Оптимальный подход к разработке программного обеспечения с использованием современных методологий и технических средств

Н.Б. Лазарева
Тихоокеанский государственный университет, Хабаровск

Аннотация: В данной статье описан подход к разработке программного обеспечения, позволяющий использовать современные методологии и технические средства с целью улучшить качество продукта, увеличить скорость его доставки и существенно повысить общую стабильность.
Ключевые слова: GitLab, Jenkins, Kubernetes, Git, DevOps, CI, CD, IaaC, инфраструктура, методологии, программное обеспечение, разработка.

Современные подходы к разработке программного обеспечения выдвигают повышенные требования к серверной инфраструктуре и процессам, связанным с ней. Инфраструктура должна обладать не только высокой надёжностью, доступностью, но и предоставлять быстрые и удобные инструменты для установки, поддержки и обновления программного обеспечения, которое разрабатывается в компании. Очень важной является возможность быстрого горизонтального и вертикального масштабирования всех возможных компонентов инфраструктуры. Команда(ы) разработки и администраторы инфраструктуры стремятся внедрить практики DevOps [1] и CI/CD [2] культур, которые предполагает общее участие в процессе разработки, общую ответственность и отлаженные механизмы технического уровня, которые способствуют реализации этих практик. В идеальной системе это дает существенный прирост к скорости доставки обновлений и новых возможностей программного обеспечения заказчику, обеспечивает надежность, а так же предполагает наличие совершенной документации, основанной на парадигме IaaC (Infrastructure-as-a-Code) [3], описывающей все, что касается процесса разработки и поставки готового решения заказчику.

С учетом вышесказанного, инфраструктуры традиционного типа, основанные на виртуальных машинах и широком применении ручных

операций и/или сложных решений по автоматизации процессов, не могут предоставить требуемого результата и их очень сложно интегрировать с парадигмами DevOps, CI/CD и IaaC. Традиционные решения, а также программное обеспечение для них, не разрабатывались для DevOps, CI/CD и IaaC, поэтому их эксплуатация в рамках этих парадигм хоть и возможна, но требует существенных временных затрат, высокого уровня ответственности инженеров и их стремления поддерживать эти парадигмы.

Современные программные средства позволяют развернуть инфраструктуру, полностью пригодную для совместной работы большого количества людей над продуктом любого размера, оставаясь при этом надежной, масштабируемой и хорошо документированной. Кроме того, они построены таким образом, что IaaC изначально является частью процесса создания и поддержки инфраструктуры, а значит, такую инфраструктуру можно назвать частично самодокументируемой.

Также стоит отметить, что традиционный подход к разработке ПО по модели “Монолит” так же сложно связать с CI/CD, так в этом случае разработка ведется долго и доставка заказчику происходит редко, но с большим количеством изменений. Микросервисная же архитектура [4] позволяет быстро доставлять исправления или новый функционал заказчику, производя изменения лишь в необходимой части ПО, и для этого существуют современные средства, предоставляющие возможности внедрения CI/CD.

Процесс разработки программного обеспечения можно условно поделить на три этапа:

1. Тестирование нового функционала или исправлений в рамках отдельных компонентов продукта (dev уровень). Члены команд(ы) разработки на данном этапе могут вести тестирование своих изменений независимо друг от друга.

2. Принятие изменений из стадии тестирования и перевод

компонентов в предрелизное тестирование (stage уровень). На этом этапе продукт тестируется полностью в условиях, максимально приближенных к реальным.

3. Доставка изменений заказчику путем обновления необходимых компонентов (prod уровень).

В качестве платформы для организации всего процесса разработки в современных реалиях хорошо подходит Kubernetes [5]. Существенными его плюсами являются встроенные возможности масштабирования, отказоустойчивости, балансировки нагрузки, сбор метрик и логов, большое сообщество в интернете. В то же время он требует высокой квалификации инженеров, занимающихся его внедрением и обслуживанием. Однако, при должном уровне квалификации и грамотной интеграции с другими средствами, использующимися для разработки ПО, Kubernetes решает практически все задачи и проблемы, связанные с инфраструктурой. Для организации всех трех этапов разработки ПО, логично развернуть три отдельных кластера Kubernetes. Они могут (и даже должны) отличаться размерами, количеством ресурсов, настройками безопасности и т.п. Самый большой кластер, не требующий при этом жестких политик безопасности и кластер для первого этапа, поскольку на нем будут одновременно тестироваться большое количество компонентов, не обладающих при этом критичными с точки зрения безопасности данными. Кластеры для этапов 2 и 3 должны быть похожи, но с точки зрения безопасности и настроек резервирования и отказоустойчивости, кластер для этапа 3 должен быть максимально продуман и настроен.

Этап 1 подразумевает широкую вариативность, поскольку при большом количестве компонентов ПО, членов команд(ы) разработки и изменений в функционале конечный набор компонентов ПО и его версий на тестовой площадке может быть сильно меняться. Это означает, что на этапе

тестирования необходимо применить механизм, позволяющий реализовать вариативность, задать те параметры для организации тестирования, которые подходят конкретной задаче. Иначе говоря, необходимо дать возможность каждому разработчику самому выбирать компоненты и их версии, которые нужно протестировать, а также предоставить для этого изолированное пространство в кластере Kubernetes. Хорошим вариантом для этого является Jenkins [6], инструмент для реализации CI/CD, максимально гибкий по возможности конфигурации, но при этом очень сложный для создания этой конфигурации и поддержки в дальнейшем.

Этап 2 не подразумевает никакой вариативности - напротив, на этом этапе продукт должен тестироваться только в том виде, в котором он планируется к поставке заказчику и только с теми версиями компонентов, которые включают в себя принятые с этапа 1 изменения и необходимые для поставки заказчику. В этом случае удобно использовать GitLab [7], поскольку он предоставляет возможность очень жестко задавать условия поставки компонентов, обладая при этом простотой настройки и интеграцией с общепринятым механизмом совместной разработки Git [8]. Его поддержка и настройка намного проще, нежели таковые в Jenkins.

Этап 3 аналогично этапу 2 подразумевает строгие настройки - заказчику должны быть доставлены все необходимые компоненты продукта тех версий, которые были полностью протестированы и одобрены к поставке. Поэтому в данном случае также лучше подходит GitLab.

Содержать одновременно два средства, предоставляющих CI/CD, накладно, однако есть возможность решить эту проблему реализацией всей логики CI/CD в GitLab (пайплайны, скрипты и тп), тогда как в Jenkins реализовать лишь меню для свободной конфигурации тестируемых изменений для этапа 1. В этом случае Jenkins будет выступать как web-интерфейс к GitLab-логике, давая возможность ее гибко настраивать. Таким

образом можно решить задачу подбора технического решения под каждый этап - для этапа 1 работа разработчиков сводится к заданию требуемой конфигурации в Jenkins, который средствами GitLab реализует ее в кластере Kubernetes; логика для этапов 2 и 3 полностью реализована в GitLab и является жестко заданной.

Поскольку логика CI/CD будет храниться в GitLab, нужно связать эту логику с логикой веток в Git, чтобы разработчики могли правильно взаимодействовать с логикой и использовать ее для нужного этапа. Для реализации сборки и доставки компонентов на каждый из этапов можно реализовать следующую схему на основе работы с Git:

1. Основная ветка - master. В этой ветке хранится состояние кода со всеми принятыми изменениями и отображает текущее стабильное состояние разработки компонента. Из этой ветки прямой доставки на какой-либо этап нет (рис. 1).

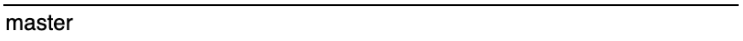

Рис. 1. – Графическое отображение жизненного цикла ветки master

2. feature/* ветки - ветки, создаваемые из ветки master для тестирования изменений. Жизненный цикл таких веток обычно довольно короткий и предполагает слияние обратно с веткой мастер в случае успешного тестирования изменений. Именно эти ветки нужно использовать при работе на этапе 1 с кластером dev уровня. Разработчик может выбирать ветки компонентов, которые желает протестировать в Jenkins и из таких веток происходит сборка и доставка на тестовый кластер этапа 1. Поскольку веток типа feature/* в рамках одного компонента может быть много, использование раздельных пространств в кластере позволяет тестировать изменения независимо друг от друга. Сборка и доставка в данном случае

инициируется из Jenkins, но выполняется пайплайнами GitLab (рис. 2).

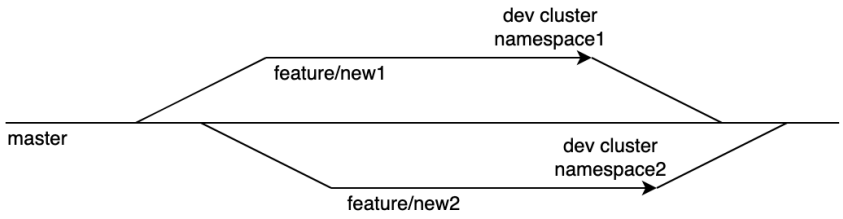

Рис. 2 – Графическое отображение жизненного цикла веток feature/*

3. staging ветка - состояние кода, представляющего собой кандидат для релиза, выпуска продукта заказчику. Она создается из ветки master и содержит только протестированные и принятые изменения. Сборка и доставка осуществляется на этапе 2 в кластер stage путем выполнения коммита в данную ветку (рис. 3). Таким образом в процессе участвует только пайплайн GitLab, Jenkins не используется.

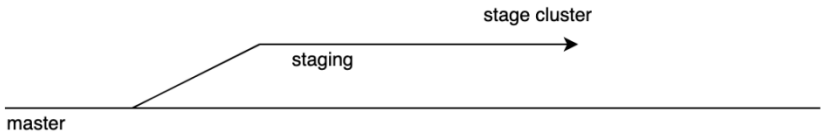

Рис. 3 – Графическое отображение жизненного цикла ветки staging

4. release/* ветки - ветки, содержащие состояние компонента для сборки и поставки заказчику. Таких веток обычно много, по одной на каждую версию продукта, чтобы иметь возможность доставки нескольких разных версий одновременно, иметь возможность откатиться на предыдущее состояние и тп. Релизные ветки создаются от ветки staging, поскольку именно в ней находится протестированный кандидат. Сборка и доставка осуществляется на этапе 3 в кластер prod путем выполнения коммита в данные ветки (рис. 4). В процессе участвует только пайплайн GitLab, Jenkins не используется.

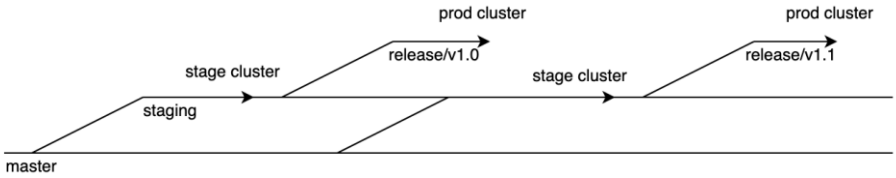

Рис. 4 – Графическое отображение жизненного цикла веток release/*

Данную схему необходимо предусмотреть не только для компонентов проекта, но и для инфраструктурных компонентов, таких как базы данных, сервисы очередей и так далее, в случае, если эти компоненты также предполагается доставлять внутри Kubernetes. Поскольку в данном случае обычно используются готовые решения, предоставленные сторонними разработчиками, то модель доставки на этапы для них немного упрощается, так как формально для них отсутствует процесс разработки. Однако может меняться их конфигурация, а это значит, что она также должна быть протестирована на этапах 1 и 2. Поэтому, хотя представленный выше процесс выглядит несколько переусложненным для инфраструктурных компонентов, на самом деле он стабилизирует процесс добавления каких-либо изменений в их конфигурации, чем существенно повышает общую стабильность продукта, так как сервисы типа баз данных или сервисов очередей как правило используются многими компонентами продукта и являются едиными точками отказа.

К созданным кластерам в Kebernetes нужно привязать систему мониторинга и сбора логов (например, широко известные Prometheus [9] и ELK-стек [10]), в этом случае у всех членов проекта появится средство наблюдения за процессом разработки в реальном времени. Появится возможность на уровне метрик оценивать те или иные изменения, вносимые на этапах тестирования, а так же появится возможность оценить общую производительность продукта в целом.

Таким образом, была выстроена модель разработки программного обеспечения с использованием современных технических средств, таких как Kubernetes, GitLab, Jenkins, Git с учетом необходимости организации быстрого и надежного процесса разработки и снижения возможных рисков при участии большого количества сотрудников. Данная модель общеприменима, масштабируема и дополняема, подразумевает тесную интеграцию с подходами DevOps и CI/CD, а также реализует подход IaaC, что существенно облегчит дальнейшую поддержку и развитие этой модели. Минусом данной модели можно назвать присутствие сразу двух средств для организации CI/CD процесса (GitLab и Jenkins), однако модель подразумевает написание логики только средствами GitLab, тогда как Jenkins выполняет только роль web-интерфейса и внутри логики не содержит. В этом случае обслуживание Jenkins минимально и сводится лишь к дополнению/обновлению web-интерфейса, выполнению резервного копирования и выделения ресурсов для его работы.

Литература

1. Что такое DevOps? aws.amazon.com URL: aws.amazon.com/ru/devops/what-is-devops/ (дата обращения: 12.10.2020)

2. Методология разработки CI/CD. URL: itglobal.com/ru-ru/company/blog/development-method-ci-cd/ (дата обращения: 12.10.2020)

3. Infrastructure as Code. URL: ibm.com/cloud/learn/infrastructure-as-code (дата обращения: 12.10.2020)

4. Pattern: Microservice Architecture. URL: microservices.io/Patterns/microservices.html (дата обращения: 12.10.2020)

5. Документация по Kubernetes. URL: kubernetes.io/ru/docs/home/ (дата обращения: 12.10.2020)

6. Jenkins User Documentation. URL: jenkins.io/doc/ (дата обращения: 12.10.2020)

7. User Docs. URL: docs.gitlab.com/ee/user/index.html (дата обращения: 12.10.2020)

8. Documentation. URL: git-scm.com/doc (дата обращения: 12.10.2020)

9. What is Prometheus? URL: prometheus.io/docs/introduction/overview/ (дата обращения: 12.10.2020)

10. What is the ELK Stack? URL: elastic.co/what-is/elk-stack (дата обращения: 12.10.2020)

References

1. Chto takoe DevOps? [What is DevOps?]. URL: ashhs.amazon.com/ru/devops/shhhat-is-devops/ (data obrashhenija: 12.10.2020)

2. Metodologija razrabotki CI/CD [CI/CD development methodology]. URL: itglobal.com/ru-ru/company/blog/development-method-ci-cd/ (data obrashhenija: 12.10.2020)

3. Infrastructure as Code. URL: ibm.com/cloud/learn/infrastructure-as-code (data obrashhenija: 12.10.2020)

4. Pattern: Microservice Architecture. URL: microservices.io/patterns/microservices.html (data obrashhenija: 12.10.2020)

5. Dokumentacija po Kubernetes [Kubernetes documentation]. URL: kubernetes.io/ru/docs/home/ (data obrashhenija: 12.10.2020)

6. Jenkins User Documentation. URL: jenkins.io/doc/ (data obrashhenija: 12.10.2020)

7. User Docs. URL: docs.gitlab.com/ee/user/index.html (data obrashhenija: 12.10.2020)

8. Documentation. URL: git-scm.com/doc (data obrashhenija: 12.10.2020)

9. What is Prometheus? URL: prometheus.io/docs/introduction/overview/ (data obrashhenija: 12.10.2020)

10. What is the ELK Stack? URL: elastic.co/what-is/elk-stack (data obrashhenija: 12.10.2020)