An Architectural Approach to Increase Adoption of the MDBCI Tool

Andrey Vasilyev, Maksim Kosterin
P.G. Demidov Yarsolavl State University
Yaroslavl, Russia
andrey.vasilyev, maksim.kosterin@fruct.org

Abstract—**This paper describes improvement of MariaDB continuous integration infrastructure (MDBCI) and the issues that were encountered on a way towards making it an easy to pick and use tool for everyday use by the end users. The ease of distribution and installation is achieved by providing MDBCI in the form of single executable file with minimum external dependencies. Contribution towards everyday use by developers is done by reducing the total execution time via introduction of multi-threaded approach.**

I. Introduction

MariaDB continuous integration infrastructure (MDBCI) is a set of tools for execution of end-to-end tests with complex database servers configurations including multiple database servers, proxy servers, high availability tools [1]. The core components provide a convenient way to automate creation and configuration of virtual machines (VMs). The main goal of implementing such automation process is to be able to create a fully configured and ready-to-use fleet of VMs with just a few commands. The whole feature set of managing VMs is split across several steps that are available to the end user as separate commands that allows:

- creation of VMs based on a template;
- automatic and reliable deployment of MariaDB, MaxScale and other applications to the created VMs;
- creation and management of VM state snapshots;
- reliable destruction of created VMs.

Besides that, MDBCI automatically generates a network configuration description file for created VMs that can be used to access the created VMs. MDBCI also maintains the list of binary repositories for each version of each supported applications for all target platforms. The MDBCI is developed as the open source project and available from the GitHub repository.

MDBCI was initially distributed as the source code that the end user must put into the destination folder and then install a set of dependencies that are required for the tool to run. The main problems with such a setup were different dependencies for different Linux distributions, dependency upon Ruby and Python language interpreters and the problems with their libraries. This lead to the low adoption of the tool even between the end users. In order to mitigate this issue we decided to provide end users with the accessible installation option.

Currently MDBCI is mainly used to enable operation of continuous integration tasks of the MariaDB MaxScale product. It is mainly used to build MaxScale packages for a variety of different Linux distributions and perform the system testing. Typical test configuration includes ten virtual machines running at the same time with eight of them having installed the two sets of database servers with different replication configurations and last two with the MaxScale itself. The configuration of the whole VM fleet takes about half an hour, however the whole system test suite takes up from four to six hours to complete.

The time of VM fleet set up for the whole test suite can be seen as negligible, but it becomes quite long for a developer to perform a subset of tests that are connected to ones recent changes. VMs can also be used during the actual development of a new feature, making setting up a set of VMs a common operation. Decreasing the overall time it takes for MDBCI to setup a VM fleet would allow developers to more frequently use the tool.

The paper is structured as follows. In Section II we describe the initial set of dependencies that MDBCI tool had and the approach we took to decrease their number and variety. Section III states the problem of packaging application for use in Linux distributions, describes the tooling that was developed to package the MDBCI and shows some adaptations needed for the tool to work. In Section IV we describe the approach we took to enable parallel configuration of VMs in a fleet and present test results. Conclusion summarizes the paper.

II. Reducing the number of MDBCI dependencies

A. Overview of MDBCI dependencies

MDBCI heavily relies on a wide set of various tools to manage VMs. The overview of essential elements used by the MDBCI is shown in Fig 1. VM are instantiated using Vagrant that utilizes Libvirt [2] and Amazon Web Services (AWS) [3] to take care of virtualization. The interaction between Vagrant and its providers, Libvirt and AWS, are handled by `vagrant-libvirt` and `vagrant-aws` [4] plugins respectively. `vagrant-aws` plugin allows Vagrant to control every aspect of an AWS virtual machine, which includes creation, destruction and snapshot management, while Libvirt machines require virsh command line tool to take end restore snapshots and perform reliable destruction.

The core dependency for MDBCI is the Ruby interpreter as the tool is written using the Ruby programming language. With that comes a list of Ruby gems that are needed to be installed

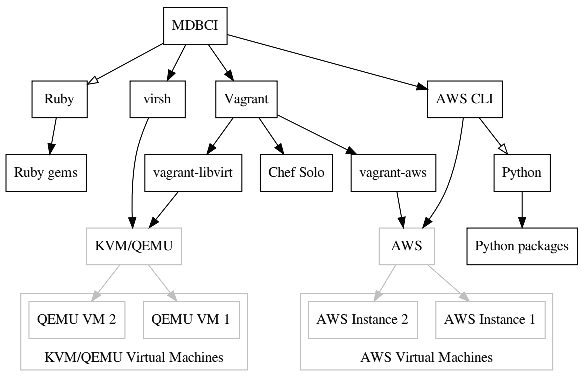

Fig. 1. Dependencies that MDBCI uses manage VMs

for the interpreter. Providing necessary version of Ruby on a target PC was one of the first major obstacles encountered on a way to an easy out-of-the-box usage of MDBCI. Installing Ruby and its gems on the target computer is not always a straightforward task considering that many of required gems have their own dependencies in the form of C or C++ libraries. Furthermore the version of Ruby required by MDBCI may conflict with the already installed one thus forcing user to switch between different versions just to be able to execute MDBCI.

As was previously mentioned, Vagrant does take care of most tasks related to VM management including customizing the configuration of state of VM by utilizing the built-in Chef solo client [5]. However it requires certain plugins to be installed to directly interact with virtualization software. For MDBCI needs, those plugins are `vagrant-libvirt` and `vagrant-aws`. To utilize Libvirt virtual machines target PC must provide the following tools:

- Libvirt — a toolkit to interact with the virtualization capabilities.
- Libvirt development library — a set of development files for the libvirt library.
- Libvirt management daemon — a server side daemon required to manage the virtualization capabilities.
- QEMU/KVM [6] — a hypervisor for hardware-assisted virtualization.
- Virsh — a command line tool to manage and control virtual machines and hypervisors.

Amazon Web Services (AWS) virtual machines have some special requirements too: AWS CLI — another command line interface that is needed to control AWS machines. This tool also has its own requirements. Even worse fact is that AWS CLI is a Python application and requires Python interpreter to be installed. It is already quite long list of things to be installed: 2 language interpreters, multiple command line tools, virtualization tools.

Besides that, some additional packages that are essential for compilation of main tools also must be installed. This, combined with everything previously stated, leads to a significant list of dependencies that every MDBCI user must provide manually. To solve this problem steps to reduce the number of dependencies for MDBCI have been taken.

B. Selection of dependencies for the removal

The first way to reduce the number of installed dependencies on the machine was to reduce the number of sources that these dependencies originate. By consolidating the package sources we can reduce the number of possible errors and provide better support for the remaining ones.

In the initial setup dependencies came from four different sources: the specific Linux distribution packages, the Vagrant distributions and it's plugins, Ruby gems and Python packages. The use of the language-specific packages over the conventional distribution packages was dictated by the absence of some required libraries in target distributions. Another reason was the library version variation across different Linux distributions.

The MDBCI is written in Ruby and corresponding interpreter is the vital requirement for the tool execution. The list of libraries is fixed across different distributions and managed by the Bundler tool [7]. Therefore this source of dependencies can not be restricted and on the contrary it can be used to substitute packages from other sources.

The Vagrant tool and it's plugins provide a lot of services and they are the foundation of all MDBCI features and can therefore can not be removed. The MDBCI interacts with the Vagrant by generating the configuration file and executing direct commands to it. The plugins are necessary to extend Vagrant with the support for Libvirt and AWS virtual machines. This set of dependencies can not be removed due to them providing a core functionality needed by the MDBCI.

The only possible targets for removal are the virsh tool and the AWS CLI that requires Python interpreter with custom installed packages. Both selected tools provide stable CLI and can be reliably used across different distributions. Both these tools can be manually used by the end users to interact with the corresponding subsystems to check the operation and status of corresponding systems.

To use the virsh tool it is only required to install one additional package along with other Libvirt packages. The AWS CLI on the contrary requires the additional interpreter and a set of Python packages to be installed. Therefore the benefit for removing the first dependency is almost absent, but it is high enough for the second one.

We have replaced this tool with the `aws-sdk-ec2` Ruby gem that allowed us to execute required commands to the AWS from the MDBCI process itself. The new set of dependencies is shown in Fig. 2. The modified components are shown with dashed borders. This modification also allowed us to consolidate the AWS configuration in the single configuration file. The performed migration allowed to reduce the number of required language interpreters for MDBCI to successfully function.

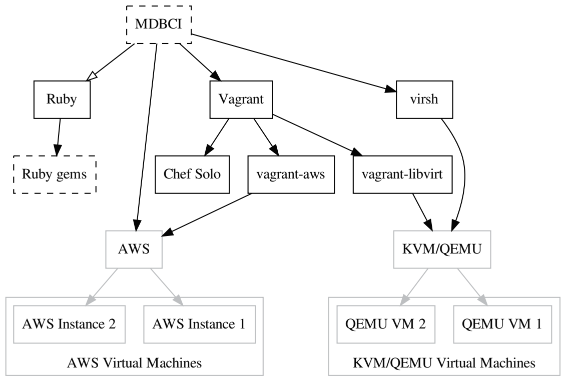

Fig. 2. The reduced set of the MDBCI dependencies

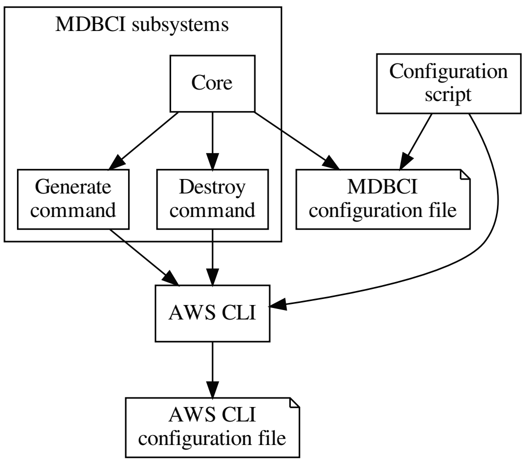

Fig. 3. The subsystems of MDBCI dependent on AWS CLI

C. *Implementation of AWS client in MDBCI*

In the earlier implementations MDBCI utilised AWS CLI to control AWS VMs. Even though the tool is a convenient way to manage AWS machines by itself, it is rather complex for automation tools like MDBCI.

The main reason for that is interaction between MDBCI and AWS CLI gose through command line. It means that all requests and responses are in the text form. While forming a command for AWS CLI is a simple task, but interpretations of its response requires text parsing. Console commands output is not guaranteed to be stable. Changes leads to MDBCI logic breaking. It makes AWS CLI usage unreliable.

Another issue is AWS CLI have to be configured outside of MDBCI prior to its usage, which requires a separate configuration file being filled by the end user. Structure the MDCBI-AWS interaction at this point can be seen on Fig. 3.

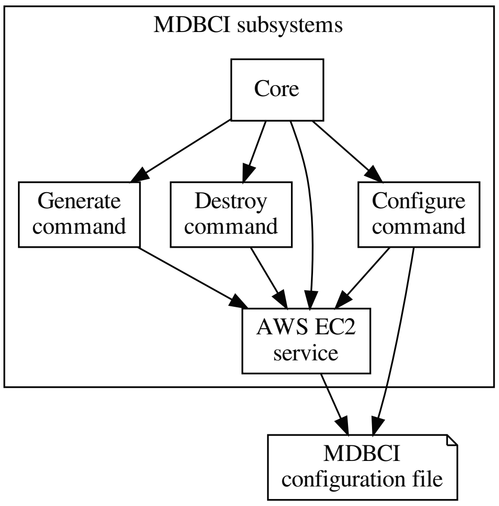

Fig. 4. The subsystems of MDBCI using the Ruby gem

Ruby gem `aws-sdk-ec2`, a replacement of the AWS CLI tool, provides the class `Aws::EC2::Client`. It has the same capabilities as its precursor, but provides interface in a form suitable to be used inside the application. Client class is wrapped into another class that provides convenient methods, the AWS Client. Main operations that are needed by MDBCI are delegated to the wrapper's methods, including key pair generation, destruction and VM instance termination.

AWS Client first needs to be configured with proper AWS credentials. To remove the need to manually create configuration file for AWS API, `configure` command was introduced. It requests AWS credentials from the user and then stores them together with other MDBCI configuration files. Another achievement was that this subsystem became optional and user may opt out to use only Libvirt as the VM provider. Final architecture can be seen on Fig. 4.

The removal of external dependency removed a whole set of problems including the issues of setting up the external tool, it's environment and the issue of configuring both MDBCI and AWS CLI separately.

III. PROVIDING MDBCI AS A SINGLE PACKAGE

A. *The issue of installing dependencies*

Even after reducing the number of required dependencies the overall installation procedure remained quite complex. The user would have to install the packages from their Linux distribution package directory, Vagrant distribution and the Ruby gems. Also the MDBCI itself was distributed from the source code meaning that the user must also install and maintain the correct source code base of the tool.

The use of external repositories for Vagrant and Ruby gems was due to difficulty of supporting different versions of dependencies across target distributions. The core issue from the MDBCI development point of view is that different versions of tools contain diverse set of issues that are needed to be somehow bypassed.

Additional issues arise when installing Ruby gems into the base operating system. In the naive way they are installed system-wide for the installed interpreter. This approach in some cases may break the execution of other tools written in Ruby or the distribution upgrade may break the set of gems that MDBCI relies upon.

The core idea behind packaging is to put everything into a single package that can be easily distributed and easily used by the end users and on the continuous integration servers.

B. Comparison of alternative packaging strategies

There are several strategies that can be used to package and deliver the application for the use in Linux distributions. The basic and classical one is to provide packages for each target distribution. Another is to use distribution-agnostic tools including AppImage, Flatpak or Snappy [8]. The least one is to use containerization technology like Docker [9].

The use of classic approach of using distribution-oriented packaging tools requires the developer to setup the repositories for each target distribution and each release of it. When the new version of the tool is released, it must be build for every supported repository. This task may be automated, but the initial investment is quite high.

In order to decrease the burden of supporting several repositories for the end-user applications the distributions-agnostic tools were developed. Along with ability to provide applications to several distributions the upside for these tools is support for running several versions of the application. This feature is useful for the end-user when the tool have regressions that take time to be resolved.

The AppImage approach to the application distribution is to create the non-modifiable image that contains the target application and all it's dependencies. When executed, the image is mounted in read-only mode to the filesystem using the filesystem in the user space (FUSE) subsystem. Then the target application is executed in the modified environment, so the bundled libraries and executable would precede those installed on the base operating system. In order to use the AppImage the end user should download the software, set executable bit for it and run.

The Flatpak and Snappy require user to install an alternative package management software on top of the base system. For the fully-supported distributions the installation is streamlined, but for the most it would require user and developers to perform similar steps as using the conventional repositories, sometimes installing the new management tool along. These tools are designed to protect the end user from the malicious applications, therefore restricting the interaction of the application with the base operating system. Being useful by itself it acts as the entry barrier that requires more investment.

The Docker containerization platform is the de facto standard for execution of server-side applications, but not limited to running such applications. It allows developers to provide the full-featured application without the need to install it's dependencies on the host machine. For the case of running MDBCI the Docker containers are too restricted: by default they do not allow to access the resources of the host machine and they do not allow to store state inside the container. These restrictions require deep modification to the way how MDBCI uses resources of the host machine and MDBCI usage scenarios.

By analyzing the possible solutions we have decided not to provide conventional packages as it would require to build the complex infrastructure. The use of Docker containers would require to heavily modify the existing CI processes and revamp the MDBCI tool itself. By comparing the distribution-agnostic tools the AppImage approach was chosen as it has the lowest requirements to the base operating system and imposed restrictions for the application being shipped and executed.

C. Packaging MDBCI into the AppImage

The AppImage project provides a tool called pkg2appimage [10] that automates creation of AppImage packages from the traditional packages. Another convenient tool is the linuxdeployqt [11], it allows to create such packages for the application compiled with the QMake or CMake build systems. The MDBCI tool does not fall for any of these categories, therefore the packaging process have been implemented on the low level.

The low level AppImage packaging process includes following steps:

1) Create the directory that will form the resulting image, we will call this target directory.
2) Compile the application, so it will use relative paths to the required resources and therefore could be relocated in the file system. The compiled application is installed into the target directory.
3) Copy all the necessary libraries that the application depends on into the target directory.
4) Put all other dependencies into the target directory.
5) Put the description of the application in form of the icon and the desktop file.
6) Provide the launcher for the AppImage package.
7) Pack the contents of the target directory into the AppImage package.

The proposed process is quite complex and requires developer to understand each step in order to achieve the desired outcome. At the time of the development there was only one publicly available tooling to bundle the ruby into the AppImage package [12]. The main downside for this approach was the inability to correctly execute installed Ruby gems. Another problem was the inability to use the proposed solution as it did not provide any extension points requiring us to copy it and modify it to our needs.

D. The tooling to create the AppImages based on Ruby interpreter

Having limitations of existing tools we have developed the tooling that allows us to create the MDBCI package and that could be re-used by other applications that are based on Ruby

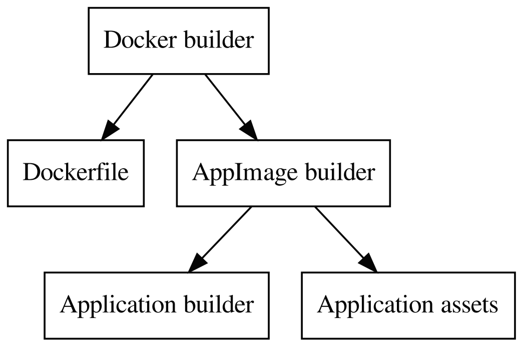

Fig. 5. The components of AppImage creation tooling

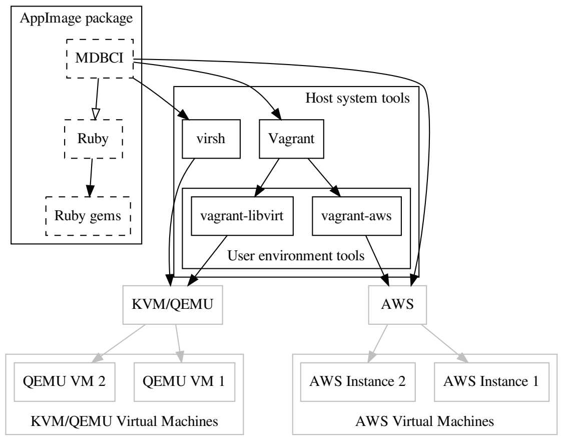

Fig. 6. The final structure of MDBCI dependencies after split into the AppImage package and the host packages

interpreter. The principle structure of the tooling is shown in Fig. 5.

The tooling provides two scripts: the Docker builder and the AppImage builder. The purpose of the first one is to launch the second one in the controlled environment. This environment allows to separate the build machine from affecting the resulting build process and create the binary executable files that depend on the oldest possible implementation of the standard libraries. It allows to execute the created binaries on the variety of available distributions.

The Docker environment is build on top Ubuntu 14.04 distribution. It includes all the standard libraries that are required to compile the resent releases of Ruby interpreter and commonly used Ruby gems. The environment also uses the GCC 8 compiler instead of provided GCC 4.8. It allows to have support for the older base library releases and the newer optimizations available in recent release of the compiler.

The AppImage builder performs all the necessary steps to create the AppImage that were described in the previous section. On top of that it compiles the Ruby interpreter and installs it to the target directory. If no application is specified to be built, the resulting package will only include the interpreter.

If the application was specified as the parameter for the AppImage builder, then the corresponding builder script will be called. The purpose of this script is to install all dependencies that are required for the application and the application code itself. The AppImage builder will also use the assets provided by the application developer. They consist of the desktop file and the icon file.

If another Ruby developer would like to use the developed tooling one must provide only the application building script and the corresponding assets. The use of Docker build environment is not mandatory, but it definitely increase the stability of the result.

E. Installation of external dependencies

During the packaging we came with issues of providing all the required tools inside the AppImage package. The overview of the final dependencies structure is shown in Fig. 6. Due to this issue we needed a way to correctly setup the host environment in which the MDBCI will run. Therefore we introduced the `setup-dependencies` MDBCI command that is responsible for the dependencies installation. The point of this command is to get rid of a need to manually install dependencies and configure environment from the end user.

Now when Ruby and all of its gems are packed into the AppImage package, the only dependencies that are left to be installed are the virtualization tools. However they may differ on each Linux distribution. The installation procedure may also differ based on the package managers and repositories available for a specific distribution. So the idea behind `setup-dependencies` command is to keep the whole process unified but break down steps that relies on package manager into two groups based on Linux distribution: for RPM-based and DEB-based distributions.

General setup process consist of 5 steps:

1) Dependencies installation using built-in package manager.
2) Addition of current user to the Libvirt group.
3) Installation of Vagrant plugins.
4) Creation of domain for Libvirt virtual machines.
5) Export of environmental variables.

Dependencies installation step is done differently among supported Linux distributions. Currently there are three different groups of installation instructions: for CentOS and RHEL, for Debian, for Ubuntu. The only difference between Debian and Ubuntu installation is the list of packages that needs to be installed. Other steps are pretty much the same. RPM installation is a bit different in that it uses different package manager, but the process is still the same. First, packages that are available through the package manager are installed, then Vagrant is updated to the target version. It’s important to note that after installing Libvirt daemon Libvirt service must be started manually in order to use Libvirt.

After packages have been successfully installed, current user must be added to the relevant Libvirt groups to be able to utilize Libvirt VMs. In our case user is added to every available Libvirt group just to be sure. There is nothing more in this step as it quite simple.

`Vagrant-libvirt` and `vagrant-aws` are the only plugins that are installed on the machine during the third step. There is nothing special about it except for one thing: on the newest version of RPM-based distributions that use Gnu C Compiler (GCC) version 8 and above (like CentOS 8, Fedora 29) there was a bug during `vagrant-libvirt` installation where Bundler (tool that Vagrant uses to install its plugins) was unable to locate Libvirt development library. So we had to run this step with explicit defining of the path to the Libvirt library if it failed at the regular installation.

After the installation is completed, a domain is created for the Libvirt virtual machines. Since Vagrant only supports virtual machines that are located on a system bus, domain itself must be created on a system level. However this leads to normal user being unable to view machines created with Vagrant without the root privilege. To solve this issue environmental variable `LIBVIRT_DEFAULT_URI` is imported to the command line interpreter configuration file (`.bashrc`) which enables system as a default source of virtual machines. This concludes the setup process. After that user is required to restart the PC in order to apply changes.

`setup-dependencies` command also has two options. The first one is the `--reinstall` options that tells MDBCI to delete previously installed dependencies before performing installation. The second one is called `--force-distro` which causes `setup-dependencies` to use installation process for one of the four supported distributions. This can be useful on distributions that are not supported by the command but share the same package manager and same packages sets as the supported ones.

We were able to successfully support the several RPM-based and DEB-based distributions including CentOS 7, 8; RHEL 7; Fedora 27, 28, 29; Ubuntu 16.04, 18.04; Debian Jessie, Stretch; Mint 18, 19.

F. Execution of applications outside AppImage package

Packaging application into AppImage package comes with a problem. When executed, the application uses the internal environment of the package, not only forcing internal libraries of the package over the system ones, but preventing access to the external environment. This breaks the execution of most external applications.

The solution to this problem is to save the environment of the OS where the AppImage package was built and use it on a target PC. During the start of the application, old environmental variables are saved by the launcher with the special prefix which helps us to recognize them later on. Some of the old variables are saved with their original names but modified values.

During the call for the external application, old environmental variables are extracted and passed to the pipe process with their prefix removed. This way we are able to use original environmental variables without modifying internal AppImage package environment.

G. Results of application packaging efforts

Removal of the external AWS CLI tool allowed us to completely get rid of the Python interpreter requirement which further lead to ability to package the whole MDBCI application into a single executable file. Packaging MDBCI into the AppImage serves as a great time saver in the sense that its removes the need for the end-user to install and manage Ruby versions. This also creates the added benefit of the easier MDBCI distribution.

Addition of the `setup-dependencies` command resulted in the significant reduction in the number of steps that needs to be performed by the user to properly set up MDBCI. What previously took on average 7 to 10 commands depending on the distribution, now can be done with a single build-in command. Overall installation process of MBDCI is much easier now, which should potentially lead to a better adaptation by the end-users.

The examined approach provides benefits for both the developers of the tooling and the target audience. The developers are guaranteed to have a stable execution environment that can be easily enriched with the new libraries to perform specific tasks, the very base - the version of the interpreter - also becomes stable. The end users are no longer need to install a development environment for the application, they can just get the executable and proceed with their task.

IV. Configuring virtual machine fleet in parallel

A. The issues of using Vagrant build-in chef client

As described in the previous paper we are using the Vagrant to setup machines on different platforms and Chef to configure instances to the required role [13]. The Vagrant itself supports the Chef as a provisioning tool. We came across the issues with such provisioning in case of running several machines at the same time:

- The machines were not brought up correctly with the Vagrant essentially breaking the configuration.
- The implementation of the Vagrant client for the Chef have several issues that may especially arise with the supporting the latest distributive releases.
- The output of the Vagrant is hard to debug, i.e. all information is dumped into the standard output out of all Chef clients running at the same time. It is almost impossible to debug the issues of configuring the machines.

The first issue was mitigated by doing several attempts of VM configuration. The idea and implementation details can be found in our previous paper.

The second and the third issues have lead us to create the internal component that allows to configure the created VMs. After introduction of this tool the interaction between the Vagrant and the MDBCI has changed. Now the Vagrant is only responsible for starting VMs and providing information on how to connect to them. The created component then uses this data to connect to running VM and perform configuration.

B. Implementing the parallel configuring of Virtual Machines

Typical test setup consists of two sets of MariaDB servers with two different replication configuration: one classical MariaDB/MySQL Master/Slave replication and another — synchronous replication based on Galera library. Finally, there are machines running MaxScale itself. Everything previously mentioned forms configuration with 10 separate machines that needs to be brought up and properly configured. This process done on a single thread can take a long time slowing down the actual testing. One way to solve this issue is to spread configuration process among multiple CPU threads.

Originally, when everything was done on a single thread, the process looked like this:

1) Old machines were destroyed.
2) Whole configuration was brought up with a single `vagrant up` which utilized the Vagrant innate parallelism.
3) Individual machines were configured one by one.
4) Additional attempts are made to configure machines that weren't configured on the previous step.
5) Machines that aren't still configured are destroyed and created again to be configured from the scratch.

After transition to the parallel configuration order of steps remained the same, but now every step is individual to each machine. That means that when we select machines for configuration, each machine is brought up, configured and possibly fixed separately on its own thread. We doesn't rely on Vagrant bringing them up in parallel anymore. Final order of operation for bringing up single VM can be seen on Fig. 7.

The important task to keep track of when transitioning to the parallel configuration is to preserve MDBCI output clean and organized. On a single thread journal is printed successively for each step. But with multiple configurations running at once output stream turns into a one giant mess. The best solution to this problem that we come up with was to store journal independently for each machine without printing it to the output stream. To keep user in tact with the general progress notification about execution state are still shown at the beginning and the end of each step. However this doesn't clutter output stream like the whole log would. When configuration is done the journal printed to the output stream in chunks, where each chunk is the execution journal for a single machine.

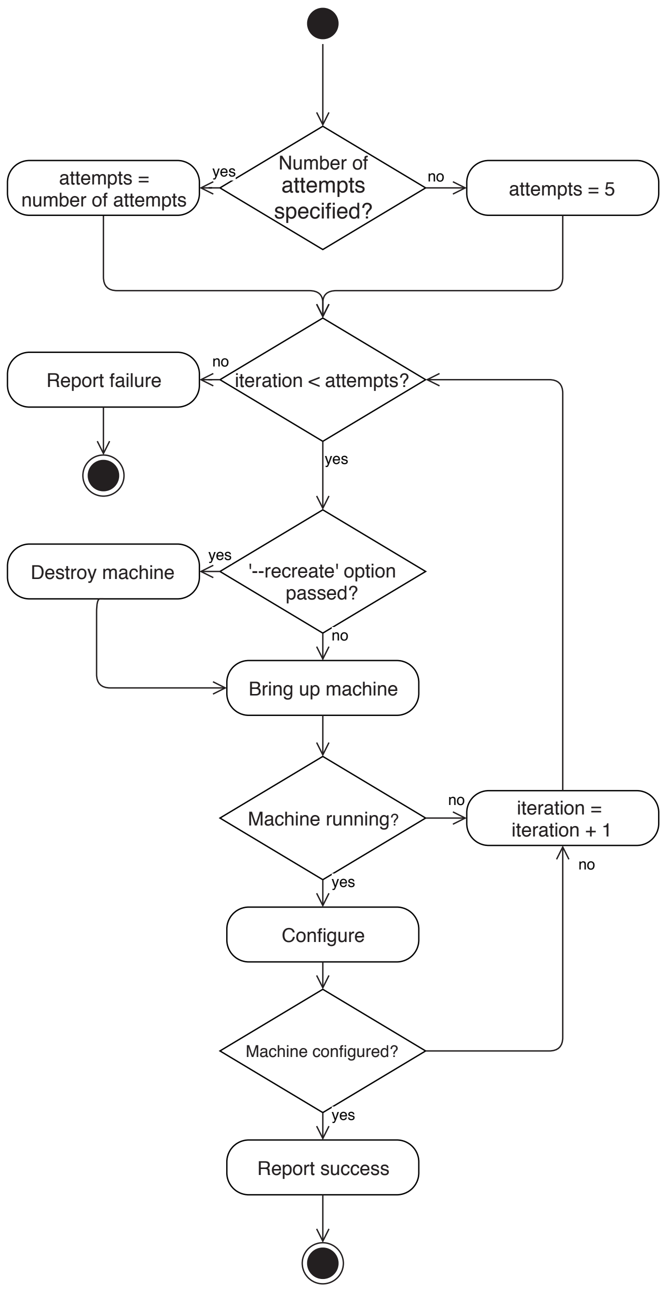

Fig. 7. Virtual machine configuration order of operations

C. Controlling the level of parallelism

To further customize level of parallelism the new option `--threads` was introduced for `up` command. It allows to set maximum number of threads used by the `mdbci up` command. The time it takes to bring up a configuration will depend on the number of threads used.

We were able to measure amount time required to bring up configuration consisting of 10 virtual machines: 4 machines running MariaDB server, 4 Galera nodes, 2 MaxScale nodes. Measurement was done on a server with 3.7 GHz Intel Xeon E5 CPU with 8 cores and 64GB of RAM. 10 measurements were made for each number of threads, up to 8 available threads. Average and median time were calculated based on gathered data. Results are present of Fig. 8.

The most noticeable jump is from 1 thread to 2 threads. Median time with 2 threads is almost 2 times smaller than with 1 thread with average time being one and a half minutes behind median time. This jump is explained by a two-fold decrease in a number of machines handled a single thread. So with 2 threads each thread has to go through half as much cycles compared to a single thread setup. Although next VM in the queue is distributed as soon as the worker is released,

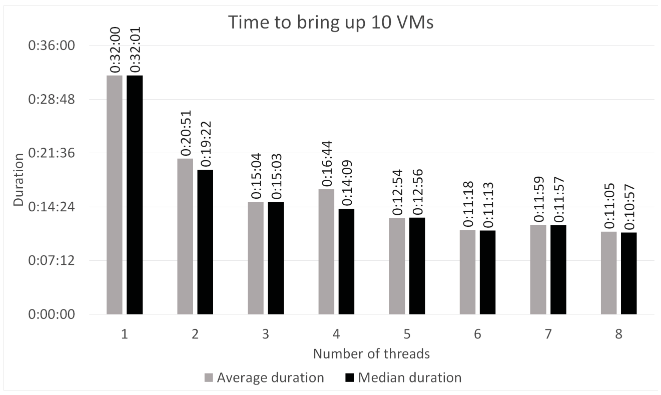

Fig. 8. Required time to bring up configurations of 10 virtual machines on the server

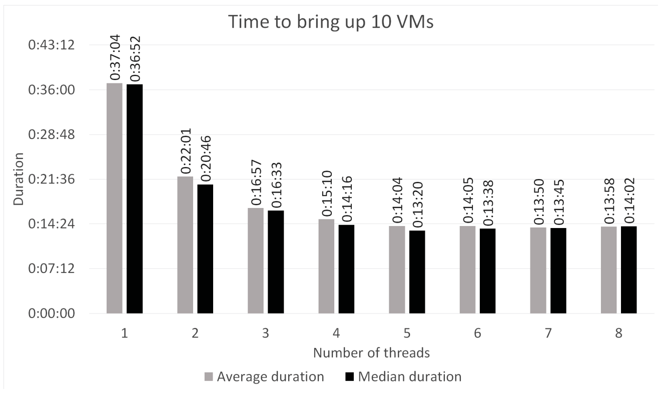

Fig. 9. Required time to bring up configurations of 10 virtual machines on the notebook

the time required to bring up a each individual machine is relatively comparable, so we can still call a batch of machine whose configuration is done at the same time a "cycle".

Time decrease for 3, 4 and 5 threads follows the same trend as spotted in the transition from 1 to 2 threads, where reduction in the amount of cycles leads to the lesser execution time. For 3, 4, 5 threads the amount of cycles needed is 4, 3 and 2 respectively. There is a noticeable spike in the average time with 4 threads, which is the result of 2 out of 10 runs resulting in a need for MDBCI to fix virtual machines. Median value still holds in line with the general trend.

Values for 6, 7, and 8 threads lies in approximately similar range, with the time for 7 threads being slightly higher due to a server load. It takes 2 cycles to finish bringing up machines with each of those number of threads, so the time shouldn't differ much.

Same measurement were made on a regular notebook with Intel Core i7-7500u CPU and SSD that can represent a PC used by a regular developer. Results are shown on Fig. 9 As we can see, values follows the same trend as shown on a previous diagram, but the average and median time is expectedly higher.

D. Performance analysis

The presented results clearly indicate that the use of several threads allows to decrease the overall configuration time by 60 percent on both configuration. The minor difference of the overall configuration time between the server and the notebook can be explained in that the resources of the server were not fully utilized during the configuration process.

The results also state that the use of more than 4 threads on both configurations provide little or no effect at all. For the notebook this seems to be a ceiling as it does not have only 2 physical processors working in the hyper-threading mode.

The inability to efficiently scale past 3 threads for the server can be seen in the input-output limitations. The process of the machine configuration requires vast amount of resources to be fetched from the Internet. They mostly come in the form of different distribution packages. In order to further speed-up the configuration process the caching proxy server can be employed.

Another possible bottleneck could be in the performance of the hard drive input output. For each machine the space is allocated on the hard drive and that space is then actively accessed.

In order to proceed further with the performance testing of the tool the measurement of the more parameters must be included in the process. They include the load of the CPU, memory, hard drives and the networking activity. This would lay the basis for finding and fixing the bottlenecks in the VM configuration process.

V. Conclusion

Development of the continuous integration support tools for any project may lead to an increased internal complexity and to use of various overlapping technologies. As we have seen in MDBCI this leads to the poor adoption of such tools and to increased cost of supporting existing sparse installation base. In order to mitigate this issue such tools should be developed as other end user application having elaborate plans on how the tool will be delivered to the end users, how it will be maintained up to date and how the found issues will be resolved.

For the MDBCI tool case it was found that modern distribution agnostic tools provide the most value due to the reasonable balance between the efforts required to maintain the package and the ability to seamlessly execute the application in different GNU/Linux distributions. Particularly the AppImage approach to packaging seems to be the best when dealing with tools that heavily interact with the applications and store state on the host system.

During the packaging effort we have developed the specialized tooling that can be used out of the box by Ruby application developers who are trying to create the portable package of their application. In order to use it one must only develop a placement strategy for the application, leaving the whole process of packaging to the tooling.

The effort of initiating the fleet configuration in parallel allowed us to speed up the whole time by a factor of three. This will not only allow for a better resource utilization and faster feedback from the continuous integration processed, but also for a deeper use of the tooling throughout developers.

The possible future directions for the improvement of the MDBCI tooling include the support for setting up parts of the fleet by label and it's deeper integration with MaxScale system test. This would allow test to have more reliable recover procedure and to have the targeted results faster as only required VMs will be initiated. Another research direction is to find and eliminate other bottle necks in VM configuration process.

VI. Acknowledgment

This project was done with financial support of MariaDB Corporation.

References

[1] S. Balandin, T. Turenko, A. Vasilyev, M. Kosterin, E. Vlasov, and R. Vlasov, *MariaDB continuous integration infrastructure (MDBCI)*, 2019 (accessed February 9, 2019). [Online]. Available: https://github.com/mariadb-corporation/mdbci

[2] H. D. Chirammal, P. Mukhedkar, and A. Vettathu, *Mastering KVM Virtualization*. Packt Publishing Ltd, 2016.

[3] M. Wittig and A. Wittig, *Amazon web services in action*. Manning, 2016.

[4] M. W. Navin Sabharwal, *Automation through Chef Opscode*. Apress, Berkeley, CA, 2014.

[5] J. Ewart, M. Marschall, and E. Waud, *Chef: Powerful Infrastructure Automation*. Packt Publishing Ltd, 2017.

[6] G. S. K. A. Binu, *Virtualization Techniques: A Methodical Review of XEN and KVM*. Springer Berlin Heidelberg, 2011.

[7] S. Wintermeyer, "Bundler and gems," in *Learn Rails 5.2*. Springer, 2018, pp. 243–255.

[8] S. Paulus, T. Smits, T. Becht, and S. Kol, "Ubiquitous learning applied to coding: A set of tools and services to deliver code-intensive learning contexts to student devices," in *Proceedings of the 3rd European Conference of Software Engineering Education*. ACM, 2018, pp. 87–92.

[9] I. Miell and A. H. Sayers, *Docker in practice*. Manning Publications Co., 2016.

[10] P. Simon, *pkg2appimage GitHub repository*, 2019 (accessed February 9, 2019). [Online]. Available: https://github.com/AppImage/pkg2appimage

[11] ——, *linuxdeployqt GitHub repository*, 2019 (accessed February 9, 2019). [Online]. Available: https://github.com/AppImage/pkg2appimage

[12] Y. Tokunaga, *ruby.appimage GitHub repository*, 2018 (accessed February 9, 2019). [Online]. Available: https://github.com/yuntan/ruby.appimage

[13] M. Zaslavskiy, A. Kaluzhniy, T. Berlenko, I. Kinyaev, K. Krinkin, and T. Turenko, "Full automated continuous integration and testing infrastructure for MaxScale and MariaDB," in *Proceedings of the 19th Conference of Open Innovations Association FRUCT*. FRUCT Oy, 2016, p. 36.