УДК 681.3.06 DOi 10.30837/ bi.2021.1(96).06

Д.А. Золотарев

Кандидат физико-математических наук, г. Харьков, Украина, denis@zolotariov.org.ua,
ORCID: 0000-0003-4907-7810

ОБ ОДНОМ ПОДХОДЕ К АВТОМАТИЗИРОВАННОМУ ВЕРСИОНИРОВАНИЮ ПРОГРАММНОГО ОКРУЖЕНИЯ ВО ВРЕМЯ ЭТАПА РАЗРАБОТКИ

Рассмотрены, исследованы и разработаны теоретические и практические рекомендации относительно формирования механизма версионирования программного окружения времени разработки, который является автоматизированным, гибким и универсальным, и не зависит от конкретных языков программирования, компонентов программного обеспечения и других особенностей программного окружения. Объединяет в себе два подхода: универсальный по-файловый и индивидуальный для программных продуктов, поддерживающих внутренние механизмы создания простых и инкрементальных файловых копий данных. Приведен анализ элементов такого механизма и функциональная нагрузка каждого из них, обоснована его роль в общей структуре.

ВЕРСИОНИРОВАНИЕ СОСТОЯНИЯ, ПРОГРАММНОЕ ОКРУЖЕНИЯ, РАЗРАБОТКА ПО, API, BASH

Золотарьов Д.О. Про один підхід до автоматизованого версіонування програмного оточення під час етапу розробки. Розглянуті, досліджені та розроблені теоретичні та практичні рекомендації щодо формування механізму версіонування програмного оточення часу розробки, що є автоматизованим, гнучким та універсальним, й не залежить від конкретних мов програмування, компонентів програмного забезпечення та інших особливостей програмного оточення. Поєднує у собі два підходи: універсальний пофайловий та індивідуальний для програмних продуктів, що підтримують внутрішні механізми створення простих та інкрементальних файлових копій даних. Наведений аналіз складових елементів такого механізму та функціонального навантаження кожного з них, обґрунтована його роль у загальній структурі.

ВЕРСІОНУВАННЯ СТАНУ, ПРОГРАМНЕ ОТОЧЕННЯ, РОЗРОБКА ПЗ, API, BASH

Zolotariov D. One approach to automated versioning of the software environment during development stage. In article considered and investigated eheoretical and practical recommendations for the formation of the mechanism of versioning the software environment of development time, which is automated, flexible and universal, and does not depend on specific programming languages, software components and other features of the software environment, are considered, researched and developed. It combines two approaches: universal file-by-file and individual for software products that support internal mechanisms for creating simple and incremental file copies of data. The analysis of the constituent elements of such a mechanism and the functional load of each of them is given, its role in the general structure is substantiated.

STATE VERSIONING, SOFTWARE ENVIRONMENT, SOFTWARE DEVELOPMENT, API, BASH

Введение

Последние несколько лет в IT-индустрии ознаменовались стремительным ростом разнообразных технологий обработки данных, включая средства разработки [1], вследствие развития микросервисной архитектуры [2-8], сервисов распределенной обработки данных [9-13] и безсерверных вычислений, высоконагруженных медиа-сайтов и веб-сервисов [14], а также IoT [15-16]. Вместе с развитием уже существующих создаются все новые и новые программные решения. А значит, непрерывно изменяется и технологический стек разрабатываемого программного обеспечения для конечных пользователей.

Это программное обеспечение сегодня разделено на множество отдельных компонентов для хранения, обработки и выдачи данных, которые подбираются разработчиками в зависимости от ожидаемого результата. Полный набор этих компонентов и формирует программное окружение разработки конечного программного продукта.

При выборе нового программного решения в качестве такого компонента на плечи разработчиков и архитекторов ложится не только проверка его возможностей и определение соответствия конкретной задаче, но и его настройка для повышения эффективности работы через непрерывное экспериментирование с используемым инструментарием, которое идет рука об руку с разработкой целевого программного продукта.

Такое экспериментирование имеет своей целью поиск способов использования преимуществ этих компонентов и нивелирования их недостатков, и включает в себя предварительное, нагрузочное и интеграционное тестирования, позволяя определить эксплуатационные особенности компонента, которые нельзя выявить на статических этапах проектирования и разработки.

В процессе такого экспериментирования разработчиками производится множество изменений в компонентах, которые имеют файлы настроек либо сохраняют в процессе работы свое состояние (набор из пользовательских или само-сгенерированных в процессе работы данных, изменяющих реакцию компонента на внешние запросы). Эти изменения вносятся неконтролируемым образом до момента выявления стабильной рабочей конфигурации, удовлетворяющей требованиям проекта. И их история нигде не сохраняется.

Отсутствие истории изменения настроек или состояния компонента может приводить к критическим

ситуациям и, как следствие, значительному замедлению процесса разработки в случае их утери с обновлением компонента, сбоя на сервере или просто по ошибке. Их восстановление не гарантировано и может занимать очень длительное время из-за того, что разработчик вынужден в таком случае опираться только на свою память.

Безусловно, можно полагаться на полное резервное копирование сервера, предусмотренное всеми современными облачными технологиями и реальными мощностями, производимое обычно раз в неделю или реже. Но за этот период программное окружение и его параметры могут измениться несколько десятков и даже сотен раз.

Единственным альтернативным решением данной проблемы на текущий момент является добавление этих файлов под систему контроля версий (VCS). Но этот подход имеет ряд недостатков. Первый — в процессе создания программного продукта множество сторонних компонентов устанавливаются по мере необходимости и удаляются так же. Что затрудняет централизованное слежение за изменениями. Второй — их файлы настроек и состояния располагаются в различных не связанных между собой местах на сервере. И при значительном количестве таких компонентов может оказаться, что под контроль VCS нужно внести практически весь сервер, что не эффективно. И третий — фиксация изменений производится только вручную.

Поэтому актуальной задачей является разработка механизма хранения подробной истории изменений конфигурации программного окружения для серверов, использующихся во время разработки. Или другими словами — механизма версионирования программного окружения.

Целью этой статьи является разработка и обоснования теоретических и практических рекомендаций относительно формирования механизма версионирования программного окружения времени разработки. Этот механизм должен быть автоматизированным, максимально упрощенным для пользователей и универсальным, то есть минимально зависеть от конкретных компонентов программного обеспечения. А также нуждаться в минимуме сторонних приложений для своей работы, чтобы иметь максимальную скорость развертывания на чистом сервере и минимальную стоимость обслуживания.

Задача статьи – определить необходимые элементы такого механизма, предоставить для каждого из них анализ функциональной нагрузки и обосновать выбор инструментов для их решения.

1. Общий подход к построению механизма

Для решения поставленной задачи необходимо отметить, что механизм версионирования должен работать аналогично системам VCS, то есть создавать версии программного окружения в локальное и удаленное хранилища, позволяющие их просматривать и использовать. Делать это в автоматическом режиме без участия пользователя. Последнее обусловлено тем, что изменения, вносимые разработчиками, администраторами или специалистами DevOps в текущие настройки программного обеспечения сервера, например: nginx, systemd, environment, redis, ... — за период одного рабочего дня могут происходить множество раз. Следить за ними в ручном режиме не возможно и экономически не целесообразно из-за высокой стоимости времени разработчика.

Механизм должен обеспечивать создание версий для групп файлов и директорий с полным сохранением их структуры, начиная от корня, и прав доступа. Следует учесть, что в одну версию могут входить также несвязанные между собой директории и файлы, относящиеся к одному программному обеспечению. Примером может служить система доставки контента Apache Kafka, сохраняющая свои настройки и состояние в различных директориях [17].

Если программный компонент имеет встроенные инструменты создания копий состояния, как, например, СУБД MySQL [18] или Redis [19], в том числе инкрементальной копии, как, например, поисковая система Elasticsearch [20], то в механизме должна присутствовать возможность использовать эти особенности и преимущества.

Создаваемые версии практически всегда содержат конфиденциальные данные, поэтому хранение их во внешних хранилищах данных должно быть безопасным.

Таким образом, цели и подходы у существующих VCS и разрабатываемого механизма одинаковые. Отличие в способе генерации версии и автоматизации этого процесса. Системы контроля версий используются для файлов разрабатываемого программного обеспечения, которые находятся всегда в одной директории, и позволяют создание новой версии только вручную. Разрабатываемый же механизм — для программного окружения, данные которого могут находиться в файлах практически в любой директории сервера или рабочей станции, а также храниться только в оперативной памяти. Кроме того создание версии полностью автоматическое.

Поэтому системы контроля версий не являются конкурентами разрабатываемому механизму, хотя и решают схожую задачу.

С учетом выше сказанного, механизм версионирования должен обладать следующими характеристиками. Копирование должно производиться относительно корня диска, для каждого файла и его родителей должны сохраняться все связанные атрибуты inode: владелец, права доступа и прочие. Результат должен сжиматься современными алгоритмами сжатия (например, использовать хорошо себя зарекомендовавший алгоритм сжатия в реальном времени zstd [21]). А передаваемый на внешние сервисы хранения

данных архив должен быть зашифрован с использованием ассиметричных алгоритмов шифрования.

Причем на самом сервере должен храниться только публичный ключ. Эта мера позволит избежать каких-либо последствий с безопасностью в случае его компрометации. Приватный ключ должен храниться только у разработчика и администраторов. Для автоматизации восстановления версий возможно его хранение также на одном доверенном сервере с повышенными мерами безопасности.

Механизм создания версии состояния приложения может быть двух видов: универсальный и индивидуальный. Первый подразумевает простое пофайловое копирование. Второй — использование программ, специально предназначенных для создания резервной копии состояния определенного программного обеспечения, к которым относятся в частности: базы данных, поисковые системы, менеджеры очередей и т.д. — требующие выполнения определенной последовательности операций для создания копии состояния на диске, часто инкрементальной.

На сервере всегда должна оставаться как минимум одна незашифрованная, но сжатая, версия. Кроме восстановления она служит для сравнения с текущими файлами и директориями при создании новой.

Принципиальная блок-схема работы механизма версионирования настроек и состояния программного обеспечения сервера представлена на рис. 1 ниже.

Процесс создания версии состояния для приложения начинается с определения «универсальный – индивидуальный». В первом случае производится простое клонирование файлов и директорий и пофайловое сравнение для определения присутствия отличий. Во втором — выполняется набор команд, который отвечает за генерацию таких файлов и их клонирование, а также определение того, появились ли изменения с момента создания последней копии.

Если наличие изменений подтверждено, из клонированных inode создается единый архив, который затем зашифровывается и загружается во внешнее хранилище данных.

О результатах работы отправляются уведомления. После чего очищаются ненужные более локальные версии и клонированные файлы.

Рассмотрим далее более подробно каждый важный элемент механизма.

2. Сравнение версий

Если компоненты хранения состояния, имеющие возможность создания инкрементальных резервных копий данных, самостоятельно определяют наличие изменения с момента создания последней копии, то при простом пофайловом копировании рекомендуется использовать стандартные инструменты сравнения.

Рекомендуется для этой цели использовать в достаточной степени гибкий и быстрый компаратор из системы контроля версий Git, как самой распространенной и надежной [22]. Кроме того, можно использовать любой инструмент сравнения файлов и директорий, позволяющий решать эту задачу с достаточной скоростью и эффективностью, в том числе простая программа diff.

Другими словами, инструмент сравнения выбирается исходя из требования, чтобы нагрузка от его выполнения была не заметна пользователю, а время выполнения не было узким местом механизма версионирования. При этом он не должен иметь как ложных срабатываний, так и пропусков срабатывания при наличии изменений.

3. Взаимодействия с API компонентов

Для удобства взаимодействия с компонентами со встроенными инструментами создания копий состояния рекомендуется разработать API-инструментарий на языке оболочки bash или любом другом, поддерживающем сетевое взаимодействие и вызов внешних программ.

Такие компоненты либо имеют клиентские программы, реализующие свой протокол взаимодействия (например, PostgreSQL или Jenkins) или поддерживают REST-взаимодействие через HTTP-протокол (например, Elasticsearch).

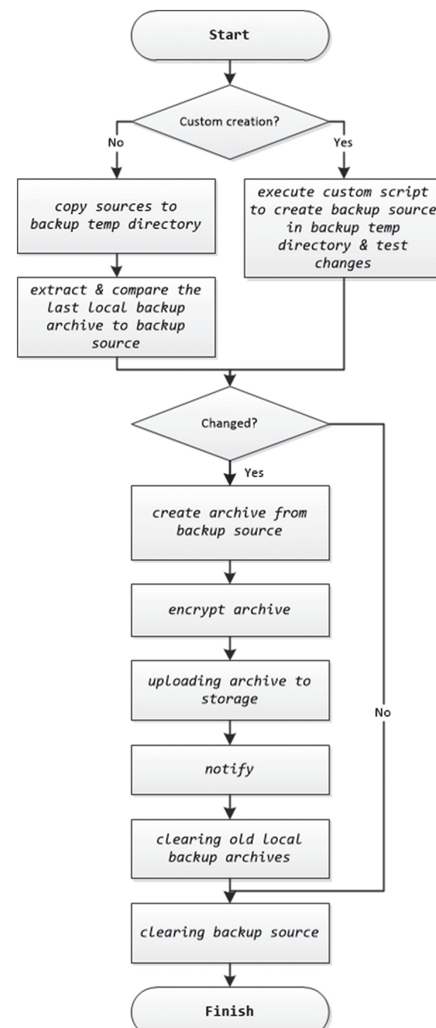

Рис. 1. Блок-схема механизма версионирования

Каждый отдельный модуль API-инструментария может иметь независимый от других внутренний механизм работы, при этом сохраняя общий интерфейс. А свои параметры получать либо полностью из командной строки при вызове, либо также из переменных окружения.

Несмотря на кажущуюся простоту, этот инструментарий должен быть достаточно гибким. Каждый программный продукт имеет свои значительные отличия в API-интерфейсе. Но даже в рамках одного типа REST каждый поддерживает его со своими особенностями — Elasticsearch, например, поддерживает стандарт REST в неполном виде [23]. Кроме того, версии одного продукта могут также иметь свои особенности API-механизма.

4. Хранение версий

Если в качестве локального хранилища может выступать просто директория, то удаленные хранилища достаточно разнообразны по способу организации и хранения.

Для удаленного хранения рекомендуется использовать системы холодного хранения данных. Желательно, несколько независимых.

В качестве таковых могут выступать сервера: Amazon S3, DigitalOcean, свои сервера или даже Dropbox или подобные файловые сервисы.

Последние могут также выполнять важную функцию автоматического копирования версий на несколько независимых носителей информации за счет автоматической фоновой синхронизации удаленного хранилища с локальным, где установлено приложение. Это может оказаться важным, если потребуется получить доступ к версии со стороннего компьютера.

5. Системные сообщения

Не все версии являются одинаково важными и для разработчиков, и администраторов или специалистов DevOps. Одни изменения влияют только на область задач конкретного разработчика, другие — могут оказывать широкое влияние на весь конечный продукт или его часть, за которую ответственны несколько специалистов.

Для разделения по степени важности, а также для удобства и универсальности отправки сообщений, следует использовать несколько каналов связи. Рекомендуется использовать как минимум два:

- только для разработчика,
- для всей группы, участвующей в разработке конечного продукта.

Блок-схема механизма представлена на рис. 2.

Его задача — отправлять персонализированные (только текущему разработчику) и групповые сообщения всем заинтересованным в изменениях в работе сервера лицам. Каналами отправки сообщений могут выступать:

- email,
- Slack, Telegram,
- и прочие.

Электронная почта должна быть обязательно включена для гарантированной доставки, так как ее работа проста, давно отлажена и предсказуема. Кроме того, механизм оповещений должен отвечать и современным требованиям к используемым технологиям — должны быть подключены все необходимые в современной разработке месенджеры.

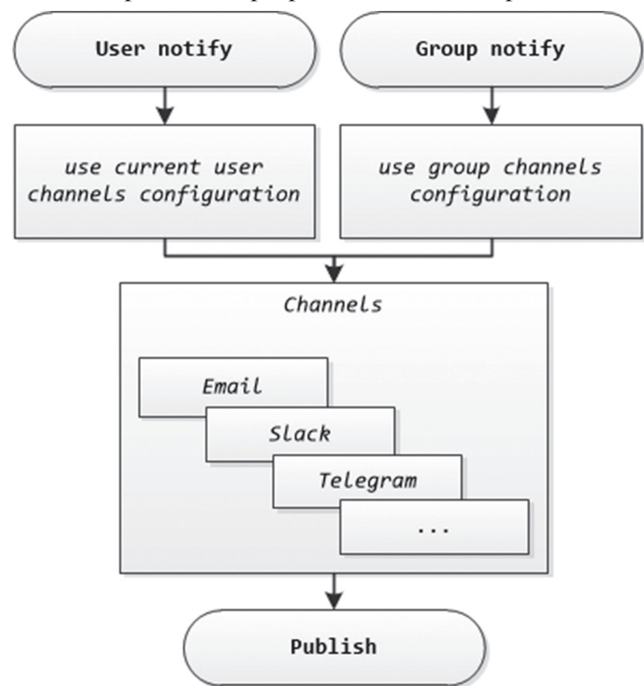

Рис. 2. Блок-схема отправки сообщений

Для группового оповещения лучше всего подходит канал в Slack, но также может быть использована группа в Telegram с использованием Telegram-бота.

Настройки каждого канала рекомендуется определять в переменных окружения. Для перечисленных выше это будут: EMAIL, TELEGRAM_BOT_TOKEN и TELEGRAM_CHAT_ID — определяющие токен Telegram-бота и id чата предназначения, а также SLACK_WEBHOOK_URL — хранящая URL к приложению для публикации сообщения в Slack.

Для удобства отправки одного сообщения в несколько каналов рекомендуется использовать единый центральный механизм (например, bash-скрипт), принимающий в качестве аргументов сообщение и список каналов, по которым оно должно быть отправлено.

Для ускорения работы этого инструментария в качестве центрального узла возможно использовать отдельный сервер на основе менеджера очередей, который принимает сообщение и список каналов связи с адресами, и далее отправляет это сообщение в каждый из них. Таким образом, можно решить проблему задержек при публикации сообщений в достаточно медленные внутренние сервисы или же внешние с негарантированной связью.

Кроме отправки сообщений в процессе создания новой версии программного компонента, этот инструментарий может быть использован также в любых других случаях, требующих внимания разработчика, администраторов или специалистов DevOps.

6. Особенности механизма

Описанное выше аналогичное работе системе контроля версий дополнение дает возможность быстро вернуться к последней или любой более ранней сохраненной копии данных при сбое в работе сервера или при неудачном его обновлении. При восстановлении оно позволяет быстро воссоздать такой же сервер даже в случае недоступности его полной резервной копии.

Применительно к дублированию — такие версии позволяют развернуть сервер на базе новых версий компонентов программного обеспечения с сохранением всех его настроек и состояния. В том числе и без создания полной копии оригинала, т.е. без нагрузки на исходный сервер.

Что может быть полезно, в том числе, при создании тестовой версии production-сервера или обновлении stage-сервера.

Также такой подход к развертыванию среды разработки через использование версий позволяет решить задачу создания нескольких вариантов протестированного окружения. Что минимизирует сбои в работе, а значит, предотвращает непроизводственные финансовые и временные расходы.

7. Практическая реализация

Для проверки возможностей предложенного механизма, он был реализован в виде пакета подпрограмм на языке оболочки bash. В качестве инструмента сравнения и определения наличия изменений в версиях использован компаратор git, в качестве инструмента сжатия — tar с алгоритмом сжатия «xz», в качестве инструмента защиты — gpg с ассиметричным шифрованием.

В качестве локального хранилища — директория «/backups», удаленного — директория в файловом сервисе Dropbox, взаимодействие с API которого осуществлялось при помощи разработанного на языке bash инструментария на базе curl согласно документации сервиса.

В качестве инструмента автоматизации были использованы задачи в cron согласно правилу — одна задача на один программный компонент. Добавление задач было реализовано через универсальную bash-подпрограмму, принимающую в качестве своих аргументов название программного компонента и набор директорий для версионирования. Последнее — только если использовалось простое клонирование файлов.

Версионирование проводилось для следующих программных компонентов: файл переменных окружения «/etc/environment», конфигурации веб-сервера nginx. А также данные из: NoSql СУБД Redis, СУБД MySQL, поисковой системы Elasticsearch. Для первых использовался простой подход файлового клонирования, для вторых — специально разработанный инструментарий на языке оболочки bash, использующий индивидуальные встроенные возможности в каждый из компонентов хранения данных основанных на [18-20].

Для каждого компонента использовалась отдельная директория в локальном и удаленном хранилищах версий. Каждая версия в своем названии имела временную метку с точностью до секунд для легкого определения хронологии.

В качестве каналов системных сообщений использовались email и канал в Telegram. В первом случае использовался пакет утилит mailutils, во втором — специально разработанный инструментарий на языке оболочки bash и curl для взаимодействия с REST-API Telegram согласно документации месенджера.

В процессе полугодовой непрерывной эксплуатации при разработке приложений на PHP/Laravel и Python на низко-производительном сервере разработанный механизм автоматического версионирования программного окружения показал себя как удовлетворяющий требованиям скорости и надежности. Его работа была не заметна пользователям, при этом он не имел ложных срабатываний или пропусков срабатывания при наличии изменений.

Выводы

В работе впервые разработаны и обоснованы теоретические и практические рекомендации относительно формирования механизма версионирования программного окружения во время этапа разработки программного обеспечения, создающего историю изменений программных компонентов.

Этот механизм является автоматизированным, гибким и универсальным, и не зависит от конкретных языков программирования, компонентов программного обеспечения и других особенностей программного окружения.

Доказано, что, несмотря на то, что цели и подходы у систем контроля версий и предложенного механизма одинаковые, они имеют существенные отличия в применении и поэтому не являются конкурентами.

Показано, что такой механизм должен состоять из следующих функциональных элементов: клонирование файлов настроек и состояния, сравнения их с последней версией, сжатия, шифрование и загрузка во внешние хранилища версий, а также отправка уведомлений о появлении новой версии.

Он объединяет два подхода: универсальный пофайловый и индивидуальный для программных продуктов, поддерживающих внутренние механизмы создания простых и инкрементальных файловых копий данных.

Каждый элемент детально описан, показана его функциональная нагрузка и обоснована его роль в общей структуре.

Приведен анализ практической реализации данного механизма для конкретных языков программирования и компонентов программного окружения, разработанного на языке оболочки bash. Отмечено, что в такой реализации его выполнение не заметно пользователям, не имеет ложных срабатываний или пропусков срабатывания при наличии изменений. Кроме того он не нуждается в сторонних приложениях для своей работы.

Показано, что отсутствие истории изменения настроек или состояния компонентов программного обеспечения может приводить к критическим ситуациям и, как следствие, значительному замедлению процесса разработки, а также то, что в результате применения предложенного механизма такие ситуации исчезают. Что ведет к таким преимуществам как усовершенствование процесса разработки, облегчение труда разработчика и сокращение непроизводственных потерь в условиях частого изменения программного окружения.

Список литературы:

[1] *Lee, CY.* Temporal Correlation Analysis of Programming Language Popularity // J. Korean Phys. Soc. 2019. Vol. 75, P. 755–763. https://doi.org/10.3938/jkps.75.755

[2] *da Silva H.H.S., de F. Carneiro G., Monteiro M.P.* An Experience Report from the Migration of Legacy Software Systems to Microservice Based Architecture // 16th International Conference on Information Technology-New Generations (ITNG 2019). Advances in Intelligent Systems and Computing. 2019. Vol 800. https://doi.org/10.1007/978-3-030-14070-0_26

[3] *Rademacher F., Sachweh S., Zündorf A.* A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems // Enterprise, Business-Process and Information Systems Modeling. BPMDS 2020, EMMSAD 2020. Lecture Notes in Business Information Processing. 2020. Vol 387. https://doi.org/10.1007/978-3-030-49418-6_21

[4] *Levcovitz, A., Terra, R., Valente, M.T.* Towards a technique for extracting microservices from monolithic enterprise systems // Proceedings of VEM'15. 2015. P. 97–104.

[5] *Munari S., Valle S., Vardanega T.* Microservice-Based Agile Architectures: An Opportunity for Specialized Niche Technologies // Reliable Software Technologies – Ada-Europe 2018. Ada-Europe 2018. Lecture Notes in Computer Science. V. Vol 10873. https://doi.org/10.1007/978-3-319-92432-8_10

[6] *Bucchiarone, A., Dragoni, N., Dustdar, S., et al.* From monolithic to microservices: an experience report from the banking domain // IEEE Softw. 2018. Vol. 35(3), P. 50–55.

[7] *Sorgalla J., Sachweh S., Zündorf A.* Exploring the Microservice Development Process in Small and Medium-Sized Organizations // Product-Focused Software Process Improvement. PROFES 2020. Lecture Notes in Computer Science. 2020. Vol 12562. https://doi.org/10.1007/978-3-030-64148-1_28

[8] *Zolotariov, D.* The distributed system of automated computing based on cloud infrastructure // Innovative Technologies and Scientific Solutions for Industries. 2020. No. 4 (14), P. 47–55. https://doi.org/10.30837/ITSSI.2020.14.047

[9] *Zolotariov, D.* The mechanism for creation of event-driven applications based on Wolfram Mathematica and Apache Kafka // Innovative Technologies and Scientific Solutions for Industries. 2021. No. 1 (15), P. 53–58. https://doi.org/10.30837/ITSSI.2021.15.053

[10] *Mahapatra, T.* Composing high-level stream processing pipelines // Journal of Big Data. 2020. Vol. 7, No. 81. https://doi.org/10.1186/s40537-020-00353-2

[11] *Jung, S., Kim, Y. & Hwang, E.* Real-time car tracking system based on surveillance videos // EURASIP Journal on Image and Video Processing. 2018. Vol. 2018, No. 133. https://doi.org/10.1186/s13640-018-0374-7

[12] *Ismail, A., Truong, HL. & Kastner, W.* Manufacturing process data analysis pipelines: a requirements analysis and survey // Journal of Big Data. 2019. Vol. 6, No. 1. https://doi.org/10.1186/s40537-018-0162-3

[13] *Kim, YK., Kim, Y. & Jeong, CS.* RIDE: real-time massive image processing platform on distributed environment // EURASIP Journal on Image and Video Processing. 2018. Vol. 2018, No. 39. https://doi.org/10.1186/s13640-018-0279-5

[14] *Kolajo, T., Daramola, O. & Adebiyi, A.* Big data stream analysis: a systematic literature review // Journal of Big Data. 2019. Vol. 6, No. 47. https://doi.org/10.1186/s40537-019-0210-7

[15] *Nasiri, H., Nasehi, S. & Goudarzi, M.* Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities // Journal of Big Data. 2019. Vol. 6, No. 52. https://doi.org/10.1186/s40537-019-0215-2

[16] *Ed-daoudy, A., Maalmi, K.* A new Internet of Things architecture for real-time prediction of various diseases using machine learning on big data environment // Journal of Big Data. 2019. Vol. 6, No. 104. https://doi.org/10.1186/s40537-019-0271-7

[17] Kafka – Apache [Електронний ресурс] // Apache. URL: https://kafka.apache.org/documentation/#configuration. – Назва з екрана.

[18] Backup and Recovery Types – MySQL 8.0 Reference Manual [Електронний ресурс] // MySQL. URL: https://dev.mysql.com/doc/refman/8.0/en/backup-types.html. – Назва з екрана.

[19] Redis Persistence – Redis [Електронний ресурс] // Redis. URL: https://redis.io/topics/persistence. – Назва з екрана.

[20] Snapshot and restore – Elasticsearch Guide [Електронний ресурс] // Elastic. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/snapshot-restore.html. – Назва з екрана.

[21] Compression Comparison Benchmarks: zstd vs brotli vs pigz vs bzip2 vs xz etc – Centmin Mod Community Support Forums [Електронний ресурс] // Sysadmin. URL: https://community.centminmod.com/threads/compression-comparison-benchmarks-zstd-vs-brotli-vs-pigz-vs-bzip2-vs-xz-etc.12764/. – Назва з екрана.

[22] 2021 Version Control Software Comparison: SVN, Git, Mercurial – Time doctor [Електронний ресурс] // Time doctor. URL: https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/. – Назва з екрана.

[23] Update API – Elasticsearch Reference [Електронний ресурс] // Elastic. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-update.html. – Назва з екрана.

Поступила в редколлегию 10.02.2021