# Genius : Sharding evolution

## Contents

## Rationale

When we started working on the new transaction manager, we came across a number of instances in Genius (and perhaps NetVirt?) where a single operation involves multiple transactions run in parallel, with the aim of separating in particular inventory and topology operations. This makes it complicated to manage the transactions, and makes rollbacks in particular much harder to get right. However, these coding inconveniences might have valid production and/or performance reasons...

## Current situation

The default upstream clustering setup uses at least six shards: default, inventory, and topology, each split into configuration and operational shards. It turns out that this setup is rooted in design considerations: operations involving inventory shouldn’t be combined with operations involving topology, so they are managed as separate shards to encourage developers to keep them separate. For *production*, the only requirement is to keep configuration and operational separate (for one thing, because they persist differently).

## Short-term analysis

In the short-term, we want to verify that there are no downsides in production to moving from the default upstream model to a simpler, default-only shard model.

## Medium-term changes

We want to avoid having single methods requiring multiple parallel transactions. Transactions should be manageable separately, using either chains or the job coordinator, with proper rollback jobs where necessary. This means either collapsing operations into single transactions, as long as that doesn’t stop us ultimately from using more shards by changing the configuration; or splitting operations up and sequencing them explicitly, with separate, sequential transactions (possibly mediated by separate events). It might also be useful to allow the transaction management layer to decide what transaction to use depending on where the changes are being made. The ultimate goal is to decorrelate the code from the shard configuration: we should get good performance from the code without having to tie it to specific shard configurations.

## Long-term changes

For performance when we run out of vertical scalability, we want to be able to use shards for horizontal scalability too. This means being able to activate some form of key-based sharding. The current design of the datastore doesn’t support this, but it doesn’t prevent it either; with a modicum of effort it should be possible to use key-based shards, including split YANG lists.