



Large Scale Web Apps: Managing Releases and Devteam Infrastructure

Jonathan Oxe

O'Reilly Open Source Convention
Portland, Oregon
July 25th, 2006

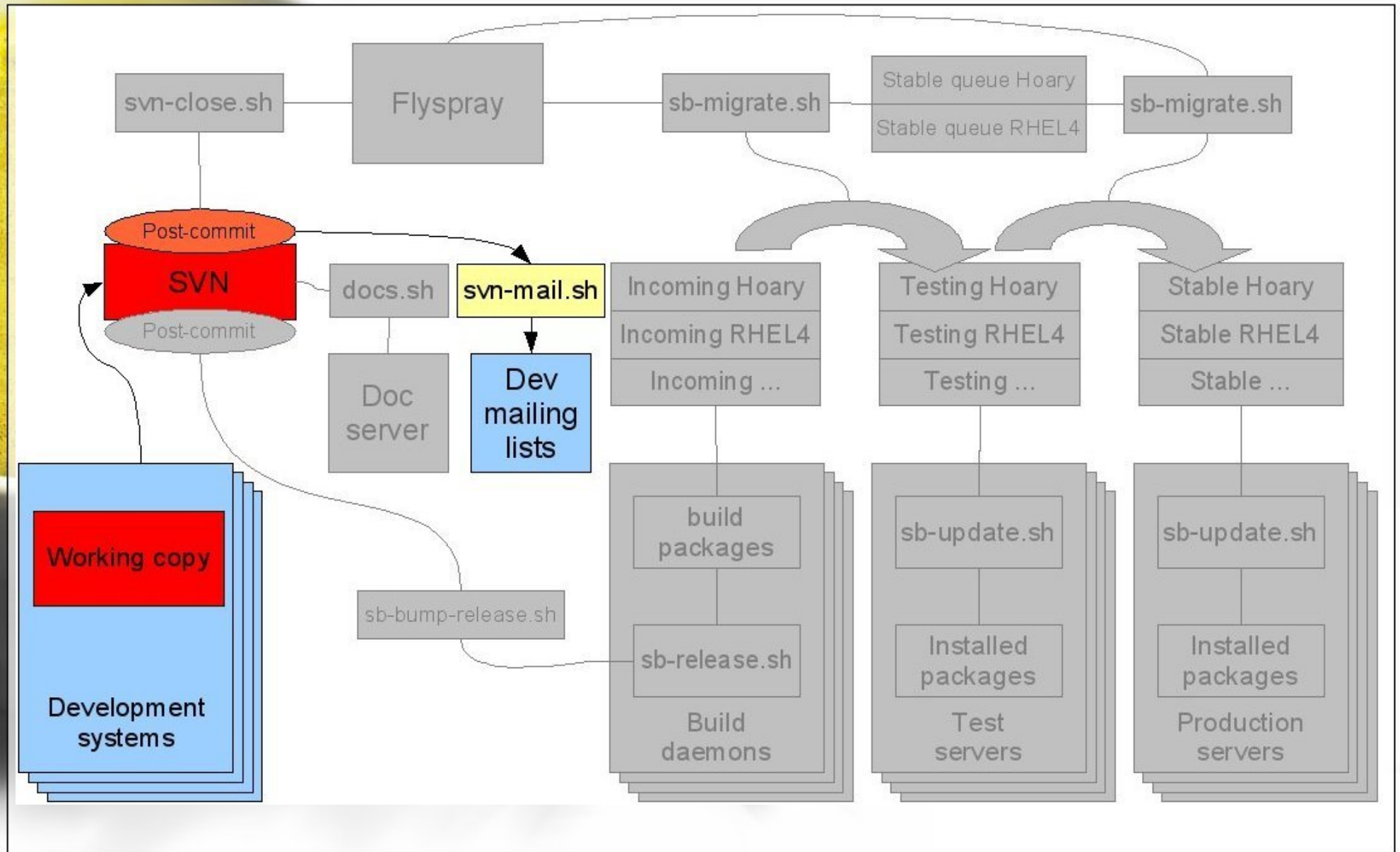
Large Scale Project Infrastructure

Don't have your developers tripping over each other.

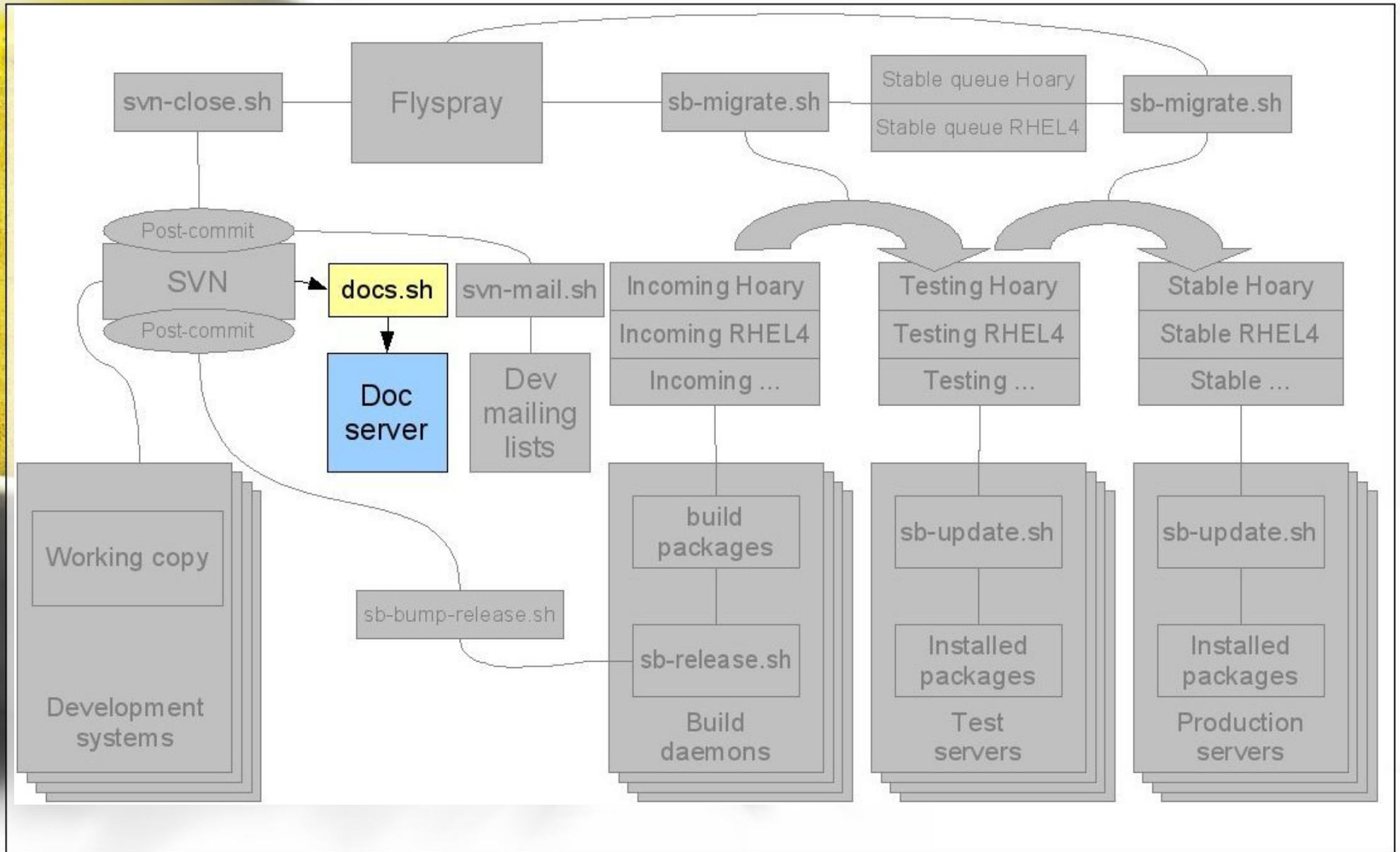
Implement some infrastructure tools to make your life soooo much easier.

Then tie those tools together with scripts, gaffa tape and fencing wire.

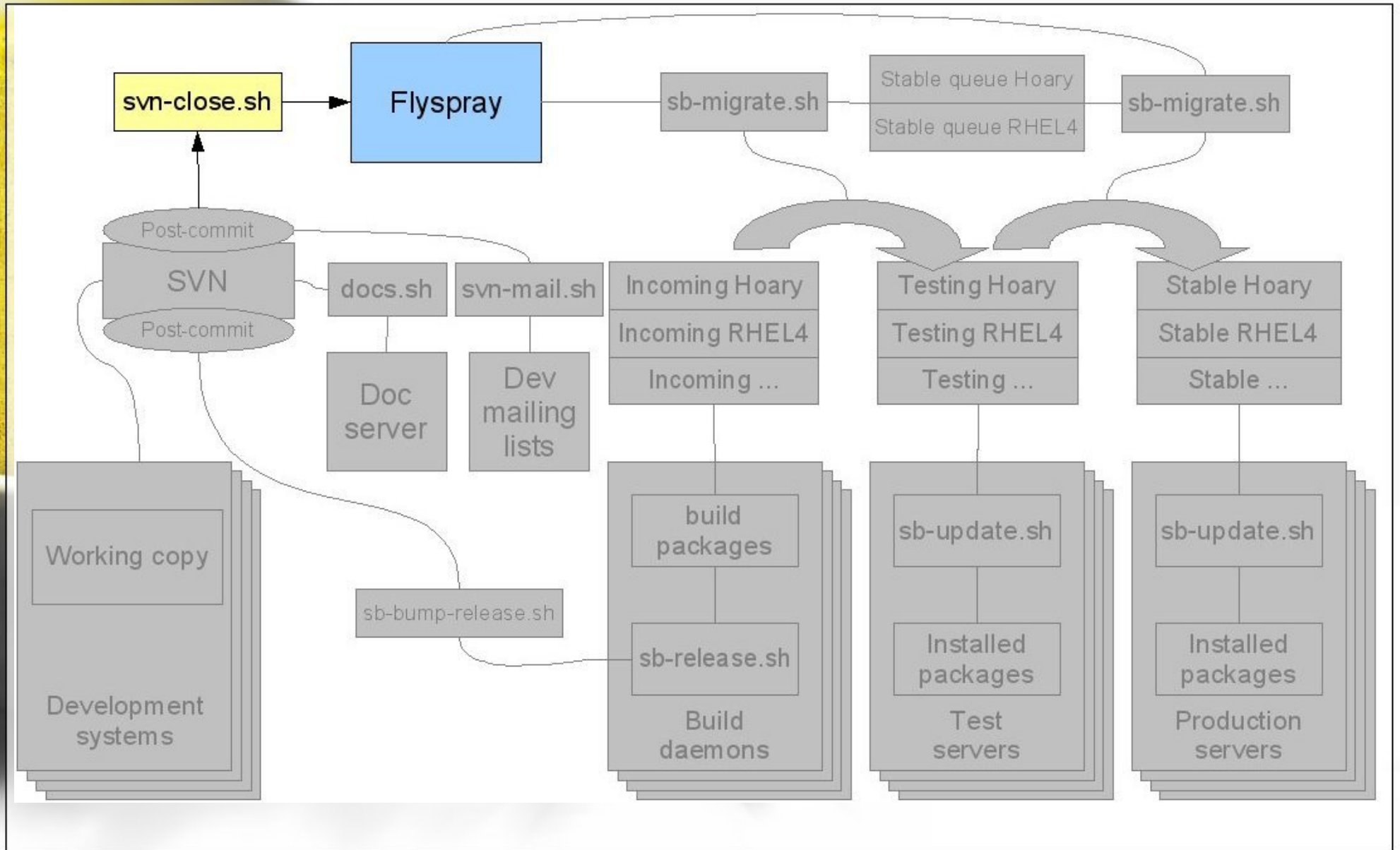
1: Source Code Management



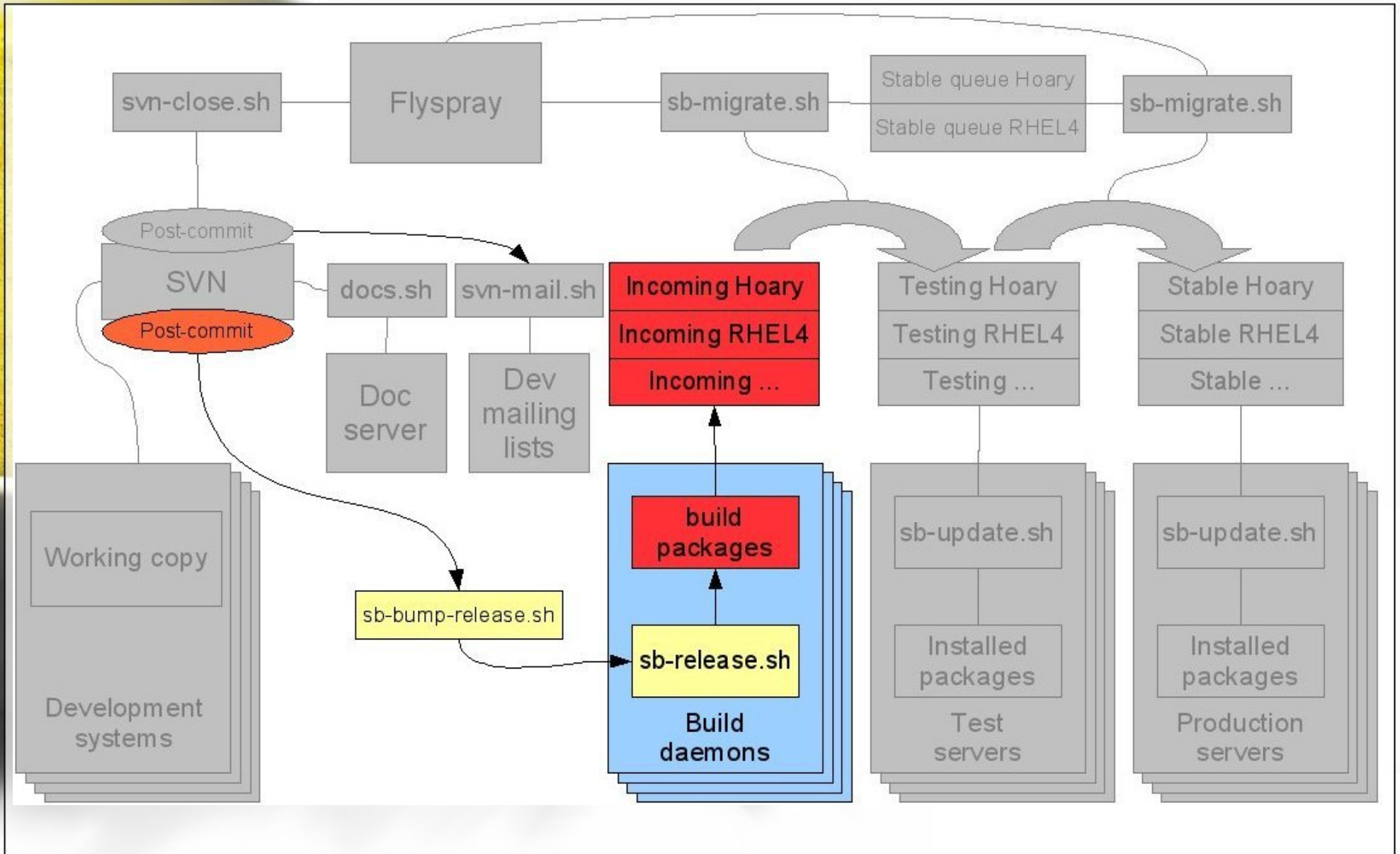
2: Internal Documentation



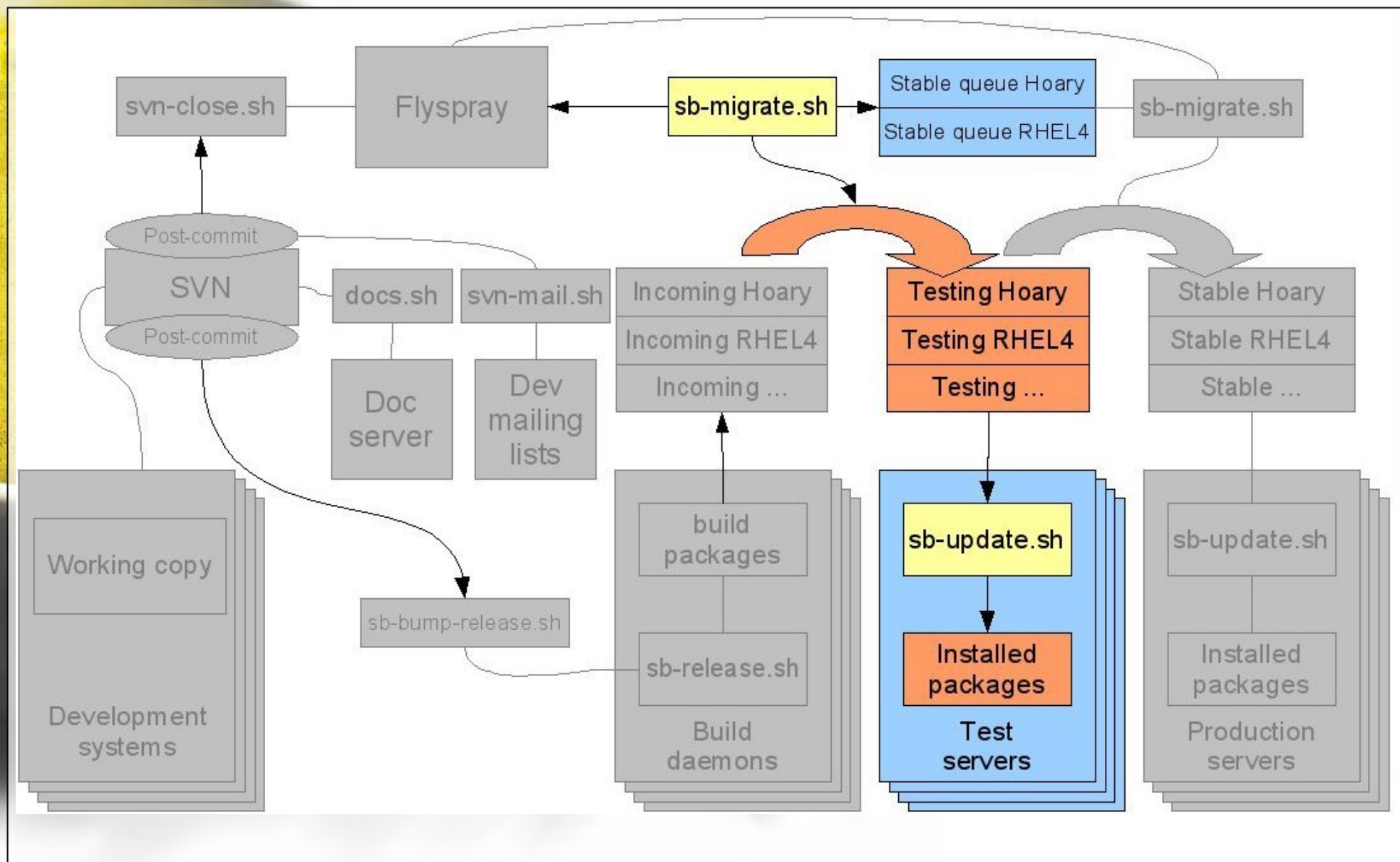
3: Bug Tracking



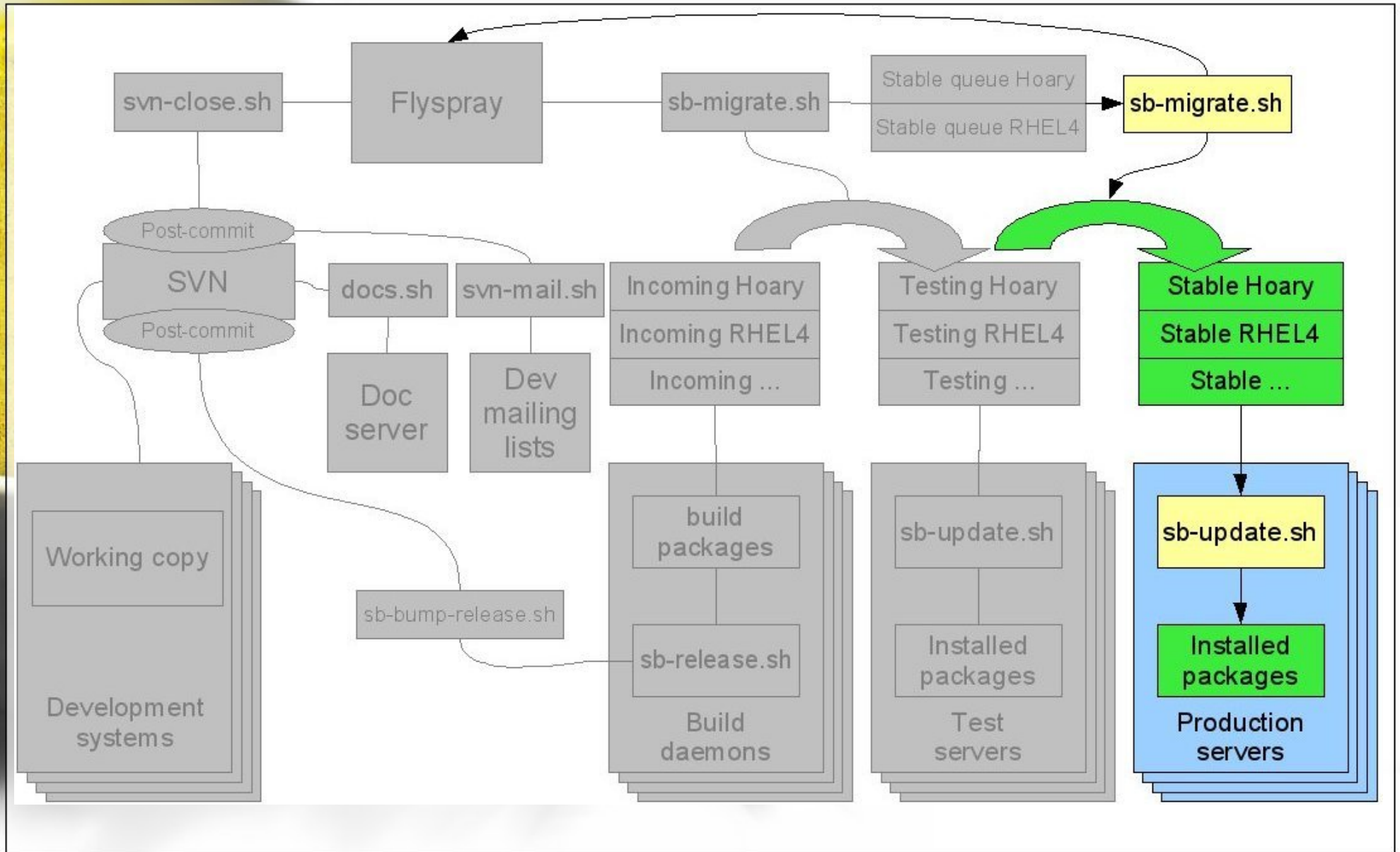
4: Packaging



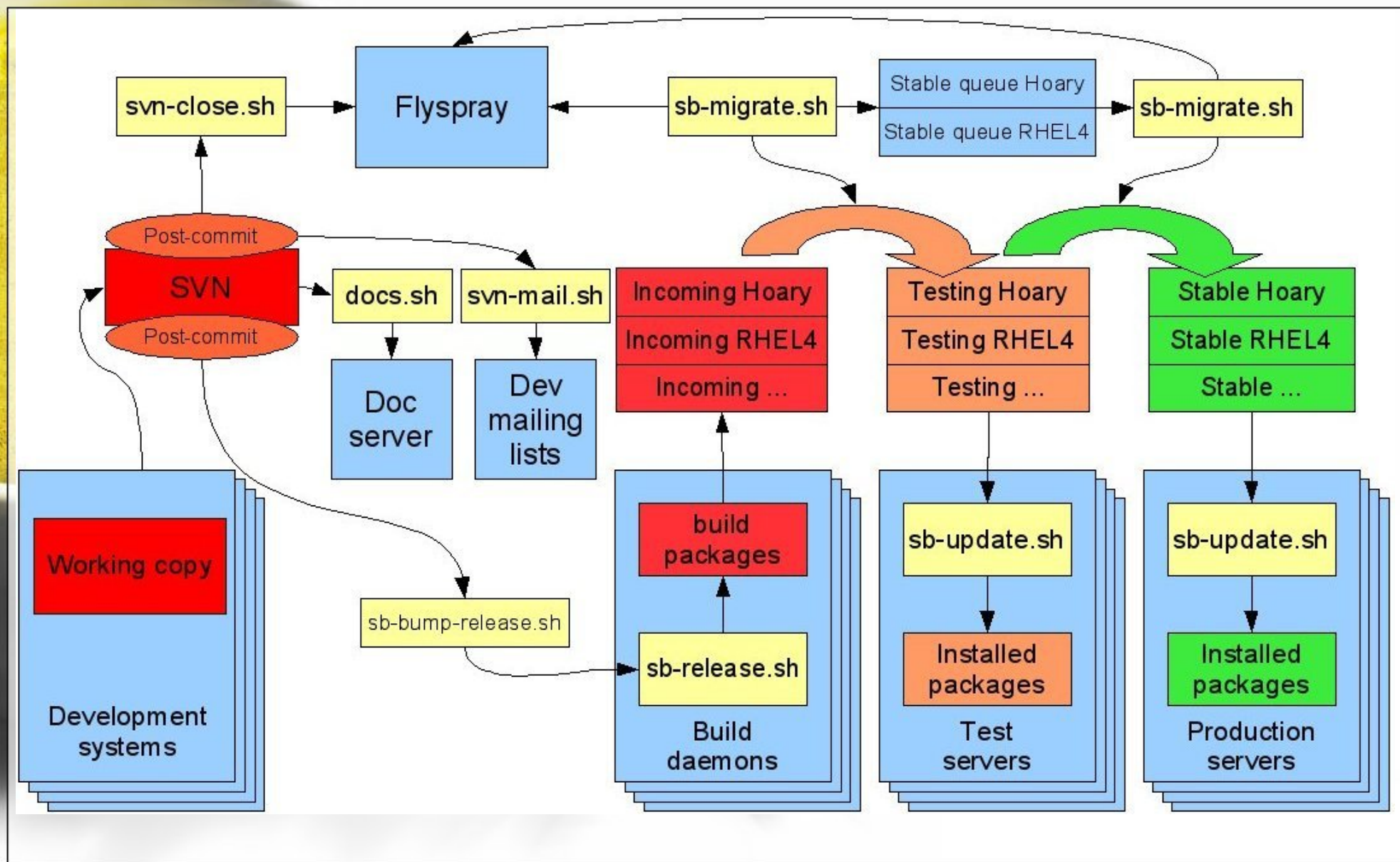
5: Pre-Release Testing



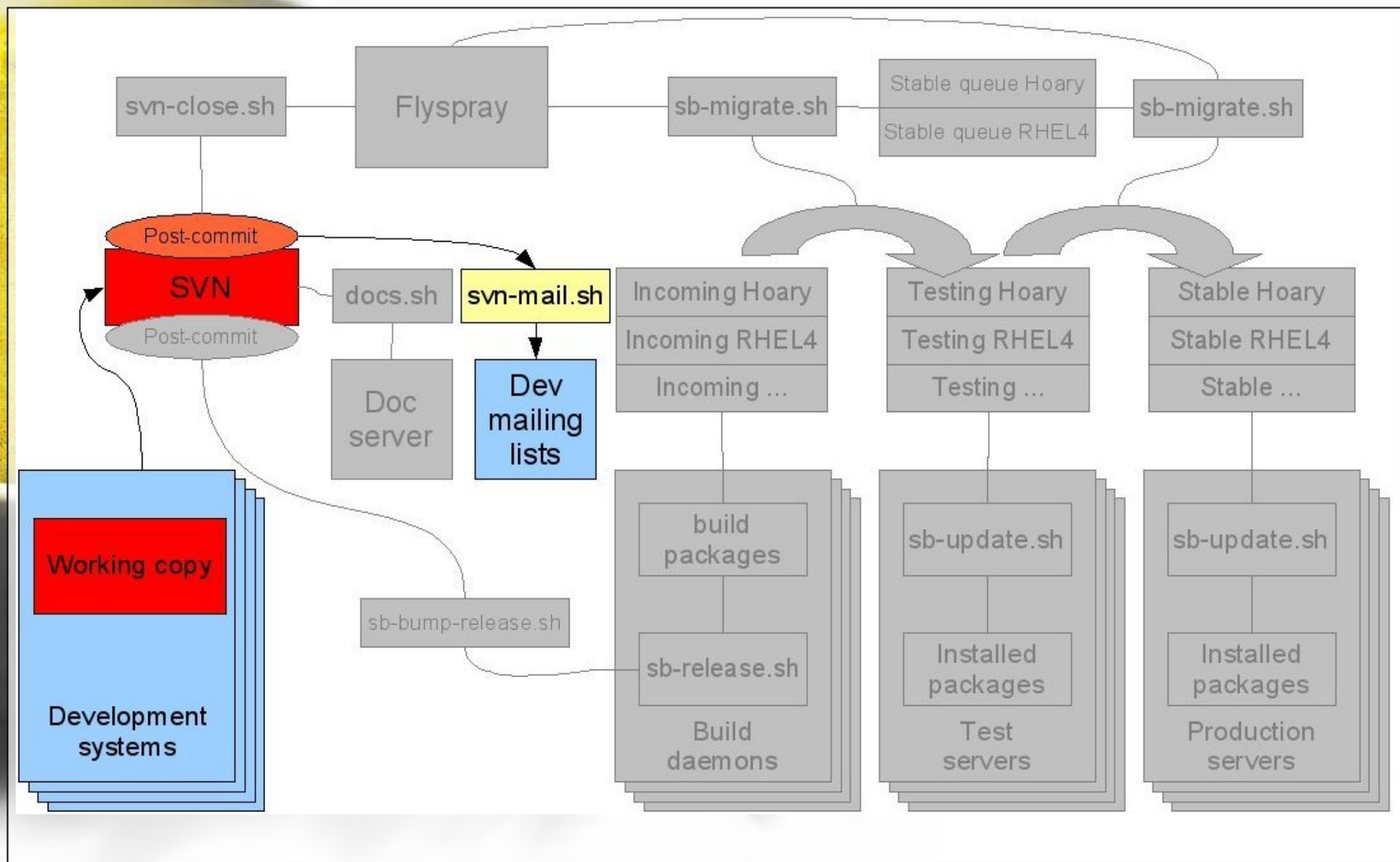
6: Package Release



Release Management Automation



1: Source Code Management



1: Source Code Management

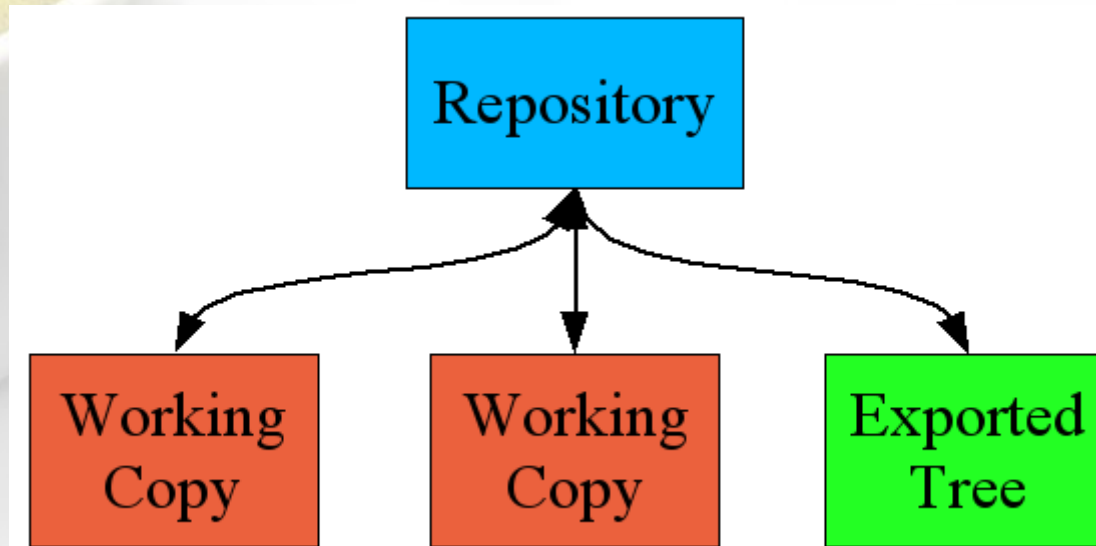
Source Code Management systems (SCMs) rock. Definitely the single most useful tool for a development team, ranking second only to a good text editor (just!).

- ♦ Multiple developers collaborating on a codebase
- ♦ Track changes
- ♦ Merge changes automatically when possible
- ♦ Assist with manual merging when necessary

Well known SCMs include CVS, Subversion, ARCH, BitKeeper, Mercurial, Bazaar, and Visual SourceSafe.

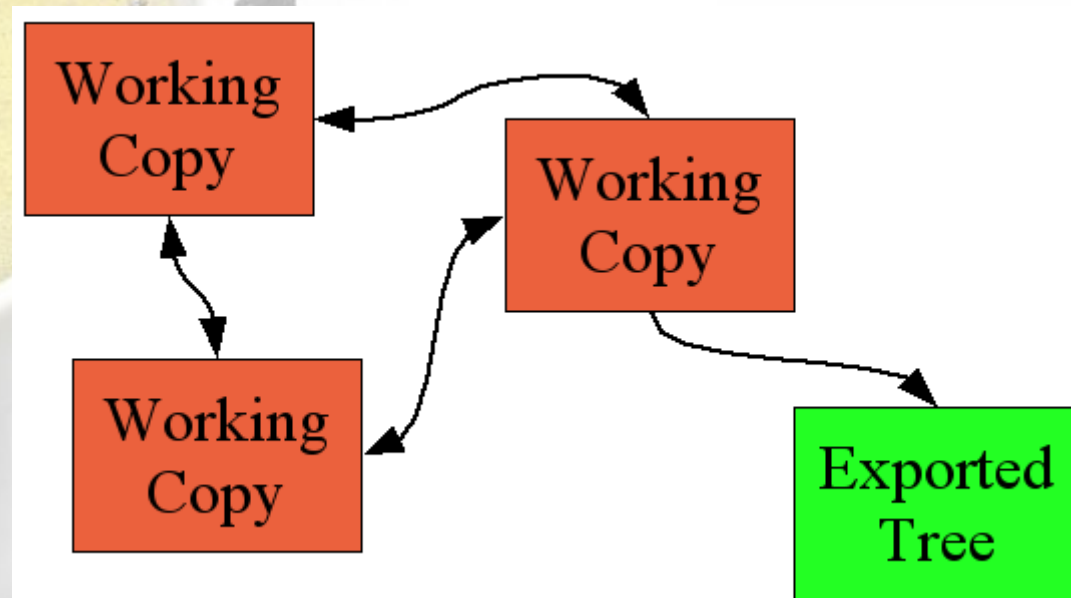
SCMs: Centralised vs Decentralised

Centralised systems use a master “repository” with all developers working on a checked-out “working copy”. All communication is between the repository and the working copies:



SCMs: Centralised vs Decentralised

Decentralised systems have no master repository, being structured more like a peering network of working copies:



SCMs: Changeset vs Patch Oriented

Changeset oriented SCMs work in terms of “revisions” or “versions” of the source tree or files, and changes are sequential.

Patch oriented SCMs work in terms of discrete “patches” which are applied to the codebase. Developers can choose to apply certain patches and discard others.

Patch-oriented SCMs are a relatively recent development and less developers are familiar with them than with changeset SCMs.

Which SCM To Choose?

Let's not start a religious war!

Developers often have strong preferences. CVS is very widely used and many developers are familiar with it but it's quite dated.

Subversion is rapidly taking over as the “default” SCM: it's a “work-alike” re-write of CVS, so many developers already have the workflow wired into their hind-brain.

Distributed systems are very interesting if you're willing to go to just a bit more effort to get your brain around them.

For an overview of SCMs see better-scm.berlios.de

Subversion Basics

Subversion comes in two parts:

1. The server, which consists of both the server-proper and a bunch of original and third party admin tools for managing repositories. The server can be configured to allow various access methods such as WebDAV, SSH and local filesystem.
2. Client tools, which allow you to connect to a server and manage a local working copy. Subversion provides powerful CLI tools and a number of third parties provide GUI tools for point-n-click simplicity.

Server Setup

1. Install using a distro package if you can: building from source isn't too hard but it uses a bunch of libraries so save yourself the versioning headaches if possible.

2. Create yourself a test repository somewhere:

```
svnadmin create /var/lib/svn/oscon
```

3. Set write privileges for your Apache2 user:

```
chown -R www-data /var/lib/svn/oscon
```

Server Setup

4. Add an entry to your “apache2.conf” so it knows how to reference your repository:

```
<Location /oscon>  
  DAV svn  
  SVNPath /var/lib/svn/oscon  
  AuthType Basic  
  AuthName "OSCON SVN Repo"  
  AuthUserFile /etc/subversion/oscon-passwords  
  Require valid-user  
</Location>
```

5. Create a password file and add a user:

```
htpasswd2 -c /etc/subversion/oscon-passwords testuser
```

Test Your Server

Your repo will now be accessible by WebDAV through your Apache2 server, so point a browser at it:

<http://localhost/oscon>

Hopefully everything will be working, but if not the things to check are:

- Did Apache2 restart cleanly? Check error log for DAV errors
- Check permissions on the repo directory
- Check authentication setup (user listed in password file?)

Optional: Install WebSVN

WebSVN is a great tool for browsing repositories. It gives you access to a bunch of meta information you wouldn't normally see through the web interface:

websvn.tigris.org

(or “apt-get install websvn” for the enlightened among us!)

Finally The Good Stuff! Adding Files

Import your source tree into the repo. Technically you can import pretty much anything you like, but for a software project set up a structure like this:

```
project/branches/  
project/tags/  
project/trunk/
```

Stick your source tree in 'trunk'.

You could just put your project files at the top level but that will give you grief down the track.

Import Your Project Tree

Import your project tree into the repo:

```
svn import project http://localhost/oscon
```

Now your files are in the repo. Point a browser at it or use WebSVN to see for yourself.

Check Out A Working Copy

Your original project tree is untouched and pretty useless because it's not being managed by subversion: **don't** make changes in it.

Instead you need to “check out” a local working copy and start working in that.

```
svn checkout http://localhost/oscon/trunk project-wc
```

You'll then have a working copy called “project-wc”, identical to your original project tree but managed by Subversion. Changes you make to the working copy will be tracked.

Working The Working Copy

Basic commands to remember are:

```
svn checkout (co) <repos_path> <local_dir>
```

```
svn update (up)
```

```
svn status (st)
```

```
svn commit (ci) (-m "commit message")
```

```
svn diff
```

```
svn revert
```

```
svn add
```

```
svn mv
```

```
svn rm
```

```
svn mkdir
```

Basic Workflow

99% of the time the workflow is very simple:

- Make changes to your working copy, then “svn up” to make sure you're up to date with changes made by other developers and merge them into your working copy.
- Use “svn status” to see the status of the files in your working copy.
- Commit your changes back up to the repo with “svn commit”.

The other 1% is when you have to deal with conflicts.

Conflict Resolution

Sometimes changes “conflict”: Subversion can't figure out how to merge them, so it asks for help.

If “svn status” shows files with a “C” status, you need to fix them. Look for conflict markers:

```
<<<<<<< .mine  
>>>>>>> .r23
```

Edit the file until you're happy with the result, then:

```
svn resolved filename.pl
```

Finally, commit to the repo.

Meta Data: Object Properties

Subversion lets you store arbitrary meta-data with objects, including directories:

```
svn propset <propertypname> 'property value' file.c
svn propedit <propertypname> file.c
svn proplist file.c
svn proplist --verbose file.c
```

Branching, Tagging And Merging

Sometimes it's not enough to just have one tree. Typically you'll need at least two: a stable tree and a development tree.

Subversion executes branching in a simple way: it has no special “branch” mechanism at all.

Use the “copy” command to snapshot any arbitrary part of the tree or label it with a tag for posterity.

Then use “svn merge” to migrate changes between your branches and/or trunk.

A “tag” is just another branch that nobody commits to.

Backups: Dump And Restore

Subversion uses a BDB or filesystem datastore, and provides tools to dump the contents (including full history) in a portable way:

```
svnadmin dump /var/lib/svn/oscon > mybackup
```

Likewise a restore:

```
svnadmin load /var/lib/svn/oscon < mybackup
```

Perfect for backups, migrating repos, and handling major upgrades to Subversion.

But watch for repo ownership.

Exporting The Tree

Sometimes you just want to output a snapshot. Don't leak working copies out to the world because they contain a bunch of hidden meta-data.

Use “svn export” just like “svn checkout”:

```
svn export http://localhost/oscon myexport
```

Use exports for release packaging.

Automate Your World: Hook Scripts

Subversion looks for specific scripts at different times.

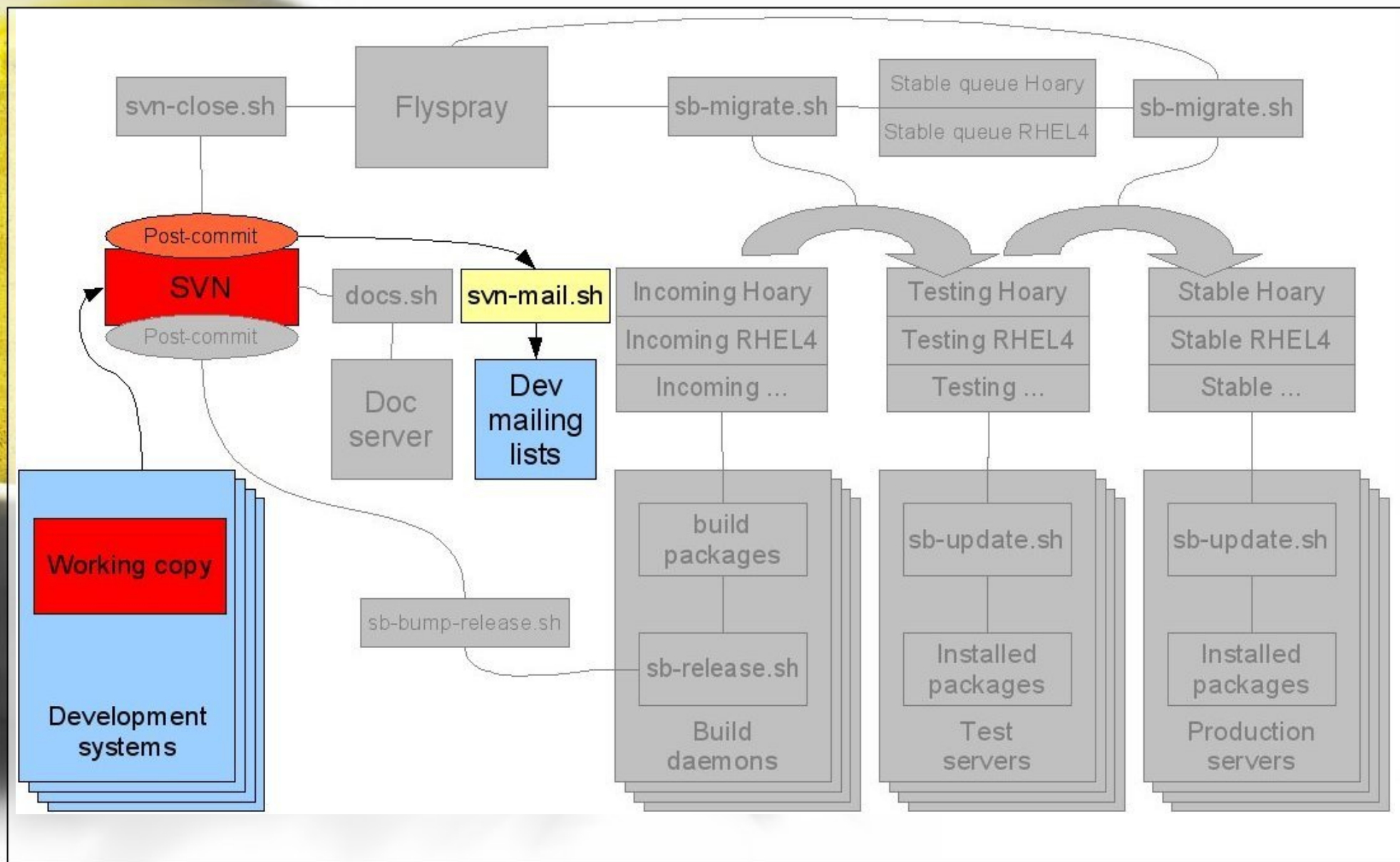
- **start-commit:** Run before a transaction even begins, and doesn't have access to much information.
- **pre-commit:** Run after a transaction has been done but before it's committed, at which point it's possible to examine what the transaction does and make decisions.
- **post-commit:** Run straight after a transaction is committed.
- **pre-revprop-change / post-revprop-change:** Used for managing versioned properties.

Changelog Emails

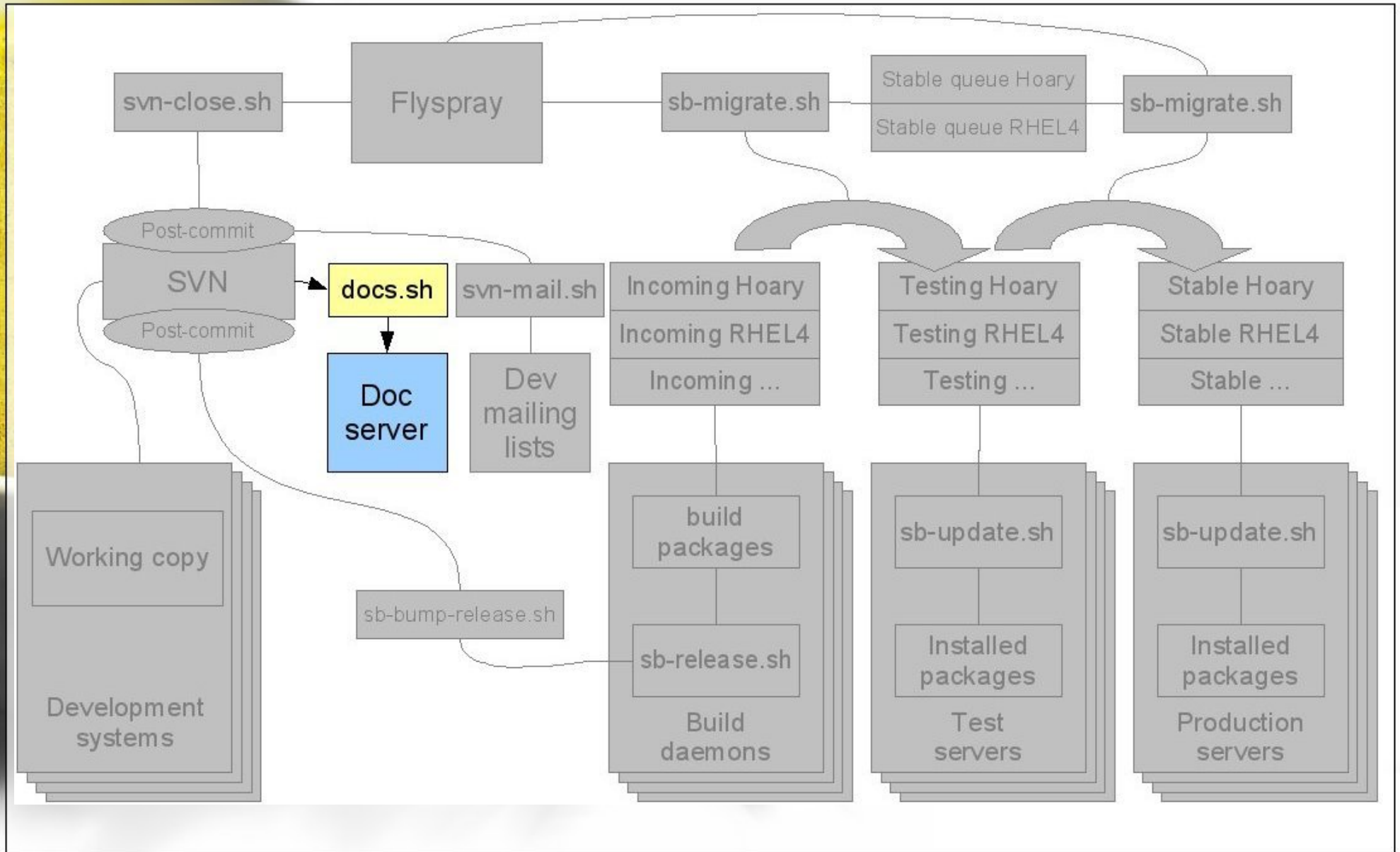
Use the “post-commit” hook to send an email to your devteam mailing list whenever a commit is made.

- Name of developer
- Files that were altered
- Text of the changelog entry
- Diffs of changes

1: Source Code Management



2: Internal Documentation



Autogenerated Documentation

“Documentation” is a four letter word to most developers.

So generate internal docs automatically!

Useful:

- As a memory refresher for yourself and your team-mates.
- When bringing new developers onto a team.

Auto-generated Docs

Be lazy like a fox: let Doxygen build docs for you.

In your source tree:

```
doxygen -g doxygen.conf
```

Then edit it to set:

```
PROJECT_NAME = YourProject  
OUTPUT_PATH = /var/www/doxygen  
INPUT = /path/to/source/tree  
RECURSIVE = YES
```

Finally:

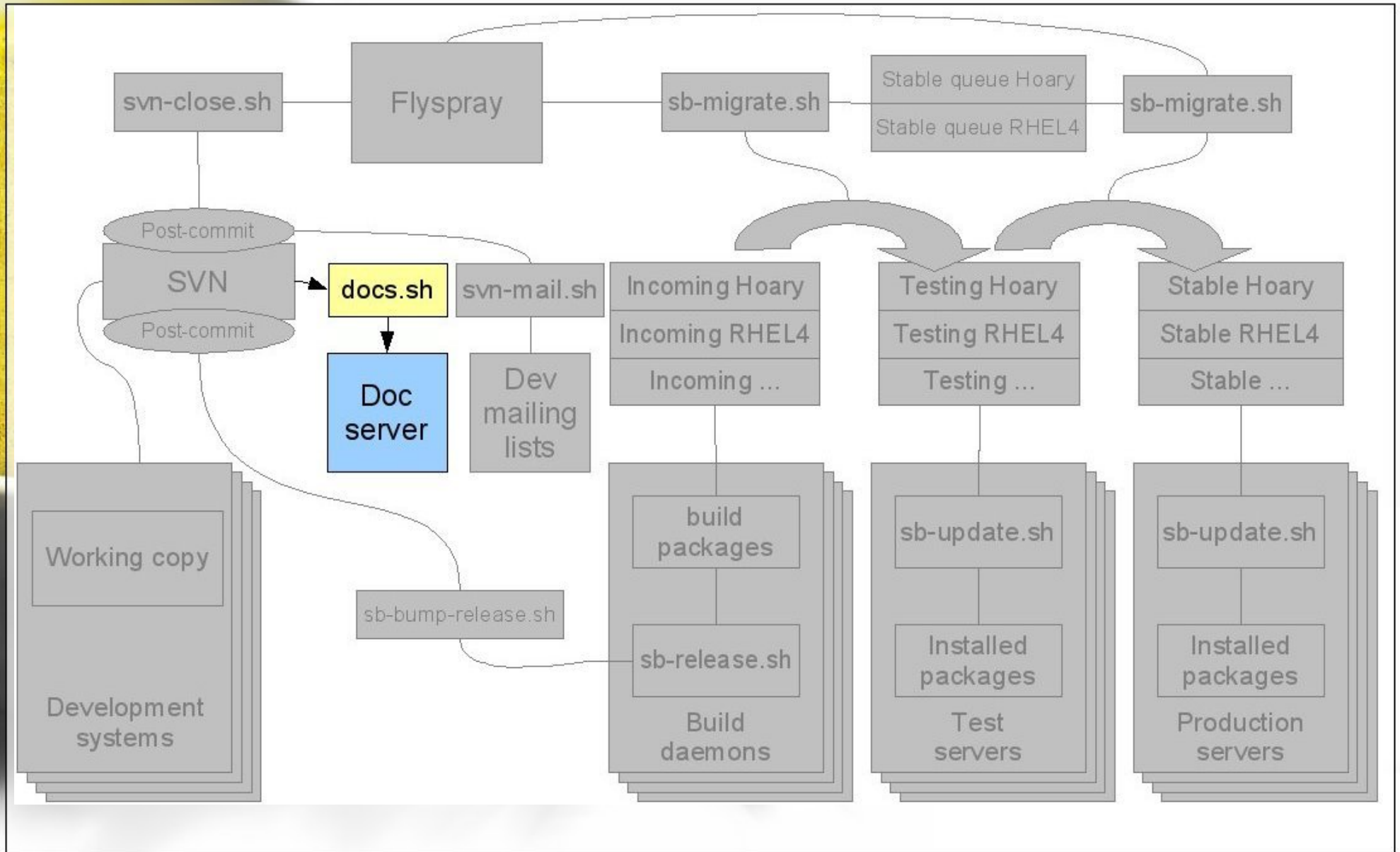
```
doxygen doxygen.conf
```

Docblock Comment Format

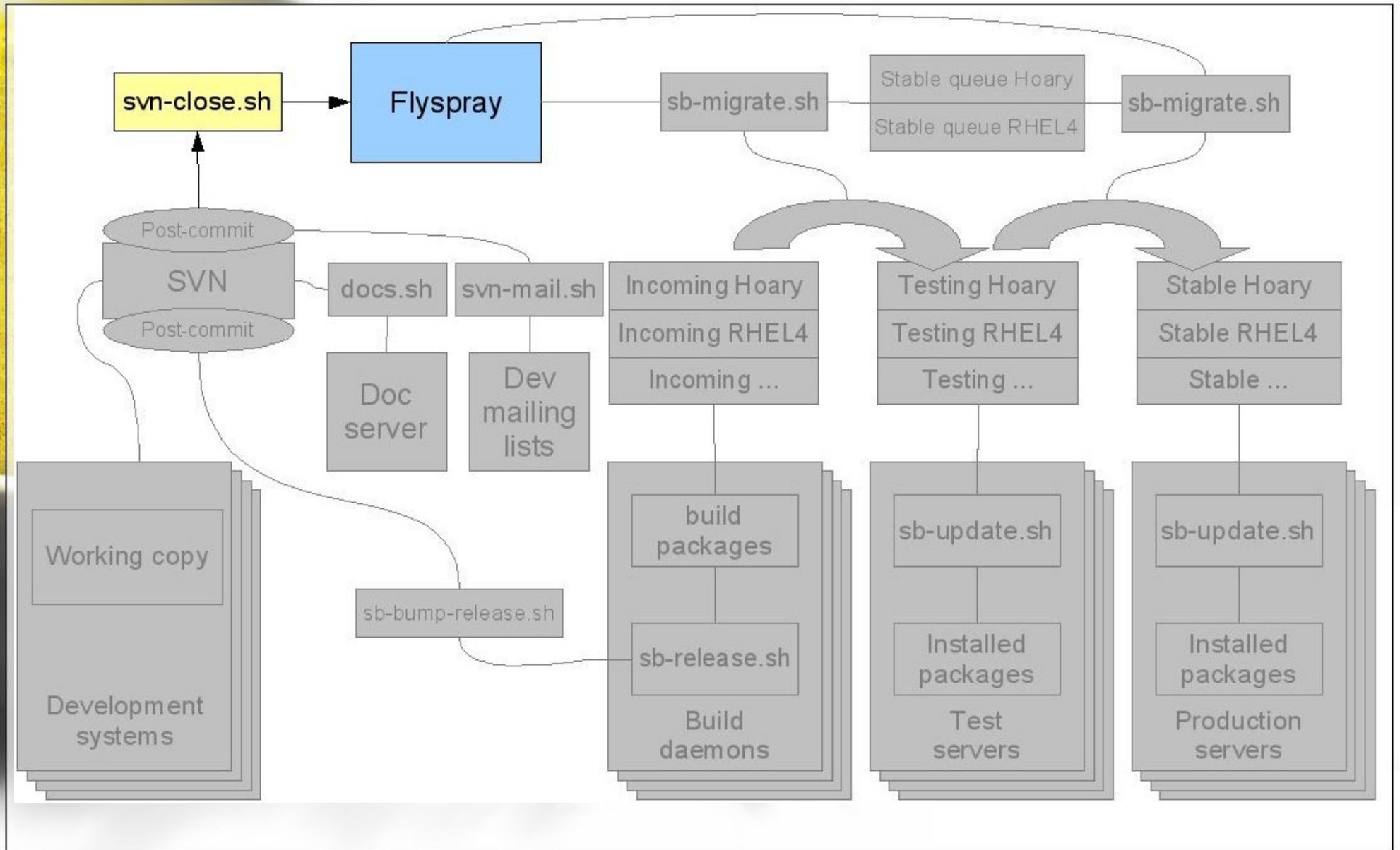
Doxygen and various PHPDoc/Javadoc implementations parse comments in your code. Put a “DocBlock” before every class, method, and function:

```
/**
 * This is my groovy function to multiply two numbers
 * @param integer $width The width of the wall
 * @param integer $height The height of the wall
 * @return integer $area The total area of the wall
 */
function find_area($width, $height)
{
    $area = $width * $height;
    return $area;
}
```

2: Internal Documentation



3: Bug Tracking



Bug Tracking Systems Rock

A Bug Tracking System (BTS) can help you assign, prioritise and report on tasks and bugs.

A BTS is really just a To Do list on steroids but it can dramatically simplify the process of managing your projects, and also provide an invaluable central location for visible collaboration.

Bug Tracking Systems Suck

A BTS is just a funky To Do list, right? So they're simple, right? And of course that means they're easy to install?

Think again.

You don't truly appreciate the word “frustration” until you've tried to deploy Debbugs or Bugzilla.

Features To Look For

Different people and groups have different requirements, so consider how you prefer to work.

- Web interface
- Email interface
- External hooks
- Access control: users and groups
- Categories and item types
- Reports
- Comments
- Dependencies
- Attachments

Shameless Plug: Flyspray

Flyspray is a BTS written in PHP and using either MySQL or Postgresql as a back-end.

It's designed for ease of installation and simplicity for first-time users.

<http://flyspray.rocks.cc>

For more information see *Killing Bugs With Flyspray*:

<http://jon.oxer.com.au/flyspray>

Bug Closures On Commit

When developers commit new code to the repo it often fixes a bug, so then the bug itself has to be closed or tagged in the BTS.

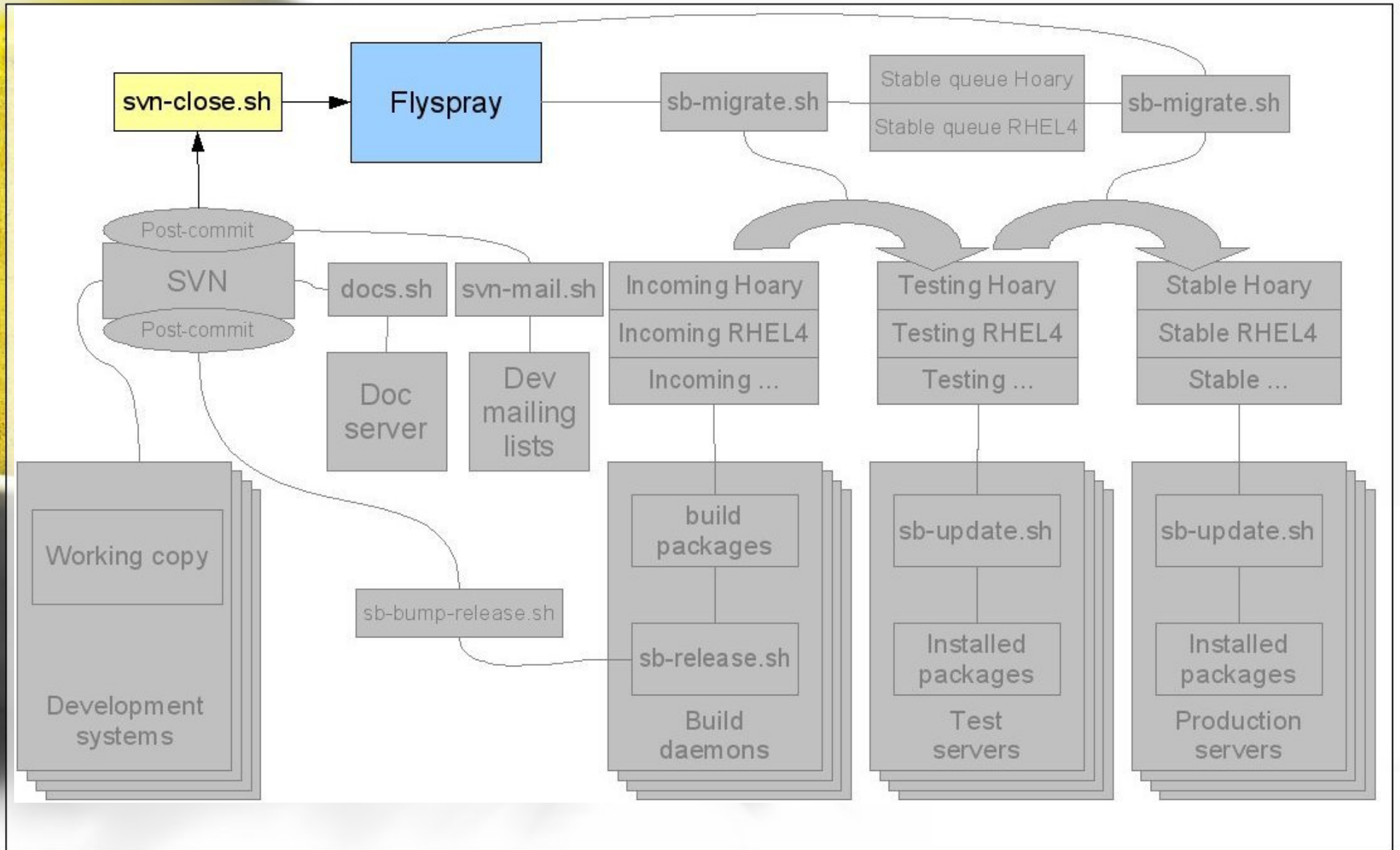
Why make developers do extra work? Close it automatically!

Use a post-commit hook that scans the changelog for entries like:

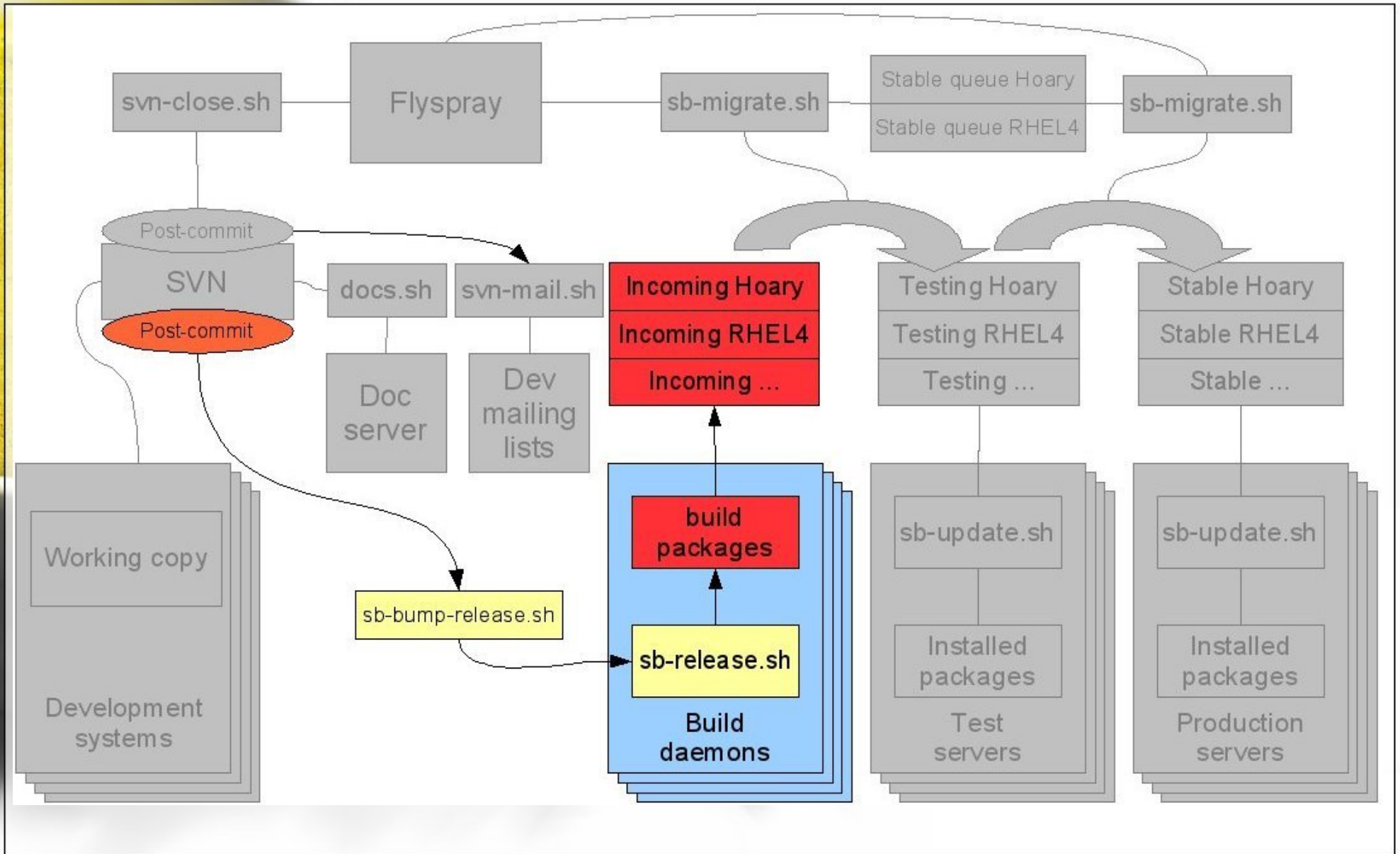
Closes: #123

then connects to your BTS and marks that bug as closed and appends the changelog entry as a comment.

3: Bug Tracking



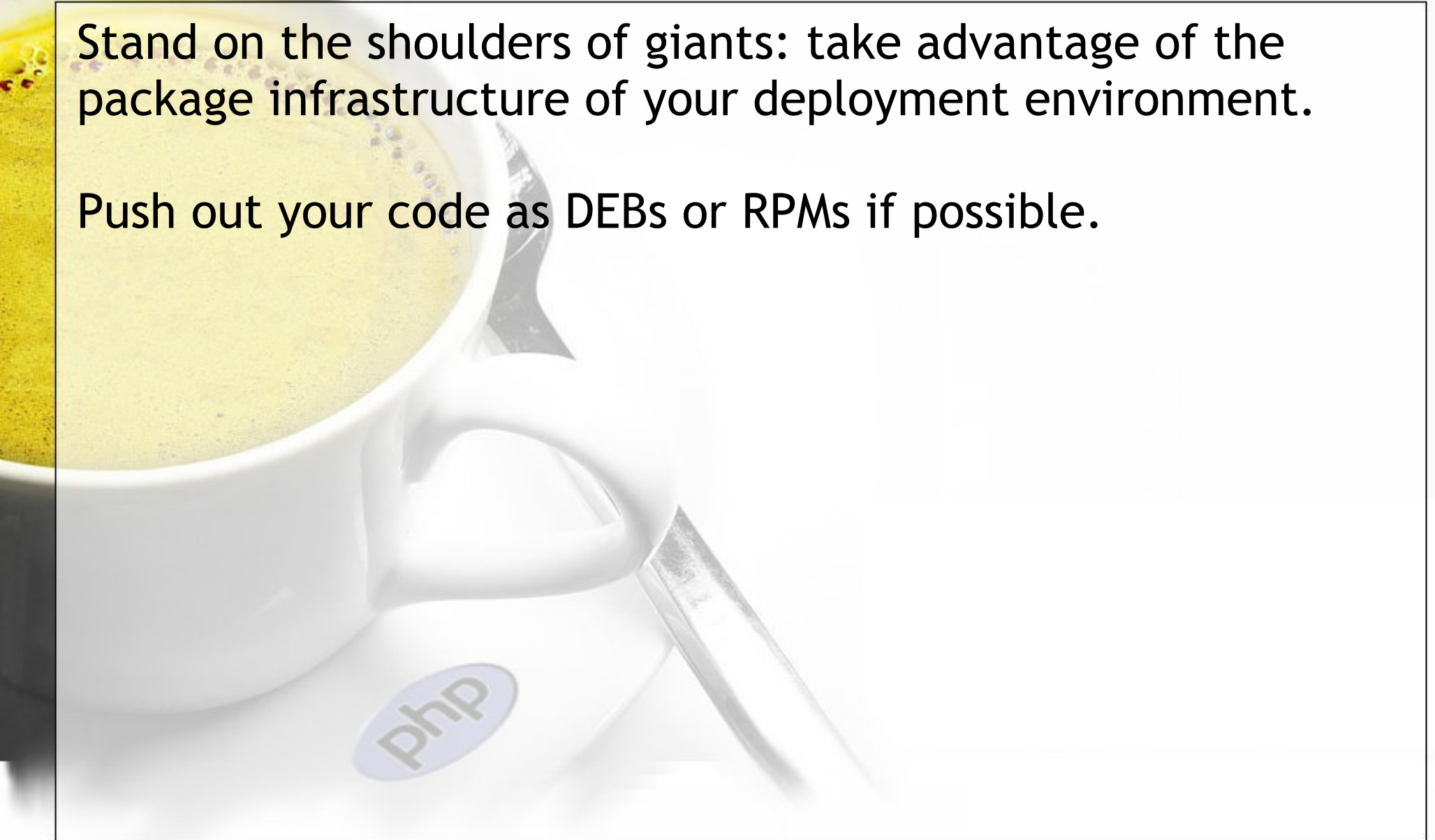
4: Packaging



Use Distro Packaging

Stand on the shoulders of giants: take advantage of the package infrastructure of your deployment environment.

Push out your code as DEBs or RPMs if possible.



Auto Distro Packaging

Trigger build scripts with a post-commit hook.

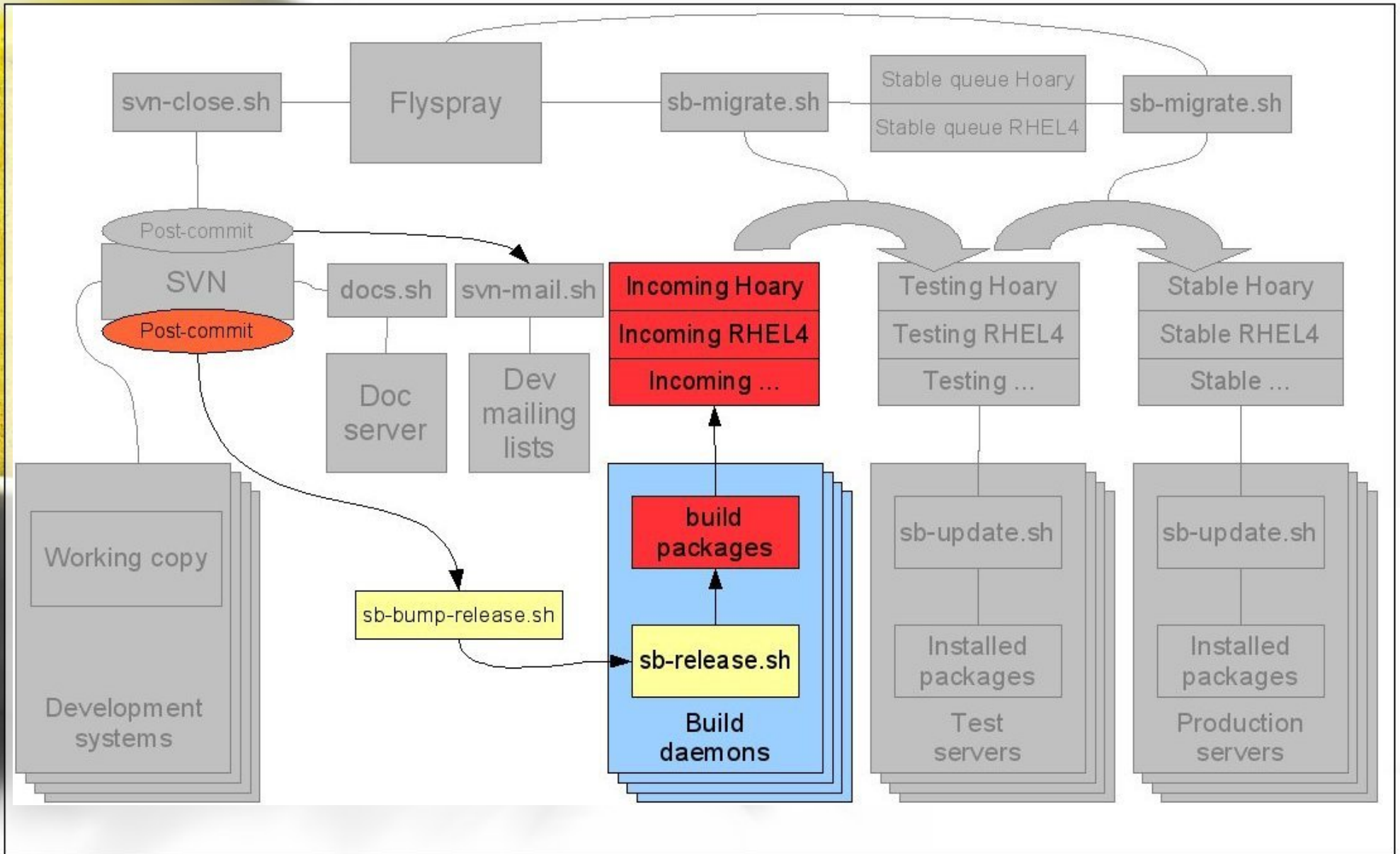
Look for:

- Entry in commit message of “release”
- A change to the changelog

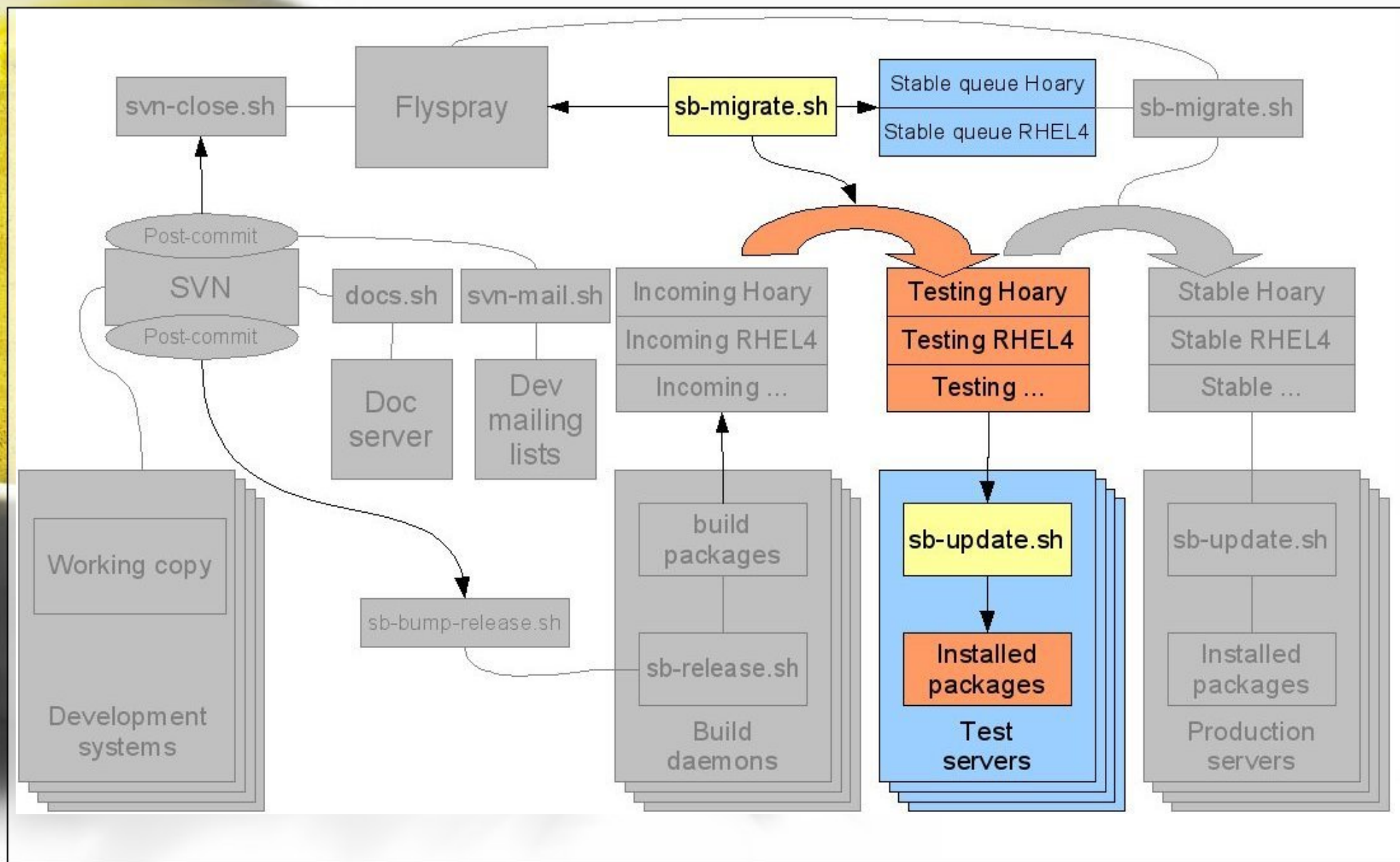
Distro-Specific Packaging

- Encode or pre-compile your source tree (optional)
- Build a package containing the end result
- Sanity-check with distro packaging tools
- Add to a software repository

4: Packaging



5: Pre-Release Testing



Deploy On Test Servers

Push to a test server for each target deployment architecture:

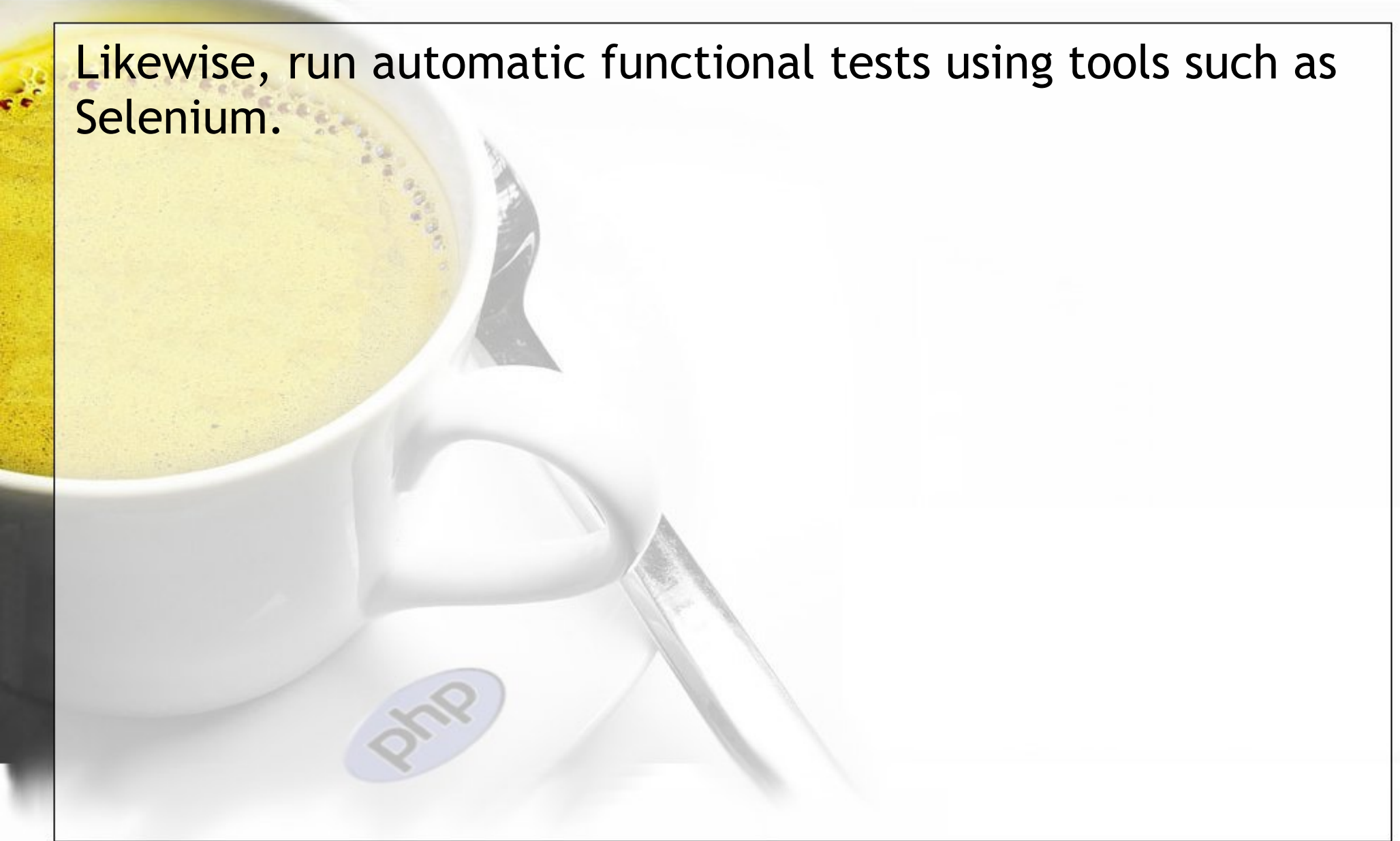
- Migrate package to “testing” software archive
- Have test servers auto-install the latest packages
- Tell someone who cares: add a “QA Test” task to the BTS

Automatic Unit Testing

- Run unit tests automatically when test machines update.
- Email test failure to your devteam or the developer who committed the change that caused the breakage.
- Append test results as a comment to the BTS entry.

Automatic Functional Testing

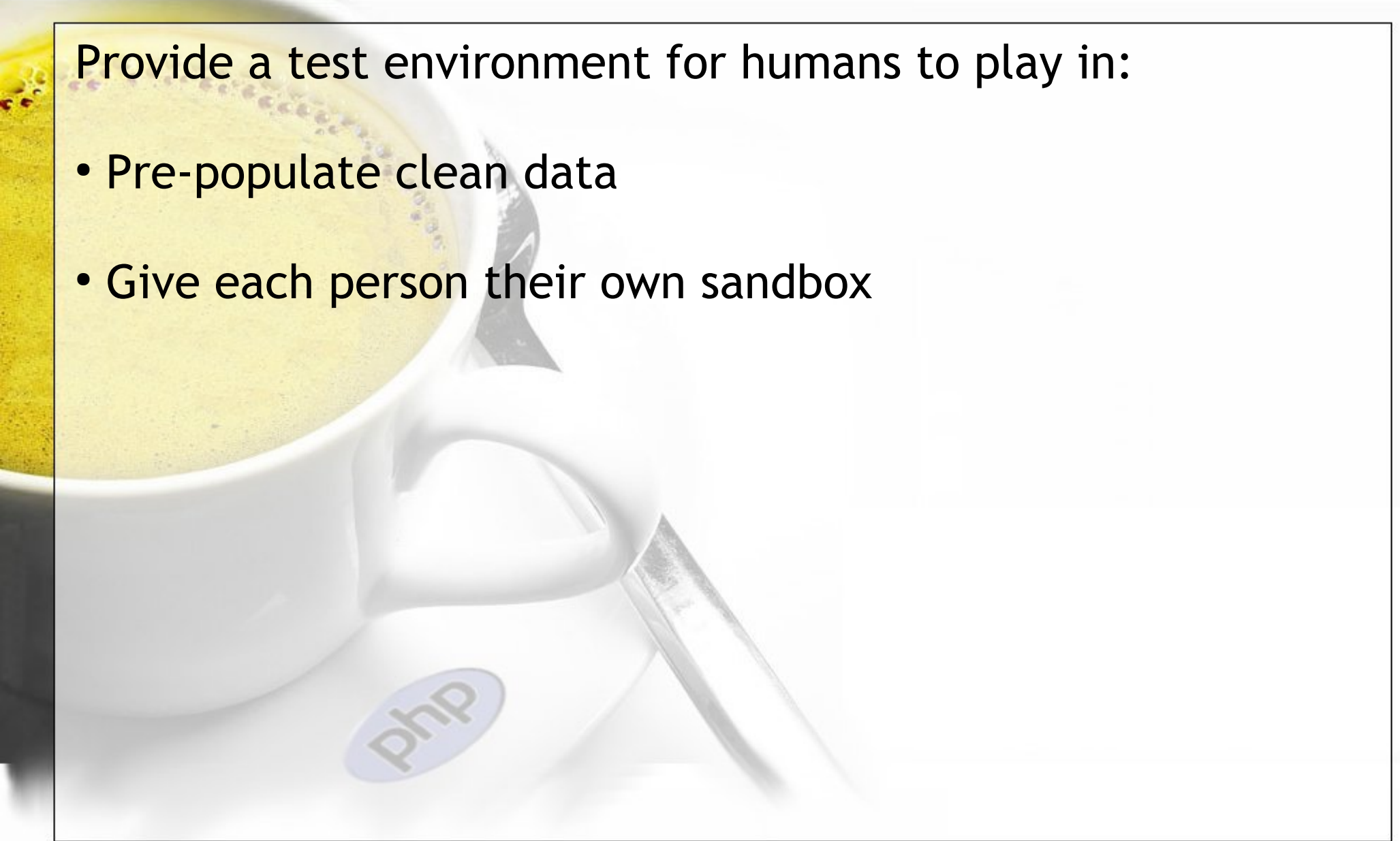
Likewise, run automatic functional tests using tools such as Selenium.



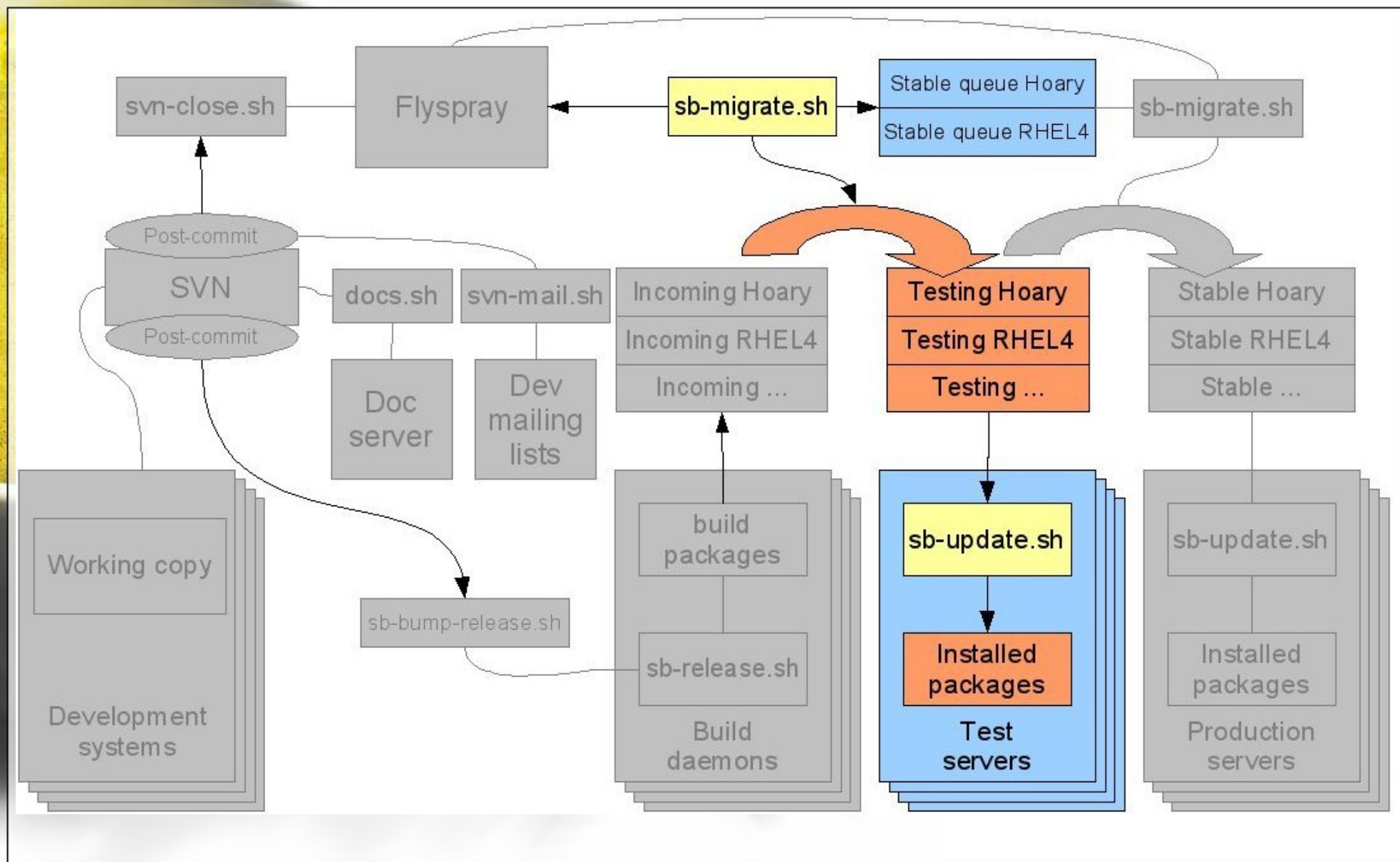
Manual Testing

Provide a test environment for humans to play in:

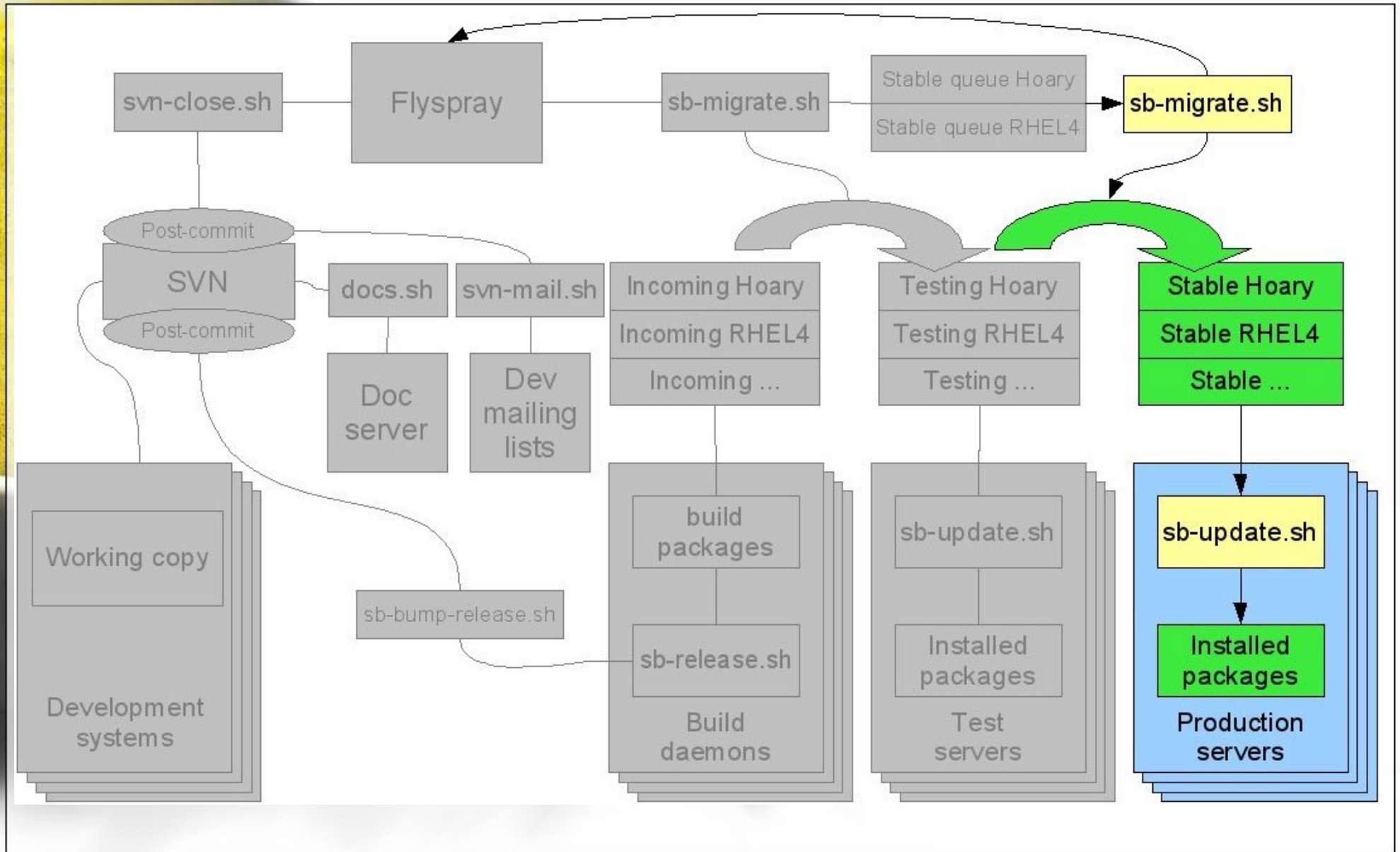
- Pre-populate clean data
- Give each person their own sandbox



5: Pre-Release Testing



6: Package Release

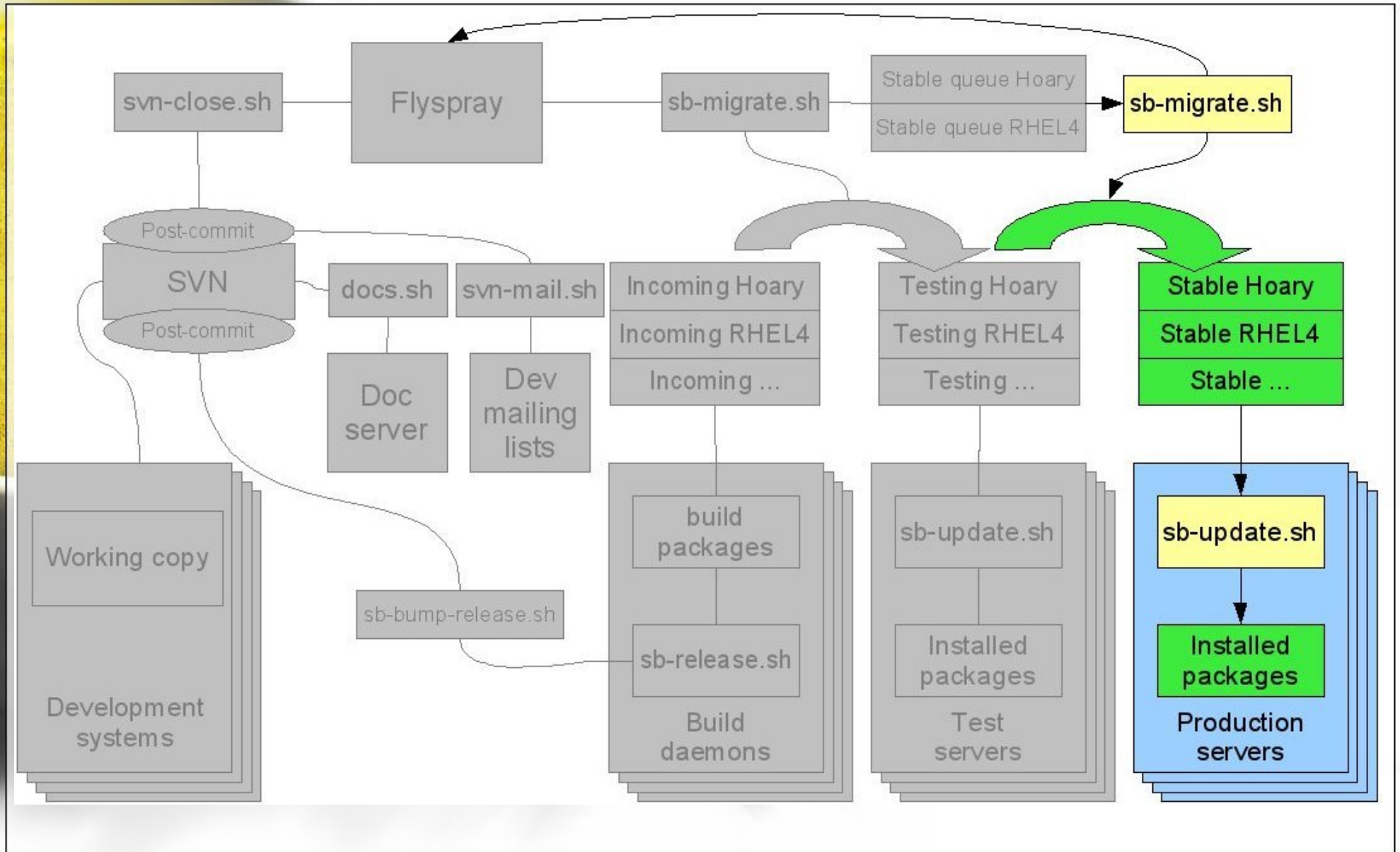


Release Approval

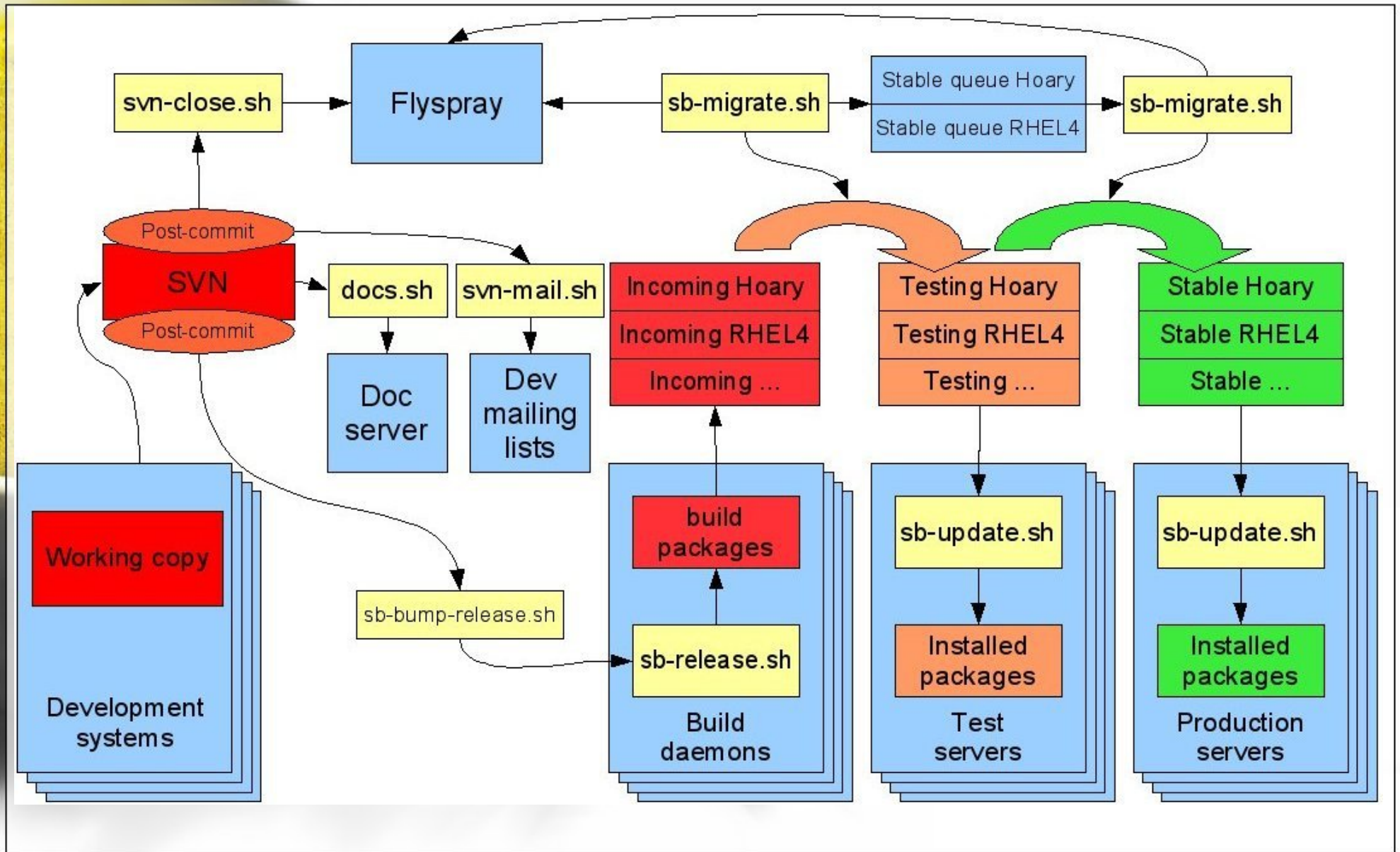
Require testers to close “QA Test” tasks to approve package release, then automatically:

- Migrate packages to “stable” package archive
- Have production servers auto-update from stable archive

6: Package Release



That Scary Architecture Diagram Again



Things I Glossed Over

Of course working as part of a productive team has many more elements to it than I have mentioned so far:

- Coding standards
- Application architecture
- A development methodology
- Good project specs
- A good team leader to take client heat
- Table tennis at lunch time

More Information



These slides: jon.oxer.com.au/talks

Subversion: subversion.tigris.org

“The SVN Book”: svnbook.red-bean.com

Flyspray: flyspray.rocks.cc

Comments? jon@oxer.com.au