Raspberry Pi 5 RTSP YOLO Pipeline with Hailo8L

Project Plan: pi-live-detect-rstp - A Modular Python Detection RTSP Pipeline for Raspberry Pi 5 and Hailo-8L

Introduction

The rapid growth of AI-powered edge devices is transforming the landscape of real-time video analytics, with single-board computers like the Raspberry Pi 5 playing a pivotal role. This project, **pi-live-detect-rstp**, aims to provide a robust, modular Python-based application capable of probing, processing, detecting, annotating, and restreaming RTSP video feeds using the computational power of a Raspberry Pi 5 (8GB) paired with a Hailo-8L AI Hat. Modern use cases such as smart surveillance, automated monitoring, and embedded AI vision require not only high inference speed but also modularity, reliability, and streamlined management.

This plan outlines a highly modular pipeline architecture implemented in discrete, independent services. Each service will interact via an in-memory database/cache, facilitating both efficient state tracking and rapid access to intermediate data. Detection is performed using YOLO optimized for Hailo hardware through TensorFlow, while video data can be readily annotated and restreamed-all managed by accessible, restartable services. The system is designed with a heavy emphasis on automation, maintainability, and developer/operator usability, featuring a sudo-aware install script, complete service management, detailed logging, and thoroughly documented GitHub materials.

Throughout this report, current best practices and up-to-date technical details are incorporated, including important insights from the recent Core Electronics guide on deploying custom YOLO scripts on the Raspberry Pi AI Hat[2]. The structure is carefully partitioned for clarity and depth, covering all necessary hardware, software, service, and documentation considerations.

System Overview

Before delving into detailed architecture, here's a succinct table summarizing all proposed services, their purposes, default configuration, and key runtime parameters.

Service	Function	Default Config	Configurable Options
Probe	RTSP stream availability check & stream metadata	rtsp://192.168.100.4:8554/stream	RTSP URL, probe interval, timeout
Collector	Frame capture & caching in memory	640x480 @ 30fps	Frame resolution, frame rate, prefetch buffer size

Detector	YOLO-based detection with Hailo-8L, TensorFlow	YOLOv5/YOLOnano, Hailo acceleration	Model path, threshold, batch size, device selection
Annotator	Overlay bounding boxes, labels, meta info	Output image: 640x480	Font, colors, overlay verbosity, object confidence threshold
Restreamer	RTSP serving of annotated video	8555/stream (e.g., rtsp://...:8555/)	Stream URL, server port, latency/buffer
Cache Service	In-memory (RAM) shared state & data store	LRU with TTLs, up to N frames	Max objects, eviction policy, TTL per entry
Logging	Rolling text file logs for daemons; console+file	Logs dir on OS drive, debug mode true	Log format, rotation policy, log level
Install tools	Sudo-aware bash scripts for install, verify, clean	Defaults to RPi 5 + AI HAT+ settings	Non-root/test mode, debug output, custom service enablement
ServiceMgmt	Status dump, reload, restart, meta info	/run/pi-live-detect.json or similar	Output format, extra diagnostic detail

Each table entry will be elaborated via dedicated, in-depth paragraphs in the subsequent sections, discussing both functional scope and implementation rationale.

Raspberry Pi 5 and Hailo-8L AI Hardware Integration

Raspberry Pi 5 (8GB Model)

At the heart of this system is the Raspberry Pi 5, equipped with 8GB LPDDR4X RAM and a 2.4 GHz quad-core Arm Cortex-A76 CPU[4][5]. This model brings a significant performance leap over its predecessor, especially vital for on-device video processing. With dual 4-lane MIPI camera connectors, Gigabit networking, two USB 3.0 and two USB 2.0 ports, as well as a PCIe 2.0 x1 interface, the Pi 5 is uniquely equipped to handle both high input/output rates and AI offload demands.
Thermal management is critical. For stable extended operation-particularly with an attached PI AI HAT (Hailo-8L)-use the official active cooler or a quality heatsink/fan combo. The board should be powered using the recommended 5V/5A USB-C PD supply to guarantee sufficient power for the HAT+ camera combination.

Hailo-8L AI Hat Integration

The Hailo-8L is a 13-TOPS NPU designed for efficient edge inference, ideally paired with the Raspberry Pi AI HAT or AI HAT+ via PCIe. Hardware integration steps are as follows:

- Attach the Hailo-8L AI Hat to the Pi 5 via the PCIe interface (using an M.2 HAT, official Raspberry Pi AI Kit, or compatible third-party expansion board[7]).

- Fit a quality passive or active heatsink to the AI module for sustained performance.
- If using a camera, verify compatibility (official Camera Module 3 recommended) and proper cable alignment.
- Adjust any enclosure or cable routing to preserve airflow for both CPU and AI Hat.

The Hailo-8L driver and firmware must be installed through the official hailo-all package, which brings in the kernel module, firmware, runtime, and associated libraries[7]. After installation and reboot, confirm operational status using hailortcli fw-control identify.
Careful consideration must be given both to software and hardware compatibility: The Hailo-8L requires models to be compiled specifically for its architecture, and conversion can only be performed on an x86_64 Linux machine using the Dataflow Compiler (DFC), not directly on the Pi [9][10]. Models and .hef files must be transferred to the Pi for inference; this process is vital for performant, error-free detection.

RTSP Probing Service

The **Probe Service** is responsible for continuously checking the availability of an RTSP stream using a configurable URL. Its main functions are to:

- Detect whether the configured RTSP stream endpoint is accessible
- Extract stream metadata (e.g., resolution, codec, frame rate)
- Update shared cache/state with latest probe results
- Initiate error logging or notifications on failure

For implementation, the service leverages GStreamer's rtspsrc element for robust RTSP protocol handling. GStreamer is highly reliable in production for both client- and server-side RTSP/RTP streaming and provides granular control over timeouts, retries, latency, and buffer settings[12].
For continuous probing, the service can construct a lightweight pipeline that connects to the stream, attempts minimal frame retrieval or RTSP OPTIONS/DESCRIBE requests, and interprets the resulting GStreamer bus messages for error or success states.
Key probe parameters (such as probe interval, timeout, and stream URL) should be externally configurable to suit various deployment environments. A good default probe interval is 10 seconds, with a timeout threshold of 2-5 seconds. Detected stream loss or protocol failures should be clearly logged and written to the in-memory cache with error details.
Multiple web guides-including community forums and Stack Overflow threads-illustrate how one can "ping" a RTSP endpoint via GStreamer and catch pipeline state transitions or error signals programmatically[12]. The probe can be constructed as a Python service using gi.repository.Gst, allowing smooth integration with the rest of the application.

Frame Collection and Buffering Service

The **Collection/Buffering Service** is dedicated to frame acquisition from the RTSP stream and temporary storage in an in-memory data structure.

Key Features:

- Decouples real-time frame acquisition from downstream processing, reducing dropped frames during detection or annotation delays.
- Uses a prefetch ring buffer (LRU or FIFO) in shared RAM, configurable in size (e.g., 30-120 frames).
- Frames are stored as numpy arrays or OpenCV-compatible buffers for pipeline compatibility.
- Implements optional frame downscaling (default resolution 640x480 at ~30 fps) on capture to meet real-time constraints.
- Service status (capturing, paused, dropped frames, latest timestamp) is written to the cache for downstream querying.

For best results, the collector should use GStreamer again, this time with rtspsrc feeding into a decoder and a sink that allows direct access to raw frames. Community code samples from GitHub and forums show how to set up such pipelines in Python (for example, using appsink or integrating with OpenCV)[11].

Multithreading or multiprocessing is advisable: frame capture should run in its own process/thread, writing to the in-memory DB, while detection or annotation services fetch the latest frame as required. Buffer overflows/dropped frames and timing metrics should be logged. The service must be robust against temporary stream drops. On loss, it should flag the cache and attempt graceful reconnection, logging each outage interval.

Detection Service (YOLO on Hailo-8L)

Function: Run real-time object detection on captured frames using YOLO models optimized for Hailo-8L hardware.

Model Selection and Compilation

The recommended workflow is:

1. **Model Selection:** Use a YOLO version optimized for the Hailo-8L's resource constraints, such as YOLOv5nano/v8nano or custom-trained smaller models.
2. **Model Conversion:** Convert an ONNX (or PyTorch) YOLO model into Hailo's .hef inferencing format via the Dataflow Compiler (required to run on x86_64 Linux with ample RAM)[8][10].
3. **Deployment:** Transfer the .hef file to /usr/local/share/hailo or a project directory on the Pi, then load with Hailo's Python SDK at runtime.

Multiple web tutorials demonstrate how to carry out ONNX-to-Hailo conversions, emphasizing the need to balance model size (for RAM and compute constraints) and accuracy for your application[9].

Pipeline Design

Upon detecting a new frame in the buffer cache, the detection service should:

- Retrieve the most recent (or next unprocessed) frame.

- Run inference using the pyHailoRT library on the Hailo-8L, collecting bounding boxes, class IDs, and confidences.
- Store detection results (metadata and/or JSON) associated with the frame in the cache.
- Log processing time, confidence threshold, and error statistics.

Batch processing is discouraged unless latency may be tolerated; for real-time single-stream feeds on Pi 5, per-frame inference is generally manageable.
Integration examples and custom scripts from the Core Electronics guide, as well as the official Hailo sample pipelines, are directly applicable to this detection stage. Toggleable debug output should provide verbose detection stats both in logs and, if enabled, in the annotated frame overlay[2].
Downstream services can query the cache for new detections as soon as the detection service completes its work for each frame.

Annotation Service

Purpose: Efficiently render bounding boxes, class names, and confidence scores onto video frames.
OpenCV is the best toolkit for this purpose-drawing boxes, text, and other overlays per detection result is straightforward and fast for 640x480 video[14]. YOLO detections yield bounding box coordinates and labels as standard outputs, which can be fed into OpenCV drawing routines.
Key considerations for the annotation overlay:

- Select contrast-friendly colors for different classes/object types and sufficient line thickness for visibility.
- Overlay informative text (object label, confidence, optional meta fields like frame timestamp).
- Allow customizable meta display (full JSON, minimal labels, timestamp watermark, etc.).
- Render to a new frame buffer (never modify the original in case of cascading failures).
- Cache and index the annotated frame for immediate restreaming and, optionally, archival.

Drawing complexity should be balanced with performance: elaborate overlays (e.g., alpha blending, non-rectangular regions) may reduce overall pipeline FPS. For advanced AR-style overlays, reference SmartOverlays techniques to ensure occlusion and readability criteria[15]. However, for 640x480 analytics applications, standard bounding boxes with minimal overlays are efficient and effective.

RTSP Restreaming Service

Task: Restream annotated frames as a new RTSP video feed, enabling downstream viewing/recording by third-party clients (VLC, web interfaces, NVRs).
Design Approach:

- Use GStreamer and the gst-rtsp-server library/module, which enables the implementation of a featureful RTSP server with a minimal Python wrapper[16][18].
- Serve frames from the annotated buffer in real time, using a configurable output URL (default rtsp://<pi-ip>:8555/stream).
- Set latency/jitter buffer parameters to optimize for local vs. remote clients. For single-viewer, low-latency setups (common with Pi+Hailo hardware), keep the buffer low.
- Periodically report streaming health, dropped frame statistics, and connected client count via logs.

Community Python projects (e.g., Frigate, GStreamer-Python samples) illustrate how to implement such a restreamer using GStreamer's Python interface, with explicit control over how frames are received from a cache or in-memory queue[13].
For best practice, the restreamer can also provide meta endpoints (e.g., a JSON-based REST status endpoint or a secondary channel with detection metadata, indexed to frame timestamps). RTSP restreaming can be highly sensitive to timing drift and frame drops, especially on busy networks-log all errors and monitor client disconnects for operational visibility.

In-Memory Database and Caching System

To link all services, a **shared in-memory cache/database** is required. This data store will maintain:

- Probe results and stream status
- Raw frame buffers
- Detection results (object metadata, per-frame)
- Annotated frames (as compressed images or buffer references)
- RTSP server/client state (connections, statuses)

Implementation Choices

1. **Python Dict (thread/process safe):**
   - Fastest option for same-process data sharing, possibly protected by threading.Lock or a multiprocessing-safe variant.
   - With separate processes, use Python’s multiprocessing.Manager or similar, but watch for IPC overhead.
2. **Redis In-Memory Store:**
   - Offers cross-process and even cross-host caching, fast TTL-based eviction, and data expiration support.
   - Suitable if more scaling, robustness, or interoperability with web endpoints is desired[19].
3. **cachetools or Custom TTLCache:**

- For less complex use cases, leverage Python's cachetools with TTL and LRU policies[21].
- Easiest for prototyping; easy disk cache extension.

4. **sqlite3 in-memory mode:**
   - For relational/dataframe access patterns, Python's builtin sqlite3 can be instantiated with :memory: target, with trivial setup[22].
   - Ideal if structured queries or advanced search/indexing features are needed.

For this system, a hybrid is recommended: Python-native dicts in a central cache manager process (or thread), with optional Redis backend for distributed scenarios or multicore optimization. Data retention should be size/TTL limited (e.g., 60 raw frames, 60 annotated, and metadata by unique frame ID and timestamp).
Each service should access the cache via a well-defined interface; for fail-safe operation, cache sanity checks and data serializations (e.g., numpy array encoding/decoding) must be implemented.

Inter-Service Coordination & Modular Microservice Design

Adopting a microservice-inspired model brings several benefits: isolation, reusability, resilience, and ease of scaling or extension[24][25].
Design Principles:

- Each pipeline stage (Prob, Collector, Detection, Annotation, Restream) is self-contained.
- Service discovery via the in-memory cache (e.g., each service checks current state before acting).
- Heartbeats and status flags maintained in cache, monitored by service management scripts.
- Well-defined hot-reload/restart logic for each module (e.g., detector reloads model on config change, restreamer picks up new annotated frame).
- Lightweight API endpoints for status checks, optional REST API for health/status queries.
- Debug mode always enabled by default, with verbose but filtered logs.

Python's asyncio (for async pipelines) or multiprocessing/threading (for parallel operations) are both viable. FastAPI or Flask can be used for lightweight service APIs or status dump endpoints, though pure Python scripts are sufficient for core tasks.
If desired, each service can be Dockerized for isolated deployment-even on resource-constrained Pi hardware, lightweight containers are possible using tools like Podman.

Logging Practices

A robust and developer-friendly logging strategy is essential for maintainability and debugging, especially in a multi-service setup.
Standards:

- All daemons/services write logs to rotating text files on the OS drive /var/log/pi-live-detect/ (example path).
- Standalone or invoked scripts log both to text (file) and terminal/console for developer convenience.
- **Debug mode** enabled by default; log level can be overridden via config/environment variable.
- Unified log format for consistency (%(asctime)s - %(service)s - %(level)s - %(message)s).

Python's logging module provides a powerful, flexible logger-allowing for multiple handlers (file/console) and granular control[27][28]. Both rolling file handlers and stdout/console handlers can be chained, with separate verbosity if required.
Logs should include:

- Service start/stop status
- Key operations (stream probe up/down, frame rate, dropped frames, detection results)
- Errors/exceptions with stacktraces
- Resource use (RAM/CPU, connection counts)
- Periodic heartbeat/status

Best practice: one log file per service to simplify parsing and troubleshooting. Rotation policy (e.g., 10MB, 5 files per service) prevents disk exhaustion.

Install, Verification, Uninstall, and Service Management Scripts

Sudo-Aware Install Script

The install script is a crucial part of developer and operator experience. Its features must include:

- Detect and check for prerequisites: Python 3.9+, pip, virtualenv, GStreamer, OpenCV, required Python packages, Hailo drivers, and firmware[2].
- Automated installation of system packages using apt, with graceful handling if dependencies are already installed.
- Deploy all Python virtual environments, install requirements, and set up directories (logs, cache, config).
- Create and install systemd unit files for each long-running service.
- Enable and start services, with suitable permissions (using sudo or checking for root where required).
- Output a summary of all installed components and their versions.
- Accept CLI flags: --no-root for dry runs, --force to overwrite configs, --debug for verbose output.

Security best practices-such as only using sudo for installation and service management steps and not throughout the entire script, validating each command, and ensuring script file permissions-must be observed[30][31].

Verify Install Script

- Checks presence and versions of all Python packages and system binaries.
- Verifies camera and RTSP availability, Hailo driver/firmware, and that all systemd services are active/running.
- Pings RTSP restream endpoint and optionally inspects video frames/detections.
- Dumps configuration and environment to a file for support.

Uninstall Script

- Stops and disables all services.
- Removes virtual environments, config, and log files if requested.
- Optionally purges system Python packages (if installed via script).
- Removes all installed binaries/scripts from PATH.
- Reports completion/errors.

Service Status Dump

- Simple CLI or script that aggregates health/status for all services (e.g., via cache keys or via systemctl status).
- Outputs summary stats (frame rates, errors, dropped frames, active connections).
- Optionally dumps detection stats or a recent detection history sample.

Scripts should output in human-readable format and also JSON for scriptability/integration.

GitHub Documentation and Project Presentation

README as a Detailed Manual

- **Intro:** Explain use, architecture, hardware/software requirements, key features.
- **Installation:** Step-by-step with caveats for various OS/RPi setups. Highlight AI HAT steps, model compilation.
- **Usage:** Configuration file format, how to start/stop services, how to custom-tune RTSP/YOLO/config.
- **Developer Guide:** Building new service modules, cache API, unit tests, model re-compilation/transfer.

- **Troubleshooting:** Common camera/RTSP/model errors and recovery.
- **Images & Diagrams:** Use GitHub-Markdown-embedded images for hardware layouts and pipeline diagrams[2].

Follow best practices for clarity and completeness, as advised by industry guides, Google's oss style, and community wisdom[33][2][35].

Changelog with Git-Friendly Versioning

- Use Keep A Changelog (KAC) format, sorted latest-first, linked to Git tags and release branches, with clear date/version metadata[37][38].
- Log all notable changes (features, fixes, breaking changes).
- File must be at top-level as CHANGELOG.md.

Project Structure and Code Practices

- Modular, maintainable directory structure: /src, /scripts, /config, /docs, /tests.
- Include requirements.txt and environment.yml.
- Prefer Poetry or pipenv if advanced dependency management is desired.

Integration of Core Electronics YOLO Guide and Community Insights

The Core Electronics guide on deploying object detection with custom Python scripts on Pi + Hailo-8L is directly relevant. Key takeaways to integrate:

- Hardware setup with correct camera module, cable, and cooling.
- HAT driver/firmware installation and post-install status verification using official utilities.
- Best practices for Python-based pipeline construction: modular scripts, explicit camera/video interface, and clean error handling.
- Use of prebuilt detection pipelines as reference for custom service construction.
- Emphasis on debugging each stage (camera, detection, overlay, restream) independently before full pipeline deployment.
- Insights into real-world performance tuning: frame rate vs. detection accuracy trade-off, handling low-light/complex scene scenarios by pre-annotating training data or switching models as necessary[2][35].

Additionally, community experiences with GStreamer tweaking, RTSP compatibility, and troubleshooting (e.g., tuning camera modules, handling frame rate drops, resolving GStreamer plugin errors) will inform robust error handling in both the install scripts and runtime logging[12].

Future Extensions and Considerations

The architecture described is designed for modularity and future expansion. Notable avenues for extension include:

- **Multi-Stream Processing:** Extending collector/service logic to handle multiple incoming RTSP streams or multiple camera modules in parallel, each with independent detection and restreaming pipelines.
- **Web Dashboard Integration:** Exposing live video feeds and detection metadata to a browser dashboard via WebRTC or HTTP endpoints, with additional controls for service management and configuration reloads.
- **Hardware Optimization:** Supporting alternate accelerators (e.g., Coral, NCS2) via modular hardware abstraction.
- **Persistent Caching:** Adding disk-backed persistent caches for long-term analytic or forensic purposes, with background expiry for storage efficiency.
- **On-the-fly Model Updates:** Enabling hot-swapping or live upgrades of detection models with minimal downtime in the detection service.
- **Notifications and Integrations:** Adding MQTT or webhook notifications for detection events, allowing integration with home automation or security systems.
- **Kubernetes or Containers:** Containerizing services for distributed deployment or orchestrated service restarts.

Conclusion

This project plan lays out a comprehensive, production-oriented design for a modular detection restreaming pipeline, tailored to the capabilities and constraints of the Raspberry Pi 5 (8GB) with a Hailo-8L AI Hat. Drawing on recent advances, best practices, and authoritative tutorials, each pipeline component-probe, collection, detection, annotation, restreaming, caching, installation, and logging-is specified in depth and tightly integrated for maintainability and scalability. Strong documentation and git integration, detailed service management, and a robust logging/debugging framework enable both rapid deployment and long-term operation. The use of in-memory caching and microservice patterns ensures performance and modularity. This project can serve as both a turnkey solution for smart RTSP analytics and a flexible foundation for custom AI video applications at the edge.

By following this plan, users and developers can confidently deploy, monitor, and extend a cutting-edge computer vision application on affordable but powerful hardware, merging the best of community insight, modern software design, and AI acceleration.

References (38)

1. *YOLO Object Detection on the Raspberry Pi AI Hat+* . https://core-electronics.com.au/guides/yolo-object-detection-on-the-raspberry-pi-ai-hat-writing-custom-python/
2. *Published January 2025 - Raspberry Pi 5*. https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf
3. *Raspberry Pi 5 8GB - PiShop*. https://www.pishop.co.za/store/raspberry-pi-5-model-b-8gb
4. *hailo-rpi5-examples/doc/install-raspberry-pi5.md at main - GitHub*. https://github.com/hailo-ai/hailo-rpi5-examples/blob/main/doc/install-raspberry-pi5.md
5. *GitHub - seapanda0/hailo-yolo-guide: Guide to Convert and Inference ….* https://github.com/seapanda0/hailo-yolo-guide
6. *Yolo full model convertion - General - Hailo Community*. https://community.hailo.ai/t/yolo-full-model-convertion/7456
7. *Detect rtsp stream before playing using gstreamer*. https://stackoverflow.com/questions/39327831/detect-rtsp-stream-before-playing-using-gstreamer
8. *RTP and RTSP support - GStreamer*. https://gstreamer.freedesktop.org/documentation/additional/rtp.html
9. *Convert ONNX Models to Hailo8L: Step-by-Step Guide Using Hailo SDK*. https://www.ridgerun.ai/post/convert-onnx-model-to-hailo8l
10. *Annotating Images Using OpenCV - LearnOpenCV*. https://learnopencv.com/annotating-images-using-opencv/
11. *SmartOverlays: A Visual Saliency Driven Label Placement for Intelligent ….* https://openaccess.thecvf.com/content_ICCVW_2019/papers/OpenEDS/Hegde_SmartOverlays_A_Visual_Salier Computer_ICCVW_2019_paper.pdf
12. *GitHub - GStreamer/gst-rtsp-server: RTSP server based on GStreamer ….* https://github.com/GStreamer/gst-rtsp-server
13. *GStreamer-Python - GitHub*. https://github.com/sahilparekh/GStreamer-Python
14. *rtsp stream*. https://gstreamer.freedesktop.org/documentation/gst-rtsp-server/rtsp-stream.html
15. *Best Practices for Implementing Redis Caching in Python*. https://anadata.com/best-practices-for-implementing-redis-caching-in-python/
16. *Caching database data in python - Stack Overflow*. https://stackoverflow.com/questions/9972351/caching-database-data-in-python
17. *Microservices Architecture with Python* . https://techifysolutions.com/blog/microservices-based-architecture-with-python-deep-dive/
18. *u-ways/fastapi-microservice-py - GitHub*. https://github.com/u-ways/fastapi-microservice-py
19. *How to Log Python Messages to Both stdout and Files*. https://www.geeksforgeeks.org/python/how-to-log-python-messages-to-both-stdout-and-files/

20. *logger configuration to log to file and print to stdout*. https://stackoverflow.com/questions/13733552/logger-configuration-to-log-to-file-and-print-to-stdout
21. *Best practices on using sudo in a bash script - Ask Ubuntu*. https://askubuntu.com/questions/939583/best-practices-on-using-sudo-in-a-bash-script
22. *Automation with Bash - Creating a Script to Install and Configure ...*. https://dev.to/devopsking/automation-with-bash-creating-a-script-to-install-and-configure-applications-on-multiple-flavours-of-os-4o0k
23. *Write changelogs for humans. A style guide. - GitHub*. https://github.com/vweevers/common-changelog
24. *READMEs - styleguide*. https://google.github.io/styleguide/docguide/READMEs.html
25. *Setup YOLO Object Detection using the Raspberry Pi AI HAT*. https://www.geeky-gadgets.com/raspberry-pi-object-detection-guide/
26. *How to connect to SQLite database that resides in the memory using ...*. https://www.geeksforgeeks.org/python/how-to-connect-to-sqlite-database-that-resides-in-the-memory-using-python/
27. *Keep A Changelog Format* . https://openchangelog.com/docs/getting-started/keep-a-changelog/