

kworkflow

patch-hub v1.0.0

GSoC 2025 Proposal

Ivin Joel Abraham

ivinjabraham@gmail.com

Table Of Contents

Table Of Contents.....	2
Synopsis.....	3
Prerequisites.....	3
Phase One.....	3
Phase Two.....	5
Phase Three.....	5
Phase Four.....	5
Objectives.....	6
Motivation.....	6
Architectural Fragility.....	7
I. Overloaded model.....	7
II. Disconnected pieces of UI and business logic.....	8
III. Popups are handled at a global level.....	8
Proposed Changes.....	9
Architectural Refinements.....	9
Test Coverage and Quality Assurance.....	12
New Features.....	12
1. Compiling modified trees.....	12
2. Reviewing patches.....	13
Timeline.....	17
Time Commitments.....	18
Additional Goals.....	18
About Me.....	19
Bio.....	19
Technical Experience.....	19
Previous Contributions.....	20
Appendix.....	22

Synopsis

This project aims to push towards a stable release of *patch-hub* v1.0.0 by adding essential features, addressing growing architectural concerns and establishing a robust codebase to build upon in the future.

patch-hub, a sub-project of *kw*, first diverged from its parent project around July of 2024¹. To accommodate its broader scope, *patch-hub* was rewritten in Rust (from Bash) as a separate project with its own repository². Since then, the project has introduced multiple end-user features as well as taken full advantage of the capabilities of a high-level language like Rust to ensure robustness, efficiency, and convenient abstractions. However, *patch-hub* still remains in pre-release, currently on version 0.1.5.

Key priorities for v1.0.0 include delivering a streamlined experience that supports the full maintainer workflow—from receiving patches to reviewing, testing, and sending feedback—alongside improving the project’s modularity and long-term maintainability and solidifying testing and documentation.

Prerequisites

As advised, I have completed all four phases of the warm-up exercises³. Due to my background in kernel development, much of the content was familiar.

Phase One

This task involved the basic workflow of a kernel developer—compiling and installing a modified version of the kernel on a Virtual Machine. I typically use an Arch Linux QEMU Machine to test dummy drivers⁴. So that part of the task was elementary. The Debian VM took just a little bit longer, primarily because I’m a bit rusty with the apt package manager and its sources.

¹ Approximate date from this blog post from one of the current maintainers: [Link](#)

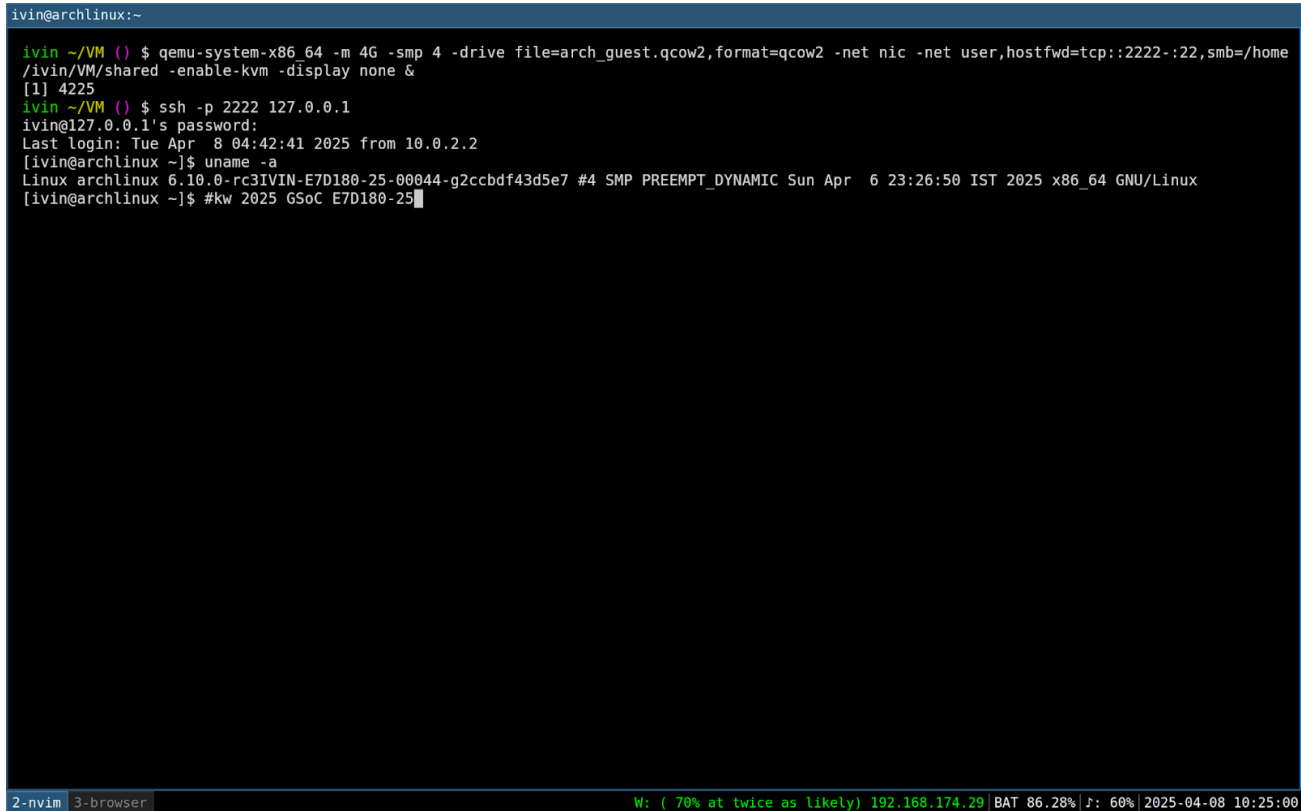
² Github repository: [Link](#)

³ Github discussion with the exercises: [Link](#)

⁴ Github repository with dummy drivers: [Link](#)

One challenge arose during Samba setup, as the recommended *fstab* method didn't work reliably. Most online solutions were geared toward Windows hosts or guests, so I adapted by manually mounting the Samba directories

```
ivin@archlinux:~  
ivin ~/VM () $ qemu-system-x86_64 -m 4G -smp 4 -drive file=arch_guest.qcow2,format=qcow2 -net nic -net user,hostfwd=tcp::2222-:22,smb=/home  
/ivin/VM/shared -enable-kvm -display none &  
[1] 4225  
ivin ~/VM () $ ssh -p 2222 127.0.0.1  
ivin@127.0.0.1's password:  
Last login: Tue Apr 8 04:42:41 2025 from 10.0.2.2  
[ivin@archlinux ~]$ uname -a  
Linux archlinux 6.10.0-rc3IVIN-E7D180-25-00044-g2ccbf43d5e7 #4 SMP PREEMPT_DYNAMIC Sun Apr 6 23:26:50 IST 2025 x86_64 GNU/Linux  
[ivin@archlinux ~]$ #kw 2025 GSoC E7D180-25
```

A terminal window showing the process of booting a QEMU VM. The user runs a command to start the VM with specific parameters. Then, they use SSH to connect to the VM. The terminal output shows the VM's login prompt, the user's password, the last login time, and the output of the 'uname -a' command, which displays the kernel version and system architecture. The bottom of the terminal shows the status of the terminal window and the system's battery level.

2-nvim 3-browser W: (70% at twice as likely) 192.168.174.29 | BAT 86.28% | r: 60% | 2025-04-08 10:25:00

Image: QEMU VM Booted with kernel appended with name, ID and version information

The second step was to compile the (modified) kernel on the host machine and run it on the guest. The performance boost when using the host's full access to hardware resources, something I had not thought of before, was definitely noticeable. As a wonderful consequence, this inspired contributions to my laptop's WMI driver to prevent overheating. I also learnt how to manually install the kernel on a Debian system since Arch has a slightly different process⁵.

The final step in phase one was to learn a little bit about Loadable Kernel Modules. As these commands were quite familiar to me, I took this time to learn more about how the kernel build system and dynamically loadable modules work.

⁵ Installing a kernel on Arch: [Link](#)

By the end of this phase, I had two QEMU VMs, with Debian and Arch and a custom kernel installed, setup with both *samba* and *ssh* access.

Phase Two

This involved exploring the *kw* tool itself. Thanks to its excellent documentation, I was able to complete this phase smoothly without external resources. I was pleasantly surprised to learn about the Pomodoro feature. The report produced is definitely a useful assessment for developers, though I felt that the user experience could potentially be improved.

Phase Three

I was initially planning to read most of the GNU Manual since it seemed quite short at first glance. I hoped to even summarize it into a quick blog post⁶ to read. However I quickly realized it would be inefficient and only focused on sections that are relatable/useful for my work and skimmed through the rest.

Phase Four

I had already visited the documents referenced while preparing a PR to improve *patch-hub*'s own documentation⁷. These resources follow many kernel-style conventions, which I've encountered multiple times—especially during the LFD103 Kernel Development course, which I've completed.

⁶ I did eventually make a post, containing some of the cool stuff I found in the manual: [Link](#)

⁷ PR that adds documentation: [Link](#)

Objectives

The primary objectives of this project are to achieve the following goals:

1. **Implement all features essential for v1.0.0:** Integrate all critical features required for the first major release, ensuring that *patch-hub* meets the core expectations of a robust and user-friendly tool for maintainers and developers alike.
2. **Resolve architectural issues:** Address the growing architectural challenges stemming from the current MVC design, which has aged and introduced issues with testability and modularity.
3. **Expand test coverage and documentation:** Develop comprehensive test suites to cover the existing codebase, which currently lacks adequate testing. Additionally, create thorough documentation to facilitate maintenance and onboarding for new contributors.

Motivation

patch-hub's primary aim is to streamline and modernize the Linux kernel patch workflow. Since its rewrite in Rust, the project has introduced a number of useful features and abstractions. But despite this progress, several growing pains now stand in the way of a stable and complete 1.0 release.

The primary challenges are both **architectural** and **functional**:

- **Architectural Fragility:** As new components were added, they were built on top of a monolithic and aging MVC structure. This has led to an overloaded central model, unclear separation of concerns, and an execution flow that's difficult to reason about or extend. Without intervention, future development will become increasingly difficult and error-prone.
- **Lack of Testability:** The architectural issues have made it hard to write reliable unit tests, and test coverage remains extremely low. This poses a risk to stability and slows down contributor confidence and iteration speed.

-
- **Missing Core Features:** patch-hub is currently limited in its support for the full maintainer workflow. For example, while it can apply patches, it lacks the ability to:
 - Build and test kernel trees with the applied patches.
 - Leave line-specific review comments—arguably the most essential part of patch review. These gaps make it difficult for maintainers to rely on patch-hub as their primary tool and limit its adoption.
 - **Onboarding Friction:** Finally, poor or missing documentation makes it hard for new contributors to get started. Without a clearer structure and guidance, the community around patch-hub can't grow.

Architectural Fragility

This section highlights some of the architectural issues that limit maintainability, testability, and future extensibility.

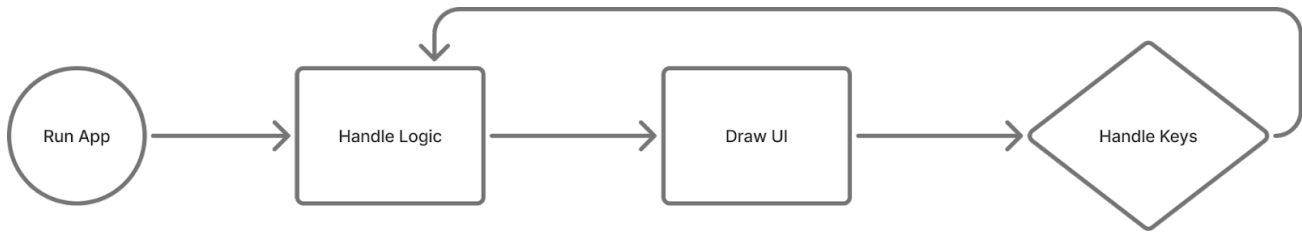
I. Overloaded model

The *App* struct^[a] acts as the central data store for the entire application, housing state and resources used across various screens. Over time, this struct has become bloated, accumulating responsibilities that go beyond the intended role of a "model" in the MVC paradigm.

- It contains fields used only by specific screens, creating tight coupling between unrelated parts of the application.
- It also stores application-wide configurations such as log paths, cache directories, and client instances—concerns that should be isolated from screen-level state and logic.

This over-centralization reduces modularity and makes it difficult to test components in isolation. Moving configuration and context-specific logic into smaller, dedicated structures is a necessary step toward a more maintainable design.

II. Disconnected pieces of UI and business logic



The application's main loop currently follows three phases: logic handling^[b], UI drawing, and key input handling. However, the "logic handling" phase is misleadingly named and poorly defined. In practice, this step does very little beyond rendering a loading screen or redirecting to the mailing list if there are no bookmarked patches.

This separation of concerns is more artificial than functional. A clearer model would integrate loading before the drawing phase—by deferring rendering until the required data is ready—thus simplifying the loop into three intuitive steps: handle input, get data and draw UI.

III. Popups are handled at a global level

Popups, though conceptually tied to specific screens (e.g., help dialogs, code review trailers dialog), are currently treated as global UI components.

- They are managed at a global level, separate from the screen they conceptually belong to.
- This leads to awkward and leaky abstractions in the input handling code. The `key_handling()`^[c] function must route events either to the current screen or to an active popup, creating mixed responsibilities and violating the principle of **separation of concerns**.

Ideally, popups should be managed and rendered within their respective screen contexts, with input events delegated to them only when active. This would allow for cleaner encapsulation and clearer input flow.

Proposed Changes

Disclaimer: all code snippets and UI mockups are for demonstration purposes only. They lack refinement and may differ significantly in terms of appearance in the final project.

Architectural Refinements

To prevent overloading *App*, we can use an extension of the MVC model—MVVM. In this model, we'll create a *ViewModel* for every screen, that is responsible for the business logic for that screen. Essentially, we split the *App* struct into multiple structs, one per screen.

```
1 struct App {
2     current_screen: Screen,
3     viewmodels: HashMap<Screen, dyn ViewModel>,
4     lore_client: LoreClient,
5     config: Config
6 }
```

Snippet: A cleaned up App struct

The pseudo-code snippet above shows how we can maintain a set of viewmodels, one for each screen that has been initialized. In the “Draw UI” part of the process, we proceed as normal and draw using *current_screen*. “Handle Keys” will access the *viewmodels* map and check if there already exists a viewmodel. If it does, then it can simply pass control to that viewmodel and if it does not, it will need to initialize one before passing control.

```
1 pub trait ViewModel {
2     fn handle_key(&mut self, key: Key) -> Result<()>;
3     fn draw(&self) -> Result<()>;
4     fn load(&mut self) -> Result<()>;
5 }
```

Snippet: A simple implementation of a ViewModel

```

1 fn key_handling(...) -> Result<ControlFlow<>> {
2     let viewmodel = app
3         .viewmodels
4         .entry(app.current_screen)
5         .or_insert_with(|| create_viewmodel(app.current_screen));
6
7     viewmodel.handle_key(key, &mut terminal)?;
8
9     Ok(ControlFlow::Continue(terminal))
10 }

```

Snippet: Key handling using viewmodels

```

1 pub fn create_viewmodel(screen: Screen) -> Box<dyn ViewModel> {
2     match screen {
3         Screen::MailingList => Box::new(MaillistVM::new()),
4         Screen::PatchList => Box::new(PatchListVM::new()),
5         Screen::PatchDetails => Box::new(PatchDetailsVM::new()),
6     }
7 }

```

Snippet: create_viewmodel() can simply initialize and return any new VMs.

We can now take this a step further and even remove the *LoreClient* out of the main model and only pass it to the required viewmodels.

```

1 pub fn create_viewmodel(
2     screen: Screen,
3     client: LoreApiClient,
4 ) -> Box<dyn ViewModel> {
5     match screen {
6         Screen::Home => Box::new(HomeViewModel::new()),
7         Screen::PatchList => Box::new(PatchListViewModel::new(client.clone())),
8         Screen::PatchDetails => Box::new(PatchDetailsViewModel::new()),
9     }
10 }

```

Snippet: Passing LoreClient only to require ViewModels

```

1 struct App {
2     current_screen: Screen,
3     viewmodels: HashMap<Screen, dyn ViewModel>,
4 }

```

Snippet: Cleaned up App

Of course, in practice, it is likely that we might need to add a few more fields to *App*, but theoretically, this should be enough.

The code snippet below shows how to take advantage of these viewmodels to implement loading. When switching to any *Loadable* screen, we must call *load_screen()* which will handle the loading for that particular screen. Once loaded, switches will no longer trigger loading.

```
1 pub struct MailListModel {
2     mailing_lists: Vec<MailingList>,
3     target_list: String::new(),
4     target_list: String::new(),
5     is_loaded: false,
6 }
7
8 pub trait Loadable {
9     async fn load_screen(&mut self) -> Result<(), E>;
10    fn is_loaded(&self) -> bool;
11 }
12
13 impl Loadable for MailListModel {
14     async fn load_screen(&mut self) -> Result<(), E> {
15         if !self.is_loaded {
16             // either call loading_screen macro (must be blocking) here
17             // or directly call the api and use the
18             // is_loaded variable to control the view
19
20             self.is_loaded = true
21         }
22     }
23
24     fn is_loaded(&self) -> bool {
25         self.is_loaded
26     }
27 }
```

Snippet: Loadable screens

Popups can be handled similarly to how it is currently handled, except it is managed within the viewmodel.

Test Coverage and Quality Assurance

As of March 15th, *patch-hub* has just under 15% test coverage⁸. This low number is not merely due to oversight, but rather the result of structural barriers that make testing difficult. In particular, the monolithic App struct creates tight coupling between components, making it hard to isolate functionality for unit tests.

A key goal of this project is to improve testability as part of a broader push toward code quality and long-term maintainability. By refactoring the architecture to use modular, screen-specific ViewModels, we can isolate logic more effectively—making it easier to write meaningful, targeted tests. This shift will not only boost test coverage but also lay a stronger foundation for future contributors to work with confidence and minimal friction.

New Features

As *patch-hub* primarily attracts maintainers, the following features are crucial to roll out before the first release.

1. Compiling modified trees

Currently *patch-hub* supports applying a patchset to a specified kernel tree. However, this can be a more useful feature if the user was able to compile and test whether the patch breaks the build or introduces regressions. The latter is difficult to test locally for most patches, since a patch may change code for different architecture than the host machine, may only affect edge cases or in specific setups and so on. There are, however, tools like *coccinelle* and *kselftest* which can find problems.

With this new feature, I primarily aim to introduce a new action, “build”, when viewing a patchset. This feature will use *kw* to build the kernel tree specified in the *config*.

⁸ Coverage report: [Link](#)

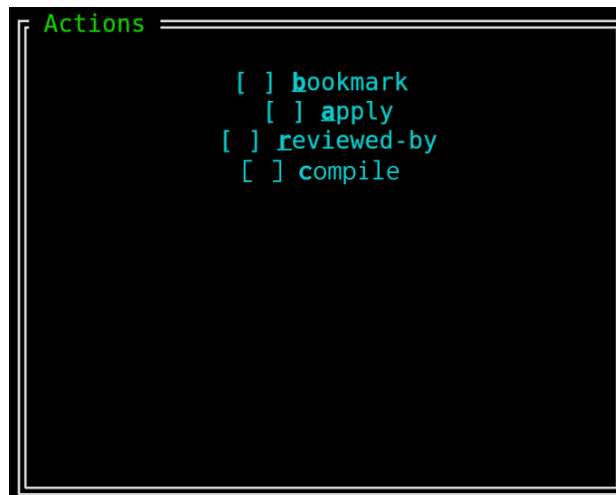


Image: Mockup of the new compile action added to the actions tab

In-case the tree does not contain `.config` file, it will automatically ask to copy the host machine's config.



Image: Mockup of a new dialog box asking for creation of a new `.config` file

2. Reviewing patches

Currently, *patch-hub* can only view patches and amend them with tags such as “Reviewed by”. Most patches sent to the kernel receive at least one comment asking for changes and the ability to do this will be a major factor in whether maintainers will use *patch-hub*.

For this, another action “Review” will be added that lets you enter review mode.

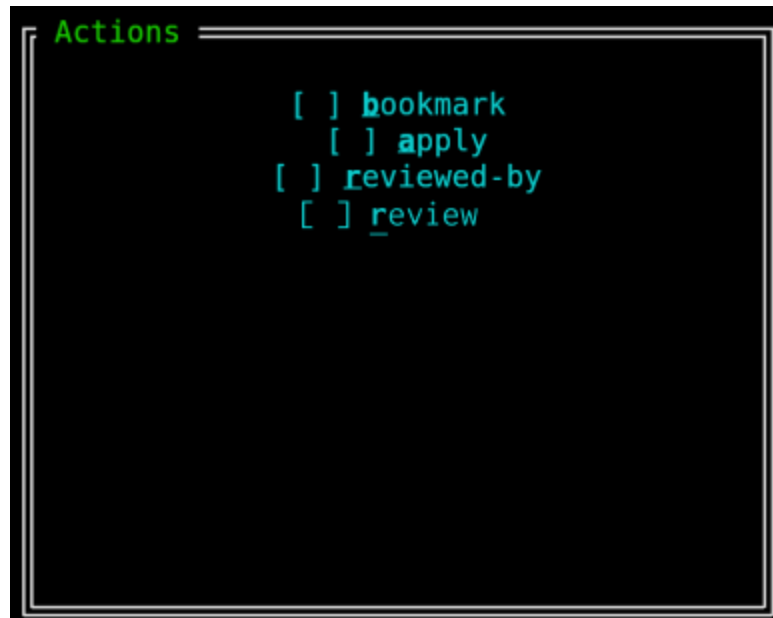


Image: Mockup of the new review action added to the actions tab

Review mode will allow you to visually select lines (just like Vi) and hit a key to add a “comment”.



Image: Mockup of the highlighting available in review mode



Image: Mockup of adding a comment on highlighted lines

Comments will be saved internally, to be sent when the user sends the email. Of course, the user will be shown the mail and given a chance to edit it before sending it in. I was also thinking of being able to set templates for the initial body (“Hi, thank you for your patch...”). If time permits, I will be able to flesh out this feature a lot more.



Image: Mockup of dialog box to edit and send email.

Timeline

Below are the tentative dates that I've scheduled to get specific tasks completed. Following the table, I have outlined my time commitments to my academics and other work which influenced the planning of this schedule.

<u>Period</u>	<u>Tasks</u>
After Proposal Submission [April 08 - May 08]	<ul style="list-style-type: none">- Discuss and review the project with the mentors- Finalize the implementation details of the new architecture with a high level design- Start documenting the existing codebase
Community Bonding Period [May 08 - Jun 1]	<ul style="list-style-type: none">- Get to know the mentors and the community of <i>patch-hub</i>- Start implementing the new architecture for the mailing list screen and patch list screen
Week 1 & 2 [Jun 02 - 15]	<ul style="list-style-type: none">- Implement the new architecture for the patch details screen and wrap up the mailing and patch list screens.- Convert existing components to fit better into the new architecture
Week 3 & 4 [Jun 16 - 29]	<ul style="list-style-type: none">- Milestone I: Wrap up work with the new architecture- Start writing tests for the codebase wherever necessary.
Week 5 & 6 [Jun 30 - Jul 13]	<ul style="list-style-type: none">- Focus on writing tests, prioritizing the latest changes- Document the new architecture, perhaps externally- Start work on the compilation feature
Midterm Evaluation [July 14]	
Week 7 & 8 [Jul 14 - 27]	<ul style="list-style-type: none">- Wrap up work on the compilation feature- Start work on the review feature
Week 9 & 10 [Jul 28 - Aug 10]	<ul style="list-style-type: none">- Milestone II: Wrap up work on the review feature- Write up tests for the new features
Week 11 & 12 [Aug 11 - 24]	<ul style="list-style-type: none">- Milestone III: Write up any remaining tests and documentation- Buffer week. Finish up any pending work.- Wrap up the project and get ready for submission.
Final Week [Aug 25 - Sep 1]	

Time Commitments

As an undergraduate student, most of my hours will be spent in classes and academic work when the semester is ongoing. My time commitment to the project will depend on my academic calendar but can be tentatively split as below:

- May 08 – Jun 09: I have regular classes during this period and will be spending around 25–30 hours per week on the project.
- Jun 11 – July 14: This month will be our final evaluation month, where we will have a plethora of projects, examinations and assignments to get through. I expect to spend around 10–15 hours per week on the project.
- July 15 – Aug 03: Our semester break averages around half a month typically, but it may be more than a month as well. I am sticking on the safe side and assuming we will get just around two weeks. During this break, I will be able to dedicate around 40–50 hours per week to make up for the lower productivity in the last month.
- Aug 03 – Sep 01: Regular classes should be in full swing and I will be able to spend around 25–30 hours per week on the project.

Additional Goals

If time permits, I hope to also enhance the user interface to make it more visually appealing as well as easier to extend with more features in the future.

About Me

Bio

I am a junior computer science engineering student at Amrita Vishwa Vidyapeetham, India. I was introduced to computers when I was a kid, with a low-end family laptop that often struggled to run any of the video games that I wanted. When I was introduced to the internet, I began learning how to optimize the system—or at the very least, understand why it won't run certain applications. If you took my search history from that period, "System requirements" would undoubtedly be the most searched term. This experience introduced me to computer architecture and systems and I've been captivated ever since.

I started programming in high school and was one of the first in my year to join the computer science club at our university, amFOSS⁹. While I originally joined to connect with a community of computer science enthusiasts like myself, I soon discovered the world of Free and Open Source software, which gave my work new purpose and direction.

Technical Experience

I spent my freshman year exploring various domains—backend, frontend and mobile development—as my seniors advised. I picked up these skills one by one and I am now quite confident in my ability to build full-fledged applications regardless of the platform. Recently, I've built an Android app in Kotlin (with a Golang backend) aimed at maintaining a healthy digital wellbeing¹⁰ for FOSSHack 2025. All of the projects I've made are open source and available on my GitHub¹¹.

But I eventually found my way to programming more foundational software—compilers, web servers, operating systems and the like. I've since focused my efforts on low-level systems programming and learned to write simple device drivers for Linux as well as quite a

⁹ Club website: [Link](#)

¹⁰ Repository: [Link](#)

¹¹ Github profile: [Link](#)

lot of theory in the field of operating systems and computer architecture. I've also made a simple web server¹² in pure GNU assembly and plan to make it a todo app.

Due to Rust's growing importance in the kernel, I started learning it late in 2023. Since then I've contributed to Rust's most popular (unofficial) Discord API library, Serenity¹³ and currently maintain my club's database backend¹⁴ and discord bot¹⁵. Now, I am focusing on working on Rust for Linux¹⁶.

I also aim to become an active contributor to the kernel. I have contributed a simple documentation patch¹⁷ while learning the kernel development workflow and am working on patches for my current laptop's drivers.

This interest in kernel development is what led me to *kworkflow* and to *patch-hub*. Although I only use *patch-hub* to browse patches as a way to learn more about the ongoing work in the kernel world, I believe this project will be significant for maintainers. As far I know, the current workflow revolves around mail clients that are not built for reviewing code. Lack of syntax highlighting and difficult review process (having to copy paste and manually quote code from a mail) are pain points that *patch-hub* will fix.

Previous Contributions

I have been studying and working with the *patch-hub* codebase quite extensively, with 12 PRs merged and three currently in-review. These PRs targeted different aspects of the project such as tests, bug fixes, documentation, refactors and infrastructure updates. This across-the-board approach has allowed me to understand the project and its needs. These PRs and the feedback I received from them has also given me the confidence to tackle major problems such as redesigning the architecture. The following is a table summarizing all of my work on *patch-hub* to-date and any remarks if necessary.

¹² Repository: [Link](#)

¹³ Serenity project: [Link](#)

¹⁴ Club backend: [Link](#)

¹⁵ Discord Bot: [Link](#)

¹⁶ Rust For Linux project: [Link](#)

¹⁷ Patch on lore.kernel.org archives: [Link](#)

Change	Remarks
Improving error handling and fault tolerance: <ul style="list-style-type: none"> Add missing error logs: #101 Replace <i>unwraps</i> with proper error handling: #105 	Merged.
Infrastructure updates: <ul style="list-style-type: none"> Adding workflows: #125, #109 Add codecov integration: #110 	Codecov integration is pending organization approval.
Refactors: <ul style="list-style-type: none"> Refactor a verbose function: #103 Implement dependency injection: #126 Alter main execution flow for performance: #127 Directory and module restructure: #128 	Directory and module restructure is pending some more research and chore git work.
Documentation: <ul style="list-style-type: none"> Add CONTRIBUTING.md and README.md: #121 Add intra-doc links to tests: #130 	Test linkage is still WIP and needs more research.
Bug Fixes: <ul style="list-style-type: none"> Fix trailing lines in patch preview: #106 Fix cargo install failing: #120 	Merged.
Misc: <ul style="list-style-type: none"> Remove redundant test: #129 	Merged.

Appendix

1. struct App

```
1  /// Type that represents the overall state of the application. It can be viewed
2  /// as the Model component of `patch-hub`.
3  pub struct App {
4      /// The current active screen
5      pub current_screen: CurrentScreen,
6      /// Screen to navigate and select the mailing lists archived on Lore
7      pub mailing_list_selection: MailingListSelection,
8      /// Screen with listing patchsets that were previously bookmarked
9      pub bookmarked_patchsets: BookmarkedPatchsets,
10     /// Screen with paginated listing of latest patchsets from a target list
11     pub latest_patchsets: Option<LatestPatchsets>,
12     /// Screen with details (metadata and previewing) and runnable actions of individual
    patchset
13     pub details_actions: Option<DetailsActions>,
14     /// Screen to edit configurations of the app
15     pub edit_config: Option<EditConfig>,
16     /// Database to track patchsets `Reviewed-by` state
17     pub reviewed_patchsets: HashMap<String, HashSet<usize>>,
18     /// Configurations of the app
19     pub config: Config,
20     /// Client to handle Lore API requests and responses
21     pub lore_api_client: BlockingLoreAPIClient,
22     pub popup: Option<Box<dyn PopUp>>,
23 }
```

2. logic_handling()

```
1 fn logic_handling<B>(mut terminal: Terminal<B>, app: &mut App) ->
  color_eyre::Result<Terminal<B>>
2 where
3     B: Backend + Send + 'static,
4 {
5     match app.current_screen {
6         CurrentScreen::MailingListSelection => {
7             if app.mailing_list_selection.mailing_lists.is_empty() {
8                 terminal = loading_screen! {
9                     terminal, "Fetching mailing lists" => {
10                         app.mailing_list_selection.refresh_available_mailing_lists()?;
11                     }
12                 };
13             }
14         }
15         CurrentScreen::LatestPatchsets => {
16             let patchsets_state = app.latest_patchsets.as_mut().unwrap();
17             let target_list = patchsets_state.target_list().to_string();
18             if patchsets_state.processed_patchsets_count() == 0 {
19                 terminal = loading_screen! {
20                     terminal,
21                     format!("Fetching patchsets from {}", target_list) => {
22                         patchsets_state.fetch_current_page()?;
23                     }
24                 };
25             }
26             app.mailing_list_selection.clear_target_list();
27         }
28     }
29     CurrentScreen::BookmarkedPatchsets => {
30         if app.bookmarked_patchsets.bookmarked_patchsets.is_empty() {
31             app.set_current_screen(CurrentScreen::MailingListSelection);
32         }
33     }
34     _ => {}
35 }
36
37 Ok(terminal)
38 }
39
```

3. key_handling()

```
1 fn key_handling<B>(  
2     mut terminal: Terminal<B>,  
3     app: &mut App,  
4     key: KeyEvent,  
5 ) -> color_eyre::Result<ControlFlow<>, Terminal<B>>>  
6 where  
7     B: Backend + Send + 'static,  
8 {  
9     if let Some(popup) = app.popup.as_mut() {  
10         if matches!(key.code, KeyCode::Esc | KeyCode::Char('q')) {  
11             app.popup = None;  
12         } else {  
13             popup.handle(key)?;  
14         }  
15     } else {  
16         match app.current_screen {  
17             CurrentScreen::MailingListSelection => {  
18                 return handle_mailing_list_selection(app, key, terminal);  
19             }  
20             CurrentScreen::BookmarkedPatchsets => {  
21                 return handle_bookmarked_patchsets(app, key, terminal);  
22             }  
23             CurrentScreen::PatchsetDetails => {  
24                 handle_patchset_details(app, key, &mut terminal)?;  
25             }  
26             CurrentScreen::EditConfig => {  
27                 handle_edit_config(app, key)?;  
28             }  
29             CurrentScreen::LatestPatchsets => {  
30                 return handle_latest_patchsets(app, key, terminal);  
31             }  
32         }  
33     }  
34     Ok(ControlFlow::Continue(terminal))  
35 }  
36
```