10 WAYS TO RUIN YOUR PROXMOX SETUP

And How Not To

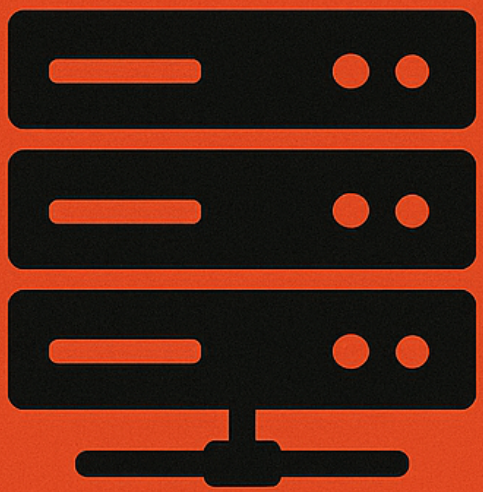

Table of Contents

Proxmox VE has become one of the most popular virtualization platforms for good reason. It's powerful, flexible, open source, and capable of running everything from a small homelab to serious production infrastructure. The web interface is intuitive, the documentation is solid, and the community is active and helpful.

And yet, Proxmox environments fail. VMs become sluggish for no apparent reason. Clusters split-brain during maintenance windows. Backups turn out to be useless when you actually need them. Storage pools degrade silently until a second disk failure turns inconvenience into catastrophe.

These failures rarely stem from bugs in Proxmox itself. Instead, they grow from decisions made during setup and configuration - decisions that seemed reasonable at the time but were based on incomplete understanding, outdated advice, or assumptions that didn't match reality. The gap between "it works" and "it works reliably" is filled with hard-won knowledge that most of us acquire the painful way: by experiencing failures firsthand.

This guide exists to help you skip some of that pain.

Who This Guide Is For

Whether you're running a homelab in your basement, managing infrastructure for a small business, or consulting for clients who need reliable virtualization, this guide addresses the mistakes that matter in your world. We're not talking about exotic edge cases or theoretical concerns - we're talking about the issues that actually break systems and ruin weekends.

The assumed baseline is basic familiarity with Proxmox and Linux administration. You don't need to be an expert, but you should be comfortable with the command line and understand what terms like VM, container, and storage pool mean. If you've set up Proxmox at least once and gotten it running, you're ready for this material.

What You'll Learn

This guide covers ten common mistakes, each explored in enough depth to understand not just what to do, but why it matters:

Storage fundamentals - why ZFS needs more RAM than you think, why the ECC debate is mostly noise, and why RAIDZ might be the wrong choice for your VM workloads.

Configuration decisions - understanding when Local-LVM makes sense again, why the "host" CPU type can sabotage your cluster, and what High Availability actually requires to function.

Operational practices - building a backup strategy that actually protects you, keeping Docker off your hypervisor, implementing monitoring that catches problems early, and maintaining the discipline to keep your Proxmox host clean.

Each chapter stands on its own, so feel free to jump to the topics most relevant to your situation. That said, reading through sequentially will give you a coherent picture of what a well-designed Proxmox environment looks like and why each piece matters.

A Note on Advice

Technology advice ages poorly. What was best practice five years ago might be outdated today, and what's current now will eventually be superseded. Proxmox itself evolves - features that were problematic in version 7 might work flawlessly in version 9.

This guide reflects the state of Proxmox VE as of 2025, with particular attention to changes in recent versions. Where historical context matters - like the improvements to Local-LVM snapshots in PVE 9 - we'll note it explicitly. But always verify that advice still applies to your specific version before implementing changes in production.

More importantly, understand the reasoning behind each recommendation. Technology changes, but the underlying principles - defense in depth, separation of concerns, knowing your failure modes - remain constant. If you understand why a practice is recommended, you'll be able to adapt when circumstances differ from what any guide anticipates.

Disclaimer

The material in this book is guidance, not a guarantee. Every environment is different; you are responsible for validating changes in your own lab before production and for ensuring compliance with your organization's policies and applicable laws.

No warranties. All advice is provided "as is" without warranty of any kind. Apply it at your own risk.

Test first. Reproduce steps in a non-production lab, make and verify backups, and have rollback plans before touching live systems.

Security and compliance. Verify that any suggested configuration meets your security baselines, logging/monitoring requirements, and regulatory obligations (e.g., data protection, industry standards).

Vendor documentation prevails. Always cross-check against current Proxmox, distro, firmware, and hardware vendor documentation; if they conflict, follow the vendor.

Get professional help when needed. For business-critical workloads, engage qualified professionals and follow formal change-management.

Let's Begin

Proxmox is capable of remarkable reliability when configured thoughtfully. The difference between an environment that runs smoothly for years and one that generates constant firefighting often comes down to decisions made during initial setup - decisions you might not realize were wrong until much later.

The goal of this guide is to help you make better decisions from the start, or to recognize and correct problematic patterns in existing environments before they cause serious trouble.

Let's look at the first mistake: underestimating what ZFS really needs to run well.

Chapter 1. ZFS Without Proper RAM Planning

ZFS is one of those technologies that makes you feel like a proper sysadmin. Checksums that catch silent corruption, snapshots that cost almost nothing, built-in compression, self-healing capabilities - it's the filesystem equivalent of a Swiss Army knife with a PhD. Once you've experienced the peace of mind that comes from knowing your data is actively protected against bit rot, it's hard to go back to traditional filesystems.

But here's what the enthusiastic forum posts and YouTube tutorials often gloss over: ZFS is hungry. It wants RAM, and not just a polite amount. Feed it properly and it will reward you with excellent performance and rock-solid reliability. Starve it, and you'll experience mysterious slowdowns, VM stuttering, and the creeping suspicion that something is wrong without any clear indication of what.

1.1. The Formula That Fails

If you've researched ZFS at all, you've encountered the rule of thumb: "ZFS needs 1 GB of RAM per terabyte of storage." It's simple, memorable, and quoted with confidence across countless forum threads and blog posts. It's also dangerously misleading for modern Proxmox environments.

This formula originated in an era of smaller storage pools and simpler workloads. A 4 TB pool on a system with 8 GB of RAM seemed reasonable, and often was. But storage has grown dramatically cheaper while our expectations for performance have grown higher. Today's homelab might casually deploy 20, 40, or even 100 TB of storage - and suddenly that simple formula produces numbers that don't account for everything else competing for memory.

I learned this lesson firsthand when building what I thought was a well-planned storage server. Twenty-four terabytes of raw storage, 16 GB of RAM - the math worked out according to the formula. What I didn't account for was that Proxmox itself needs memory to operate, VMs need memory to run, and ZFS's Adaptive Replacement Cache was going to fight for every byte it could get.

The result was a system that worked but never felt right. VMs would occasionally stutter. Storage performance varied wildly depending on what else was happening. The ARC was constantly being squeezed by memory pressure, which meant ZFS couldn't cache effectively, which meant more disk I/O, which meant worse performance across the board. Everything technically functioned, but nothing functioned well.

1.2. Understanding ZFS Memory Architecture

To plan properly, you need to understand where ZFS memory actually goes. The operating system doesn't just hand ZFS a chunk of RAM and walk away - there are several distinct components, each with their own requirements.

The base overhead for ZFS itself runs around 2 GiB regardless of pool size. This covers the kernel modules, metadata structures, and basic operational requirements. You pay this cost whether you have 1 TB or 100 TB of storage.

Beyond that baseline, ZFS needs roughly 1 GiB of RAM for every TiB of usable storage. This memory holds metadata about your files and datasets - the information ZFS needs to locate and verify your data. Larger pools mean more metadata, which means more memory required just to operate.

The ARC is where things get interesting. This is ZFS's read cache, and it's remarkably effective at accelerating access to frequently-used data. By default, ZFS will try to use as much RAM as it can get for the ARC, gracefully releasing memory when other applications need it. In theory, this is elegant - ZFS uses spare memory productively without starving other processes. In practice, the "graceful release" can lag behind actual demand, causing memory pressure that affects VM performance.

If you're considering L2ARC - a second-level cache on fast storage like SSDs - be aware that it requires RAM too. The metadata for L2ARC entries lives in main memory, consuming roughly 1 GiB of RAM for every 50 GiB of L2ARC space. A 500 GB L2ARC device costs you 10 GiB of RAM just to track what's in it.

And then there's deduplication, which is where memory requirements can explode entirely. Deduplication requires ZFS to maintain a table of every unique block in your pool so it can identify duplicates. This Deduplication Table needs approximately 5 GiB of RAM per TiB of stored data. A modest 10 TB pool with deduplication enabled demands 50 GB of RAM just for the DDT - before accounting for anything else.

1.3. Practical Planning

So what does this mean for your actual hardware decisions? Let's work through realistic scenarios.

For a small homelab pool - say 8 TiB usable - you're looking at 2 GiB base overhead plus 8 GiB for metadata. That's 10 GiB just for ZFS to operate comfortably, before your VMs get a single byte. On a system with 16 GB total, you have perhaps 6 GB remaining for Proxmox overhead and all your virtualized workloads. It'll work, but you'll be constantly bumping against limits.

Scale that up to a 24 TiB pool and the math gets uncomfortable quickly. Base overhead plus metadata puts you at 26 GiB for ZFS alone. Now you need 32 GB of RAM minimum, and 64 GB is far more comfortable if you want your VMs to have breathing room.

The practical floor for serious ZFS usage is 16 GB of system RAM, and that only works for smaller pools with modest VM workloads. For pools in the 10-20 TiB range, 32 GB should be your starting point. Larger pools or heavier workloads push you toward 64 GB or beyond.

As for deduplication - just don't. Unless you have a very specific use case where you know you'll see massive duplication ratios and you have the memory budget to support it, deduplication causes more problems than it solves. The memory overhead is punishing, write performance suffers significantly, and for most workloads the space savings don't justify the costs. Use compression instead - LZ4 or ZSTD provide meaningful space savings with negligible performance impact and no additional memory requirements.

1.4. Monitoring What Matters

The good news is that ZFS is transparent about its memory usage. You don't have to guess whether your system is healthy - you can see exactly what's happening with the ARC and respond

accordingly.

The `arc_summary` command provides a comprehensive overview of ARC statistics, including current size, hit rates, and configuration limits. Run it periodically to understand how your cache is behaving:

```
arc_summary
```

For a quick check of key metrics without the full report, you can query the kernel statistics directly:

```
cat /proc/spl/kstat/zfs/arcstats | grep -E "^size|^c_max|^c_min|^hits|^misses"
```

The numbers you want to watch are `size` (current ARC size in bytes), `c_max` (the maximum the ARC is allowed to grow), and `c_min` (the minimum the ARC will fight to maintain). If your ARC is consistently sitting at or near `c_min` while the system shows memory pressure, you don't have enough RAM for your workload.

The hit rate matters too. A healthy ARC should be serving the vast majority of read requests from cache. If your hit rate is low - say below 80% for a typical workload - either your working set doesn't fit in the available cache or your access patterns are unusually random. Either way, it's worth investigating.

For ongoing visibility, integrate ZFS metrics into whatever monitoring system you're using. Checkmk, Prometheus with appropriate exporters, or even a simple script that logs key values - any of these will help you spot trends before they become problems. A gradually declining hit rate or steadily increasing memory pressure gives you warning that something needs attention.

1.5. Constraining the ARC

In memory-constrained environments, you might need to explicitly limit how much RAM ZFS can claim. Left to its own devices, ZFS will use as much as it can get, which sounds efficient but can cause problems when that memory isn't released quickly enough for other workloads.

Setting a maximum ARC size gives you predictable behavior at the cost of some potential cache performance. The configuration lives in `/etc/modprobe.d/zfs.conf`:

```
# Limit ARC to 8 GiB (value in bytes)
options zfs zfs_arc_max=8589934592
```

After making changes, update the initial ramdisk and reboot for the setting to take effect:

```
update-initramfs -u
reboot
```

Choose your limit based on your total RAM minus what you need for Proxmox, VMs, and a

reasonable buffer. The ARC operates well even when constrained - you'll see more disk reads than with an unlimited cache, but the system behavior becomes much more predictable.

1.6. The Bottom Line

ZFS is absolutely worth using. The data integrity features alone make it a compelling choice for any storage where the data matters. But it's not a technology you can deploy casually based on oversimplified formulas and hope for the best.

Understand what ZFS actually requires. Plan your memory budget honestly, accounting for the base system, your virtualization workload, and ZFS with appropriate headroom. Monitor the ARC and respond when the numbers suggest problems. And resist the temptation of deduplication unless you truly understand the costs.

Your future self, enjoying a Proxmox environment that performs consistently without mysterious slowdowns, will appreciate the planning you put in today.

Chapter 2. ECC RAM is Mandatory for ZFS

Few topics in the storage community generate as much heated debate - and as much confident misinformation - as the question of ECC memory and ZFS. Spend any time in forums discussing ZFS deployment, and you'll encounter emphatic warnings that running ZFS without ECC RAM is reckless, dangerous, or will inevitably lead to catastrophic data loss.

The intensity of these warnings can be intimidating, especially for homelab builders working within a budget. ECC memory requires compatible motherboards and CPUs, which often means more expensive server-grade hardware. The prospect of needing to scrap your existing system because random forum commenters insist it's unsuitable for ZFS is enough to make anyone second-guess their plans.

Here's the thing: the conventional wisdom is wrong. Not entirely wrong - ECC memory is genuinely beneficial - but wrong in the specific claim that ZFS has some unique dependency on ECC that other filesystems don't share.

2.1. Where the Myth Comes From

The argument typically goes something like this: ZFS caches heavily in RAM, both for reads through the ARC and for writes that haven't yet been flushed to disk. If a bit flip occurs in RAM - due to cosmic rays, electrical noise, or failing memory - that corrupted data will be written to disk. ZFS will then happily calculate a checksum of the corrupted data, and from that point forward, ZFS will consider the corruption to be valid data. The self-healing features that make ZFS special become complicit in preserving the damage.

On the surface, this sounds alarming. And the underlying premise is technically accurate - a bit flip in RAM can indeed cause corrupted data to be written to disk and subsequently checksummed as valid. But there's a critical flaw in treating this as a ZFS-specific problem.

Every filesystem writes data from RAM to disk. When ext4 writes a file, that data comes from RAM. When NTFS updates a document, the changes flow through RAM. When XFS commits a transaction, RAM is involved. None of these filesystems validate data integrity in RAM before writing - they can't, because they don't have ZFS's checksumming capability.

If a bit flip corrupts data in RAM before ext4 writes it to disk, ext4 will happily store the corrupted data with no indication that anything went wrong. At least ZFS has a fighting chance of detecting the problem later if another copy of the data exists or if the corruption affects metadata in a detectable way. Traditional filesystems would serve you that corrupted data forever without a hint that anything was amiss.

The argument that ZFS specifically needs ECC is exactly backwards. ZFS is more resilient to storage-level corruption than traditional filesystems, and it's no more vulnerable to RAM corruption than anything else. ECC benefits your entire system equally - ZFS doesn't need it more than your database, your running applications, or your kernel.

2.2. What ECC Actually Provides

To have an informed opinion, it helps to understand what ECC memory actually does and what failure modes it addresses.

Standard RAM stores each bit as a tiny electrical charge. Over time, or due to various environmental factors, those charges can spontaneously change state - a one becomes a zero, or vice versa. These bit flips are rare but not negligible. A large-scale Google study found that over 8% of DIMMs experienced at least one error per year, with error rates orders of magnitude higher than previous laboratory studies suggested. However, rates vary significantly based on hardware quality, DIMM age, and environmental factors - and the study also found that most errors are hard errors in specific faulty DIMMs rather than random cosmic ray events affecting all memory equally.

ECC memory adds redundant bits that allow the memory controller to detect and correct single-bit errors, and detect (without correcting) multi-bit errors. When a single bit flips, ECC catches it and fixes it transparently. When multiple bits flip - a much rarer occurrence - ECC detects that something is wrong and can alert the system, even if it can't fix the problem.

This protection is genuinely valuable. Undetected memory errors can cause data corruption, application crashes, or system instability. In environments where reliability is paramount - production servers, financial systems, medical equipment - ECC is standard and appropriate.

But note what ECC doesn't do: it doesn't specifically protect ZFS more than anything else. Every byte your system processes passes through RAM. ECC protects all of it equally - your filesystem, your applications, your kernel, your network stack. Singling out ZFS as having special ECC requirements misunderstands both ZFS and ECC.

2.3. Risk Assessment in the Real World

So should you use ECC memory? It depends on your situation, your risk tolerance, and your budget.

For production environments handling important data - business systems, customer information, financial records - ECC is standard practice and you should plan for it. The incremental cost of ECC-compatible hardware is minor compared to the value of the data and the cost of potential corruption or downtime. Server-grade hardware typically supports ECC by default, so this may not even be a decision you need to make explicitly.

For homelabs and personal systems, the calculus is different. ECC-compatible platforms often cost significantly more than consumer hardware, and the actual risk of memory errors causing meaningful problems is relatively low. A bit error rate of one per gigabyte per year sounds concerning, but most of those errors affect data that's transient - memory that will be freed and reallocated soon anyway. The chances of a bit flip hitting exactly the wrong byte at exactly the wrong time and causing permanent data corruption are low.

More importantly, there are other risks that are far more likely to cause you data loss. Hardware failure. Accidental deletion. Ransomware. Fire or theft. A power surge that fries your system. If you're choosing between ECC memory and a proper backup system, the backup system protects you against a much wider range of threats.

2.4. What Actually Protects Your Data

If you're serious about protecting your data - and you should be, regardless of your RAM type - focus on the fundamentals that actually matter.

Backups are your single most important protection. They should follow the 3-2-1 rule: three copies of your data, on two different types of media, with one copy offsite. Test your restores regularly to verify that your backups actually work. No amount of ECC RAM will save you from a dead disk, a ransomware infection, or a house fire - but backups will.

Regular ZFS scrubs verify data integrity across your entire pool. A scrub reads every block and compares it against its stored checksum, detecting any corruption regardless of its source. Schedule scrubs monthly at minimum:

```
# Start a scrub
zpool scrub tank

# Check scrub status and results
zpool status tank
```

A scrub that reports checksum errors is telling you something is wrong - potentially a dying disk, a controller issue, or yes, possibly a memory problem. Either way, you're learning about corruption in time to address it, rather than discovering it when you need that data.

Speaking of memory problems, periodic RAM testing is valuable regardless of whether you have ECC. Memtest86+ is the standard tool:

```
apt install memtest86+
```

After installation, you'll find memtest in your boot menu. Running it overnight catches most memory issues before they cause data problems. A RAM stick that's failing will typically show errors in memtest long before it causes noticeable corruption.

For systems with ECC, you can monitor for corrected errors through the EDAC subsystem:

```
apt install edac-utils
edac-util -s
```

Regular corrected errors indicate memory that's marginal - ECC is saving you from immediate problems, but replacement should be on your roadmap.

2.5. The Social Dynamics of Forum Advice

It's worth noting why the "ECC is mandatory" advice persists despite being technically questionable. Online communities often develop strong conventions that take on moral dimensions. Running ZFS without ECC becomes not just a technical choice but a character flaw - evidence of recklessness or ignorance that invites criticism.

This dynamic discourages nuanced discussion. People who run ZFS successfully without ECC often don't mention it to avoid lectures. People who are considering ZFS get warned away from consumer hardware even when it would serve them fine. The conversation becomes about conformity to group standards rather than informed technical assessment.

When evaluating advice - about ZFS, ECC, or anything else - look for explanations of why something matters, not just assertions that it does. Ask for sources. Consider whether the advice accounts for different use cases and risk tolerances. Be skeptical of claims that frame complex technical decisions as obvious and anyone who disagrees as foolish.

2.6. Best Practices

If your hardware supports ECC and the cost is manageable, use it. It's a genuine reliability improvement, and server-grade hardware often includes it by default. For new builds in professional or business contexts, plan for ECC from the start.

For homelabs on consumer hardware, don't let ECC anxiety stop you from using ZFS. The benefits of ZFS - checksumming, snapshots, replication, self-healing - are too valuable to give up because forum posts made you nervous. Plenty of people run ZFS on non-ECC systems without problems, and the actual risks are much smaller than the heated discussions suggest.

Whatever your RAM situation, get the fundamentals right. Maintain tested backups. Run regular scrubs. Monitor your systems for signs of trouble. These practices protect you against the full spectrum of threats, not just the narrow category of memory errors that ECC addresses.

And when someone tells you that your homelab is irresponsible because you're running ZFS without ECC, feel free to ask them how often they test their backup restores. That question tends to be more revealing than their RAM configuration.

2.7. The Bottom Line

ECC memory is a good thing that makes systems more reliable. It is not a ZFS-specific requirement, and running ZFS on non-ECC hardware is not the reckless act that forum orthodoxy claims. Make your hardware decisions based on your actual requirements, budget, and risk tolerance - not on oversimplified rules and social pressure from people who don't know your situation.

Focus on what actually protects your data: backups, monitoring, regular integrity checks, and understanding your failure modes. Get those right, and the question of ECC becomes far less important.

Chapter 3. Using RAIDZ for VM Storage

RAIDZ is one of ZFS's most appealing features for anyone planning storage infrastructure. The promise is compelling: combine multiple disks into a pool that survives disk failures while maximizing usable capacity. Unlike traditional RAID, ZFS handles everything in software with no special controller required. You get redundancy, you get capacity efficiency, and you get all the other ZFS benefits like checksumming and snapshots.

On paper, RAIDZ looks like the obvious choice for a Proxmox storage pool. Why waste half your disks on mirrors when RAIDZ1 loses only one disk to parity, or RAIDZ2 just two? The storage efficiency calculators make it look like free space you'd be foolish to leave on the table.

And then you actually try running VMs on it.

3.1. The Day I Learned This Lesson

When I built my first "serious" Proxmox server, I was proud of my storage design. Six disks in RAIDZ2, giving me four disks worth of usable space while surviving any two simultaneous disk failures. The redundancy math was solid. The capacity was generous. The ZFS pool creation went smoothly and looked beautiful in the web interface.

Then I started actually using it for VMs. The performance wasn't terrible, exactly - VMs booted, applications ran, files saved. But everything felt slightly sluggish in a way I couldn't quite pin down. Database operations that should have been instant had noticeable latency. Multiple VMs doing disk-intensive work simultaneously brought the system to its knees. The numbers on my capacity calculator hadn't warned me about any of this.

It took embarrassingly long to understand what was happening, because nothing was actually broken. The system was working exactly as designed. The problem was that RAIDZ is designed for a different kind of workload than VMs generate.

3.2. Why RAIDZ and VMs Don't Mix

The root issue comes down to how RAID parity works and what kind of I/O patterns virtual machines create.

RAIDZ distributes data and parity across all disks in the vdev. When you write data, ZFS must calculate parity information and write both the data and parity to multiple disks. When you modify existing data, ZFS must read the old data, read the old parity, calculate new parity based on the changes, then write the new data and new parity. This is the classic RAID write penalty, and for small random writes, it's significant.

Virtual machines are small random write machines. The guest operating system has its own filesystem, its own caching, its own access patterns, and none of them know or care that they're running on top of ZFS. A VM doing ordinary work - booting up, running applications, saving files - generates a stream of small, random I/O operations scattered across its virtual disk. This is exactly the workload that RAIDZ handles worst.

The counterintuitive result is that a RAIDZ vdev with six disks delivers random IOPS roughly

equivalent to a single disk. Your beautiful six-disk array, when handling the random I/O that VMs generate, performs like one disk. The other five disks are busy doing parity calculations and coordinated writes, but they're not multiplying your performance - they're just maintaining redundancy.

For sequential workloads, RAIDZ performs much better. When you're writing large files continuously - streaming video, backup archives, sequential data transfers - the writes are large enough that the parity overhead is amortized across substantial data. This is why RAIDZ makes sense for backup storage, media libraries, and bulk file servers. But VM workloads are the opposite of sequential.

3.3. The Mirror Alternative

Mirrored vdevs work completely differently, and the difference matters enormously for VM performance.

In a mirror, each disk in the pair contains a complete copy of the data. Writes go to both disks, but there's no parity calculation - just straightforward duplication. More importantly, reads can come from either disk, and ZFS is smart about distributing read requests across mirror members.

If you have three mirror pairs (six disks total, like my RAIDZ2), reads scale with the number of disks. Each mirror can serve a read independently, and within each mirror, either disk can respond. For random read workloads, you're getting close to six disks worth of IOPS instead of one.

Writes scale with the number of mirror pairs. Each pair operates independently, so three pairs can handle roughly three times the random write IOPS of a single disk. That's still not six times - mirrors write to both members - but it's dramatically better than RAIDZ's effective single-disk performance.

The cost is capacity. Mirrors use 50% of your raw storage for redundancy, compared to RAIDZ1's one-disk overhead or RAIDZ2's two-disk overhead. Those capacity calculators that made RAIDZ look so attractive aren't lying - you genuinely get more usable space. The question is whether that space is worth the performance trade-off for your specific workload.

3.4. Seeing the Difference

If you're skeptical - and healthy skepticism about storage advice is appropriate - you can test this yourself. The `fio` tool generates I/O patterns that let you benchmark actual storage performance:

```
apt install fio

# Random 4K read test - typical of VM read patterns
fio --name=random-read \
    --ioengine=libaio \
    --direct=1 \
    --bs=4k \
    --iodepth=32 \
    --rw=randread \
    --size=1G \
    --runtime=60 \
    --filename=/path/to/your/pool/testfile

# Random 4K write test - typical of VM write patterns
fio --name=random-write \
    --ioengine=libaio \
    --direct=1 \
    --bs=4k \
    --iodepth=32 \
    --rw=randwrite \
    --size=1G \
    --runtime=60 \
    --filename=/path/to/your/pool/testfile
```

Run these tests on a RAIDZ pool and then on a mirror pool of similar total capacity. The random I/O numbers will tell a story that's hard to argue with. Sequential tests will show RAIDZ in a much better light, which is precisely the point - RAIDZ excels at sequential workloads and struggles with random ones.

You can also observe this in production through ZFS's built-in monitoring:

```
# Watch I/O statistics in real-time
zpool iostat -v 5
```

On a RAIDZ pool under VM load, you'll typically see all disks active but throughput limited by the parity overhead. On mirrors, you'll see the I/O distributed more effectively across the vdevs.

3.5. The RAIDZ1 Rebuild Problem

Beyond performance, there's another reason to be cautious with RAIDZ, particularly RAIDZ1: the resilver risk with modern disk sizes.

When a disk fails in a RAIDZ pool, ZFS must rebuild the replacement disk by reading from all surviving disks and recalculating the missing data. This process - called resilvering - reads every byte from every remaining disk. With today's multi-terabyte drives, resilvering can take days.

During the entire resilver, you're running without your expected redundancy. A RAIDZ1 pool resilvering after a single disk failure has zero fault tolerance until the rebuild completes. If another

disk fails during those two or three days - or even just develops a few unreadable sectors that went unnoticed - you lose the pool.

The math on this has gotten uncomfortable as disk sizes have grown. A 16 TB disk takes far longer to resilver than a 1 TB disk did, but the expected lifetime and failure rates haven't improved proportionally. The window of vulnerability has stretched from hours to days, and the probability of a second failure during that window has increased accordingly.

This is why experienced ZFS administrators often consider RAIDZ1 obsolete for large disks. RAIDZ2 survives two failures, giving you meaningful protection during resilver operations. RAIDZ3 takes this further but costs three disks worth of capacity. But even RAIDZ2 and RAIDZ3 don't address the performance issue - they're still fundamentally limited by parity overhead for random I/O.

Mirrors, by contrast, resilver quickly because they only need to copy data from the surviving disk to the replacement. A mirror resilver is essentially a single-disk read and single-disk write operation, completing in a fraction of the time a RAIDZ resilver takes.

3.6. Designing Storage for Mixed Workloads

The solution isn't to avoid RAIDZ entirely - it's to use the right storage topology for each workload. Most Proxmox environments have multiple distinct storage needs, and there's no reason they all need to use the same pool design.

VM storage needs IOPS. The random I/O patterns of virtual machines demand storage that can handle many small operations per second. Mirrors deliver this, so use mirrors for your VM disks.

Backup storage needs capacity and sequential throughput. Proxmox Backup Server writes large backup chunks sequentially, exactly the pattern RAIDZ handles well. Put your PBS datastore on RAIDZ2 and enjoy the capacity efficiency.

Media and archive storage falls into the same category. Large files, sequential access patterns, prioritizing space efficiency over random I/O - RAIDZ is perfectly suited.

A practical implementation might look like this:

```
# High-performance pool for VM disks (mirrors)
zpool create vm-storage \
    mirror /dev/sda /dev/sdb \
    mirror /dev/sdc /dev/sdd

# Capacity-oriented pool for backups (RAIDZ2)
zpool create backup-storage \
    raidz2 /dev/sde /dev/sdf /dev/sdg /dev/sdh /dev/sdi /dev/sdj
```

This separation gives you fast storage where performance matters and efficient storage where capacity matters. Your VMs get the IOPS they need; your backups get the space they need.

3.7. Best Practices

Match your storage topology to your workload's actual I/O patterns. VM storage should use mirrors for acceptable random I/O performance. Backup and archive storage can use RAIDZ for capacity efficiency.

Avoid RAIDZ1 with modern disk sizes. The resilver window is too long and the risk of a second failure during rebuild is too high. If you're using RAIDZ at all, RAIDZ2 should be your minimum.

Keep VM storage and archive storage separate. Mixing workloads with different I/O characteristics on the same pool forces compromises that serve neither workload well. Separate pools let you optimize each for its purpose.

Consider special vdevs for write-heavy workloads. A small SSD configured as a ZFS Intent Log (SLOG) can improve synchronous write performance on RAIDZ pools. This doesn't fix the fundamental random I/O limitation, but it helps with specific workloads. However, adding complexity to work around a topology mismatch is usually a sign you should reconsider the topology.

3.8. The Bottom Line

RAIDZ is excellent technology used in countless production environments worldwide - for the right workloads. Sequential I/O, large files, backup storage, media archives - these are where RAIDZ shines. The capacity efficiency is real, and the redundancy is solid.

But virtual machine storage isn't that workload. VMs generate random I/O that RAIDZ handles poorly, and no amount of adding disks to a RAIDZ vdev will fix that fundamental mismatch. If your VMs feel sluggish on RAIDZ storage, they're not broken - they're experiencing exactly what the architecture delivers.

Match your storage to your workload. Your VMs want IOPS, and mirrors deliver IOPS. Give them what they need.

Chapter 4. Dismissing Local-LVM as "Inflexible"

If you've been around the Proxmox community for a while, you've probably absorbed the conventional wisdom: Local-LVM is the boring default that serious admins replace with ZFS as soon as possible. Live migration is a pain, and honestly, it just feels a bit… legacy.

For years, parts of this assessment were correct. But if you're still operating on that assumption in 2025, you might be making your infrastructure more complicated than it needs to be.

4.1. Understanding What "Local-LVM" Actually Is

Before we go further, let's clear up a common point of confusion. When people talk about "Local-LVM" in Proxmox, they're usually referring to the default `local-lvm` storage that the installer creates. This is actually **LVM-thin** - a thin-provisioned storage pool that has supported snapshots natively for years.

The distinction matters because Proxmox VE 9 introduced a new snapshot mechanism for a different type of storage: **thick-provisioned LVM**. This is the kind of LVM you'd typically use with iSCSI LUNs or Fibre Channel storage from a SAN. If you're running a standard single-node Proxmox installation with the default storage layout, the new feature doesn't change anything for you - your LVM-thin storage already handles snapshots just fine.

4.2. The Old Reputation

So where did Local-LVM's bad reputation come from? A few places:

The most legitimate complaint was always about migration. Moving a VM from one node to another with local storage meant either shutting down the VM, copying the entire disk image, and starting it on the new node - or setting up shared storage just to avoid the hassle. For clusters where you needed flexibility, any local storage seemed like a dead end.

There was also confusion about traditional LVM snapshots - the old-style snapshots that used a copy-on-write mechanism with significant performance penalties. But LVM-thin snapshots work differently and don't suffer from the same issues. Many people conflated the two, assuming that "LVM snapshots are problematic" applied to their Proxmox setup when it didn't.

So the community advice became: use ZFS for anything serious, or set up Ceph if you're building a proper cluster. Local storage was relegated to "only if you really have to" status.

4.3. What Changed in Proxmox VE 9

With the release of Proxmox VE 9, Proxmox introduced **Snapshots as Volume Chains** - a technology preview that brings snapshot support to thick-provisioned LVM storage. This is significant for enterprise environments using traditional SANs with iSCSI or Fibre Channel, where thick LVM on shared LUNs is common.

Instead of relying on LVM's problematic native snapshot mechanism, Proxmox now creates qcow2-

based volume chains on top of thick LVM volumes. This approach avoids the I/O penalties and dangerous space-exhaustion behavior of traditional LVM snapshots.

For environments using thick LVM on shared storage, this is genuinely useful - you can now snapshot VMs on your SAN without depending on the storage array's native snapshot features.

However, this feature comes with important caveats:

- It's currently a **technology preview**, not fully production-hardened
- Virtual disks must use **qcow2 format**, not raw - existing raw disks need to be converted
- Each snapshot allocates a **full-sized thick volume** on the underlying storage
- Only **linear snapshot chains** are supported - you can only roll back to the most recent snapshot, not to arbitrary points in history
- Some configurations don't work yet, notably VMs with TPM devices
- The qcow2 layer adds **performance overhead** - benchmarks suggest 30-50% impact in some workloads, though this varies
- The feature must be explicitly enabled with `snapshot-as-volume-chain 1` in your storage configuration

For the default LVM-thin storage that most single-node Proxmox installations use, none of this is relevant - LVM-thin already handles snapshots natively and efficiently, without these limitations.

4.4. Live Migration: Still a Consideration

Let's be honest about what hasn't changed: live migration with local storage is still not as smooth as with shared storage. When you migrate a VM between nodes with local storage, Proxmox needs to copy the disk data over the network while the VM continues running. This works, and it works reliably, but it's inherently slower than the near-instantaneous migrations you get when both nodes access the same shared storage backend.

For a homelab or small business environment where you might migrate VMs a few times a year during maintenance windows, this is perfectly acceptable. For a busy production cluster where you're constantly rebalancing workloads or need sub-second failover, you'll still want shared storage or ZFS replication.

The key insight is that **not every environment needs the same capabilities**. The question isn't whether shared storage is better for migrations - of course it is - but whether the added complexity is justified for your specific use case.

4.5. The Simplicity Advantage

There's something to be said for keeping things simple, and LVM-thin excels at simplicity. There's no separate storage system to understand, no ARC tuning to worry about, no memory planning beyond what your VMs need. It uses the storage stack that Linux has relied on for decades, which means troubleshooting is straightforward and documentation is abundant.

For single-node setups especially, this simplicity translates directly into reliability. Fewer moving

parts means fewer things that can break, and when something does go wrong, the debugging process is more straightforward. You're not wondering whether the problem is in ZFS, the ARC, the L2ARC, the SLOG, or somewhere else entirely - you're working with a storage stack that behaves predictably and has well-understood failure modes.

This isn't to say that ZFS's additional features aren't valuable - they absolutely are, in the right context. But features you don't need aren't free; they come with cognitive overhead, additional configuration, and more potential failure points.

4.6. Making the Choice

The decision between LVM-based storage and ZFS (or other backends) should be based on your actual requirements, not on what feels more "professional" or what the forums recommend by default.

LVM-thin (the default local-lvm) makes sense when:

You're running a single-node setup where migration isn't a concern, or you're building a small cluster where occasional slower migrations during maintenance windows are acceptable. It's also a solid choice when you want to keep your Proxmox host as simple as possible, perhaps because you're already running a separate NAS for bulk storage and don't need ZFS's data management features on the hypervisor itself. You get native snapshot support without any special configuration.

Thick LVM on shared storage makes sense when:

You have an existing SAN infrastructure with iSCSI or Fibre Channel and want to leverage it for Proxmox. With Proxmox VE 9's new snapshot feature, you can now get basic snapshot functionality on these environments - just be aware of the technology preview status and limitations.

ZFS (or Ceph) makes sense when:

You need frequent live migrations with minimal disruption, your data integrity requirements justify the additional complexity, or you want built-in features like compression, checksumming, native replication, or the ability to easily expand storage with additional vdevs. Clusters with three or more nodes that require true high availability will generally benefit from shared storage regardless of the underlying technology.

4.7. Checking Your Current Setup

If you're not sure what storage backend you're currently using, Proxmox makes it easy to check:

```
# List all configured storage
pvesm status

# Show detailed storage configuration
cat /etc/pve/storage.cfg
```

Look for `lvmthin:` entries (that's LVM-thin with native snapshot support) versus `lvm:` entries (that's thick LVM, which needs the new volume-chain feature for snapshots).

And if you want to see how your VMs' disks are actually allocated:

```
# List logical volumes (for LVM-based storage)
lvs

# Show detailed info about a specific volume
lvdisplay /dev/pve/vm-100-disk-0

# Check if you're using thin pools
lvs -o+pool_lv
```

4.8. Enabling Snapshots on Thick LVM (If Needed)

If you're using thick LVM storage (typically on shared SAN LUNs) and want to try the new snapshot feature, you'll need to:

1. Add `snapshot-as-volume-chain 1` to your storage configuration in `/etc/pve/storage.cfg`
2. Create new VMs with qcow2 format disks, or convert existing raw disks
3. Be aware that this is a technology preview with the limitations mentioned earlier

```
# Example storage.cfg entry for thick LVM with snapshots
lvm: san-storage
    vgname my-san-vg
    content images,rootdir
    shared 1
    snapshot-as-volume-chain 1
```

4.9. Best Practices

Rather than following blanket recommendations, take the time to evaluate what your environment actually needs:

For single-node setups: The default LVM-thin storage is genuinely capable. You get snapshots, thin provisioning, and straightforward management. Don't add ZFS complexity unless you specifically need its features like checksumming, compression, or native replication.

For small clusters: Think carefully about your migration patterns. How often do you actually need to move VMs between nodes? Can those migrations happen during maintenance windows, or do they need to be seamless and immediate? For environments where frequent, fast migrations are essential, ZFS replication or shared storage like Ceph remains the better choice - but don't add that complexity if you don't need it.

For enterprise SAN environments: The new thick LVM snapshot feature in Proxmox VE 9 is worth evaluating, but treat it as what it is - a technology preview. Test thoroughly before relying on it in production, and keep an eye on the Proxmox release notes for improvements and bug fixes.

Whatever you choose, make the decision consciously rather than defaulting to whatever seems

most sophisticated. The best storage architecture is the one that meets your requirements with the least unnecessary complexity.

4.10. The Bottom Line

The reputation of LVM-based storage in Proxmox deserves reconsideration. The default LVM-thin storage has always been more capable than many give it credit for - it's had working snapshots all along. For thick LVM users on traditional SANs, Proxmox VE 9 brings new snapshot capabilities, though still as a technology preview with some limitations.

For many environments - particularly single-node setups and small clusters with modest migration needs - LVM-thin is a perfectly respectable choice that's simpler to manage than ZFS. Don't let old forum posts and outdated conventional wisdom push you toward complexity you don't need. Evaluate your actual requirements, and choose accordingly.

Chapter 5. Always Using the “host” CPU Type

When configuring a new VM in Proxmox, you’ll encounter a dropdown menu for the CPU type. Among the options, “host” stands out as the obvious choice - it passes through your physical CPU’s full feature set to the VM, promising maximum performance with zero artificial limitations. Why would you ever choose anything else?

As it turns out, there are some very good reasons. And if you’re running a cluster, choosing “host” without understanding the implications might leave you staring at a failed migration at the worst possible moment.

5.1. The Appeal of “host”

The “host” CPU type is seductive because it makes intuitive sense. Your physical CPU has a specific set of capabilities - instruction set extensions like AVX, AVX2, AVX-512, AES-NI, and dozens of others that have accumulated over generations of processor development. These extensions can significantly accelerate certain workloads, from encryption to scientific computing to video encoding.

When you select “host,” Proxmox tells QEMU to present all of these capabilities directly to the VM. The guest operating system sees exactly what’s available on the physical hardware and can take full advantage of it. No translation layer, no artificial restrictions, just raw CPU power.

For a single-node setup, this is genuinely the optimal choice. You bought that CPU with all its fancy features; you might as well use them. Performance-sensitive workloads will thank you, and there’s no downside when migration isn’t in the picture.

The problems begin when you add a second node to the equation.

5.2. The Migration Trap

Live migration is one of the most powerful features of modern virtualization. The ability to move a running VM from one physical host to another without downtime enables maintenance without service interruption, load balancing across your cluster, and graceful handling of hardware issues. It’s the foundation of true high availability.

But live migration has a fundamental requirement that’s easy to overlook: the destination host must support every CPU feature that the VM is currently using. Not “most of them” or “the important ones” - every single one.

Here’s where “host” becomes dangerous. When a VM starts on a host with a newer CPU, it discovers and potentially starts using all available features. The guest kernel loads optimized code paths, applications detect and utilize advanced instructions, and everything runs beautifully. Then you try to migrate that VM to a node with an older CPU that lacks some of those features.

The migration fails. Sometimes with a clear error message, sometimes with a cryptic QEMU complaint, and occasionally the VM just crashes on the destination host because it tried to execute an instruction that doesn’t exist. None of these outcomes are pleasant, especially if you’re migrating because the source host is having problems and you need that VM running somewhere else

immediately.

5.3. It's Not Just About CPU Generations

You might think this is only a concern if you're mixing wildly different hardware - say, a decade-old server with a brand-new workstation. But CPU feature mismatches can occur in surprisingly subtle situations.

Even within the same CPU generation from the same manufacturer, different models can have different feature sets. A Xeon might have features that an equivalent-era Core processor lacks. Microcode updates can enable or disable features. Even two seemingly identical systems might report different capabilities if one has newer firmware.

The situation becomes even more complex if you're mixing Intel and AMD processors in the same cluster. While both support the common x86-64 instruction set, their extensions diverge significantly. A VM configured with "host" on an Intel system simply cannot migrate to an AMD system, and vice versa - the feature sets are fundamentally incompatible.

5.4. Understanding CPU Models in Proxmox

Proxmox offers several predefined CPU models that represent standardized feature sets rather than specific physical processors. These models define a consistent set of capabilities that the VM will see, regardless of what the underlying hardware actually supports (as long as it supports at least those features).

You can see what's available on your system:

```
# View available CPU models (from QEMU)
cat /usr/share/pve-qemu-kvm/cpu-models.conf

# Or check the Proxmox documentation for the full list:
# https://pve.proxmox.com/pve-docs/pve-admin-guide.html#chapter_qm_vcpu_list
```

Since Proxmox VE 8.0, the default CPU type when creating a new VM through the web interface is **x86-64-v2-AES**, which adds hardware AES acceleration on top of the v2 feature set. This is a sensible middle ground for most clusters.

Model	Minimum CPU	Notes
x86-64-v2	Intel Nehalem (2008) / AMD Opteron G3	SSE4.2, POPCNT - very broad compatibility
x86-64-v3	Intel Haswell (2013) / AMD Excavator (2015)	AVX2, FMA - note: some Intel Atom CPUs lack this
x86-64-v4	Intel Skylake-SP Server (2017) / AMD Zen 4 (2022)	AVX-512 - **not** available on Intel consumer CPUs since Alder Lake

These models correspond to the x86-64 microarchitecture levels defined by processor vendors,

providing a standardized way to specify “at least this capable” without tying yourself to specific hardware.

5.5. Checking Your Cluster’s CPU Features

Before deciding on a CPU model for your cluster, it’s worth understanding exactly what each node supports. You can compare feature sets across your nodes:

```
# Show CPU flags on the current node
lscpu | grep Flags

# Or more readable, one flag per line
cat /proc/cpuinfo | grep flags | head -1 | tr ' ' '\n' | sort
```

For a quick comparison between nodes, you can extract just the flags and diff them:

```
# On each node, save flags to a file
cat /proc/cpuinfo | grep flags | head -1 | tr ' ' '\n' | sort >
/tmp/cpu_flags_$(hostname).txt

# Then compare between nodes
diff /tmp/cpu_flags_node1.txt /tmp/cpu_flags_node2.txt
```

Any flags that appear on one node but not another represent potential migration failures if you’re using the “host” CPU type.

5.6. Finding the Right Balance

The choice of CPU model is ultimately a tradeoff between performance and flexibility. The “host” setting maximizes performance but eliminates migration compatibility. The conservative models like x86-64-v2 maximize compatibility but leave some performance on the table. Your job is to find the right balance for your environment.

For single-node installations, there’s no tradeoff to make. Use “host” and enjoy full performance. Migration isn’t a concern, so there’s no reason to artificially limit your VMs.

For homogeneous clusters where every node has truly identical hardware - same CPU model, same microcode version, same BIOS settings - the “host” type can work safely. However, “truly identical” is a strong requirement that becomes harder to maintain over time as you replace failed hardware or expand your cluster.

For heterogeneous clusters, which is what most real-world clusters eventually become, you’ll want to choose a CPU model based on your oldest or least-capable node. This ensures that any VM can migrate to any node at any time.

The x86-64-v2-AES model (the Proxmox 8+ default) is the safest choice for mixed environments. If all your nodes have CPUs from 2013 or newer with confirmed AVX2 support, x86-64-v3 offers additional vector instructions that can benefit numerical workloads.

5.7. Changing CPU Type on Existing VMs

If you've already deployed VMs with the "host" CPU type and want to change them for better migration compatibility, you can modify the setting without recreating the VM:

```
# Check current CPU configuration
qm config <vmid> | grep cpu

# Change to a specific model
qm set <vmid> --cpu x86-64-v3
```

Note that the VM needs to be stopped and restarted (not just rebooted from inside the guest) for CPU model changes to take effect. The guest operating system will detect the "new" CPU features on the next boot, which is usually seamless but occasionally requires reinstalling CPU-optimized software.

5.8. Best Practices

For single-node setups, feel free to use "host" and extract maximum performance from your hardware. There's no migration to worry about, so there's no reason to compromise.

When building a cluster, decide on your CPU model strategy before deploying VMs, not after. Changing CPU types later is possible but disruptive, and you might encounter compatibility issues with software that was compiled or optimized for features that are no longer available.

For clusters with mixed hardware, identify the lowest common denominator and configure all VMs to use that feature level. Yes, this means your newest, most powerful node won't be fully utilized - but it also means you can migrate VMs freely without nasty surprises.

Document your cluster's CPU model policy somewhere visible. Future you, or whoever inherits your infrastructure, will appreciate knowing why things are configured the way they are and what constraints apply when adding new nodes.

5.9. The Bottom Line

The "host" CPU type isn't wrong - it's just not universally appropriate. In a single-node setup, it's the right choice. In a cluster, it's a decision that trades migration flexibility for performance, and you should make that trade consciously rather than by default.

Understand what your cluster actually looks like, choose a CPU model that works across all your nodes, and save yourself the frustration of discovering feature incompatibilities during an emergency migration at 3 AM. A few percentage points of CPU performance aren't worth that headache.

Chapter 6. Configuring HA Without Meeting the Prerequisites

“We have two nodes, so let’s enable HA.” This sentence has been the starting point for countless frustrating debugging sessions. High Availability sounds like exactly what you want - automatic failover when something goes wrong. But enabling HA without understanding its requirements doesn’t give you reliability; it gives you a new category of problems.

6.1. What HA Actually Means in Proxmox

Before diving into the pitfalls, let’s clarify what Proxmox’s HA system actually does. When you mark a VM or container as “HA managed,” you’re telling the cluster to monitor that workload and automatically restart it on a different node if the current node becomes unavailable. The cluster watches for node failures and takes action without human intervention.

This sounds straightforward, but the mechanism that makes it work - and the reason for the strict requirements - is the concept of quorum.

6.2. The Quorum Problem

Proxmox uses Corosync for cluster communication and membership management. Corosync implements a voting system where each node gets one vote, and the cluster can only operate when a majority of votes - more than 50% - are present. This majority is called quorum.

Why does this matter? Imagine a two-node cluster where the network connection between the nodes fails. From each node’s perspective, the other node has disappeared. Without quorum rules, both nodes would conclude that the other has failed and attempt to start the HA-managed VMs. You’d end up with the same VM running on both nodes simultaneously, potentially writing to shared storage from two places at once. This is called a split-brain scenario, and it can corrupt your data beyond recovery.

Quorum prevents this by requiring a majority to act. But here’s the math problem with two nodes: if one fails or becomes unreachable, the remaining node has exactly 1 out of 2 votes - that’s 50%, not a majority. The surviving node cannot achieve quorum, so HA services won’t failover. Your cluster becomes completely paralyzed.

This isn’t a bug; it’s the system working as designed to prevent split-brain. But it means that a two-node cluster with HA enabled is actually less reliable than no HA at all, because a single node failure takes down everything instead of just half your workloads.

6.3. The Three-Node Minimum

The official Proxmox recommendation is a minimum of three nodes for HA, and the math explains why. With three nodes and three votes, losing one node leaves two votes - still a majority. The cluster maintains quorum, and HA can function properly. Workloads from the failed node migrate to the survivors, and your services stay online.

This extends to larger clusters as well. A five-node cluster can survive two simultaneous failures.

Seven nodes can lose three. The formula is simple: a cluster with N nodes can tolerate (N-1)/2 failures, rounded down.

NOTE	When using a QDevice with an odd number of nodes, the behavior changes significantly. The QDevice provides (N-1) votes in that case, which allows all but one node to fail - but if the QDevice itself fails, no additional node failures are tolerated. For this reason, Proxmox recommends QDevices only for even-numbered clusters.

6.4. The QDevice Alternative

Not everyone can justify three full Proxmox nodes. Hardware costs money, uses power, and takes up space. For environments where a third node isn't practical, Proxmox supports an alternative: the QDevice.

A QDevice is a lightweight arbitrator that provides an additional vote without being a full cluster member. It can run on minimal hardware - a Raspberry Pi, a small VM on a separate hypervisor, or even your NAS if it runs Linux. The QDevice doesn't run VMs or store data; it just participates in quorum decisions.

IMPORTANT	The QDevice must run on hardware that is physically separate from your cluster nodes. Running a QDevice as a VM on one of your cluster nodes defeats the entire purpose - if that node fails, you lose both the node's vote and the QDevice's vote.

With two Proxmox nodes plus a QDevice, you have three votes. If one Proxmox node fails, the remaining node plus the QDevice still have two out of three votes - a majority. HA can now function correctly.

Setting up a QDevice requires a host running the `corosync-qnetd` daemon:

```
# On the QDevice host (any Debian/Ubuntu system, Raspberry Pi, etc.)
apt update
apt install corosync-qnetd

# On one of your Proxmox nodes
pvecm qdevice setup <qdevice-ip-address>
```

After setup, verify the configuration:

```
# Check cluster status including QDevice
pvecm status

# Verify quorum votes - should show 3 total votes
corosync-quorumtool
```

You should see three votes total: one from each Proxmox node and one from the QDevice. The QDevice status shows flags like `A,V,NMW` where `A` means Alive and `V` means it's casting a Vote.

6.5. Fencing: The Other Half of HA

Quorum determines whether the cluster can act. Fencing determines how it acts. When a node is declared dead and its workloads need to move elsewhere, the cluster must be absolutely certain that the failed node won't suddenly come back and try to access shared resources.

Watchdog-Based Self-Fencing (Default)

Proxmox uses watchdog-based self-fencing by default. This is simpler and often more reliable than external fencing mechanisms. Here's how it works:

1. The `pve-ha-lrm` service continuously resets a watchdog timer
2. If a node loses quorum for approximately 60 seconds, the service stops resetting the watchdog
3. The watchdog triggers and reboots the node
4. When the node comes back up, it won't have quorum and won't start HA services, preventing split-brain

The Linux kernel's software watchdog (softdog) is enabled by default - no installation or configuration required. This is intentional and works well for most environments.

```
# Verify watchdog is active (look for watchdog-mux)
systemctl status watchdog-mux

# Check which watchdog module is loaded
cat /etc/default/pve-ha-manager
```

Hardware Watchdog (Optional)

For higher reliability, you can use a hardware watchdog if your server supports one. Hardware watchdogs are independent of the operating system and will trigger even if the kernel crashes.

```
# Edit the HA manager configuration
nano /etc/default/pve-ha-manager

# Uncomment and set the appropriate module for your hardware:
# Intel servers: iTCO_wdt
# HP servers: hpwdt
# Dell with IPMI: ipmi_watchdog
WATCHDOG_MODULE=iTCO_wdt
```

After changing the watchdog module, restart the watchdog-mux service:

```
systemctl restart watchdog-mux
```

External Fencing (STONITH)

For environments requiring active fencing - where surviving nodes forcibly power off the failed node rather than waiting for self-fencing - Proxmox supports hardware fence devices through IPMI, iDRAC, iLO, or managed PDUs.

```
# Check current fencing mode
grep fencing /etc/pve/datacenter.cfg

# View hardware fencing configuration (if configured)
cat /etc/pve/ha/fence.cfg
```

To enable hardware fencing, install the fence agents and configure your devices:

```
# Install fence agents
apt install fence-agents

# Configure via GUI: Datacenter → HA → Fencing → Add
# Or set the mode in datacenter.cfg:
# fencing: hardware
# fencing: both    (uses watchdog AND hardware fencing)
```

For homelab environments without IPMI or out-of-band management, the default watchdog-based self-fencing is typically sufficient and more reliable than attempting to configure complex external fencing.

6.6. Network Redundancy

HA depends entirely on cluster communication. If Corosync can't reach other nodes, it assumes they're dead - even if they're perfectly healthy and just unreachable due to a network issue. A single network failure shouldn't trigger HA failovers and potentially cause the split-brain scenario that quorum is designed to prevent.

The solution is redundant networks for cluster communication. Proxmox supports multiple Corosync links, allowing you to define separate network paths:

```
# View current Corosync configuration
cat /etc/pve/corosync.conf
```

When setting up a cluster, you can specify multiple links:

```
# When joining a cluster with redundant links
pvecm add <existing-node> --link0 <ip-link0> --link1 <ip-link1>
```

Ideally, these links should use physically separate networks - different switches, different subnets, maybe even different network interface cards. The goal is eliminating single points of failure in

cluster communication.

TIP	For critical environments, consider using dedicated network interfaces for Corosync traffic, separate from VM traffic and storage networks. This prevents heavy VM network usage from impacting cluster communication.

6.7. Testing Before Production

HA configurations should be tested before you rely on them. This means actually simulating failures and verifying that failover works as expected.

```
# Check HA status
ha-manager status

# View HA resource configuration
cat /etc/pve/ha/resources.cfg

# View HA groups
cat /etc/pve/ha/groups.cfg

# Check cluster health
pvecm status
```

Test scenarios should include:

- **Graceful shutdown**: Shut down a node cleanly and verify VMs migrate to surviving nodes. This tests the basic HA mechanism.
- **Hard power-off**: Pull the power cord (or use IPMI to force power off) to simulate sudden hardware failure. The remaining nodes should detect the failure and restart services after the fencing timeout (~2 minutes with watchdog fencing).
- **Network disconnect**: Unplug the cluster network cable from one node. The isolated node should self-fence (reboot) after losing quorum. This is the most important test for split-brain prevention.
- **QDevice failure**: If using a QDevice, shut it down while all Proxmox nodes are healthy. The cluster should continue operating. Then test what happens if a Proxmox node fails while the QDevice is down.

WARNING	Always test in a maintenance window with proper backups. Test with non-critical VMs first. Never test by disconnecting storage - that can cause data corruption regardless of HA status.

Don’t wait for a real failure to discover that your HA configuration doesn’t actually work. By then, you’ll be dealing with an outage and trying to debug complex cluster behavior under pressure.

6.8. Common Pitfalls

Ignoring the two-minute timeout: Watchdog-based fencing has an intentional delay of approximately two minutes. This is not a bug - it's necessary to ensure the failed node has truly fenced itself before services restart elsewhere. Don't try to reduce this timeout; it exists to prevent split-brain.

Running QDevice on cluster storage: If your QDevice VM's storage is on the Proxmox cluster itself, a storage failure takes down both your cluster and your quorum arbitrator simultaneously.

Single network for everything: Using one network for Corosync, VM traffic, and storage means any network congestion can trigger false HA failovers. Separate your traffic.

Not testing after changes: Any change to cluster configuration, network setup, or storage should be followed by HA testing. What worked before might not work after the change.

6.9. Best Practices

Only enable HA when you've genuinely met the prerequisites. Three nodes minimum, or two nodes with a properly configured QDevice. Anything less isn't HA - it's a configuration that will make failures worse.

Understand your fencing mechanism before enabling HA on production workloads. Know whether you're using software watchdog, hardware watchdog, or external fencing, and understand the implications of each.

Use redundant networks for Corosync communication. A single switch failure shouldn't take down your entire cluster or trigger unnecessary failovers.

Test your HA setup by actually causing failures in a controlled manner. Verify that failover works, that fencing triggers correctly, and that services come back up on surviving nodes. Document the expected behavior so you know what "working correctly" looks like.

Monitor your cluster continuously. Set up alerts for quorum issues, fencing events, and HA state changes. Problems with HA often start as intermittent issues that become critical failures if ignored.

6.10. The Bottom Line

High Availability requires actual redundancy, not just a checkbox in the Proxmox interface. Two nodes without a QDevice isn't an HA cluster; it's a cluster that will lock up completely when you need HA most. Meet the prerequisites, test your configuration, and understand the mechanisms involved. Only then does HA deliver on its promise of keeping your services running through hardware failures.

Chapter 7. Incomplete or Missing Backup Strategy

Backups are like insurance - everyone agrees they're important, but somehow there's always something more urgent to deal with. The new VM that needs deploying, the performance issue that needs investigating, the update that can't wait. Backup configuration gets pushed to "next week" until one day a disk fails, a ransomware attack hits, or someone accidentally deletes the wrong VM.

At that point, "next week" becomes "too late."

7.1. The Uncomfortable Truth

Here's a statistic that should keep you up at night: most backup failures aren't discovered until a restore is needed. Systems dutifully report successful backup jobs for months or years, but when disaster strikes, the backups turn out to be corrupted, incomplete, or simply not what anyone expected.

This happens because backup configuration is often treated as a one-time setup task rather than an ongoing operational responsibility. You configure vzdump, see it run successfully a few times, and move on. Meanwhile, new VMs get created without backup jobs, storage fills up and starts silently failing, and retention policies delete the one backup you actually needed.

A backup you haven't tested is not a backup. It's a hope.

7.2. vzdump: Where Most People Start

Proxmox includes vzdump as its built-in backup tool, and for simple setups it works reasonably well. You can schedule backups to local storage, NFS shares, or SMB mounts, and the process is straightforward to configure through the web interface.

```
# Manual backup of a single VM
vzdump 100 --storage backup-storage --mode snapshot

# Check backup job configuration
cat /etc/pve/jobs.cfg
```

For a homelab with a handful of VMs backing up to a NAS, vzdump on a schedule is perfectly adequate. But as your environment grows, its limitations become apparent.

Each backup is a complete copy of the VM. If you have a 500 GB VM and back it up daily, you're writing 500 GB every single day, even if only a few megabytes actually changed. Storage fills up quickly, backup windows grow longer, and network bandwidth becomes a bottleneck if you're backing up to remote storage.

7.3. Proxmox Backup Server: The Modern Approach

Proxmox Backup Server (PBS) is Proxmox's purpose-built solution for these problems, and it's

become the recommended approach for any serious Proxmox deployment. If you're still relying solely on vzdump to local or network storage, PBS deserves your attention.

The fundamental difference is that PBS uses chunk-based deduplication. Instead of storing complete VM images, PBS breaks data into chunks, calculates checksums, and stores each unique chunk only once. When you back up a 500 GB VM where only 100 MB changed since yesterday, PBS transfers and stores only those 100 MB of new chunks. Everything else is deduplicated against existing data.

The storage savings are dramatic. Environments with many similar VMs - think identical base images with different configurations - can see deduplication ratios of 10:1 or higher. Even diverse environments typically achieve 2:1 to 5:1, meaning your backup storage goes much further.

But deduplication isn't even the biggest advantage. PBS also provides:

Incremental backups that transfer only changed blocks, reducing backup windows from hours to minutes and dramatically cutting network bandwidth requirements.

Built-in encryption that protects your backups at rest and in transit. You control the keys, so even if someone gains access to the backup storage, the data remains protected.

Integrity verification that goes beyond "did the backup job complete." PBS can verify that stored chunks are readable and match their recorded checksums, catching storage corruption before you need to restore.

Granular restore options including file-level restore that lets you pull individual files from a backup without restoring the entire VM. When someone deletes an important document, you don't need to spin up the whole backup image to retrieve it.

7.4. Setting Up PBS Integration

Adding a PBS server to your Proxmox environment is straightforward. On the Proxmox VE side, you add PBS as a storage target:

```
# Add PBS storage
# Use --password without value for interactive prompt (recommended)
pvesm add pbs pbs-storage --server <pbs-ip> \
    --datastore <datastore-name> \
    --username backup@pbs \
    --password \
    --fingerprint <server-fingerprint>
```

Once configured, PBS appears as a backup destination in your backup job configuration. You can then schedule backups just like with vzdump, but the underlying mechanism is dramatically more efficient.

On the PBS side, you'll want to configure retention policies that match your recovery requirements:

```
# Example retention in PBS datastore.cfg:
# prune-schedule daily
# keep-last 3
# keep-daily 7
# keep-weekly 4
# keep-monthly 6
```

7.5. The 3-2-1 Rule

No discussion of backup strategy is complete without mentioning the 3-2-1 rule: maintain three copies of your data, on two different types of media, with one copy offsite.

In a Proxmox context, this might look like:

1. **Production data** on your local storage (copy one)
2. **PBS backups** on a dedicated backup server (copy two, different media if PBS uses different disks)
3. **Offsite replication** to a remote PBS instance, cloud storage, or physical media stored elsewhere (copy three, offsite)

PBS supports remote sync to another PBS server, making the offsite component straightforward to implement:

```
# On the remote PBS, create a sync job pulling from the primary
# Configured via PBS web interface under Datastore > Sync Jobs
```

For homelab environments where a second physical location isn't practical, even syncing critical backups to cloud storage provides protection against local disasters like fire or theft.

7.6. Testing Your Restores

This point cannot be emphasized enough: **regularly test your restores**. Not just "click restore and see if it starts," but actually verify that the restored system works correctly.

Schedule periodic restore tests as part of your operational routine:

```
# Restore a VM to verify backup integrity (to a different VMID)
qmrestore <backup-file-or-pbs-reference> <new-vmid> --storage <target-storage>
```

Boot the restored VM on an isolated network, verify that applications work, check that recent data is present. Then delete the test restore and document that you verified backups on this date. When disaster strikes, you'll have confidence that your recovery process actually works.

PBS includes built-in verification jobs that check backup integrity without performing full restores:

```
# Verification runs can be scheduled in PBS
# They read and verify all chunks in a backup
```

This catches storage-level corruption but doesn't verify that the backup is actually bootable and functional. Both types of testing have their place.

7.7. Common Backup Mistakes

Beyond not having backups at all, several common mistakes undermine backup strategies:

Not backing up new VMs: You create a VM, get it working, and forget to add it to backup jobs. Weeks later, that VM fails, and you discover it was never backed up. Audit your backup coverage regularly.

Insufficient retention: Keeping only the last few backups seems space-efficient until you need to recover from something that happened two weeks ago and your oldest backup is from yesterday. Ransomware in particular often lurks for days or weeks before activating.

No offsite copies: Your backup server sitting next to your production server provides no protection against theft, fire, flood, or electrical events that take out your entire setup.

Backing up to the same storage: If your VM disks and backups live on the same physical storage, a storage failure destroys both. Backups should always target separate physical media.

7.8. Best Practices

Implement PBS if you're running more than a handful of VMs. The efficiency gains and additional features are worth the setup effort, and it's free to use.

Follow the 3-2-1 rule as closely as your budget and infrastructure allow. Two copies on the same server isn't a backup strategy; it's a false sense of security.

Test restores regularly and document the results. Quarterly full restore tests for critical systems isn't overkill - it's due diligence.

Monitor your backup jobs actively. Configure notifications for failures, and review backup logs periodically even when things appear to be working. Silent failures are the most dangerous kind.

Audit backup coverage whenever your environment changes. New VMs, new containers, new critical data - all should trigger a review of what's being backed up and whether coverage is adequate.

7.9. The Bottom Line

Backups are the last line of defense against data loss, and they only work if they're comprehensive, tested, and properly maintained. A backup strategy isn't something you configure once and forget; it's an ongoing operational responsibility that requires attention, testing, and regular verification. Invest the time now, while everything is working, so you're not scrambling to recover from disaster without a safety net.

Chapter 8. Running Docker Directly on the Proxmox Host

It starts innocently enough. You need a quick container for a small service - maybe a monitoring agent, a reverse proxy, or a simple web application. Spinning up a whole VM feels like overkill for something so lightweight. Docker is already familiar, the container image exists, and installing Docker on the Proxmox host takes just a few commands.

What could possibly go wrong?

As it turns out, quite a lot. And the Proxmox documentation explicitly warns against this practice for good reasons.

8.1. Why It's Tempting

The appeal is understandable. Docker containers are lightweight, start in seconds, and there's an image for practically everything. Compared to creating a VM, installing an OS, updating packages, and then deploying your application, pulling a Docker image feels delightfully efficient.

Proxmox is just Debian under the hood, after all. Docker runs fine on Debian. The installation works, containers start, and everything seems fine. For days, weeks, maybe even months, you might not notice any problems.

Then one day your VMs can't reach the network, or the Proxmox firewall stops working correctly, or you encounter bizarre permission errors that make no sense. Welcome to the consequences of mixing container runtimes on your hypervisor.

8.2. The iptables Conflict

The most common and most frustrating issue involves iptables, the Linux kernel's packet filtering framework. Both Proxmox and Docker manipulate iptables rules to manage network traffic, and they don't coordinate with each other.

Proxmox uses iptables for its built-in firewall, managing rules that control traffic to and from VMs and the host itself. Docker also uses iptables extensively - it creates rules for container networking, port forwarding, and network isolation between containers.

When both systems modify iptables independently, conflicts are inevitable. Docker might insert rules that bypass Proxmox's firewall entirely, exposing services you thought were protected. Proxmox firewall changes might break Docker networking, leaving containers unable to communicate. The symptoms vary, but the root cause is the same: two systems fighting over the same resource.

```
# Check current iptables rules - you might be surprised what you find
iptables -L -n -v

# Docker adds its own chains
iptables -L DOCKER -n -v
```

After installing Docker, you'll see chains like `DOCKER`, `DOCKER-USER`, and `DOCKER-ISOLATION` interleaved with Proxmox's rules. Debugging network issues becomes an exercise in figuring out which system added which rule and why.

8.3. Kernel and Namespace Conflicts

Beyond iptables, Docker and Proxmox's virtualization stack can conflict at deeper levels. Both systems rely heavily on Linux kernel features like namespaces and cgroups for isolation.

Proxmox uses these features to isolate VMs and LXC containers from each other and from the host. Docker uses the same features for its containers. While the kernel can technically handle multiple users of these subsystems, the management layers don't always play nicely together.

Resource limits set by Proxmox might affect Docker containers in unexpected ways. Docker's manipulation of cgroups might interfere with Proxmox's resource accounting. The interactions are complex and often manifest as strange, hard-to-diagnose behavior rather than clear error messages.

8.4. Security Implications

Running Docker on the hypervisor expands your attack surface significantly. The Proxmox host is the most privileged component in your infrastructure - compromise it, and an attacker has access to every VM and container on the system.

Docker containers, despite their isolation, run on the host kernel. Container escape vulnerabilities, while rare, do exist. Running Docker directly on your hypervisor means a container escape gives the attacker control of your entire virtualization infrastructure, not just one VM.

This is why defense in depth matters. When Docker runs inside a VM, a container escape compromises only that VM. The hypervisor remains protected by an additional layer of isolation.

Additionally, Docker's default networking model can expose services unexpectedly. Containers bound to `0.0.0.0` are accessible on all host interfaces unless you're careful with your iptables rules - which, as we discussed, are already being fought over by multiple systems.

8.5. The Upgrade Risk

Proxmox major version upgrades are generally smooth, but they do modify system components. When you've installed Docker and potentially other software directly on the host, upgrades become riskier. Package conflicts, configuration file changes, and dependency issues are all more likely when the host is running software beyond what Proxmox expects.

I've seen upgrades fail because Docker's dependencies conflicted with updated Proxmox packages.

Recovering from a failed upgrade on your hypervisor - with all your VMs inaccessible - is not a fun experience.

Keeping your Proxmox host minimal reduces upgrade risk and makes troubleshooting easier. When something breaks, you're not wondering whether it's a Proxmox issue, a Docker issue, or some interaction between the two.

8.6. The Right Approach: Docker in a VM

The solution is simple: run Docker in a dedicated VM. The overhead is minimal with modern virtualization, and the benefits are substantial.

Create a lightweight VM for Docker workloads:

```
# Create a minimal VM for Docker (via CLI or web interface)
qm create 200 --name docker-host --memory 2048 --cores 2 \
    --net0 virtio,bridge=vmbr0 --scsihw virtio-scsi-single
```

Install your preferred Linux distribution - Debian, Ubuntu, or even a purpose-built Docker host like Photon OS - and then install Docker inside that VM.

This approach provides:

Clean separation between your hypervisor and your container workloads. Proxmox manages virtualization; the Docker VM manages containers.

Independent management of Docker updates, configuration changes, and experiments without risking your hypervisor.

Better security through an additional isolation layer. Container escapes stay contained within the VM.

Easier backups since the Docker VM can be backed up like any other VM, capturing all containers and their data in a consistent state.

Simpler troubleshooting because you know which system is responsible for what.

8.7. What About LXC Containers?

Proxmox's native LXC containers offer another path for lightweight workloads. LXC containers are more tightly integrated with Proxmox, avoiding the conflicts that Docker causes.

For applications where an LXC container works well, it's often the better choice on Proxmox. Many services that you might run in Docker are also available as LXC templates or can be installed directly in a minimal LXC container.

```
# Download and create an LXC container
pveam download local debian-12-standard_12.2-1_amd64.tar.zst
pct create 300 local:vztmpl/debian-12-standard_12.2-1_amd64.tar.zst \
    --hostname my-container --memory 512 --net0 name=eth0,bridge=vmbr0,ip=dhcp
```

However, if you specifically need Docker's ecosystem - the image library, Docker Compose workflows, or compatibility with container-based CI/CD pipelines - then a Docker VM is the cleaner solution.

You can even run Docker inside an LXC container. With Proxmox 7+, this works in unprivileged containers by enabling the `nesting` and `keyctl` features. However, there are caveats: ZFS-backed containers may need `fuse-overlayfs` for the Docker storage driver, and backups can be problematic. A VM remains the safer and more straightforward option.

8.8. Best Practices

Keep your Proxmox host clean. It should run Proxmox and essential management tools, nothing more. Resist the temptation to install "just one quick thing" directly on the hypervisor.

Create a dedicated VM for Docker workloads. The resource overhead is minimal, and the isolation benefits are significant. If you have multiple Docker use cases with different security requirements, consider separate VMs rather than mixing everything together.

Use Proxmox's native LXC containers when they fit your needs. They integrate better with Proxmox's management tools and avoid the conflicts that Docker introduces.

If you absolutely must run containers directly on the host for some reason, consider Podman as an alternative to Docker. Podman is daemonless and rootless by default, avoiding some (though not all) of the conflicts Docker creates. But even then, a VM is usually the better choice.

Think of your Proxmox host as infrastructure, not as a general-purpose server. The more focused its role, the more reliable and maintainable it becomes.

8.9. The Bottom Line

Docker on the Proxmox host works until it doesn't, and when it fails, the problems are frustrating and time-consuming to debug. The convenience of skipping a VM isn't worth the networking conflicts, security risks, and upgrade complications.

Take the extra five minutes to create a Docker VM. Your future self, not debugging iptables rules at midnight, will thank you.

Chapter 9. No Monitoring for Nodes, Storage, and Backups

There's a particular kind of silence in IT infrastructure that feels peaceful but shouldn't. It's the silence of a system running without monitoring - where everything appears fine because nothing is telling you otherwise. No alerts, no warnings, no dashboards turning red. Just quiet confidence that your Proxmox environment is humming along perfectly.

Then a disk fails. Or rather, a disk failed three weeks ago, and the degraded array you didn't know about just lost a second disk. Or your backup storage filled up two months ago, and every backup since has silently failed. Or your ZFS pool has been accumulating checksum errors that would have told you a disk was dying if anyone had been watching.

Monitoring isn't about generating pretty graphs for a dashboard you glance at occasionally. It's about knowing the health of your infrastructure before problems become emergencies, and having the information you need to make decisions when something does go wrong.

9.1. The False Comfort of "No News"

Proxmox provides a web interface that shows the current state of your nodes, VMs, and storage. It's useful for day-to-day management, but it's not monitoring. It shows you what's happening right now, when you choose to look. It doesn't alert you when something changes, track trends over time, or wake you up at 3 AM when a critical threshold is crossed.

The distinction matters because infrastructure problems rarely announce themselves with immediate, obvious failures. A disk develops bad sectors gradually. Memory errors accumulate over time. Storage fills up megabyte by megabyte. CPU temperatures creep higher as fans accumulate dust. By the time these issues cause visible symptoms - VMs crashing, services failing, data corruption - the underlying problem has often progressed far beyond easy remediation.

Effective monitoring catches these trends early. It notices when disk latency increases before users complain about slowness. It alerts you to SMART warnings before the disk actually fails. It tracks storage consumption and tells you when you'll run out of space at the current growth rate, not when you've already run out.

9.2. What You Should Be Watching

A comprehensive monitoring setup for Proxmox environments covers several distinct areas, each with its own metrics and alert thresholds.

Node Health

Your Proxmox hosts are the foundation everything else depends on. At minimum, monitor CPU utilization, memory usage, system load, and network throughput. But don't stop at simple utilization metrics - context matters.

High CPU utilization during a scheduled backup window is expected. The same utilization at 3 PM on a Tuesday might indicate a runaway process or a VM under attack. Memory pressure that causes

swapping degrades performance across all VMs, not just the one consuming memory. Network errors and dropped packets can indicate cable problems, switch issues, or driver bugs.

Temperature monitoring is often overlooked but particularly important for homelab environments where cooling might be less robust than a proper data center. A node running hot isn't just inefficient; it's aging its components faster and heading toward thermal throttling or shutdown.

```
# Quick node health check
echo "=== CPU ===" && top -bn1 | head -5
echo "=== Memory ===" && free -h
echo "=== Load ===" && uptime
echo "=== Temperatures ===" && sensors 2>/dev/null || echo "Install lm-sensors"
```

Storage Health

For any storage system, but especially for ZFS, monitoring is non-negotiable. ZFS provides extensive self-reporting capabilities that tell you exactly what's happening with your data - but only if you're listening.

Pool status should be checked regularly for degraded vdevs, ongoing scrubs or resilvers, and any errors:

```
# Comprehensive ZFS pool status
zpool status -v

# Check for any errors (should return nothing if healthy)
zpool status -x
```

A healthy pool reports "all pools are healthy." Anything else demands immediate attention.

Scrub results are your primary insight into data integrity. ZFS scrubs read all data in a pool and verify checksums, detecting silent corruption that wouldn't otherwise be visible. Check when scrubs last ran and what they found:

```
# Last scrub details are in zpool status output
zpool status tank | grep -A5 "scan:"
```

A scrub that reports checksum errors is telling you something is wrong - possibly a dying disk, bad memory, or a controller issue. Don't ignore these warnings. Investigate the source before a recoverable situation becomes data loss.

For traditional storage, SMART monitoring provides early warning of disk failures:

```
# Check SMART status for all disks
for disk in /dev/sd?; do
    echo "=== $disk ==="
    smartctl -H $disk
```

```
done

# Detailed SMART attributes for a specific disk
smartctl -A /dev/sda
```

Key SMART attributes to watch include Reallocated Sector Count, Current Pending Sector Count, and Uncorrectable Sector Count. Any non-zero values in these fields indicate a disk that's starting to fail. It might keep working for months, or it might die tomorrow - but it will die, and you should plan accordingly.

Backup Health

A backup job that completes isn't necessarily a backup job that succeeded. Backup monitoring should verify not just that jobs ran, but that they produced valid, restorable backups of the systems you intended to protect.

Track backup job success and failure rates. A single failed backup might be a transient issue; repeated failures indicate a problem that needs investigation. More insidiously, watch for backups that succeed but shrink unexpectedly - this can indicate that something in the source VM changed and the backup is no longer capturing what you think it is.

For Proxmox Backup Server, the verification feature should be part of your monitoring strategy:

```
# On PBS, check recent backup and verification status
proxmox-backup-manager task list --limit 20
```

Storage consumption trends matter for backup destinations. Running out of space doesn't just stop new backups; depending on your retention policies, it might prevent pruning of old backups too, creating a deadlock that requires manual intervention.

9.3. Checkmk: A Monitoring Solution That Fits

While many monitoring solutions exist, Checkmk deserves special attention for Proxmox environments. It strikes an excellent balance between comprehensive out-of-the-box functionality and the flexibility to customize when needed. The Raw Edition is fully open source and remarkably capable - you don't need to pay for a license to get professional-grade monitoring.

What makes Checkmk particularly well-suited for Proxmox is its native understanding of virtualization environments. Rather than treating your hypervisor as just another Linux server, Checkmk includes specific checks for Proxmox VE that understand the relationship between hosts, VMs, containers, and storage.

Setting Up Checkmk for Proxmox

Getting started requires a Checkmk server - this can run in a VM on your Proxmox cluster or on separate hardware. For homelabs, running it as a VM is perfectly fine; for production environments, consider placing it on independent infrastructure so it remains available even if your main cluster has issues.

```
# Download and install Checkmk Raw Edition on a Debian/Ubuntu system
# Check checkmk.com for the latest version
wget https://download.checkmk.com/checkmk/2.3.0p1/check-mk-raw-
2.3.0p1_0.jammy_amd64.deb
apt install ./check-mk-raw-2.3.0p1_0.jammy_amd64.deb

# Create a monitoring site
omd create mysite
omd start mysite
```

Once your Checkmk server is running, you'll install agents on your Proxmox hosts. The agent is a simple shell script that collects system information and outputs it in a format Checkmk understands:

```
# On each Proxmox host, install the Checkmk agent
apt install check-mk-agent

# Verify the agent works
check_mk_agent | head -100
```

The agent output includes everything from CPU and memory statistics to disk health, network interfaces, and running processes. Checkmk's server periodically queries this agent and processes the results.

Proxmox-Specific Monitoring with Checkmk

Beyond the standard Linux monitoring, Checkmk offers a dedicated Proxmox VE special agent that queries the Proxmox API directly. This provides visibility into Proxmox-specific objects that the standard Linux agent can't see:

- VM and container status, resource allocation, and actual usage
- Cluster health and quorum status
- Storage pool status across all configured storage backends
- Backup job status and history
- Replication status for ZFS replication jobs
- Node membership and cluster communication health

To enable Proxmox monitoring, you'll configure the special agent in Checkmk's web interface. You'll need API credentials from Proxmox:

```
# On Proxmox, create an API user for monitoring (read-only)
pveum user add monitoring@pve
pveum aclmod / -user monitoring@pve -role PVEAuditor
pveum user token add monitoring@pve checkmk-token -privsep 0
# Note: In Checkmk, use "monitoring@pve" as username and the token secret
```

Store the token ID and secret - you'll enter these in Checkmk when configuring the Proxmox special agent rule.

The Proxmox integration automatically discovers your VMs and containers, creating monitoring objects for each. You can track which host each VM runs on, receive alerts when VMs are stopped unexpectedly, and monitor resource usage at both the VM and host level.

ZFS Monitoring in Checkmk

Checkmk includes robust ZFS monitoring that goes far beyond basic pool status. The ZFS checks monitor:

- Pool health status with alerts for degraded or faulted vdevs
- Scrub status including time since last scrub and any errors found
- Capacity utilization with configurable warning and critical thresholds
- ARC statistics including hit rates and size
- Individual vdev and disk status

The ZFS checks are automatically discovered when you add a host running ZFS. No additional configuration required - Checkmk detects the ZFS pools and creates appropriate service checks.

```
# The Checkmk agent includes ZFS sections automatically
check_mk_agent | grep -A20 "<<<zpool_status>>>"
```

For environments where ZFS is critical - which is most Proxmox environments - these checks provide early warning of developing problems. A scrub that finds checksum errors, a vdev showing elevated read errors, or ARC hit rates dropping significantly all generate alerts before they escalate into data loss.

SMART Disk Monitoring

Checkmk monitors SMART attributes for all physical disks, tracking the metrics that predict disk failure. Configure the SMART check to alert on:

- Reallocated sector counts above zero
- Current pending sectors indicating bad blocks waiting for reallocation
- Spin retry counts suggesting mechanical issues
- Temperature readings outside safe ranges
- Raw read error rates trending upward

When a disk starts showing SMART warnings, you'll know about it with time to order a replacement and migrate data gracefully - rather than discovering the failure when the disk stops responding entirely.

Building Useful Dashboards

Checkmk's interface lets you create custom dashboards that show the information you care about at a glance. For a Proxmox environment, a useful dashboard might include:

- Overall cluster health status (all nodes, quorum status)
- Storage utilization across all pools with trend graphs
- Top VMs by CPU and memory usage
- Recent alerts and problems
- Backup job status for the last 24 hours

The dashboard becomes your morning check-in point - a quick glance confirms everything is healthy, or immediately highlights what needs attention.

Notification Configuration

Checkmk supports flexible notification rules that control who gets alerted about what, and through which channels. Configure notifications based on:

- Severity (warning vs. critical)
- Time of day (different rules for business hours vs. nights/weekends)
- Host or service groups (storage alerts to one team, network alerts to another)
- Escalation (if the first contact doesn't acknowledge within an hour, notify the next person)

For homelab use, email notifications are often sufficient. Checkmk also supports SMS, Slack, Microsoft Teams, PagerDuty, and many other notification methods. The key is ensuring that critical alerts actually reach you - test your notification configuration regularly.

```
# Checkmk notification test from command line
# Useful for verifying email delivery works
cmk --notify test
```

9.4. Other Monitoring Options

While Checkmk is excellent, it's not the only choice. Your existing infrastructure and preferences might point toward different solutions.

Prometheus with Grafana has become the de facto standard for cloud-native, metrics-focused monitoring. It's highly flexible, scales well, and has an enormous ecosystem of exporters. The learning curve is steeper than Checkmk, but the customization possibilities are extensive. For Proxmox, you'll want the PVE exporter alongside the standard node exporter.

Zabbix occupies a middle ground - more configurable than Checkmk, more turnkey than Prometheus. It handles both metrics and log-based monitoring, supports complex trigger conditions, and has extensive template libraries including Proxmox templates.

For smaller environments or homelabs, even simpler solutions can work. A well-crafted shell script running via cron, checking critical metrics and sending alerts, provides better visibility than no

monitoring at all. But as your environment grows, you'll appreciate having a proper monitoring system with history, dashboards, and sophisticated alerting.

9.5. Alerting: The Point of Monitoring

Collecting metrics is only valuable if something acts on them. Alerting transforms monitoring from passive data collection into active infrastructure management.

Effective alerts share several characteristics. They're actionable - when an alert fires, there's something specific you can and should do in response. They have appropriate thresholds - sensitive enough to catch real problems, but not so sensitive that you're drowning in false positives. And they reach you through channels you'll actually notice.

Define clear severity levels and response expectations. A warning about 80% disk utilization can wait until morning. An alert about a degraded ZFS pool needs attention within hours. A critical alert about active data corruption needs immediate response regardless of the hour.

Whatever monitoring system you choose, invest time in tuning your alerts. Start with conservative thresholds and adjust based on experience. Track false positives and refine the conditions that trigger them. The goal is a monitoring system where every alert means something - so that when an alert fires, you take it seriously rather than assuming it's another false alarm.

9.6. Building a Monitoring Culture

Technical implementation is only part of effective monitoring. You also need habits and processes that ensure monitoring remains useful over time.

Review your dashboards regularly, even when nothing is alerting. Look for trends that haven't yet crossed alert thresholds. Notice when metrics look different than usual, even if "different" isn't "bad." This kind of proactive attention catches issues that automated alerts miss.

Keep your monitoring configuration updated as your infrastructure changes. New VMs should be added to monitoring automatically or as part of your provisioning checklist. Decommissioned systems should be removed so they don't generate confusing alerts or clutter your dashboards.

Document your monitoring setup - what's being monitored, what the alert thresholds mean, and what response each alert expects. When you're troubleshooting an incident at 2 AM, clear documentation saves precious time and reduces errors.

Test your alerting periodically. Intentionally trigger an alert condition and verify that notifications arrive through all configured channels. Discover notification failures during a test, not during an actual emergency.

9.7. Best Practices

Deploy monitoring before you think you need it. Setting up monitoring while everything is working gives you a baseline of normal behavior. Setting it up during an incident means you're learning the tools while also fighting a fire.

Monitor the monitoring. If your monitoring system itself fails, you want to know. Configure health

checks for your monitoring infrastructure, and consider where those alerts go if the primary alerting system is down.

Retain historical data long enough to be useful. Being able to compare current performance to last month, last quarter, or last year provides context that's invaluable for capacity planning and troubleshooting. Storage is cheap; insight is valuable.

Start simple and expand. A basic monitoring setup that you actually maintain is better than an elaborate system you set up once and then ignore because it's too complex. Begin with critical metrics, add coverage over time, and continuously refine based on what proves useful.

Treat monitoring gaps as risks. If a system isn't monitored, assume you don't know its health. Unmonitored systems have a way of failing silently until the failure cascades into something visible - by which point the original problem may be much harder to diagnose.

9.8. The Bottom Line

Monitoring is the difference between proactive infrastructure management and reactive firefighting. Without it, you're operating blind, discovering problems only when they've already impacted your services or your data. With proper monitoring in place, you see issues developing, address them on your schedule, and maintain the kind of visibility that transforms system administration from stressful guesswork into informed decision-making.

The time to implement monitoring is now, while everything is running smoothly. Your future self, calmly addressing a warning alert before it becomes a critical failure, will be grateful you made the investment.

Chapter 10. Deploying Services Directly on the Proxmox Host

It starts with a simple thought: “I just need a small web server for internal documentation.” Or maybe it’s a database for a quick project, a file synchronization service, or a VPN endpoint. The Proxmox host is right there, already running, with plenty of spare resources. Creating a whole VM for something so simple feels like overkill. So you run `apt install` directly on the hypervisor, configure the service, and move on with your day.

Weeks or months later, that “one small service” has been joined by a few friends. There’s a reverse proxy handling SSL termination, a monitoring agent you installed manually, maybe a cron job that syncs files somewhere. The Proxmox host has gradually transformed from a focused hypervisor into a general-purpose server that also happens to run virtual machines.

This path leads somewhere you don’t want to go.

10.1. The Creeping Complexity Problem

The fundamental issue isn’t any single service - it’s the accumulation. Each additional package installed on the Proxmox host adds dependencies, configuration files, running processes, and potential interactions with the base system. Individually, these might be harmless. Collectively, they create a system that’s increasingly difficult to understand, maintain, and troubleshoot.

Proxmox VE is built on Debian, which means it participates in Debian’s package management ecosystem. When you install additional software, you’re adding packages that have their own dependencies, which might overlap with or conflict with packages that Proxmox itself depends on. Today everything works fine. But the next time you run `apt upgrade`, a dependency resolution might not go the way you expected.

I’ve seen Proxmox upgrades fail because a manually installed service required a library version that conflicted with updated Proxmox packages. The resulting state - half-upgraded, with broken dependencies - is exactly where you don’t want your hypervisor to be, especially if you have production VMs depending on it.

10.2. The Upgrade Risk

Speaking of upgrades, major Proxmox version upgrades deserve special consideration. Moving from one major version to the next involves significant package changes, and Proxmox provides upgrade guides that assume a relatively standard installation. The more you’ve customized your host, the more likely you are to encounter unexpected issues.

Proxmox’s upgrade documentation even explicitly warns about this. They can’t test every possible combination of additional software, and they can’t guarantee that your custom services will survive the upgrade intact. A clean Proxmox host with minimal modifications follows the documented upgrade path smoothly. A host that’s accumulated years of additional software becomes an upgrade adventure with uncertain outcomes.

This isn’t theoretical. Every major Proxmox version upgrade brings forum posts from users whose

custom configurations broke during the process. Sometimes it's a service that won't start. Sometimes it's a configuration file that got overwritten. Sometimes it's a subtle incompatibility that doesn't manifest until weeks later. The common factor is almost always software that shouldn't have been on the hypervisor in the first place.

10.3. Security Surface Area

Every service running on a system represents potential attack surface. A web server might have vulnerabilities. A database might be misconfigured. Even well-maintained software occasionally has security issues discovered and exploited before patches are available.

On a typical server, a compromised service is bad but contained. The attacker gains access to that server and its data. On your Proxmox hypervisor, a compromised service potentially gives the attacker access to every VM and container running on that host. They can access virtual disk images, intercept network traffic, or simply shut down your entire infrastructure.

The principle of least privilege suggests that your most critical infrastructure components should have the smallest possible attack surface. For a hypervisor, that means running the minimum necessary to perform virtualization - and nothing else. Every additional service is another potential entry point that, if compromised, affects not just itself but everything the hypervisor manages.

10.4. Troubleshooting Becomes Archaeology

When something goes wrong on a minimal system, the debugging process is relatively straightforward. You know what's supposed to be running, you know what the normal state looks like, and deviations are easy to spot.

On a system that's accumulated services over time, troubleshooting becomes an archaeological expedition. Is this process supposed to be running? Who installed that package and why? What's this configuration file doing, and does anyone still need it? The system's behavior becomes hard to reason about because no one fully understands everything that's on it anymore.

This complexity compounds during incidents. When your VMs are inaccessible and you're trying to figure out why, the last thing you need is uncertainty about what the hypervisor is supposed to be doing. Every unfamiliar process becomes a suspect. Every unknown configuration file might be the cause. The time spent investigating red herrings is time not spent solving the actual problem.

Documentation helps, but only if it's maintained rigorously - and it usually isn't. Services get installed during a late-night troubleshooting session and never documented. Configuration changes get made "temporarily" and become permanent. Over time, the gap between documentation and reality grows until the documentation is more misleading than helpful.

10.5. The Right Architecture

The solution is conceptually simple: maintain a clear separation between your infrastructure layer and your application layer.

The infrastructure layer - Proxmox itself, the storage backends, the network configuration, the backup system - runs directly on the host because it has to. These components are what make

virtualization possible.

The application layer - web servers, databases, file services, monitoring systems, and everything else that serves your actual use cases - belongs in VMs or containers. These components use the infrastructure but shouldn't be part of it.

This separation provides multiple benefits. Your hypervisor stays clean and predictable. Your applications get their own isolated environments where they can have whatever dependencies they need without affecting each other or the host. Upgrades on either layer become independent operations. Security boundaries exist between components. Backups are cleaner because application data is contained in VM images rather than scattered across the host filesystem.

```
# Check what's installed on your Proxmox host
dpkg --get-selections | wc -l

# Compare to a fresh installation to see what's been added
# A fresh Proxmox 8 install has roughly 500-600 packages
# Significantly more suggests accumulated software
```

10.6. What Actually Belongs on the Host

Not everything beyond base Proxmox is inappropriate. Some tools genuinely belong on the hypervisor because they need direct access to hardware or kernel features that aren't available inside VMs.

Hardware monitoring tools like `lm-sensors` for temperature readings or `smartmontools` for disk health need direct hardware access. These belong on the host.

The Checkmk or other monitoring agents that report on the host's own health metrics need to run on the host to see host-level information. They belong on the host - but they should just be agents that report to a monitoring server running elsewhere, not the monitoring server itself.

Storage utilities for managing ZFS, LVM, or other storage backends need to run on the host because they're managing host-level storage. They belong on the host.

Basic network utilities for troubleshooting connectivity issues are reasonable to have available. Keep them minimal.

The test is simple: does this tool need to be on the hypervisor to function? If a VM or container could run it equally well, that's where it should be.

10.7. Migration Strategy

If your Proxmox host has already accumulated services, migrating them away is worthwhile even though it requires effort. The process is straightforward, though it takes time.

Inventory what's running on the host beyond base Proxmox. Check running processes, installed packages, listening ports, and cron jobs:

```
# What's listening on network ports?
ss -tlnp

# What services are enabled?
systemctl list-unit-files --state=enabled

# What cron jobs exist?
ls -la /etc/cron.* && crontab -l

# What packages are installed that aren't Proxmox dependencies?
apt-mark showmanual
```

For each identified service, create an appropriate new home. Lightweight services often fit well in LXC containers; more complex applications or anything requiring significant isolation belongs in a VM. Move the configuration and data, verify the service works in its new location, then remove it from the hypervisor.

Take the opportunity to document each service as you migrate it. Why does this exist? Who uses it? What does it depend on? This documentation will be valuable going forward.

10.8. LXC vs. VM: Choosing the Right Container

Proxmox offers two isolation technologies for your application workloads, and choosing appropriately helps keep your infrastructure clean.

LXC containers share the host kernel and provide lightweight isolation. They start quickly, use minimal overhead, and work well for services that don't need strong security isolation or kernel customization. A DNS server, a lightweight web proxy, or a file synchronization service are good LXC candidates.

Virtual machines provide stronger isolation with their own kernel instance. They're appropriate for workloads that need specific kernel versions, run untrusted code, or require defense-in-depth security boundaries. Anything public-facing or security-sensitive generally belongs in a VM.

When in doubt, use a VM. The resource overhead on modern hardware is minimal, and the stronger isolation is worth it for most use cases.

```
# Create a minimal LXC container for a lightweight service
pct create 400 local:vztmpl/debian-12-standard_12.2-1_amd64.tar.zst \
    --hostname internal-dns \
    --memory 256 \
    --cores 1 \
    --net0 name=eth0,bridge=vmbr0,ip=dhcp \
    --unprivileged 1
```

10.9. Best Practices

Treat your Proxmox host as infrastructure, not as a general-purpose server. Its job is to run virtual machines and containers reliably. Everything else is a distraction that adds risk.

Before installing anything on the host, ask whether it could run in a VM or container instead. If the answer is yes, put it there. The few minutes spent creating a container pays dividends in system cleanliness and upgrade safety.

Document exceptions rigorously. If something genuinely needs to run on the host, document what it is, why it's there, and who's responsible for it. Review this documentation periodically and remove anything that's no longer needed.

Keep the host updated independently from your application workloads. When Proxmox security updates are released, you should be able to apply them without worrying about breaking unrelated services.

Plan for hypervisor replacement. If your Proxmox host died today and you had to rebuild it from scratch, how long would that take? A clean host with minimal customization can be rebuilt in an hour or two. A host with accumulated services and undocumented configurations might take days - if you can reconstruct it at all.

10.10. The Bottom Line

Your Proxmox hypervisor is the foundation that everything else depends on. Foundations should be stable, predictable, and well-understood. Every service you add directly to the host undermines these qualities, trading short-term convenience for long-term risk.

Keep it clean. Keep it minimal. Let Proxmox do what it's designed to do - run your VMs and containers reliably - and put everything else where it belongs: inside those VMs and containers.

Conclusion

If you've made it through all ten chapters, you now have a solid understanding of where Proxmox deployments commonly go wrong - and more importantly, how to avoid those pitfalls in your own environment.

Let's step back and look at the bigger picture.

The Common Thread

Reading through these mistakes, you might notice a pattern. Very few of them are about Proxmox being broken or badly designed. Instead, they're about mismatched expectations, outdated assumptions, and the gap between "technically possible" and "actually advisable."

ZFS works with limited RAM - it just doesn't work well. RAIDZ provides redundancy - just not the kind of performance VMs need. Docker runs on the Proxmox host - until it conflicts with something important. HA can be enabled on two nodes - it just won't actually provide high availability.

The recurring theme is that Proxmox gives you tremendous flexibility, and with that flexibility comes the responsibility to understand the implications of your choices. The tool won't stop you from doing things that seem reasonable but lead to problems. That's not a flaw; it's the nature of powerful, flexible software.

Principles Over Checklists

Rather than memorizing specific rules, internalize the principles behind them:

Understand your resource requirements. Whether it's RAM for ZFS, nodes for quorum, or IOPS for VM storage, know what your chosen technology actually needs - not what forum posts vaguely suggest.

Separate concerns cleanly. Hypervisor versus applications. VM storage versus backup storage. Infrastructure layer versus application layer. Clear boundaries make systems easier to understand, maintain, and troubleshoot.

Plan for failure. Backups that aren't tested aren't backups. HA that can't achieve quorum isn't highly available. Monitoring that doesn't alert you isn't monitoring. Think through what happens when things go wrong, not just when they go right.

Keep it simple until you need complexity. Local-LVM might be enough. A single node might be enough. Basic vzdump backups might be enough. Add complexity when you have concrete requirements that demand it, not because it seems more professional.

Maintain visibility. You can't manage what you can't see. Monitoring, logging, regular health checks - these aren't optional extras for serious environments. They're how you know whether your infrastructure is healthy before users start complaining.

Your Environment Is Unique

This guide provides general recommendations, but your specific situation might call for different choices. A homelab running non-critical workloads has different requirements than a production cluster serving paying customers. A single-node setup doesn't need to worry about migration compatibility. An environment with excellent backups can tolerate more risk in other areas.

Use the principles from this guide to evaluate your own requirements and constraints. The goal isn't to implement every recommendation blindly - it's to make informed decisions that match your actual needs.

Continuous Improvement

A well-running Proxmox environment isn't a destination; it's an ongoing practice. Technology evolves. Your workloads change. New features become available while old assumptions become outdated.

Schedule periodic reviews of your infrastructure. Are your backups still covering everything important? Has your storage grown beyond what your original RAM allocation supports? Are there new Proxmox features that would improve your setup? Is your monitoring still catching what matters?

The administrators who avoid disasters aren't the ones who set things up perfectly once - they're the ones who pay attention continuously and address small issues before they become big ones.

Final Thoughts

Proxmox VE is genuinely excellent software that powers countless reliable environments around the world. When things go wrong, the cause is usually human decisions rather than software defects. That's actually good news: it means you have control over your outcomes.

Build your environment thoughtfully. Monitor it diligently. Back it up religiously. Keep learning as the technology evolves.

And the next time you're tempted to take a shortcut - to skip the backup test, to install just one more service on the host, to enable HA on a two-node cluster because surely it'll be fine - remember that future you is the one who'll deal with the consequences.

Be kind to future you.

Good luck with your Proxmox deployments.