Practical Assignment 1: Foundational Server Administration

Gestió de Sistemes i Xarxes

Deadline: 20 March 2026

Abstract

In this assignment, you will work in pairs to design and implement a foundational Linux server infrastructure for a small software startup. Rather than following a recipe of step-by-step instructions, you will face realistic challenges that require reasoning, problem-solving, and collaboration. Through six progressive weeks, you will build a working, observable, and documented system that mirrors real-world sysadmin responsibilities.

This is not about memorizing commands—it's about **understanding why systems are designed the way they are** and **learning to troubleshoot independently**.

Overview

Duration

6 weeks (Weeks 1–6)

- **Start Date:** 10 February 2026
- **Deadline:** 20 March 2026
- **Oral Interview:** Week 7 (23–27 March 2026)

Format

- **Group work:** Pairs of two students
- **Weekly lab sessions:** 2 hours per week with professor
- **Autonomous work:** Most development happens outside lab sessions
- **Tool:** Version control (Git) to document all progress. Private GitHub repository. VirtualBox VMs for the server environment.

Learning Outcomes

By completing this assignment, you will demonstrate the ability to:

1. Design and implement foundational server infrastructure with automation best practices
2. Install, configure, and troubleshoot Linux systems (Debian-based)
3. Create reliable, observable services using `systemd`
4. Manage processes, signals, and system resources
5. Administer users, groups, and access control with least-privilege principles
6. Design and implement storage, backup, and recovery strategies
7. Document infrastructure decisions and create operational runbooks
8. Collaborate effectively on infrastructure projects

1 The Scenario: "GreenDevCorp Startup Infrastructure"

Imagine you are junior system administrators hired by **GreenDevCorp**, a small software startup. The company is transitioning from ad-hoc cloud usage to managing its own internal infrastructure. Here's the situation:

The Team

- 4 developers who need to collaborate on shared code and data
- 1 operations engineer (you'll take this role)
- Growing from 4 to 10 people over the next 6 months

Current State

- A single Debian Linux server (minimally configured)
- Sensitive project data that must not be lost
- Legacy and new applications that need to coexist
- No formal infrastructure documentation or disaster recovery plan
- Ad-hoc, manual administration (no automation)

Your Mission

Design and implement the foundational infrastructure to support the startup's operations. You'll tackle realistic challenges week by week, documenting your decisions and learning what it takes to build reliable systems.

Success Criteria

At the end of 6 weeks:

- The system is **usable, reliable, and observable**
- All infrastructure is tracked in version control (Git)
- Configuration is repeatable and idempotent (running scripts multiple times produces the same result)
- All decisions are documented with clear rationale
- The system can be handed off to another sysadmin using your documentation
- Both team members can explain and defend every design choice

2 Weekly Challenges

Each week presents a new challenge that reflects real operational needs. Your job is to **discover solutions**, document your reasoning, and demonstrate your system to the professor.

Important: You Will Be Asked "Why?"

Throughout this assignment, professors will ask questions like:

- "Why did you choose this approach?"
- "What trade-offs did you make?"
- "How would you handle this if the requirements changed?"
- "What happens if this fails?"

These questions are not meant to trick you—they help you develop the **reasoning skills** that distinguish a good sysadmin from someone who just follows instructions. Be prepared to explain your thinking clearly.

Week 0: "Getting Started: Setting Up Your Environment"

Objective

Both team members must arrive at Week 1 with a working Debian virtual machine they can access locally. The goal is only environment setup—no configuration beyond installation.

Do This at Home Before the First Session

1. Download and install VirtualBox
   - `https://www.virtualbox.org/`
   - Optional: Install the VirtualBox Extension Pack (matching version)
2. Download Debian ISO image
   - `https://www.debian.org/distrib/`
   - Use amd64 netinst or full ISO
   - Optional but recommended: Verify checksum (SHA256) from Debian downloads page
3. Create a new VM in VirtualBox (Unattended Install)
   (a) Name: `debian-gsx`
   (b) Choose a folder for VM data and select the Debian ISO
   (c) Select unattended installation
   (d) Set user: `gsx` and a password you both know (store securely)
   (e) Hostname: `debian-gsx`
   (f) Domain: `gsx.virtualbox.org`
   (g) Install in background (no GUI)
   (h) Enable Guest Additions installation
   (i) Hardware: 2 GB RAM, 1 CPU
   (j) Disk: 20 GB, dynamically allocated (default)
   (k) Start the VM and wait for installation to finish
4. Create a snapshot of the clean VM
   - Name it `clean-install` and add a short description (fresh Debian, user `gsx`)
   - This snapshot lets you roll back quickly if needed

Quick Verification

- Log in on the VM console with user `gsx`
- Confirm network connectivity: `ping -c 1 debian.org`
- Update package index: `sudo apt update` (if sudo is not present, you will add it in Week 1)
- Check Guest Additions: window resizing works; shared clipboard optional

Recommendations

- Host machine: 8 GB RAM or more recommended
- Networking: Default NAT is fine
- Keep the ISO and note the exact Debian version used
- If unattended fails, perform a normal minimal install (no desktop environment)
- You will not use the GUI; after Week 1, all work is via SSH or terminal

Deliverable (Start of Week 1)

- Each student has a working `debian-gsx` VM with a clean snapshot and can log in locally

Week 1: "The Server Exists, But Nothing Works"

Foundation & Remote Access

The Challenge

The startup has a Debian Linux server VM. It's minimally configured, and you can only access it via a physical console (VM GUI). You need to:

- Enable **remote access** so both team members can work from different workstations
- Set up a way to **escalate privileges safely** for administrative tasks
- Organize administration so both of you can work together
- **Track configuration changes** so you can redo them if the system fails
- **Automate repetitive setup tasks** to save time and avoid errors

What You Need to Figure Out

Before Week 1 lab session, research and plan:

1. **Remote Access:** How do Linux systems enable remote administration? What tools are available? What are the security implications?
2. **Privilege Escalation:** The root user has all power, but you shouldn't log in as root directly. How do modern Linux systems handle this?
3. **Version Control:** How can you use Git to track system configuration? What should be in version control, and what should not?
4. **Automation:** Write shell scripts that set up your administrative environment. How do you make scripts safe and repeatable?
5. **Documentation:** What decisions did you make, and why? What would another sysadmin need to know to maintain your system?

Required Deliverables (End of Week 1)

1. **VM Setup:** Both team members have a working Debian VM (local), equally configured (automatically via scripts)
2. **Functional SSH access:** Both team members can remotely access (their copy of) the server (from the host machine)
3. **Git repository:** Private repository (e.g., GitHub) with initial configuration and scripts
4. **Admin directory:** A shared directory structure for administrative files and scripts
5. **Setup scripts:**
   - A script that installs basic packages and tools
   - A script that sets up the administrative directory structure
   - A script that verifies all setup is correct (and re-applies it if needed—idempotence)
   - A script that packages sensitive data for backup and keeps file/dir attributes (e.g., tar with encryption)
6. **Documentation:**
   - README explaining how to access the system
   - Design notes: Why did you choose SSH over other options? Why this directory structure?
   - Reflection: What did you learn about infrastructure automation?

Concepts to Explore

- SSH (secure shell), authentication, key-based login
- Sudo and privilege escalation
- Shell scripting best practices (error handling, safety, idempotence)
- Version control workflows for infrastructure
- .gitignore and what not to commit (passwords, private keys, etc.)

Hints and Questions to Guide Your Thinking

Answer these questions within your documentation:

- What are the security implications of using passwords vs. keys for SSH?
- If you have to reinstall the system, can your scripts restore the entire configuration? If not, what's missing?
- How would you prevent both team members from accidentally running the same setup script at the same time?
- What information should be in Git, and what should only live on the server (why)?

Week 2: "We Need Services, But They Keep Breaking"

Services, Observability & Automation

The Challenge

The startup needs to run Nginx (web server) and other services. However:

- Services are manually started/stopped (fragile and error-prone)
- When something breaks, there's no visibility—"It just stopped working"
- Backups are done manually (frequently forgotten)

You need to implement **automated, observable service management**:

- Services start automatically and recover from failures
- When something goes wrong, there's a clear audit trail
- Backups run automatically and can be verified

What You Need to Figure Out

1. **Service Management:** How do modern Linux systems ensure services run reliably? What happens if a service crashes?
2. **Observability:** When a service fails, how do you diagnose what went wrong? Where are logs stored, and how do you query them?
3. **Automatic Tasks:** How do you run backup and maintenance tasks automatically without manual intervention?
4. **Custom Services:** Can you create your own services that integrate with the system's service management?
5. **Log Management:** How do you prevent logs from consuming all disk space?

Required Deliverables (End of Week 2)

1. **Nginx Service:**
   - Nginx runs reliably using `systemd`
   - Service starts automatically on system boot
   - Nginx automatically restarts if it crashes
   - Nginx is discoverable with standard tools (`systemctl`, `journalctl`)
2. **Backup Service:**
   - A `systemd` service that runs your Week 1 backup script
   - A `systemd` timer that triggers the backup service (e.g., daily)
   - Verification that backups are running automatically
3. **Logging and Observability:**
   - All service activity is logged to `journald`
   - Scripts to query and display service logs
   - Evidence that you can diagnose service failures from logs
4. **Updated Git Repository:**
   - Service and timer unit files
   - Updated setup scripts
   - Documentation explaining the service architecture

Concepts to Explore

- `systemd` services (`.service` files)
- `systemd` timers (`.timer` files)
- `journald` and structured logging
- Log rotation (logrotate, journal retention)
- Signals and process lifecycle
- Service dependencies and ordering

Hints and Questions to Guide Your Thinking

Answer these questions as part of your documentation:

- What should happen if Nginx crashes at 3 AM? Who finds out, and how?
- How do you test that a service will actually restart automatically? (You can't just wait for it to fail!)
- If backups are failing silently, how would you know? What metrics matter?
- How would you explain a service failure to the startup's team using only the logs?

Week 3: "The Server is Slow, and We Don't Know Why"

Process Management & Resource Control

The Challenge

One of the developers reports that the server feels sluggish. Some jobs are taking much longer than expected. You need to:

- Identify which processes are consuming resources
- Understand process lifecycle and control processes effectively
- Prevent one job or user from monopolizing the system
- Ensure the system remains responsive even under heavy load

What You Need to Figure Out

1. **Process Inspection:** How do you discover what's running on a system? Where do you look, and what tools are available?
2. **Resource Usage:** How do you identify which processes consume CPU, memory, or I/O? What's normal, and what's concerning?
3. **Process Control:** How do you influence process behavior (pause, resume, kill, change priority) without rebooting?
4. **Resource Limits:** How do you prevent one runaway process from taking down the entire system?
5. **Service Resilience:** How can your service (from Week 2) signal gracefully and handle resource constraints?

Required Deliverables (End of Week 3)

1. **Process Diagnostics Scripts:**
   - Script to list top resource consumers (CPU, memory)
   - Script to show process tree and relationships
   - Script to extract and display specific process metrics
2. **Process Control Demonstration:**
   - Create a "workload script" that runs background processes (check the '`yes`' command)
   - Demonstrate signal handling (`SIGINT`, `SIGUSR1`, `SIGUSR2`)
   - Show how to gracefully stop processes vs. force-kill
3. **Service with Resource Limits:**
   - Your backup, Nginx, or workload (ran as service) service configured with CPU/memory limits
   - Evidence that limits are enforced (using cgroups)
   - Monitoring that shows resource usage with limits applied
4. **Updated Documentation:**
   - Process concepts explained
   - Troubleshooting guide: "The server feels slow. What do I check?"
   - Examples of signal handling and graceful shutdown
   - Demonstrations of process control and resource limiting

Concepts to Explore

- Process inspection (`/proc` , `ps` , `top` , `htop`)
- Process lifecycle (fork, exec, wait, exit)
- Signals and signal handling (`SIGINT` , `SIGTERM` , `SIGKILL` , `SIGUSR1` , `SIGUSR2`)
- Process priority (`nice` , `renice`)
- cgroups (control groups) and resource limiting
- Background jobs and process groups

Hints and Questions to Guide Your Thinking

Answer these questions as part of your documentation:

- How is killing a process with `SIGTERM` different from `SIGKILL` ? When would you use each?
- If your service receives a signal, how should it respond? Should it save state before exiting?
- How do you verify that a resource limit is actually working? (Can you create a test that would fail without the limit?)
- If a developer's job is using 90% CPU, is that a problem? How do you decide?

Week 4: "Team Collaboration"

Users, Groups & Access Control

The Challenge

Four developers now need to work on the same server. They must:

- Have their own secure, private directories
- Access shared team resources and code
- Not interfere with each other's work
- Not accidentally or intentionally access sensitive data they shouldn't

You need to design a secure, collaborative environment based on **least privilege**: each user has exactly the permissions they need, and no more.

What You Need to Figure Out

1. **User & Group Structure:** How do you organize users and groups to reflect the team's roles? What's the difference between Unix groups and organizational teams?
2. **File Permissions:** How do you allow shared work without exposing private data? What are the limits of traditional Unix permissions?
3. **Advanced Access Control:** Beyond basic permissions, what tools exist to apply fine-grained access policies?
4. **Resource Limits per User:** How do you prevent one user from exhausting system resources (memory, processes)?
5. **Environment Personalization:** How do you give each user a customized shell environment without manual setup?
6. **Security Reasoning:** How do you verify that your setup actually enforces least privilege? What could go wrong?

Required Deliverables (End of Week 4)

1. **User and Group Structure:**
   - Create group '`greendevcorp`' representing the development team
   - Create users '`dev1`', '`dev2`', '`dev3`', '`dev4`' with appropriate group membership
   - Each user has a private home directory
2. **Directory Structure with Appropriate Permissions:**
   - '`/home/greendevcorp/bin`': shared scripts, executable by team members only
   - '`/home/greendevcorp/shared`': shared work directory with setgid and sticky bit
   - '`/home/greendevcorp/done.log`': team activity log, readable by all, writable by authorized user only. It contains a record of completed tasks. All can read the list of tasks, but only `dev1` can add new entries.
   - All permissions reflect least-privilege principles
3. **Resource Limits:**
   - Per-user CPU and memory limits configured via PAM
   - Max number of processes per user enforced
   - Max open files per user enforced

- Verification that limits work (with test scripts)

4. **Environment Personalization:**
   - Shared shell configuration in '`/etc/profile.d/`' for team members
   - Common aliases and PATH customization
   - Test that new team member login inherits correct environment
5. **Security Verification Script:**
   - Script that tests access control: users can access what they should, cannot access what they shouldn't
   - Script that verifies resource limits
   - Evidence of testing and validation
6. **Updated Documentation:**
   - User/group design rationale
   - Permission model explained
   - Troubleshooting: "A user can't access a shared file. How do I debug?"
   - Onboarding guide for new team members

Concepts to Explore

- User and group administration (`useradd`, `groupadd`, `usermod`)
- Unix file permissions (rwx for user/group/others)
- Special permission bits (setuid, setgid, sticky bit)
- POSIX Access Control Lists (ACLs)
- PAM (Pluggable Authentication Modules) and `limits.conf`
- Shell configuration files (`.bashrc`, `.profile`, `/etc/profile.d/`)
- Sudoers and privilege escalation policies

Hints and Questions to Guide Your Thinking

Answer these questions as part of your documentation:

- If a file is in a shared directory and owned by '`dev1`', but needs to be readable by all team members, what permissions would you set? Why?
- What's the difference between setgid on a directory vs. a file? Why would you use each?
- If you misconfigure permissions and a user can't access a file they need, how do you troubleshoot? (What tools would you use?)
- How would you verify that your permission model actually enforces the security policy you intended?

Week 5: "Data Lives Forever (Or It Should)"

Storage, Backup & Recovery

The Challenge

The startup's data is growing, and management is (rightfully) concerned about disaster recovery. You need to:

- Add new storage to the system as data grows
- Design a backup strategy that survives common failure scenarios
- Verify that backups actually work (before you need them)
- Enable the team to access shared storage across the network

This is perhaps the most critical responsibility of a sysadmin: **ensuring that data is not lost**.

What You Need to Figure Out

1. **Storage Management:** How do you add new disks to a system without disrupting running services? What tools and filesystems are available?
2. **Backup Strategy:** What's the 3-2-1 principle? How do full, incremental, and differential backups differ? What trade-offs do they involve?
3. **Automation:** How do you schedule and automate backups so they happen reliably?
4. **Verification:** How do you know if a backup is good? Can you actually recover from it?
5. **Networked Storage:** How do you share storage across systems (e.g., Nginx on one machine, backups on another)?
6. **Disaster Planning:** What's your recovery-time objective (RTO) and recovery-point objective (RPO)? How does this influence your strategy?

Required Deliverables (End of Week 5)

1. **Storage Setup:**
   - New disk added and partitioned (create a new virtual disk in VirtualBox)
   - Filesystem created and mounted
   - Persistent mount configuration in '`/etc/fstab`'
   - Documentation of storage layout and capacity planning
2. **Backup Strategy Document:**
   - Explained: What data needs to be backed up? What's the retention policy?
   - Defined: Full, incremental, or differential strategy? How often?
   - Justified: Why does this strategy meet the startup's needs?
   - 3-2-1 principle applied: Where are backups stored? How many copies?
3. **Automated Backup Implementation:**
   - Backup scripts (using `tar`, `rsync`, or equivalent)
   - Scheduled backups (systemd timer or cron)
   - Logging that shows each backup succeeded
4. **Backup Verification:**
   - Test restore procedures (restore to alternate location, verify all files present)

- Document what could fail and how you'd detect it
- Automated tests that verify backup integrity

5. **Networked Storage (Optional but Recommended):**
   - NFS or SMB configuration for shared team storage
   - Use snapshot to create a second VM acting as NFS client
   - Secure mount configuration
   - Tested access from development VMs
6. **Updated Documentation:**
   - Storage architecture and capacity planning
   - Backup procedures and schedules
   - Recovery procedures (step-by-step)
   - Disaster recovery runbook

Concepts to Explore

- Disk partitioning (`fdisk` , `parted`)
- Filesystems (ext4, XFS, journaling concepts)
- LVM (Logical Volume Management)
- Backup tools (`tar` , `rsync` , `dump`)
- Backup strategies (full, incremental, differential)
- 3-2-1 backup principle
- NFS (Network File System)
- `fstab` and persistent mounts
- RTO and RPO concepts

Hints and Questions to Guide Your Thinking

Answer these questions as part of your documentation:

- If you back up every file every night, you'll have massive backups. How could you reduce storage overhead? What's the trade-off?
- If you lose the main server, can you restore from backup? Have you actually tested this?
- How long would recovery take? Is that acceptable for the startup?
- If one backup location is corrupted, do you have another copy? (This is why 3-2-1 matters.)
- How would you handle a database that's currently being written to? (You can't just copy a live database file.)

Week 6: "Documentation & Handoff"

Operations & Reflection

The Challenge

The startup hires a new operations engineer. Your job is to document everything so they can maintain the system effectively. Additionally, you'll reflect on what you've learned and prepare for the final oral interview.

What You Need to Deliver

All documentation and code must be in the Git repository, that is your main deliverable for this assignment. You need to provide:

1. **Operations Runbook:**
   - Common tasks: "How do I ...?"
     - Add a new developer to the team?
     - Handle a team member leaving?
     - Check if services are running correctly?
     - Diagnose a slow system?
     - Restore from backup?
   - Troubleshooting guide: Problems you might face and how to solve them
   - Escalation procedures: When to call for help and who to contact
2. **System Architecture Documentation:**
   - Diagram (text or image) showing system components
   - Design rationale: Why did you make these choices?
   - Trade-offs: What would you do differently? What would you keep the same?
   - Future planning: How would the system scale to 20 people? 100 people?
3. **Verified Backup and Recovery:**
   - Document of full recovery procedure
   - Evidence that recovery has been tested
   - Timeline: How long does recovery take?
4. **Clean Git Repository:**
   - Commit history that tells a story (good commit messages)
   - Clear structure: scripts, configs, documentation, weekly deliverables
   - README explaining how to use the repository
5. **Reflection Essay (Each Student):**
   - What was the most challenging aspect of this project?
   - What would you do differently if you started over?
   - How has your understanding of system administration changed?
   - What's one thing you'd want to learn more about?

Grading and Assessment

Your assignment will be evaluated on three dimensions:

1. Process (40%): Decision-Making and Infrastructure-as-Code

How well did you apply automation, version control, and best practices?

- **Automation:** Scripts are idempotent, safe, and well-documented
- **Version Control:** Git history is clear; commits have meaningful messages
- **Configuration:** Setup is repeatable; clear separation of concerns
- **Documentation:** Design decisions are explained; rationale is clear
- **Collaboration:** Both team members contributed meaningfully; evidence of discussion

2. Functionality (30%): System Works Reliably

Does the system actually work as designed?

- **Access:** Remote access works; escalation is secure
- **Services:** Nginx and backup service run reliably; automatic restart works
- **Observability:** Logs are available; diagnostics tools work
- **Resource Management:** Limits are enforced; processes are manageable
- **Access Control:** Users have correct permissions; least privilege enforced
- **Backup:** Backups run automatically; recovery has been tested

3. Communication and Reasoning (30%): Explaining Your System

Can you explain and defend your design choices?

- **Reasoning:** You can explain why each component was designed the way it was
- **Trade-offs:** You acknowledge alternatives and explain your choices
- **Troubleshooting:** You can diagnose issues using your knowledge
- **Reflection:** You've thought deeply about what you've learned

Oral Interview (Week 7)

Each group will have a 20–30 minute interview with the professor to verify authorship and understanding. You'll be asked to:

- Demonstrate your system (remote access, services running, logs visible)
- Explain your architectural choices and trade-offs
- Troubleshoot a problem scenario presented by the professor
- Discuss what you learned and how you'd improve the system

Both team members should be prepared to explain the entire system. It's not sufficient for one person to know everything.

Guidelines and Expectations

Collaboration

- Both team members must contribute meaningfully to every week
- Rotate roles (one person leads research, the other leads implementation, etc., and switch weekly)
- Discuss architectural decisions together before implementing
- Use Git to coordinate work; regular commits show progress

Problem-Solving Approach

- **Start with questions, not commands.** Ask yourself: "What do I need to accomplish? What tools might help? How do they work?"
- **Research before implementing.** Read documentation, experiment in a test environment, understand the trade-offs
- **Document as you go.** Don't wait until the end; write notes about decisions while they're fresh
- **Test thoroughly.** Verify that systems work, recover from failure, and respond to user actions as expected
- **Iterate and improve.** First solution rarely perfect; refine based on what you learn

Version Control Best Practices

- Commit frequently with clear, descriptive messages
- Example good message: "Add systemd timer for daily backups; configure journald retention policy"
- Example poor message: "Fixed stuff"
- Use a `.gitignore` to exclude passwords, private keys, and other secrets
- Don't commit large files (backups, logs, VMs) to Git

Safety and Security

- Test in a different VM or sandboxed environment first
- Back up your work regularly (ironic, but important!)
- Don't store passwords or private keys in Git
- Use SSH keys instead of passwords where possible
- Follow the principle of least privilege everywhere

Getting Help

- Lab sessions are opportunities to ask questions and get guidance
- Review documentation and man pages first; `man systemd.service`, `man sudoers`, etc.
- Collaborate with other groups (but do your own work!)
- Bring specific problems to the professor, not just "It doesn't work"

Deliverables Checklist

At the end of each week, verify that you have:

Week	Deliverables
1	SSH access, Git repo, setup scripts, documentation
2	Nginx service, backup service with timer, observability scripts, docs
3	Process diagnostics, workload scripts, resource limits, docs
4	Users/groups/permissions, resource limits per user, environment setup, docs
5	New storage, backup strategy, automated backups, recovery tests, docs
6	Operations runbook, architecture docs, clean Git, reflection

Key Resources

Recommended Reading

- *UNIX and Linux System Administration Handbook*, 5th Edition (Chapters on systemd, user management, storage, etc.)
- *The Practice of System and Network Administration*, 3rd Edition (Chapters on runbooks, documentation, operations)
- Official Debian documentation: `https://www.debian.org/doc/`
- systemd documentation: `https://systemd.io/`

Key Tools (You'll Learn These Deeply)

- **Shell scripting:** bash, sh (error handling, safety)
- **Version control:** Git, GitHub
- **Service management:** `systemd`, `journalctl`
- **User administration:** `useradd`, `groupadd`, `usermod`, `sudo`
- **Storage:** `fdisk`, `mkfs`, `mount`, `fstab`, `LVM`, `rsync`, `tar`
- **Process management:** `ps`, `top`, `kill`, `nice`, `renice`
- **Network access:** `SSH`, `scp`, `rsync`

Final Thoughts

This assignment mirrors real system administration work. You'll face ambiguous problems, discover solutions through research and experimentation, document your decisions, and collaborate with teammates. This is what the job actually entails.

The most important skill you'll develop is not memorizing commands—it's **learning to think like an operations engineer**: "What could go wrong? How do I monitor for it? How do I recover if it does?"

Good luck, and remember: the best sysadmins are never done learning.