName)
32                      return item
33              }
34
35              return null
36          }
37      }
38
39      override fun getCPULoad() = CPULoad
40
41      override fun getRAMLoad() = RAMLoad
42
43      override fun getName() = name
44  }
```

Сбор статистики использования ресурсов виртуальной машиной под управлением ”OpenNebula” может быть осуществлён с помощью ”OpenNebulaInstance”, представленного в листинге 8.

Листинг 8: Исходный код реализации интерфейса ”AppInstance” для ”OpenNebula” (OpenNebulaInstance.kt)

```
1  internal class OpenNebulaInstance(vm: VirtualMachine) : AppInstance {
2      private val name: String = vm.name ?: throw AppClientException("Unknown vm name")
3      private val cpuLoad = getUsage("CPU", vm) / 100
```

```
4       private val memoryLoad = getUsage("MEMORY", vm) * 1024
5
6       override fun getCPULoad() = cpuLoad
7
8       override fun getRAMLoad() = memoryLoad
9
10      override fun getName() = name
11
12      private fun getUsage(resource: String, vm: VirtualMachine): Double {
13          val root = getRootElement(vm.info())
14          return getNumber(root, "MONITORING/$resource")
15      }
16
17      override fun toString(): String {
18          return "OpenNebulaNode(name='$name', cpuLoad=$cpuLoad, memoryLoad=$memoryLoad)"
19      }
20  }
```

ПРИЛОЖЕНИЕ 2. ПЕРЕЧЕНЬ ТАБЛИЦ БАЗЫ ДАННЫХ

Таблица 2: Описание полей сущности ”application_entity”

Имя	**Описание**
id	Первичный ключ сущности, уникальное число для идентификации записи.
kubernetes_config	Первичный ключ сущности ”kubernetes_config_entity” для связи приложения и его конфигурации в системе ”Kubernetes”. Может отсутствовать, если это приложение сконфигурировано в ”OpenNebula”.
open_nebula_config	Первичный ключ сущности ”open_nebula_config_entity” для связи приложения и его конфигурации в системе ”OpenNebula”. Может отсутствовать, если это приложение сконфигурировано в ”Kubernetes”.

Таблица 3: Описание полей сущности ”kubernetes_config_entity”

Имя	**Описание**
id	Первичный ключ сущности, уникальное число для идентификации записи.
namespace	Имя того namespace ”Kubernetes”, в котором развёрнуто управляемое приложение.
deployment	Имя того deployment ”Kubernetes”, к которому относится управляемое приложение.
yaml	Конфигурация подключения к API ”Kubernetes” в формате ”YAML”, эту конфигурацию ещё называют ”kube config”.

Таблица 4: Описание полей сущности ”open_nebula_config_entity”

Имя	**Описание**
id	Первичный ключ сущности, уникальное число для идентификации записи.
address	URL, по которому расположен XML RPC API платформы ”OpenNebula”.
login	Логин пользователя.
password	Пароль пользователя.
role	Идентификатор роли внутри группы ВМ, к которой относится управляемое приложение.
template	Идентификатор шаблона VM, согласно которому будут создаваться новые ВМ управляемого приложения.
vmgroup	Идентификатор группы ВМ, к которой относится управляемое приложение.

ПРИЛОЖЕНИЕ 3. ПЕРЕЧЕНЬ МЕТОДОВ И ФОРМАТОВ СООБЩЕНИЙ REST API ВЕБ-СЕРВИСА

Таблица 5: Описание запроса сохранения параметров подключения к ”Kubernetes”-приложению

Метод	POST		
Адрес	/api/v1/kubernetes/{namespace}/{deployment}/		
Параметры запроса	**Имя**	**Тип**	**Описание**
	namespace	String	Namespace, к которому относится приложение.
	deployment	String	Deployment, к которому относится приложение.
Тело запроса	Конфигурация подключения к ”Kubernetes” в формате YAML.		

Таблица 6: Описание ответа на запросы сохранения и удаления параметров подключения в случае успеха проведения операции

Код ответа	200		
Тело ответа	Тело ответа передаётся в формате JSON.		
	Имя	**Тип**	**Описание**
	appId	int64	Идентификатор приложения, которое только что было создано или удалено.

Таблица 7: Описание запроса обновления сохранённых параметров подключения к ”Kubernetes”-приложению

Метод	PUT		
Адрес	/api/v1/kubernetes/{namespace}/{deployment}/{id}/		
Параметры запроса	**Имя**	**Тип**	**Описание**
	namespace	String	Namespace, к которому относится приложение.
	deployment	String	Deployment, к которому относится приложение.
	id	int64	Идентификатор существующей конфигурации, которая должна быть обновлена.
Тело запроса	Конфигурация подключения к ”Kubernetes” в формате YAML.		

Таблица 8: Описание запроса списка сохранённых приложений

Метод	GET
Адрес	/api/v1/app

Таблица 9: Описание запроса удаления сохранённого приложения

Метод	DELETE		
Адрес	/api/v1/app/{id}		
Параметры запроса	**Имя**	**Тип**	**Описание**
	id	int64	Идентификатор удаляемого приложения.

Таблица 10: Описание запроса состояния сохранённого приложения

Метод	GET		
Адрес	/api/v1/app/{id}		
Параметры запроса	**Имя**	**Тип**	**Описание**
	id	int64	Идентификатор приложения, состояние которого должно быть возвращено.

Таблица 11: Описание запроса масштабирования приложения

Метод	PATCH		
Адрес	/api/v1/app/{id}		
Тело запроса	Тело запроса передаётся в формате JSON.		
	Имя	**Тип**	**Описание**
	incrementBy	int64	Количество ВМ или контейнеров, на которое должно быть увеличены доступные приложению ресурсы. В случае, если число отрицательное, такое количество ВМ или контейнеров приложения будет отключено.

Таблица 12: Описание запроса сохранения параметров подключения к ”OpenNebula”-приложению

Метод	POST		
Адрес	/api/v1/opennebula/		
Тело запроса	Тело запроса передаётся в формате JSON.		
	Имя	**Тип**	**Описание**
	address	String	URL, по которому находится XML RPC API OpenNebula, к которой осуществляется подключение.
	login	String	Логин пользователя, с которым осуществляется подключение.
	password	String	Пароль пользователя, с которым осуществляется подключение.
	password	String	Пароль пользователя, с которым осуществляется подключение.
	role	int64	Идентификатор роли, к которой относятся ВМ управляемого приложения.
	template	int64	Идентификатор шаблона, по которому создаются новые ВМ управляемого приложения.
	vmgroup	int64	Идентификатор группы ВМ, в которую входят ВМ управляемого приложения.

Таблица 13: Описание запроса обновления параметров подключения к ”OpenNebula”-приложению

Метод	PUT		
Адрес	/api/v1/opennebula/{id}/		
Тело запроса	Тело запроса передаётся в формате JSON.		
	Имя	**Тип**	**Описание**
	address	String	URL, по которому находится XML RPC API OpenNebula, к которой осуществляется подключение.
	login	String	Логин пользователя, с которым осуществляется подключение.
	password	String	Пароль пользователя, с которым осуществляется подключение.
	password	String	Пароль пользователя, с которым осуществляется подключение.
	role	int64	Идентификатор роли, к которой относятся ВМ управляемого приложения.
	template	int64	Идентификатор шаблона, по которому создаются новые ВМ управляемого приложения.
	vmgroup	int64	Идентификатор группы ВМ, в котороую входят ВМ управляемого приложения.
Параметры запроса	**Имя**	**Тип**	**Описание**
	id	int64	Идентификатор конфигурации OpenNebula приложения, которая должна быть обновлена.

Таблица 14: Описание общего для всех запросов формата описания ошибки

Код ответа	**Код**	**Причина**
	500	Внутренняя ошибка сервера, вызванная непредусмотренным набором параметров запроса.
	400	В полученном запросе обнаружена ошибка, запрос нужно сформировать заново.
	404	Полученный запрос был направлен на неизвестный серверу путь, запрос нужно сформировать заново.

Тело ответа	Тело ответа передаётся в формате JSON.		
	Имя	**Тип**	**Описание**
	timestamp	String	Время возникновения ошибки. Формат записи времени соответствует стандарту ISO 8601.
	status	int64	HTTP-код ответа.
	error	String	Текстовое описание HTTP-кода ответа на английском языке.
	message	String	Текстовое описание возникшей ошибки на английском языке.
	path	String	Путь на сервере, на который был получен запрос. Вычисляется относительно корневого пути, без адреса самого сервера.

Таблица 15: Описание ответа на запрос обновления конфигурации

Код ответа	200
Тело ответа	Нет

Таблица 16: Описание ответа на запрос списка сохранённых приложений

Код ответа	200
Тело ответа	Тело ответа передаётся в формате JSON. **Имя** **Тип** openNebulaApps OpenNebulaApp[] kubernetesApps KubernetesApp[]
Тип данных OpenNebulaApp	Объекты этого типа используются для описания сохранённой конфигурации параметров подключения к приложению под управлением ”OpenNebula”. **Имя** **Тип** **Описание** appId int64 Идентификатор этого приложения. address String URL-адрес, по которому расположен XML RPC API системы ”OpenNebula”. role int64 Идентификатор роли, к которой относятся ВМ этого приложения. template int64 Идентификатор шаблона, по которому создаются ВМ этого приложения. template int64 Идентификатор группы ВМ, в которую входят ВМ этого приложения.
Тип данных KubernetesApp	Объекты этого типа используются для описания сохранённой конфигурации параметров подключения к приложению под управлением ”Kubernetes”. **Имя** **Тип** **Описание** appId int64 Идентификатор этого приложения. namespace String Namespace, в который входит это приложение. deployment String Deployment, в который входит это приложение.

Таблица 17: Описание ответа на запросы состояния и масштабирования приложения

Код ответа	200
Тело ответа	Тело ответа передаётся в формате JSON. Сам ответ является массивом объектов типа Instance.
Тип данных Instance	Объекты этого типа используются для описания состояния одного контейнера или ВМ приложения.

Имя	**Тип**	**Описание**
name	String	Имя контейнера или ВМ, по которому его можно идентифицировать внутри платформы облачных вычислений.
cpuload	float64	Загрузка ЦПУ (CPU). Это значение показывает какое количество вычислительных ядер использовалось в среднем последнюю минуту. Например, 2.3 означает, что 2 вычислительных ядра исполняло команды только этой задачи, а третье 70% времени исполняло команды других задач.
ramload	float64	Загрузка оперативной памяти (RAM). Это значение показывает какое количество байт памяти было занято данными этой задачи в последнюю минуту. Например, 1024 означает, что 1 КиБ памяти был использован этой задачей.