ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH2003

TOWARD CONTINUOUS DELIVERY OF A NONTRIVIAL DISTRIBUTED SOFTWARE SYSTEM *

S. Wai†, SARAO, Cape Town, South Africa

Abstract

The MeerKAT Control and Monitoring(CAM) solution is a mature software system that has undergone multiple phases of construction and expansion. It is a distributed system with a run-time environment of 15 logical nodes featuring dozens of interdependent, short-lived processes that interact with a number of long-running services. This presents a challenge for the development team to balance operational goals with the continued discovery and development of useful enhancements for its users (astronomers, telescope operators). Continuous Delivery is a set of practices designed to always keep software in a releasable state. It employs the discipline of release engineering to optimise the process of taking changes from source control to production. In this paper, we review the current path to production (build, test and release) of CAM, identify shortcomings and introduce approaches to support further incremental development of the system. By implementing patterns such as deployment pipelines* and immutable release candidates we hope to simplify the release process and demonstrate increased throughput of changes, quality and stability in the future.

INTRODUCTION

The Control and Monitoring Subsystem (CAM) for the MeerKAT radio telescope consists of many different software components that work in tandem to allow operation of the telescope as a single, cohesive instrument[1]. However, the large amount of moving parts and differentiation presents a challenge in terms of software engineering complexity[1]. So far, this has been mitigated by leveraging virtualisation with automated deployments[3], as well as continuous integration and automated testing[4]. This has worked well, allowing the software team to adopt an incremental development model for extending the system. Enhancements are developed and released, usually 2-3 months apart[2].

The "Last Mile"

Despite having a high level of automation, the process of releasing newly developed functionality has not been problem-free. Misconfiguration, exceptional states and errors still manifest during some deployments. These prompt long, arduous fault-finding and troubleshooting efforts for the System Team[3] during releases.

Fortunately, releases are scheduled on "maintenance" days during which science operations are ceased so that engineering teams can work on the relevant subsystems and on-site maintenance – such as replacing of physical parts – can take place. This gives a window of opportunity for deploying CAM to site, with a full deployment (and verification activities) taking 2-3 hours in the best case scenario, but can take up the full day if issues are encountered.

This is a common problem in software development: the presence of bottlenecks, often in the final stages of software development lifecycle (the "last mile")[4]. Continuous Delivery[5] offers an approach to software delivery that addresses these problems by focusing on engineering for *feedback*, early and frequently. Some informal analysis was undertaken with this in mind to identify some of the root causes of the problems. They are outlined below:

Gaps in Automated Testing In spite of the sophisticated integration testing in place by the Automated Qualified Framework(AQF)[4], it failed to catch some categories of errors. AQF tests are executed in a static, single-node environment that is only partially deployed on each run: a python script, `kat-update.py` would pull the latest `master` branch of all CAM Python packages on the AQF node. In this scenario, the full deployment procedure is not exercised, thus some problems relating to configuration changes outside of code changes to Python packages cannot be discovered in AQF. Additionally, since it was a long-lived static environment it was vulnerable to *configuration drift*[6].

Build Provenance A consequence of running AQF on a scheduled timer instead of being triggered by changes, is that the provenance of a test result depends on the state of `master` at the time it was run and completed. Once an AQF test execution succeeds, a special branch (`stable`) is updated to mirror the state of `master`, for all relevant CAM packages on the AQF node. Given that an AQF run can take up to 3-4 hours, there is risk that the state of `master` at the time of completion is not the same as when the tests began[5].

* Work supported by South African Radio Astronomy Observatory, National Research Foundation
† swai@ska.ac.za

[1] The system design itself has proven to be robust as it is extensible: CAM has been chosen as the reference architecture for the Square Kilometre Array's Telescope Manager[2].

[2] At the time of writing (September 2019), version 23 of CAM system was deployed to production.

[3] A subset of the Software Team responsible for deployment of the CAM subsystem and overall health of the datacentre servers running CAM.

[4] It is apparent that *continuous integration* alone is insufficient, as most of the problems occur when trying to release already integrated code to the users.

[5] Fortunately, the team is small enough that this is addressed through clear communication and coordination of "code freezes" at the appropriate times.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH2003

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

Non-repeatable Deployments The deployment of the system is orchestrated by a collection of Python Fabric scripts that are responsible for provisioning CAM nodes to a target group[3]. Provisioning means to enact the following operations in order:

1. Boot up the VMs on the target environment's Proxmox cluster that correspond to the nodes in the configuration database
2. Sequentially apply some actions, based on the roles assigned to the node
3. Install the appropriate Python packages from the requirements file (specified as an argument)

A significant problem the team recognised is that not all packages were pinned to any particular version. The versioning of CAM's own Python packages are signified by *release branch* rather than pinned to specific versions. This is problematic, since every deployment has a chance of pulling different transitive dependencies that may override the initial specification. A suboptimal workaround is in place that exploits the imperative execution style of the scripts: by strategically specifying the packages in an order that works. This was not ideal, however relying on *release branches* as the primary versioning strategy did allow the team to mitigate some of the issues, at the cost of additional complexity (package version can be updated directly in source control). Automated scripts, even when written imperatively, are immensely powerful first step to optimising deployments. However, because many sequences of actions are not idempotent when executed together, changes are difficult to manage and their effects tricky to predict[6]. This was a source of many problems during deployments, due to reliance on a large set of brittle scripts that are opaque in their intended outcome.

Late Integration The **katconfig** package contains the configurations for physical and logical devices in the telescope (receptors, subarrays), to be parsed and served by **katconfserver**[8]. It has been extended to also hold telescope *pointing model* data so that operators and astronomers may update them as needed using GitHub's Pull Request model[7]. Production changes are kept in a separate branch (`karoo`), that is pulled periodically on to site by a scheduled job. The `karoo` branch was needed in order to prevent changes during development propagating directly into the production environment without going through the verification process. Unfortunately there is no automated job in place to integrate these changes back into `master`, so it is currently a manual step that must happen during release time.

[6] A rigorous treatment of the problems with imperative actions in configuration management can be found in [7].

[7] The process is followed by development teams for code changes and works extremely well for change review and auditability.

SOLUTION APPROACH

To avoid the trap of endless local optimisations, we focused on improving the entire process end-to-end. First, we sought to develop some conceptual models derived from first principles. Also, as software evolves, the implementation details will differ based on context but the principles that govern them should not vary[5]. These conceptual models could then be appropriately reified based on judgement of the developer at the time of implementation.

Change Management

The activity of software development is fundamentally concerned with introducing change to some system by modifying some of its constituent parts. We don't know if the system's behaviour can be fully deterministic, but we can say its initial state S_0 is determined by some instance of its *configuration* C. We adopt Couch & Sun's definition of *configuration* as the "invariant part of a system that is not modified during normal operation or by normal users who are not also administrators"[9]. Illustrated in general functional notation:

$$S_0 = C$$

In more concrete terms, a system's configuration could be composed of *operating system*, *directory structure*, *system libraries*, *language packages*, *services* and so on.

$$C = f(\text{python packages, OS packages}, \cdots)$$

When defined as such, it becomes apparent that a significant task of *release engineering*[10] is the selection of appropriate *configuration* to constrain the system's initial state (and subsequently its behaviour) to desired bounds. What counts as "appropriate" must meet some qualification criteria through testing or more formal methods.

Releases can be characterised as the the process of replacing a set of configurations with another. The set of changes (ΔC) can be expressed as the composition of normal software development operations of building from source (b), packaging (p) and deployment (d). Thus a chain of determinism can be established back to the original source code that produces the change.

$$b = build(sourcecode, \cdots)$$
$$\Delta C = d \circ p \circ b$$

Path to Production

By treating the set of changes as a first class citizen, we can conceive an appropriate composition of operations for it to be succesfully included in the changeset. In practice this is known as the *path to production* for a change. It is often modelled as a sequence of stages that the change is promoted through until it reaches production.

The *path to production* directly corresponds to the construct of a *deployment pipeline*[11, 5]: the multi-stage, automated workflow that forms the foundation of Continuous Delivery and DevOps practices.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH2003

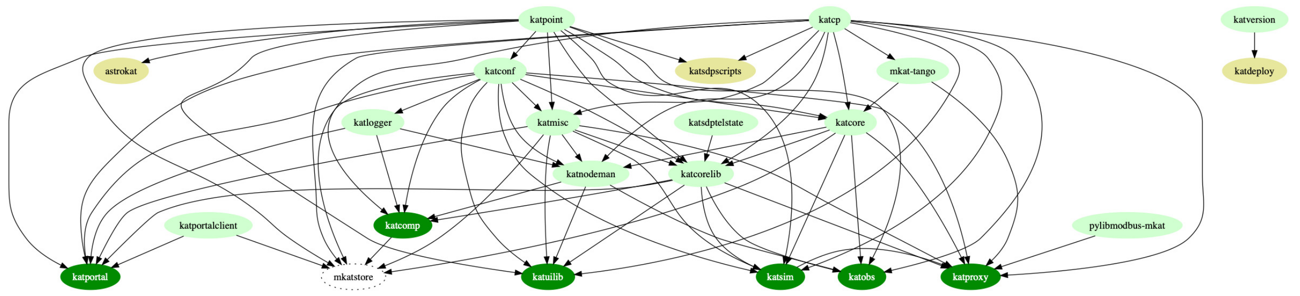

Figure 1: Python packages dependency tree for CAM generated by `pipdeptree`.

There are implicit dependency relationships between software packages in the CAM system, as illustrated by the dependency tree[8] in Fig. 1. In order to map the *path to production* for code contributions to CAM we needed to understand the full structure of CAM's dependencies. By understanding the dependency tree, we can design the continuous integration and deployment pipelines in a way that allows every change to propagate and be verified incrementally. That is, when a package is successfully built (i.e. its unit tests pass), other packages downstream to it should immediately also build, using the former's latest built version. In this way, we can achieve *provenance* while also discovering integration failures early.

Release Candidate

A fundamental idea in Continuous Delivery is that every set of changes that is succesfully integrated should be potentially releasable. In other words, it has passed some measure of quality that might make it fit for deployment and ultimately used in the production environment. Humble & Farley go as far as to state that "every change[...] is a release candidate"[5] and it is the starting assumption that the continuous integration system must set out to disprove[9]. For a complex, distributed system such as CAM, a *release candidate* is not so easily defined: the notion of a single artifact that encapsulates the changes to the *configuration* of a system did not exist. The *configuration* of CAM converges upon deployment to an environment and the procedure was not determinstic.

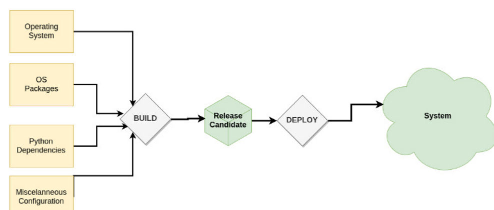

Figure 2: Release Candidate server image concept.

However by leveraging one of CAM's original design principles to favour homogenous nodes[8], we can encapsulate the *release candidate* concept in a server image (Proxmox LXC template). So every *build* would result in a server image that could be deployed to an environment corresponding to a deterministic snapshot of the entire system's baseline, or starting state (Fig. 2). This is an approach employed by Netflix[13] in their build and deployment system for a famously complex and robust production environment, comprising of thousands of nodes. A key benefit of this is that it allows the clear separation of *build* and *run* stages, so that we can gain traceability and provenance of changes. Multiple candidates can be deployed and verified simultaneously and by baking the packages into the server image, we can introduce some determinism into the build and deploy process without needing to deepdive into the set of opaque deployment scripts.

THE CAM DEPLOYMENT PIPELINE

Based on the package dependency tree (Fig. 1), we designed a deployment pipeline(Fig. 3) that is triggered when the six leaf nodes(*katcomp*, *katobs*, *katportal*, *katproxy*, *katsim*, *katuilib*) are successfully built. To avoid unnecessary implementation effort, only these six packages are selected. Their builds are also sufficient to demonstrate that they integrate with packages above them in the dependency tree.

The `ci.pin-dependencies` job is responsible for producing a canonical, list of requirements with their version pinned. `pip-compile`[14] is used to resolve version conflicts between dependencies. The requirements are provided as input to the `ci.build-candidate` job, which effectively provisions a full node and then bakes it into an LXC template image (`autobuild-*.tar.gz`). This image is uniquely named and corresponds with single build that was triggered by changes. The server image is then deployed into two environments by the `ci.aqf-pipeline` meta-job so that they can be verified for quality and correctness.

The final outcome is that we have encapsulated the concept of a *release candidate* into a single artifact with a very clear, deterministic relationship to the resulting system. This greatly simplifies the build, verification and deployment process for a distributed system. The chosen branching strategy becomes irrelevant, and the jobs themselves are composable, i.e. `ci.build-candidate`

[8] this graph was constructed by a tool called `pipdeptree`[12] which inspects the requirements specified in each package's `setup.py`. Third-party packages were deliberately omitted.

[9] Any change that does not meet the quality criteria must have its *path to production* curtailed. Deployment pipelines are a natural fit for this.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH2003

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

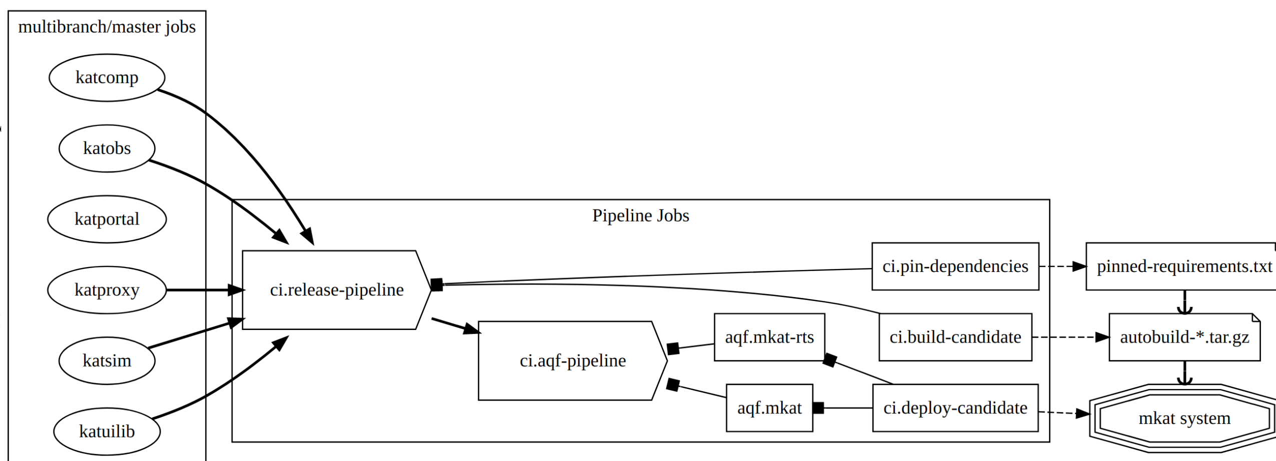

Figure 3: CAM Deployment Pipeline in Jenkins

needs no knowledge about the system except how to convert any given `pinned-requirements.txt` into an `autobuild-*.tar.gz`.

FURTHER WORK

There is still a lot more scope for improving the delivery of CAM and the subsystems that support it. We see opportunities to make AQF tests more reliable and execute faster. That is, lower the rate of false negative test results and refactor tests into smarter slices of scenarios.

Dependency-driven CI

Creating a comprehensive build process that fully resembles the CAM package dependency tree is another area to be pursued so that all changes trigger the correct builds at the right levels. Any and all changes should *fan-in* to producing a complete system that can be verified. This implies daisy-chaining each component's build process in the appropriate order.

Intra-Release Path to Production

An automated *path to production* is still required For changes that need to be applied to the system for which a full redployment of all the nodes is not feasible – such as critical fixes and security patches. CAM unfortunately does not implement a loosely coupled, services-oriented architecture such that each component can be independently packaged in a container[10]. In lieu of this, strategies could include leveraging OS and language package managers[11] to deliver changes to running nodes.

Network-independent Image Creation

Lastly, the image baking process can be sped up by refactoring out the need to boot and provision a node. An approach chosen because the deployment scripts are numerous and untested, so they difficult to verify and therefore difficult to change. Introducing infrastructure testing would allow verification of the resulting server image configuration and therefore modify the operations that create it.

[10] Sometimes broadly referred to as a *microservices* architecture.

[11] `apt` for Ubuntu and `pip` for Python.

ACKNOWLEDGEMENTS

The author would like to thank The South African Radio Astronomy Observatory and the National Research Foundation for supporting this work. Much gratitude is also extended to the exceptional colleagues in the Software Team who together are responsible for the world-class development and upkeep of Control and Monitoring aspects of MeerKAT. Your support and contributions are immeasurable.

REFERENCES

[1] N. Marais, "MeerKAT Control and Monitoring System Architecture", in *Proceedings, 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015)*, Melbourne, Australia, October 2015, Paper MOPGF067. DOI: 10.18429/JACoW-ICALEPCS2015-MOPGF067.

[2] L. Brederode and L. Van den Heever, "MeerKAT Project Status Report", in *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017)*, Barcelona, Spain, October 8-13, 2017, aper THPHA066. DOI: 10.18429/JACoW-ICALEPCS2017-THPHA066.

[3] N. Marais and P. S. Swart, "Virtualization and Deployment Management for the KAT-7 / MeerKAT Control and Monitoring System", in *Proceedings of ICALEPCS2013*, San Francisco, CA, USA, 2013, THCOBA06.

[4] B. Xaia, T. Gatsi, and O. Mokone, "Automated Software Testing for Control and Monitoring a Radio Telescope", in *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017)*, Barcelona, Spain, October 8-13, 2017, Paper THPHA164. DOI: 10 . 18429 / JACoW - ICALEPCS2017 - THPHA164.

[5] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH2003

[6] K. Morris, *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.

[7] A. Couch and Y. Sun, "On the algebraic structure of convergence", in *International Workshop on Distributed Systems: Operations and Management*, Springer, 2003, pp. 28–40.

[8] L. van den Heever, S. SA, *et al.*, "Meerkat control and monitoring-design concepts and status", *Proceedings of ICALEPCS2013*, San Francisco, CA, USA, 2013.

[9] A. Couch and Y. Sun, "On observed reproducibility in network configuration management", *Science of Computer Programming*, vol. 53, no. 2, pp. 215–253, 2004.

[10] R. Penners and A. Dyck, "Release engineering vs. devops-an approach to define both terms", *Full-scale Software Engineering*, pp. 49–54, 2015.

[11] J. Humble, C. Read, and D. North, "The deployment production line", in *AGILE 2006 (AGILE'06)*, IEEE, 2006, 6–pp.

[12] V. Naik, *Naiquevin/pipdeptree*, original-date: 2014-02-02T17:45:23Z, Sep. 24, 2019. https://github.com/naiquevin/pipdeptree

[13] N. T. Blog, *How We Build Code at Netflix*, Mar. 2016, https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15

[14] *Jazzband/pip-tools*, original-date: 2012-09-10T08:50:26Z, Sep. 28, 2019, https://github.com/jazzband/pip-tools

© Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.