EGEE-III

DEFINITION AND DOCUMENTATION OF THE REVISED SOFTWARE LIFE-CYCLE PROCESS

MSA3.4.1

Document identifier:	EGEE-III-SA3-TEC-MSA3.4.1-v0_7.doc
Date:	**14/10/08**
Activity:	SA3
Document status:	**DRAFT**
Author:	**Oliver Keeble & Markus Schulz**
Document link:	https://edms.cern.ch/document/973115/1

Abstract:

This document describes the gLite release process. It explains the motivation behind the current approach to change management and describes the actors and states in the process.

Document Log

Issue	Date	Comment	Author/Partner
0-3	16/10/08	Ready for Review	

Document Change Record

Issue	Item	Reason for Change

Table of contents

TABLE OF TABLES

1. INTRODUCTION

1.1. PURPOSE

A description of the gLite middleware release process.

1.2. APPLICATION AREA

This document is relevant to all participants in the gLite release process, as well as all middleware stakeholders with an interest in the maintenance of gLite.

1.3. REFERENCES

Table 1: Table of references

R 1	EGEE-II MSA3.2
R 2	EGEE-III MSA3.6 - Developer Guide – currently in work
R 3	EGEE-III DSA1.1 - Plan for the continued development of the global Grid User Support (GGUS) infrastructure.
R 4	Savannah - https://savannah.cern.ch/patch/?group=jra1mdw
R 5	EGEE-III Description of Work - https://edms.cern.ch/document/886385/

1.4. DOCUMENT AMENDMENT PROCEDURE

Amendments, comments and suggestions should be sent to the authors. The procedures documented in the EGEE "Document Management Procedure" will be followed: http://project-EGEE-III-na1-qa.web.cern.ch/project-EGEE-III-NA1-QA/EGEE-III/Procedures/DocManagmtProcedure/DocMngmt.htm.

1.5. TERMINOLOGY

This subsection provides the definitions of terms, acronyms, and abbreviations required to properly interpret this document. A complete project glossary is provided in the EGEE glossary http://egee-technical.web.cern.ch/egee-technical/documents/glossary.htm.

Glossary

yaim	gLite configuration utility
cluster of competence	collocated JRA1 and SA3 resources
subsystem	A collection of related middleware components under a single administrative control

2. INTRODUCTION

2.1. SUMMARY OF THE CURRENT SITUATION

EGEE-III inherits a 'continuous release process' which was defined and optimised during EGEE-II and described in the SA3 milestone MSA3.2 [R1]. The guiding principle of this approach is to manage independent updates in isolation from one another, allowing each to progress through the full software lifecycle without interference. This has proven to bring numerous benefits, particularly in accelerating the adoption of important changes which can be properly prioritised and which are not held back by other unrelated updates.

For this process to work, the production infrastructure is in a state of continual managed evolution; updates must be backward compatible and as reliable as possible. The process described here has been designed to promote a stable production infrastructure and relies on the fact that the middleware has now passed its experimental phase and is of sufficient maturity to support such a process.

2.1.1. Current throughout

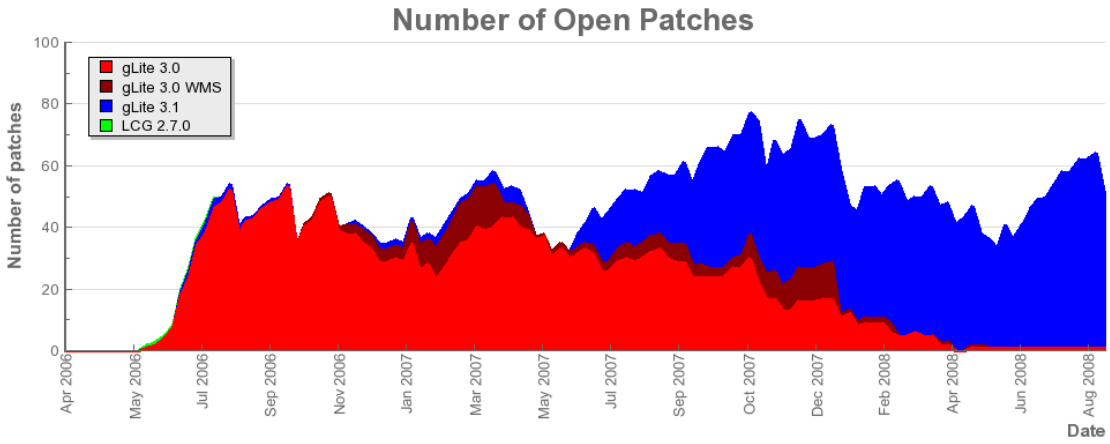

Illustration 1: Patch processing rate

The diagram above shows how many patches (see definition later) the process is handling at any one time. It indicates that at any one time the project may be treating over 60 independent changes to the middleware.

2.2. DEFINITIONS

The terms used here are based on existing objects in our release tracking tool, savannah [R4]. As such the terminology has a precise meaning which may be slightly counter-intuitive.

2.2.1. Bug

A 'bug' is a 'change request'. It may represent the reporting of a 'defect' or the wish for an 'enhancement'.

2.2.2. Patch

A 'patch' is a self-consistent change to the middleware stack, and is the principal object on which the release process acts. It implements some combination of defect fixes and new functionality. It is a set of software packages and associated meta-information. Guidelines for what exactly a patch should contain are given in the Developers' Guide MSA3.6 R[2].

3. NEW PROCESS DEFINITION

3.1. DESCRIPTION OF THE LIFECYCLE

The process presented here describes software change management from the provision of release candidate packages to their deployment on the production infrastructure. It is not concerned directly with development and build issues (which are handled in the Developers' Guide MSA3.6 [R2]) and it applies equally to software provided by third parties – the introduction and management of such components is handled in the same way as those supplied from within the project.

It should be noted that this process is flexible enough to support the introduction of entire new services to the gLite stack. Although more substantial pre-certification validation is required (see section 3.8), the standard patch process can be used thereafter to manage the introduction of the new service to production. The process can equally handle the introduction of support for a new platform. Removal of a component from the release is handled differently and is discussed in section 3.9

3.2. PROCESS OVERVIEW

3.2.1. Actors and roles

Developer

The developer is responsible for the maintenance of a particular part of the codebase, which includes fixing bugs and adding new features to components. The developer should fix the bugs assigned to him/her, and the priorities for fixing bugs should come from the subsystem integrator and the EMT. The developers should check the fixes into CVS and inform the subsystem integrator that the fix is available.

Subsystem Integrator

The subsystem integrator is responsible for releasing the subsystem. To make a release, they have to tag CVS and trigger the appropriate build. In the case of 3rd party contributions to the stack this role involves collecting the necessary binaries from the upstream project. The role typically involves initial triage and assignment of bugs within the subsystem.

Patch Provider

The Patch Provider is responsible for verifying that a middleware update is of sufficient quality to be considered for release, and for creating the patch itself. This is typically an SA3 member of a 'cluster of competence' but can be a developer or the representative of a 3rd party project.

Release Manager

The release manager is responsible for overall coordination. Main duties include:

- Chair the EMT and ensure that its agenda reflects all relevant issues of the day
- Assure the good functioning of the release process
- Manage release delivery and scheduling
- Interact with software suppliers, including those external to the project

Certifier

This is the SA3 member responsible for verifying that a particular patch meets the certification criteria, i.e. that the software update is deployable.

Repository Manager

SA3 role responsible for ensuring that the various repositories used by certification, PPS and production have the correct packages and function properly.

3.2.2. Decision-making bodies

The **EMT** manages the short-term release priorities for the project. It meets twice a week and manages development and certification effort with respect to issues of current importance. Each JRA1 development cluster has a representative responsible for conveying decisions to the developers. SA3 chairs the meeting and SA1 is represented.

The **TMB** legislates on medium and long-term direction, such as the introduction or retirement of services and platform support.

These bodies are described further in the EGEE-III Description of Work [R5].

3.3. BUG STATES

3.3.1. None

Initially a new requested feature or a defect is entered in the tracking system as an unassigned **bug** in the state *none*. This is automatically assigned to the appropriate subsystem manager who then reassigns this to a developer.

Note that the 'severity' of a bug is chosen by the submitter. The subsystem manager must take this into account, but it is not the definitive expression of the bug's impact on the project. This issue is discussed in section **Error! Reference source not found.**3.6.5.

3.3.2. Accepted

If the developer accepts the bug as valid, they put it to the state *accepted* . This state can be skipped if the developer is able to start work immediately on the issue.

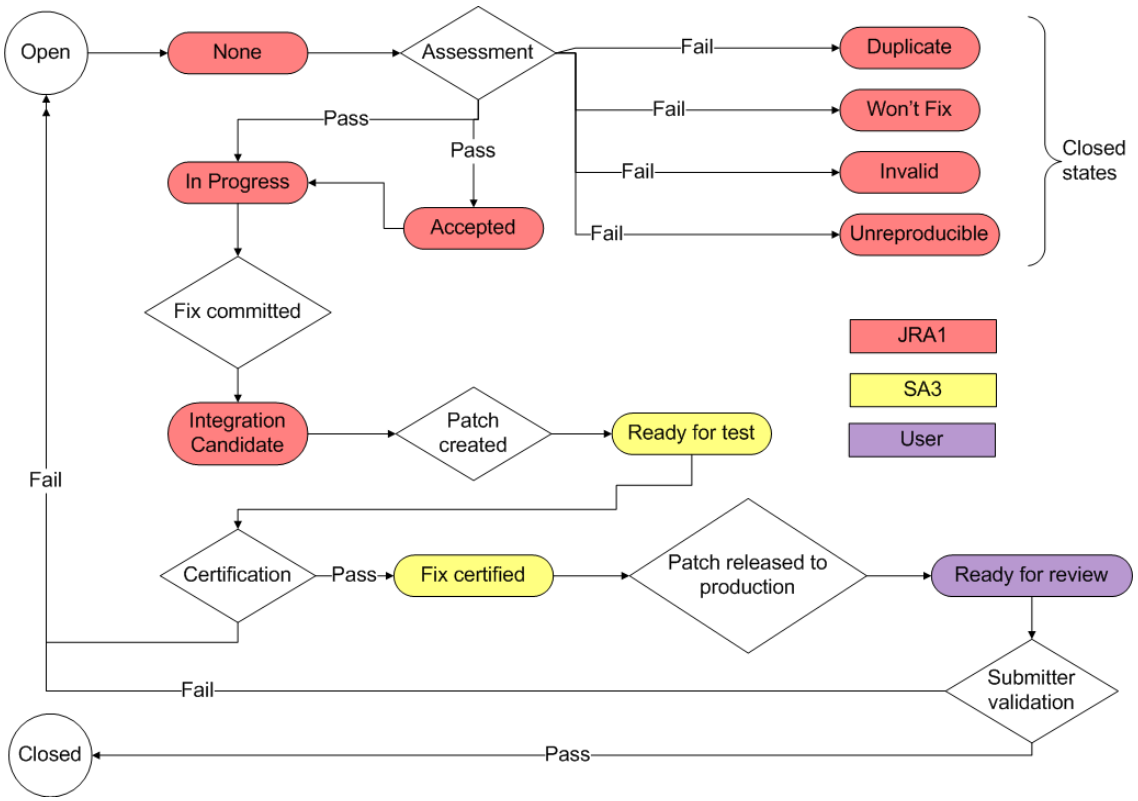

Illustration 2: Bug States

The developer puts the bug in the state *in progress* when they start work.

3.3.4. Integration Candidate

When the developer has fixed the problem, checks the fix into CVS and the bug state is changed to *integration candidate.* The fix should have been tested by the developer but will not necessarily have been checked by any other party.

3.3.5. Ready For Test

The subsystem integrator will monitor all the bugs in this state and decided when to make a release. To make a release the subsystem integrator tags CVS with a release tag and creates a **patch** that includes all the information required. The **bug** is put into the *ready for test* state, and remains in this state throughout the certification process.

3.3.6. Fix Certified

When the corresponding patch has been certified, the certifier will move the bug state to *fix certified*. If the certifier demonstrates that the bug has not in fact been fixed, he/she will remove the

bug from the patch and set the bug state back to *None*. If a patch is rejected for any reason, none of its bugs should be set to *fix certified* as the replacement patch may contain a regression.

3.3.7. Ready For Review

The process envisages that a bug can be closed in the 'area' where it was detected. A developer who registers a bug during the development process can close it themselves, but a bug found in production must be demonstrated to have been fixed in production before it can be closed.

Once a patch is released to production, its bugs are placed in the state 'ready for review' which allows the original submitter to comment or close the bug. After 2 weeks in this state, the bug will automatically be closed. It can always be reopened if necessary.

3.3.8. Other states

There exist various other self-explanatory states for a bug, such as *unreproducible*, *invalid* and *duplicate*. A developer can move a bug into any of these states, with appropriate justification, if they do not *accept* the bug (this can happen from the *in progress* state too if necessary). The bug should then be closed.

The state *won't fix* also exists to cover other scenarios not explicitly covered above.. The reason for transition to this state should be clearly explained (eg insufficient effort available).

3.3.9. A note on user support

EGEE has a mature process in place for handling user support (GGUS) which is described in DSA1.1 [R3]. In certain cases it results in a software bug being registered in Savannah. Experienced users can also submit bugs directly, bypassing the user support system.

The GGUS route does introduce a complication – it is the GGUS Ticket Processing Manager (TPM) who actually submits the Savannah bug, not the originator of the GGUS ticket.

3.4. PATCH STATES

Patches are explained in section **Error! Reference source not found.**2.2.2. A patch should only be opened by the person responsible for the software that is providing the fix or new functionality to the middleware stack.

3.4.1. With provider

A developer or representative of an external software project can leave a patch in this state as long as they wish. All relevant bugs fixed should be attached to the patch at this time. Some projects may choose to use this state in order to establish criteria for the eventual submission of the patch.

3.4.2. Ready for Integration

The patch provider moves the patch into the state *ready for integration* when they wish to trigger the release process. At this stage, a certifier will check that the patch satisfies various

acceptance criteria (detailed later). If the patch does not satisfy these criteria, it is not rejected but is put back into the state *with provider*, with appropriate comments.

This state represents a statement by the Cluster of Competence that the software is a credible release candidate, thus is should already have undergone all pre-certification testing available.

The release manager will limit the number of *with provider – ready for integration* iterations which are allowed in practise (the limit will be around 5 but is discretionary).

3.4.3. Ready for certification

The repository manager has prepared the repository for the patch and it is ready to be taken by the certification team.

3.4.4. In Configuration/Configured

If a middleware change requires a corresponding change to the configuration utility yaim, the patch will pass through these two states before entering certification.

3.4.5. In certification

The patch is being certified. Reports and dialogue about the patch will accumulate in the savannah patch itself.

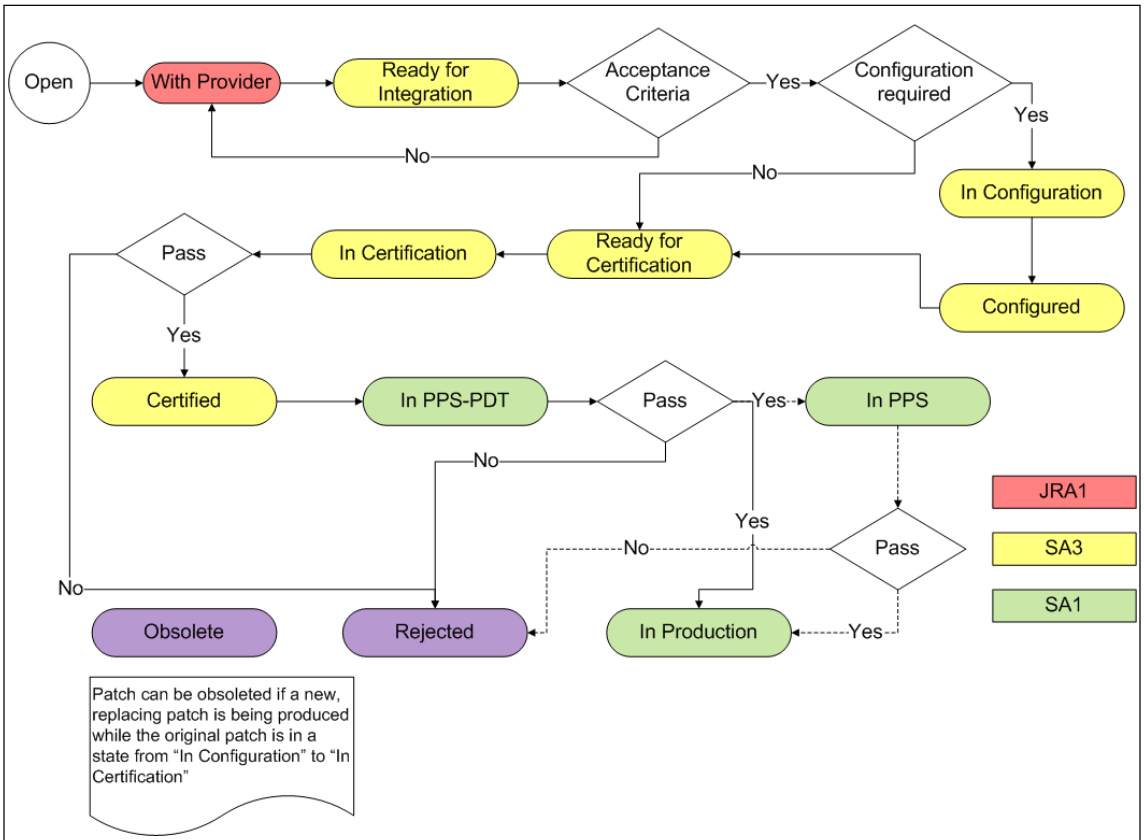

Illustration 3: Patch States

3.4.6. Certified

The patch has passed certification. The criteria for this are updated continually according to the available tests and the requirements of the particular patch, but as a minimum the patch must represent a beneficial change to production which is deployable and introduces no regressions.

At this stage the certification itself is checked by the certification coordinator. If problems are found, the patch is put back into the state *In Certification*.

3.4.7. Rejected

If the patch is found to break existing functionality, or not provide the planned fix, it will be rejected. Rejection happens when new packages would be required to advance further in the process. A full description of rejection criteria is provided by the certification team.

The case of rejection from PPS is not currently used. The patch is put on a priority 'on hold' and a newer patch is awaited.

The case of rejection from Production must be considered. One way to handle rollback of a problematic production release is to produce newer versions of the relevant packages which represent the older, known-good code. While this works in the case of a bugfix release, when a new major

version of the component is released, an upgrade carrying the new major version cannot regress to earlier more limited functionality.

Packages may sometimes have to be removed from the production repository, i.e. a patch can be rejected from production. This is problematic and at the moment involves at least the following steps;

- An announcement is made, which will be referred to as a 'release' even though it does not introduce any new middleware
- Updated metapackages have to be provided to revert dependencies to earlier packages
- Administrators have to manually intervene on their nodes to uninstall the offending components before reinstalling the older packages
- Documentation, 'known issues' etc have to be updated to be consistent with what is in the repository

3.4.8. Obsolete

If a newer patch for the same component arrives while the first patch is in any state except production, the older patch is considered to be *obsolete* and is closed. In many cases, the older patch will have accumulated important information in it which must be transferred to the newer patch.

If a patch is obsoleted from PPS, it is not removed from the repository, but it progresses no further.

3.4.9. PPS Deployment Test & PPS

Releases of currently certified patches are regularly made to the Pre-Production Service. The patches move through the two states above during this process, but it should be noted that PPS is under review and in future these states may represent a number of 'sub states' tracked within the Pre-Production Activity itself.

3.4.10. Production

The patch has been released to production and announced. Note that this state gives no guarantees about the deployment status of a particular update.

3.5. ACCEPTANCE CRITERIA

On the introduction of a new service or a major upgrade, the following acceptance criteria apply. In the case of existing services these criteria can be retrospectively applied in a progressive manner.

3.5.1. Before Certification

An instance of the service and client available on a set of test nodes installed and configured by the developer. The machines will be provided with a base OS preinstalled to ensure that dependencies are consistent. The developers then add their components and configure them by whatever means. Simple functional tests have to be provided. These do not need to follow any

framework. The savannah patch has to be properly filled in (according to the instructions linked from the patch submission page). If the patch introduces new dependencies to the project, this must be discussed at the EMT.

Minimal required documentation at this stage:

1. Functional description of the service;
2. User documentation (man style quality) to allow testers to start;
3. List of "sub services" and their role, this includes location and description of log files;
4. List of processes that are expected to run, giving a typical load of the service;
5. A description on how state information is managed. Examples are: file xxx contains the state of active transfers, DB table YYY is used to maintain the state of all jobs ever run;
6. A statement on whether the state be rebuilt from other sources;
7. Description of how to follow audit trails;
8. Description of configuration, no detailed document, but a (simple "do this, do that");
9. Port list, including which services are expected to connect to the specified ports or port ranges;
10. Description on how to start, stop and inquire service state;
11. Notes on the results of the pre-certification testing;
12. Description of how to query the version of a service or client;

SA3 enforces the above criteria.

3.5.2. Between Certified and In Pre Production

At this stage the patch has passed already through certification and is on its way to be moved to the distributed pre-production test bed. Therefore much more stringent requirements at the deployability and operational friendliness are required:

1. All of the above;
2. Configuration tools and detailed description of the configuration parameters;
3. If not build via gLite ETICS instance a finalized list of dependencies;
4. Statement on 32/64 bit compliance;
5. Statement of functionality that will be supported including an estimated scale. This can be a subset of the functionality expressed via the API;
6. Tests for supported subset functionality in a form that can be adopted by SAM;
7. A statement on whether this service can or must be cohosted with any others. The assumption is otherwise that it will be installed standalone;
8. Statement on whether the component can be installed and configured from user level;
9. An initial operations guide containing information such as:
    1. how to drain a service;

2. effects of restarting services;
3. what actions are needed that configuration changes become active;
4. effects of services being stopped abruptly, what cleanup process is needed in this case;
5. effect of service unavailability on other services, a good example is the MyProxy service whose absence will make all long running jobs fail;
6. A statement on service maintenance; how to ensure the good running of the service over extended periods (for example recommended log rotation and DB cleanup strategies.);

SA3 enforces the above criteria.

3.5.3. Between In Pre-Production and In Production

At this stage some experience with a component has been gained by site managers and users. The components have been used in production-like environments for extended periods. Moving the services out of the hands of the experienced PPS site managers and users and deploying them in scale on the Production Service with at least an order of magnitude more sites, resources, and users of various degrees of expertise, makes it mandatory that additional conditions are met:

1. All of the above;
2. A statement on accounting and resource partitioning between different Vos;
3. End user documentation, covering at least the common use cases;
4. An expanded operations manual including information on:
    1. Load-balanced deployment (although this may not always be possible or required);
    2. High-availability deployment scenarios, if available and required;
    3. A description of common operations problems and the recommended solutions;
    4. Migration of services between nodes;
5. Reference to a statement by the TMB that endorses the deployment of the service;
6. Tests integrated into SAM;
7. A description of typical deployment scenarios which will allow sites and regions to plan;
8. Installation and configuration documentation covering common deployment scenarios;
9. A statement on support provisioning for the component;
10. A note on any known incompatibilities with former component versions, and a migration plan if necessary;
11. A statement by a system operator relating to the memory consumption and other resource requirements of the service(s) under pre-production load;

SA1 enforces the above criteria.

3.6. NOTES ON IMPLEMENTATION

This guide has deliberately kept details on the actual implementation of the release process to a minimum. Nevertheless, some comments are warranted;

3.6.1. Savannah

The bug and patch process are tracked in the savannah tool.

3.6.2. Other savannah fields

The bug 'priority' field is currently used by the EMT to indicate which bugs it is actively tracking.

3.6.3. Process Monitoring

SA3 actively monitors the release process, taking the necessary information from savannah, either via the standard web interface or via a read-only dump of the database which we have arranged with the service provider. There are various tools which serve different purposes;

Process Overview – a synoptic view of the whole process is offered, indicating which patches are where. This allows the release manager to see which patches are in need of intervention because they have remained too long in a given state, or have not been updated recently enough. http://lxbra1902.cern.ch/gLite/Savannah/PatchTaskMonitor/ptMon-index.php.

Consistency checking - we can check, for example, whether bugs attached to a patch are in the correct state (e.g. 'ready for review' for a patch in production) and send alerts in the case of inconsistency.

Statistics – We generate retrospective release-process statistics, such as the number of patches released over time, average time spent in a particular state, average time to fix a certain type of defect etc. This allows the activity to identify the areas most amenable to improvement and allows us to quantitatively determine whether the measures we take as a consequence are effective. http://glite.web.cern.ch/glite/statistics/PatchStatistics.asp

3.6.4. Process automation

Efforts are being made to automate as much of the release process as possible. This is difficult because the tracking tool, Savannah, does not offer an API. Nevertheless, progress is being made; a command line tool, which mimics the behaviour of a human-driven browser, is being developed and offers limited programmatic 'read/write' access to Savannah.

3.6.5. Objective classification of bug severity

As previously noted, the severity of bugs is defined at submission time by the submitter. Experience shows that this does not always reflect the general impact on the production infrastructure; a defect which affects only a single user but which prevents them from working, will be 'critical'.

The EMT should discuss bugs of severity major or critical and adjust the severity if appropriate.

3.7. DEPLOYMENT AND ROLLOUT

Release to production is currently the last stage of the formal process, but this leaves an ambiguity at the end of the road; a user cannot be guaranteed that a particular service is deployed and available to them, even if it has been released; the software has to be rolled out.

Along with SA1 we are investigating how better to manage this 'last mile'.

3.8. VALIDATION OF LARGE CHANGES

Under 'large changes' we categorise the introduction of new services, significantly reengineered components, the introduction of significantly increased critical functionality or non-backward compatible changes.

Often it is desirable to validate these changes early, sometimes even before the component reaches the certification state. This is the case especially for new services.

In many cases, acceptance of a new or significantly changed service into the middleware stack requires demonstration of acceptable scalability. What is acceptable is defined on a per-service basis and can be checked in a number of ways; if not in pre-certification testing, then via one of the approaches described below. Certification itself does not generally check scalability issues.

Given the wide variety of possible scenarios it is not practical to define a process that addresses the specific needs of all of the scenarios. We will describe a few categories and introduce a process for planning and coordinating the different activities. This is required, because this work has an impact on the pre production activity, the developers, SA3, SA1 and the end users.

3.8.1. Pilot Services

These are services that lead the way into production. The service has been properly integrated and built using SA3's standard process, configuration management via YAIM is provided and the service has gone through functional testing in the SA3 certification team. The service will be operated at a small number of selected sites to verify in an production environment one or more of the following qualities; operability, usability, scalability, performance.

The decision to launch a service first as a pilot will be taken at the TMB and coordinated by SA3. SA1 identifies and coordinates the pre production activity partners. NA4 identifies interested user communities that are willing to participate in the validation. If no interested user community can be identified no pilot service should be provided.

The experience is documented by all partners that are involved, with SA1 coordinating the activities and reporting progress to the TMB. Defects are reported and tracked via Savannah.

Before a pilot service is launched the expected results have to be agreed on. When all partners agree that these have been achieved, the component is rolled out to production.

3.8.2. Experimental Services

Experimental Services are setup before the certification stage to help components to mature more quickly, especially in terms of scalability and stability under production load. For this, rapid feedback between developers, operators, and users is required.

The TMB agrees on the setup of Experimental Services based on a written proposal by SA3. This has to identify partners that are willing to host and operate the service, users that agree to run typical use cases, developers that are committed to address found problems quickly. In addition quantitative targets have to be specified and agreed on. The maximal lifetime of an experimental service has to be defined before it is launched.

Progress reporting and coordination is handled in the same way as for the Pilot Service, except that here issues are reported directly to the developers. The use of Savannah in not mandatory, but found issues have to be documented.

After the Experimental Service meets the agreed targets a release has to be produced by the development team, using the standard build and packaging tools. After it has been verified that a service based on these components show similar behaviour as the Experimental Service, the component enters the standard release process. Scalability tests are for efficiency reasons not repeated.

3.8.3. Preview Services

User requirements and functional descriptions of new components and functionality are often ambiguous. To clarify their behaviour it is occasionally required to setup an early prototype to be used by the targeted user community.

The process for the definition of preview service is the same as the one described for the experimental service. However, preview services should have only a very limited lifetime and if possible be embedded in local test beds. In most cases the preview services will be installed and maintained by the developer teams. After termination preview services can, if successful enter the standard software lifecycle process.

3.9. END OF LIFE

We have described a mature process which serves essentially for adding features and services to gLite. Without an equivalent mechanism for retiring old services or platforms, we risk spreading our resource too thinly and creating confusion over what is really supported. For this reason SA3 has triggered a process within the TMB to formalise this retirement process; we expect it to form part of MSA3.4.2 (the 2nd year revision of this document). A consultation exercise is ongoing within SA1.

3.10. RELEASE DELIVERY

As currently described, the release process is independent of exactly what is being released, both on the level of the software itself and its packaging. In reality, our releases consist of rpms and a the action of 'releasing' involves updating an apt or yum repositories. There are however some other release delivery mechanisms and processes worth mentioning.

3.10.1. Shared Area

A number of EGEE VOs integrate middleware clients into their applications, which they then distribute across the grid independently. In order to better support this, SA3 makes available a special release of the clients in the form of a hierarchical, versioned directory structure on shared storage, which enables the applications to target a specific version and platform of a particular binary. This repository is updated whenever a client release is made.

3.10.2. Binary client tarball

Binary tarballs suitable for installing WNs or UIs are distributed whenever a client update is made. These tarballs serve a number of purposes.

For the UI, they allow installation in user space, and equally allow the provision of a UI on a shared disk, where a user simply has to invoke the correct environment to use it. This is how the much-used UI on the CERN lxplus service is maintained.

For the WN, this allows a single shared area to provide middleware to an entire farm of WNs. It also gives the possibility of remote installation of middleware, as it can be performed entirely in userspace. This option is being investigated with SA1 as a rollout mechanisms.

3.10.3. Source

There is a requirement for distribution of middleware source alongside the binaries. This is for two distinct reasons

- Verifiability – sites would like the right to inspect the source of the binaries they install to satisfy themselves that the software is appropriate for them.
- Local builds – sites would like to build their own distributions of the middleware, either to support a particular architecture or to implement localised changes or optimisations.

gLite is a complex project with a build to match. Making a buildable source distribution would require an investment of effort currently prohibited by other priorities. However, the first requirement can be met simply by distributing source tarballs of what was built by ETICS (with no build instructions). This is an outstanding requirement from SA3 on the ETICS project.

3.10.4. RESPECT

The NA4 activity maintains a program called RESPECT which “identifies software packages that integrate with the core gLite software and provide functionality of interest to the EGEE user community”. This is a release route for software which can form part of a larger ecosystem without being explicitly part of gLite itself.

3.11. SUMMARY OF IMPLEMENTATION IMPROVEMENTS FORESEEN

Here are summarised the main improvements foreseen (all mentioned elsewhere in the text).

- Dead patch states - 4.1.1 – expected PM12

- Rollback - 3.4.7 - expected PM12
- Process automation - 3.6.4 - ongoing
- End-of-life - 3.9 – expected PM12
- Validation services - 3.8 – expected PM15

4. MOTIVATION

This chapter gives some detail on the reasons behind the changes introduced in this revision of the release process definition. It is intended as background material and is not necessary for understanding the process itself.

4.1. ISSUES

During the course of EGEE-II, a number of issues and opportunities for enhancement of the process were noted. These are briefly discussed here in order to further explain the redefined process described in this document.

4.1.1. JRA1/SA3 handover

It was noted that during EGEE-II around 50% of submitted patches did not reach production. The certification process is expensive, has considerable tracking overheads and, as it involves a number of actors distributed globally, suffers from communication delays. Thus every effort should be made to find problems as early as possible, preferably before the formal certification process is invoked. It is for this reason that EGEE-III sees 'Clusters of Competence' , i.e. the collocation of dedicated SA3 resources alongside JRA1 development effort, in order to provide a lightweight and responsive testing service which will trap basic errors before certification begins.

We also noted that the process of EGEE-II suffered from unnecessary delays where patches remained in 'waiting' states, for example awaiting a release window. These delays could constitute a considerable fraction of the entire patch lifetime and changes to scheduling are envisaged to mitigate this.

Developers have found that their release planning can be helped by creating patches early, even to the point of using a patch to define release criteria, and then working towards the completion of that patch. This work pattern has been reflected now in the 'patch process'.

4.1.2. SA3/SA1 handover

The concept of 'obsoleting' turned out to be in need of better definition, particularly when applied to production. In any other state than 'production', 'obsolescence' indicates that a newer patch addressing the issue is available. In production, it represents the retirement of a service (not currently tracked through this process).

The release process as described has no specific provision for 'fast track' releases. It is hoped that by construction this is natively supported by the process which allows independent prioritisation and propagation of any particular change.

One weakness of the process as described is the inability to roll back changes which have an adverse impact on production. The best solution here is to have a build programme which allows the

rapid provision of a rebuild (with later version) of the previous (known good) version of the package in question. This is not currently realisable with any real reliability, and until then we will have to be prepared to simply remove packages from the production repository in extremis. This presents its own problems, particularly as it requires manual intervention on sites which have already upgraded and is difficult to represent with our current process.

The policy of requiring any bug found in production to have its fix validated by the original submitter before it can be closed has led to a large accumulation of open bugs in the final state, 'ready for review'. The policy is retained in this document, but an automatic closure of bugs (with an appropriate comment in the tracker) is envisaged.

4.1.3. Validation of large changes in the PPS

During EGEE-II we built up a service to provide early access to new or updated gLite components. This service, the PPS, was meant to be used by two groups, end users and site administrators. Both groups have a different focus and the PPS tried to address both.

The users expected to be able to use the upcoming versions and services under conditions similar to the production environment. This required a system of significant scale and stability. The PPS successfully integrated a few thousands CPUs and several sites to allow scalability tests.

The system administrators and regional operations teams used the PPS as a deployment testbed to verify that changes can be rolled out and have no negative impact on the production environment.

The two use cases conflict with each other. While users are eager to test as quickly as possible and at a significant scale, site administrators profit most if they do the same fabric integration that has to be done for their production systems. This local, or regional, integration is time consuming.

The operational experience of the PPS showed that very similar deployment scenarios were used to quickly introduce changes. While this limited the value of the deployment tests, the tests nevertheless identified several packaging, configuration and documentation problems.

End users from VOs with complex, automated workflows are usually dependent on services with state, such as catalogues and storage systems. Redirection these workflows from the production system to the PPS and back proved to be too costly to be pursued on a regular basis. The accounting records collected by EGEE during the last 12 months reflect the situation. All end users submitted a total of 25 jobs to PPS, the infrastructure monitoring system run more than 9500 jobs during the year. It has to be noted that end users during this period have been interested in testing new versions and services. This was mainly achieved by either distributing new clients as part of their applications, or by special production service instances that have been setup outside the standard process.

The other main lesson that was learned during the last years was that synthetic tests can’t reliably uncover all defects in the middleware that affect the production use cases under full load.

Based on the experience with the PPS during the last years we suggest a new approach which can be summarised as follows. Backward compatible client changes and services will be introduced in

the production service gradually. With the help of information system head nodes the PPS service becomes a special view of the production infrastructure. Deployment tests become more realistic, because the services are deployed as part of the production infrastructure. For this approach it is important that releases can be quickly rolled back. For clients this will be achieved by the parallel deployment of multiple versions between which the user switches via environment variables, set via the JDL. For services the situation is more difficult and we have to depend on the PPS site administrator to be willing and capable to quickly move between different instances. Another advantage of the staged introduction of services is that in case of a successful rollout, which is the most frequent situation, the site can continue to run the service in production.

4.1.4. General

In response to recommendations of the EGEE reviewers, the savannah tracking tool has been updated to remove the ambiguity between 'defects' and 'enhancement requests'. In this document, both are referred to as ‘bugs’, as explained in section 2.2.1.

4.2. CHANGING REQUIREMENTS

In addition to specific issues highlighted in the preceding section, new requirements on the process have also evolved during the course of EGEE-II, or can be anticipated for EGEE-III.

4.2.1. Multiplatform

The introduction of multiplatform support in the middleware presents specific challenges to the implementation of the release process. Particularly, one change in the source will produce a number of changed packages downstream which can propagate independently. At the moment, the process envisages treating each such change (e.g. a bugfix on SL4/32, SL4/64, SL5/32, SL5/64 and Debian4/32) as independently certifiable and releasable.

4.2.2. Clusters of Competence

The introduction of 'Clusters of Competence', as described above, raises expectations about the quality of patches which reach certification. The process described here has been modified appropriately.

4.2.3. EGI/devolution

With one of the stated aims of EGEE being to prepare for the introduction of a European grid infrastructure based on a federation of National Grid Initiatives, the process has to be able to support a maximum of heterogeneity in terms of its inputs (which could ultimately be multiple independent software projects) and its platform support.