LogicVein

LogicVein Playbooks

Practical Use Cases & Automation Recipes

A hands-on guide to automating network operations with Playbook workflows

Table of Contents

1 Introduction

This guide provides practical, real-world examples of how to use LogicVein's Playbooks to automate common network operations tasks. While the User Manual covers basic operations, this document focuses on combining nodes into end-to-end workflows that you can adapt to your own environment.

Each use case includes a process overview, step-by-step node configuration, and tips for handling errors and notifications.

1.1 Revision History

Version	Last Modified	Revision Details
1.0	2026-03-30	First edition for NA

2 Playbook Nodes Overview

In the LogicVein products, a “playbook” is a visual workflow built by connecting nodes with connector lines. Each node represents a discrete action. Nodes execute in sequence, with conditional branches directing the flow based on command results, regex matches, or error states.

For a complete reference of available node types, see the User Manual for netLD or ThirdEye.

User Manuals can be found here: https://logicvein.com/manual/#user-manuals.

NOTE: The Incident node and related features described in Section 3.4 are available only in ThirdEye Suite.

3 Use Cases

This section provides examples of various use cases for playbooks. Use these examples to learn how playbooks work and adapt these examples to your real-world requirements.

3.1 Device OS/Firmware Updates

Updating device firmware is one of the most error-prone tasks in network operations – multiple steps, long wait times, and the risk of mismatched revisions. A playbook automates the entire procedure and provides consistent error checking and notification at every stage.

3.1.1 Process Overview

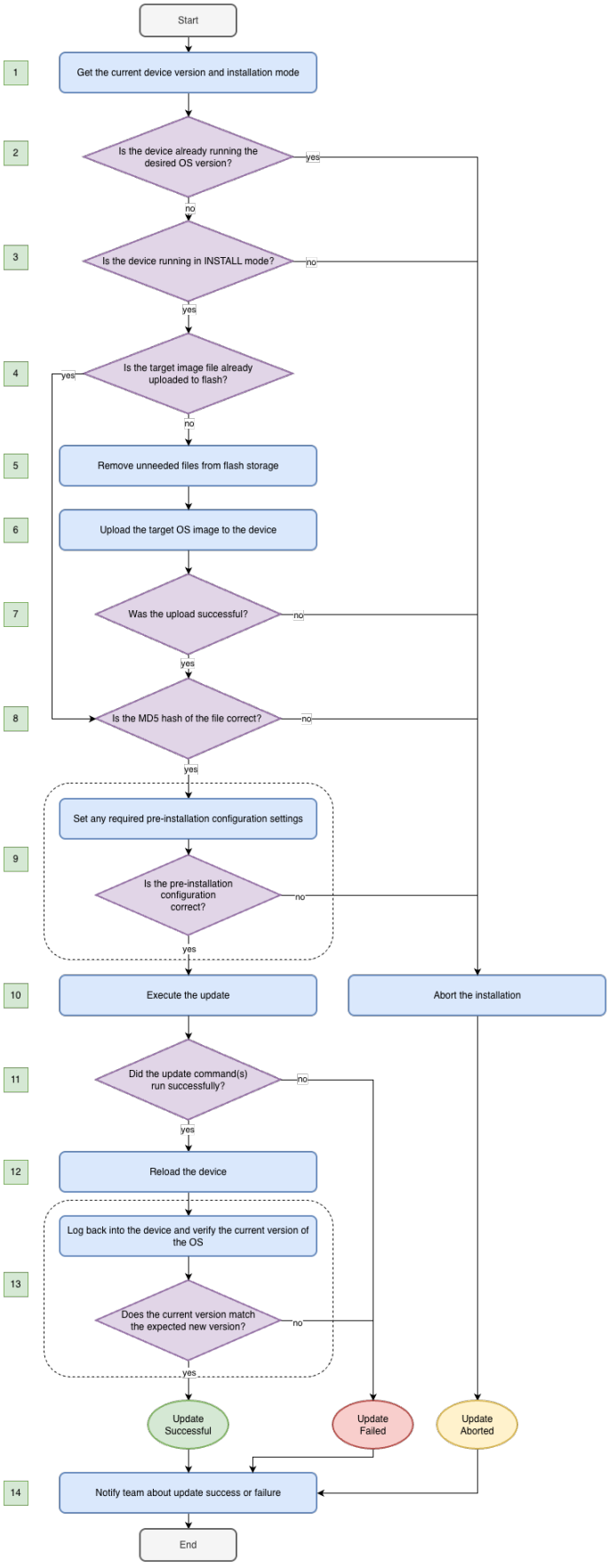
Before building the playbook, write down the steps you would normally follow if you were to perform an OS update manually.

Specific steps will vary based on device vendor, type, make and model. However, an OS update workflow typically follows a variation of the following steps:

1) Get the current version of the device.
2) If the device is already at the target version, do nothing since an update is not needed.
3) If applicable, verify that the device is running in INSTALL mode (vs BUNDLE mode) and abort the update if it is not in the correct mode.
4) If the image file for the target version has already been uploaded to flash, skip to step 8.
5) Ensure there is sufficient space in the device's flash storage for the new software image and remove unneeded files.
6) Upload the OS file to the target device.
7) Ensure that the upload was successful.
8) Verify that the MD5 checksum of the image file matches the expected checksum.
9) Ensure that any required pre-installation configuration settings are in place, such as setting the BOOT variable or configuration register value.
10) Execute the update command(s).
11) Abort if the update command(s) produced an error.
12) Reload the device, if required.
13) Confirm that the device is at the new OS version.
14) Notify your team on whether the device was updated successfully, encountered an error, or if the update was aborted at any previous step.

You may also choose to map out the process in a flowchart, covering the decision points and error-handling paths. An example of a possible flowchart is shown below, with the numbered labels corresponding to each of the steps listed above. Any failure

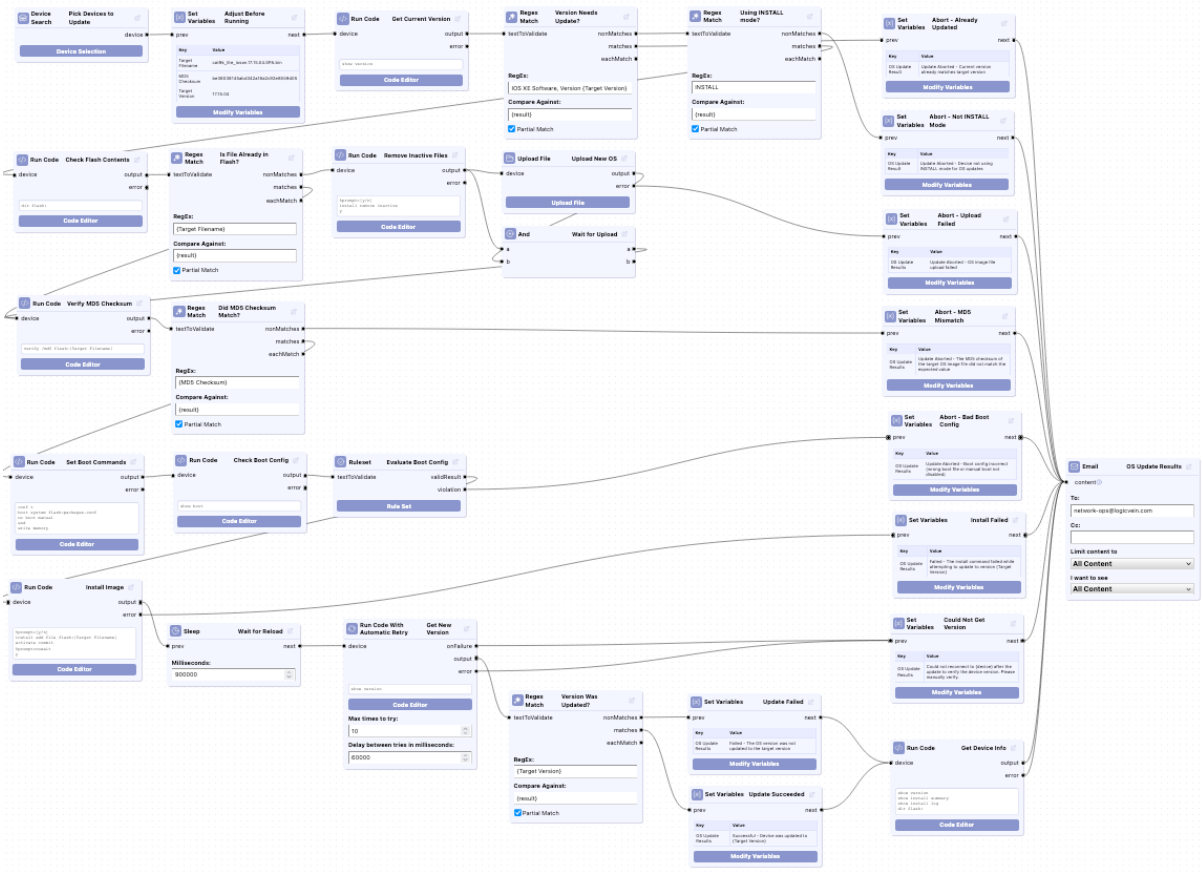
triggers an email notification, so the operator knows immediately which step failed and on the device.



3.1.2 OS Updates

This section provides an example of a working OS update playbook. In this case, the playbook is designed to update Cisco Catalyst 9200 series switches.

First, let’s show the complete playbook, zoomed out.



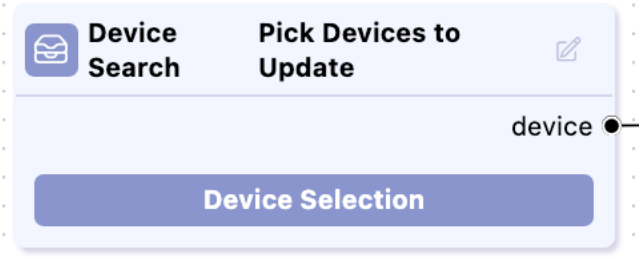
Here's the same playbook, with the steps from our flowchart overlayed over it.

We'll zoom in and break down the individual parts in the sections below.

It is recommended that you read this section sequentially, as some base topics are explained earlier in the section and then assumed to be understood later.

3.1.2.1 Select the Target Device(s)

Start by configuring which devices the playbook will run against using the Device Search node. This occurs before "step 1" from our flowchart.



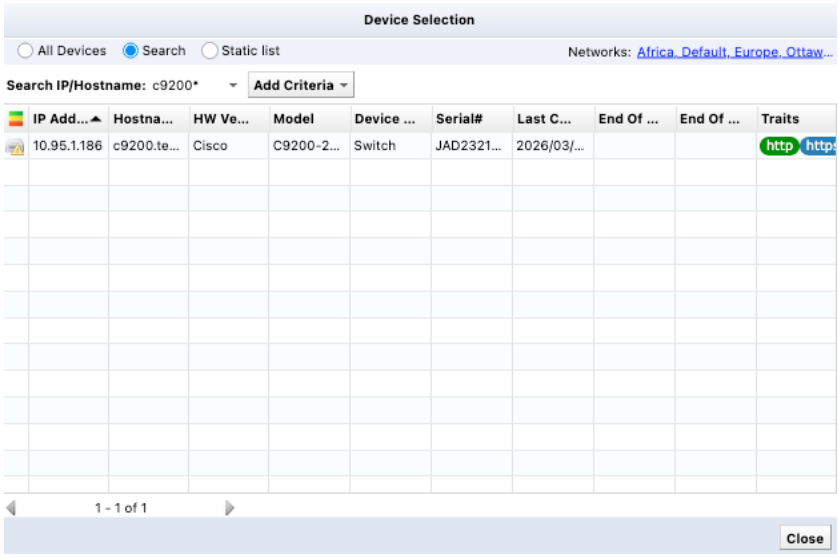
NOTE: The description field in the top right corner of any playbook node is editable. In the example above, the description has been set to "Pick Devices to Update". By default, this description will be populated with a random two-word sequence, like "Empathetic_Yacht". As a best practice, edit this field to give your node a descriptive name for what it's supposed to do. (There is a 25-character limit.)

Click the **Device Selection** button to open the selection window.

There are three selection methods:

All Devices – Targets every device present in the Inventory tab.

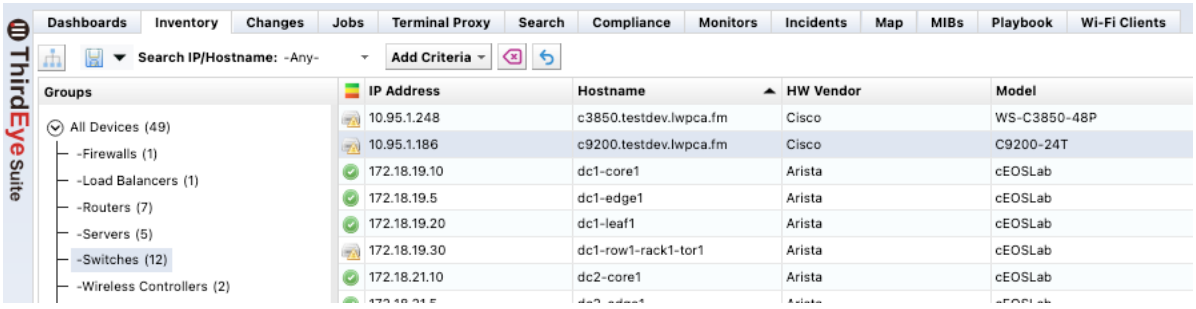
Search – Filters devices by criteria such as vendor, model, version, device type, device name, IP address, etc. This uses the same filters available from the Inventory tab. This is the most common method of device selection.



Static List – Targets specific devices you have pre-selected in the Inventory tab.

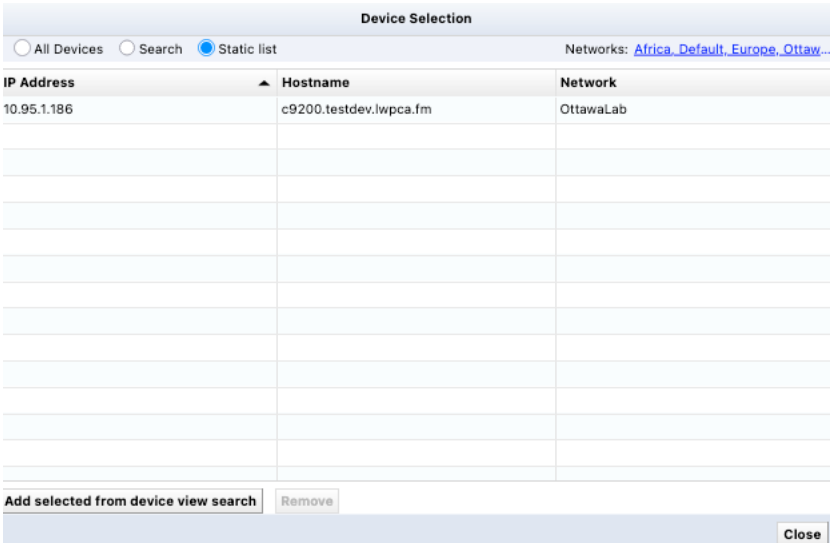
To use a static list:

1) Before editing the playbook, navigate to the Inventory tab and select the devices you want to target (use your keyboard's Ctrl or Command key to multi-select devices if desired).



2) Click on the Playbook tab, edit your playbook, then open the Device Search node and then the "**Static list**" option.

3) Click on the "**Add selected from device view search**" button. The selected devices appear in the node's device list. Click Close.



3.1.2.2 Assign and Set Variables

When possible, rather than hardcoding values that are likely to change over time, create variables using the **Set Variables** node. This occurs before "step 1" from our flowchart, but after the device selection.

This increases the reusability of the playbook and makes it easier to adjust later.

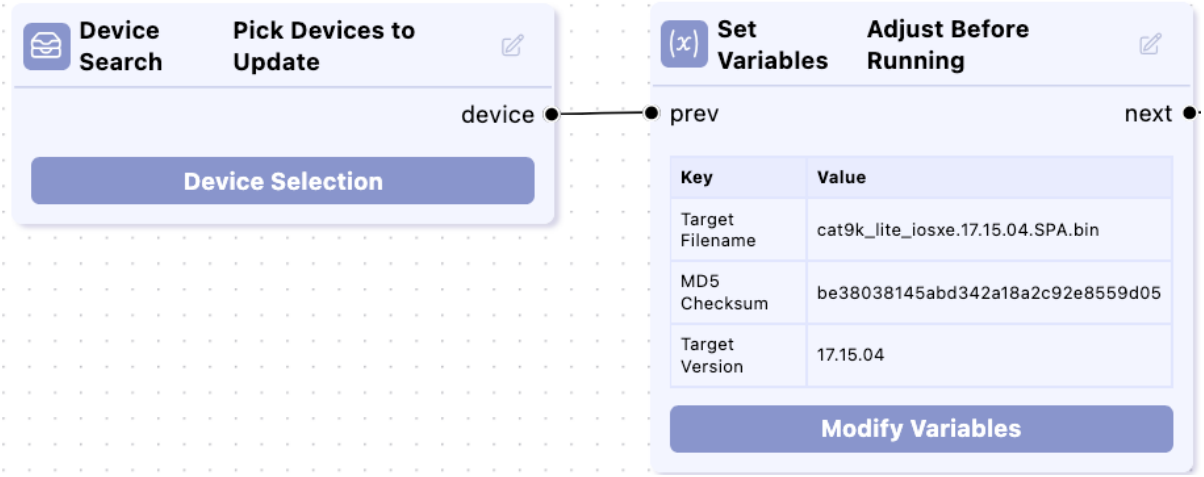
In our OS update playbook, we'll create three variables:

- **Target Filename –** This is name of the image file that will be uploaded to the device.
- **MD5 Checksum –** This is the expected checksum that should be returned when verifying the integrity of the uploaded file. If we get an MD5 checksum mismatch after the upload, this means that the file may have been corrupted during the upload, or we are using the incorrect file.
- **Target Version –** This is the expected version that the device should report after an update has been completed and we run a "show version" command.

Here's how our **Set Variable** node looks in our example playbook.

We'll join this node to the output of our **Device Search** node. This is a logical place to put it as we'll likely want to pick our devices and set our variables in one go.

Here are these two nodes joined together:



3.1.2.3 Check the Current Device Version and the Installation Mode

Next, we want to run some checks to determine if we should proceed with attempting to upgrade the selected device.

This section corresponds to steps 1, 2 and 3 from our flowchart.

In our example OS update playbook, we want to abort the playbook if either of the following things are true:

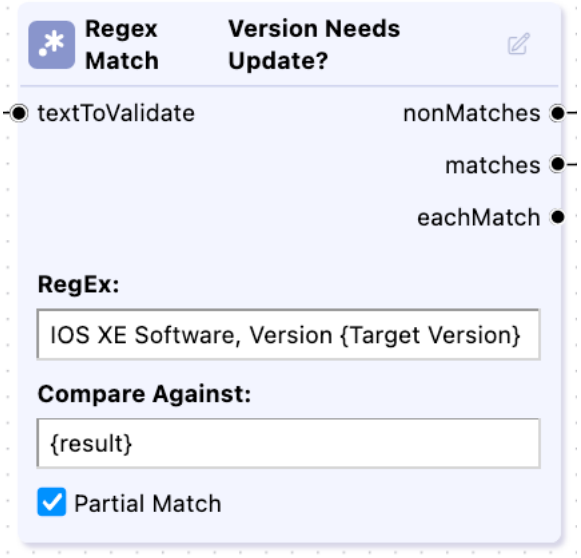
- The device is already at the desired OS version; or
- The device is not configured to perform updates in INSTALL mode.

In our case, both pieces of information can be obtained from the output of the "show version" command. To do this, we place a **Run Code** node directly after the **Set Variables** node we described above. We then instruct the **Run Code** node to execute the "show version" command.

A thing to know about the **Run Code** node is that all command output will be sent out of the "output" element of the node in the form of a variable called **{result}**. The **{result}** variable can then be referenced by downstream nodes, as we'll see next.

From the output of this node, we'll chain together two **Regex Match** nodes. The first one checks whether our **Target Version** variable (that we set in the **Set Variables** node) is found in the **{result}** of the previous "show version" command.

Note that when using any of our variables from the **Set Variables** node in other nodes, we must surround the variable name with curly braces.

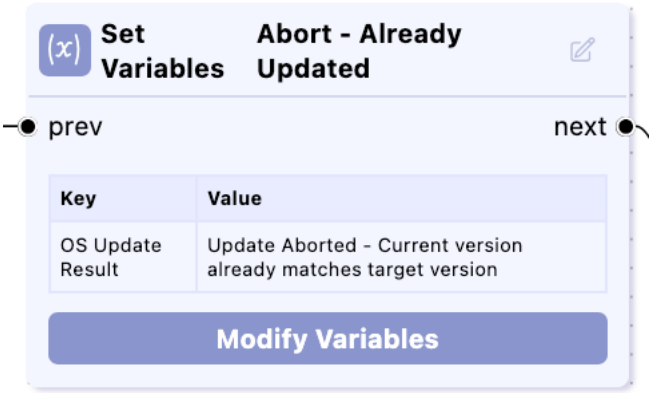


If we have a match, this means we're already running the desired OS version and we should abort. Otherwise, the "nonMatches" branch will be taken, which is linked to our second **Regex Match** node, described below.

A pattern used in this playbook is anytime we reach an "end" state we want to send an email to our network team to let them know whether the update was successful, failed or aborted.

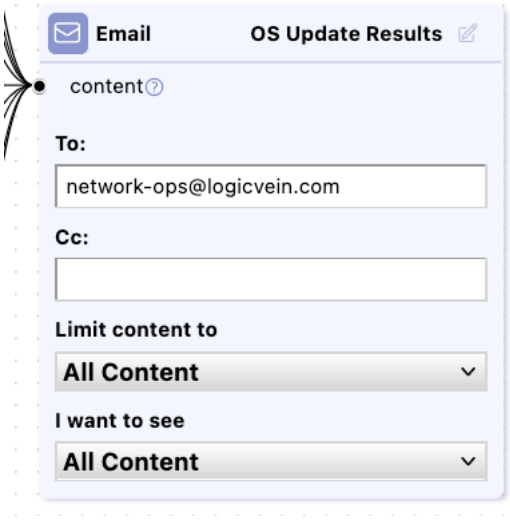
However, before we send the email, we'll place a descriptive message in a new variable called "OS Update Results" and then pass this variable to our email node.

So, when we abort because the version is already up to date, the message variable is constructed like this:

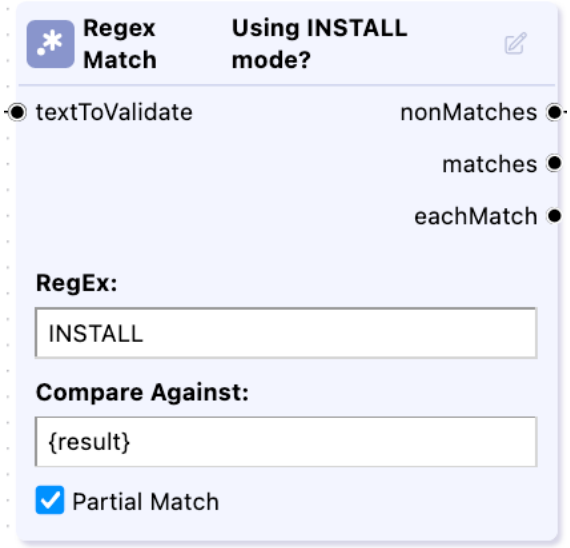


The “next” element of this **Set Variables** node is then joined to our **Email** node, which is configured to send “All Content”.

Note that all other end-state nodes also connect to this **Email** node, meaning we will always receive an email notification of some kind when the playbook finishes executing.

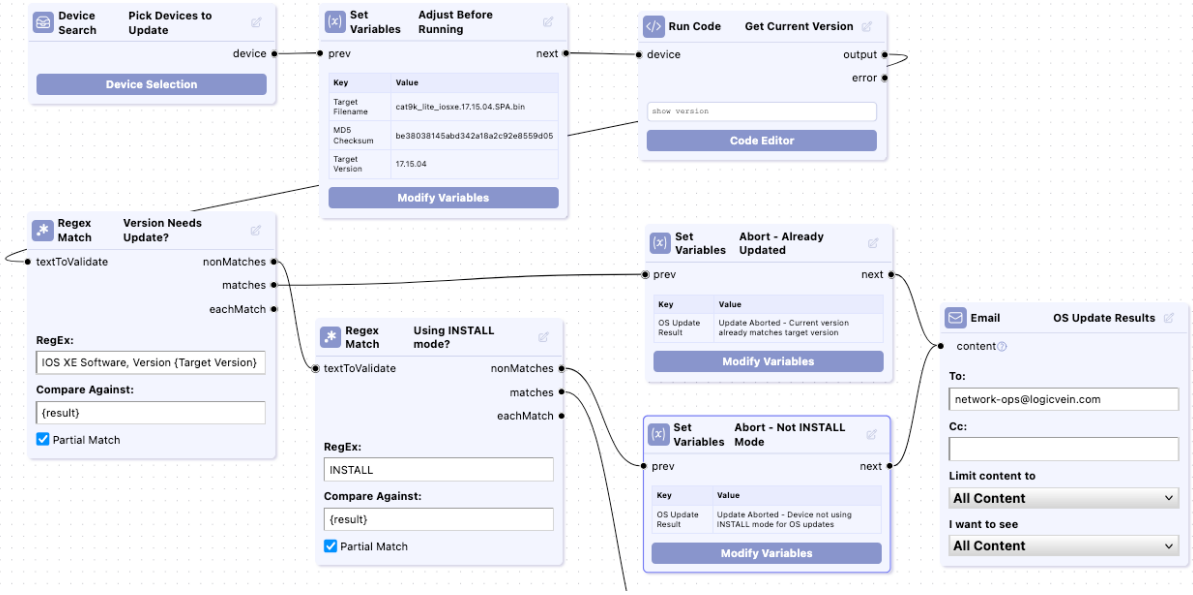


The next **Regex Match** node looks for the word “INSTALL” to be present in the output of the “show version” command.



If we see the word “INSTALL” then the “matches” branch is followed, which is connected to the next step in our playbook. Otherwise, the “nonMatches” branch is followed which leads to another custom message placed in a **Set Variables** node and then sent to the **Email** node.

Here's the complete playbook flow discussed in this section. For clarity, other nodes not discussed in this section have been removed, and the nodes have been laid out differently than the "zoomed out" view shown earlier, but the connections are the same.



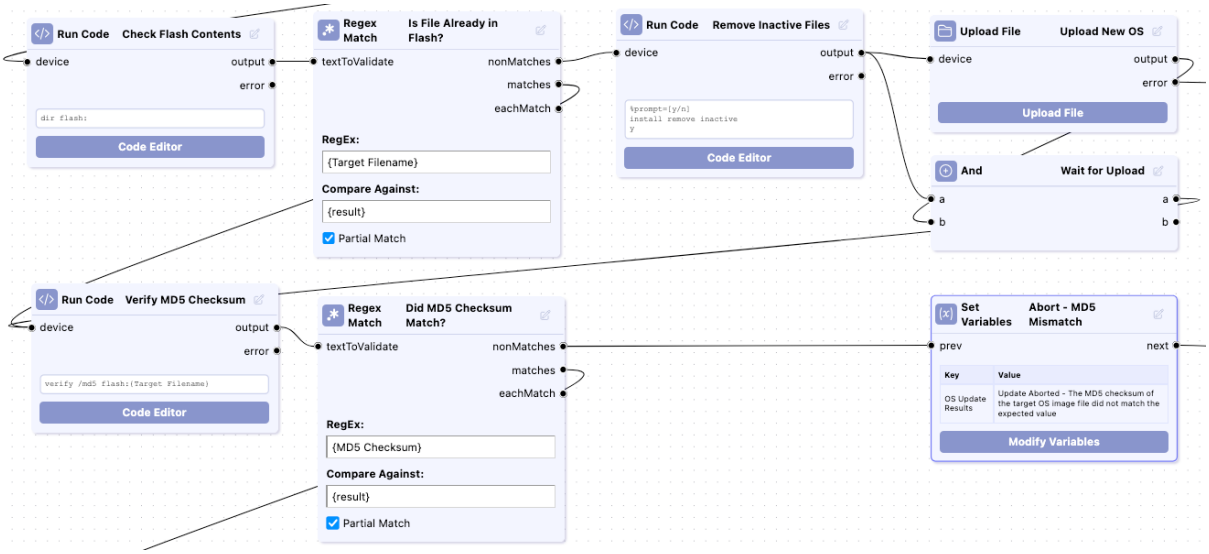
The "matches" branch on the second **Regex Match** node leads to the flow discussed in the next section.

3.1.2.4 Upload and Verify the OS File

Once the existing device version and installation mode are confirmed, we want to upload the OS image to the device and verify its MD5 checksum.

This section corresponds to steps 4 to 7 from our flowchart.

Here's the flow discussed in this section:



We start at **Run Code** node with the description “Check Flash Contents”. (The input to this node is the “matches” branch from the second **Regex Match** node discussed in the previous section.)

This node runs the “dir flash:” command, which will give us a listing of existing files in the device’s flash memory. The output of this node is linked to **Regex Match** node that is looking for our **{Target Filename}** in our command **{result}**.

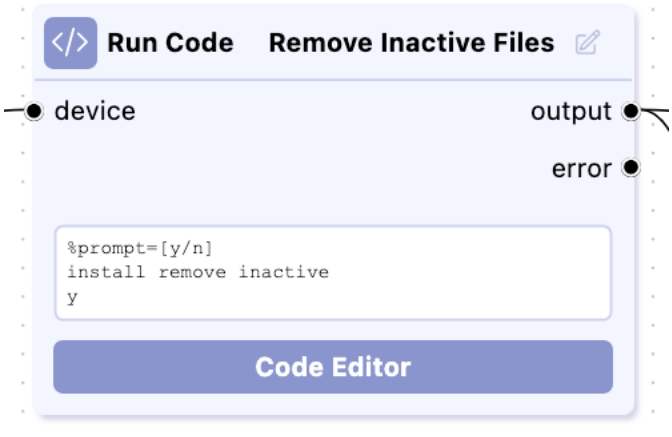
Reminder:

- We defined our **Target Filename** in the initial **Set Variables** node.
- The **{result}** variable contains the contents of the most recently executed **Run Code** node.
- When we want to use any variable in a playbook, it must be surrounded by curly braces.

If this **Regex Match** node finds the **Target Filename** in the output of the “dir flash:” command, then the “matches” branch is followed, otherwise the “nonMatches” branch is followed.

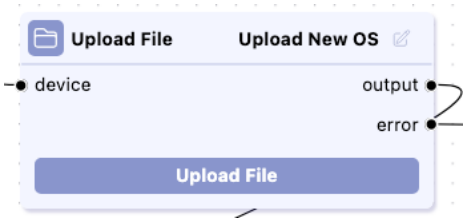
From the “nonMatches” branch we want to upload our **Target Image**, since it was not found in the device’s flash. First however, we want to clean up our device’s flash storage by removing unneeded files. In the case of the Cisco Catalyst 9200 switch, this can be done with the “install remove inactive” command.

We place this command in new **Run Code** node that is connect to the “nonMatches” branch of the **Regex Match** node.



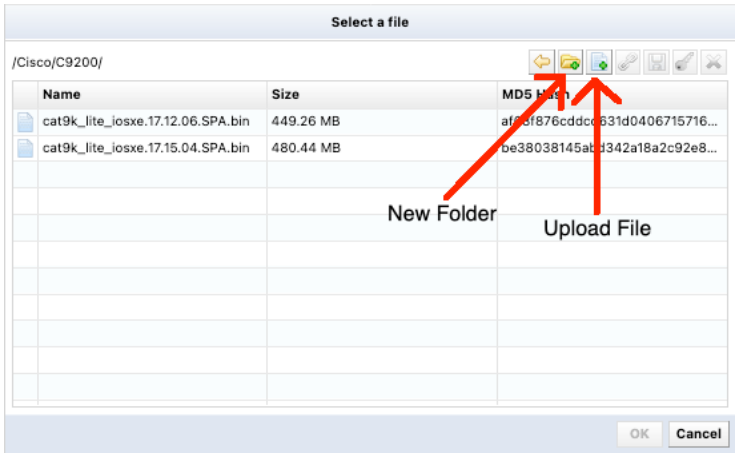
Notice that just above the “install remove inactive” command, we have the text “%prompt=[y/n]”. This tells the **Run Code** node that the “install remove inactive” command will present the user with a “[y/n]” confirmation prompt after that command has been run. We will then answer this prompt with “y” (on the third line of code), meaning “yes”.

Following this output of this node, we have the **Upload File** node.



Clicking on the **Upload File** button lets us select the **Transfer Type** (FTP, TFTP or SCP), the **Source** file and the **Destination**.

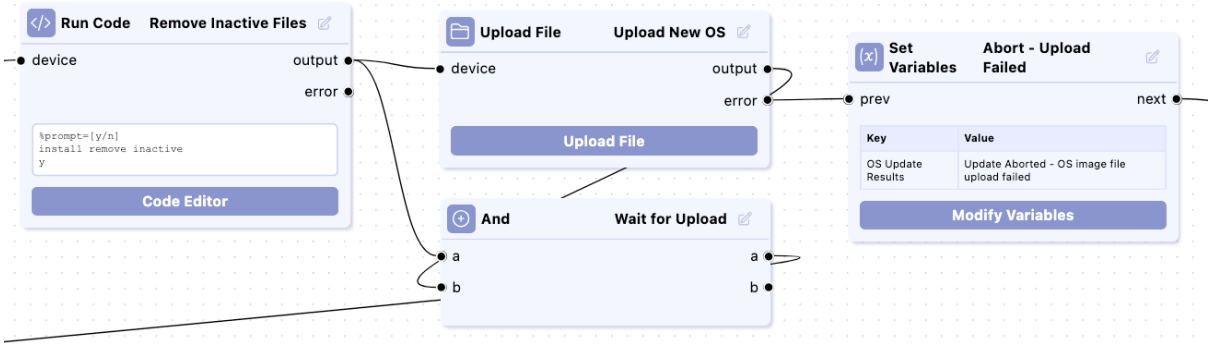
Important concept: The source file is the OS image file that is hosted on the LogicVein server. As a preliminary step, you will want to upload your OS image file(s) to your LogicVein instance. When clicking on the **Source** field in the **Upload File** window, you will be presented with a file selector, where you can browse to the hosted OS image file you want to use. If the file you need has not been uploaded to your LogicVein instance yet, you can add a new file from your local system by clicking on the Upload Files button. You can also organize your files into nested folders using the New Folder button.



Note that uploading the file to your LogicVein instance does not push it to your devices at that time. LogicVein will only push the file to your selected devices once the playbook has been executed and the **Upload File** node runs.

In the **Destination** field, enter the full remote path including the filename.

In this playbook, the output of the **Run Code** node to remove inactive files also goes to an **And** node, into input “a”. The output of the **Upload File** node goes into input “b”, as shown:



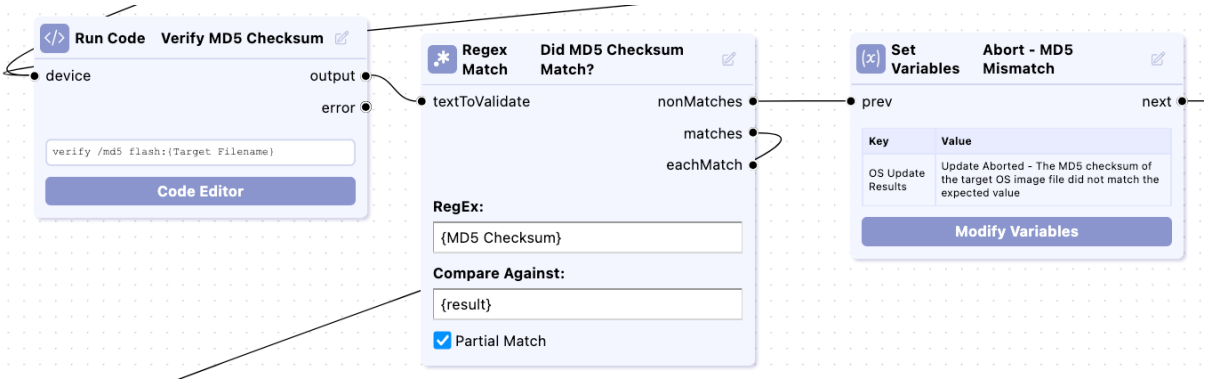
The **And** node acts as a gate that will only proceed once it receives input into both of its “a” and “b” inputs and then proceed to the next node via output “a”.

Why are we doing this? In this case, we’re doing this because once the **Upload File** node runs, in its “output” we lose all our previous configured variables (from our initial **Set Variables** node), but the output of the **Run Code** node preserves these variables.

Since we want to keep using these variables later, we use the **And** node to wait until the file upload is complete and then send the “output” stream from the **Run Code** into input “a” of the **And** node, which then exits from output “a” of that same node and goes to the next node in our playbook, but only once the file upload has completed. (We don’t need to carry forward the “b” input, so output “b” doesn’t connect to anything.)

If the **Upload File** node experienced an error, the flow would go to the **Set Variables** node that sets a message variable to indicate that the upload failed and then would pass this message on to the final **Email** node, ending the playbook.

Next are the nodes to check the MD5 checksum of the uploaded OS file.



First is a **Run Code** node that runs the “verify /md5” command against our **{Target Filename}**. Note that there are two possible input connections to this node, either:

- From the “matches” output of the **Regex Match** node that checks whether the file was already in the device’s flash storage, or,
- From the **And** node that waits for a successful file upload of our target OS file.

In either case, once the verification command is run, it is sent to a **Regex Match** node to compare the output of our verification command against the expected value. The expected checksum was defined in the **{MD5 Checksum}** variable that we initially configured at the top of the playbook. The output of the verification command is present in the **{result}** variable.

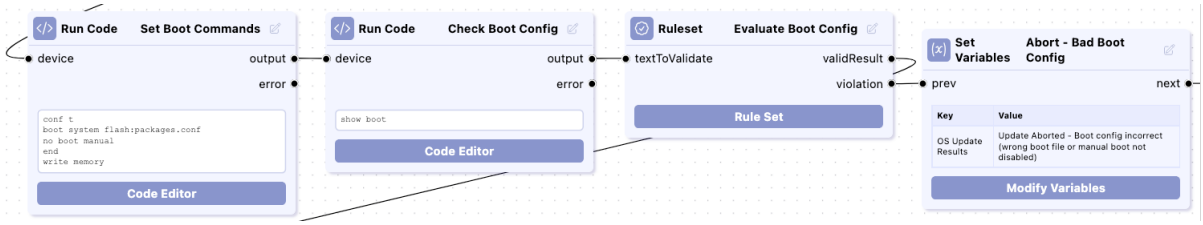
If there not a match, we exit from the “nonMatches” output, which links to a **Set Variables** node that creates a message that indicates that the checksum verification failed, and then passes this on to the **Email** node before ending the playbook.

If there is a successful match between the checksum values, we exit of the “matches” output, onto the **Run Code** node that sets the boot commands, described in the next section.

3.1.2.5 Configure and Verify the Boot Commands

Once we’ve uploaded and verify the integrity of the new OS image file, we want to configure and verify any needed pre-installation boot commands. The specifics of what this entails will vary by device make, model and OS. However, for our example playbook built to update the Cisco Catalyst 9200, we need to ensure that the device is configured to boot from “packages.conf” and we need to ensure that the boot mode is not set to “manual”.

This section corresponds to step 9 from our flowchart.



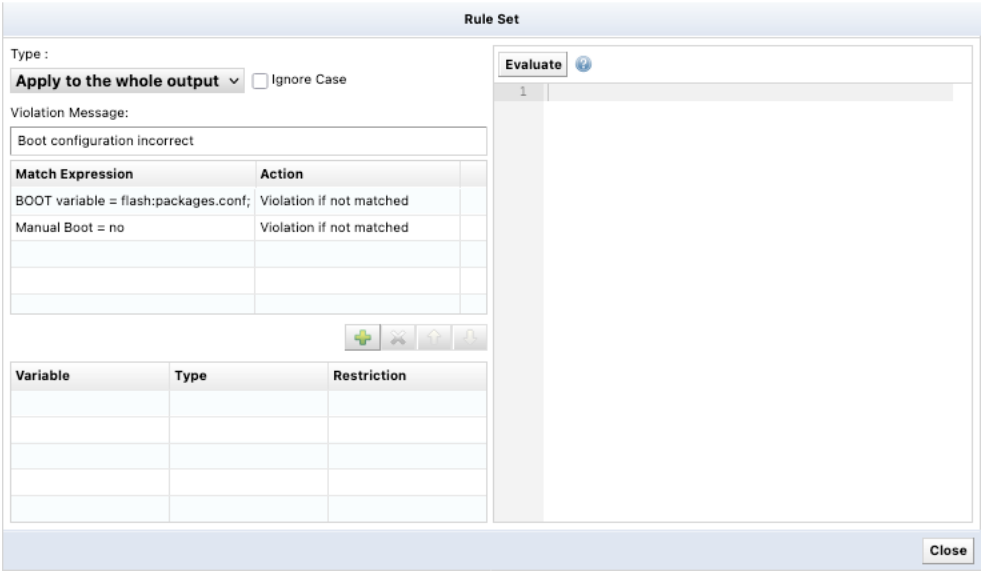
The first node in this section is a **Run Code** node that takes its input from the "matches" output of the prior **Regex Match** node that verified the MD5 checksum of the uploaded file.

This node contains the following commands:

```
conf t
boot system flash:packages.conf
no boot manual
end
write memory
```

The output of this node flows into a second **Run Code** node that runs the "show boot" command, which in turn is sent to a **Ruleset** node.

By clicking of the **Rule Set** button, we can define the contents of the rule set, which in our example looks like this:



Our rule set defines two rules, each set to "Violation if not matched". The match expressions are:

- BOOT variable = flash:packages.conf;
- Manual Boot = no

This means that if either of these two lines are not present in the output of the previous "show boot" command, then the rule set will be in a "violation" state. However, if both lines are present in the command output, then the rule set will be in a "validResult" state.

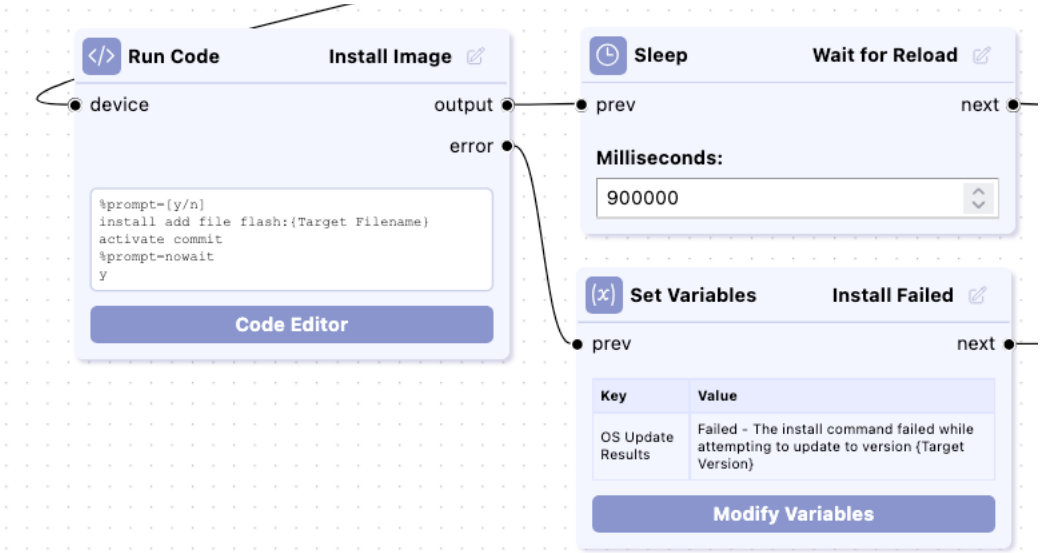
If the rule set is in a “violation” state, then the playbook continues into the **Set Variables** node that sets a results variable indicating that the installation will be aborted, which in turn connects to the final **Email** node, ending the playbook.

Otherwise, if we have a “validResult” state, we continue into the **Run Code** node that executes the OS update commands, described in the section.

NOTE: A valid alternative to using the **Ruleset** node would be to use two chained **RegEx Match** nodes, with each one verifying a one of the required configuration lines. We used that approach in the nodes discussed in section 3.2.1.3.

3.1.2.6 Apply the OS Update

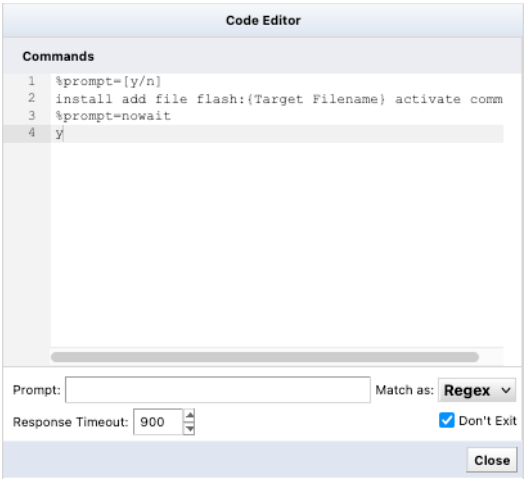
With the OS file uploaded & verified, and pre-installation configurations verified, we can now apply the OS image update.



This section corresponds to steps 10, 11 and 12 from our flowchart.

First is a **Run Code** node that is linked to the **Ruleset** node that checked the boot configuration, as discussed in the previous section.

Clicking on the **Code Editor** button, we can see the following:



This code will run the command:

```
install add file flash:{Target Filename} activate commit
```

Where **{Target Filename}** is a variable that we defined at the top of the playbook.

Note that just above this command we see:

```
%prompt=[y/n]
```

As discussed previously, this is a special directive that tells the **Run Code** node that the next command will have present a prompt that differs from the standard device prompt. In this case, we are expecting the prompt to contain the text “[y/n]”.

We answer this prompt with “y”. However, just above that line we can see another special directive:

```
%prompt=nowait
```

What this directive means is “do not wait for a device prompt after running the next command”. In this case the next command is “y”, as an answer to the previous prompt.

Why are we doing this? We’re doing this because after answering “y” to the prompt that appears after running the “install add file flash:{Target Filename} activate commit” command, the device will reboot itself. Since the device will be rebooted, we should not wait for the command prompt to return after running the confirmation command.

This directive goes together with the selection of the “Don’t exit” checkbox as seen in the **Code Editor**. This means “don’t end the playbook because the device connection was lost”. This is important because we want to run some verification commands after the device reboots.

If for some reason running the installation commands produces an error, we proceed to the **Set Variables** node that sets a failure message in a variable, which is then passed to the final **Email** node, ending the playbook.

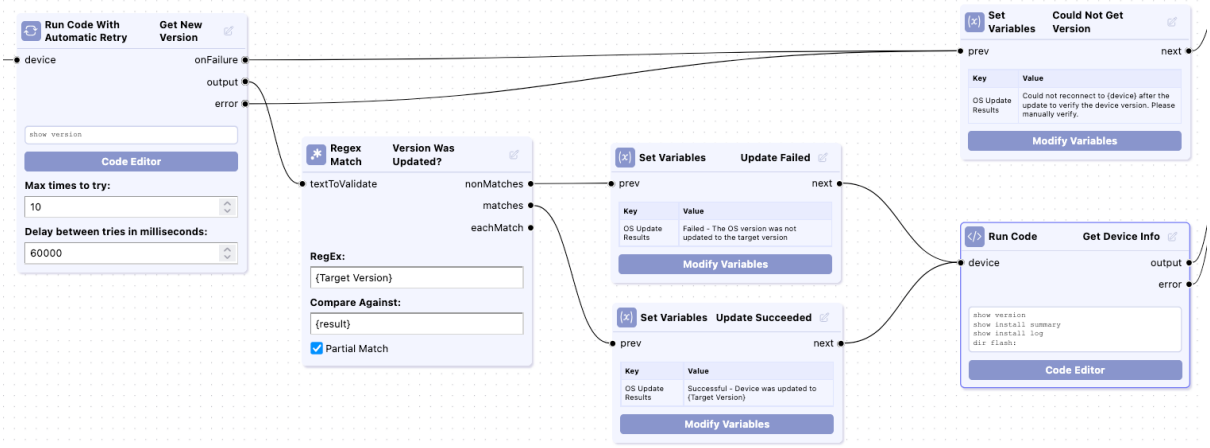
If there was no error, we then continue to a **Sleep** node. The purpose of this node is to allow our device sufficient time to reboot. The correct value to use here may vary based on the type of device. In our example, we wait 900000 milliseconds (15 minutes), giving the device plenty of time to reboot.

After the sleep time, we proceed to the post-installation verification steps, discussed in the next section.

NOTE: In this example playbook, the execution of the installation commands automatically reboots the device. For other types of devices, it may be required to introduce a separate “reload” command.

3.1.2.7 Post-Update Verification

After the update was completed and the device was rebooted, we’ll want to run some verification commands to ensure that the update did in fact complete successfully.



This section corresponds to step 13 from our flowchart.

The first **Run Code with Automatic Retry** node is connected to the **Sleep** node discussed in the previous section. This node runs the “show version” command. If the command fails for any reason (such as not being able to connect to the device because it is still booting), the node will wait for 60000 milliseconds (1 minute) before retrying and will retry a maximum of 10 times. The wait time between attempts and the maximum number of times to try are configurable.

If the node still fails after the 10 tries, a **Set Variables** node will prepare a message indicating that the post-update verification tasks could not be run. This message is then connected to the final **Email** node.

If the node successfully runs the “show version” command, its output is sent to a **Regex Match** node that compares the **{Target Version}** (which we set at the top of the playbook in our initial **Set Variables** node) to the **{result}** variable, which contains the output of the “show version” command.

If a match is not found, a message is prepared in a **Set Variables** node to indicate that the update failed. If a match is found, a different **Set Variables** node is run to prepare a message indicating that the update was a success. In both cases, the next node is a **Run Code** node that will run a set of commands to display useful information that the network operations team can review to either confirm that the update succeeded, or to help them troubleshoot why the update may have failed.

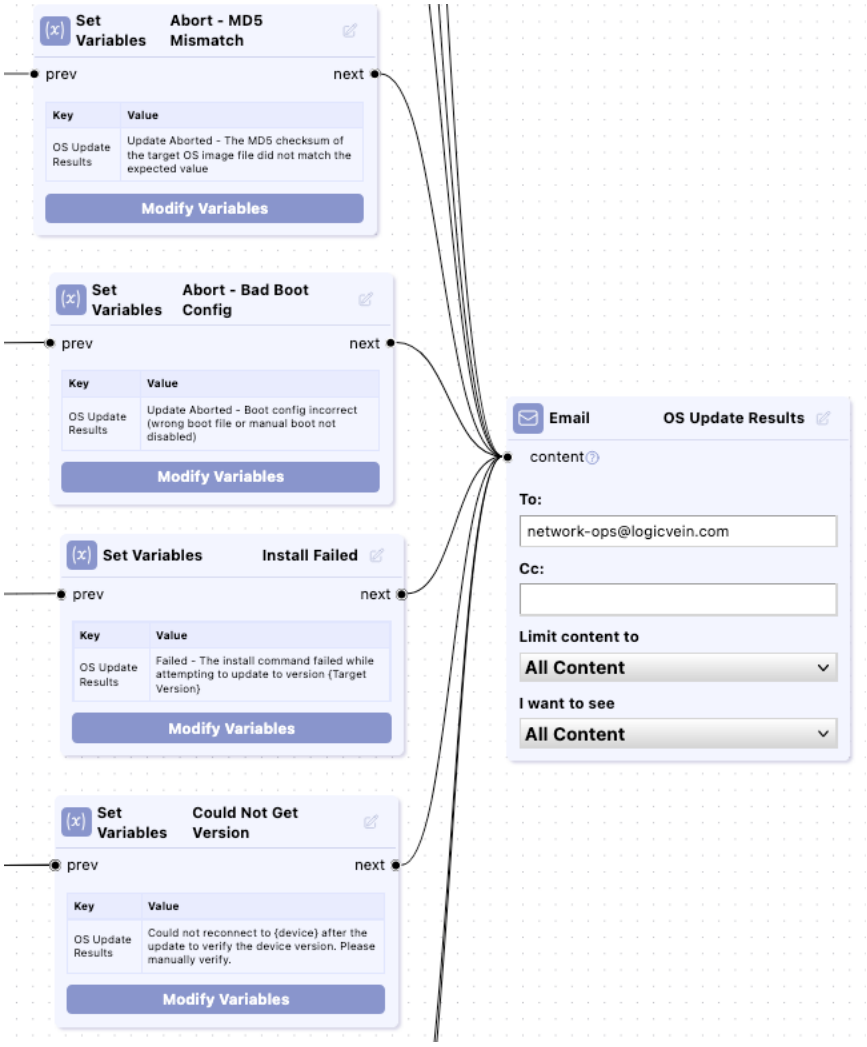
3.1.2.8 Email Notification

The final node is the **Email** node that will send the results of the verification commands along with the output of the post-install verification commands. This node is configured to send **All Content** to the recipients.

This section corresponds to step 14, the final step, of our workflow.

The playbook is designed so that any condition causing the playbook to terminate early (aborting the install), or any error condition, or a successful update, all flow into the **Email** node as the final node of the playbook.

As seen in earlier sections, this playbook also sets a custom variable called “OS Update Results” that is passed to the **Email** node. This variable’s content changes based on the reason why the playbook ended.



Once configured, this node sends an automatic notification whenever the playbook finishes, giving the team remote visibility into the update’s end status.

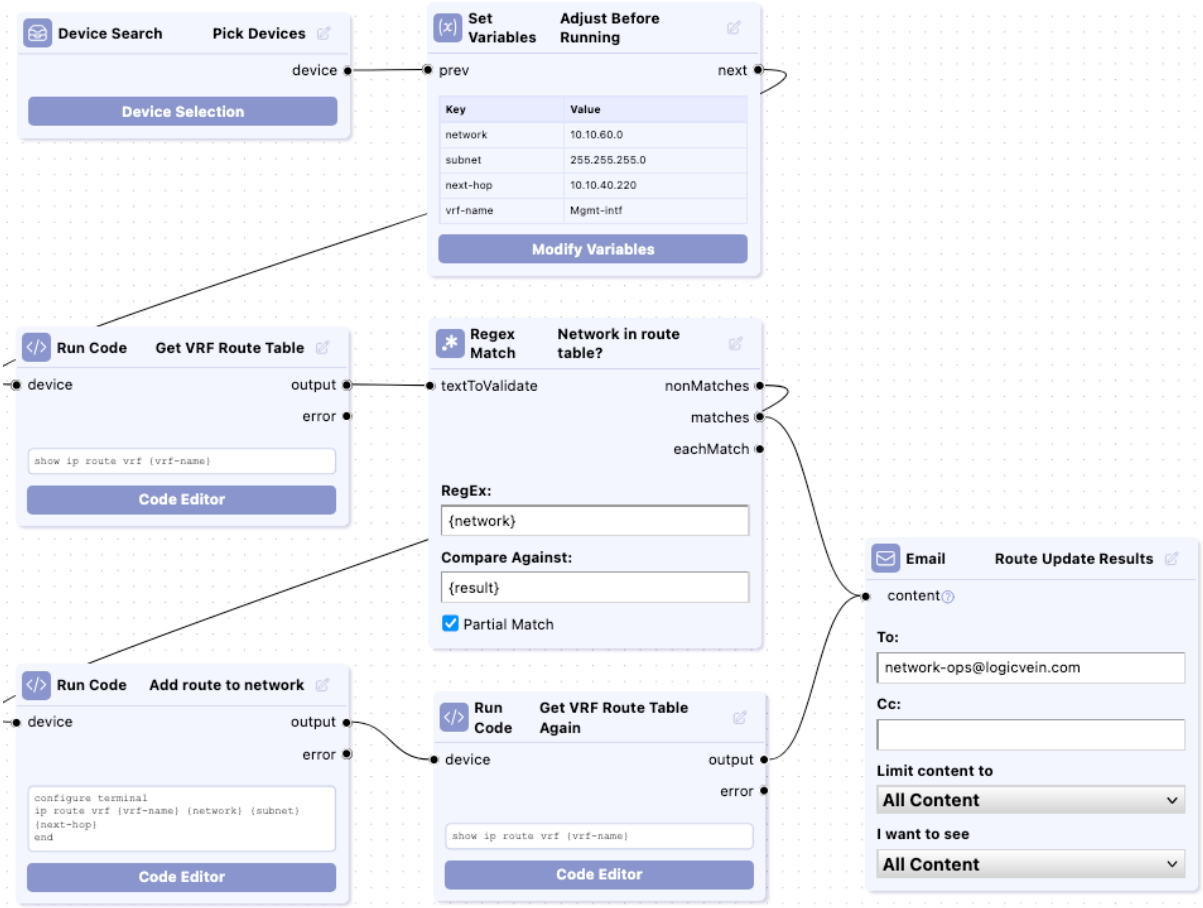
This completes the OS update automation. Running this playbook end-to-end ensures a consistent, error-checked update process across all target devices.

3.2 Adding Static Routes

Adding static routes is one of the most common tasks in day-to-day network operations. Automating it with a playbook reduces configuration errors and makes the process repeatable.

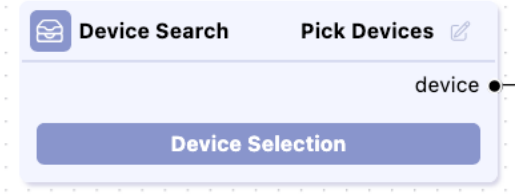
3.2.1 Playbook Walkthrough

The entire playbook is shown below. The next sections will break it down by node.



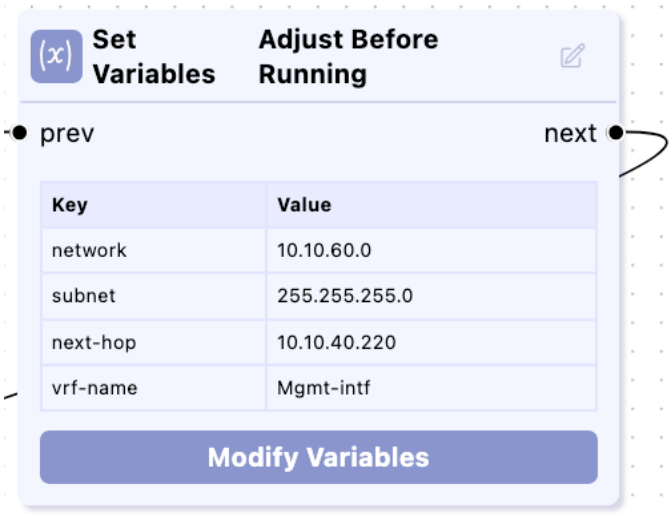
3.2.1.1 Select the Target Device(s)

Configure the target device(s) using the **Device Search** node. For setup details, see Section 3.1.2.1.



3.2.1.2 Define Variables for Reuse

The **Set Variables** node lets you centrally manage the values used throughout the playbook. This makes it easy to reuse the same workflow with different settings – just update the variables.



Field	Description
Key	The variable name used as a reference in subsequent nodes. Reference a variable in other nodes by surrounding the variable name in curly braces, for example **{network}** will contain the value of the “network” variable.
Value	The value assigned to the variable. Changing this value lets you reuse the playbook under different conditions without editing individual nodes.

For this use case, define the following variables:

Key	Value
vrf-name	The VRF to which the route will be added.
next-hop	The next-hop gateway address.
subnet	The subnet mask for the route.
network	The destination network address.

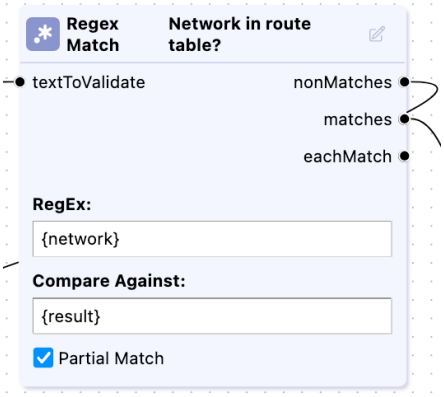
3.2.1.3 Get the VRF Route Table

Use a **Run Code** node to retrieve the routing table for the specified VRF. Reference the variable **{vrf-name}** directly in the command.

3.2.1.4 Check Whether the Route Already Exists

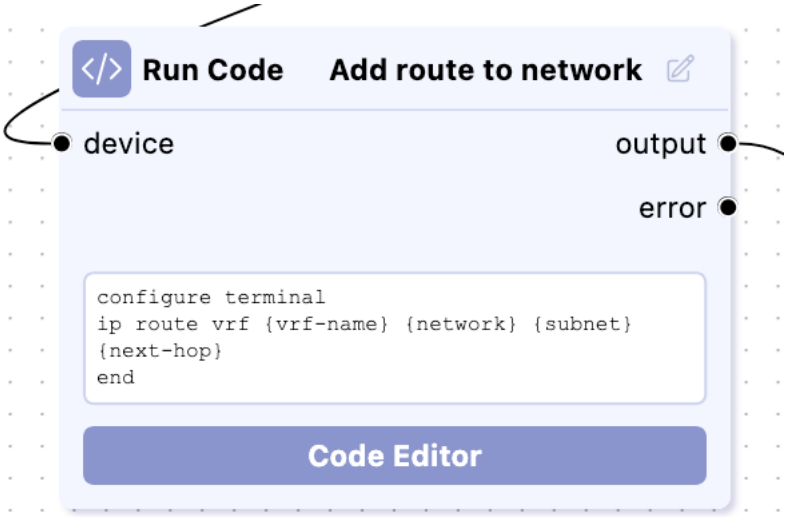
Use a Regex Match node to determine whether the target network already appears in the routing table. Enter **{network}** in the Regular Expression field to reference the variable.

If the network is not already configured, the flow follows the “nonMatches” path to proceed with adding the route.



3.2.1.5 Configure the Static Route

Use another **Run Code** node to configure the static route using the variables defined earlier, **{vrf-name}**, **{next-hop}**, **{subnet}**, and **{network}** as command arguments. This approach lets you configure various static routes simply by updating the variable values.

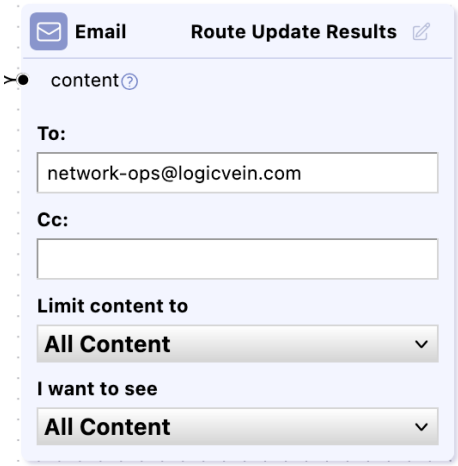


3.2.1.6 Get Post-Configuration Data

Use another **Run Code** node to run the show command again to retrieve the updated routing table and confirm that the static route was correctly configured.

3.2.1.7 Send the Final Results Notification

Notify your team via an **Email** node of the playbook's outcome: whether the route was added successfully, was already present, or encountered an error. An alternative would be to use the **Chat App (Webhook)** node.

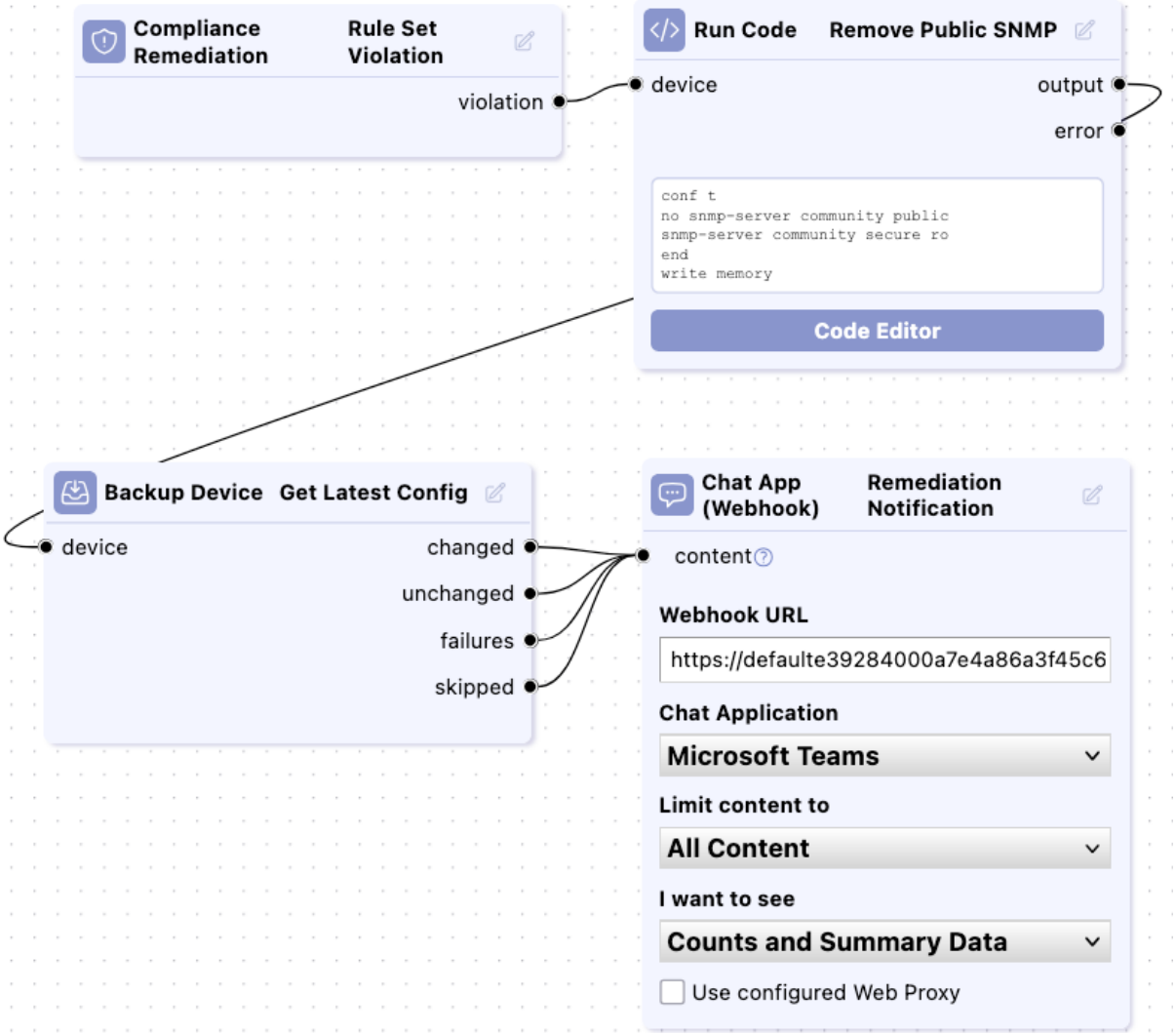


3.3 Compliance Remediation

When a compliance violation is detected, this playbook automatically remediates the device configuration based on predefined policies – maintaining the integrity and security of your network without manual intervention.

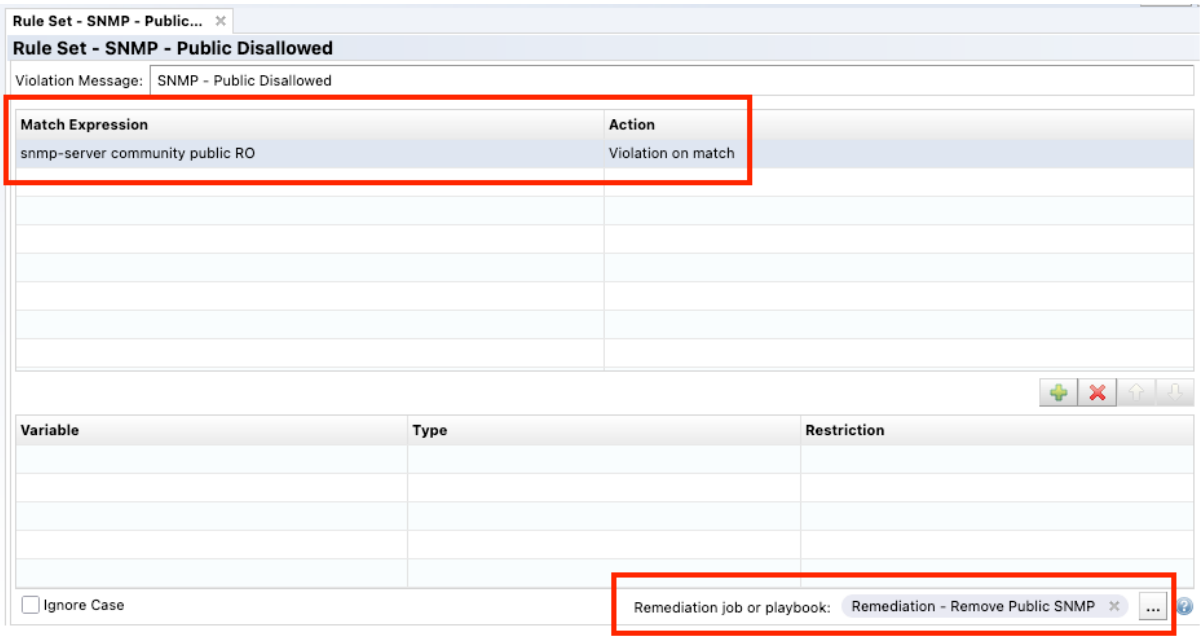
3.3.1 Playbook Description

This scenario detects a compliance violation, runs a remediation command on the offending device, backs up the corrected configuration, and notifies the team via a chat application.



3.3.2 Compliance Policy Prerequisites

Before running this playbook, ensure that your compliance policies and associated rule sets are configured in the system. A rule set must be built to detect the violation condition, which in this case is the presence of a “public” SNMP community string. The rule set’s “Remediation job or playbook” option must be associated with this playbook for auto-remediation to take effect.

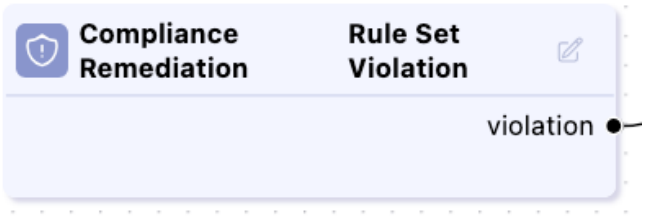


3.3.2.1 Detecting Compliance Violations

Start with a **Compliance Remediation** node. It triggers the playbook when a non-compliant configuration is detected by a rule set that is linked to this playbook, as described above.

Information about the violating device is passed to the “violation” output for use by downstream nodes.

NOTE: A **Device Search** node is not used or required in this playbook, as the affected devices will have been determined by the associated policy rule set.



3.3.2.2 Run the Remediation Command

Place a **Run Code** node to apply the corrective configuration to the offending device. Link the “violation” output of the **Compliance Remediation** node to the “device” input of this **Run Code** node.

NOTE: Review the linked compliance policy rule set before configuring the remediation commands. Applying commands that are not relevant to the issue raised by the rule set can cause unintended configuration changes.

3.3.2.3 Back Up the Corrected Configuration

Place a **Backup Device** node to save the remediated configuration. Link the “output” of the **Run Code** node to the “device” input of this node.

3.3.2.4 Send a Chat Notification

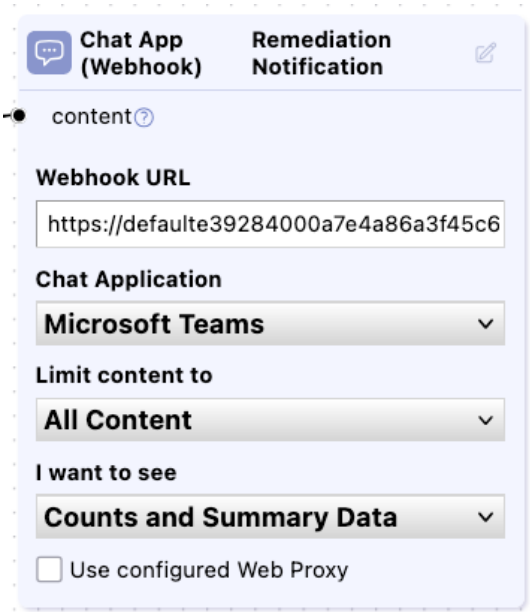
Notify administrators in real time that a violation was detected and remediated.

Place a **Chat App (Webhook)** node and configure it as follows:

Setting	Description
Webhook URL	The incoming webhook URL for your chat application. Create this in your chat app’s settings.
Chat Application	Select your platform: Microsoft Teams, Slack, Mattermost, Webex, or LINE WORKS.
Restrict Content	Select “Status Only”, “Error Only”, or “All Content”.
Content	Select “Count by Node Output” or “Count and Summary Data”.

Setting	Description
Use Configured Web Proxy	Enable this if your environment routes external traffic through a proxy server (configured in Settings > Web Proxy).

Connect all of the outputs from the **Backup Device** node to the “content” input of this node, since we want to always want to receive a chat notification at the end of this playbook, whether the backup detected a config change or not, was skipped, or had an error.



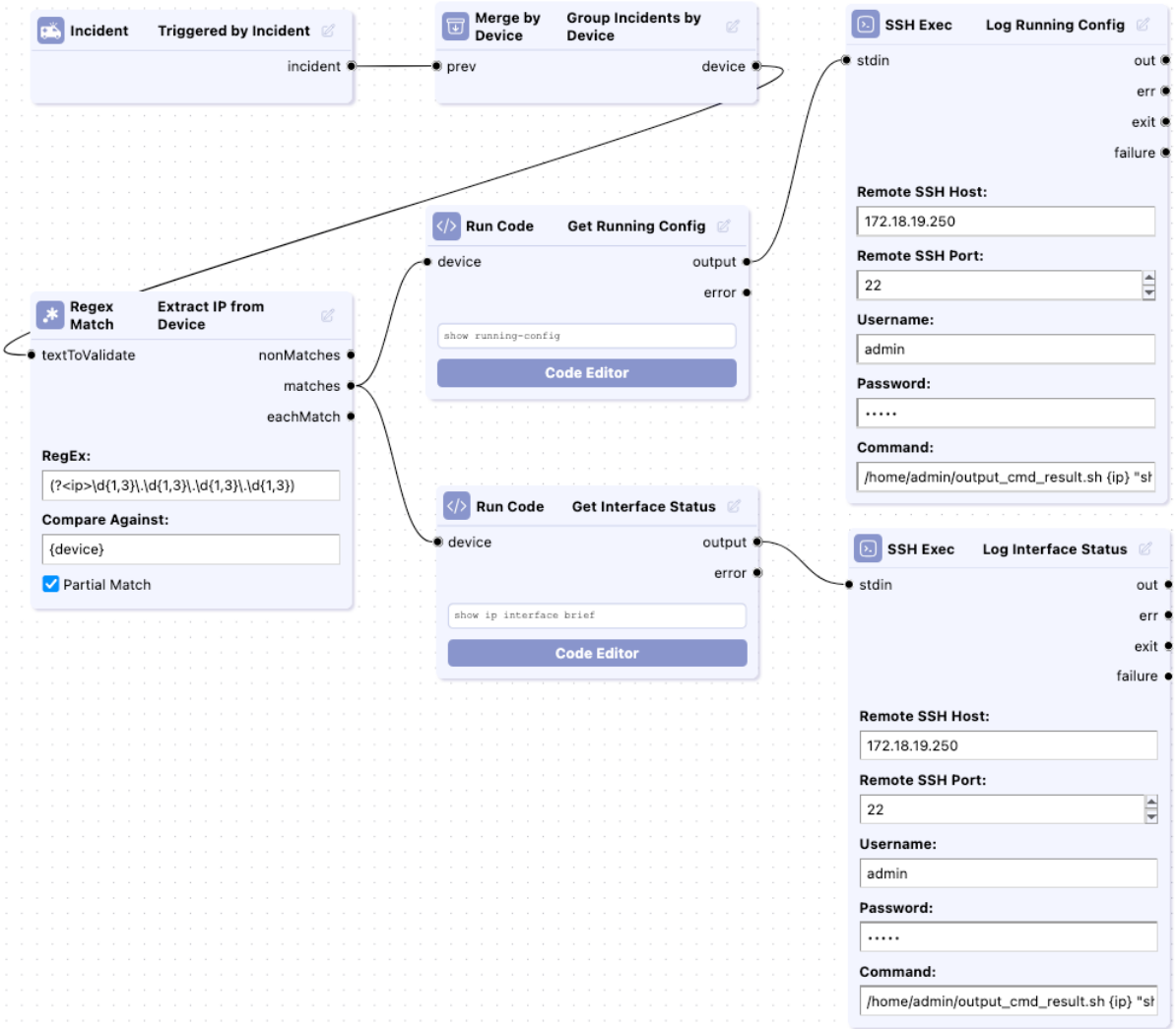
3.4 Automated Incident Response

Section 3.4 covers a scenario that uses the **Incident** node, which is exclusive to ThirdEye Suite.

When a device fault is detected, this playbook automatically collects diagnostic information and stores it on a remote server. The stored data can then be used for root-cause analysis and recovery planning.

3.4.1 Playbook Description

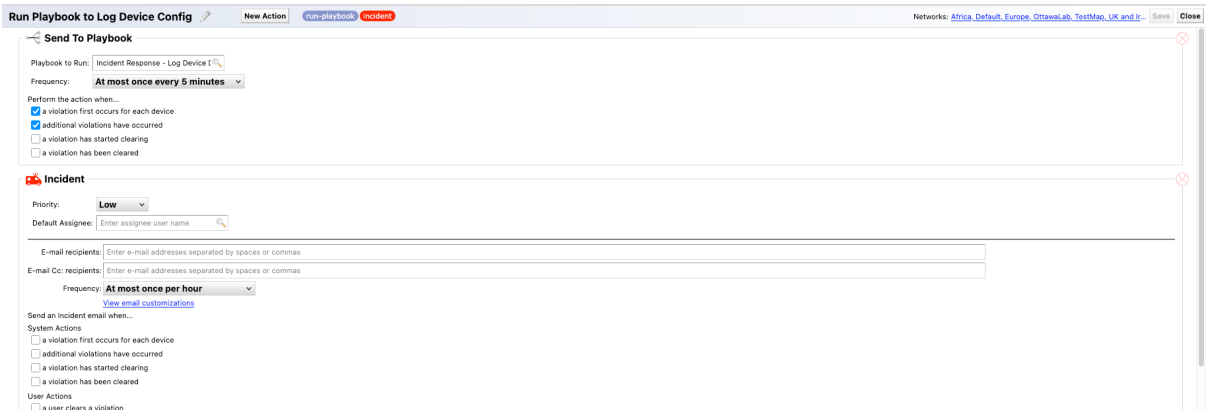
This playbook triggers on a device incident, extracts the device IP address, runs diagnostic commands, and executes a shell script on a Linux server to store the results as files.



3.4.1.1 Prerequisites

Ensure that alert policies are configured so incidents are generated when device anomalies occur. The alert policy should be configured with the **Incident** action and

the **Send to Playbook** action. Select the name of this playbook as the target for the “Playbook to Run” field.



The remote Linux server must accessible via SSH and have the storage script pre-deployed and made executable. The script used in this example is named “output_cmd_result.sh” and the contents of the script are as follows:

```
#!/bin/bash

# Check if exactly 3 arguments are provided
if [ "$#" -ne 3 ]; then
    echo "Usage: $0 <IP_ADDRESS> <COMMAND_LABEL> <COMMAND_RESULT>"
    exit 1
fi

# Assign arguments to descriptive variables
IP_ADDR=$1
CMD_LABEL=$2
CMD_RESULT=$3

# Replace spaces with underscores in the command label for the filename
CLEAN_CMD=$(echo "$CMD_LABEL" | tr ' ' '_')

# Construct the filename
FILENAME="${IP_ADDR}_${CLEAN_CMD}.csv"

# Write the result to the file
# Using printf or echo with quotes ensures multiline text is preserved
echo "$CMD_RESULT" > "$FILENAME"

echo "File created successfully: $FILENAME"
```

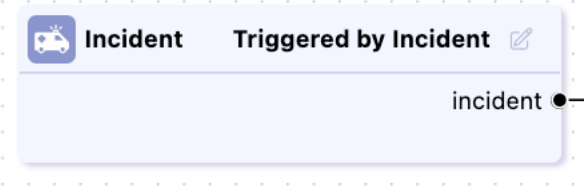
NOTE: This example uses a Linux SSH server, but remote execution on a Windows server is also possible by running a PowerShell script. A SSH Exec node command to run a Windows PowerShell script would look something like this:

`powershell -File 'C:\Users\admin\scripts\output_cmd_result.ps1'`

3.4.1.2 Detect the Incident

Place an **Incident** node as the starting node. It triggers when a monitoring alert policy generates an incident and the alert policy contains an action to run this playbook.

The node captures incidents that occur during the configured monitoring period and passes the device information to downstream nodes.



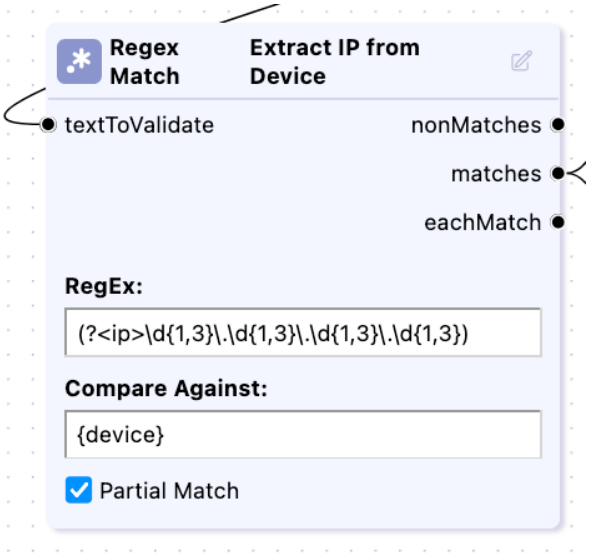
3.4.1.3 Merge Incidents per Device

Use a **Merge by Device** node to consolidate multiple simultaneous incidents for the same device. Depending on your monitoring rules, a single device may generate multiple violations. This node merges them so that downstream processing runs once per device.

3.4.1.4 Extract the Device IP Address

The device IP address is used to name the output files. It can be extract from the built-in variable **{device}**, which is a string in the form “<ip_address>@<network>”. Use a Regex Match node to extract the IP address from the device information passed by the Incidents node.

Place a Regex Match node and enter a regular expression that matches an IP address pattern.



In this case, our regex is using a named capture group called <ip>.

The entire regex used here is:

```
(?<ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})
```

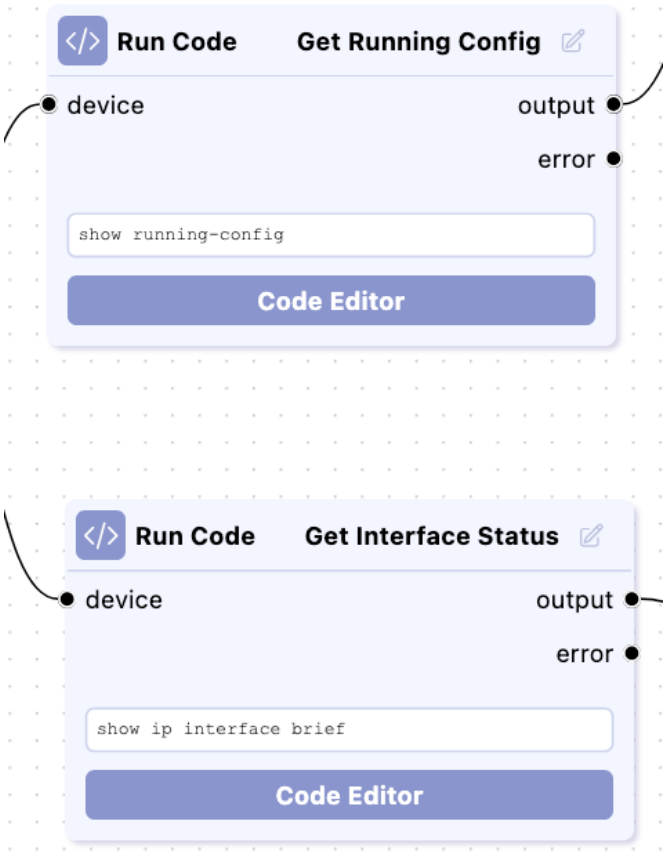
The matched result can then be used in the form of **{ip}** in downstream nodes.

NOTE: To capture variables in regex, follow this pattern:

- Start by creating an empty **non-capture group** like this:
- (?)
- Place your **match expression** inside the non-capture group after the “?”. To match an IP address, a possible pattern we can use is:
- (?\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})
- Insert the name of the **variable** you want to store the match in immediately after the “?”, surrounded by **angle brackets**:
- (?<ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})
- The variable name must start with a letter and contain only letters and numbers.

3.4.1.5 Collect Device Information

Place one or more **Run Code** nodes after the “matches” output of the **RegEx Match** node to run “show” commands or other diagnostic commands on the faulty device. Using separate nodes per command stores each result independently, making post-incident analysis more organized.



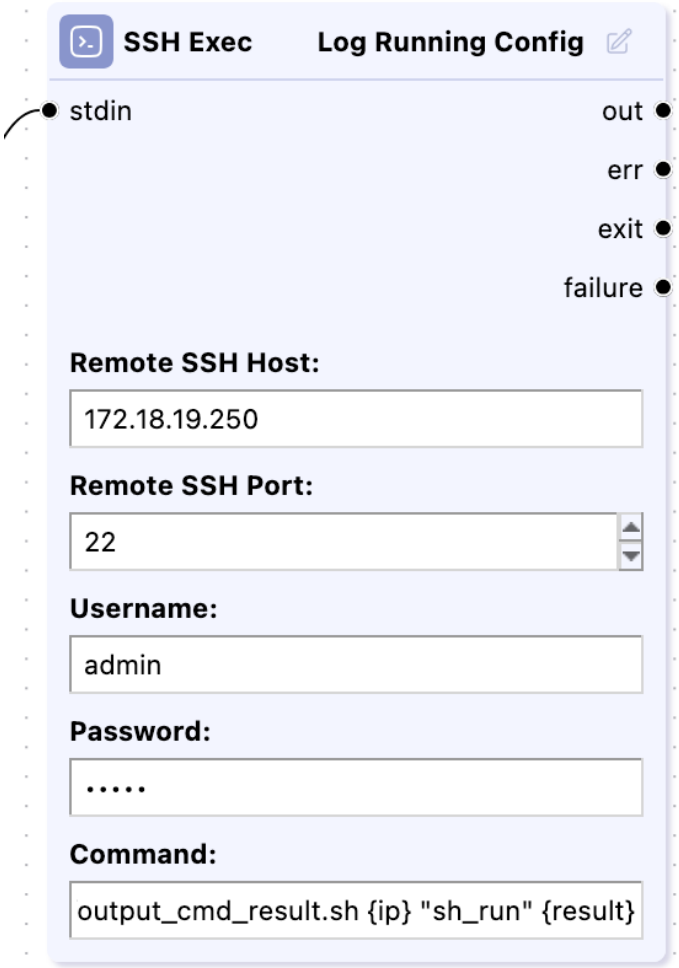
3.4.1.6 Execute the Storage Script via SSH

Use an **SSH Exec** node to connect to the remote Linux server and execute a shell script that stores the collected data.

Place an **SSH Exec** node after each **Run Code** node and configure the following settings:

Setting	Description
Remote SSH Host	The hostname or IP address of the server where the script is stored.
Remote SSH Port	The SSH port number (default: 22).
Username	The login username for the SSH server.
Password	The login password for the SSH server.
Commands	The script command to execute. You can pass variables from previous nodes as arguments:

Setting	Description
	• {ip} (from the named capture group that capture the IP address from the {device} variable) • a literal string such as “sh run” describing the command that was run or the type of data collected • {result} (the Run Code node output data) Example of a full command: `/home/admin/output_cmd_result.sh {ip} "sh_run" {result}`



When the script executes, it creates a file for each device and command combination on the Linux server.

```
admin@dc1-host1:~$ ls -l
total 64
-rw-r--r--. 1 admin admin  9629 Mar 30 16:47 10.95.1.248_sh_int.csv
-rw-r--r--. 1 admin admin 13067 Mar 30 16:47 10.95.1.248_sh_run.csv
-rw-r--r--. 1 admin admin   693 Mar 30 18:29 10.95.1.46_sh_int.csv
-rw-r--r--. 1 admin admin 25936 Mar 30 18:29 10.95.1.46_sh_run.csv
-rwxr-xr-x. 1 admin admin   609 Mar 30 16:31 output_cmd_result.sh
```

```
admin@dc1-host1:~$ cat 10.95.1.248_sh_run.csv
show running-config
Building configuration...

Current configuration : 12499 bytes
!
! Last configuration change at 13:11:32 UTC Mon Mar 30 2026 by admin
!
version 16.12
no service pad
service timestamps debug datetime msec
service timestamps log datetime msec
service call-home
service unsupported-transceiver
no platform punt-keepalive disable-kernel-core
platform management port rate-limt-enabled
!
hostname c3850-s2
!
!
vrf definition Mgmt-vrf
 !
 address-family ipv4
 exit-address-family
 !
 address-family ipv6
 exit-address-family
!
!
aaa new-model
!
!
aaa authentication login default local
aaa authentication enable default enable
aaa authorization exec default local
aaa authorization commands 15 default local
!
!
!
!
!
!
aaa session-id common
switch 1 provision ws-c3850-48p
switch 2 provision ws-c3850-48p
!
!
!
!
```

4 Appendix

4.1 matches vs eachMatch

The **Regex Match** node offers two output paths when multiple results are found:

Output	Behavior
matches	All matching results are combined into a single match value (one match per device). Use this when you want to process all results together.
eachMatch	Each individual match is stored as a separate value, and the downstream node runs once per match. Use this when you want to process results one at a time.

Example: If **{result}** contains multiple GigabitEthernet interfaces:

matches – stores all matched interfaces in a single value and runs the next node once.

eachMatch – stores each interface separately and runs the next node once per interface.

4.2 Node Element Reference

The table below lists the output elements available from each node type. Use these element names when referencing data in downstream nodes.

NOTES:

- When using a variable generated by a node in downstream nodes, the downstream nodes must surround the referenced variable name with curly braces, for example, **{result}**.
- The **Set Variables** node allows for the generation of arbitrary variables names.
- Named capture groups in **Regex Match** nodes also allow for the creation of arbitrary variable names.
- The **Ruleset** node may also introduce arbitrary variable names if that node defines any variables.
- A playbook that was executed from a **Compliance Remediation** node will carry forward any variables that were defined in the **Rule Set** that was defined (from the Compliance tab) and linked to the playbook.
- The **Incident** node is available in ThirdEye Suite only.
- Nodes that only have an entry point (and no exit point) are not listed in this table.

Node	Exit Point	Variable Name	Description
(All nodes)		Execution	Execution timestamp
(All nodes)		runBy	User who initiated the run
(All nodes)		node	Node name
(All nodes)		PlaybookStartedAt	Playbook start time
(All nodes)		PlaybookName	Playbook name
(All nodes)		PlaybookAuthor	Playbook author
Compliance Remediation	violation	device	Non-compliant device
Compliance Remediation	violation	result	Rule set message

Node	Exit Point	Variable Name	Description
Compliance Remediation	violation	occurrence	Violation timestamp
Compliance Remediation	violation	severity	Severity level
Device Search	device	device	Selected device information
Incident	incident	cleared	CLEARED or NOT_CLEARED
Incident	incident	creation_timestamp	Incident creation time
Incident	incident	device	Originating device
Incident	incident	event_type	VIOLATION
Incident	incident	is_violation	true
Incident	incident	result	Incident message
Incident	incident	occurrences	Number of occurrences
Incident	incident	severity	Severity level
Run Code	output	status	OK
Run Code	output	device	Device information
Run Code	output	result	Command output
Run Code	error	status	ERROR
Run Code	error	result	Error details
Run Code with Automatic Retry	output	status	OK
Run Code with Automatic Retry	output	device	Device information

Node	Exit Point	Variable Name	Description
Run Code with Automatic Retry	output	result	Command output
Run Code with Automatic Retry	error	status	ERROR
Run Code with Automatic Retry	error	result	Error details
Run Code with Automatic Retry	onFailure	device	Device info
Run Code with Automatic Retry	onFailure	< other variables >	Variables defined in rulesets, regular expressions, and variable settings
Regex Match	matches	match	All matches combined (one per device)
Regex Match	eachMatch	match	Individual match values
Regex Match	nonMatch	device	Device info
Regex Match	nonMatch	result	Prior result
SSH Exec	out	result	The stdout stream of the command or script executed on the remote host
SSH Exec	err	result	The stderr stream of the command or script executed on the remote host (such as a command error)
SSH Exec	exit	result	Shell exit code, typically 0 if no error, or 1 if an error occurred
SSH Exec	failure	result	SSH connection error message

Node	Exit Point	Variable Name	Description
Set Variable	next	device	Device info
Set Variable	next	status	OK
Set Variable	next	result	Prior result
Set Variable	next	User defined variables	Variables created by the user while configuring the Set Variable node
Upload File	output	device	Device info
Upload File	output	status	OK
Upload File	output	result	Upload result, will contain name of uploaded file on success
Upload File	error	device	Device info
Upload File	error	status	ERROR
Upload File	error	result	Error details
And	a	All variables from the “a” input	The “a” exit point, if followed, will carry forward all variables that where defined when entering the “a” input point.
And	b	All variables from the “b” input	The “b” exit point, if followed, will carry forward all variables that where defined when entering the “b” input point.
Merge by Device	device	device	Consolidated device info
Merge by Device	device	result	All results
Merge by Device	device	match	Regex matches (if applicable)

Node	Exit Point	Variable Name	Description
Backup	changed	device	Device information
Backup	unchanged	device	Device information
Backup	failures	device	Device information
Backup	failures	error	Error details
Backup	skipped	device	Device information
Convert to CSV	content	content	CSV output
Convert to JSON	content	content	JSON output
Sleep	next	All variables	All variables present when entering the “prev” input will be passed through to the “next” output.