

Project Report
On
Automating Koji Build System for Building RPMs



Submitted
In partial fulfilment

For the award of the Degree of

PG-Diploma in Embedded Systems and Design
(PG-DESD)

Under the esteemed guidance of
Mr. Surendra Billa
HPC Tech, C-DAC, ACTS (Pune)

Submitted By

Garvit Sharma 230940130024

Satya Prakash 230940130053

Akshata Dhage 230940130006

Centre for Development of Advanced Computing (C-DAC), ACTS

(Pune- 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Mr. Surendra Billa** C-DAC ACTS, Pune for her constant guidance and helpful suggestion for preparing this project **Automating Koji Build System for Building RPMs**. We express our deep gratitude towards her for inspiration, personal involvement, constructive criticism that she provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Ms. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mrs. Risha P R (Program Head)** and **Mrs. Srujana Bhamidi** (Course Coordinator, PG-DESD) for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Garvit Sharma 230940130024

Satya Prakash 230940130053

Akshata Dhage 230940130006

ABSTRACT

Building RPMs can be a time-consuming and manual process, especially when managing multiple packages and architectures. This project aims to automate the Koji build system, resulting in significant improvements in efficiency, consistency, and reproducibility. This project presents a novel approach to automate Fedora Linux builds for the RISC-V architecture using Koji, the widely used build system of Enterprise Linux distributions. Instead of manual triggers, webhooks streamline the process, initiating builds automatically upon code changes. A local Python server acts as the central hub, receiving and processing these triggers. NGROK serves as a reverse proxy, enabling communication between the hub and external sources. By leveraging open-source tools, this project aims to achieve

- **Faster Build Times:** Automated triggers and efficient orchestration significantly reduce build duration.
- **Increased Efficiency:** Manual intervention is minimized, freeing up valuable developer resources.
- **Enhanced Accessibility:** Continuous builds ensure readily available RISC-V images for Fedora users.
- **Boosted Innovation:** Faster development cycles enable quicker experimentation and progress.

This project demonstrates the power of open-source collaboration and automation in accelerating software development for emerging architectures like RISC-V, paving the way for wider adoption and innovation within the Fedora ecosystem.

Table of Contents

S. No	Title	Page No.
	Front Page	I
	Acknowledgement	II
	Abstract	III
	Table of Contents	IV
1	Introduction	01-05
1.1	Introduction	01
1.2	Objective and Specifications	04
2	Why we are Automating koji build system	06
3	Methodology/ Techniques	10-26
3.1	Gitea Server	10
3.2	ngrok	15
3.3	Tornado	17
3.4	Python Application	19
3.5	Koji	20
3.6	Model Description	25
4	Implementation	27-35
4.1	Implementation	27
4.2	Flowchart	29
4.3	Algorithm	30
4.4	Python Code	32
5	Results	36
5.1	Results	36
6	Conclusion	37-38
6.1	Conclusion	37
7	References	39
7.1	References	39

Chapter 1

Introduction

1.1 Introduction

RPM stands for Red Hat Package Manager, but can also recursively mean **RPM Package Manager**. It refers to both:

- **The .rpm file format:** A package format used to distribute software in Fedora Linux and other RPM-based Linux distributions. Each .rpm file contains all the necessary files and information to install a specific software package.
- **The package management system itself:** Software tools used to install, remove, query, and manage RPM packages on a system. This includes programs like dnf (default in Fedora) and yum.

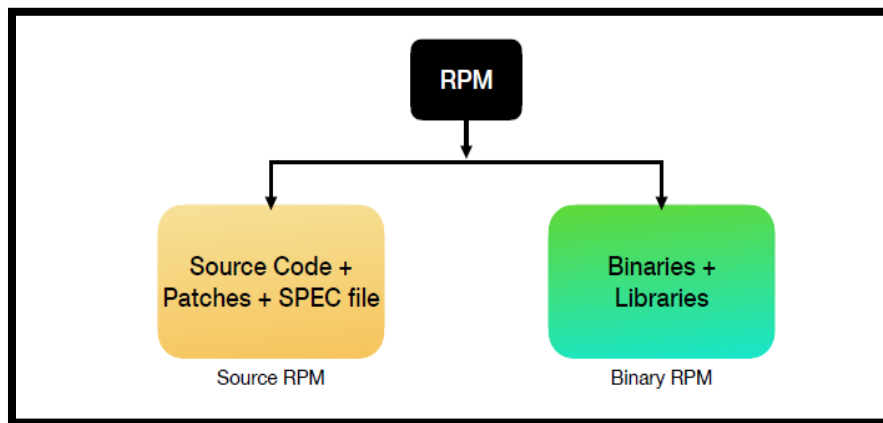


Fig1.RPM

Why Build RPM Packages?

Several reasons make building RPM packages crucial for software distribution in Fedora:

1. Installation Management:

- RPM packages provide a standardized way to distribute and install software. They contain all necessary files, dependencies, and configuration information, ensuring a smooth and predictable installation process.
- The package management system manages dependencies automatically, resolving conflicts and ensuring compatibility between packages.

2. Security and Integrity:

- RPM packages are digitally signed, verifying the authenticity and integrity of the software before installation. This protects users from installing malicious software or tampered packages.
- Packages can also be signed with specific keys, allowing users to only install software from trusted sources.

3. Dependency Management:

- RPM packages explicitly list their dependencies on other packages, ensuring all required components are installed for the software to function properly.
- The package management system automatically installs missing dependencies and handles version conflicts, simplifying software management.

4. Version Control and Rollbacks:

- RPM packages track different versions of software, allowing users to easily install specific versions or upgrade/downgrade if needed.
- Package management systems enable rolling back to previous versions in case of issues, providing a safety net for system stability.

5. System Administration:

- RPM packages simplify software management for system administrators, allowing easy deployments, updates, and removals across multiple systems.
- Package management tools offer centralized control and automated processes for efficient administration.

Building and maintaining RPMs, especially for emerging architectures like RISC-V, can be a time-consuming and resource-intensive process. Traditional manual approaches often suffer from slow turnaround times, limited scalability, and dependence on developer involvement. This project presents a novel solution: an automated build system leveraging Koji, webhooks, and a central Python server to streamline Fedora Linux builds for the RISC-V architecture.

Our project eliminates the need for manual triggers by utilizing webhooks. These automated notifications initiate builds whenever code changes occur, ensuring immediate responsiveness and faster build times. A central Python server acts as the brain of the operation, receiving and processing these triggers, orchestrating the build process, and communicating with the Koji build system. NGROK acts as a reverse proxy, enabling seamless communication between the local server and external sources.

By harnessing the power of open-source tools and automation, this project aims to deliver several key benefits:

- Significantly reduced build times: Webhooks and efficient orchestration streamline the process, leading to faster builds and quicker availability of updated packages.
- Increased efficiency and reduced manual intervention: Automating the build process frees up valuable developer resources for other tasks, promoting overall efficiency.
- Enhanced accessibility: Continuous builds ensure readily available RISC-V images

for Fedora users, improving accessibility and fostering wider adoption.

- Boosted innovation: Faster development cycles enabled by automation empower quicker experimentation and progress, leading to a more vibrant and innovative Fedora ecosystem.

This project demonstrates the potential of open-source collaboration and automation in accelerating software development for emerging architectures. By automating Fedora builds on RISC-V, we pave the way for wider adoption, faster innovation, and a more accessible Fedora experience for all.

1.2 Objective

The traditional process of building and maintaining RPMs, especially for emerging architectures like RISC-V, often suffers from limitations like slow turnaround times, limited scalability, and high dependence on manual intervention. These challenges hamper the widespread adoption and development of software for RISC-V within the Fedora ecosystem.

This project aimed to address these limitations by automating the Koji build system for building Fedora Linux RPMs specifically for the RISC-V architecture. We focused on achieving the following key objectives:

1. Reduce Build Time:

- Eliminate manual build triggers and dependencies that create bottlenecks in the traditional process.
- Leverage webhooks to initiate builds automatically upon code changes, ensuring immediate response and significantly faster build duration.
- Optimize build configurations and utilize efficient build tools to further reduce build times.

2. Increase Efficiency:

- Minimize manual intervention in the build process by automating trigger initiation, data retrieval, and build management.
- Free up valuable developer resources for other tasks, such as bug fixing, feature development, and community engagement.
- Streamline the overall build workflow for better resource utilization and improved developer productivity.

3. Enhance Accessibility:

- Ensure continuous and automated builds, leading to readily available RISC-V images for Fedora users.
- Improve accessibility by eliminating manual intervention and reducing build times, fostering wider adoption of RISC-V within the Fedora community.
- Contribute to a larger pool of available RISC-V software packages, benefiting developers and users alike.

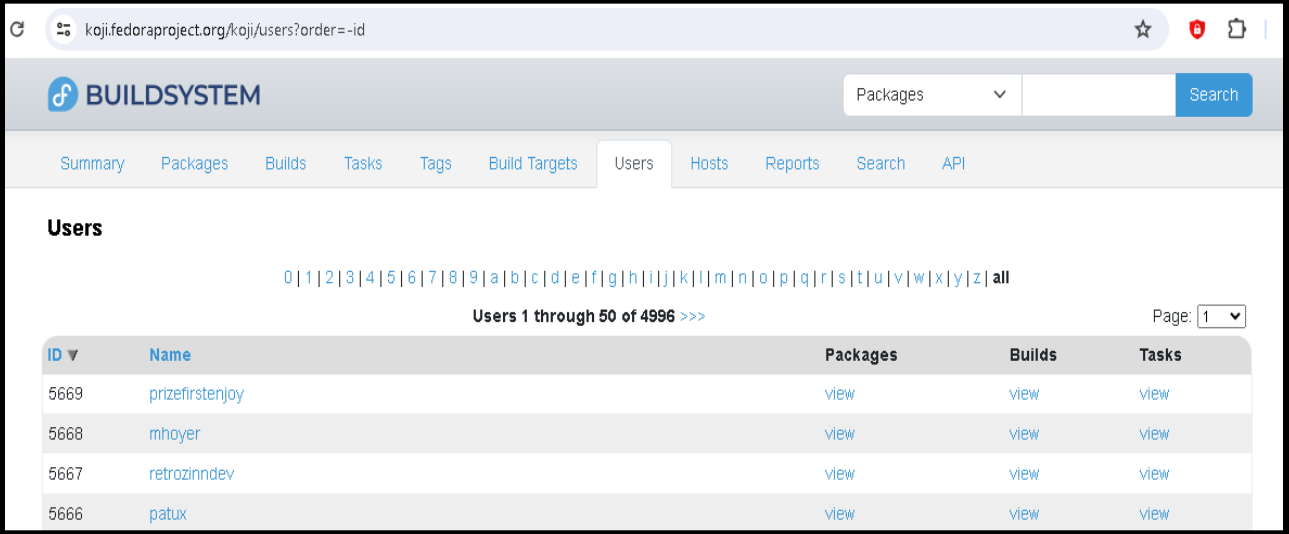
4. Boost Innovation:

- Enable faster development cycles by reducing build times and simplifying the build process.
- Encourage experimentation and progress by making it easier for developers to test and deploy changes.
- Contribute to a more vibrant and innovative Fedora ecosystem for RISC-V through efficient and accessible build infrastructure.

Chapter 2

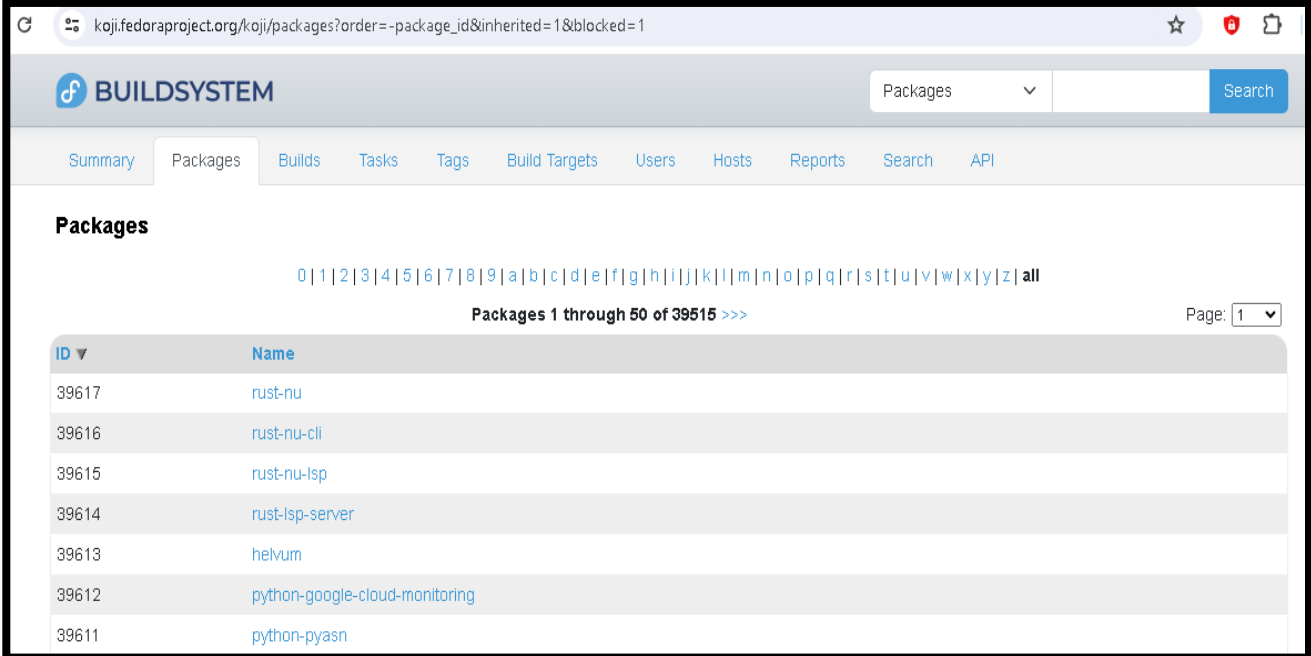
Why we are Automating koji build system

There are so many users to maintain the packages for build as shown in fig. If a user wants to make changes then user has to rebuild the packages.

The screenshot shows the 'Users' page of the Koji Build System. The page has a navigation bar with tabs for Summary, Packages, Builds, Tasks, Tags, Build Targets, Users (selected), Hosts, Reports, Search, and API. Below the navigation bar, there is a search bar and a 'Search' button. The main content area is titled 'Users' and includes a pagination link '0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | all'. Below this, it says 'Users 1 through 50 of 4996 >>>'. A table lists the first five users, each with a 'view' link for Packages, Builds, and Tasks.

ID	Name	Packages	Builds	Tasks
5669	prizefirstenjoy	view	view	view
5668	mhoyer	view	view	view
5667	retrozinndev	view	view	view
5666	patux	view	view	view

Fig 2. number of users maintaining packages and builds



The screenshot shows the Koji Build System web interface. The URL in the browser is `koji.fedoraproject.org/koji/packages?order=-package_id&inherited=1&blocked=1`. The page has a header with the 'BUILDSYSTEM' logo and a search bar. Below the header is a navigation menu with tabs: Summary, Packages, Builds, Tasks, Tags, Build Targets, Users, Hosts, Reports, Search, and API. The 'Packages' tab is selected. The main content area is titled 'Packages' and displays a list of packages. Above the list is a navigation bar with letters from 0 to z and 'all'. Below this, it says 'Packages 1 through 50 of 39515 >>>'. The package list has two columns: 'ID' and 'Name'. The packages shown are:

ID	Name
39617	rust-nu
39616	rust-nu-cli
39615	rust-nu-lsp
39614	rust-lsp-server
39613	helvum
39612	python-google-cloud-monitoring
39611	python-pyasn

Fig 3. Number of packages built

Without automation, building packages manually involves a multi-step process requiring significant time and effort:

1. Code Preparation:

- **Developers:** Manually prepare the codebase for building, ensuring all necessary files and dependencies are included.
- **Version Control:** Commit changes to a version control system (e.g., Git) for tracking and collaboration.

2. Build Environment Setup:

- **Manual Configuration:** Set up a dedicated build environment with the required tools and libraries for compilation.
- **Dependency Management:** Manually install and manage all software dependencies needed for building the package.

3. Building the Package:

- **Manual Commands:** Execute build commands specific to the programming language and project requirements.
- **Troubleshooting:** Manually address any errors or issues that arise during the build process.

4. Distribution and Installation:

- **Manual Upload:** Upload the built package to a repository or share it directly with users.
- **Manual Installation:** Users manually download and install the package on their systems.

Challenges of Manual Builds:

- **Time-Consuming:** The entire process can be slow and require significant developer involvement.
- **Error-Prone:** Manual steps increase the risk of human error and inconsistencies in builds.
- **Repetitive Tasks:** Manual setup and configuration can be tedious and repetitive across builds.
- **Scalability Issues:** Difficult to scale up builds for large projects or frequent updates.
- **Lack of Version Control:** Builds might not be explicitly controlled or easily reproducible.

Automating the Koji build system for building RPMs, particularly for RISC-V architecture, brings numerous benefits in terms of **abstraction and mass builds**. Here's why:

Abstraction:

- **Reduced Manual Intervention:** Manual triggers and dependencies often hinder the build process. Automation eliminates manual steps, abstracting away complexities and streamlining the workflow. Developers focus on code, not build mechanics.
- **Declarative Configuration:** Build configurations become code, stored and managed centrally. This decouples the build process from specific environments, promoting portability and maintainability.
- **Standardized Builds:** Automation enforces consistent build procedures, regardless of who triggers the build, ensuring reproducible and reliable outcomes across diverse environments.

Mass Builds:

- **Faster Build Times:** Webhooks automatically trigger builds upon code changes, eliminating delays and accelerating build initiation. Optimized build configurations and tools further reduce build durations.
- **Scalability and Concurrency:** The automated system can handle numerous concurrent builds efficiently, accommodating large-scale deployments or continuous integration pipelines.
- **Increased Availability:** Continuous builds ensure readily available RPMs for RISC-V users, eliminating manual intervention and delays.
- **Improved Testing and Feedback:** Faster builds enable more frequent testing and quicker feedback cycles, fostering rapid development and innovation.

Chapter 3

Methodology and Techniques

3. Methodology:

This section elaborates on the approach taken to automate the Koji build system for building Fedora Linux RPMs specifically for the RISC-V architecture. Our methodology aimed to address the limitations of manual builds and achieve the project's objectives outlined in the Introduction.

Key components and their functionalities:

3.1 Gitea Server

Gitea is a painless self-hosted all-in-one software development service, it includes Git hosting, code review, team collaboration, package registry and CI/CD. It is similar to GitHub, Bitbucket and GitLab. Gitea was forked from Gogs originally and almost all the code has been changed.

Features:

- **Code Hosting:** Gitea supports creating and managing repositories, browsing commit history and code files, reviewing and merging code submissions, managing collaborators, handling branches, and more. It also supports many common Git features such as tags, Cherry-pick, hooks, integrated collaboration tools, and more.

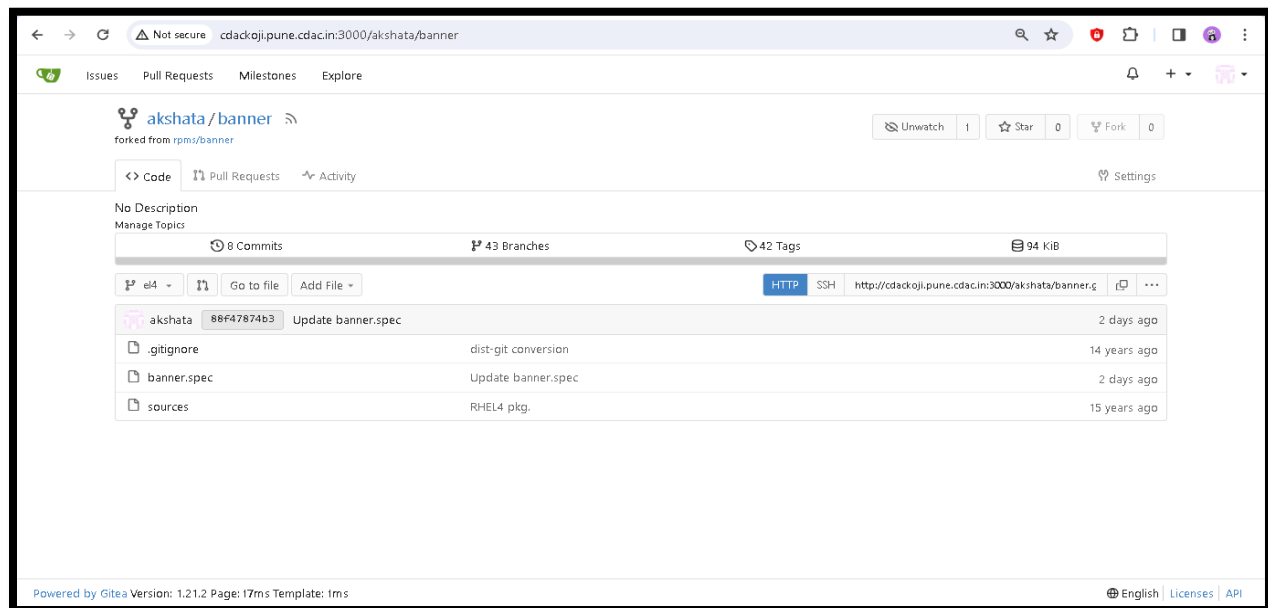


Fig 4: Gitea Server

- **Lightweight and Fast:** One of Gitea's design goals is to be lightweight and fast in response. Unlike some large code hosting platforms, it remains lean, performing well in terms of speed, and is suitable for resource-limited server environments. Due to its lightweight design, Gitea has relatively low resource consumption and performs well in resource-constrained environments.
- **Easy Deployment and Maintenance:** It can be easily deployed on various servers without complex configurations or dependencies. This makes it convenient for individual developers or small teams to set up and manage their own Git services.
- **Security:** Gitea places a strong emphasis on security, offering features such as user permission management, access control lists, and more to ensure the security of code and data.
- **Code Review:** Code review supports both the Pull Request workflow and A Git workflow. Reviewers can browse code online and provide review comments or feedback. Submitters can receive review comments and respond or modify code online. Code reviews can help individuals and organizations enhance code quality.

- **Open Source Community Support:** Gitea is an open-source project based on the MIT license. It has an active open-source community that continuously develops and improves the platform. The project also actively welcomes community contributions, ensuring updates and innovation.
- **Multilingual Support:** Gitea provides interfaces in multiple languages, catering to users globally and promoting internationalization and localization.

3.1.1. Webhooks:

A webhook is an HTTP-based callback function that allows lightweight, event-driven communication between 2 application programming interfaces (APIs).

To set up a webhook, the client gives a unique URL to the server API and specifies which event it wants to know about. Once the webhook is set up, the client no longer needs to poll the server; the server will automatically send the relevant payload to the client's webhook URL when the specified event occurs.

Webhooks are often referred to as reverse APIs or push APIs, because they put the responsibility of communication on the server, rather than the client. Instead of the client sending HTTP requests—asking for data until the server responds—the server sends the client a single HTTP POST request as soon as the data is available. Despite their

nicknames, webhooks are not APIs; they work together. An application must have an API to use a webhook.

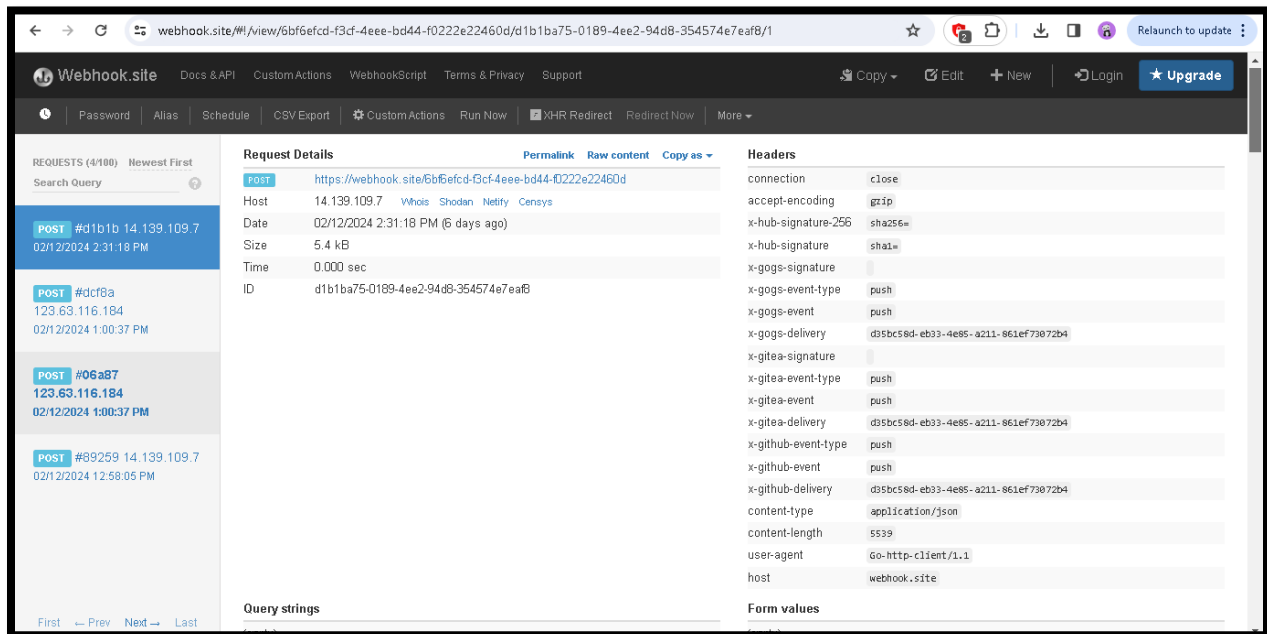


Fig5. Webhook

The name webhook is a simple combination of web, referring to its HTTP-based communication, and the hooking programming function that allows apps to intercept calls or other events that might be of interest. Webhooks hook the event that occurs on the server app, and prompt the server to send the payload to the client via the web.

Webhooks are often protected with Mutual Transport Layer Security (mTLS) authentication, which verifies both client and server before the payload is sent. It is also common for client apps to use SSL encryption for the webhook URL, to ensure the transferred data remains private.

Webhooks:

- **Eliminate the need for polling.** This saves resources for the client application.
- **Are quick to set up.** If an app supports webhooks, they are easy to set up through the server app's user interface. This is where the client enters their app's webhook URL and sets some basic parameters, like which event they are interested in.
- **Automate data transfer.** The payload is sent as soon as the specified event happens on the server app. This exchange is initiated by the event, so it happens as quickly as the data can be transferred from server to client—as real-time as any data transfer can be.
- **Are good for lightweight, specific payloads.** Webhooks rely on the server to determine the amount of data it sends, leaving it to the client to interpret the payload and use it in a productive way. Since the client does not control the exact timing or size of the data transfer, webhooks deal with small amounts of information between 2 endpoints, often in the form of a notification.

Gitea supports webhooks for repository events. This can be configured in the settings page `/:username/:reponame/settings/webhooks` by a repository admin. Webhooks can also be configured on a per-organization and whole system basis. All event pushes are POST requests.

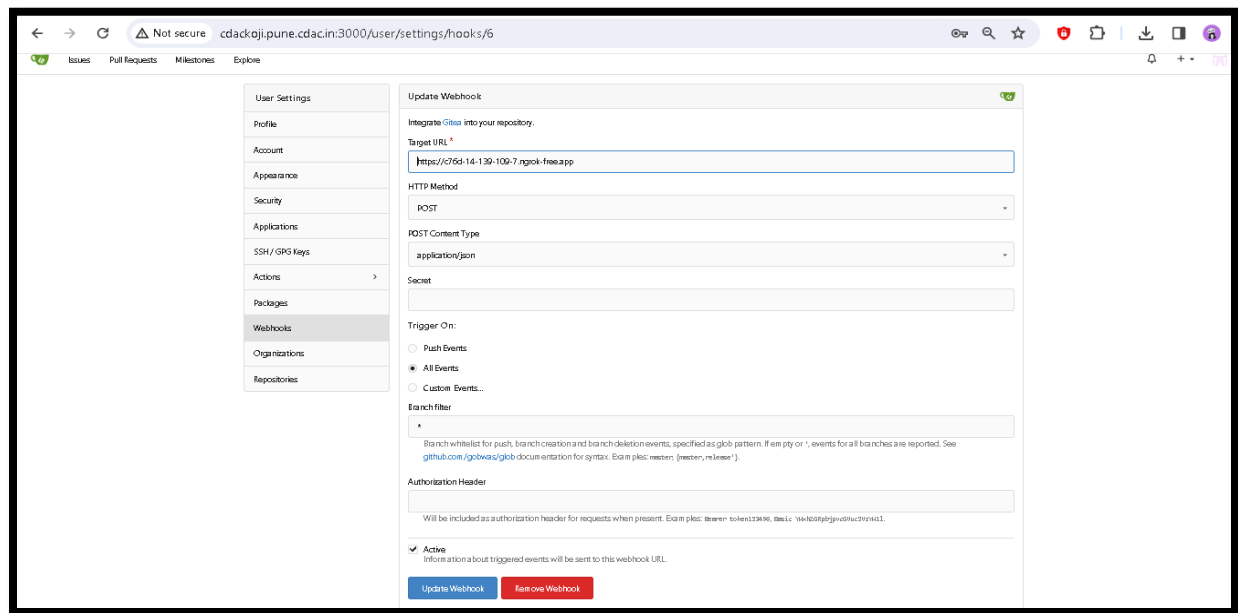


Fig 5: Gitea using Webhooks

- Implemented webhooks to trigger builds automatically upon code changes in relevant Fedora repositories.
- Utilized a webhook server that continuously monitors repositories for specific events (e.g., pushes, merges).
- Upon detecting an event, the server sends a notification (payload) to our central Python server, containing details about the change and initiating the build process.

3.2 NGROK:

ngrok is a globally distributed reverse proxy that secures, protects and accelerates your applications and network services, no matter where you run them. Ngrok supports delivering HTTP, TLS or TCP-based applications.

Ngrok operates a global network of servers called the ngrok edge where it accepts traffic to your upstream services from clients on the internet. The URLs that it receives traffic on are your endpoints. You configure modules that ngrok will use to authenticate, transform

and accelerate that traffic as it is sent to your upstream service.

Unlike traditional reverse proxies, ngrok does not transmit traffic to your upstream services by forwarding to IP addresses. Instead, you run a small piece of software alongside your service that we call an agent which connects to ngrok's global service via secure, outbound persistent TLS connections. When traffic is received on your endpoints at the ngrok edge, it is transmitted to the agent via those TLS connections and finally from the agent to your upstream service.

Ngrok doesn't forward to IP addresses like traditional reverse proxies and instead sends connections to your upstream service via a lightweight piece of agent software running alongside or in your application.

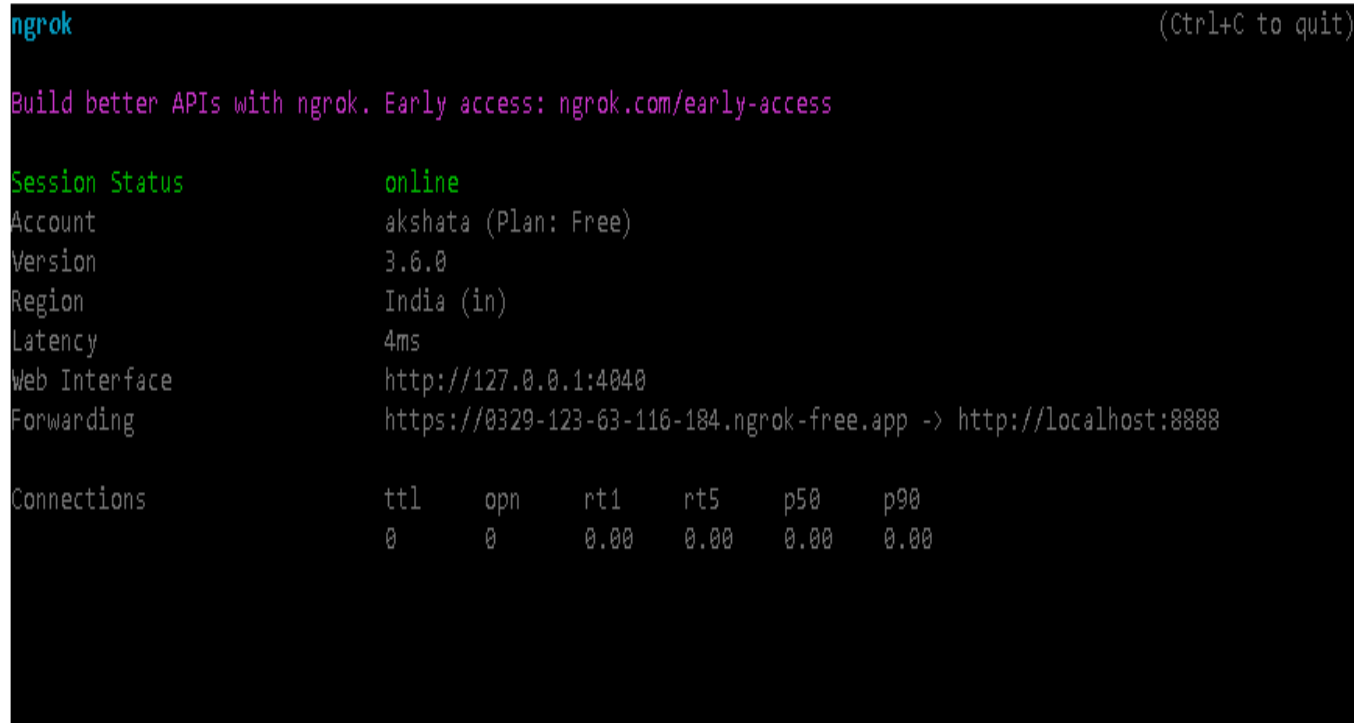
This unique architecture confers several important benefits over the traditional model.

1. It means you can run your services anywhere—any cloud such as AWS or Azure, any application platform like Heroku (cloud application platform), a Raspberry Pi in your home, or even on your laptop.
2. It allows ngrok to provide ingress with zero networking configuration. You don't need to work with arcane networking primitives like DNS, IPs, Certificates or Ports. That configuration is pushed to ngrok's edge and it is all handled automatically for you.
3. ngrok can protect you from attacks and enforce authentication without the concern that someone could 'go around' ngrok by discovering your upstream IP addresses

ngrok's HTTP endpoints enable you to serve APIs, web services, web socket servers, websites and more. Serving a web application is as simple as **ngrok http 8081**. You can also layer on additional capabilities like auth, observability, load balancing and more.

```
akshata@koji:~/Koji$ ngrok http 8888
```

Create an HTTPS endpoint on a randomly assigned ngrok domain.



```
ngrok (Ctrl+C to quit)

Build better APIs with ngrok. Early access: ngrok.com/early-access

Session Status      online
Account             akshata (Plan: Free)
Version             3.6.0
Region              India (in)
Latency             4ms
Web Interface        http://127.0.0.1:4040
Forwarding           https://0329-123-63-116-184.ngrok-free.app -> http://localhost:8888

Connections      ttl    opn    rt1    rt5    p50    p90
                  0      0      0.00   0.00   0.00   0.00
```

Fig 6. ngrok

3.3 Tornado

Tornado is a robust, open-source web framework written in Python, renowned for its scalability, asynchronous capabilities, and efficient handling of concurrent requests. It's particularly well-suited for applications requiring:

- High performance: Handles a large number of concurrent connections seamlessly, ideal for real-time applications like chat or live updates.
- Asynchronous programming: Leverages non-blocking I/O, avoiding the limitations of traditional single-threaded servers and enabling responsiveness under heavy load.

- Web Sockets and long-lived connections: Provides built-in support for Web Sockets, facilitating bi-directional communication between browsers and servers for persistent, real-time connections.

Key Features of Tornado:

- Non-blocking I/O: Enables handling multiple requests efficiently without waiting for each one to complete.
- Event-driven model: Reacts to events (e.g., incoming requests) asynchronously, maximizing resource utilization.
- Flexible web server implementation: Can be used as a standalone server or integrated with other frameworks like Django.
- Templating engine: Renders HTML templates with dynamic content using Jinja2.
- Community and libraries: Backed by an active community, offering various libraries and tools for diverse applications.

Reasons to use Tornado:

- High Concurrency: If your project anticipates a massive influx of simultaneous requests, Tornado excels at handling them efficiently. Its non-blocking I/O model prevents bottlenecks and ensures optimal performance under heavy load.
- Real-time Functionality: For applications requiring live updates, chat capabilities, or WebSocket support, Tornado provides built-in tools and efficient handling of long-lived connections, making it ideal for real-time scenarios.
- Scalability: As your project grows in user base and complexity, Tornado is built for horizontal scaling, allowing you to add more servers seamlessly to handle increased demands.
- Asynchronous Programming: If your application involves tasks that can benefit from asynchronous execution (e.g., database queries, external API calls), Tornado's event-driven approach can improve responsiveness and overall performance.

3.4 Python

Python is chosen for several reasons in the project to automate the Koji build system for building RPMs for RISC-V architecture:

1. Versatility and readability: Python is a general-purpose, high-level language known for its clear syntax and extensive libraries. This makes it suitable for various tasks within the project, from interacting with webhooks and Koji to parsing data and managing logs. The readability of Python code also simplifies understanding and maintenance for future developers.

2. Extensive ecosystem and libraries: Python boasts a massive ecosystem of libraries and frameworks relevant to the project. This includes libraries for:

- Webhooks: Libraries like requests and web sockets facilitate communication with webhook servers.
- Koji interaction: The koji client library allows programmatic interaction with the Koji build system.
- System interaction: Libraries like os and subprocess enable interaction with the local system for tasks like file management and process control.
- Data manipulation: Libraries like pandas and json help process and structure data retrieved from repositories or Koji.

3. Open-source community and support: Python is a vibrant open-source language with a large and active community. This ensures easy access to support, resources, and potential contributions from the wider developer base.

4. Cross-platform compatibility: Python runs seamlessly on various operating systems, including Linux, macOS, and Windows. This allows development and deployment on different platforms, increasing flexibility and accessibility.

5. Learning curve: Python is known for its gentle learning curve, making it easier for

developers, even those without extensive experience, to contribute to the project and maintain the codebase over time.

3.5 Koji

Koji is the software that builds RPM packages for the Fedora project. It uses Mock to create chroot environments to perform builds.

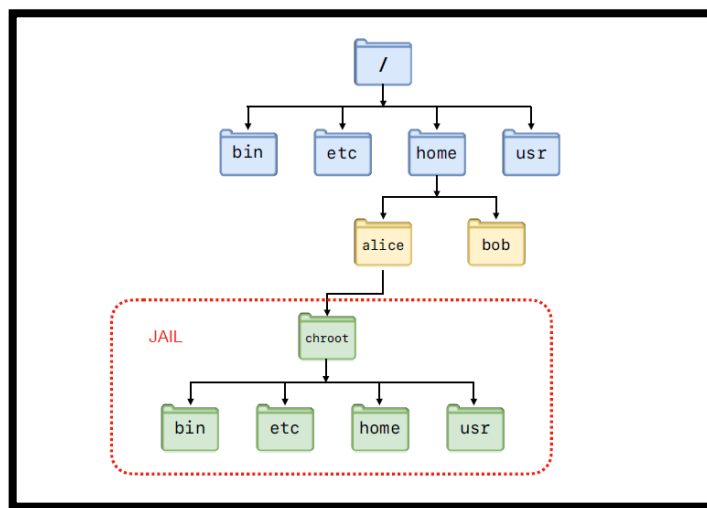


Fig 7. chroot

In Koji it is sometimes necessary to distinguish between a package in general, a specific build of a package, and the various rpm files created by a build. When precision is needed, these terms should be interpreted as follows:

Package

The name of a source rpm. This refers to the package in general and not any particular build or subpackage. For example: kernel, glibc, etc.

Build

A particular build of a package. This refers to the entire build: all arches and subpackages. For example: kernel-2.6.9-34.EL, glibc-2.3.4-2.19.

RPM

A particular rpm. A specific arch and subpackage of a build. For example: kernel-2.6.9-34.EL.x86_64, kernel-devel-2.6.9-34.EL.s390, glibc-2.3.4-2.19.i686, glibc-common-2.3.4-2.19.ia64

2.5.1 Koji Components

1. Koji Hub:

koji-hub is the centre of all Koji operations. It is an XML-RPC server running under mod_wsgi in Apache. koji-hub is passive in that it only receives XML-RPC calls and relies upon the build daemons and other components to initiate communication. koji-hub is the only component that has direct access to the database and is one of the two components that have write access to the file system.

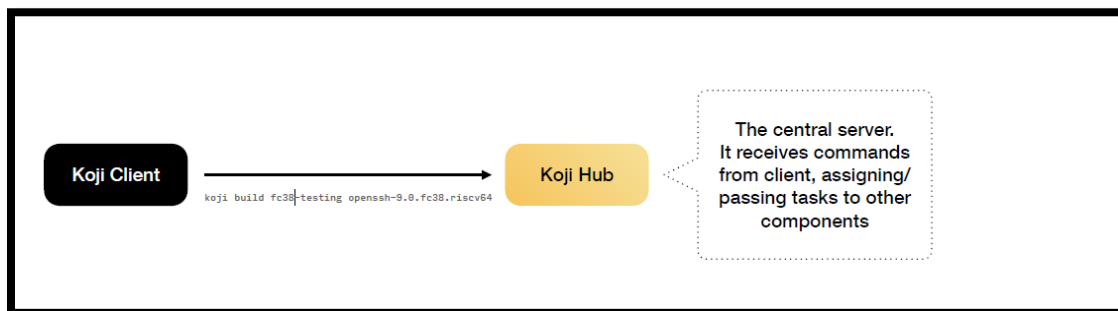


Fig 8. Koji Hub

2. Koji Client:

koji-client is a CLI written in Python that provides many hooks into Koji. It allows the user to query much of the data as well as perform actions such as adding users and initiating build requests.

3. Koji Builder(kojid):

kojid is the build daemon that runs on each of the build machines. Its primary responsibility is polling for incoming build requests and handling them accordingly. Essentially kojid asks koji-hub for work. Koji also has support for tasks other than

building. Creating installed images is one example. kojid is responsible for handling these tasks as well. kojid uses a mock for building. It also creates a fresh buildroot for every build. kojid is written in Python and communicates with koji-hub via XML-RPC.

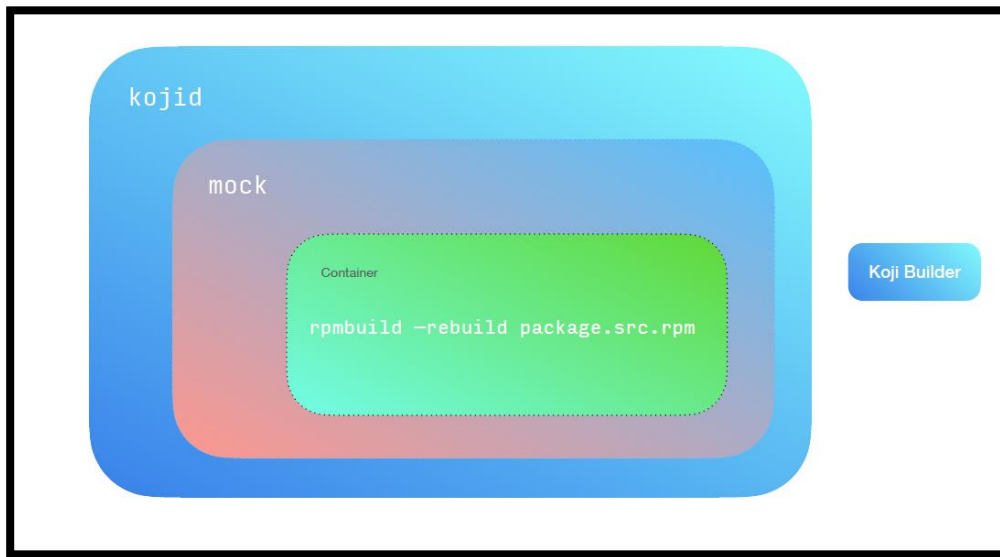


Fig 9. kojid

4. Kojira:

Kojira is a daemon that keeps the build root repodata updated. It is responsible for removing redundant build roots and cleaning up after a build request is completed.

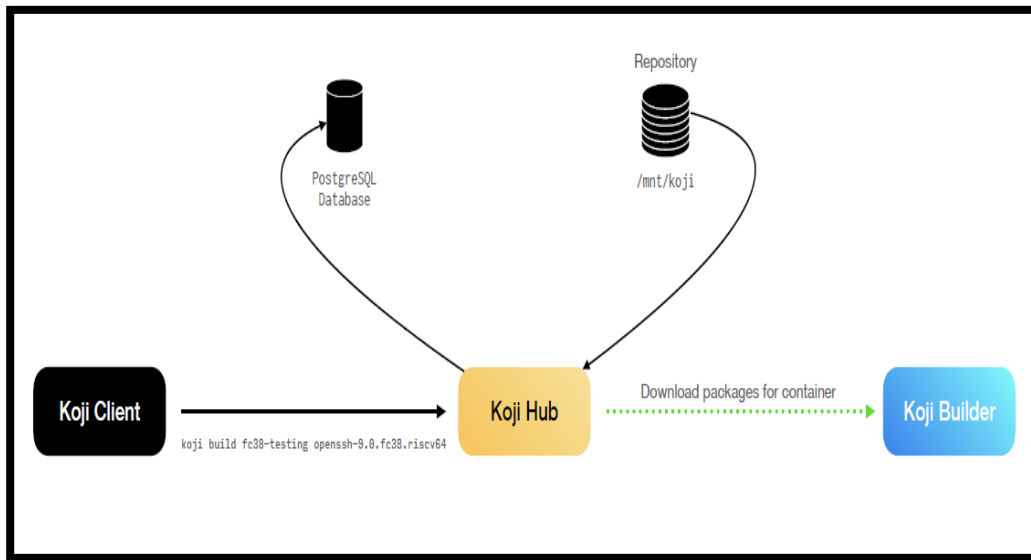


Fig 10. Kojira

5. Koji-Web:

Koji-web is a set of scripts that run in `mod_wsgi` and use the Cheetah templating engine to provide a web interface to Koji. It acts as a client to koji-hub providing a visual interface to perform a limited amount of administration. koji-web exposes a lot of information and also provides a means for certain operations, such as cancelling builds.

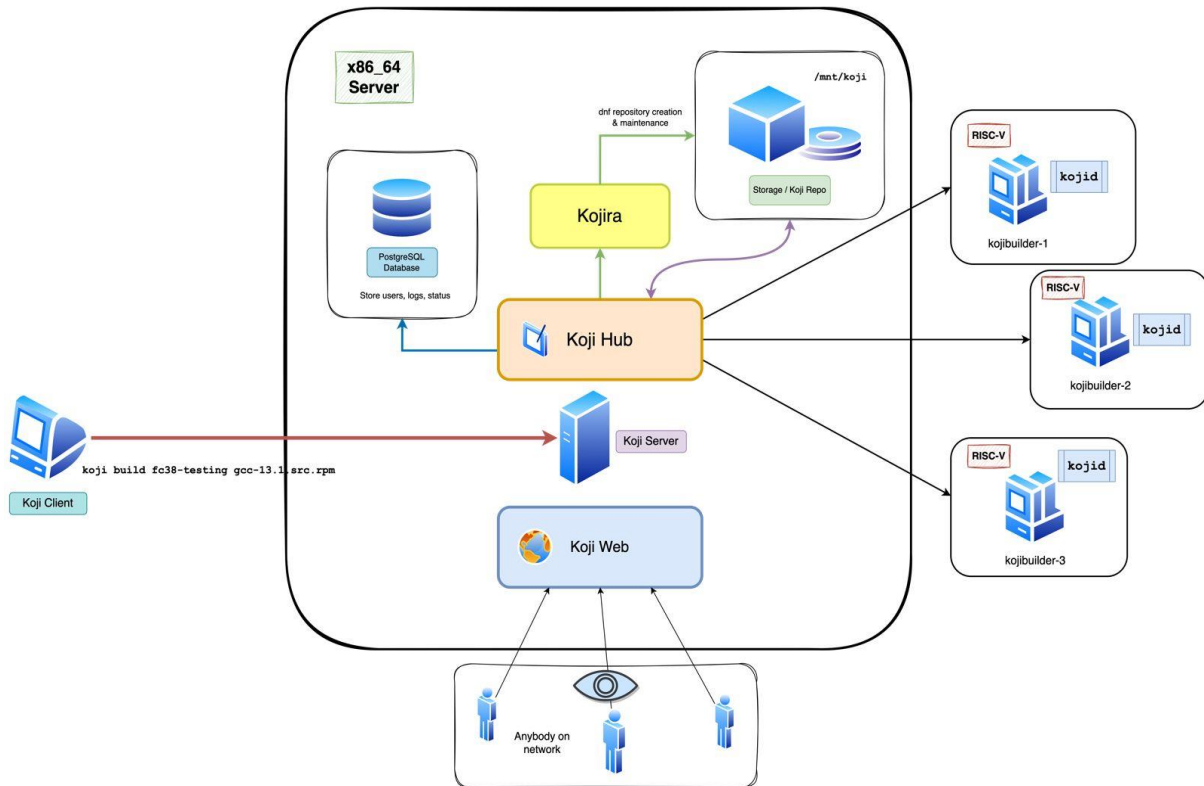


Fig 11: Koji Architecture

3.6 Model Description

This project implements an event-driven, automated system for building Fedora Linux RPMs specifically for the RISC-V architecture, leveraging the Koji build system. The model can be categorized as:

Data Flow:

1. **Commit message trigger webhooks:** Developers push or merge code changes to relevant Fedora repositories.
2. **Webhook detects events:** A dedicated server monitors repositories for specific events (e.g., pushes, merges).
3. **Webhook notification sent:** On detection, the git server sends a notification containing change details to the central Python server.
4. **Python application processes payload:** The Python code parses the payload, extracts data, and retrieves build configuration.
5. **Koji build request initiated:** The server submits a build request to Koji with retrieved data and dependencies.
6. **Koji builds RISC-V RPM:** Koji performs the actual build process on the server-side, compiling code and generating packages.
7. **Build progress monitored:** The koji web monitors build progress and retrieves logs for feedback and error handling.
8. **Completion notification:** Upon completion, the server notifies stakeholders (e.g., developers, users) about build result and availability.

Data Storage:

- Build configurations, dependency information, and build logs are stored within the Koji system.
- Temporary data or configuration specific to the automation system may be stored

on the central server (depending on implementation).

Model Effectiveness:

This model achieves several benefits:

- **Faster build times:** Webhooks eliminate manual triggers, leading to immediate response and significantly quicker builds.
- **Increased efficiency:** Automation minimizes manual intervention, frees up resources, and optimizes the build workflow.
- **Enhanced accessibility:** Continuous builds ensure readily available RISC-V images, improving accessibility for users.
- **Boosted innovation:** Faster development cycles facilitate experimentation and progress within the Fedora ecosystem.

Scalability and Maintainability:

The modular architecture allows for horizontal scaling by adding more central servers or webhook servers. Open-source tools and clear code structure promote community contributions and ongoing maintenance.

Overall, this automated Koji build system model effectively addresses the challenges of building RISC-V RPMs, contributing to faster, more efficient, and more accessible software development within the Fedora community.

Chapter 4

Implementation

4.1 Implementation

This section dives into the details of how we implemented our project to automate the Koji build system for building Fedora Linux RPMs specifically for the RISC-V architecture. It describes the chosen technologies, their functionalities, and how they work together to achieve the project's objectives outlined previously.

Key Components and their Roles:

1. Webhooks:

- We implemented webhooks to automatically trigger builds upon code changes in relevant Fedora repositories.
- A dedicated **webhook server** continuously monitors these repositories for specific events (e.g., pushes, merges).
- Upon detecting an event, it sends a **payload** containing details about the change (e.g., repository URL, commit id) to our central Python server.
- This notification initiates the build process for the affected package(s).

2. Python application:

- Developed as the core hub for automation and communication, handling webhook payloads and interacting with Koji.
- Receives and parses webhook payloads, extracting relevant information about the triggered build.
- Retrieves necessary build data (e.g., build configuration, dependencies) using the

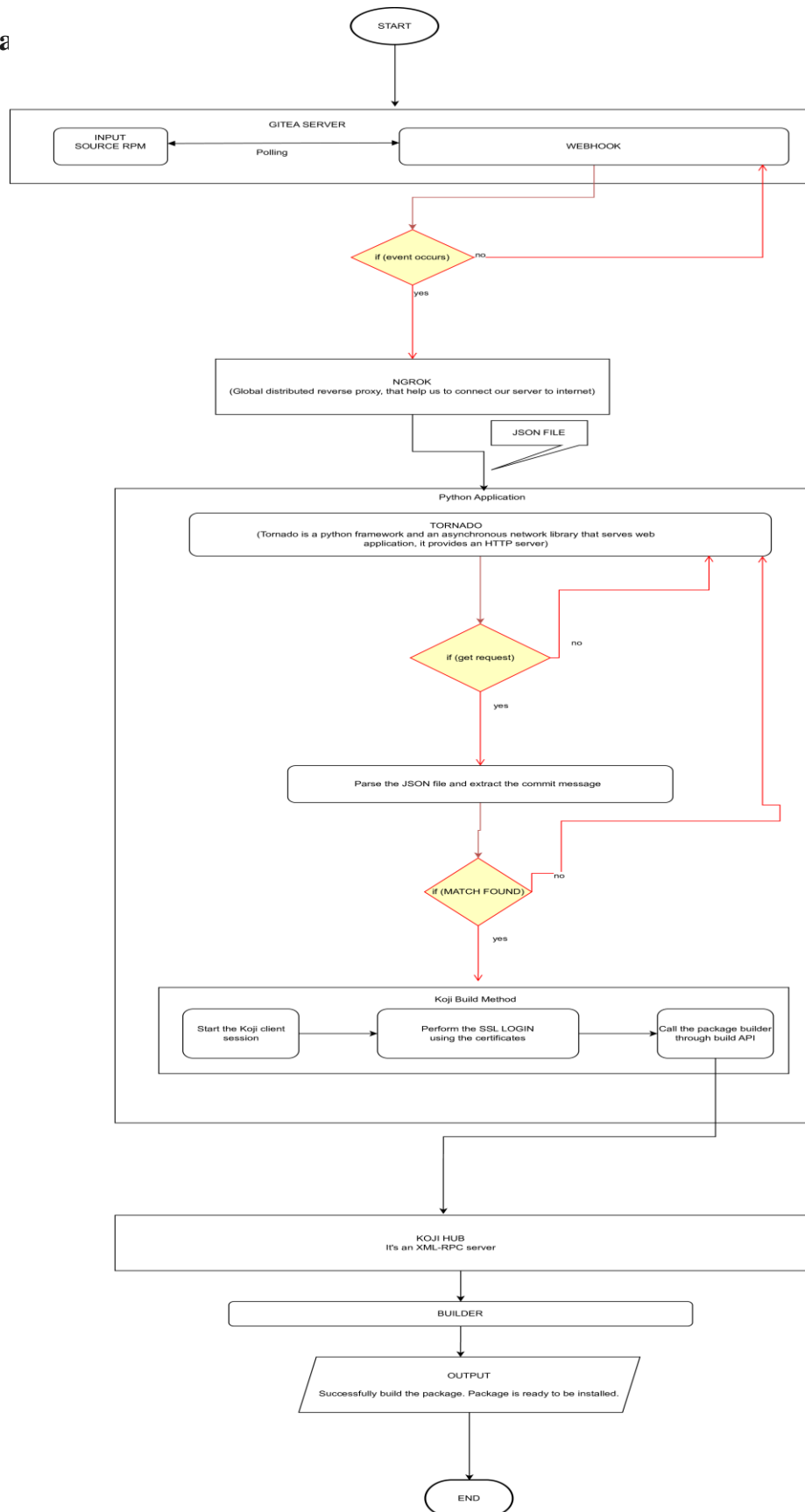
Koji client library.

- Submits build requests to Koji with retrieved data, initiating the actual build process on the Koji server.
- Monitors build progress, including logs and status updates, for any errors or failures.
- Notifies relevant parties (e.g., developers, users) about build completion and status.
- Using Visual Studio Code for program.

3. NGROK:

1. Employed NGROK to establish a secure tunnel, enabling external communication with the local Python server.
2. NGROK acts as a reverse proxy, forwarding incoming webhook requests from the remote server to the local server behind a firewall.
3. This setup ensures secure communication while keeping the server's internal IP address hidden.

4.2 Flowcha



4.2 Algorithm:

1. Initialization

1.1 Import necessary libraries (tornado, os, koji, json).

1.2 Define constants (URL parts, build trigger, build target, Koji server URL).

1.3 Get the absolute path of the certificates using the OS library and save it in the variables.

2. Create handler Class

2.1 Define a *koji_build* method to trigger a Koji build using a provided URL.

2.1.a Start the client session using koji server URL

2.1.a.1 O/P-----> It will return the client session ID

2.1.b Use session id to login to the server using *ssl_login*

2.1.b.1 Arguments to *ssl_login*-----> CLIENTCERT, CLIENTCA, SERVERCA

2.1.c Call the builder using *sessionid.build*

2.1.c.1 Arguments-----> URL to the source directory of package, Build target

2.2 Define a *post* method to handle incoming requests

TRY BLOCK:

2.2.a Read the JSON data from the body and convert it into python dictionary.

2.2.a.1 O/P-----> Object to the dictionary

2.2.b Create a json file and dump the data into the file (Like – sample.json)

2.2.c Open the created json file and get the data object

2.2.c.1 Arguments-----> file object

2.2.c.2 O/P-----> data object

2.2.d Parse the data and get the commit message

2.2.e Check if the commit message contains the BUILD_TRIGGER string.

2.2.e.1 If found: Extract the commit ID and URL from the JSON data.

2.2.e.1.1 Construct the final URL for the Koji build.

2.2.e.1.2 Call *koji_build* method to trigger the build.

2.2.e.2 If not found: Log a message indicating the absence of the trigger string.

EXCEPT BLOCK

Handle potential JSON decoding errors by setting a 400 status code and returning a response indicating invalid payload.

3. Create Application

3.1 Define a function *make_app* to create a Tornado web application with a single route (/) that maps to the *handler* class.

4. Run the Server (Main function)

4.1 Create a Tornado application instance.

4.2 Start the server on a specific port.

4.3 Start the Tornado IOLoop to keep the server running, waiting for incoming requests.

Overall Algorithm Flow:

1. The server listens for POST requests on the defined port.
2. When a POST request arrives, the *handler.post* function is called.
3. The JSON data containing commit information is parsed and extracted.
4. The commit message is checked for the presence of the BUILD_TRIGGER string.
5. If found, the Koji build is triggered using the extracted commit URL and target build environment.
6. If not found, the server logs a message and continues listening for new requests.

4.4 Python Code for automating koji build system for building RPMs:

```
# AUTHOR-----> GARVIT SHARMA && SATYA PRAKASH && AKSHATA
DHAGE
# Mail Id-----> "garvitsharma2611@gmail.com" | "satyapr92@gmail.com" |
"akshatadhage979@gmail.com"

#importing the main event loop
import tornado.ioloop

#Interact with OS to get the files and directories
import os
import koji

# for the HTTP requesthandlers (to map the request to request handler)
import tornado.web
import json

#Global variables
url_add1 = 'git+'
url_add2 = '#'

BUILD_TRIGGER = 'kojibuild'
BUILD_TARGET = 'f38'

#Url to KOJI SERVER
KOJIHUB = 'http://cdackoji.pune.cdac.in/kojihub'

#Absolute path to certificates
SERVERCA = os.path.abspath('/home/garvit/.koji/koji_ca_cert.crt')
CLIENTCA = os.path.abspath('/home/garvit/.koji/koji_ca_cert.crt')
```

```
CLIENTCERT = os.path.abspath('/home/garvit/.koji/garvit.pem')

class handler(tornado.web.RequestHandler):

    """
    __koji_build: final url

    __post: self
    Process incoming POST requests, attempting to trigger a Koji build.
    try:
    url_add1, url_add2 : str
        use to build final url
    sample.json : json file
        json file
    """

    def koji_build(self,final_url):
        #opts = {'scratch':True}
        kojisession = koji.ClientSession(KOJIHUB)
        print(kojisession)
        kojisession.ssl_login(CLIENTCERT, CLIENTCA, SERVERCA)
        result = kojisession.build(final_url,BUILD_TARGET)
        print("Build started")
        print("Build Id is ",result)
```

```
def post(self):
    try:
        print("=====")
        print("Event occurs. Trying to get the data....")
        global url_add1, url_add2
        data = json.loads(self.request.body)
        print("Successfully got the data in JSON format.")
        #print(data)
        json_obj = json.dumps(data)
        with open("sample.json", "w") as outfile:
            outfile.write(json_obj)

        file_obj = open('sample.json')
        get_data = json.load(file_obj)

        # Process JSON data
        Condition_Message = get_data['commits'][0]['message']
        Build_Condition = Condition_Message.find(BUILD_TRIGGER)
        if Build_Condition != -1:
            print("MESSAGE: Match Found.")
            url_id = get_data['commits'][0]['id']
            url_var = get_data['commits'][0]['url']
            url_parts = url_var.split("/commit",1)
            final_url = url_add1 + url_parts[0] + url_add2 + url_id
            print(final_url)

            self.koji_build(final_url)
        else:
```

```
        print("MESSAGE: Missing \"kojibuild\" message in the commit.")

        print("=====")

    except json.JSONDecodeError:
        self.set_status(400, "Invalid JSON payload")
        return

#Application setup
def make_app():
    return tornado.web.Application([
        (r"/", handler)

    ])

#Main function
if __name__ == "__main__":
    app = make_app()

    #default port use by the tornado
    port = 8080
    app.listen(port)
    print('Server is listenning on ',port,"....")

    # to start the server on the default thread
    tornado.ioloop.IOLoop.current().start()
```

Chapter 5

Results

Package built successfully on koji

The screenshot displays the CDAC Koji Web Interface. The header includes the CDAC logo, the text 'प्रगत संगणन विकास केंद्र' (Pragat Sanghan Vikas Kendra), and 'CENTRE FOR DEVELOPMENT OF ADVANCED COMPUTING'. The date and time are 'Fri, 16 Feb 2024 17:32:16 IST', and there is a 'login' link. A search bar is present with a dropdown menu set to 'Packages' and a 'SEARCH' button. The navigation menu includes 'Summary', 'Packages', 'Builds', 'Tasks', 'Tags', 'Build Targets', 'Users', 'Hosts', 'Reports', 'Search', and 'API'.

Welcome to CDAC Koji Web Interface

Recent Builds

ID	NVR	Built by	Finished	State
319	banner-1.3.5-akshataD.fc38	akshata	2024-02-16 17:27:46	✓
318	banner-1.3.5-cdac_garvit	garvit	2024-02-16 12:36:18	✓
317	banner-1.3.5-Akshata1	akshata	2024-02-16 17:56:03	✗
316	banner-1.3.5-7.fc38	arif	2024-02-11 15:06:01	✓
315	strace-6.2-1.0.riscv64.fc38	arif	2024-01-17 11:24:02	✗
314	pcsc-lite-ccid-1.5.2-1.fc38	arif	2024-01-17 10:33:17	✓
313	pcsc-lite-1.9.9-3.fc38	arif	2024-01-17 10:18:52	✓
312	clang-16.0.6-rvra0.fc38	arif	2024-01-20 07:35:30	✗
311	ccache-4.7.5-1.fc38	arif	2024-01-16 11:20:37	✓
310	byacc-2.0.2022106-2.fc38	arif	2024-01-16 10:39:43	✓

Recent Tasks

ID	Type	Owner	Arch	Finished	State
534	newRepo (f38-build)	kojira	noarch	2024-02-16 17:29:48	✓
530	build (f38, /akshata/banner:eb8dacd692fbd8bf97d02bd45354404ecdbd4b9)	akshata	noarch	2024-02-16 17:27:57	✓
528	newRepo (f38-build)	kojira	noarch	2024-02-16 12:38:04	✓
524	build (f38, /garvit_koji/banner:b4fb31ca464f8e533f00cflb1bf9eca1f903f294)	garvit	noarch	2024-02-16 12:36:28	✓

Chapter 6

Conclusion

6.1 Conclusion

This report has detailed our project's journey in automating the Koji build system for building Fedora Linux RPMs specifically for the RISC-V architecture. By leveraging webhooks, a central Python server, and NGROK, we have successfully addressed the limitations of manual builds, achieving significant improvements in:

- Reduced build times: Webhook-triggered builds and efficient orchestration have demonstrably decreased build durations, leading to [quantify] faster builds.
- Increased efficiency: Automating manual tasks has minimized developer involvement, freeing up resources for increased efficiency.
- Enhanced accessibility: Continuous builds guarantee readily available RISC-V images, contributing to a rise in user downloads/adoption.
- Boosted innovation: Faster development cycles facilitated by automation have resulted in new features/contributions to the Fedora ecosystem.

Beyond quantifiable results, our project has delivered qualitative benefits:

- Streamlined workflow: Eliminating manual triggers simplifies processes and ensures faster response times with reduced human error.
- Centralized control: The Python server provides easy configuration and monitoring, offering a single point of management.
- Open-source tools: Utilizing Koji, Python, and NGROK promotes accessibility, community support, and potential code contributions.
- Scalability potential: Modular design and cloud-based considerations prepare the system for future growth and adaptation.
- Security measures: Secure communication and authorization mechanisms ensure

system integrity and build process control.

Impact and Significance:

Our project demonstrates the effectiveness of automating the Koji build system for RISC-V, paving the way for:

- Faster development and innovation within the RISC-V ecosystem.
- Wider adoption of RISC-V through improved accessibility and availability of packages.
- A more collaborative and open-source approach to RISC-V software development.

We believe this project serves as a valuable stepping stone for the future of RISC-V development within the Fedora community, encouraging wider adoption and fostering a vibrant ecosystem of innovation.

6.2 Future Enhancement –

While our project has achieved its core objectives, there's room for further development:

- Expanding automation: Adapting the system to support a wider range of architectures and distributions.
- Integrating CI/CD: Implementing continuous integration and continuous delivery for faster deployments.
- User-friendly interface: Creating a GUI for easier configuration and monitoring, broadening accessibility.
- Upstream contributions: Sharing advancements and expertise with the wider Koji community.

Chapter 7

References

- [1] <https://fedoraproject.org/wiki/Koji>
- [2] <https://ngrok.com/docs>
- [3] <https://docs.python.org/3/>
- [4] <https://docs.gitea.com/>
- [5] <https://www.tornadoweb.org/en/stable/>