# Upgrading to the latest release

If you’re familiar with previous releases of Hyperledger Fabric, you’re aware that upgrading the nodes and channels to the latest version of Fabric is, at a high level, a four step process.

1. Backup the ledger and MSPs.
2. Upgrade the orderer binaries in a rolling fashion to the latest Fabric version.
3. Upgrade the peer binaries in a rolling fashion to the latest Fabric version.
4. Update the orderer system channel and any application channels to the latest capability levels, where available. Note that some releases will have capabilities in all groups while other releases may have few or even no new capabilities at all.

For more information about capabilities, check out [Channel capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/capabilities_concept.html).

For a look at how these upgrade processes are accomplished, please consult these tutorials:

1. [Upgrading your components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html). Components should be upgraded to the latest version before updating any capabilities.
2. [Updating the capability level of a channel](https://hyperledger-fabric.readthedocs.io/en/latest/updating_capabilities.html). Completed after updating the versions of all nodes.
3. [Enabling the new chaincode lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/enable_cc_lifecycle.html). Necessary to add organization specific endorsement policies central to the new chaincode lifecycle for Fabric v2.0.

As the upgrading of nodes and increasing the capability levels of channels is by now considered a standard Fabric process, we will not show the specific commands for upgrading to the newest release. Similarly, there is no script in the fabric-samples repo that will upgrade a sample network from the previous release to this one, as there has been for previous releases.

**Note**

It is a best practice to upgrade your SDK to the latest version as a part of a general upgrade of your network. While the SDK will always be compatible with equivalent releases of Fabric and lower, it might be necessary to upgrade to the latest SDK to leverage the latest Fabric features. Consult the documentation of the Fabric SDK you are using for information about how to upgrade.

**Upgrading to the latest release**

* [Considerations for getting to v2.0](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html)
* [Upgrading your components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html)
* [Updating the capability level of a channel](https://hyperledger-fabric.readthedocs.io/en/latest/updating_capabilities.html)
* [Enabling the new chaincode lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/enable_cc_lifecycle.html)

# Considerations for getting to v2.0

## Chaincode lifecycle

The new chaincode lifecycle that debuts in v2.0 allows multiple organizations to agree on how a chaincode will be operated before it can be used on a channel. For more information about the new chaincode lifecycle, check out [Chaincode for operators](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html).

It is a best practice to upgrade all of the peers on a channel before enabling the Channel and Application capabilities that enable the new chaincode lifecycle (the Channel capability is not strictly required, but it makes sense to update it at this time). Note that any peers that are not at v2.0 will crash after enabling either capability, while any ordering nodes that are not at v2.0 will crash after the Channel capability has been enabled. This crashing behavior is intentional, as the peer or orderer cannot safely participate in the channel if it does not support the required capabilities.

After the Application capability has been updated to V2\_0 on a channel, you must use the v2.0 lifecycle procedures to package, install, approve, and commit new chaincodes on the channel. As a result, make sure to be prepared for the new lifecycle before updating the capability.

The new lifecycle defaults to using the endorsement policy configured in the channel config (e.g., a MAJORITY of orgs). Therefore this endorsement policy should be added to the channel configuration when enabling capabilities on the channel.

For information about how to edit the relevant channel configurations to enable the new lifecycle by adding an endorsement policy for each organization, check out [Enabling the new chaincode lifecycle](https://hyperledger-fabric.readthedocs.io/en/latest/enable_cc_lifecycle.html).

## Chaincode shim changes (Go chaincode only)

The recommended approach is to vendor the shim in your v1.4 Go chaincode before making upgrades to the peers and channels. If you do this, you do not need to make any additional changes to your chaincode.

If you did not vendor the shim in your v1.4 chaincode, the old v1.4 chaincode images will still technically work after upgrade, but you are in a risky state. If the chaincode image gets deleted from your environment for whatever reason, the next invoke on v2.0 peer will try to rebuild the chaincode image and you’ll get an error that the shim cannot be found.

At this point, you have two options:

1. If the entire channel is ready to upgrade chaincode, you can upgrade the chaincode on all peers and on the channel (using either the old or new lifecycle depending on the Application capability level you have enabled). The best practice at this point would be to vendor the new Go chaincode shim using modules.
2. If the entire channel is not yet ready to upgrade the chaincode, you can use peer environment variables to specify the v1.4 chaincode environment ccenv be used to rebuild the chaincode images. This v1.4 ccenv should still work with a v2.0 peer.

## Chaincode logger (Go chaincode only)

Support for user chaincodes to utilize the chaincode shim’s logger via NewLogger() has been removed. Chaincodes that used the shim’s NewLogger() must now shift to their own preferred logging mechanism.

For more information, check out [Logging control](https://hyperledger-fabric.readthedocs.io/en/latest/logging-control.html#chaincode).

## Peer databases upgrade

The databases of all peers (which include not just the state database but the history database and other internal databases for the peer) must be rebuilt using the v2.0 data format as part of the upgrade to v2.0. To trigger the rebuild, the databases must be dropped before the peer is started.

For information about how to upgrade peers, check out our documentation on [upgrading components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html). During the process for [upgrading your peers](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html#upgrade-the-peers), you will need to pass a peer node upgrade-dbs command to drop the databases of the peer.

Follow the commands to upgrade a peer until the docker run command you see to launch the new peer container (you can skip the step where you set an IMAGE\_TAG, since the upgrade dbs command is for the v2.0 release of Fabric only, but you will need to set the PEER\_CONTAINER and LEDGERS\_BACKUP environment variables). Instead of the docker run command to launch the peer, run this one instead:

docker run --rm -v /opt/backup/$PEER\_CONTAINER/:/var/hyperledger/production/ \

-v /opt/msp/:/etc/hyperledger/fabric/msp/ \

--env-file ./env<name of node>.list \

--name $PEER\_CONTAINER \

hyperledger/fabric-peer:2.0 peer node upgrade-dbs

This will drop the databases of the peer. Then issue this command to start the peer using the 2.0 tag:

docker run -d -v /opt/backup/$PEER\_CONTAINER/:/var/hyperledger/production/ \

-v /opt/msp/:/etc/hyperledger/fabric/msp/ \

--env-file ./env<name of node>.list \

--name $PEER\_CONTAINER \

hyperledger/fabric-peer:2.0 peer node start

Because rebuilding the databases can be a lengthy process (several hours, depending on the size of your databases), monitor the peer logs to check the status of the rebuild. Every 1000th block you will see a message like [lockbasedtxmgr] CommitLostBlock -> INFO 041 Recommitting block [1000] to state database indicating the rebuild is ongoing.

If the database is not dropped as part of the upgrade process, the peer start will return an error message stating that its databases are in the old format and must be dropped using the peer node upgrade-dbs command above. The node will then need to be restarted.

## Capabilities

As can be expected for a 2.0 release, there is a full complement of new capabilities for 2.0.

* **Application** V2\_0: enables the new chaincode lifecycle as described in [Chaincode for Operators](https://hyperledger-fabric.readthedocs.io/en/latest/chaincode4noah.html).
* **Channel** V2\_0: this capability has no changes, but is used for consistency with the application and orderer capability levels.
* **Orderer** V2\_0: controls UseChannelCreationPolicyAsAdmins, changing the way that channel creation transactions are validated. When combined with the -baseProfile option of configtxgen, values which were previously inherited from the orderer system channel may now be overridden.

As with any update of the capability levels, make sure to upgrade your peer binaries before updating the Application and Channel capabilities, and make sure to upgrade your orderer binaries before updating the Orderer and Channel capabilities.

For information about how to set new capabilities, check out [Updating the capability level of a channel](https://hyperledger-fabric.readthedocs.io/en/latest/updating_capabilities.html).

## Define ordering node endpoint per org (recommend)

Starting with version v1.4.2, it was recommended to define orderer endpoints in both the system channel and in all application channels at the organization level by adding a new OrdererEndpoints stanza within the channel configuration of an organization, replacing the the global OrdererAddresses section of channel configuration. If at least one organization has an ordering service endpoint defined at an organizational level, all orderers and peers will ignore the channel level endpoints when connecting to ordering nodes.

Utilizing organization level orderer endpoints is required when using service discovery with ordering nodes provided by multiple organizations. This allows clients to provide the correct organization TLS certificates.

If your channel configuration does not yet include OrdererEndpoints per org, you will need to perform a channel configuration update to add them to the config. First, create a JSON file that includes the new configuration stanza.

In this example, we will create a stanza for a single org called OrdererOrg. Note that if you have multiple ordering service organizations, they will all have to be updated to include endpoints. Let’s call our JSON file orglevelEndpoints.json.

{

"OrdererOrgEndpoint": {

"Endpoints": {

"mod\_policy": "Admins",

"value": {

"addresses": [

"127.0.0.1:30000"

]

}

}

}

}

Then, export the following environment variables:

* CH\_NAME: the name of the channel being updated. Note that all system channels and application channels should contain organization endpoints for ordering nodes.
* CORE\_PEER\_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
* CORE\_PEER\_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
* TLS\_ROOT\_CA: the absolute path to the root CA certificate of the organization proposing the system channel update.
* ORDERER\_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
* ORGNAME: The name of the organization you are currently updating. For example, OrdererOrg.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config).

Once you have a modified\_config.json, add the lifecycle organization policy (as listed in orglevelEndpoints.json) using this command:

jq **-**s ".[0] \* {\"channel\_group\":{\"groups\":{\"Orderer\": {\"groups\": {\"$ORGNAME\": {\"values\": .[1].${ORGNAME}Endpoint}}}}}}" config**.**json **./**orglevelEndpoints**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

If every ordering service organization performs their own channel edit, they can edit the configuration without needing further signatures (by default, the only signature needed to edit parameters within an organization is an admin of that organization). If a different organization proposes the update, then the organization being edited will need to sign the channel update request.

# Upgrading your components

Audience: network administrators, node administrators

For information about special considerations for the latest release of Fabric, check out [Upgrading to the latest release of Fabric](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html).

This topic will only cover the process for upgrading components. For information about how to edit a channel to change the capability level of your channels, check out [Updating a channel capability](https://hyperledger-fabric.readthedocs.io/en/latest/updating_capabilities.html).

Note: when we use the term “upgrade” in Hyperledger Fabric, we’re referring to changing the version of a component (for example, going from one version of a binary to the next version). The term “update,” on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term “migration” or “migrate” here.

## Overview

At a high level, upgrading the binary level of your nodes is a two step process:

1. Backup the ledger and MSPs.
2. Upgrade binaries to the latest version.

If you own both ordering nodes and peers, it is a best practice to upgrade the ordering nodes first. If a peer falls behind or is temporarily unable to process certain transactions, it can always catch up. If enough ordering nodes go down, by comparison, a network can effectively cease to function.

This topic presumes that these steps will be performed using Docker CLI commands. If you are utilizing a different deployment method (Rancher, Kubernetes, OpenShift, etc) consult their documentation on how to use their CLI.

For native deployments, note that you will also need to update the YAML configuration file for the nodes (for example, the orderer.yaml file) with the one from the release artifacts.

To do this, backup the orderer.yaml or core.yaml file (for the peer) and replace it with the orderer.yaml or core.yaml file from the release artifacts. Then port any modified variables from the backed up orderer.yaml or core.yaml to the new one. Using a utility like diff may be helpful. Note that updating the YAML file from the release rather than updating your old YAML file **is the recommended way to update your node YAML files**, as it reduces the likelihood of making errors.

This tutorial assumes a Docker deployment where the YAML files will be baked into the images and environment variables will be used to overwrite the defaults in the configuration files.

## Environment variables for the binaries

When you deploy a peer or an ordering node, you had to set a number of environment variables relevant to its configuration. A best practice is to create a file for these environment variables, give it a name relevant to the node being deployed, and save it somewhere on your local file system. That way you can be sure that when upgrading the peer or ordering node you are using the same variables you set when creating it.

Here’s a list of some of the **peer** environment variables (with sample values — as you can see from the addresses, these environment variables are for a network deployed locally) that can be set that be listed in the file. Note that you may or may not need to set all of these environment variables:

CORE\_PEER\_TLS\_ENABLED**=**true

CORE\_PEER\_GOSSIP\_USELEADERELECTION**=**true

CORE\_PEER\_GOSSIP\_ORGLEADER**=**false

CORE\_PEER\_PROFILE\_ENABLED**=**true

CORE\_PEER\_TLS\_CERT\_FILE**=/**etc**/**hyperledger**/**fabric**/**tls**/**server**.**crt

CORE\_PEER\_TLS\_KEY\_FILE**=/**etc**/**hyperledger**/**fabric**/**tls**/**server**.**key

CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**etc**/**hyperledger**/**fabric**/**tls**/**ca**.**crt

CORE\_PEER\_ID**=**peer0**.**org1**.**example**.**com

CORE\_PEER\_ADDRESS**=**peer0**.**org1**.**example**.**com:7051

CORE\_PEER\_LISTENADDRESS**=**0.0**.**0.0:7051

CORE\_PEER\_CHAINCODEADDRESS**=**peer0**.**org1**.**example**.**com:7052

CORE\_PEER\_CHAINCODELISTENADDRESS**=**0.0**.**0.0:7052

CORE\_PEER\_GOSSIP\_BOOTSTRAP**=**peer0**.**org1**.**example**.**com:7051

CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT**=**peer0**.**org1**.**example**.**com:7051

CORE\_PEER\_LOCALMSPID**=**Org1MSP

Here are some **ordering node** variables (again, these are sample values) that might be listed in the environment variable file for a node. Again, you may or may not need to set all of these environment variables:

ORDERER\_GENERAL\_LISTENADDRESS**=**0.0**.**0.0

ORDERER\_GENERAL\_GENESISMETHOD**=**file

ORDERER\_GENERAL\_GENESISFILE**=/**var**/**hyperledger**/**orderer**/**orderer**.**genesis**.**block

ORDERER\_GENERAL\_LOCALMSPID**=**OrdererMSP

ORDERER\_GENERAL\_LOCALMSPDIR**=/**var**/**hyperledger**/**orderer**/**msp

ORDERER\_GENERAL\_TLS\_ENABLED**=**true

ORDERER\_GENERAL\_TLS\_PRIVATEKEY**=/**var**/**hyperledger**/**orderer**/**tls**/**server**.**key

ORDERER\_GENERAL\_TLS\_CERTIFICATE**=/**var**/**hyperledger**/**orderer**/**tls**/**server**.**crt

ORDERER\_GENERAL\_TLS\_ROOTCAS**=**[**/**var**/**hyperledger**/**orderer**/**tls**/**ca**.**crt]

ORDERER\_GENERAL\_CLUSTER\_CLIENTCERTIFICATE**=/**var**/**hyperledger**/**orderer**/**tls**/**server**.**crt

ORDERER\_GENERAL\_CLUSTER\_CLIENTPRIVATEKEY**=/**var**/**hyperledger**/**orderer**/**tls**/**server**.**key

ORDERER\_GENERAL\_CLUSTER\_ROOTCAS**=**[**/**var**/**hyperledger**/**orderer**/**tls**/**ca**.**crt]

However you choose to set your environment variables, note that they will have to be set for each node you want to upgrade.

## Ledger backup and restore

While we will demonstrate the process for backing up ledger data in this tutorial, it is not strictly required to backup the ledger data of a peer or an ordering node (assuming the node is part of a larger group of nodes in an ordering service). This is because, even in the worst case of catastrophic failure of a peer (such as a disk failure), the peer can be brought up with no ledger at all. You can then have the peer re-join the desired channels and as a result, the peer will automatically create a ledger for each of the channels and will start receiving the blocks via regular block transfer mechanism from either the ordering service or the other peers in the channel. As the peer processes blocks, it will also build up its state database.

However, backing up ledger data enables the restoration of a peer without the time and computational costs associated with bootstrapping from the genesis block and reprocessing all transactions, a process that can take hours (depending on the size of the ledger). In addition, ledger data backups may help to expedite the addition of a new peer, which can be achieved by backing up the ledger data from one peer and starting the new peer with the backed up ledger data.

This tutorial presumes that the file path to the ledger data has not been changed from the default value of /var/hyperledger/production/ (for peers) or /var/hyperledger/production/orderer (for ordering nodes). If this location has been changed for your nodes, enter the path to the data on your ledgers in the commands below.

Note that there will be data for both the ledger and chaincodes at this file location. While it is a best practice to backup both, it is possible to skip the stateLeveldb, historyLeveldb, chains/index folders at /var/hyperledger/production/ledgersData. While skipping these folders reduces the storage needed for the backup, the peer recovery from the backed up data may take more time as these ledger artifacts will be re-constructed when the peer starts.

If using CouchDB as state database, there will be no stateLeveldb directory, as the state database data would be stored within CouchDB instead. But similarly, if peer starts up and finds CouchDB databases are missing or at lower block height (based on using an older CouchDB backup), the state database will be automatically re-constructed to catch up to current block height. Therefore, if you backup peer ledger data and CouchDB data separately, ensure that the CouchDB backup is always older than the peer backup.

## Upgrade ordering nodes

Orderer containers should be upgraded in a rolling fashion (one at a time). At a high level, the ordering node upgrade process goes as follows:

1. Stop the ordering node.
2. Back up the ordering node’s ledger and MSP.
3. Remove the ordering node container.
4. Launch a new ordering node container using the relevant image tag.

Repeat this process for each node in your ordering service until the entire ordering service has been upgraded.

### Set command environment variables

Export the following environment variables before attempting to upgrade your ordering nodes.

* ORDERER\_CONTAINER: the name of your ordering node container. Note that you will need to export this variable for each node when upgrading it.
* LEDGERS\_BACKUP: the place in your local filesystem where you want to store the ledger being backed up. As you will see below, each node being backed up will have its own subfolder containing its ledger. You will need to create this folder.
* IMAGE\_TAG: the Fabric version you are upgrading to. For example, 2.0.

Note that you will have to set an **image tag** to ensure that the node you are starting using the correct images. The process you use to set the tag will depend on your deployment method.

### Upgrade containers

Let’s begin the upgrade process by **bringing down the orderer**:

docker stop $ORDERER\_CONTAINER

Once the orderer is down, you’ll want to **backup its ledger and MSP**:

docker cp $ORDERER\_CONTAINER:/var/hyperledger/production/orderer/ ./$LEDGERS\_BACKUP/$ORDERER\_CONTAINER

Then remove the ordering node container itself (since we will be giving our new container the same name as our old one):

docker rm -f $ORDERER\_CONTAINER

Then you can launch the new ordering node container by issuing:

docker run -d -v /opt/backup/$ORDERER\_CONTAINER/:/var/hyperledger/production/orderer/ \

-v /opt/msp/:/etc/hyperledger/fabric/msp/ \

--env-file ./env<name of node>.list \

--name $ORDERER\_CONTAINER \

hyperledger/fabric-orderer:$IMAGE\_TAG orderer

Once all of the ordering nodes have come up, you can move on to upgrading your peers.

## Upgrade the peers

Peers should, like the ordering nodes, be upgraded in a rolling fashion (one at a time). As mentioned during the ordering node upgrade, ordering nodes and peers may be upgraded in parallel, but for the purposes of this tutorial we’ve separated the processes out. At a high level, we will perform the following steps:

1. Stop the peer.
2. Back up the peer’s ledger and MSP.
3. Remove chaincode containers and images.
4. Remove the peer container.
5. Launch a new peer container using the relevant image tag.

### Set command environment variables

Export the following environment variables before attempting to upgrade your peers.

* PEER\_CONTAINER: the name of your peer container. Note that you will need to set this variable for each node.
* LEDGERS\_BACKUP: the place in your local filesystem where you want to store the ledger being backed up. As you will see below, each node being backed up will have its own subfolder containing its ledger. You will need to create this folder.
* IMAGE\_TAG: the Fabric version you are upgrading to. For example, 2.0.

Note that you will have to set an **image tag** to ensure that the node you are starting is using the correct images. The process you use to set the tag will depend on your deployment method.

Repeat this process for each of your peers until every node has been upgraded.

### Upgrade containers

Let’s **bring down the first peer** with the following command:

docker stop $PEER\_CONTAINER

We can then **backup the peer’s ledger and MSP**:

docker cp $PEER\_CONTAINER:/var/hyperledger/production ./$LEDGERS\_BACKUP/$PEER\_CONTAINER

With the peer stopped and the ledger backed up, **remove the peer chaincode containers**:

CC\_CONTAINERS=$(docker ps | grep dev-$PEER\_CONTAINER | awk '{print $1}')

if [ -n "$CC\_CONTAINERS" ] ; then docker rm -f $CC\_CONTAINERS ; fi

And the peer chaincode images:

CC\_IMAGES=$(docker images | grep dev-$PEER | awk '{print $1}')

if [ -n "$CC\_IMAGES" ] ; then docker rmi -f $CC\_IMAGES ; fi

Then remove the peer container itself (since we will be giving our new container the same name as our old one):

docker rm -f $PEER\_CONTAINER

Then you can launch the new peer container by issuing:

docker run -d -v /opt/backup/$PEER\_CONTAINER/:/var/hyperledger/production/ \

-v /opt/msp/:/etc/hyperledger/fabric/msp/ \

--env-file ./env<name of node>.list \

--name $PEER\_CONTAINER \

hyperledger/fabric-peer:$IMAGE\_TAG peer node start

You do not need to relaunch the chaincode container. When the peer gets a request for a chaincode, (invoke or query), it first checks if it has a copy of that chaincode running. If so, it uses it. Otherwise, as in this case, the peer launches the chaincode (rebuilding the image if required).

### Verify peer upgrade completion

It’s a best practice to ensure the upgrade has been completed properly with a chaincode invoke. Note that it should be possible to verify that a single peer has been successfully updated by querying one of the ledgers hosted on the peer. If you want to verify that multiple peers have been upgraded, and are updating your chaincode as part of the upgrade process, you should wait until peers from enough organizations to satisfy the endorsement policy have been upgraded.

Before you attempt this, you may want to upgrade peers from enough organizations to satisfy your endorsement policy. However, this is only mandatory if you are updating your chaincode as part of the upgrade process. If you are not updating your chaincode as part of the upgrade process, it is possible to get endorsements from peers running at different Fabric versions.

## Upgrade your CAs

To learn how to upgrade your Fabric CA server, click over to the [CA documentation](http://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html#upgrading-the-server).

## Upgrade Node SDK clients

Upgrade Fabric and Fabric CA before upgrading Node SDK clients. Fabric and Fabric CA are tested for backwards compatibility with older SDK clients. While newer SDK clients often work with older Fabric and Fabric CA releases, they may expose features that are not yet available in the older Fabric and Fabric CA releases, and are not tested for full compatibility.

Use NPM to upgrade any Node.js client by executing these commands in the root directory of your application:

npm install fabric**-**client@latest

npm install fabric**-**ca**-**client@latest

These commands install the new version of both the Fabric client and Fabric-CA client and write the new versions to package.json.

## Upgrading CouchDB

If you are using CouchDB as state database, you should upgrade the peer’s CouchDB at the same time the peer is being upgraded.

To upgrade CouchDB:

1. Stop CouchDB.
2. Backup CouchDB data directory.
3. Install the latest CouchDB binaries or update deployment scripts to use a new Docker image.
4. Restart CouchDB.

## Upgrade Node chaincode shim

To move to the new version of the Node chaincode shim a developer would need to:

1. Change the level of fabric-shim in their chaincode package.json from their old level to the new one.
2. Repackage this new chaincode package and install it on all the endorsing peers in the channel.
3. Perform an upgrade to this new chaincode. To see how to do this, check out [Peer chaincode commands](https://hyperledger-fabric.readthedocs.io/en/latest/commands/peerchaincode.html).

## Upgrade Chaincodes with vendored shim

For information about upgrading the Go chaincode shim specific to the v2.0 release, check out [Chaincode shim changes](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html#chaincode-shim-changes).

A number of third party tools exist that will allow you to vendor a chaincode shim. If you used one of these tools, use the same one to update your vendored chaincode shim and re-package your chaincode.

If your chaincode vendors the shim, after updating the shim version, you must install it to all peers which already have the chaincode. Install it with the same name, but a newer version. Then you should execute a chaincode upgrade on each channel where this chaincode has been deployed to move to the new version.

# Updating the capability level of a channel

Audience: network administrators, node administrators

If you’re not familiar with capabilities, check out [Capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/capabilities_concept.html) before proceeding, paying particular attention to the fact that **peers and orderers that belong to the channel must be upgraded before enabling capabilities**.

For information about any new capability levels in the latest release of Fabric, check out [Upgrading your components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html#Capabilities).

Note: when we use the term “upgrade” in Hyperledger Fabric, we’re referring to changing the version of a component (for example, going from one version of a binary to the next version). The term “update,” on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term “migration” or “migrate” here.

## Prerequisites and considerations

If you haven’t already done so, ensure you have all of the dependencies on your machine as described in [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/latest/prereqs.html). This will ensure that you have the latest versions of the tools required to make a channel configuration update.

Although Fabric binaries can and should be upgraded in a rolling fashion, it is important to **finish upgrading binaries before enabling capabilities**. Any binaries which are not upgraded to at least the level of the relevant capabilities will crash to indicate a misconfiguration which could otherwise result in a ledger fork.

Once a capability has been enabled, it becomes part of the permanent record for that channel. This means that even after disabling the capability, old binaries will not be able to participate in the channel because they cannot process beyond the block which enabled the capability to get to the block which disables it. As a result, once a capability has been enabled, disabling it is neither recommended nor supported.

For this reason, think of enabling channel capabilities as a point of no return. Please experiment with the new capabilities in a test setting and be confident before proceeding to enable them in production.

## Overview

In this tutorial, we will show the process for updating capabilities in all of the parts of the configuration of both the ordering system channel and any application channels.

Whether you will need to update every part of the configuration for all of your channels will depend on the contents of the latest release as well as your own use case. For more information, check out [Upgrading to the latest version of Fabric](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html). Note that it may be necessary to update to the newest capability levels before using the features in the latest release, and it is considered a best practice to always be at the latest binary versions and capability levels.

Because updating the capability level of a channel involves the configuration update transaction process, we will be relying on our [Updating a channel configuration](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html) topic for many of the commands.

As with any channel configuration update, updating capabilities is, at a high level, a three step process (for each channel):

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

We will enable these capabilities in the following order:

1. [Orderer system channel](https://hyperledger-fabric.readthedocs.io/en/latest/updating_capabilities.html#orderer-system-channel-capabilities)

* Orderer group
* Channel group

1. [Application channels](https://hyperledger-fabric.readthedocs.io/en/latest/updating_capabilities.html#enable-capabilities-on-existing-channels)

* Orderer group
* Channel group
* Application group

While it is possible to edit multiple parts of the configuration of a channel at the same time, in this tutorial we will show how this process is done incrementally. In other words, we will not bundle a change to the Orderer group and the Channel group of the system channel into one configuration change. This is because not every release will have both a new Orderer group capability and a Channel group capability.

Note that in production networks, it will not be possible or desirable for one user to be able to update all of these channels (and parts of configurations) unilaterally. The orderer system channel, for example, is administered exclusively by ordering organization admins (though it is possible to add peer organizations as ordering service organizations). Similarly, updating either the Orderer or Channel groups of a channel configuration requires the signature of an ordering service organization in addition to peer organizations. Distributed systems require collaborative management.

### Create a capabilities config file

Note that this tutorial presumes that a file called capabilities.json has been created and includes the capability updates you want to make to the various sections of the config. It also uses jq to apply the edits to the modified config file.

Note that you are not obligated to create a file like capabilities.json or to use a tool like jq. The modified config can also be edited manually (after it has been pulled, translated, and scoped). Check out this [sample channel configuration](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#sample-channel-configuration) for reference.

However, the process described here (using a JSON file and a tool like jq) does have the advantage of being scriptable, making it suitable for proposing configuration updates to a large number of channels. This is why it is **the recommended way to update channels**.

In this example, the capabilities.json file looks like this (note: if you are updating your channel as part of [Upgrading to the latest version of Fabric](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html) you will need to set the capabilities to the levels appropriate to that release):

{

"channel": {

"mod\_policy": "Admins",

"value": {

"capabilities": {

"V2\_0": {}

}

},

"version": "0"

},

"orderer": {

"mod\_policy": "Admins",

"value": {

"capabilities": {

"V2\_0": {}

}

},

"version": "0"

},

"application": {

"mod\_policy": "Admins",

"value": {

"capabilities": {

"V2\_0": {}

}

},

"version": "0"

}

}

Note that by default peer organizations are not admins of the orderer system channel and will therefore be unable to propose configuration updates to it. An orderer organization admin would have to create a file like this (without the application group capability, which does not exist in the system channel) to propose updating the system channel configuration. Note that because application channel copy the system channel configuration by default, unless a different channel profile is created which specifies capability levels, the Channel and Orderer group capabilities for the application channel will be the same as those in the network’s system channel.

## Orderer system channel capabilities

Because application channels copy the configuration of the orderer system channel by default, it is considered a best practice to update the capabilities of the system channel before any application channels. This mirrors the process of updating ordering nodes to the newest version before peers, as described in [Upgrading your components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html).

Note that the orderer system channel is administered by ordering service organizations. By default this will be a single organization (the organization that created the initial nodes in the ordering service), but more organizations can be added here (for example, if multiple organizations have contributed nodes to the ordering service).

Make sure all of the ordering nodes in your ordering service have been upgraded to the required binary level before updating the Orderer and Channel capability. If an ordering node is not at the required level, it will be unable to process the config block with the capability and will crash. Similarly, note that if a new channel is created on this ordering service, all of the peers that will be joined to it must be at least to the node level corresponding to the Channel and Application capabilities, otherwise they will also crash when attempting to process the config block. For more information, check out [Capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/capabilities_concept.html).

### Set environment variables

You will need to export the following variables:

* CH\_NAME: the name of the system channel being updated.
* CORE\_PEER\_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the orderer organizations.
* TLS\_ROOT\_CA: the absolute path to the TLS cert of your ordering node(s).
* CORE\_PEER\_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
* ORDERER\_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

### Orderer group

For the commands on how to pull, translate, and scope the channel config, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config). Once you have a modified\_config.json, add the capabilities to the Orderer group of the config (as listed in capabilities.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].orderer}}}}}' config**.**json **./**capabilities**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

Note that because you are updating the system channel, the mod\_policy for the system channel will only require the signature of ordering service organization admins.

### Channel group

Once again, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config). Once you have a modified\_config.json, add the capabilities to the Channel group of the config (as listed in capabilities.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"values": {"Capabilities": .[1].channel}}}' config**.**json **./**capabilities**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

Note that because you are updating the system channel, the mod\_policy for the system channel will only require the signature of ordering service organization admins. In an application channel, as you’ll see, you would normally need to satisfy both the MAJORITY Admins policy of both the Application group (consisting of the MSPs of peer organizations) and the Orderer group (consisting of ordering service organizations), assuming you have not changed the default values.

## Enable capabilities on existing channels

Now that we have updating the capabilities on the orderer system channel, we need to updating the configuration of any existing application channels you want to update.

As you will see, the configuration of application channels is very similar to that of the system channel. This is what allows us to re-use capabilities.json and the same commands we used for updating the system channel (using different environment variables which we will discuss below).

**Make sure all of the ordering nodes in your ordering service and peers on the channel have been upgraded to the required binary level before updating capabilities. If a peer or an ordering node is not at the required level, it will be unable to process the config block with the capability and will crash**. For more information, check out [Capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/capabilities_concept.html).

### Set environment variables

You will need to export the following variables:

* CH\_NAME: the name of the application channel being updated. You will have to reset this variable for every channel you update.
* CORE\_PEER\_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of your peer organization.
* TLS\_ROOT\_CA: the absolute path to the TLS cert of your peer organization.
* CORE\_PEER\_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
* ORDERER\_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

### Orderer group

Navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config). Once you have a modified\_config.json, add the capabilities to the Orderer group of the config (as listed in capabilities.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].orderer}}}}}' config**.**json **./**capabilities**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

Note the mod\_policy for this capability defaults to the MAJORITY of the Admins of the Orderer group (in other words, a majority of the admins of the ordering service). Peer organizations can propose an update to this capability, but their signatures will not satisfy the relevant policy in this case.

### Channel group

Navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config). Once you have a modified\_config.json, add the capabilities to the Channel group of the config (as listed in capabilities.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"values": {"Capabilities": .[1].channel}}}' config**.**json **./**capabilities**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

Note that the mod\_policy for this capability defaults to requiring signatures from both the MAJORITY of Admins in the Application and Orderer groups. In other words, both a majority of the peer organization admins and ordering service organization admins must sign this request.

### Application group

Navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config). Once you have a modified\_config.json, add the capabilities to the Application group of the config (as listed in capabilities.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Application": {"values": {"Capabilities": .[1].application}}}}}' config**.**json **./**capabilities**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

Note that the mod\_policy for this capability defaults to requiring signatures from the MAJORITY of Admins in the Application group. In other words, a majority of peer organizations will need to approve. Ordering service admins have no say in this capability.

**As a result, be very careful to not change this capability to a level that does not exist**. Because ordering nodes neither understand nor validate Application capabilities, they will approve a configuration to any level and send the new config block to the peers to be committed to their ledgers. However, the peers will be unable to process the capability and will crash. And even it was possible to drive a corrected configuration change to a valid capability level, the previous config block with the faulty capability would still exist on the ledger and cause peers to crash when trying to process it.

This is one reason why a file like capabilities.json can be useful. It prevents a simple user error — for example, setting the Application capability to V20 when the intent was to set it to V2\_0 — that can cause a channel to be unusable and unrecoverable.

## Verify a transaction after capabilities have been enabled

It’s a best practice to ensure that capabilities have been enabled successfully with a chaincode invoke on all channels. If any nodes that do not understand new capabilities have not been upgraded to a sufficient binary level, they will crash. You will have to upgrade their binary level before they can be successfully restarted.

# Enabling the new chaincode lifecycle

Users upgrading from v1.4.x to v2.0 will have to edit their channel configurations to enable the new lifecycle features. This process involves a series of [channel configuration updates](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html) the relevant users will have to perform.

Note that the Channel and Application [capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/capabilities_concept.html) of your application channels will have to be updated to V2\_0 for the new chaincode lifecycle to work. Check out [Considerations for getting to 2.0](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html#chaincode-lifecycle) for more information.

Updating a channel configuration is, at a high level, a three step process (for each channel):

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

We will be performing these channel configuration updates by leveraging a file called enable\_lifecycle.json, which contains all of the updates we will be making in the channel configurations. Note that in a production setting it is likely that multiple users would be making these channel update requests. However, for the sake of simplicity, we are presenting all of the updates as how they would appear in a single file.

## Create enable\_lifecycle.json

Note that in addition to using enable\_lifecycle.json, this tutorial also uses jq to apply the edits to the modified config file. The modified config can also be edited manually (after it has been pulled, translated, and scoped). Check out this [sample channel configuration](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#sample-channel-configuration) for reference.

However, the process described here (using a JSON file and a tool like jq) does have the advantage of being scriptable, making it suitable for proposing configuration updates to a large number of channels, and is the recommended process for editing a channel configuration.

Note that the enable\_lifecycle.json uses sample values, for example org1Policies and the Org1ExampleCom, which will be specific to your deployment):

{

"org1Policies": {

"Endorsement": {

"mod\_policy": "Admins",

"policy": {

"type": 1,

"value": {

"identities": [

{

"principal": {

"msp\_identifier": "Org1ExampleCom",

"role": "PEER"

},

"principal\_classification": "ROLE"

}

],

"rule": {

"n\_out\_of": {

"n": 1,

"rules": [

{

"signed\_by": 0

}

]

}

},

"version": 0

}

},

"version": "0"

}

},

"org2Policies": {

"Endorsement": {

"mod\_policy": "Admins",

"policy": {

"type": 1,

"value": {

"identities": [

{

"principal": {

"msp\_identifier": "Org2ExampleCom",

"role": "PEER"

},

"principal\_classification": "ROLE"

}

],

"rule": {

"n\_out\_of": {

"n": 1,

"rules": [

{

"signed\_by": 0

}

]

}

},

"version": 0

}

},

"version": "0"

}

},

"appPolicies": {

"Endorsement": {

"mod\_policy": "Admins",

"policy": {

"type": 3,

"value": {

"rule": "MAJORITY",

"sub\_policy": "Endorsement"

}

},

"version": "0"

},

"LifecycleEndorsement": {

"mod\_policy": "Admins",

"policy": {

"type": 3,

"value": {

"rule": "MAJORITY",

"sub\_policy": "Endorsement"

}

},

"version": "0"

}

},

"acls": {

"\_lifecycle/CheckCommitReadiness": {

"policy\_ref": "/Channel/Application/Writers"

},

"\_lifecycle/CommitChaincodeDefinition": {

"policy\_ref": "/Channel/Application/Writers"

},

"\_lifecycle/QueryChaincodeDefinition": {

"policy\_ref": "/Channel/Application/Readers"

},

"\_lifecycle/QueryChaincodeDefinitions": {

"policy\_ref": "/Channel/Application/Readers"

}

}

}

**Note: the “role” field of these new policies should say 'PEER' if [NodeOUs](https://hyperledger-fabric.readthedocs.io/en/latest/msp.html" \l "organizational-units) are enabled for the org, and 'MEMBER' if they are not.**

## Edit the channel configurations

### System channel updates

Because configuration changes to the system channel to enable the new lifecycle only involve parameters inside the configuration of the peer organizations within the channel configuration, each peer organization being edited will have to sign the relevant channel configuration update.

However, by default, the system channel can only be edited by system channel admins (typically these are admins of the ordering service organizations and not peer organizations), which means that the configuration updates to the peer organizations in the consortium will have to be proposed by a system channel admin and sent to the relevant peer organization to be signed.

You will need to export the following variables:

* CH\_NAME: the name of the system channel being updated.
* CORE\_PEER\_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the ordering service organizations.
* CORE\_PEER\_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
* TLS\_ROOT\_CA: the absolute path to the root CA certificate of the organization proposing the system channel update.
* ORDERER\_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.
* ORGNAME: the name of the organization you are currently updating.
* CONSORTIUM\_NAME: the name of the consortium being updated.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config).

Once you have a modified\_config.json, add the lifecycle organization policy (as listed in enable\_lifecycle.json) using this command:

jq **-**s ".[0] \* {\"channel\_group\":{\"groups\":{\"Consortiums\":{\"groups\": {\"$CONSORTIUM\_NAME\": {\"groups\": {\"$ORGNAME\": {\"policies\": .[1].${ORGNAME}Policies}}}}}}}}" config**.**json **./**enable\_lifecycle**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

As stated above, these changes will have to be proposed by a system channel admin and sent to the relevant peer organization for signature.

### Application channel updates

#### Edit the peer organizations

We need to perform a similar set of edits to all of the organizations on all application channels.

Note that unlike the system channel, peer organizations are able to make configuration update requests to application channels. If you are making a configuration change to your own organization, you will be able to make these changes without needing the signature of other organizations. However, if you are attempting to make a change to a different organization, that organization will have to approve the change.

You will need to export the following variables:

* CH\_NAME: the name of the application channel being updated.
* ORGNAME: The name of the organization you are currently updating.
* TLS\_ROOT\_CA: the absolute path to the TLS cert of your ordering node.
* CORE\_PEER\_MSPCONFIGPATH: the absolute path to the MSP representing your organization.
* CORE\_PEER\_LOCALMSPID: the MSP ID of the organization proposing the channel update. This will be the MSP of one of the peer organizations.
* ORDERER\_CONTAINER: the name of an ordering node container. When targeting the ordering service, you can target any particular node in the ordering service. Your requests will be forwarded to the leader automatically.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config).

Once you have a modified\_config.json, add the lifecycle organization policy (as listed in enable\_lifecycle.json) using this command:

jq **-**s ".[0] \* {\"channel\_group\":{\"groups\":{\"Application\": {\"groups\": {\"$ORGNAME\": {\"policies\": .[1].${ORGNAME}Policies}}}}}}" config**.**json **./**enable\_lifecycle**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

#### Edit the application channels

After all of the application channels have been [updated to include V2\_0 capabilities](https://hyperledger-fabric.readthedocs.io/en/latest/upgrade_to_newest_version.html#capabilities), endorsement policies for the new chaincode lifecycle must be added to each channel.

You can set the same environment you set when updating the peer organizations. Note that in this case you will not be updating the configuration of an org in the configuration, so the ORGNAME variable will not be used.

Once you have set the environment variables, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config).

Once you have a modified\_config.json, add the channel endorsement policy (as listed in enable\_lifecycle.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Application": {"policies": .[1].appPolicies}}}}' config**.**json **./**enable\_lifecycle**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

For this channel update to be approved, the policy for modifying the Channel/Application section of the configuration must be satisfied. By default, this is a MAJORITY of the peer organizations on the channel.

#### Edit channel ACLs (optional)

The following [Access Control List (ACL)](https://hyperledger-fabric.readthedocs.io/en/latest/access_control.html) in enable\_lifecycle.json are the default values for the new lifecycle, though you have the option to change them depending on your use case.

"acls": {

"\_lifecycle/CheckCommitReadiness": {

"policy\_ref": "/Channel/Application/Writers"

},

"\_lifecycle/CommitChaincodeDefinition": {

"policy\_ref": "/Channel/Application/Writers"

},

"\_lifecycle/QueryChaincodeDefinition": {

"policy\_ref": "/Channel/Application/Readers"

},

"\_lifecycle/QueryChaincodeDefinitions": {

"policy\_ref": "/Channel/Application/Readers"

You can leave the same environment in place as when you previously edited application channels.

Once you have the environment variables set, navigate to [Step 1: Pull and translate the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-1-pull-and-translate-the-config).

Once you have a modified\_config.json, add the ACLs (as listed in enable\_lifecycle.json) using this command:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Application": {"values": {"ACLs": {"value": {"acls": .[1].acls}}}}}}}' config**.**json **./**scripts**/**policies**.**json **>** modified\_config**.**json

Then, follow the steps at [Step 3: Re-encode and submit the config](https://hyperledger-fabric.readthedocs.io/en/latest/config_update.html#step-3-re-encode-and-submit-the-config).

For this channel update to be approved, the policy for modifying the Channel/Application section of the configuration must be satisfied. By default, this is a MAJORITY of the peer organizations on the channel.

## Enable new lifecycle in core.yaml

If you follow [the recommended process](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html#overview) for using a tool like diff to compare the new version of core.yaml packaged with the binaries with your old one, you will not need to whitelist \_lifecycle: enable because the new core.yaml has added it under chaincode/system.

However, if you are updating your old node YAML file directly, you will have to add \_lifecycle: enable to the system chaincodes whitelist.

For more information about upgrading nodes, check out [Upgrading your components](https://hyperledger-fabric.readthedocs.io/en/latest/upgrading_your_components.html).