Open source development best practices

Ibrahim Haddad, Ph.D.
VP Strategic Programs, Linux Foundation
Executive Director, LF AI Foundation

This deck was contributed by Ibrahim Haddad to LF Energy and is licensed under CC BY 4.0.


@IbrahimAtLinux


linkedin.com/in/ibrahimhaddad/


github.com/ibrahimhaddad


IbrahimAtLinux.com


Ibrahim@Linux.com

LF ENERGY

Description

This presentation focuses on delivering open source development best practices that developers in your organization should follow to ensure successful and fruitful participation in any open source project. It provides a pragmatic view of what other successful companies participating in open development do, with extensive best practices and examples drawn from various open source development projects.

Outline of discussion areas

- Requirements
- Design
- Implementation
- Test / Integration
- Maintenance

Best Practices: Requirements

Understand the project fundamentals

- Follow the project discussions (mailing lists, IRC, Slack, etc) to understand ongoing efforts
  - Who are core developers?
  - What are areas of high priorities?
  - Release cycles
  - Major bugs
  - Who's working on what?
  - Where help is needed?
  - Etc.
- Experiment with latest stable and experimental releases
- Determine how your interests and priorities align with those of the project
- Find others who share your interests and priorities, and work with them

Propose new features, signal intent

- Public and transparent discussions are a prerequisite
  - Ensures maintainers are aware of the need and the problems you are working to solve
  - Recruit others to help do the work
  - Gather feedback on the usefulness and design considerations before doing a lot of work (and being rejected)
  - Some projects have active mailing lists, multiple tries may be needed

- Over-communicate
  - Assume you have one chance to convince others

Propose new features, signal intent

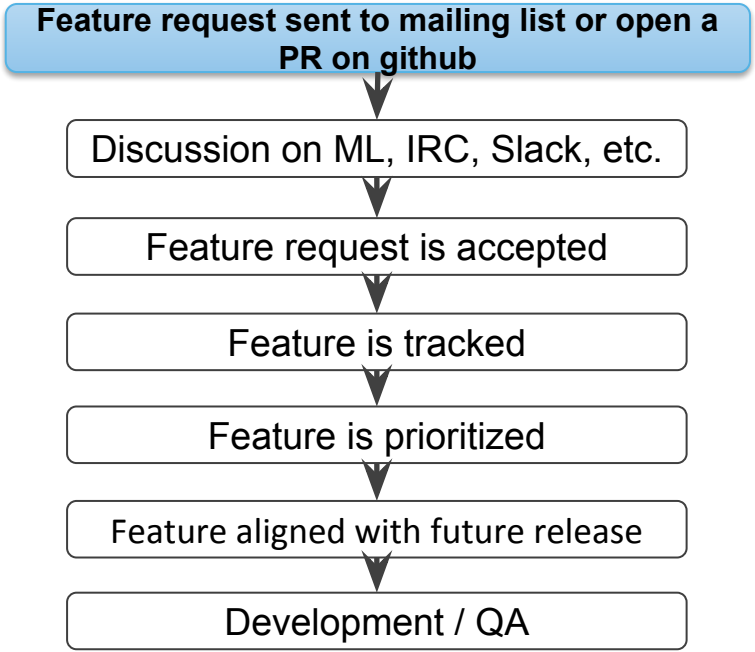
- Accept incoming feedback and work with it
  - Incorporating feedback increases the likelihood a contribution will be accepted
  - Feedback is generally legitimate, as others will only take time to respond if what you're doing is important to them

Getting buy-in for a feature request

Contributed features should be:

- Useful to others and not just to your specific usage models: Features that benefit only a few users will tend to be rejected if there is no benefit to the majority
- Implemented in small parts, and delivered in a way that provides immediate benefit
- Strong on security: Open source developers tend to be more security conscientious than closed source developers
- Backed by resources ready to implement and maintain

Typical feature request process

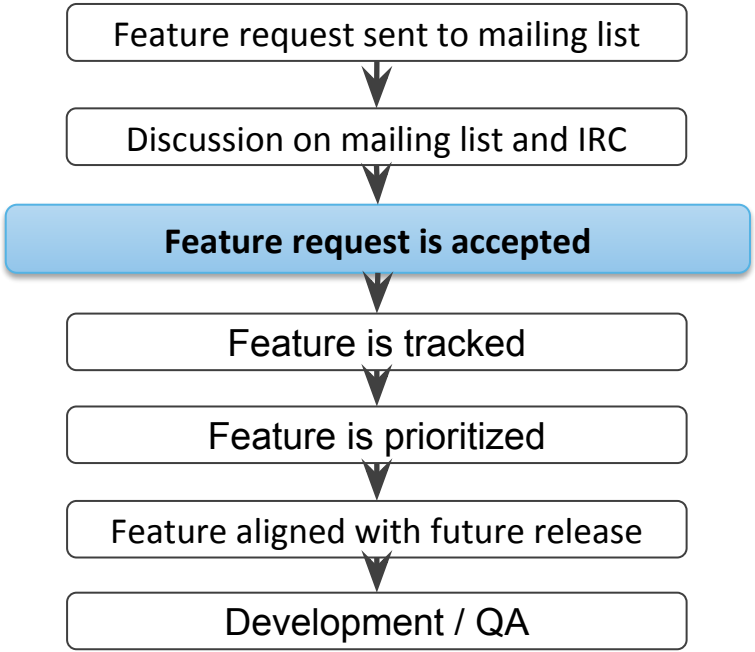


- What happens in this phase:
  - Requester's opportunity to make the case for why the project should plan for their feature.
  - In most cases, the requester already has enough resources to implement the feature.
- Typical actions:
  - Requester/developer sends email to a project mailing list
    - The need for the feature
    - The problem it solves
    - How it fits into the project
    - Possible implementations

Typical feature request process

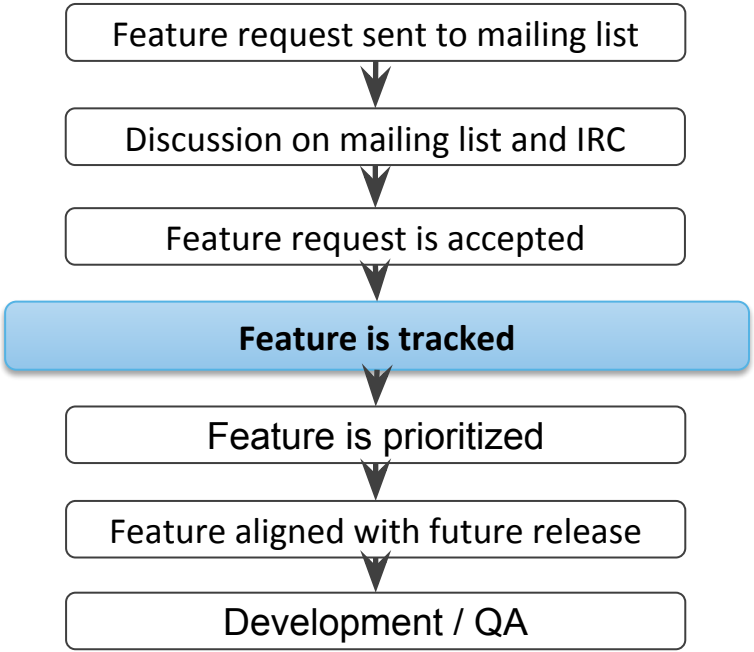
- What happens in this phase:
  - Interested parties are given an opportunity to comment
  - Major or intrusive feature requests will typically be discussed in great detail
  - Other Opportunity to recruit other development resources to help implement
- Typical actions:
  - Initial email stimulates discussion on the mailing list
  - Maintainer may comment on when the project would be ready to accept feature
  - Comments range from brief approval, to detailed implementation considerations

Typical feature request process



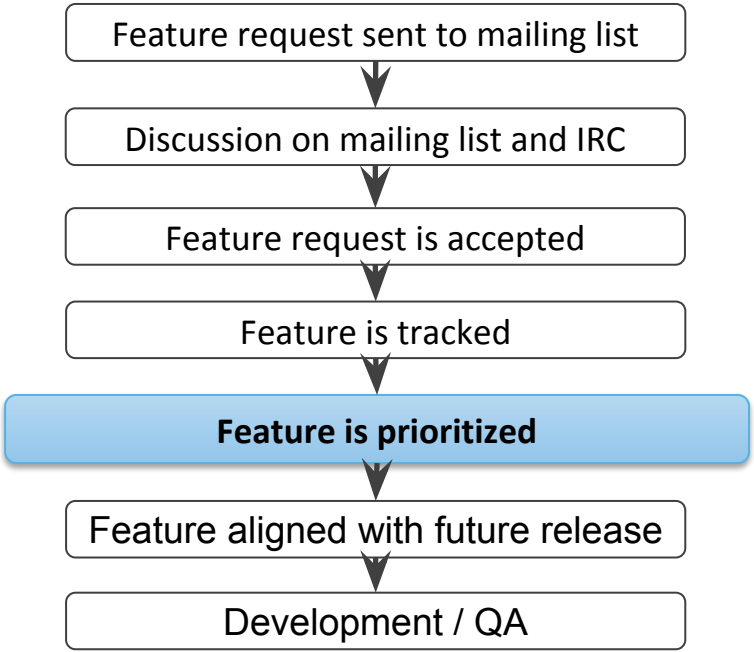
- What happens in this phase:
  - Maintainer and other interested parties agree
    - Feature is strategic to project
    - Belongs on roadmap
    - Implementation is ok
- Typical actions:
  - General consensus that feature will be beneficial to project
  - Acknowledgement that feature will not impact
    - Stability
    - Security
    - Functionality of project
  - Maintainer gives approval

Typical feature request process



- What happens in this phase:
  - Most projects or subsystems within a project have a process for tracking features in upcoming releases
  - This is often considered the authoritative record of what was proposed and accepted
- Typical actions:
  - Developer adds detailed feature information into tracking tool

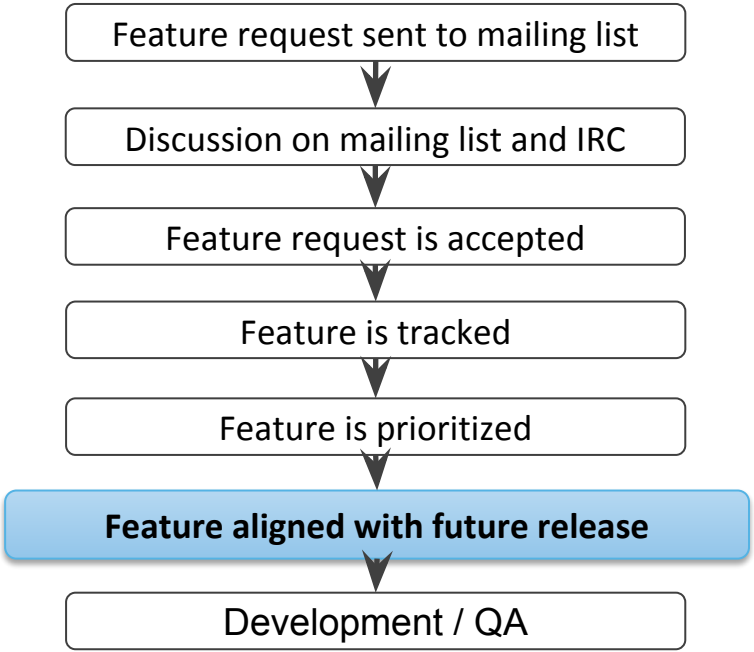
Typical feature request process



- What happens in this phase:
  - Maintainers prioritize features
    - What is core to the project
    - What is needed, and when
  - Maintainers generally have a global view of dependencies and future deliverables, and may prioritize to align features.
  - Minor features and enhancements may be requested as soon as they are ready.

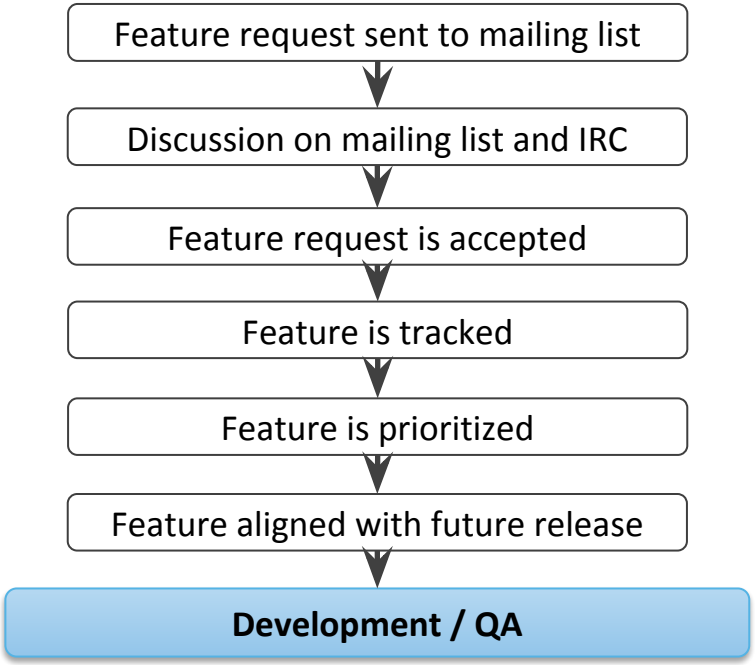
- Typical actions:
  - Maintainer may determine the priority of the request relative to other queued features

Typical feature request process



- What happens in this phase:
  - Project maintainer communicates when they will be ready to integrate the feature.
  - This helps them plan for a manageable number of features per release.
- Typical actions:
  - Maintainer may assign feature to a specific release.

Typical feature request process



- What happens in this phase:
  - Developer and any other resources recruited in former steps begin implementing the feature, targeting the release set by the maintainer.
- Typical actions:
  - Development team begins work.

Mailing list etiquette

- Keep on topic
  - Try to stick to one topic per email, and make the subjects descriptive
  - Keep subject lines intact, as archives and mail clients use them for threading
  - If the topic changes, start a new thread
- Don't use attachments
  - If you want to email code or documents, copy/paste in the email message or provide a link to a web site
- Don't use HTML email
  - It may confuse some mail clients, and cause formatting problems
- Reply and forward messages “inline”

Mailing list etiquette

- Posts with CAPITAL LETTERS will be ignored

- The way you communicate matters
  - You will never lose points for professionalism
  - Many participants are international, so be as clear and straightforward as possible
  - Avoid sarcasm, don't take things personally
  - Not everyone will be professional, "Don't feed the trolls"

Best Practices: Design

Design in the open

- Communicate early and often on mailing lists
- Provide examples and possibly reference implementations
- Anticipate feedback
  - Acknowledge good feedback and re-work your contribution
- Respond promptly to questions, particularly from potential contributors
  - Signal willingness to adapt your design if someone else will do the work
- Plan for modularity, even if the first designs are not

Recruit others

- Scratch your own itch, nobody else will scratch it for you… unless they have the same itch
- If you are wanting to decrease your development burden, write code that will attract others for the same reason
  - Make sure your contribution is scoped broadly enough to attract other contributors
  - Be responsive and proactive if someone indicates interest in your email to the mailing list
- Don't be surprised if it's a competitor
  - Often the companies with the most to gain from a feature in an open source project are in the same line of business
  - There is a rich history of collaboration between competitors

Design for acceptance

- Design your contribution to be written and integrated in the smallest parts possible
  - Smaller patches are easier for maintainers to integrate
  - Many open source projects favor a modular approach, because it promotes extensibility
- Scope the design, and subdivide your plans if necessary
  - Larger changes are more likely to be adopted if a series of smaller changes with concrete milestones
  - Communicate the overall plan to provide context, but don't expect universal buy-in
- Be as non-intrusive as possible to other subsystems
  - If you believe you need to change a core system component, communicate far in advance and solicit input from their maintainers before getting started

Best Practices: Implementation

Be agile

- The line between design and implementation is very blurry in open source development
  - Multiple iterations are expected and encouraged

- Don't expect perfection the first time
  - Code stabilization is part of the community process
  - Allow the developer community to help guide and shape the code

Begin on a fresh pull from the right tree

- It is not uncommon to have many versions of the same project
  - Each maintainer may have their own tree
  - Some projects maintain stable and development trees
- In almost all cases, you should base your work on the tree that builds the official release
- Ultimately the project maintainer is final arbiter on what gets accepted
  - If your patch does not apply cleanly to their tree, it will be rejected
- Never assume code as a dependency that isn't already part of mainline, unless you're submitting it yourself
  - It's impossible to test without replicating your custom setup, and maintainers usually don't have the resources to do this

Implement functionality in the smallest reasonable chunks

- Will result in more constructive feedback
  - Easier to understand small patches
- Simplified testing
  - A small change is less like to have unintended consequences
  - Very important because of lack of traditional test phase
- You will be expected to submit in small parts

Reuse existing, accepted code wherever possible

- It makes your contribution smaller
- It reduces the code you must personally debug and maintain
- It increases your chance of acceptance
  - Maintainer already familiar with accepted code
  - Most maintainers are very averse to duplicating existing functionality
- If existing code has most of the functionality you need, submit a patch to extend it
  - Always try this before building your own
  - If this is for a key dependency, get your patch accepted before beginning work in earnest on the main feature

Respect coding style

- Coding style is important as it makes code easier to read
- Code can be rejected for being hard to read
- Always review any code you are submitting to adhere to the project's specific coding style guidelines

When to start submitting code for review

- After you have written code that works well enough to address your final goal
  - It should compile and build
  - It should be complete enough that others can understand it, and provide meaningful feedback
- Before you have spent weeks/months polishing the code into "final" form
  - Code is often accepted into the Linux kernel to stabilize over a series of releases, if it doesn't break anything or introduce security holes
- Remember: The size of a submission and the coding style matter every single time

Some signs your code is not ready

- If it creates security vulnerabilities
- If it interferes with stable and functioning code elsewhere in the project
- If it does not compile
- If it does not meet proper coding style
- If it depends upon code that is in another tree

Typical patch submission process

Creating a patch

- The patch should be created against the most recent version of the mainline source code
- The patch should be offset from the root of the tree
- The patch must apply cleanly
- A freshly-patched version of the code should build without errors
- A patch should do one thing, and do it well

Be patient and persistent

- Send patches and responses in public, never in private
- Accept criticism, and rework the code
- Make incremental changes that are well communicated
- Resubmit the patch
- Be persistent and polite

Responding to feedback

- Feedback is likely to come in short, small bursts.
  - Read, revise, and retransmit the code
  - If you get someone that is reading and commenting on the code each time, that's great! Keep sending the code.
- Feedback from developers may be brief or strongly worded
  - Do not take it personally
  - Maintainers process a lot of code, and must triage quickly
  - Their goal is to improve code quality through intensive review
- Work with the community
  - Take the good suggestions you get and incorporate them into your code
  - If there are solid technical reasons why bad suggestions are bad, explain those reasons clearly

What if my patch is rejected?

- Code may be rejected for any reason
  - Poor quality, inconsistent formatting
  - Too much function in one submission
  - Inconsistent with broader subsystem strategy

- This doesn't make you a bad coder
  - Revise and try again
  - When replying, filter unnecessary feedback and focus just on the technical aspects

Escalation

- Sometimes patches slip through the cracks

- If you haven't received an acknowledgement after a reasonable amount of time, resend it
  - Typically one week
  - Add this line to the top of the email:
    - "Patch escalation: no response for 7 days"

Checklist before submitting

- Patch follows all coding style guidelines
- Patch applies cleanly against main project tree
- Project builds cleanly after patch is applied
- Patch is tested to do what it claims
- Header of file is documented properly

Reporting bugs

- Not uncommon to uncover bugs in other people's' code during development
  - Search for duplicate bugs first
  - Report the bug in relevant components
  - File separate bugs for separate issues

Best Practices: Testing

Testing is continuous

- Source code change management processes favor continuous testing
  - Every developer works on a complete branch of the tree
- Community processes require basic QA from the start
  - A patch must compile and meet basic standards before it will be considered at any level
- Tools like git are built with continuous testing and integration in mind
  - Patches that cause problems can be found and reverted
- Automated build suites help detect problems quickly
  - May automatically file bugs
- Some projects create and maintain test suites specific to their codebase

Best Practices: Maintenance

Maintaining your code

- Don't "dump and run"
  - Abandoned code will be worked around and eventually removed
- Disclose problems quickly, and provide workarounds and fixes
  - Pretending nothing is wrong will only buy you trouble
- If you can't maintain it, pass responsibility along to a successor, or arrange to have it removed
  - If your code is important, someone else will step up

Accepting patches to your code

- If your code is useful, others may want to enhance it
  - Be open to the community process
  - Be available to the maintainer if asked for your input on a patch
  - Consider the technical comments people make on the code, and justify any disagreements that you have with them

Open source development best practices

Ibrahim Haddad, Ph.D.
VP Strategic Programs, Linux Foundation
Executive Director, LF AI Foundation