Here is the implementation of the strategy discussed. This solution includes the specific "Chef Guard" logic you requested to solve the frequency issue, followed by a complete cookbook structure to handle the performance conflicts between fapolicyd, MQ, and Splunk.

1. The "Once-a-Day" Chef Guard

You asked how to run chef-client every 30 minutes but prevent heavy operations (like recursive permission checks) from killing performance.

The best way to do this is using a Sentinel File. We check the timestamp of a specific file; if it was touched less than 24 hours ago, we skip the heavy resource.

The Code Pattern:

```
# Define a path for a "timestamp" file
sentinel_file = '/var/cache/chef/mq_heavy_maintenance.last_run'

# 1. The Heavy Task (e.g., recursive SELinux restore)
execute 'heavy_recursive_restorecon_mq' do
  command 'restorecon -R /var/mqm'
  action :run
  # THE GUARD: Check if the sentinel file is less than 24 hours (86400 seconds) old
  not_if do
    File.exist?(sentinel_file) &&
    (Time.now - File.mtime(sentinel_file)) < 86400
  end
  # Only update the timestamp if this command actually runs
  notifies :touch, "file[#{sentinel_file}]", :immediately
end

# 2. The Sentinel File Resource (updates the timestamp)
file sentinel_file do
  action :nothing # strictly driven by the execute resource above
end
```

2. The Complete Cookbook Implementation

Cookbook Name: mq_performance_tuning

A. Cookbook Structure

```
mq_performance_tuning/
├── recipes/
│   └── default.rb          # Main logic
├── templates/
│   ├── fapolicyd.conf.erb  # Main fapolicyd config (Queue/Cache tuning)
│   ├── 80-mq-trust.rules.erb # fapolicyd Trust Rules
│   └── limits.conf.erb     # Splunk throttling
└── attributes/
    └── default.rb          # Default values for tuning
```

B. The Recipe (recipes/default.rb)

This recipe ties together all the solutions: Kernel tuning, fapolicyd queues, Splunk throttling, and the guarded MQ maintenance.

```
# recipes/default.rb

# ------------------------------------------------------
# 1. KERNEL & SYSCTL TUNING (The Foundation)
# ------------------------------------------------------
# Increase open file limits for MQ to prevent "Too many open files"
file '/etc/security/limits.d/99-mqm.conf' do
  content "mqm soft nofile 10240\nmqm hard nofile 65535"
  mode '0644'
  action :create
end

# Tune Kernel Semaphores and File Max for high throughput
sysctl 'fs.file-max' do
  value 524288
  action :apply
end

sysctl 'kernel.sem' do
  value '500 256000 250 1024' # Standard IBM MQ recommended values
  action :apply
end

# ------------------------------------------------------
# 2. FAPOLICYD TUNING (The "I/O Tax" Fix)
# ------------------------------------------------------
# Deploy main config with increased Queue Size to prevent bottlenecks
template '/etc/fapolicyd/fapolicyd.conf' do
  source 'fapolicyd.conf.erb'
  owner 'root'
  group 'fapolicyd'
  mode '0644'
  variables(
    q_size: 32768,      # Critical for MQ bursts
    obj_cache: 4096     # Increased cache
  )
  notifies :restart, 'service[fapolicyd]', :delayed
end

# Add Trust Rules for MQ (Treat the whole directory as safe)
# RHEL 8 supports rules.d - we place a specific file there
template '/etc/fapolicyd/rules.d/80-ibm-mq.rules' do
```

```
  source '80-mq-trust.rules.erb'
  owner 'root'
  group 'fapolicyd'
  mode '0644'
  notifies :restart, 'service[fapolicyd]', :delayed
end

service 'fapolicyd' do
  action [:enable, :start]
end

# ------------------------------------------------------
# 3. SPLUNK THROTTLING (The "I/O Storm" Fix)
# ------------------------------------------------------
# Limit Splunk's disk bandwidth so it doesn't starve MQ
template '/opt/splunkforwarder/etc/system/local/limits.conf' do
  source 'limits.conf.erb'
  owner 'splunk'
  group 'splunk'
  mode '0644'
  variables(
    maxKBps: 10240  # Cap Splunk at 10MB/s
  )
  notifies :restart, 'service[splunk]', :delayed
  only_if { ::File.directory?('/opt/splunkforwarder') }
end

# ------------------------------------------------------
# 4. MQ DIRECTORY HANDLING (The "Thundering Herd" Fix)
# ------------------------------------------------------
# CRITICAL: recursive false prevents Chef from scanning 1 million files
# every 30 minutes, which would freeze the Queue Manager.
directory '/var/mqm' do
  owner 'mqm'
  group 'mqm'
  mode '0755'
  recursive false
  action :create
end

# ------------------------------------------------------
# 5. HEAVY MAINTENANCE GUARD (The "Once-a-Day" Fix)
# ------------------------------------------------------
sentinel_file = '/var/chef/cache/mq_heavy_maintenance.last_run'
```

```
# Ensure the cache directory exists
directory '/var/chef/cache' do
  recursive true
end

# Execute heavy SELinux restore only if 24h have passed
execute 'heavy_mq_restorecon' do
  command 'restorecon -R /var/mqm /opt/mqm'
  action :run
  # The Guard: Check if sentinel exists AND is < 24h old
  not_if do
    File.exist?(sentinel_file) &&
    (Time.now - File.mtime(sentinel_file)) < 86400
  end
  # Only notify the file update if the command actually runs
  notifies :touch, "file[#{sentinel_file}]", :immediately
  ignore_failure true # Don't fail the chef run if this heavy task hiccups
end

file sentinel_file do
  action :nothing
end
```

C. Templates

1. templates/fapolicyd.conf.erb

Note: This template focuses on the queue tuning variables.

```
# /etc/fapolicyd/fapolicyd.conf
# Managed by Chef - Do Not Edit

permissive = 0
# CRITICAL PERFORMANCE TUNING
# Default is often 800-1000. We bump to <%= @q_size %> for MQ.
q_size = <%= @q_size %>
uid = fapolicyd
gid = fapolicyd
do_stat_report = 1
detailed_report = 1
db_max_size = 0
subj_cache_size = 1549
# Increased cache for performance
obj_cache_size = <%= @obj_cache %>
watch_fs = ext4,xfs,tmpfs
trust = rpmdb,file
```

integrity = none
syslog_format = rule,dec,perm,auid,pid,exe,:,path,ftype,trust

2. templates/80-mq-trust.rules.erb
Note: This rule whitelists the entire MQ directory tree. This stops fapolicyd from checking every single file open inside these directories.
IBM MQ Trust Rules
Trust everything inside /opt/mqm and /var/mqm regardless of RPM status

allow perm=any all : dir=/opt/mqm/ trust=0
allow perm=any all : dir=/var/mqm/ trust=0

3. templates/limits.conf.erb
Note: This applies to the Splunk Universal Forwarder.
/opt/splunkforwarder/etc/system/local/limits.conf

[throughput]
# Throttle Splunk to <%= @maxKBps %> KBps to protect MQ I/O
maxKBps = <%= @maxKBps %>

[inputproc]
# Prevent Splunk from opening too many files at once
max_fd = 100

Next Steps for You
 * Generate the Cookbook: Run chef generate cookbook mq_performance_tuning.
 * Copy the Code: Paste the snippets above into the respective files.
 * Test in Permissive Mode: Before enforcing this on production, ensure fapolicyd is in permissive = 1 (in the config template) to verify that your queue size is sufficient. You can check the fapolicyd stats to see if the queue is overflowing.

—----------_—---------:----------
This is a classic "high-friction" stack. You are combining a high-throughput, low-latency application (IBM MQ) with a blocking security layer (fapolicyd & SELinux) and a heavy read/write monitoring agent (Splunk), all automated by Chef.
If not tuned correctly, fapolicyd will throttle IBM MQ's I/O, Splunk will fight MQ for disk access, and SELinux will silently drop packets or deny file access.
Here are the specific performance factors and solutions for this stack.
1. The "fapolicyd" I/O Tax
The Conflict: IBM MQ is an I/O-heavy application. It constantly opens and closes queue files, transaction logs, and internal process communications.
fapolicyd uses the fanotify kernel API to intercept every single open() and exec() call to check if the file is trusted.

* Performance Hit: In a high-volume MQ environment, fapolicyd becomes a bottleneck. If the fapolicyd internal queue fills up, it forces the kernel to wait, causing MQ transactions to hang or time out.

The Solution:

* Tune the Queue: You must increase the q_size in /etc/fapolicyd/fapolicyd.conf. The default is often too low for middleware like MQ. Bump it significantly (e.g., to 32768 or higher) to prevent queue overflows.
* Trust the Directory (Don't Check Files individually): Instead of trusting individual MQ binaries, add a trusted rule for the entire MQ installation path (/opt/mqm and /var/mqm) so fapolicyd doesn't waste cycles re-verifying internal MQ files repeatedly.
* Cache Tuning: Increase the obj_cache_size in fapolicyd.conf so it remembers "safe" MQ files longer.

2. Splunk vs. MQ (The "I/O Storm")

The Conflict: IBM MQ writes transaction logs (recovery logs) and error logs (FDC files) to disk. Splunk Universal Forwarder tries to read these files as soon as they are written.

* Performance Hit:
  * Disk Contention: Splunk fighting MQ for disk IOPS can increase "Disk Wait" times, causing MQ to back up.
  * Double Security Check: When Splunk tries to open an MQ log file, fapolicyd intercepts that request too. Now you have two fapolicyd checks for every log event (one for MQ writing, one for Splunk reading).

The Solution:

* Ignore the "Noise": Do not let Splunk ingest the MQ AMQERR*.LOG files blindly. These can be massive during an incident. Use a whitelist in your Splunk inputs.conf to only ingest critical errors, or route these logs to a separate partition.
* Dedicated Disks: Ensure MQ data (/var/mqm) is on a separate LUN/Disk from the OS and Splunk.
* Splunk Throttling: Limit the Splunk Forwarder's bandwidth (maxKBps in limits.conf) so it doesn't starve MQ of I/O cycles during log bursts.

3. Chef "Thundering Herd" & Recursive Checks

The Conflict: Chef runs periodically (e.g., every 15-30 minutes) to ensure the system is in the desired state.

* Performance Hit: If you write a Chef recipe that says "ensure the owner of directory /var/mqm is mqm:mqm," Chef may attempt to recursively check the permissions of millions of messages and log files. This will spike CPU to 100% and freeze the MQ Queue Manager.

The Solution:

* Disable Recursion: When managing the MQ data directory in Chef, strictly use recursive false. Only check the top-level folder permissions.
* Splay Runs: Ensure your Chef runs are "splayed" (randomized start times) so that all your MQ servers don't launch a CPU-intensive Chef client run at the exact same second.

4. SELinux Context Switching

The Conflict: IBM MQ uses shared memory and semaphores heavily. If SELinux contexts are wrong, it will deny access. Even if it allows access, a high volume of "AVC Denial" logging for non-fatal issues can flood /var/log/audit/audit.log.

* Performance Hit: Massive audit logging slows down the system (I/O) and fills up the disk, potentially crashing auditd, which (depending on configuration) might panic the kernel or halt the system.

The Solution:

* Proper Contexts: Ensure mqm_var_lib_t and mqm_exec_t contexts are correctly applied via Chef.
* DontAudit Rules: If you see benign SELinux denials that don't impact functionality but flood the logs, use dontaudit rules to silence them and save CPU/Disk cycles.

Summary of Tuning Required

Component	Tuning Parameter	Recommended Action for MQ Support
fapolicyd	q_size	Increase to >32000 to handle MQ file open bursts.
fapolicyd	Rules	Trust the entire /opt/mqm path to reduce check overhead.
Splunk	limits.conf	Set maxKBps to prevent Splunk from consuming all disk I/O.
Chef	directory resource	NEVER use recursive true on /var/mqm.
RHEL 8	ulimit	Set nofile (open files) to 10,240+ for the mqm user.
RHEL 8	sysctl	Tune fs.file-max and kernel.sem (semaphores) for MQ needs.

Would you like a specific fapolicyd.rules snippet that safely whitelists IBM MQ while keeping the rest of the system secure?

—-_—----------

Running the Chef client every 2 hours will technically work and will definitely help with the performance contention we discussed, but it introduces a significant availability risk for this specific stack (RHEL 8 + fapolicyd + IBM MQ).

Here is the breakdown of the trade-offs for a 2-hour interval versus the standard 30-minute interval.

1. The Performance Win (Pros)

Moving to a 2-hour interval is excellent for reducing "background noise" on high-performance MQ servers.

* Reduced CPU Spikes: Chef is Ruby-based and CPU-intensive. It has to load the entire resource compilation every time it runs. Doing this 12 times a day (every 2 hours) instead of 48 times a day (every 30 mins) recovers significant CPU cycles for MQ.
* Quieter Logs: You generate 75% less logging noise in /var/log/messages and Splunk, making it easier to spot real application errors.

2. The Availability Risk (Cons)

The danger lies in the connection between fapolicyd and Chef.

In this stack, Chef is likely your "fixer." If something breaks or drifts, Chef puts it back.

* The "Update" Trap: Let's say a junior admin or an automated patch process updates an MQ binary or a Splunk config file.
  * Scenario: The file hash changes.
  * Reaction: fapolicyd sees a new, unknown file and blocks it.
  * Result: IBM MQ or Splunk stops working immediately.

* The 2-Hour Problem: If Chef is the tool responsible for updating the fapolicyd trust database or fixing the SELinux context, your system stays down for up to 2 hours until the next scheduled run occurs.

3. The "Drift" Window

With a 2-hour interval, your Mean Time to Recovery (MTTR) for configuration drift increases dramatically.

* If someone accidentally changes permissions on /var/mqm (breaking MQ), the system remains broken for 119 minutes in the worst-case scenario.
* In a 30-minute window, the system self-heals much faster.

Recommendation: The Hybrid Approach

If you want the performance benefits of a 2-hour run without the availability risk, you should implement one of the following strategies:

Option A: Optimized 30-Minute Runs (Preferred)

Keep the 30-minute interval but modify your Chef recipes to be "lightweight."

* Action: Wrap heavy resource checks (like verifying permissions on large MQ directories) in a conditional block so they only run once a day, while keeping critical security checks (like fapolicyd rules) running every 30 minutes.

Option B: 2-Hour Runs + Event-Driven Triggering

Set the schedule to 2 hours, but ensure you have an external trigger mechanism.

* Action: If you deploy a change or patch the system, you must have an automation step (like Ansible, Rundeck, or knife ssh) that forces a chef-client run immediately. You cannot rely on the scheduled daemon.

Summary Verdict

Metric	30-Minute Interval	2-Hour Interval
MQ Performance	Lower (Frequent CPU spikes)	Higher (Less overhead)
Security Compliance	High (Drift fixed quickly)	Low (Drift lasts hours)
Risk of Outage	Low (Self-heals quickly)	High (System stays down if blocked)

My advice: For a critical IBM MQ server protected by fapolicyd, stick to 30 minutes (or 60 at most). The risk of fapolicyd accidentally blocking a critical process for 2 hours is too high compared to the CPU savings.

Would you like me to show you how to use a Chef "Guard" to make the heavy MQ permission checks run only once a day, while keeping the rest of the run fast?

In Chef, a Guard is a built-in property available on any Chef resource (like package, service, execute, or template) that evaluates a condition at runtime to determine whether that resource should execute.

It is the primary mechanism Chef uses to ensure idempotency (the ability to run the same code multiple times without changing the result if the system is already correct).

The Two Types of Guards

There are exactly two guard properties you can use:

 * only_if: The resource will execute only if the condition returns true.
   * Think: "Do this strict thing only if this specific condition is met."
 * not_if: The resource will skip execution if the condition returns true.
   * Think: "Don't do this if it's already done."

How It Works

When Chef runs, it goes through a "Compile Phase" (reading your code) and an "Execute Phase" (running the actions). Guards are evaluated during the Execute Phase.

You can write guards in two ways:

1. Command Guard (String)

Chef runs a shell command. If the command returns exit code 0 (success), the condition is true.

```
# Restart Splunk ONLY IF the config file is valid
service 'splunk' do
  action :restart
  # Runs the command. If exit code is 0 (true), the service restarts.
  only_if '/opt/splunk/bin/splunk btool check'
end
```

2. Ruby Block Guard (Code)

Chef executes raw Ruby code. If the code returns true, the condition is met. This is what we used in your MQ solution.

```
# Create a user ONLY IF a specific directory exists
user 'mqm' do
  action :create
  # Ruby logic: returns true or false
  only_if { File.exist?('/opt/mqm') }
end
```

Why "Guards" Are Critical for Your Stack

In the context of your IBM MQ + fapolicyd + Chef stack, guards are the safety valve preventing performance disasters.

Without guards, Chef is "dumb"—it tries to enforce the state every single time it runs.

 * Without a Guard: Chef runs restorecon -R /var/mqm every 30 minutes. It scans 100,000 files, spiking CPU and possibly locking MQ files.
 * With a Guard (not_if): Chef checks a tiny timestamp file first. If it was touched recently, Chef skips the entire resource. The heavy restorecon command never even launches.

Summary Table

Guard	Logic	Example Usage
only_if	Run YES if True	"Run this upgrade command only if the version is old."
not_if	Run NO if True	"Do not download this file if it already exists on disk."

Would you like to see how to combine multiple guards on a single resource for complex logic?